O'REILLY®

Второе издание

Веб-разработка с применением Node и Express

Полноценное использование стека JavaScript

Итан Браун

SECOND EDITION

Web Development with Node and Express

Leveraging the JavaScript Stack

Ethan Brown

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Веб-разработка с применением Node и Express

Полноценное использование стека JavaScript

Второе издание

Итан Браун

ПИТЕР®

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

2021

ББК 32.988.02-018
УДК 004.738.5
Б87

Браун И.

Б87 Веб-разработка с применением Node и Express. Полноценное использование стека JavaScript. 2-е издание. — СПб.: Питер, 2021. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0590-8

Создавайте динамические веб-приложения с применением Express — ключевого компонента из стека разработки Node/JavaScript. Итан Браун описывает работу с Express 5 на примере создания полноценного приложения. В книге рассматриваются все этапы и компоненты — от серверного рендеринга до разработки API для работы с одностраничными приложениями (SPA).

Express является золотой серединой между устоявшимся фреймворком и отсутствием фреймворка вообще, поэтому он оставляет вам определенную свободу при архитектурном выборе. Эта книга предоставит лучшие решения для фронтенд- и бэкенд-разработчиков, использующих Express. Научитесь смотреть на веб-разработку под новым углом!

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1492053514 англ.

Authorized Russian translation of the English edition of Web Development with Node and Express: Leveraging the JavaScript Stack 2nd Edition
ISBN 9781492053514

ISBN 978-5-4461-0590-8

Краткое содержание

Введение ... 16
Благодарности ... 21
Об авторе ... 23
Глава 1. Знакомство с Express ... 24
Глава 2. Первые шаги с Node ... 33
Глава 3. Экономия времени благодаря Express ... 44
Глава 4. Наводим порядок ... 55
Глава 5. Обеспечение качества ... 65
Глава 6. Объекты запроса и ответа ... 84
Глава 7. Шаблонизация с помощью Handlebars ... 98
Глава 8. Обработка форм ... 115
Глава 9. Cookie-файлы и сеансы ... 128
Глава 10. Промежуточное ПО ... 139
Глава 11. Отправка электронной почты ... 148
Глава 12. Промышленная эксплуатация ... 160
Глава 13. Персистентность данных ... 174
Глава 14. Маршрутизация ... 202
Глава 15. REST API и JSON ... 214
Глава 16. Одностраничные приложения ... 223
Глава 17. Статический контент ... 244
Глава 18. Безопасность ... 252
Глава 19. Интеграция со сторонними API ... 280
Глава 20. Отладка ... 296
Глава 21. Ввод в эксплуатацию ... 307
Глава 22. Поддержка ... 319
Глава 23. Дополнительные ресурсы ... 329
Об иллюстрации на обложке ... 335

Оглавление

Введение ... 16
Для кого эта книга ... 16
Примечание ко второму изданию ... 17
Структура книги ... 17
Учебный сайт ... 18
Условные обозначения ... 19
Использование примеров исходного кода ... 20
От издательства ... 20

Благодарности ... 21

Об авторе ... 23

Глава 1. Знакомство с Express ... 24
Революция JavaScript ... 24
Знакомство с Express ... 26
Приложения на стороне сервера и на стороне клиента ... 27
Краткая история Express ... 28
Node: новая разновидность веб-сервера ... 29
Экосистема Node ... 30
Лицензирование ... 31
Резюме ... 32

Глава 2. Первые шаги с Node ... 33
Установка Node ... 33
Использование терминала ... 34

Редакторы ... 35
npm ... 36
Простой веб-сервер с помощью Node ... 37
Hello World! ... 38
Событийно-ориентированное программирование ... 39
Маршрутизация ... 39
Раздача статических ресурсов ... 40
Вперед к Express ... 43

Глава 3. Экономия времени благодаря Express ... 44
Скаффолдинг ... 44
Сайт Meadowlark Travel ... 45
Первые шаги ... 45
Представления и макеты ... 49
Статические файлы и представления ... 52
Динамический контент в представлениях ... 53
Резюме ... 54

Глава 4. Наводим порядок ... 55
Структура файлов и каталогов ... 55
Лучшие решения ... 56
Контроль версий ... 56
Как использовать Git с этой книгой ... 57
Если вы набираете примеры самостоятельно ... 57
Если вы используете официальный репозиторий ... 59
Пакеты npm ... 60
Метаданные проекта ... 62
Модули Node ... 62
Резюме ... 64

Глава 5. Обеспечение качества ... 65
План по обеспечению качества ... 66
QA: стоит ли оно того? ... 67
Логика и визуализация ... 68
Виды тестов ... 69
Обзор методов QA ... 69

Установка и настройка Jest ... 70
Модульное тестирование ... 71
Mock-объекты ... 71
Рефакторинг приложения для упрощения его тестирования ... 71
Наш первый тест ... 72
Поддержка тестов ... 74
Покрытие кода тестами ... 74
Интеграционное тестирование ... 76
Линтинг ... 78
Непрерывная интеграция ... 82
Резюме ... 83

Глава 6. Объекты запроса и ответа ... 84
Составные части URL ... 84
Методы запросов HTTP ... 86
Заголовки запроса ... 86
Заголовки ответа ... 87
Типы данных Интернета ... 87
Тело запроса ... 88
Объект запроса ... 88
Объект ответа ... 89
Получение более подробной информации ... 91
Разбиваем на части ... 92
Рендеринг контента ... 92
Обработка форм ... 94
Предоставление API ... 95
Резюме ... 97

Глава 7. Шаблонизация с помощью Handlebars ... 98
Нет абсолютных правил, кроме этого ... 100
Выбор шаблонизатора ... 100
Pug: другой подход ... 101
Основы Handlebars ... 102
Комментарии ... 103
Блоки ... 104

Серверные шаблоны ... 105
Представления и макеты ... 106
Использование (или неиспользование) макетов в Express ... 109
Секции ... 109
Частичные шаблоны ... 111
Совершенствование шаблонов ... 113
Резюме ... 114

Глава 8. Обработка форм ... 115
Отправка данных с клиентской стороны на сервер ... 115
HTML-формы ... 116
Кодирование ... 117
Различные подходы к обработке форм ... 117
Обработка форм посредством Express ... 119
Отправка данных формы с помощью fetch ... 121
Загрузка файлов на сервер ... 123
Загрузка файлов посредством fetch ... 125
Усовершенствование интерфейса пользователя для загрузки файлов ... 126
Резюме ... 127

Глава 9. Cookie-файлы и сеансы ... 128
Внешнее хранение данных доступа ... 130
Cookie-файлы в Express ... 131
Просмотр cookie-файлов ... 132
Сеансы ... 132
Хранилища в памяти ... 133
Использование сеансов ... 134
Использование сеансов для реализации уведомлений ... 135
Для чего использовать сеансы ... 137
Резюме ... 138

Глава 10. Промежуточное ПО ... 139
Принципы работы промежуточного ПО ... 140
Примеры промежуточного ПО ... 141
Распространенное промежуточное ПО ... 144
Промежуточное ПО сторонних производителей ... 146
Резюме ... 147

Глава 11. Отправка электронной почты 148
SMTP, MSA и MTA 148
Получение сообщений электронной почты 149
Заголовки сообщений электронной почты 149
Форматы сообщений электронной почты 150
Сообщения электронной почты в формате HTML 150
Nodemailer 151
Отправка писем 152
Отправка писем нескольким адресатам 153
Рекомендуемые варианты для массовых рассылок 154
Отправка писем в формате HTML 154
Изображения в письмах в формате HTML 155
Использование представлений для отправки писем в формате HTML 156
Инкапсуляция функциональности электронной почты 158
Резюме 159

Глава 12. Промышленная эксплуатация 160
Среды выполнения 160
Отдельные конфигурации для различных сред 161
Запуск процесса в Node 163
Масштабируем сайт 164
Горизонтальное масштабирование с помощью кластеров приложений 165
Обработка неперехваченных исключений 167
Горизонтальное масштабирование с несколькими серверами 170
Мониторинг сайта 170
Сторонние мониторы работоспособности 171
Стресс-тестирование 171
Резюме 173

Глава 13. Персистентность данных 174
Хранение данных в файловой системе 174
Хранение данных в облаке 176
Хранение данных в базе данных 177
Замечания относительно производительности 178
Абстрагирование слоя базы данных 178
Установка и настройка MongoDB 180

Mongoose ... 181
Подключение к базе данных с помощью Mongoose ... 182
Создание схем и моделей ... 183
Определение начальных данных ... 184
Извлечение данных ... 186
Добавление данных ... 188
PostgreSQL ... 190
Добавление данных ... 197
Использование баз данных для хранения сеансов ... 198
Резюме ... 201

Глава 14. Маршрутизация ... 202
Маршруты и SEO ... 204
Поддомены ... 204
Обработчики маршрутов — промежуточное ПО ... 206
Пути маршрутов и регулярные выражения ... 207
Параметры маршрутов ... 208
Организация маршрутов ... 209
Объявление маршрутов в модуле ... 210
Логическая группировка обработчиков ... 211
Автоматический рендеринг представлений ... 212
Резюме ... 213

Глава 15. REST API и JSON ... 214
JSON и XML ... 215
Наш API ... 215
Выдача отчета об ошибках API ... 217
Совместное использование ресурсов между разными источниками ... 218
Наши тесты ... 218
Использование Express для предоставления API ... 221
Резюме ... 222

Глава 16. Одностраничные приложения ... 223
Краткая история разработки веб-приложений ... 223
Технологии SPA ... 226
Создание приложения React ... 227

Основы React 228
Домашняя страница 230
Маршрутизация 231
Страница Туры: графический дизайн 234
Страница Туры: интеграция сервера 236
Отправка информации серверу 238
Управление состоянием 241
Варианты развертывания 242
Резюме 243

Глава 17. Статический контент 244
Вопросы производительности 245
Сети доставки контента 246
Проектирование для CDN 247
Веб-сайт с рендерингом на стороне сервера 247
Одностраничные приложения 248
Кэширование статических ресурсов 249
Изменение статического содержимого 249
Резюме 251

Глава 18. Безопасность 252
HTTPS 252
Создание собственного сертификата 253
Использование бесплатного сертификата 255
Покупка сертификата 255
Разрешение HTTPS для вашего приложения в Express 258
Примечание о портах 258
HTTPS и прокси 259
Межсайтовая подделка запроса 260
Аутентификация 262
Аутентификация или авторизация 262
Проблема с паролями 263
Сторонняя аутентификация 263
Хранение пользователей в вашей базе данных 264
Аутентификация или регистрация и пользовательский опыт 265

Passport 266
Авторизация на основе ролей 276
Добавление дополнительных поставщиков аутентификации 277
Резюме 279

Глава 19. Интеграция со сторонними API 280
Социальные медиа 280
Плагины социальных медиа и производительность сайта 280
Поиск твитов 281
Отображение твитов 285
Геокодирование 287
Геокодирование с Google 288
Геокодирование ваших данных 289
Отображение карты 291
Метеоданные 293
Резюме 295

Глава 20. Отладка 296
Первый принцип отладки 296
Воспользуйтесь REPL и консолью 297
Использование встроенного отладчика Node 298
Инспекторы Node 299
Отладка асинхронных функций 303
Отладка Express 304
Резюме 306

Глава 21. Ввод в эксплуатацию 307
Регистрация домена и хостинг 307
Система доменных имен 308
Безопасность 309
Домены верхнего уровня 309
Поддомены 310
Сервер имен 311
Хостинг 313
Развертывание 315
Резюме 318

Глава 22. Поддержка 319
Принципы поддержки 319
Имейте многолетний план 319
Используйте контроль версий 321
Используйте систему отслеживания ошибок 321
Соблюдайте гигиену 322
Не откладывайте 322
Регулярно контролируйте качество 322
Отслеживайте аналитику 323
Оптимизируйте производительность 323
Уделяйте первостепенное внимание отслеживанию потенциальных покупателей 324
Предотвратите незаметные случаи неудачи 325
Повторное использование и рефакторинг кода 326
Приватный реестр npm 326
Промежуточное ПО 327
Резюме 328

Глава 23. Дополнительные ресурсы 329
Онлайн-документация 329
Периодические издания 330
Stack Overflow 330
Содействие развитию Express 332
Резюме 334

Об иллюстрации на обложке 335

Эта книга посвящается моей семье: отцу Тому, привившему мне любовь к технике, маме Энн, пристрастившей меня к сочинительству, и сестре Мэрис — неизменной собеседнице.

Введение

Для кого эта книга

Эта книга предназначена для программистов, желающих создавать веб-приложения (обычные сайты, одностраничные приложения с использованием React, Angular или Vue, REST API или что-то среднее между ними) с помощью JavaScript, Node и Express. Разработка для Node, использующей доступный и гибкий JavaScript, привела к созданию абсолютно нового круга программистов, среди которых оказалось немало самоучек.

Никогда еще в истории информатики программирование не было таким доступным. Количество и качество онлайн-ресурсов для его изучения (и получения помощи в случае возникновения проблем) потрясает и вдохновляет. Так что приглашаю вас стать одним из этих новых (возможно, выучившихся самостоятельно) программистов.

Кроме того, конечно, есть программисты вроде меня, давно работающие в этой сфере. Подобно многим разработчикам моего времени, я начал с ассемблера и языка BASIC, а затем имел дело с Pascal, C++, Perl, Java, PHP, Ruby, C, C# и JavaScript. В университете я узнал о языках программирования более узкого применения, таких как ML, LISP и PROLOG. Многие из них близки и дороги моему сердцу, но ни один из этих языков не кажется мне столь многообещающим, как JavaScript. Так что я пишу эту книгу и для таких программистов, как я сам, с богатым опытом и, возможно, более философским взглядом на определенные технологии.

Опыт работы с Node вам не понадобится, а вот хотя бы небольшой навык владения JavaScript необходим. Если вы новичок в программировании, рекомендую Codecademy (http://www.codecademy.com/tracks/javascript). Если же вы опытный разработчик, советую книгу Дугласа Крокфорда «JavaScript: сильные стороны»[1]. Приведенные в ней примеры актуальны для любой операционной системы, на которой работает Node, включая Windows, OS X и Linux. Они предназначены для людей, работающих с командной строкой (терминалом), так что вам понадобится хотя бы элементарное ее знание в рамках вашей системы.

Самое главное: это книга для программистов-энтузиастов, с оптимизмом смотрящих в будущее Интернета и желающих участвовать в его развитии, а также для

---

[1] *Крокфорд Д.* JavaScript: сильные стороны. — СПб.: Питер, 2013.

тех, кто полон энтузиазма в изучении новых технологий и подходов к разработке веб-приложений. Если, мой уважаемый читатель, вы еще не относите себя к этой категории, надеюсь, что станете таким, когда прочтете книгу до конца.

Примечание ко второму изданию

Я получил удовольствие от написания первого издания этой книги и по сей день доволен практическими рекомендациями, которые мне удалось дать, а также теплым приемом со стороны читателей. Первое издание было опубликовано одновременно с выходом бета-версии Express 4.0, и хотя версия Express по-прежнему 4.x, сопутствующие ему промежуточное программное обеспечение и инструменты претерпели *значительные* изменения. Кроме того, развивался сам JavaScript и даже в способе разработки веб-приложений произошел тектонический сдвиг: от чистого рендеринга на стороне сервера к одностраничным приложениям (single-page applications, SPA). Хотя многие принципы, приведенные в первом издании, все еще актуальны и используются, конкретные методы и инструменты изменились, поэтому было решено выпустить второе издание. В нем, ввиду возрастающей роли SPA, основное внимание уделено Express как серверу для API и статических ресурсов. В книгу также включен пример SPA.

Структура книги

Главы 1 и 2 познакомят вас с Node и Express, а также с инструментами, которые вы будете использовать по ходу чтения книги.

В главах 3 и 4 вы начнете использовать Express и строить каркас учебного сайта, используемого в качестве примера в книге.

В главе 5 обсуждаются тестирование и контроль качества, а в главе 6 описываются некоторые наиболее важные структурные компоненты Node, их расширение и использование в Express.

Глава 7 знакомит с шаблонизацией (использованием семантической системы веб-шаблонов Handlebars) и закладывает основы практического построения сайтов с помощью Express.

Главы 8 и 9 содержат информацию о cookies, сеансах и обработчиках форм, а также очерчивают круг тем, знание которых понадобится для построения сайтов с базовой функциональностью с помощью Express.

В главе 10 исследуется промежуточное ПО (middleware) — центральная концепция Express.

В главе 11 объясняется, как использовать промежуточное ПО для отправки сообщений электронной почты с сервера, а также обсуждаются шаблоны сообщений и относящиеся к электронной почте вопросы безопасности.

Глава 12 предлагает предварительный обзор вопросов, связанных с вводом в эксплуатацию. Хотя на этом этапе прочтения книги у вас еще не будет всей информации,

необходимой для создания готового к использованию сайта, обдумывание ввода в эксплуатацию сейчас избавит вас от множества проблем в будущем.

В главе 13 рассказывается о персистентности данных с упором на MongoDB (одну из основных документоориентированных баз данных) и PostgreSQL (популярную свободно распространяемую реляционную систему управления базами данных).

В главе 14 мы углубимся в подробности маршрутизации в Express (в то, как URL сопоставляются с контентом), а в главе 15 рассмотрим, как создавать API с помощью Express.

В главе 16 проводится реорганизация кода примера с преобразованием его в одностраничное приложение с Express в качестве сервера, предоставляющего API, который был написан в главе 15.

Глава 17 описывает подробности представления статического контента с упором на максимизацию производительности.

В главе 18 обсуждается безопасность: как встроить в ваше приложение аутентификацию и авторизацию (с упором на использование стороннего провайдера аутентификации), а также как организовать доступ к сайту по протоколу HTTPS.

Глава 19 объясняет, как осуществить интеграцию со сторонними сервисами. В качестве примеров приводятся социальная сеть Twitter, картографический сервис Google Maps и американский сервис службы погоды National Weather Service.

Главы 20 и 21 готовят вас к важному моменту — запуску сайта. Они описывают отладку, благодаря которой вы сможете избавиться от каких-либо недостатков перед запуском, и процесс запуска в эксплуатацию.

Глава 22 рассказывает еще об одном важном этапе — сопровождении.

Завершает книгу глава 23, в которой указаны дополнительные источники информации на случай, если вы захотите продолжить изучение Node и Express, а также места, где можно получить помощь и консультацию.

Учебный сайт

Начиная с главы 3, на протяжении всей книги будет использоваться единый пример — сайт турфирмы Meadowlark Travel. Первое издание я написал сразу же после возвращения из поездки в Лиссабон. На тот момент у меня на уме были путешествия, поэтому сайт, выбранный для примера, предназначен для вымышленной туристической фирмы из моего родного штата Орегон (western meadowlark — западный луговой трупиал — певчая птица — символ штата Орегон). Meadowlark Travel связывает путешественников с местными экскурсоводами-любителями и сотрудничает с фирмами, выдающими напрокат велосипеды и мотороллеры и предлагающими туры по окрестностям с акцентом на экотуризме.

Сайт Meadowlark Travel — вымышленный, но это работающий пример, демонстрирующий множество проблем, с которыми сталкиваются реальные сайты (интеграция сторонних компонентов, геолокация, электронная коммерция, безопасность).

Поскольку в центре внимания книги находится инфраструктура серверной части, учебный сайт не будет завершенным. Он лишь послужит вымышленным примером реального сайта, чтобы обеспечить наглядность в рамках определенного контекста. Если вы работаете над собственным сайтом, то сможете использовать Meadowlark Travel в качестве шаблона.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Применяется для обозначения новых понятий и терминов, которые автор хочет особо выделить.

Шрифт без засечек

Используется для обозначения адресов электронной почты и URL-адресов.

`Моноширинный`

Используется для текста (листингов) программ, а также внутри абзацев для выделения элементов программ: имен переменных или функций, названий баз данных, типов данных, имен переменных среды, инструкций и ключевых слов, имен файлов и каталогов.

`Моноширинный полужирный`

Обозначает команды и другой текст, который должен быть набран пользователем.

`Моноширинный курсив`

Показывает в тексте те элементы, которые должны быть заменены значениями, подставляемыми пользователем или определяемыми контекстом.

Этот значок означает совет или предложение.

Такой значок указывает на примечание общего характера.

Этот значок обозначает предупреждение.

Использование примеров исходного кода

Дополнительные материалы (примеры кода, упражнения и т. п.) доступны для скачивания по адресу https://github.com/EthanRBrown/web-development-with-node-and-express-2e.

Эта книга призвана помочь вам в работе. Примеры кода из нее вы можете использовать в своих программах и документации. Если объем кода несущественный, связываться с нами для получения разрешения не нужно. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. А вот для продажи или распространения компакт-диска с примерами из книг издательства O'Reilly нужно получить разрешение. Ответы на вопросы с использованием цитат из этой книги и примеров кода разрешения не требуют. Но для включения объемных примеров кода из этой книги в документацию по вашему программному продукту разрешение понадобится.

Мы не требуем ссылки на первоисточник, но ценим, если вы не забываете об этом. Ссылка на первоисточник включает название, автора, издательство и ISBN. Например, «Веб-разработка с применением Node и Express, Итан Браун, Питер, 2020. 978-1-492-05351-4».

Если вам покажется, что использование кода примеров выходит за рамки оговоренных выше условий и разрешений, свяжитесь с нами по адресу permissions@oreilly.com.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Благодарности

В создании этой книги огромную роль сыграло множество людей. Она не появилась бы без тех, кто повлиял на мою жизнь и сделал меня тем, кто я сейчас.

Я хотел бы начать с благодарности всем сотрудникам Pop Art: работа в компании усилила мою страсть к разработке. Помимо этого, я многому научился у тех, кто здесь работает. Без их поддержки ничего бы не вышло. Я благодарен Стиву Розенбауму за создание столь воодушевляющего места для работы и Дэлу Олдсу — за приглашение в команду, радушный прием и благородство. Спасибо Полу Инману за неизменную поддержку и вдохновляющее отношение к разработке и Тони Алферезе — за содействие и за то, что помог найти время для написания книги без последствий для Pop Art. Наконец, спасибо всем тем выдающимся специалистам, с которыми я работал и поддержку которых ощущал: Джону Скелтону, Дилану Халлстрему, Грегу Юнгу, Куинну Майклсу, Си Джею Стритцелю, Колвину Фритц-Муру, Диане Холланд, Сэму Вилски, Кори Бакли и Демиону Мойеру.

Я в большом долгу перед командой из Value Management Strategies, Inc, с которой сейчас работаю. Роберт Стюарт и Грег Бринк познакомили меня с бизнес-стороной программного обеспечения, а от Эшли Карсон я невероятно много узнал о процессах коммуникации в команде, сотрудничестве и эффективности. Я благодарен Scratch Chromatic за неиссякаемую поддержку. Терри Хейс, Черил Крамер и Эрик Тримбл, спасибо вам за тяжелый труд и помощь! Я также благодарен Демьяну Йетеру, Тайлеру Брентону и Брэду Веллсу за проделанную работу по анализу требований и управлению проектом. Спасибо вам за талантливых и увлеченных разработчиков, которые трудились со мной не покладая рук в VMS: Адама Смита, Шейна Райана, Джереми Лосса, Дэна Мэйса, Майкла Миау, Джулиану Сойфер, Мата Накатани и Джейка Фельдмана.

Спасибо всем моим товарищам по группе в школе рока! Какое это было сумасшедшее путешествие и какая колоссальная творческая разрядка! Особая благодарность преподавателям, которые делятся своими знаниями и страстью к музыке: Джошу Томасу, Аманде Слоун, Дэйву Конильо, Дэну Ли, Дереку Блэкстоуну и Кори Уэст. Спасибо всем за предоставленную мне возможность стать рок-звездой!

Зак Мейсон, спасибо тебе за вдохновение. Это, может быть, и не «Утраченные книги “Одиссеи”», но это *моя* книга, и я не знаю, рискнул бы я писать ее, если бы не твой пример.

Элизабет и Эзра, спасибо за ваши подарки. Я буду любить вас вечно.

Абсолютно всем я обязан своей семье. Я не мог бы желать лучшего воспитания, чем то, что мне дали родители. Я вижу, как и в моей сестре отразилась их исключительная забота о детях.

Большое спасибо Саймону Сен-Лорану за предоставленный шанс и Брайану Андерсону за ободрение и редактуру. Спасибо всему коллективу O'Reilly за преданность делу и энтузиазм. Спасибо Алехандре Ольвера-Новак, Четану Каранду, Брайану Слеттену, Тамасу Пиросу, Дженнифер Пирс, Майку Вилсону, Рэю Виллалобосу и Эрику Эллиоту за детальные конструктивные рецензии.

Кэти Робертс и Ханна Нельсон, спасибо вам огромное за бесценные советы и критику моих сумасшедших идей. Вы помогли этой книге стать реальностью. Спасибо Крису Ковелл-Шаху за великолепный отзыв на главу о тестировании и контроле качества.

И наконец, спасибо моим дорогим друзьям, без которых я просто сошел бы с ума. Байрон Клейтон, Марк Буз, Кэти Роберс, Кимберли Кристенс — люблю вас всех.

Об авторе

Итан Браун — вице-президент компании VMS. Он отвечает за архитектуру и реализацию VMSPro. За плечами Итана более 20 лет программирования в разных предметных областях. Он считает стек JavaScript той платформой, которая обеспечит будущее веб-программирования.

1 Знакомство с Express

Революция JavaScript

Прежде чем начать повествование об основном предмете этой книги, я должен рассказать вам о предпосылках и историческом контексте вопроса, то есть поговорить о JavaScript и Node.

Поистине наступила эпоха JavaScript. Скромно начав свой путь в качестве языка скриптов, выполняемых на стороне клиента, JavaScript стал распространенным языком программирования серверных приложений. Эта его роль резко выросла благодаря Node.

Перспективы всецело основанного на JavaScript стека технологий очевидны: больше никаких переключений контекста! Вам теперь не нужно переключать свои мысли с JavaScript на PHP, C#, Ruby, Python или любой другой серверный язык программирования. Более того, такая особенность позволяет разработчикам клиентской части перейти к программированию серверных приложений. Не стоит думать, однако, что программирование серверных приложений состоит исключительно из языка программирования. Для этого необходимо изучить немало дополнительного материала, но с JavaScript препятствием не будет по крайней мере язык.

Эта книга для всех, кто видит перспективы стека технологий JavaScript. Возможно, вы разработчик клиентской части, который хочет получить опыт серверной разработки. Может быть, вы опытный разработчик клиентских приложений, как я, рассматривающий JavaScript в качестве жизнеспособной альтернативы традиционным серверным языкам.

Если вы пробыли разработчиком программного обеспечения столько, сколько я, вы были свидетелем того, как в моду входили разные языки программирования, фреймворки и API. Некоторые из них остались на плаву, другие морально устарели. Вы, вероятно, гордитесь своим умением быстро изучать новые языки и системы. Каждый новый встречающийся вам язык кажется немного знакомым: один вы изучали в университете, другим пользовались, когда несколько лет назад

выполняли какую-то работу. Смотреть на вещи с такой точки зрения неплохо, однако утомительно. Иногда вам хочется просто *что-то сделать* без необходимости изучать целую новую технологию или вспоминать навыки, которые вы не использовали на протяжении нескольких месяцев или даже лет.

JavaScript на первый взгляд может показаться маловероятным победителем в этом состязании. Если бы в 2007 г. вы сказали мне, что я не только стану считать JavaScript предпочтительным для себя языком, но и напишу книгу о нем, я счел бы вас сумасшедшими. Я разделял все типичные предрассудки относительно JavaScript: думал, что это игрушечный язык программирования, что-то для любителей и дилетантов, чтобы все уродовать и делать неправильно. Ради справедливости скажу, что JavaScript действительно снизил планку для любителей. Это привело к огромному количеству сомнительного JavaScript-кода, что не улучшило репутацию языка.

Очень жаль, что у людей много предрассудков относительно JavaScript. Это мешает им понять, насколько силен, гибок и элегантен этот язык. Многие только сейчас начинают воспринимать его всерьез, хотя язык в нынешнем виде существует примерно с 1996 г. (правда, многие из наиболее привлекательных его возможностей были добавлены лишь в 2005 г.).

Однако, раз вы купили эту книгу, вы, вероятно, свободны от подобных предрассудков. Возможно, подобно мне, вы избавились от них или у вас их никогда и не было. В любом случае вам повезло, и мне не терпится познакомить вас с Express — технологией, которая стала возможной благодаря этому восхитительному и неожиданному языку.

В 2009 г., спустя годы после осознания программистами мощи и выразительности JavaScript как языка написания клиентских скриптов, Райан Даль разглядел потенциал JavaScript как серверного языка и создал Node.js. Это было плодотворное время для интернет-технологий. Ruby (а также Ruby on Rails) заимствовал немало отличных идей из теории вычислительной техники, объединив их с некоторыми собственными новыми идеями, и продемонстрировал миру более быстрый способ создания сайтов и веб-приложений. Корпорация Microsoft в героической попытке прийти в соответствие эпохе Интернета сделала с .NET удивительные вещи, учтя ошибки не только Ruby on Rails, но и языка Java.

Сегодня благодаря технологиям транскомпиляции, таким как Babel, веб-разработчики могут свободно использовать самые последние функции языка JavaScript, не боясь оттолкнуть пользователей с более старыми браузерами. Webpack стал универсальным решением для управления зависимостями в веб-приложениях и обеспечения производительности. Такие фреймворки, как React, Angular и Vue, меняют подход к веб-разработке, а библиотеки для работы с декларативной объектной моделью документов (DOM) (такие как jQuery) — это уже вчерашний день.

Сейчас чудесное время для приобщения к интернет-технологиям. Повсеместно появляются замечательные новые идеи (или воскрешаются прекрасные старые). В наши дни дух инноваций и пробуждения ощутим как никогда.

Знакомство с Express

Сайт Express характеризует Express как «минималистичный и гибкий фреймворк для веб-приложений Node.js, обеспечивающий набор возможностей для построения веб- и мобильных приложений». Что же это такое на самом деле? Разобьем описание на составные части.

- *Минималистичный*. Один из наиболее привлекательных аспектов Express. Много раз разработчики фреймворков забывали, что лучше меньше, да лучше. Философия Express заключается в обеспечении *минимальной* прослойки между вами и сервером. Это не говорит о слабой надежности фреймворка или недостаточном количестве его полезных возможностей. Просто он в меньшей степени становится у вас на пути, позволяя более полно выражать свои идеи. Express предоставляет вам минималистичный фреймворк, а вы можете добавлять необходимый функционал в разных его частях, заменяя то, что вам никогда не понадобится. Это глоток свежего воздуха, ведь многие фреймворки создают раздутые, непонятные и сложные проекты еще до того, как вы написали хотя бы одну строчку кода. Зачастую первой задачей становится отсечение ненужного функционала или замена того, который не соответствует требованиям. Express практикует другой подход, позволяя вам добавлять то, что нужно, там, где нужно.
- *Гибкий*. Механизм действия Express очень прост: он принимает HTTP-запрос от клиента (которым может быть браузер, мобильное устройство, другой сервер, десктопное приложение — одним словом, все, что использует HTTP) и возвращает HTTP-ответ. Этот базовый шаблон описывает практически все, связанное с Интернетом, что делает приложения Express чрезвычайно гибкими.
- *Фреймворк для веб-приложений*. Наверное, более точным описанием будет «серверная часть фреймворка для веб-приложений». На сегодня под «фреймворками для веб-приложений» обычно имеются в виду фреймворки одностраничных приложений, такие как React, Angular или Vue. Тем не менее, за исключением небольшой группы отдельных приложений, большинство веб-приложений должно обмениваться данными и взаимодействовать с другими сервисами. Обычно это происходит через веб-API, который можно рассматривать как серверный компонент фреймворка веб-приложений. Обратите внимание, что построение всего приложения с рендерингом только на стороне сервера по-прежнему возможно (а иногда и желательно). В этом случае Express вполне может быть фреймворком всего веб-приложения!

В дополнение к характеристикам Express, явно указанным в его описании, я бы добавил две собственные.

- *Быстрый*. Express, став одним из лучших фреймворков для платформы Node.js, привлек много внимания со стороны больших компаний с высокопроизводи-

тельными и сильно нагруженными веб-сайтами. Это вынудило команду по разработке Express сконцентрироваться на производительности, и теперь Express — лидер в этом сегменте.

- *Некатегоричный*. Одними из отличительных особенностей экосистемы JavaScript являются ее размер и разнообразие. Ввиду того что Express зачастую занимает центральную позицию в разработке веб-приложений в Node.js, существуют сотни (если не тысячи) страниц сообщества, где подробно рассматриваются приложения Express. Разработчики Express, в свою очередь, предоставили чрезвычайно гибкую систему разработки промежуточного ПО, которая упрощает использование выбранных вами компонентов при создании приложения. В процессе разработки можно заметить, что Express отключает встроенные компоненты в пользу настраиваемого промежуточного ПО.

Я упомянул, что Express является «серверной частью» фреймворка для веб-приложений, поэтому следовало бы рассмотреть взаимодействие между приложениями на стороне сервера и на стороне клиента.

Приложения на стороне сервера и на стороне клиента

В *приложении на стороне сервера* рендеринг страниц (включая HTML, CSS, изображения и другие мультимедийные объекты и JavaScript) происходит на сервере, и затем они отправляются клиенту. В *приложении на стороне клиента*, напротив, большая часть пользовательского интерфейса отображается из исходного кода приложения, отправка которого происходит однократно. То есть после получения начального (обычно очень минимального) HTML-кода браузер динамически преобразует DOM с помощью JavaScript. Отображение новых страниц не зависит от сервера, хотя необработанные данные по-прежнему поступают с него.

Приложения на стороне сервера были стандартом до 1999 г. Фактически термин *«веб-приложение»* был официально введен в этом году. Я считаю период между 1999 и 2012 гг. эпохой Web 2.0, во время которой происходило развитие технологий и технических средств, ставших в дальнейшем приложениями на стороне клиента. К 2012 г., когда смартфоны прочно вошли в обиход, отправка как можно меньшего объема информации через сеть стала общепринятым подходом, что способствовало развитию приложений на стороне клиента.

Приложения на стороне сервера часто называются *приложениями с рендерингом на стороне сервера* (server-side rendered, SSR), а приложения на стороне клиента — *одностраничными приложениями* (SPA). Приложения на стороне клиента полностью реализуются в таких средах, как React, Angular и Vue. Мне всегда казалось, что «одностраничные приложения» — не совсем точное название, ведь с точки зрения пользователя они могут состоять из множества страниц. Единственное

отличие состоит в том, что страницы загружаются с сервера или динамически отображаются на стороне клиента.

На самом деле грань между приложениями на стороне сервера и на стороне клиента размыта. В большинстве приложений на стороне клиента есть 2–3 HTML-пакета, которые могут быть отправлены клиенту (например, открытый интерфейс авторизации или обычный интерфейс и интерфейс администратора). Кроме того, для увеличения производительности загрузки первой страницы и для поисковой оптимизации (search engine optimization, SEO) одностраничные приложения зачастую комбинируются с рендерингом на стороне сервера.

Как правило, если с сервера отправляется небольшое количество HTML-файлов (обычно от одного до трех), а интерфейс пользователя многофункциональный, с большим количеством страниц и основан на динамическом управлении объектной моделью документов, то применяется рендеринг на стороне клиента. Бóльшая часть данных (обычно в формате JSON) и мультимедийных ресурсов для различных представлений по-прежнему загружается из сети.

Для Express не имеет значения, создаете вы приложение на стороне сервера или на стороне клиента: он хорош в обоих случаях. Для него безразлично, какое количество HTML-пакетов вы предоставляете: один или сотню.

Хотя преобладание архитектуры одностраничных веб-приложений неоспоримо, в начале книги приводятся примеры для приложений на стороне сервера. Они все еще актуальны, а разница между предоставлением одного или нескольких HTML-пакетов незначительна. Пример одностраничного приложения приведен в главе 16.

Краткая история Express

Создатель Express Ти Джей Головайчук описывает Express как веб-фреймворк, вдохновленный основанным на Ruby веб-фреймворком Sinatra. Ничего удивительного, что Express заимствует идеи у фреймворка, написанного на Ruby: последний, будучи нацеленным на бóльшую эффективность веб-разработки, ее ускорение и упрощение ее сопровождения, стал источником множества замечательных подходов к веб-разработке.

Хотя Express был вдохновлен фреймворком Sinatra, он тесно переплетен с Connect — подключаемой библиотекой для Node. Connect использует термин *«промежуточное ПО»* (middleware) для описания подключаемых модулей Node, которые могут в различной степени обрабатывать веб-запросы. В 2014 г. в версии 4.0 Express была удалена зависимость от Connect, но этот фреймворк по-прежнему включает позаимствованную из Connect концепцию промежуточного ПО.

Express был подвергнут весьма существенной переработке между версиями 2.x и 3.0, а затем еще раз между 3.x и 4.0. Данная книга посвящена версии 4.0.

Node: новая разновидность веб-сервера

В некотором смысле у Node много общего с другими популярными веб-серверами, такими как разработанный Microsoft веб-сервер Internet Information Services (IIS) или Apache. Но интереснее узнать, в чем его отличия. Так что начнем с этого.

Аналогично Express подход Node к веб-серверам чрезвычайно минималистичен. В отличие от IIS или Apache, для освоения которых могут понадобиться годы, Node легок в установке и настройке. Это не значит, что настройка серверов Node на максимальную производительность в условиях промышленной эксплуатации тривиальна, просто конфигурационные опции проще и яснее.

Другое базовое различие между Node и более традиционными веб-серверами — однопоточность Node. Сначала это может показаться шагом назад, но впоследствии становится ясно, что это гениальная идея. Однопоточность чрезвычайно упрощает задачу написания веб-приложений, а если вам требуется производительность многопоточного приложения, можете просто запустить больше экземпляров Node и, в сущности, получить преимущества многопоточности. Дальновидный читатель, вероятно, посчитает это каким-то шаманством. В конце концов, разве реализация многопоточности с помощью серверного параллелизма (в противоположность параллелизму приложений) просто не перемещает сложность в другое место вместо ее устранения? Возможно, но мой опыт говорит о том, что сложность оказывается перемещенной именно туда, где она и должна быть. Более того, с ростом популярности облачных вычислений и рассмотрения серверов как обычных товаров этот подход становится более разумным. IIS и Apache, конечно, мощные веб-серверы, разработанные для того, чтобы выжимать из современного сильнейшего аппаратного обеспечения производительность до последней капли. Это, однако, имеет свою цену: чтобы добиться такой производительности, для их установки и настройки работникам необходима высокая квалификация.

Если говорить о способе написания приложений, то приложения Node больше похожи на приложения PHP или Ruby, чем на приложения .NET или Java. Движок JavaScript, используемый Node (V8, разработанный компанией Google), не только компилирует JavaScript во внутренний машинный код (подобно C или C++), но и делает это прозрачным образом[1], так что с точки зрения пользователя код ведет себя как чистый интерпретируемый язык программирования. Отсутствие отдельного шага компиляции уменьшает сложность обслуживания и развертывания: достаточно просто обновить файл JavaScript, и ваши изменения автоматически станут доступны.

Другое захватывающее достоинство приложений Node — невероятная независимость Node от платформы. Это не первая и не единственная платформонезависимая серверная технология, но независимо от платформы важнее предлагаемое

[1] Это часто называется компиляцией на лету или JIT-компиляцией (от англ. just in time — «точно в срок»).

разнообразие платформ, чем сам факт ее наличия. Например, вы можете запустить приложение .NET на сервере под управлением Linux с помощью Mono, но это очень нелегкая задача ввиду разнородности документации и системной несовместимости. Аналогично можете выполнять PHP-приложения на сервере под управлением Windows, но их настройка обычно не так проста, как на машине с Linux. В то же время Node элементарно устанавливается на всех основных операционных системах (Windows, OS X и Linux) и облегчает совместную работу. Для команд веб-разработчиков смесь PC и компьютеров Macintosh вполне обычна. Определенные платформы, такие как .NET, задают непростые задачи разработчикам и дизайнерам клиентской части приложений, часто использующим компьютеры Macintosh, что серьезно сказывается на совместной работе и производительности труда. Сама идея возможности подключения работающего сервера на любой операционной системе за считаные минуты (или даже секунды!) — мечта, ставшая реальностью.

Экосистема Node

В сердцевине стека, конечно, находится Node. Это программное обеспечение, которое позволяет выполнять JavaScript-код на сервере без участия браузера, что, в свою очередь, позволяет использовать фреймворки, написанные на JavaScript, такие как Express. Другим важным компонентом является база данных, которая более подробно будет описана в главе 13. Все веб-приложения, кроме самых простых, потребуют базы данных, и существуют базы данных, которые лучше других подходят экосистеме Node.

Ничего удивительного, что имеются интерфейсы для всех ведущих реляционных баз данных (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server): было бы глупо пренебрегать этими признанными китами. Однако наступление эпохи разработки под Node дало толчок новому подходу к базам данных — появлению так называемых NoSQL-баз данных. Не всегда полезно давать чему-либо определение через то, чем оно не является, так что я добавлю, что эти NoSQL-базы данных корректнее было бы называть документоориентированными базами данных или базами данных типа «ключ — значение». Они реализуют более простой с понятийной точки зрения подход к хранению данных. Таких баз данных множество, но MongoDB — одна из лидеров, и именно ее мы будем использовать в книге.

Поскольку создание работоспособного сайта зависит сразу от нескольких технологических составляющих, были придуманы акронимы для описания стека, на котором основан сайт. Например, сочетание Linux, Apache, MySQL и PHP именуется стеком *LAMP*. Валерий Карпов, программист из MongoDB, изобрел акроним *MEAN*: Mongo, Express, Angular и Node. Действительно легко запоминающийся, он имеет и свои ограничения: существует столько разных вариантов выбора баз данных и инфраструктуры разработки приложений, что MEAN не охватывает всего разнообразия экосистемы (а также оставляет за скобками то, что я считаю важным компонентом, — механизм рендеринга).

Придумывание включающего в себя все это акронима — интересная задача. Обязательный компонент, конечно, Node. Хотя есть и другие серверные JavaScript-контейнеры, Node становится преобладающим. Express тоже не единственный доступный фреймворк веб-приложений, хотя он и приближается к Node по распространенности. Два других компонента, обычно существенных для разработки веб-приложений, — сервер баз данных и механизм рендеринга (это либо шаблонизатор наподобие Handlebars, либо фреймворк одностраничных приложений типа React). Для двух последних компонентов очевидных лидеров нет, так что здесь, я полагаю, было бы неверно налагать какие-то ограничения.

Все эти технологии объединяет JavaScript, поэтому, чтобы включить все, я буду называть это *стеком JavaScript*. В данной книге это означает Node, Express и MongoDB (кроме того, в главе 13 приведен пример реляционной базы данных).

Лицензирование

При разработке веб-приложений Node вы можете обнаружить, что уделяете лицензированию больше внимания, чем когда-либо раньше (я определенно уделяю). Одно из преимуществ экосистемы Node — огромный набор доступных пакетов. Однако у каждого из этих пакетов свои правила лицензирования, хуже того, каждый пакет может зависеть от других пакетов, а значит, условия лицензирования различных частей написанного вами приложения могут оказаться запутанными.

Однако есть и хорошие новости. Одна из наиболее популярных лицензий для пакетов Node — лицензия MIT исключительно либеральна и позволяет вам делать *практически* все, что хотите, включая использование пакета в программном обеспечении с закрытым исходным кодом. Но не следует просто предполагать, что каждый используемый вами пакет лицензирован MIT.

В npm доступны несколько пакетов, которые могут попробовать выяснить лицензии для каждой зависимости в вашем проекте. Поищите в npm nlf или license-report.

Хотя MIT — самая распространенная лицензия, вы можете также увидеть следующие лицензии.

- *Стандартная общественная лицензия GNU (GNU General Public License, GPL).* GPL — распространенная лицензия для программного обеспечения с открытым исходным кодом, искусно разработанная для того, чтобы сохранить свободу программного обеспечения. Это значит, что, если вы используете лицензированный по GPL код в своем проекте, проект *тоже* обязан быть GPL-лицензированным. Естественно, это означает, что проект не может иметь закрытый исходный код.

- *Apache 2.0*. Эта лицензия, подобно MIT, позволяет использовать другую лицензию для проекта, в том числе лицензию с закрытым исходным кодом. Вы обязаны, однако, включить уведомление о компонентах, использующих лицензию Apache 2.0.
- *Лицензия Университета Беркли для ПО (Berkeley Software Distribution, BSD)*. Подобно Apache, эта лицензия позволяет использовать для проекта какую угодно лицензию при условии включения уведомления об использующих лицензию BSD компонентах.

Программное обеспечение иногда бывает с двойным лицензированием (лицензировано под двумя различными лицензиями). Часто это происходит из-за желания разрешить использование программного обеспечения как в проектах GPL, так и в проектах с более либеральным лицензированием (чтобы можно было использовать компонент в GPL-лицензированном программном обеспечении, этот компонент сам должен быть GPL-лицензированным). Схема лицензирования, которую я часто использую в своих проектах, — двойное лицензирование GPL и MIT.

Наконец, если вы сами будете писать пакеты, вам следует подобрать лицензию для программного обеспечения и правильно ее задокументировать. Нет ничего более неприятного для разработчика, чем необходимость при использовании чужого пакета копаться в исходном коде, чтобы выяснить, каково лицензирование, или, что еще хуже, обнаружить, что он вообще не лицензирован.

Резюме

Надеюсь, что эта глава дала вам представление об Express и его месте в обширной экосистеме Node и JavaScript, а также внесла некоторую ясность во взаимосвязь между веб-приложениями на стороне сервера и на стороне клиента.

Не стоит беспокоиться, если вы все еще не уверены в том, что понимаете суть Express. Порой для того, чтобы понять что-то, стоит начать это использовать. Эта книга поможет вам начать создавать веб-приложения с Express, но сначала мы рассмотрим Node, ведь это важные базовые сведения, необходимые для понимания работы Express.

2 Первые шаги с Node

Если у вас нет опыта работы с Node, эта глава для вас. Понимание Express и его полезности требует хотя бы минимальных знаний о Node. Если у вас есть опыт создания веб-приложений с помощью Node, спокойно пропускайте эту главу. В ней мы будем создавать веб-сервер с минимальной функциональностью, используя Node. В следующей главе мы увидим, как сделать то же самое с помощью Express.

Установка Node

Установить Node в вашу операционную систему проще простого. Команда разработчиков постаралась сделать так, чтобы процесс установки был максимально прост и ясен на всех основных платформах.

Зайдите на домашнюю страницу Node (http://nodejs.org/) и нажмите большую зеленую кнопку, на которой указан номер версии с пометкой «LTS (рекомендована для большинства пользователей)». Сокращение LTS расшифровывается как Long-Term Support, или *«долгосрочная поддержка»*. Эта версия более стабильна, чем текущая, отмеченная как Current. Она имеет новый функционал и улучшенную производительность.

После выбора Windows или macOS загрузится инсталлятор, который проведет вас по процедуре установки. Если вы предпочитаете Linux, то быстрее будет, вероятно, использовать систему управления пакетами (http://bit.ly/36UYMxI).

Если вы пользователь Linux и хотите работать с системой управления пакетами, убедитесь, что следуете указаниям с вышеупомянутой веб-страницы. Многие дистрибутивы Linux могут установить очень старую версию Node, если вы не добавите соответствующий репозиторий пакетов.

Можно также скачать автономную программу установки (https://nodejs.org/en/download). Она будет полезна, если вы используете Node для работы в организации.

Использование терминала

Я убежденный поклонник терминала (он же — *консоль*, или *командная строка*), и не перестаю восхищаться его мощи и производительности. Все примеры в этой книге предполагают его использование. Если вы не очень дружите с терминалом, настоятельно рекомендую потратить некоторое время на ознакомление с предпочтительным для вас вариантом. У многих утилит, упомянутых в этой книге, есть графический интерфейс, поэтому, если вы категорически против использования терминала, у вас есть выбор, но вам придется разбираться с этим самостоятельно.

Если вы работаете на macOS или Linux, можете выбрать одну из нескольких хорошо зарекомендовавших себя командных оболочек (интерпретаторов командной строки). Наиболее популярная — bash, хотя и у zsh есть свои сторонники. Главная причина, по которой я тяготею к bash (помимо длительного знакомства с ней), — повсеместная распространенность. Сядьте перед любым компьютером, использующим Unix, и в 99 % случаев командной оболочкой по умолчанию будет bash.

Если вы пользователь Windows, дела обстоят не так радужно. Компания Microsoft никогда не была заинтересована в удобстве работы пользователей с терминалом, так что вам придется немного потрудиться. Git любезно включает командную оболочку Git bash, предоставляющую возможности Unix-подобного терминала (в ней есть только небольшое подмножество обычно доступных в Unix утилит командной строки, однако оно весьма полезно). Хотя Git bash предоставляет вам минимальную командную оболочку bash, этот терминал все равно использует встроенную консоль Windows, что приводит лишь к расстройству (даже простые функции вроде изменения размеров окна консоли, выделения текста, вырезания и вставки интуитивно не понятны и выполняются неуклюже). Поэтому я рекомендую установить более продвинутый терминал, такой как ConsoleZ (https://github.com/cbucher/console) или ConEmu (https://conemu.github.io/). Для опытных пользователей Windows, особенно разработчиков .NET, а также для системных или сетевых администраторов есть другой вариант: PowerShell от Microsoft. PowerShell вполне оправдывает свое название[1]: с ее помощью можно делать потрясающие вещи. Опытный пользователь PowerShell может посоревноваться с гуру командной строки Unix. Но, если вы переходите с macOS/Linux на Windows, я все равно рекомендую использовать Git bash из соображений совместимости.

Если у вас Windows 10 или более поздней версии, вы можете установить Ubuntu Linux прямо на Windows! Это не многовариантная загрузка или виртуализация, а отличная работа по интеграции Linux в Windows, проведенная по поручению команды Microsoft по разработке программного обеспечения с открытым исходным кодом. Вы можете установить Ubuntu на Windows через Microsoft App Store (http://bit.ly/2KcSfEI).

---

[1] От англ. power shell — «производительная командная оболочка». — *Примеч. пер.*

Последним вариантом для пользователей Windows является виртуализация. При мощности и архитектуре современных компьютеров производительность виртуальных машин (ВМ) практически не отличается от производительности реальных. Мне когда-то очень повезло с бесплатным VirtualBox от Oracle.

Наконец, независимо от того, в какой операционной системе вы работаете, существуют такие замечательные среды разработки приложений в облаке, как Cloud9 (теперь выпускается AWS) (https://aws.amazon.com/cloud9/). Cloud9 ускорит развертывание новой среды разработки Node, и начать быстро работать с Node будет чрезвычайно просто.

Как только вы выбрали командную оболочку, которая вас устраивает, рекомендую потратить немного времени на изучение основ. Так вы сбережете нервы. В Интернете можно найти немало замечательных руководств (https://guide.bash.academy/). Как минимум вам нужно знать, как передвигаться по каталогам, копировать, перемещать и удалять файлы, а также выходить из программы командной строки (обычно нажатием сочетания клавиш Ctrl+C). Если вы хотите стать ниндзя командной строки, я предлагаю выучить, как осуществлять поиск текста в файлах, поиск файлов и каталогов, объединять команды в последовательность (старая философия Unix) и перенаправлять вывод.

Во многих Unix-подобных системах сочетание клавиш Ctrl+S имеет специальное назначение: его нажатие «замораживает» терминал (когда-то этот прием использовался, чтобы быстро остановить прокрутку страницы). Поскольку это весьма распространенная комбинация клавиш для сохранения (Save), ее легко нажать случайно, что приводит к ситуации, сбивающей с толку большинство людей (со мной это случалось гораздо чаще, чем хотелось бы). Чтобы «разморозить» терминал, просто нажмите Ctrl+Q. Таким образом, если вы когда-нибудь окажетесь сбитыми с толку внезапно замершим терминалом, попробуйте нажать сочетание Ctrl+Q и посмотрите, поможет ли это.

Редакторы

Немногие темы вызывают столь ожесточенные споры среди программистов, как выбор текстового редактора, и на то есть причины. Текстовый редактор — это ваш основной рабочий инструмент. Я предпочитаю редактор vi[1] (или редактор, у которого есть режим vi). vi — редактор не для всех (мои сотрудники обычно смотрят на меня большими глазами, когда я рассказываю им, как удобно было бы сделать в vi то, что они делают), но если вы найдете мощный редактор и научитесь им

[1] В настоящее время vi фактически является синонимом vim (vi improved, усовершенствованный vi). В большинстве систем vi прячется под псевдонимом vim, однако я предпочитаю набирать vim для уверенности, что использую именно его.

пользоваться, ваши производительность и, осмелюсь утверждать, удовольствие от работы значительно возрастут. Одна из причин, по которым я особенно люблю vi (хотя вряд ли самая главная), — то, что, как и bash, он есть везде. Если у вас есть доступ к Unix — у вас есть vi. У большинства популярных текстовых редакторов есть режим vi, который позволяет использовать команды клавиатуры vi. Как только вы к нему привыкнете, вам будет трудно представить себя использующим что-либо другое. vi непрост в освоении, но результат себя оправдывает.

Если, как и я, вы видите смысл в знакомстве с повсеместно доступным редактором, для вас есть еще один вариант — Emacs. Я почти не имел с ним дела (обычно вы или сторонник Emacs, или сторонник vi), но безоговорочно признаю его мощь и гибкость. Рекомендую обратить внимание на Emacs, если модальный подход к редактированию vi не для вас.

Хотя знание консольного текстового редактора, такого как vi или Emacs, может быть полезным, вы, вероятно, захотите использовать более современный редактор. Популярностью пользуется Visual Studio Code (https://code.visualstudio.com/) (не стоит путать его с Visual Studio без Code). Я рекомендую его, ведь это хорошо спроектированный, быстрый и эффективный текстовый редактор, подходящий для разработки в Node и на JavaScript. Другой часто используемый редактор — Atom (https://atom.io/). Он также популярен в сообществе JavaScript. Оба этих редактора бесплатны и поддерживаются Windows, macOS и Linux (и в них обоих есть режим vi!).

Теперь, когда у нас есть хорошее средство для редактирования кода, давайте переключим внимание на npm. С его помощью можно получать пакеты, написанные другими разработчиками, что позволит воспользоваться преимуществом большого и активного сообщества JavaScript.

npm

npm — повсеместно распространенная система управления пакетами для Node (именно таким образом мы получим и установим Express). В отличие от PHP, GNU, WINE и других, образованных странным способом сокращений, npm — не акроним (именно поэтому пишется строчными буквами). Это скорее рекурсивная аббревиатура для «npm — не акроним».

В целом двумя основными задачами системы управления пакетами являются установка пакетов и управление зависимостями. npm — быстрая и эффективная система управления пакетами, которой, как мне кажется, экосистема Node обязана своим быстрым ростом и разнообразием.

У npm есть успешный конкурент, использующий ту же базу данных, — менеджер пакетов Yarn. Мы будем применять его в главе 16.

npm устанавливается при инсталляции Node, так что, если вы следовали перечисленным ранее шагам, она у вас уже есть. Так приступим же к работе!

Основная команда, которую вы будете использовать с npm (что неудивительно), — `install`. Например, чтобы установить nodemon (популярную утилиту для автоматической перезагрузки программы Node после внесения изменений в исходный код), можно выполнить следующую команду в консоли:

```
npm install -g nodemon
```

Флаг `-g` сообщает npm о необходимости *глобальной* установки пакета, означающей его доступность по всей системе. Это различие будет понятнее, когда мы рассмотрим файлы `package.json`. Пока же примем за эмпирическое правило, что утилиты JavaScript, такие как nodemon, обычно будут устанавливаться глобально, а специфические для вашего веб-приложения пакеты — нет.

В отличие от таких языков, как Python, который претерпел коренные изменения между версиями 2.0 и 3.0, что потребовало наличия возможности удобного переключения между различными средами, платформа Node достаточно нова, так что, вероятно, вам следует всегда использовать последнюю версию Node. Но если потребуется поддержка нескольких версий Node, загрузите nvm (https://github.com/creationix/nvm) или n (https://github.com/tj/n), который позволит переключаться между средами. Вы можете узнать, какая версия Node установлена на вашем компьютере, набрав node --version.

Простой веб-сервер с помощью Node

Если вы когда-либо создавали статический сайт или работали с PHP или ASP, вероятно, вам привычна идея веб-сервера (например, Apache), выдающего статические файлы таким образом, что браузер может видеть их по сети. Например, если вы создаете файл `about.html` и помещаете его в соответствующий каталог, то можете затем перейти по адресу http://localhost/about.html. В зависимости от настроек веб-сервера вы можете даже опустить `.html`, но связь между URL и именем файла очевидна: веб-сервер просто знает, где на компьютере находится файл, и выдает его браузеру.

localhost в полном соответствии со своим названием относится к компьютеру, за которым вы работаете. Это распространенный псевдоним для кольцевых адресов 127.0.0.1 (IPv4) и ::1 (IPv6). Часто вместо него применяют 127.0.0.1, но в этой книге я буду использовать localhost. Если вы работаете на удаленном компьютере (с помощью SSH, например), помните, что переход по адресу localhost не соединит вас с ним.

Node предлагает парадигму, отличную от той, что имеет обычный веб-сервер: создаваемое вами приложение и является веб-сервером. Node просто обеспечивает вас фреймворком для создания веб-сервера.

Возможно, вы скажете: «Но я же не хочу писать веб-сервер!» Это вполне естественная реакция: вы хотите писать приложение. Однако Node превращает написание этого веб-сервера в простейшее действо (иногда всего несколько строк), и контроль над вашим приложением, который вы получаете взамен, более чем стоит того.

Вы уже установили Node, освоились с терминалом и теперь самое время приступить к делу.

Hello World!

Я всегда сожалел, что каноническим вводным примером программирования является вывод на экран скучной фразы «Hello World!». Тем не менее будет святотатством бросить вызов столь могучей традиции, так что начнем с этого примера, а после перейдем к чему-то более интересному.

В своем любимом редакторе создайте файл под названием `helloworld.js` (`ch02/00-helloworld.js` в прилагаемом к книге репозитории):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello world!')
})

server.listen(port, () => console.log(`сервер запущен на порте ${port}; ` +
  'нажмите Ctrl+C для завершения...'))
```

В зависимости от того, где и когда вы изучали JavaScript, вас может смутить тот факт, что в данном примере нет точек с запятой. Раньше я был ярым сторонником этого знака препинания и скрепя сердце, перестал его использовать после того, как стал больше разрабатывать в React, где точку с запятой принято опускать. Спустя некоторое время пелена спала с моих глаз и я стал недоумевать, почему вообще точка с запятой вызывала у меня такой энтузиазм! Как покажут примеры в этой книге, я не отношусь к числу решительных противников точки с запятой. Это личный выбор каждого, и вы вольны использовать этот знак препинания, если вам так нравится.

Убедитесь, что вы находитесь в той же папке, что и `helloworld.js`, и наберите `node helloworld.js`. Затем откройте браузер, перейдите на http://localhost:3000, и — вуаля! — ваш первый веб-сервер. Конкретно этот веб-сервер не выдает HTML-код,

скорее он просто показывает сообщение Hello world! в вашем браузере в виде неформатированного текста. Если хотите, можете поэкспериментировать с отправкой HTML вместо этого: просто поменяйте `text/plain` на `text/html` и замените `'Hello world!'` строкой, содержащей корректный HTML-код. Я не буду этого делать, поскольку стараюсь избегать написания HTML-кода внутри JavaScript (причины детально описаны в главе 7).

Событийно-ориентированное программирование

Базовым принципом Node является *событийно-ориентированное программирование*. Для вас как программиста это означает необходимость понимать, какие события вам доступны и как на них реагировать. Многие люди знакомятся с событийно-ориентированным программированием в процессе реализации пользовательского интерфейса: пользователь что-то нажимает — и вы обрабатываете *событие клика*. Это хороший способ, но очевидно, что программист не контролирует момент, когда пользователь что-то нажимает или собирается нажать, поэтому событийно-ориентированное программирование должно быть интуитивно понятным. Мысленный переход к реагированию на события на сервере может оказаться чуть более сложным, но и здесь принцип прежний.

В предыдущем примере кода событие неявное: обрабатываемое событие — HTTP-запрос. Метод `http.createServer` принимает функцию в качестве аргумента, она будет вызываться каждый раз при выполнении HTTP-запроса. Наша простая программа просто устанавливает в качестве типа содержимого неформатированный текст и отправляет строку `Hello world!`.

Как только вы начнете думать, опираясь на термины событийно-ориентированного программирования, вы будете видеть события повсюду. Одним из таких событий является переход пользователя с одной страницы или области приложения на другую. То, как приложение отвечает на этот переход, называется *маршрутизацией*.

Маршрутизация

Маршрутизация относится к механизму выдачи клиенту контента, который он запрашивал. Для клиент-серверных веб-приложений клиент определяет желаемый контент в URL, а именно путь и строку запроса (составные части URL будут подробнее рассмотрены в главе 6).

По сложившейся традиции серверная маршрутизация напрямую зависит от пути и строки запроса, хотя доступна и другая информация: заголовки, домен, IP-адрес и т. д. Благодаря этому сервер может учитывать, например, приблизительное физическое местоположение пользователя или предпочитаемый им язык.

Расширим наш пример с «Hello world!» так, чтобы происходило что-то более интересное. Создадим минимальный сайт, состоящий из домашней страницы, страниц О нас и Не найдено. Пока будем придерживаться предыдущего примера и просто начнем выдавать неформатированный текст вместо HTML-кода (`ch02/01-helloworld.js` в прилагаемом к книге репозитории):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  // Приводим URL к единому виду, удаляя
  // строку запроса, необязательную косую черту
  // в конце строки и переводя в нижний регистр.
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
      break
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' })
      res.end('Not Found')
      break
  }
})

server.listen(port, () => console.log(`сервер запущен на порте ${port}; ` +
  'нажмите Ctrl+C для завершения...'))
```

Если вы запустите это, то обнаружите, что можете переходить на домашнюю страницу (http://localhost:3000) и страницу О нас (http://localhost:3000/about). Любые строки запроса будут проигнорированы, так что http://localhost:3000/?foo=bar вернет вас на домашнюю страницу, а любой другой URL — на страницу Не найдено.

Раздача статических ресурсов

Теперь, когда заработала простейшая маршрутизация, раздадим какой-нибудь реальный HTML-код и логотип. Они носят название *статических ресурсов*, поскольку обычно не изменяются (в отличие, например, от тикера[1]: каждый раз, когда вы перезагружаете страницу, биржевые котировки меняются).

[1] Биржевой инструмент, передающий котировки ценных бумаг. — *Примеч. пер.*

Раздача статических ресурсов с помощью Node подходит для нужд разработки и небольших проектов. В проектах побольше вы, вероятно, захотите использовать для этих целей прокси-сервер, такой как Nginx или CDN. Смотрите главу 17 для получения более подробной информации.

Если вы работали с Apache или IIS, то, вероятно, просто создавали HTML-файл, переходили к нему и автоматически открывали в браузере. Node работает иначе: нам придется выполнить работу по открытию файла, его чтению и отправке его содержимого браузеру. Так что создадим в нашем проекте каталог `public` (в следующей главе станет понятно, почему мы не называем его `static`). В нем создадим `home.html`, `about.html`, `404.html`, подкаталог с названием `img` и изображение с именем `img/logo.jpg`. Выполните эти шаги самостоятельно: раз вы читаете эту книгу, то, вероятно, знаете, как создать HTML-файл и найти картинку. В ваших HTML-файлах ссылайтесь на логотип следующим образом: `<img src="/img/logo.jpg" alt="logo">`.

Теперь внесем изменения в файл `helloworld.js` (`ch02/02-helloworld.js` в репозитории, прилагаемом к книге):

```
const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000

function serveStaticFile(res, path, contentType, responseCode = 200) {
  fs.readFile(__dirname + path, (err, data) => {
    if(err) {
      res.writeHead(500, { 'Content-type': 'text/plain' })
      return res.end('500 – Внутренняя ошибка')
    }
    res.writeHead(responseCode, { 'Content-type': contentType })
    res.end(data)
  })
}

const server = http.createServer((req,res) => {
  // Приводим URL к единому виду, удаляя
  // строку запроса, необязательную косую черту
  // в конце строки и переводя в нижний регистр.
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
```

```
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})

server.listen(port, () => console.log(`сервер запущен на порте ${port}; ` +
  'нажмите Ctrl+C для завершения...'))
```

В этом примере мы проявили не слишком много изобретательности в вопросе маршрутизации. Если перейдете по адресу http://localhost:3000/about, получите файл public/about.html. Путь и файл можно поменять на любые, какие только пожелаете. Например, если у вас есть отдельная страница О нас на каждый день недели, у вас могут быть файлы public/about_mon.html, public/about_tue.html и т. д., а логика маршрутизации может быть построена так, чтобы выдавать соответствующую страницу при переходе пользователя по адресу http://localhost:3000/about.

Обратите внимание, что мы создали функцию-хелпер, `serveStaticFile`, выполняющую большой объем работы. `fs.readFile` — асинхронный метод для чтения файлов. Существует синхронная версия этой функции — `fs.readFileSync`, но чем быстрее вы начнете мыслить асинхронно, тем лучше. В функции `fs.readFile` используется шаблон под названием *«обратный вызов»* (callbacks). Вы предоставляете *функцию обратного вызова* (callback function), и после того, как работа выполнена, происходит вызов этой функции (так сказать, она «вызывается обратно»). В этом случае `fs.readFile` читает содержимое указанного файла и выполняет функцию обратного вызова по завершении чтения файла. Если файла не существует или были проблемы с правами доступа при чтении файла, устанавливается значение переменной `err` и функция возвращает код состояния `HTTP 500`, указывающий на ошибку сервера. Если файл был прочитан успешно, он отправляется клиенту с заданным кодом ответа и типом содержимого. Коды ответа подробнее рассмотрены в главе 6.

Запись __dirname будет соответствовать каталогу, в котором находится выполняемый скрипт. Если ваш скрипт размещен в /home/sites/app.js, __dirname будет соответствовать /home/sites. Использовать такую удобную глобальную переменную везде, где возможно, — хорошая идея. Если этого не сделать, можно получить трудно диагностируемые ошибки при запуске приложения из другого каталога.

Вперед к Express

Вероятно, Node еще сильнее вас впечатлил. Мы, по сути, повторили то, что Apache или IIS делают автоматически, однако теперь вы понимаете, как работает Node и насколько вы можете им управлять. Мы пока не сделали ничего впечатляющего, но вы могли увидеть, как все это можно использовать в качестве отправного пункта для реализации более сложных вещей. Если продолжать движение по данному пути, создавая более сложные приложения Node, можно прийти к чему-то очень похожему на Express.

К счастью, нам не нужно этого делать: Express уже существует и спасает вас от реализации огромного количества трудоемкой инфраструктуры. Так что теперь, когда у нас за плечами есть небольшой опыт работы с Node, мы готовы перейти к изучению Express.

3 Экономия времени благодаря Express

В главе 2 вы узнали, как создать простой веб-сервер с помощью лишь одного Node. В этой главе мы воссоздадим этот же сервер с помощью Express, что станет отправной точкой для работы с книгой и познакомит с основами Express.

Скаффолдинг

Скаффолдинг[1] — идея не новая, но многие люди, включая меня самого, впервые познакомились с этой концепцией благодаря Ruby. Идея проста: большинству проектов требуется определенное количество так называемого *шаблонного*, или *стандартного, кода*, а кому хочется заново писать этот код при создании каждого нового проекта? Простой способ решения проблемы — создать черновой каркас проекта и всякий раз, когда необходимо, просто копировать этот каркас, иначе говоря, шаблон.

Ruby on Rails развивает эту концепцию, обеспечивая программу, автоматически генерирующую скаффолдинг. Преимущество данного подхода в том, что этим способом можно сгенерировать более совершенный фреймворк, чем тот, что получается при обычном выборе из набора шаблонов.

Express следует примеру Ruby on Rails и предоставляет вспомогательную программу для генерации начального скаффолдинга для вашего проекта Express.

Хотя утилита скаффолдинга Express полезна, на мой взгляд, настройку Express стоит изучить с нуля. Вы не только лучше его узнаете, но и получите больше контроля над структурой вашего проекта и над тем, что будет установлено. К тому же утилита скаффолдинга Express приспособлена под генерацию HTML на стороне сервера и менее значима для API и одностраничных приложений.

Хотя мы и не будем использовать утилиту скаффолдинга, рекомендую обратить на нее внимание, прочитав данную книгу: к тому времени вы будете вооружены всеми знаниями, необходимыми для оценки того, подходит ли вам генерируемый

---

[1] От англ. scaffolding — «строительные леса». — *Примеч. пер.*

ею скаффолдинг. За более подробной информацией обращайтесь к документации по генератору приложений Express (http://bit.ly/2CyvvLr).

Сайт Meadowlark Travel

В данной книге будет использоваться единый пример — вымышленный сайт турфирмы Meadowlark Travel, компании, предлагающей услуги тем, кто посещает замечательный штат Орегон. Если вы больше заинтересованы в создании API, не волнуйтесь: помимо собственно функциональности, сайт Meadowlark Travel будет предоставлять и API.

Первые шаги

Начнем с создания корневого каталога проекта. В данной книге везде, где речь о каталоге проекта, каталоге приложения или корневом каталоге, имеется в виду этот каталог.

Возможно, вы захотите хранить файлы вашего веб-приложения отдельно от остальных файлов, обычно сопутствующих проекту, таких как заметки с совещаний, документация и т. п. Чтобы было проще, советую сделать корневым каталогом проекта отдельный подкаталог того каталога, в котором вы будете хранить всю относящуюся к проекту информацию. Например, для сайта Meadowlark Travel я могу держать проект в ~/projects/meadowlark, а корневым каталогом будет ~/projects/meadowlark/site.

npm хранит описание зависимостей проекта — как и относящиеся к проекту метаданные — в файле `package.json`. Простейший способ создать этот файл — выполнить команду `npm init`: программа задаст вам ряд вопросов и сгенерирует `package.json` для начала работы (на вопрос относительно точки входа (entry point) введите `meadowlark.js` или используйте название своего проекта).

Каждый раз при запуске npm вы будете получать предупреждение о том, что поля для описания и репозитория не заполнены. Вы можете игнорировать эти предупреждения, но, если хотите от них избавиться, отредактируйте файл package.json, заполнив поля, как того требует npm. Более подробные сведения о полях в этом файле смотрите в документации по npm package.json (http://bit.ly/2O8HrbW).

Первым шагом будет установка Express. Выполните следующую команду `npm`:

```
npm install express
```

Выполнение `npm install` установит указанный (-е) пакет (-ы) в каталог `node_modules` и обновит файл `package.json`. Поскольку каталог `node_modules` в любой

момент может быть восстановлен с помощью npm, мы не станем сохранять его в нашем репозитории. Чтобы убедиться, что мы не добавили его случайно в репозиторий, создадим файл с именем `.gitignore`:

```
# Игнорировать установленные npm пакеты
node_modules

# Поместите сюда любые другие файлы, которые
# не хотите вносить, такие как .DS_Store (OSX), *.bak и т. д.
```

Теперь создадим файл `meadowlark.js`. Это будет точка входа нашего проекта. На протяжении книги мы будем ссылаться на этот файл как на файл приложения (`ch03/00-meadowlark.js` в прилагаемом репозитории):

```
const express = require('express')

const app = express()

const port = process.env.PORT || 3000

// Пользовательская страница 404
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 – Не найдено')
})

// Пользовательская страница 500
app.use((err, req, res, next) => {
  console.error(err.message)
  res.type('text/plain')
  res.status(500)
  res.send('500 – Ошибка сервера')
})

app.listen(port, () => console.log(
  `Express запущен на http://localhost:${port}; ` +
  `нажмите Ctrl+C для завершения.` ))
```

Многие руководства, равно как и генератор скаффолдинга Express, призывают вас называть основной файл app.js (иногда index.js или server.js). Если вы не пользуетесь хостингом или системой развертывания, требующей определенного имени главного файла приложения, то основной файл лучше назвать по наименованию проекта. Каждый, кто когда-либо вглядывался в кучу закладок редактора, поголовно называвшихся index.html, сразу же оценит мудрость такого решения. npm init по умолчанию даст имя index.js. Если вам нужно другое имя, не забудьте изменить свойство main в файле package.json.

Теперь у вас есть минимальный сервер Express. Можете запустить его (`node meadowlark.js`) и перейти на http://localhost:3000. Результат будет неутешительным: вы не предоставили Express никаких маршрутов, поэтому он просто выдаст вам обобщенную страницу 404, указывающую, что запрошенной страницы не существует.

Обратите внимание, как мы выбрали порт, на котором хотим запустить наше приложение: const port = process.env.PORT || 3000. Это позволяет переопределить порт путем установки переменной среды перед запуском сервера. Если ваше приложение не запускается на порте 3000 при запуске этого примера, проверьте, установлена ли переменная среды PORT.

Добавим маршруты для домашней страницы и страницы О нас. Перед обработчиком ошибки 404 добавляем два новых маршрута (`ch03/01-meadowlark.js` в прилагаемом репозитории):

```
app.get('/', (req, res) => {
  res.type('text/plain')
  res.send('Meadowlark Travel');
})

app.get('/about', (req, res) => {
  res.type('text/plain')
  res.send('О Meadowlark Travel')
})

// Пользовательская страница 404
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 – Не найдено'))
})
```

`app.get` — метод, с помощью которого мы добавляем маршруты. В документации Express вы увидите `app.METHOD`. Не стоит думать, что существует метод с буквальным названием `METHOD`. Это просто заполнитель для ваших (набранных в нижнем регистре) HTTP-методов (наиболее распространенные — GET и POST). Этот метод принимает два параметра: путь и функцию.

Путь — то, что определяет маршрут. Заметьте, `app.METHOD` выполняет за вас тяжелую работу: по умолчанию он игнорирует регистр и косую черту в конце строки, а также не берет во внимание строку запроса при выполнении сравнения. Так что маршрут для страницы О нас будет работать для `/about`, `/About`, `/about/`, `/about?foo=bar`, `/about/?foo=bar` и т. п.

Созданная вами *функция* будет вызываться при совпадении маршрута. Передаваемые этой функции параметры — объекты запроса и ответа, о которых мы больше узнаем в главе 6. Пока же просто возвращаем неформатированный текст с кодом состояния 200 (в Express код состояния по умолчанию равен 200 — необходимости указывать его явным образом нет).

Я настоятельно рекомендую установить плагин к браузеру, который показывал бы код состояния HTTP-запроса, равно как и любые происходящие перенаправления. Это упростит обнаружение проблем перенаправления в вашем коде или ошибочных кодов состояния, которые часто остаются незамеченными. Для браузера Chrome замечательно работает Redirect Path компании Ayima. В большинстве браузеров код состояния вы можете увидеть в разделе «Сеть» инструментов разработчика.

Вместо низкоуровневого метода Node `res.end` мы будем использовать расширения от Express `res.send`. Помимо этого, заменим метод Node `res.writeHead` методами `res.set` и `res.status`. Express также предоставляет нам для удобства метод `res.type`, устанавливающий заголовок `Content-Type`. Несмотря на то что по-прежнему можно использовать методы `res.writeHead` и `res.end`, делать это не рекомендуется.

Обратите внимание, что наши пользовательские страницы 404 и 500 должны обрабатываться несколько иначе. Вместо использования `app.get` применяется `app.use`. `app.use` — метод, с помощью которого Express добавляет *промежуточное ПО*. Более детально мы рассмотрим его в главе 10, а пока вы можете думать о промежуточном ПО как об обобщенном обработчике всего, для чего не находится совпадающего маршрута. Это натолкнет нас на очень важный вывод: *в Express порядок добавления маршрутов и промежуточного ПО имеет значение*. Если мы вставим обработчик ошибки 404 перед маршрутами, домашняя страница и страница `О нас` перестанут функционировать. Их URL будут приводить к странице 404. Пока что наши маршруты довольно просты, но они также поддерживают метасимволы, что может вызвать проблемы с определением порядка следования. Например, если мы хотим добавить к странице `О нас` такие подстраницы, как `/about/contact` и `/about/directions`, следующий код не будет работать ожидаемым образом:

```
app.get('/about*', (req,res) => {
  // Отправляем контент...
}) app.get('/about/contact', (req,res) => {
  // Отправляем контент...
}) app.get('/about/directions', (req,res) => {
  // Отправляем контент...
})
```

В этом примере обработчики `/about/contact` и `/about/directions` никогда не будут достигнуты, поскольку первый обработчик содержит метасимвол в своем пути: `/about*`.

Express может различить обработчики ошибок 404 и 500 по количеству аргументов, принимаемых их функциями обратного вызова. Ошибочные маршруты будут подробнее рассмотрены в главах 10 и 12.

Теперь вы можете снова запустить сервер и убедиться в работоспособности домашней страницы и страницы `О нас`.

До сих пор мы не делали ничего, что нельзя было бы так же легко выполнить без Express, но Express уже предоставил нам некоторую не совсем тривиальную функциональность. Помните, как в предыдущей главе приходилось приводить `req.url` к единому виду, чтобы определить, какой ресурс был запрошен? Нужно было вручную убрать строку запроса и косую черту в конце строки, а также преобразовать к нижнему регистру. Маршрутизатор Express теперь обрабатывает эти нюансы автоматически. И хотя сейчас это может показаться не такой уж и важной вещью, это лишь малая часть того, на что способен Express.

Представления и макеты

Если вы хорошо знакомы с парадигмой «Модель — представление — контроллер», то концепция представления для вас не нова. По сути, *представление* — то, что выдается пользователю. В случае с сайтом это, как правило, HTML-код, хотя вы также можете выдавать PNG, PDF или что-то другое, что может быть отображено клиентом. В нашем случае будем полагать, что представление — это HTML-код.

Отличие представления от статического ресурса, такого как изображение или файл CSS, в том, что представление не обязано быть статическим: HTML-код может быть создан на лету для формирования персонализированной страницы для каждого запроса.

Express поддерживает множество разных механизмов представлений, обеспечивающих различные уровни абстракции. В какой-то степени он отдает предпочтение механизму представления *Pug* (что неудивительно, ведь это тоже детище Ти Джея Головайчука). Используемый Pug подход весьма минималистичен: то, что вы пишете, вообще не похоже на страницу HTML — количество набираемого текста уменьшилось, нет никаких угловых скобок и закрывающих тегов. Движок Pug получает это на входе и преобразует в HTML-код.

Первоначально Pug назывался Jade, но название изменили при релизе второй версии из-за проблем с товарным знаком.

Pug привлекателен, однако подобный уровень абстракции имеет свою цену. Если вы разработчик клиентской части, вам нужно хорошо понимать HTML-код даже в том случае, когда вы фактически пишете свои представления в Pug. Большинство моих знакомых разработчиков клиентской части ощущают дискомфорт

при одной лишь мысли об абстрагировании их основного языка разметки. Поэтому я рекомендую использовать другой, менее абстрактный, фреймворк шаблонизации — *Handlebars*.

Handlebars, основанный на популярном, независимом от языка программирования языке шаблонизации Mustache, не пытается абстрагировать HTML: вы пишете HTML-код с помощью специальных тегов, позволяющих Handlebars внедрять контент.

За годы, прошедшие после первого издания этой книги, React покорил мир… и это абстрагирует HTML от разработчиков фронтенда! Ранее я предположил, что фронтенд-разработчики не захотят абстрагироваться от HTML, но мой прогноз не выдержал проверки временем. Тем не менее я был отчасти прав, ведь JSX — расширение языка JavaScript, которым пользуется большинство разработчиков, практически идентичен написанию HTML.

Чтобы обеспечить поддержку Handlebars, будем использовать пакет `express-handlebars`, созданный Эриком Феррайоло. Выполните следующее в вашем каталоге проекта:

```
npm install express-handlebars
```

Затем модифицируйте несколько первых строк в `meadowlark.js` (`ch03/02-meadowlark.js` в прилагаемом репозитории):

```
const express = require('express')
const expressHandlebars = require('express-handlebars')

const app = express()

// Настройка механизма представлений Handlebars.
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```

Это создает механизм представления и настраивает Express для использования по умолчанию. Теперь создадим каталог `views` с подкаталогом `layouts`. Если вы опытный веб-разработчик, то, вероятно, хорошо знакомы с понятием *макетов* (иногда их называют *шаблонами страниц*). Когда вы создаете сайт, одни и те же (или почти одни и те же) определенные фрагменты HTML-кода повторяются на каждой странице. Переписывать весь этот дублирующийся код не только утомительно, но и рискованно: если вы захотите изменить что-то на каждой странице, вам придется изменять *все* файлы. Макеты освобождают вас от этой необходимости, обеспечивая общий фреймворк для всех страниц вашего сайта.

Итак, создадим шаблон для нашего сайта. Создайте файл `views/layouts/main.handlebars`:

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

Единственное, чего вы, вероятно, до сих пор не видели, — это `{{{body}}}`. Данное выражение будет замещено HTML-кодом для каждого представления. Обратите внимание, что при создании экземпляра Handlebars мы указываем макет по умолчанию (`defaultLayout:'main'`). Это значит, что, если вы не укажете иное, для любого представления будет использоваться именно этот макет.

Теперь создадим страницы представления для нашей домашней страницы, `views/home.handlebars`:

```
<h1>Добро пожаловать в Meadowlark Travel</h1>
```

Затем для страницы О нас, `views/about.handlebars`:

```
<h1>О Meadowlark Travel</h1>
```

Затем для страницы Не найдено, `views/404.handlebars`:

```
<h1>404 – Не найдено</h1>
```

И наконец, для страницы Ошибка сервера, `views/500.handlebars`:

```
<h1>500 – Ошибка сервера</h1>
```

Вероятно, вы захотите, чтобы ваш редактор ассоциировал .handlebars и .hbs (другое распространенное расширение для файлов Handlebars) с HTML, дабы иметь возможность использовать подсветку синтаксиса и прочие возможности редактора. Что касается vim, можете добавить строку au BufNewFile,BufRead *.handlebars set file type=html в ваш файл ~/.vimrc. В случае использования другого редактора загляните в его документацию.

Теперь, когда мы задали представления, необходимо заменить старые маршруты новыми, использующими эти представления (`ch03/02-meadowlark.js` в прилагаемом репозитории):

```
app.get('/', (req, res) => res.render('home'))

app.get('/about', (req, res) => res.render('about'))

// Пользовательская страница 404
app.use((req, res) => {
  res.status(404)
```

```
  res.render('404')
})

// Пользовательская страница 500
app.use((err, req, res, next) => {
console.error(err.message)
  res.status(500)
  res.render('500')
})
```

Замечу, что нам больше не нужно указывать тип контента или код состояния: механизм представления по умолчанию будет возвращать тип контента `text/html` и код состояния 200. В обобщенном обработчике, отвечающем за пользовательскую страницу 404, и обработчике ошибки 500 нам приходится устанавливать код состояния явным образом.

Если вы запустите свой сервер и проверите домашнюю страницу или страницу `О нас`, то увидите, что представления были отображены. Если посмотрите на исходный код, обнаружите, что там есть шаблонный HTML из `views/layouts/main.handlebars`.

Эти пути рассматриваются как *динамический контент*, даже если при каждом посещении домашней страницы вы получаете один и тот же HTML, так как мы можем принимать разные решения для каждого запроса (с чем мы неоднократно столкнемся позже в этой книге). В то же время контент, который действительно никогда не меняется, то есть статический, важен и широко используется, поэтому мы рассмотрим его ниже.

Статические файлы и представления

При обработке статических файлов и представлений Express полагается на *промежуточное ПО*. Это понятие более подробно рассмотрено в главе 10, пока же достаточно знать, что промежуточное ПО обеспечивает разбиение на модули, упрощая обработку запросов.

Промежуточное ПО `static` позволяет объявлять один из каталогов как содержащий статические ресурсы, которые довольно просты для того, чтобы их можно было предоставлять пользователю без какой-либо особой обработки. Именно сюда вы можете поместить картинки, файлы CSS и клиентские файлы на JavaScript.

В своем каталоге проекта создайте подкаталог с именем `public` (мы назвали его так, поскольку все в нем будет раздаваться без каких-либо дополнительных вопросов). Затем перед объявлением маршрутов добавьте промежуточное ПО `static` (файл `ch03/02-meadowlark.js` в прилагаемом репозитории):

```
app.use(express.static(__dirname + '/public'))
```

Промежуточное ПО `static` приводит к тому же результату, что и создание для каждого статического файла, который вы хотите выдать, маршрута, отображающего

файл и возвращающего его клиенту. Так что создадим подкаталог `img` внутри каталога `public` и поместим туда наш файл `logo.png`.

Теперь мы просто можем сослаться на `/img/logo.png` (обратите внимание: мы не указываем `public` — этот каталог невидим для клиента), и промежуточное ПО `static` выдаст этот файл, установив соответствующий тип контента. Сейчас исправим наш макет так, чтобы логотип появлялся на каждой странице:

```
<body>
  <header>
    <img src="/img/logo.png" alt="Логотип Meadowlark Travel">
  </header>
  {{{body}}}
</body>
```

Помните, что промежуточное ПО обрабатывается в определенном порядке, и промежуточное ПО static, которое обычно объявляется первым или очень рано, переопределит другие маршруты. Например, если вы поместите файл index.html в каталог public, то обнаружите, что вместо настроенного вами маршрута будет выдано содержимое этого файла! Таким образом, если вы получили неправильный результат, проверьте ваши статические файлы и убедитесь, что маршрут не совпадает с чем-то неожиданным.

Динамический контент в представлениях

Представления — не просто усложненный способ выдачи статического HTML (хотя они, конечно, могут делать и это). Настоящая сила представлений в том, что они могут содержать динамическую информацию.

Допустим, мы хотим на странице `О нас` выдавать виртуальное печенье с предсказаниями. Определим в файле `meadowlark.js` массив печенья с предсказаниями:

```
const fortunes = [
  "Победи свои страхи, или они победят тебя.",
  "Рекам нужны истоки.",
  "Не бойся неведомого.",
  "Тебя ждет приятный сюрприз.",
  "Будь проще везде, где только можно.",
]
```

Измените представление (`/views/about.handlebars`) для отображения предсказаний:

```
<h1>О Meadowlark Travel</h1>
{{#if fortune}}
  <p>Твое предсказание на сегодня:</p>
  <blockquote>{{fortune}}</blockquote>
{{/if}}
```

Теперь поменяем маршрут `/about` для выдачи случайного печенья с предсказанием:

```
app.get('/about', (req, res) => {
  const randomFortune = fortunes[Math.floor(Math.random()*fortunes.length)]
  res.render('about', { fortune: randomFortune })
})
```

Сейчас, если вы перезапустите сервер и загрузите страницу `/about`, то увидите случайное предсказание и будете получать новое при каждой перезагрузке страницы. Шаблонизация крайне полезна. Ее мы более подробно рассмотрим в главе 7.

Резюме

Мы создали сайт с помощью Express. Несмотря на свою простоту, он содержит все основное, что необходимо для полнофункционального сайта. В следующей главе мы расставим точки над i в подготовке к добавлению более продвинутой функциональности.

4 Наводим порядок

В двух предыдущих главах мы еще только экспериментировали. Прежде чем приступить к более сложной функциональности, займемся кое-какими организационными вопросами и возьмем на вооружение несколько полезных для работы привычек.

В этой главе мы всерьез возьмемся за проект Meadowlark Travel. Однако перед тем, как приступить к созданию сайта, убедимся в наличии всех необходимых для этого инструментов.

Вы не обязаны следовать единому примеру, изложенному в этом издании. Если вы стремитесь создать собственный сайт, можете соблюдать структуру примера, изменяя его так, чтобы по прочтении книги у вас получился законченный сайт.

Структура файлов и каталогов

Вокруг структуры приложений идет множество споров, и единственно верного способа ее создания не существует. Тем не менее есть несколько общепринятых соглашений, о которых стоит знать.

Обычно стараются ограничить количество файлов в корневом каталоге проекта. Как правило, здесь вы найдете конфигурационные файлы (такие как `package.json`), файл `README.md` и несколько каталогов. Бóльшая часть исходного кода зачастую находится в папке `src`. Из соображений краткости я не буду использовать это соглашение в данной книге (удивительно, но приложение скаффолдинга Express также этого не делает). Работая над реальными проектами, вы, вероятно, рано или поздно обнаружите, что размещение исходного кода в корневом каталоге проекта приводит к беспорядку в нем, и захотите поместить эти файлы в каталог наподобие `src`.

Я также упоминал, что предпочитаю давать основному файлу моего проекта (который иногда называют точкой входа) имя, соответствующее названию самого

проекта (например, `meadowlark.js`), а не какое-нибудь стандартное, как `index.js`, `app.js` или `server.js`.

В целом структура проекта остается на ваше усмотрение, но я рекомендую предоставлять ее схему в файле `README.md` (или в связанном с ним файле `readme`).

Помимо этого, я советую, чтобы в корневом каталоге вашего проекта всегда было как минимум два файла: `package.json` и `README.md`. Все остальное зависит от вашего воображения.

Лучшие решения

Словосочетание «лучшие решения» — одно из тех, которые сейчас очень модно использовать. Оно подразумевает, что вам следует делать все правильно и не срезать углы (через минуту мы обсудим, что это конкретно значит). Вы, вероятно, слышали фразу: «Есть три возможных варианта: быстро, дешево и хорошо, но из них всегда нужно выбрать два». В этой модели меня всегда не устраивало то, что она не учитывает *накопление навыков* при правильном выполнении какого-то действия. Первый раз, когда вы делаете что-то правильно, это может занять в пять раз больше времени по сравнению с выполнением того же действия кое-как. Во второй раз это потребует лишь втрое больше времени, а к тому моменту, когда вы верно выполните задачу раз десять, вы будете делать это так же быстро, как если бы делали все тяп-ляп.

У меня был тренер по фехтованию, который всегда говорил, что практика не приводит к безупречности. Она дает *стабильность*. То есть если вы делаете что-либо снова и снова, постепенно эти действия доводятся до автоматизма. Это правда, однако при этом ничего не говорится о качестве выполняемых вами действий. Если вы практикуете плохие навыки, то именно они и будут доведены до автоматизма. Вместо этого следуйте правилу: *безупречная* практика делает результат безупречным. Поэтому я призываю вас так придерживаться дальнейших примеров из этой книги, как если бы вы делали реальный сайт и ваша репутация, а также гонорар зависели от качества результатов. Используйте эту книгу для оттачивания *хороших* навыков, а не просто для приобретения новых.

Решения, на которых мы сосредоточимся, — контроль версий и обеспечение качества (QA). В этой главе мы обсудим контроль версий, а в следующей — обеспечение качества.

Контроль версий

Надеюсь, мне не нужно убеждать вас в важности контроля версий (если бы и пришлось, то это заняло бы целую книгу). Он дает следующие преимущества.

- *Наличие документации.* Возможность обратиться к истории проекта и посмотреть, какие решения были приняты и какова очередность разработки компонентов,

способна дать ценную информацию. Наличие технической истории вашего проекта может быть весьма полезным.

- *Установление авторства*. Если вы работаете в команде, установление авторства может оказаться чрезвычайно важным. Всякий раз, когда вы обнаруживаете в коде неясные или сомнительные места, знание того, кто выполнил соответствующие изменения, может сберечь часы работы. Возможно, сопутствующих изменению комментариев окажется достаточно для ответа на вопросы, но, если нет, вы будете знать, к кому обращаться.
- *Экспериментирование*. Хорошая система контроля версий дает возможность экспериментировать. Вы можете отклониться от курса и попробовать что-то новое, не боясь повлиять на стабильность своего проекта. Если эксперимент оказался успешным, можно включить его в проект, если нет — отбросить.

Много лет назад я перешел на распределенную систему контроля версий (DVCS). Я сузил свой выбор до Git и Mercurial и в итоге остановился на Git из-за его гибкости и распространенности. Обе — великолепные бесплатные системы контроля версий, и я рекомендую использовать одну из них. В этой книге мы будем использовать Git, но вы можете заменить ее на Mercurial (или другую систему контроля версий).

Если вы не знакомы с Git, советую замечательную книгу Иона Лелигера «Управление версиями с помощью Git»[1] (http://bit.ly/Version_Ctrl_Git). Кроме того, в GitHub есть прекрасный список образовательных ресурсов Git (https://try.github.io/).

Как использовать Git с этой книгой

Во-первых, убедитесь, что у вас есть Git. Наберите в командной строке `git --version`. Если в ответ вы не получили номер версии, нужно установить Git. Обратитесь к документации Git за инструкциями по инсталляции (https://git-scm.com/).

Есть два способа работы с примерами из этой книги. Один из них — набирать примеры вручную и сверять их с помощью команд Git. Другой — клонировать репозиторий Git, используемый мной для примеров, и извлекать код из соответствующих файлов для каждого примера. Некоторые люди лучше учатся, набирая примеры, в то время как другие предпочитают просто смотреть и запускать их, не прибегая к набору руками.

Если вы набираете примеры самостоятельно

У нас уже есть приблизительный скелет нашего проекта: представления, макет, логотип, основной файл приложения и файл `package.json`. Пойдем дальше — создадим репозиторий Git и внесем в него все эти файлы.

---

[1] *Loeliger J.* Version Control with Git. — O'Reilly, 2012.

Во-первых, заходим в каталог проекта и инициализируем там репозиторий Git:

```
git init
```

Теперь, перед тем как добавить все файлы, создадим файл `.gitignore`, чтобы защититься от случайного добавления того, что мы добавлять не хотим. Создайте файл с именем `.gitignore` в каталоге вашего проекта. В него вы можете добавить любые файлы или каталоги, которые Git по умолчанию должен игнорировать (по одному на строку). Этот файл также поддерживает метасимволы. Например, если ваш редактор создает резервные копии файлов с тильдой в конце (вроде `meadowlark.js~`), вы можете вставить строку `*~` в файл `.gitignore`. Если вы работаете на компьютере Macintosh, вам захочется поместить в этот файл `.DS_Store`. Вы также захотите вставить туда `node_modules` (по причинам, которые мы скоро обсудим). Итак, пока файл будет выглядеть примерно так:

```
node_modules
*~
.DS_Store
```

Элементы файла .gitignore применяются также к подкаталогам. Если вы вставили *~ в файл .gitignore в корневом каталоге проекта, то все подобные резервные копии файлов будут проигнорированы, даже если они находятся в подкаталогах.

Теперь можем внести все имеющиеся у нас файлы. Есть много способов сделать это в Git. Я предпочитаю `git add -A`, как имеющий наибольший охват из всех вариантов. Если вы новичок в Git, рекомендую вносить файлы по одному (например, `git add meadowlark.js`), если вы хотите зафиксировать изменения только в одном или двух файлах; или использовать `git add -A`, если хотите внести все изменения, включая любые файлы, которые вы могли удалить. Поскольку мы хотим внести все, что мы сделали, используем следующее:

```
git add -A
```

Новичков в Git обычно сбивает с толку команда git add: она вносит изменения, а не файлы. Так, если вы изменили файл meadowlark.js и затем набираете git add meadowlark.js, вы вносите выполненные изменения.

У Git есть область подготовленных файлов, куда попадают изменения, когда вы выполняете `git add`. Так, внесенные нами изменения еще не были зафиксированы, но готовы к этому. Чтобы зафиксировать их, используйте `git commit`:

```
git commit -m "Первоначальный коммит."
```

Фрагмент `-m "Первоначальный коммит."` позволяет написать сообщение, связанное с этим коммитом. Git даже не разрешит вам выполнить коммит без сообщения, и на то есть серьезные причины. Всегда старайтесь писать содержательные сообщения при фиксации изменений: они должны сжато, но выразительно описывать выполненную вами работу.

Если вы используете официальный репозиторий

Чтобы получить официальный репозиторий для этой книги, выполните `git clone`:

```
git clone https://github.com/EthanRBrown/web-development-with-node-and-express-2e
```

В этом репозитории есть каталог для каждой[1] главы, содержащий примеры кода. Например, исходный код для данной главы можно найти в каталоге `ch04`. Примеры кода для каждой главы в основном названы так, чтобы на них было проще ссылаться. Я щедро добавил по всему репозиторию файлы `README.md` с дополнительными примечаниями к примерам.

В первом издании этой книги я применял другой подход к использованию репозитория — с линейной историей, как если бы вы разрабатывали постепенно усложняющийся проект. Хотя этот подход удачно отражал ход разработки проекта в реальном мире, он доставлял множество хлопот как мне, так и читателям. С изменением пакетов npm примеры кода тоже изменились бы, а хорошего способа обновить репозиторий или отметить изменения в тексте (за исключением переписывания всей истории репозитория) не было бы. Хотя подход с каталогом на главу является более искусственным, он позволяет точнее синхронизировать текст с репозиторием, а также облегчает сообществу внесение своего вклада.

По мере обновления и уточнения этой книги репозиторий будет обновляться. При этом я буду добавлять тег версии, так что вы сможете извлечь версию репозитория, соответствующую версии читаемой вами книги. Текущая версия репозитория — 2.0.0. Здесь я более или менее следую принципам *семантического контроля версий* (semantic versioning), о чем подробнее будет рассказано далее в этой главе. Увеличение PATCH (последнего номера) соответствует незначительным изменениям, которые не должны повлиять на выполнение примеров из этой книги. То есть если версия репозитория 2.0.15, то он по-прежнему должен соответствовать версии, которая используется в книге. Однако если MINOR (средний номер) отличается (например, 2.1.0), то это говорит о том, что сопутствующий книге репозиторий может расходиться с тем, который вы читаете, и вам может понадобиться версия, начинающаяся с тега 2.0.

[1] Кроме главы 2, к которой нет кода в репозитории. — *Примеч. пер.*

В прилагаемом к книге репозитории для добавления дополнительных пояснений к примерам кода широко используются файлы `README.md`.

Если в какой-то момент вы захотите поэкспериментировать, помните, что извлечение тега из репозитория приводит к состоянию, которое Git именует detached HEAD[1]. Несмотря на то что в этом случае вы можете редактировать любые файлы, фиксировать какие-либо изменения без предварительного создания ветки небезопасно. Если вы действительно хотите создать экспериментальную ветку на основе тега, просто создайте новую ветку и извлеките ее из репозитория, что можно выполнить командой git checkout -b experiment (где experiment — название вашей ветки; вы можете использовать любое, какое пожелаете). После этого спокойно редактируйте и фиксируйте изменения в этой ветке столько, сколько захотите.

Пакеты npm

Пакеты npm, от которых зависит ваш проект, находятся в каталоге `node_modules` (плохо, что он называется `node_modules`, а не `node_packages`, так как модули Node — другое, хотя и связанное понятие). Не стесняйтесь заглядывать в этот каталог, чтобы удовлетворить любопытство или отладить свою программу, но никогда не меняйте там код. Во-первых, это плохая практика, а во-вторых, все ваши изменения легко могут оказаться уничтоженными npm.

Если вам нужно внести изменение в пакет, от которого зависит проект, правильным шагом будет создание собственного ответвления. Если вы пойдете по этому пути и обнаружите, что ваши изменения могли быть полезными кому-то еще, поздравляю: теперь вы участвуете в проекте с открытым исходным кодом! Вы можете представить свои изменения, и если они соответствуют стандартам проекта, то будут включены в официальный пакет. Внесение вклада в существующие пакеты и создание пользовательских сборок выходит за рамки данной книги, но существует активное сообщество разработчиков, которые помогут, если вы захотите внести вклад в существующие пакеты.

У файла `package.json` есть два основных предназначения: описать ваш проект и перечислить зависимости. Посмотрите на свой файл `package.json` прямо сейчас. Вы должны увидеть что-то подобное (конкретные номера версий будут, вероятно, другими, так как эти пакеты часто обновляются):

```
{
  "dependencies": {
    "express": "^4.16.4",
    "express-handlebars": "^3.0.0"
  }
}
```

[1] Букв. «оторванная голова». — *Примеч. пер.*

Пока файл `package.json` содержит только информацию о зависимостях. Знак вставки (`^`) перед версиями пакетов обозначает, что любая версия, начинающаяся с указанного номера версии, вплоть до следующего номера мажорной версии будет работать. Например, этот `package.json` говорит, что работоспособной будет любая версия Express, начинающаяся с 4.0.0. Так, и 4.0.1, и 4.9.9 — обе будут работать, а 3.4.7 — нет, как и 5.0.0. Подобная специфика версионности принята по умолчанию при использовании `npm install` и в общем случае представляет собой довольно безопасную стратегию. Таким образом, если вы хотите перейти к новой версии, вам придется отредактировать файл `package.json`, указав в нем новую версию. Вообще, это неплохо, поскольку предотвращает ситуацию, при которой ваш проект (но без вашего ведома) перестанет работать из-за изменений в зависимостях. Анализ номеров версий в npm осуществляет компонент под названием *semver* (от semantic versioning — семантический контроль версий). Больше информации о контроле версий в npm вы можете получить, обратившись к спецификации семантического контроля версий (Semantic Versioning Specification) и статье Тамаса Пироса (http://try.github.io/).

В спецификации семантического контроля версий указано, что программное обеспечение, которое использует семантический контроль версий, должно объявлять «публичный API». Такая формулировка всегда казалась мне непонятной; на самом деле имелось в виду, что «кто-то должен позаботиться о взаимодействии с вашим ПО». В широком смысле это можно трактовать как угодно, поэтому не зацикливайтесь на этой части спецификации; важные моменты связаны с форматом.

Поскольку файл `package.json` перечисляет все зависимости, каталог `node_modules`, по сути, вторичный артефакт. Если бы вы его удалили, то для восстановления работоспособности проекта нужно было бы лишь выполнить команду `npm install`, которая заново создала бы этот каталог и поместила в него все необходимые зависимости. Именно поэтому я рекомендовал вам поместить `node_modules` в файл `.gitignore` и не включать его в систему контроля исходного кода. Однако некоторые люди считают, что репозиторий должен включать все необходимое для работы проекта, и предпочитают держать `node_modules` в системе контроля исходного кода. Я же считаю, что это мусор в репозитории, и предпочитаю его избегать.

Начиная с версии 5, npm создает дополнительный файл package-lock.json. В то время как в package.json допускается свобода в спецификации версий зависимостей (используются модификаторы версий ^ и ~), в package-lock.json точно указываются установленные версии. Это может пригодиться, если вам понадобится воссоздать точные версии зависимостей в вашем проекте. Я рекомендую зарегистрировать этот файл в системе контроля исходного кода и не изменять его вручную. За более подробной информацией обращайтесь к документации package-lock.json (http://bit.ly/2O8IjNK).

Метаданные проекта

Другой задачей файла `package.json` является хранение метаданных проекта, таких как его название, сведения об авторах, информация о лицензии и т. д. Если вы воспользуетесь командой `npm init` для первоначального создания файла `package.json`, она заполнит файл всеми необходимыми полями и вы в любое время сможете их изменить. Если вы собираетесь сделать свой проект доступным в npm или GitHub, эти метаданные становятся критическими. Если хотите получить больше информации о полях в файле `package.json`, загляните в документацию `package.json` (http://bit.ly/2X7GVPs). Другая важная часть метаданных — файл `README.md`. Этот файл — удобное место для описания как общей архитектуры сайта, так и любой критической информации, которая может понадобиться тому, кто будет иметь дело с проектом впервые. Он находится в текстовом вики-формате, называемом Markdown. За получением более подробной информации обратитесь к документации по Markdown (http://bit.ly/2q7BQur).

Модули Node

Как уже упоминалось, модули Node и пакеты npm — понятия связанные, но имеющие различия. *Модули Node*, как понятно из названия, предоставляют механизм модуляризации и инкапсуляции. *Пакеты npm* обеспечивают стандартизированную схему для хранения проектов, контроля версий и ссылок на проекты, которые не ограничиваются модулями. Например, мы импортируем сам Express в качестве модуля в основном файле приложения:

```
const express = require('express')
```

`require` — функция Node для импорта модулей. По умолчанию Node ищет модули в каталоге `node_modules` (неудивительно, что внутри `node_modules` будет каталог `express`). Тем не менее Node также предоставляет механизм создания ваших собственных модулей (лучше никогда не создавать собственные модули в каталоге `node_modules`). Кроме модулей, устанавливаемых менеджером пакетов в `node_modules`, Node предоставляет еще более 30 базовых модулей, таких как fs, http, os и path. Полный список можно найти на Stack Overflow (http://bit.ly/2NDIkKH) и в официальной документации по Node (https://nodejs.org/en/docs/).

Посмотрим, как мы можем разбить на модули функциональность печений-предсказаний, реализованную в предыдущей главе.

Сначала создадим каталог для хранения модулей. Вы можете назвать его как угодно, но обычно он носит название `lib` (сокращение от library — «библиотека»). В этой папке создайте файл с названием `fortune.js` (`ch04/lib/fortune.js` в прилагаемом репозитории):

```
const fortuneCookies = [
  "Победи свои страхи, или они победят тебя.",
```

```
  "Рекам нужны истоки.",
  "Не бойся неведомого.",
  "Тебя ждет приятный сюрприз.",
  "Будь проще везде, где только можно.",
]

exports.getFortune = () => {
  const idx = Math.floor(Math.random()*fortuneCookies.length)
  return fortuneCookies[idx]
}
```

Здесь важно обратить внимание на использование глобальной переменной `exports`. Если вы хотите, чтобы что-то было видимым за пределами модуля, необходимо добавить это в `exports`. В приведенном примере функция `getFortune` доступна извне нашего модуля, но массив `fortunes` будет *полностью скрыт*. Это хорошо: инкапсуляция позволяет создавать менее подверженный ошибкам и более надежный код.

Есть несколько методов экспорта функциональности из модуля. Мы будем рассматривать различные методы по ходу книги и подытожим их в главе 22.

Теперь в `meadowlark.js` мы можем удалить массив `fortunes` (хотя ничего страшного не случится, если его оставить: он никоим образом не будет конфликтовать с одноименным массивом, определенным в `lib/fortune.js`). Принято (хотя и не обязательно) указывать импортируемые объекты вверху файла, так что добавьте вверху файла `meadowlark.js` (`ch04/meadowlark.js` в прилагаемом репозитории) следующую строку:

```
const fortune = require('./lib/fortune')
```

Обратите внимание, что мы поставили перед нашим модулем префикс `./`. Это предупреждает Node о том, что он не должен искать модуль в каталоге `node_modules`; если бы мы опустили этот префикс, произошла бы ошибка.

Теперь в маршруте для страницы О нас мы можем использовать метод `getFortune` из нашего модуля:

```
app.get('/about', (req, res) => {
  res.render('about', { fortune: fortune.getFortune() } )
})
```

Если вы набираете примеры самостоятельно, зафиксируем эти изменения:

```
git add -A
git commit -m "Moved 'fortune cookie' functionality into module."
```

Вы обнаружите, что модули — чрезвычайно мощный и удобный способ инкапсуляции функциональности, улучшающий общую концепцию и сопровождаемость

вашего проекта, а также облегчающий его тестирование. За получением более подробной информации обратитесь к официальной документации Node (https://nodejs.org/api/modules.html).

Модули Node иногда называют модулями CommonJS (CJS), ссылаясь на более старую спецификацию, которая была взята за основу Node. Язык JavaScript в данный момент принимает официальный механизм пакетов под названием ECMAScript Modules (ESM). Если вам приходилось писать на JavaScript в React или другом прогрессивном языке для фронтенда, вы можете быть знакомы с ESM, использующим import и export вместо exports, module.exports и require. За более подробными сведениями обращайтесь к блог-посту доктора Акселя Раусшмайера «Окончательный синтаксис модулей ECMAScript 6» (http://bit.ly/2X8ZSkM).

Резюме

Теперь, вооружившись информацией о Git, npm и модулях, мы готовы рассмотреть возможность создания лучшего продукта путем применения практик обеспечения качества (QA) в нашем коде.

Я призываю учитывать следующие выводы из этой главы.

- Контроль версий делает процесс разработки ПО более безопасным и предсказуемым, и я рекомендую использовать его даже в маленьких проектах. Это формирует хорошие привычки.
- Модуляризация — важная техника для управления сложностью ПО. В дополнение к богатой экосистеме модулей, созданной другими разработчиками в npm, вы можете разместить ваш собственный код в модулях для улучшения структуры проекта.
- У модулей Node (также называемых CJS) другой синтаксис, чем у модулей ECMAScript (ESM), и, вполне вероятно, вам придется переключаться между двумя синтаксисами при переходе между кодом клиентской и серверной частей. Хорошо, если вы будете знать их оба.

5 Обеспечение качества

Обеспечение качества — словосочетание, наводящее ужас на разработчиков (чего быть не должно). Разве вы не хотите делать качественное программное обеспечение? Конечно же, хотите. Так что камень преткновения здесь не конечная цель, а отношение к данному вопросу. Я обнаружил, что в веб-разработке возможны два типичных положения дел.

- *Большие или богатые организации*. В них обычно имеется подразделение QA и, к сожалению, нередко возникает соперничество между QA и разработчиками. Это худшее, что может произойти. Оба подразделения играют в одной команде, стремятся к одной цели, но QA часто считают успехом нахождение как можно большего числа ошибок, а разработчики — порождение как можно меньшего их количества, что становится источником конфликтов и соперничества.
- *Небольшие и бюджетные организации*. В них часто нет подразделения QA. Предполагается, что разработчики будут выполнять двойную функцию: обеспечивать качество и разрабатывать программное обеспечение. Это не полет фантазии и не конфликт интересов, но QA значительно отличается от разработки: оно привлекает совсем других специалистов и требует иных талантов. Такая ситуация не является вымышленной. Безусловно, существуют разработчики со складом ума тестировщиков, но практика показывает, что, как только на горизонте замаячит дедлайн, обеспечению качества уделяется меньше внимания, что наносит вред проекту.

Широко распространена практика возлагать функции, которые традиционно выполнялись QA, на разработчиков, делая последних ответственными за обеспечение качества. В этой парадигме специалисты в области QA выступают фактически как консультанты разработчиков, помогая им встраивать обеспечение качества в их технологические процессы. Обособлены роли QA или объединены, очевидно, что понимание этого процесса полезно для разработчиков.

Эта книга — не для профессионалов в области QA. Она предназначена для разработчиков. Так что моя цель — не сделать из вас эксперта в QA, а всего лишь

обеспечить немного практики в этой сфере. Если в вашей организации есть отдельный штат специалистов по QA, вам станет легче общаться и сотрудничать с ними. Если же нет, все изложенное мной станет отправной точкой для создания исчерпывающего плана по обеспечению качества вашего проекта.

В этой главе вы изучите следующие вопросы.

- Основы обеспечения качества и эффективные привычки.
- Типы тестов (модульное и интеграционное тестирование).
- Написание модульных тестов с Jest.
- Интеграционное тестирование с Puppeteer.
- Как настроить ESLint, чтобы избежать распространенных ошибок.
- Что такое непрерывная интеграция и с чего начать ее изучение.

План по обеспечению качества

Разработка — это творческий процесс, в ходе которого воображаемое становится действительным. QA, напротив, больше относится к области валидации и порядка. Таким образом, значительная часть QA сводится к тому, что *нужно получить сведения о том, что должно быть сделано*, и *убедиться, что все необходимое было реализовано*. Эта дисциплина хорошо подходит для списков проверок, процедур и документации. Я бы даже сказал, что первоочередная задача QA заключается не в тестировании ПО, а в *создании исчерпывающего, воспроизводимого плана по обеспечению качества*.

Рекомендую создавать план по обеспечению качества для каждого проекта независимо от его размера (да, даже для вашего проекта, которым вы занимаетесь по выходным ради развлечения!). План по обеспечению качества не должен быть большим или подробным, вы можете разместить его в текстовом файле, документе текстового редактора или вики. Он предназначен для записи всех шагов, предпринимаемых вами для того, чтобы убедиться, что ваш продукт функционирует должным образом.

План по обеспечению качества в любой его форме — актуализируемый документ. Вы будете обновлять его в следующих случаях.

- Появление новой функциональности.
- Изменения в существующей функциональности.
- Удаление функциональности.
- Изменения в технологиях или методах тестирования.
- Выявление недостатков, не учтенных в плане по обеспечению качества.

Последний пункт заслуживает особого внимания. Недостатки выявляются независимо от того, насколько надежно ваше QA. Когда это происходит, вам нужно

спросить себя: «Как я мог это предотвратить?» Ответ на этот вопрос позволит внести в план по обеспечению качества изменения, которые в будущем помогут избежать подобных недоработок.

Теперь вы можете прочувствовать, каких существенных затрат требует QA, и у вас, вероятно, возникнет вполне обоснованный вопрос: «Сколько усилий я готов приложить?»

QA: стоит ли оно того?

QA может обойтись весьма недешево. Так стоит ли овчинка выделки? Это непростая формула со многими входными параметрами. Большинство предприятий работают по какой-либо схеме окупаемости инвестиций. Если вы тратите деньги, то ожидаете получить обратно как минимум столько же (желательно больше). С QA, однако, это соотношение может быть неочевидным. Хорошо зарекомендовавшему себя солидному продукту проблемы с качеством могут сходить с рук дольше, чем новому, никому не известному проекту. Безусловно, никто *не хочет* производить заведомо низкокачественный продукт, но давление обстоятельств в сфере технологий весьма высоко. Время выхода на рынок может быть критическим, и иногда лучше выйти на него прямо сейчас с чем-то не совсем идеальным, чем с идеальным, но с опозданием на два месяца.

В веб-разработке качество может рассматриваться в четырех аспектах.

- *Охват*. Данный аспект характеризует степень проникновения вашего продукта на рынок, а именно указывает на количество людей, просматривающих ваш сайт или использующих ваш сервис. Существует прямая корреляция между охватом и доходностью: чем выше посещаемость сайта, тем больше людей покупает продукт или сервис. С точки зрения разработки на охват сильнее всего влияет оптимизация поисковых систем (SEO), поэтому мы включим это понятие в план по обеспечению качества.
- *Функциональность*. Если люди посещают ваш сайт (или используют сервис), степень его функциональности будет сильно влиять на удержание пользователей: сайт, работающий так, как утверждает его реклама, с большей вероятностью привлечет повторных посетителей, чем тот, что работает хуже. Тестирование функциональности проще всего автоматизировать.
- *Удобство использования*. В то время как функциональность связана с правильностью работы, удобство использования относится к оценке взаимодействия «человек — компьютер» (human-computer interaction, HCI). Основной вопрос здесь: «Подходит ли способ предоставления функциональности для целевой аудитории?» Он часто интерпретируется как «Удобно ли это использовать?», хотя погоня за удобством может вредить гибкости или производительности: кажущееся удобным программисту, может не быть таковым для потребителя, не имеющего соответствующего уровня подготовки и технических знаний.

Другими словами, при оценке удобства использования стоит учитывать целевую аудиторию. Поскольку главным источником информации для измерения удобства использования является пользователь, оно не поддается автоматизации. Как бы то ни было, пользовательское тестирование необходимо включить в план по обеспечению качества.

- *Эстетика*. Эстетика — наиболее субъективная и по этой причине наименее относящаяся к разработке характеристика. Когда дело доходит до эстетичности сайта, проблем, касающихся непосредственно разработки, возникает мало. Тем не менее регулярный анализ эстетичности сайта должен входить в план по обеспечению качества. Покажите сайт репрезентативной пробной аудитории и выясните, не кажется ли он устаревшим, вызывает ли желаемую реакцию. Помните, что эстетика зависит от времени (эстетические стандарты меняются с годами) и аудитории (то, что привлекает одних людей, может быть совершенно неинтересно другим).

Хотя все четыре аспекта должны быть представлены в вашем плане по обеспечению качества, автоматизированно во время разработки можно выполнить только функциональное тестирование и тестирование SEO, поэтому в данной главе мы сосредоточимся именно на них.

Логика и визуализация

В вашем сайте есть два «царства»: *логика* (часто называемая *бизнес-логикой* — это термин, которого я избегаю по причине его коммерческого уклона) и *визуализация*. Вы можете рассматривать логику вашего сайта как нечто существующее в чисто интеллектуальной области. Например, на нашем сайте Meadowlark Travel может быть принято правило, согласно которому для аренды скутера клиент обязан иметь действующее водительское удостоверение. Это очень простое в отношении баз данных правило: при каждом бронировании скутера пользователю необходим номер действующего водительского удостоверения. Визуализация этого отделена от логики. Возможно, пользователь должен просто поставить флажок в итоговой форме заказа или указать номер действующего водительского удостоверения, который Meadowlark Travel затем проверит. Это важное отличие, поскольку в логической области все должно быть настолько просто и ясно, насколько это возможно, тогда как визуализация может быть такой сложной (или простой), как это необходимо. Визуализация относится к вопросам удобства использования и эстетики, в то время как область бизнес-логики — нет.

Везде, где только возможно, вы должны четко разделять логику и визуализацию. Существует много способов сделать это. В данной книге мы сосредоточимся на инкапсуляции логики в модулях JavaScript. В то же время визуализация будет сочетанием HTML, CSS, мультимедиа, JavaScript и библиотек клиентской части, таких как React, Vue или Angular.

Виды тестов

Тесты, которые мы будем обсуждать в этой книге, делятся на две обширные категории: *модульные* и *интеграционные* (комплексное тестирование я рассматриваю как подвид интеграционного). Модульное тестирование осуществляется на уровне очень мелких структурных единиц — это тестирование отдельных компонентов, призванное проверить, функционируют ли они должным образом, в то время как интеграционное тестирование испытывает взаимодействие многих компонентов или системы целиком.

В целом модульное тестирование удобнее и лучше подходит для тестирования логики. Интеграционное тестирование пригодно для обеих областей.

Обзор методов QA

В этой книге для выполнения всестороннего тестирования мы будем использовать следующие методы и программное обеспечение.

- *Модульное тестирование*. Модульные тесты распространяются на наименьшие функциональные единицы приложения, обычно это отдельные функции. Эти тесты почти полностью пишутся разработчиками, а не QA, хотя последние должны обладать полномочиями оценки качества и охвата модульных тестов. В этой книге мы будем использовать Jest для модульного тестирования.
- *Интеграционное тестирование*. Интеграционные тесты охватывают бóльшие функциональные единицы, задействовав при этом многие части приложения (функции, модули, подсистемы и т. д.). Поскольку мы создаем веб-приложения, окончательный интеграционный тест будет состоять из визуализации приложения в браузере, манипуляций с этим браузером и проверки соответствия поведения приложения ожидаемому. Эти тесты, как правило, сложнее в настройке и поддержке, и, поскольку QA не является целью этой книги, на эту тему будет дан лишь один простой пример с использованием Puppeteer и Jest.
- *Линтинг*[1]. Это не просто поиск ошибок, а обнаружение *потенциальных* ошибок. Общая концепция линтинга: нахождение участков, являющихся источниками потенциальных ошибок, или нестабильных компонентов, которые могут привести к ошибкам в будущем. Для линтинга мы будем использовать ESLint.

  Начнем с Jest — фреймворка для тестирования, в котором будут запускаться как модульные, так и интеграционные тесты.

---

[1] Это название происходит от Unix-утилиты lint — первоначально статического анализатора кода для языка программирования C, выполнявшего поиск подозрительных выражений или выражений, потенциально не переносимых на другие компиляторы/платформы. — *Примеч. пер.*

Установка и настройка Jest

Решение о том, какой фреймворк для тестирования использовать в этой книге, далось мне с трудом. Jest начал свое существование в качестве фреймворка для тестирования приложений React и по-прежнему остается очевидным кандидатом на эту роль. Но Jest не предназначен исключительно для React и является великолепным универсальным фреймворком для тестирования. При этом Mocha (https://mochajs.org/), Jasmine (https://jasmine.github.io/), Ava (https://github.com/avajs/ava) и Tape (https://github.com/substack/tape) — тоже хороший выбор.

В итоге я остановился на Jest, потому что, как мне кажется, он является лучшим с точки зрения общего опыта применения. Это мнение основано на отличных оценках, которые Jest получил по результатам опроса JavaScript-разработчиков, проведенного по итогам 2018 г. (http://bit.ly/33ErHUE). Тем не менее у перечисленных здесь фреймворков много общего, поэтому вы сможете применить полученные знания в работе с вашим любимым фреймворком для тестирования.

Для установки Jest запустите следующую команду из корневого каталога вашего проекта:

```
npm install --save-dev jest
```

Заметьте, что здесь мы используем `--save-dev`, чтобы сказать npm о том, что это зависимость, обусловленная разработкой, и приложение может функционировать без нее. Она будет указана в разделе `devDependencies` файла `package.json` вместо раздела зависимостей.

Прежде чем мы приступим к работе с Jest, нам нужно определить, каким способом он будет запускаться на выполнение (при этом будут выполняться все тесты в нашем проекте). Общепринятым способом является добавление скрипта в `package.json`. Отредактируйте файл `package.json` (`ch05/package.json` в прилагаемом репозитории), изменив свойство `scripts` или добавив его, если такового не существует:

```
"scripts": {
  "test": "jest"
},
```

Теперь вы сможете запустить на выполнение все тесты в вашем проекте, просто набрав:

```
npm test
```

Если вы попробуете сделать это сейчас, то, скорее всего, получите сообщение о том, что нет настроенных тестов, ведь мы их еще не добавили. Так давайте напишем какой-нибудь модульный тест!

Обычно если вы добавляете скрипт в файл package.json, то запускаете его с помощью npm run. Например, добавив скрипт foo, для его запуска на выполнение вы бы набирали npm run foo. Тем не менее скрипт test является настолько распространенным, что для его запуска достаточно напечатать npm test.

Модульное тестирование

Ввиду того что целью модульного тестирования является выделение отдельных функций или компонентов, нам в первую очередь нужно познакомиться с методикой, позволяющей осуществить это выделение, — с mock-объектами.

Mock-объекты

Одной из часто возникающих трудностей является необходимость написания кода, пригодного для тестирования. Вообще, код, содержащий излишнюю функциональность или множество зависимостей, тестировать труднее, чем узкоспециализированный код, в котором зависимостей мало или совсем нет.

Чтобы тестирование было эффективным, зависимости нужно имитировать *mock-объектами*. Например, наша первая зависимость — Express — уже всесторонне протестирована, поэтому нам нужно тестировать не сам Express, а только то, *как мы его используем*. Определить корректность использования Express можно лишь посредством моделирования самого Express.

Созданные нами маршруты (домашняя страница, страница О нас, страницы 404 и 500) достаточно сложно тестировать, потому что у них есть три зависимости от Express, считая приложение Express (то есть `app.get`), а также объекты запроса и ответа. К счастью, зависимость от самого приложения Express довольно просто исключить. С объектами запроса и ответа дела обстоят хуже. Об этом будет рассказано позже. Наше положение упрощает тот факт, что мы используем не так уж много функциональности из объекта ответа (только метод `render`), поэтому его будет легко подменить mock-объектом, что мы вкратце рассмотрим.

Рефакторинг приложения для упрощения его тестирования

В нашем приложении еще не так много кода для тестирования. На данный момент мы добавили только несколько обработчиков маршрутов и функцию `getFortune`.

Чтобы наше приложение было легче тестировать, мы *извлечем* действительные обработчики маршрутов в их собственную библиотеку. Создайте файл `lib/handlers.js` (`ch05/lib/handlers.js` в прилагаемом репозитории):

```
const fortune = require('./fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) =>
  res.render('about', { fortune: fortune.getFortune() })

exports.notFound = (req, res) => res.render('404')

exports.serverError = (err, req, res, next) => res.render('500')
```

Для того чтобы воспользоваться этими обработчиками, мы можем переписать файл приложения `meadowloark.js` (`ch05/meadowlark.js` в прилагаемом репозитории):

```
// Обычно находится в начале файла
const handlers = require('./lib/handlers')

app.get('/', handlers.home)

app.get('/about', handlers.about)

// Пользовательская страница 404
app.use(handlers.notFound)

// Пользовательская страница 500
app.use(handlers.serverError)
```

Теперь эти обработчики проще тестировать: они стали просто функциями, принимающими объекты запроса и ответа, и нам нужно подтвердить, что мы корректно используем эти объекты.

Наш первый тест

Есть много способов идентификации тестов в Jest. Двумя наиболее распространенными являются размещение тестов в подкаталоге с названием `__test__` (по два нижних подчеркивания до и после `test`) и добавление к именам файлов расширения `.test.js`. Мне нравится комбинировать эти методы, поскольку, на мой взгляд, они оба служат этой цели. Размещение тестов в каталогах `__test__` предотвращает захламление каталогов с исходным кодом. В противном случае будет казаться, что в папке с исходным кодом все продублировано: у вас будет `foo.test.js` для каждого файла `foo.js`. Расширения `.test.js` позволят легко различать тесты и исходный код на вкладках.

Создадим файл под названием `lib/__tests__/handlers.test.js` (`ch05/lib/__tests__/handlers.test.js` в прилагаемом репозитории):

```
const handlers = require('../handlers')

test('home page renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.home(req, res)
  expect(res.render.mock.calls[0][0]).toBe('home')
})
```

Если вы новичок в тестировании, то все происходящее может показаться довольно странным, поэтому давайте разберемся.

Сначала мы импортируем код, который нужно протестировать (в нашем случае это обработчики маршрутов). Затем идет описание каждого теста, в данном случае нам нужно удостовериться, что домашняя страница отображается.

Для обращения к рендеру понадобятся объекты запроса и ответа. На то чтобы полностью смоделировать объекты запроса и ответа, ушла бы целая неделя, но, к сча-

стью, в этом нет необходимости. В данном случае нам ничего не нужно от объекта запроса, поэтому мы просто воспользуемся пустым объектом. А от объекта ответа нам требуется только метод `render`. Обратите внимание на создание функции рендеринга: мы просто вызываем метод Jest под названием `jest.fn()`. При этом создается обобщенная функция-заглушка, которая отслеживает то, как вызывается `render`.

Наконец, мы подобрались к важной части теста — утверждениям. Мы проделали много работы по вызову кода, который тестируем, но как мы докажем, что он делает то, что должен? В данном случае код должен вызывать метод `render` объекта ответа со строкой `home`. Функция-заглушка из Jest отслеживает все его вызовы, поэтому нам нужно лишь проверить, что он вызывается только один раз (при его двукратном вызове, вероятно, возникли бы проблемы). Это именно то, что сначала выполняет метод `expect`, а затем он вызывается с `home` в качестве первого аргумента. Первый индекс массива указывает на номер вызова, а второй индекс — на номер аргумента.

Многократные перезапуски тестов при каждом внесении изменений в код могут быть утомительными. К счастью, в большинстве фреймворков есть метод, который постоянно отслеживает изменения в вашем коде и тестах и перезапускает их автоматически. Для запуска тестов в режиме отслеживания наберите npm test -- --watch (дополнительный двойной дефис нужен для того, чтобы npm передал Jest аргумент --watch).

Измените ваш обработчик домашней страницы так, чтобы он отображал что-нибудь другое, нежели представление домашней страницы. Вы заметите, что ваш тест провалился и у вас была обнаружена ошибка!

Теперь можно добавить тесты для других маршрутов.

```
test('страница О нас отображается с предсказанием', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.about(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('about')
  expect(res.render.mock.calls[0][1])
    .toEqual(expect.objectContaining({
        fortune: expect.stringMatching(/\W/),
    }))
})

test('рендеринг обработчика ошибки 404', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.notFound(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('404')
})

test('рендеринг обработчика ошибки 500', () => {
  const err = new Error('some error')
```

```
  const req = {}
  const res = { render: jest.fn() }
  const next = jest.fn()
  handlers.serverError(err, req, res, next)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('500')
})
```

Обратите внимание на дополнительную функциональность в тестах для ошибки сервера и страницы О нас. Функция отображения страницы О нас вызывается с предсказанием, поэтому мы добавили ожидание того, что функции будет передано предсказание в виде строки, содержащей как минимум один символ. Описание всей функциональности, предоставляемой Jest и его методом `expect`, выходит за рамками данной книги, но вы можете найти исчерпывающую документацию на домашней странице Jest. Имейте в виду, что обработчик ошибки сервера принимает четыре аргумента, а не два, в связи с чем понадобятся дополнительные заглушки.

Поддержка тестов

Как вы уже, вероятно, поняли, тесты — это не то, что можно сделать и забыть. Например, если мы по уважительным причинам переименуем представление домашней страницы, наш тест провалится и придется исправлять не только код, но и тест.

Из-за этого команды разработчиков прилагают значительные усилия к установлению реалистичных ожиданий относительно того, какими и насколько конкретными должны быть тесты. Например, для того, чтобы увидеть, что обработчик страницы О нас был вызван с предсказанием, нам не нужна проверка. Это избавит нас от необходимости исправлять тест, если мы отбросим данную характеристику.

Кроме того, я не могу дать много советов насчет того, насколько обстоятельно нужно тестировать код. Полагаю, что стандарты, которые применяются к тестированию кода для авиационной электроники и медицинского оборудования, будут существенно отличаться от тех, которыми вы пользуетесь в случае с коммерческим веб-сайтом.

Могу лишь предложить ответ на вопрос «Как много моего кода протестировано?». Это будет обсуждаться далее как *покрытие кода тестами*.

Покрытие кода тестами

Покрытие кода тестами дает численный ответ о том, насколько ваш код протестирован, но, как и в большинстве задач программирования, простых решений здесь нет.

Jest любезно предоставляет средства автоматического анализа покрытия кода. Чтобы увидеть, насколько ваш код протестирован, запустите следующую команду:

```
npm test -- --coverage
```

Если вы набирали примеры вручную, то должны увидеть несколько ободряюще зеленых показателей покрытия кода на 100 % для файлов из каталога `lib`. Jest вы-

даст отчет о выраженном в процентах покрытии операторов (Stmts), ветвлений, функций (Funcs) и строк.

Показатель покрытия операторов соотносится с операторами JavaScript, такими как всевозможные выражения, операторы управления потоком выполнения и т. д. Нужно отметить, что даже при 100%-ном покрытии строк операторы могут быть покрыты не полностью. Это связано с тем, что в JavaScript возможно размещение нескольких операторов в одной строке. Показатель покрытия ветвлений подразумевает операторы управления потоком выполнения, такие как `if-else`. Если в операторе `if-else` тест проходит только по ветке `if`, то показатель покрытия ветвлений для этого оператора составит 50 %.

Как вы могли заметить, `meadowlark.js` покрыт тестами не на 100 %. Это не всегда стоит считать проблемой. Взглянув на файл `meadowlark.js` после рефакторинга, можно увидеть, что большую его часть теперь составляют настройки; мы просто соединяем все вместе. Настраиваем Express с соответствующим промежуточным ПО и запускаем сервер. Тестировать этот код сложно и не нужно, поскольку в нем всего лишь происходит компоновка хорошо протестированного кода.

Можно поспорить, что от тестов, которые мы до сих пор писали, нет особой пользы. К тому же в них мы просто проверяем правильность настройки Express.

Снова у меня нет простых ответов. В конечном итоге то, насколько глубоко вы погрузитесь в дебри тестирования, в значительной степени будет обусловлено типом создаваемого приложения, вашим опытом, размером и составом вашей команды. Я призываю отдавать предпочтение *чрезмерному* тестированию над *недостаточным*, а, накопив опыт, вы сами найдете золотую середину.

ТЕСТИРОВАНИЕ ЭНТРОПИЙНОЙ ФУНКЦИОНАЛЬНОСТИ

Тестирование *энтропийной* функциональности (функциональности, являющейся случайной) ставит непростые задачи. Еще один тест, который мы могли бы добавить для нашего генератора печений-предсказаний, — тест для проверки возврата им действительно *случайного* печенья-предсказания. Но как мы можем узнать, случайна ли эта случайность? Одним из подходов может быть получение большого числа предсказаний, например тысячи, с последующим измерением распределения ответов. Если функция вполне случайна, ни один ответ не будет выделяться. Недостаток этого подхода в его недетерминистичности: можно (хотя и маловероятно) получить одно предсказание в десять раз чаще, чем любое другое. Если такое произойдет, тест завершится неудачно (здесь важно, насколько жесткий порог того, что считать случайным, вы установили), но на самом деле это может быть лишь следствием тестирования энтропийных систем и никак не показателем того, что тестируемая система не работает. Для нашего генератора предсказаний было бы уместно сгенерировать 50 предсказаний и ожидать появления хотя бы трех различных. В то же время, если бы мы разрабатывали источник случайных чисел для научного моделирования или компонента системы безопасности, нам бы, возможно, захотелось намного более обстоятельных тестов. Суть в том, что тестирование энтропийной функциональности является трудным делом и требует более тщательного продумывания.

Интеграционное тестирование

На данный момент в нашем приложении нет ничего интересного для тестирования; у нас в наличии лишь несколько страниц и нет никакого взаимодействия. Поэтому, прежде чем писать интеграционный тест, нам нужно добавить функциональность, которую можно тестировать. Для простоты пусть это будет ссылка, позволяющая перейти с домашней страницы на страницу `О нас`. Нет ничего проще! И все же, каким бы простым все это ни казалось, с точки зрения пользователя, это реальный интеграционный тест, ведь он проверяет не только два обработчика маршрутов Express, но и взаимодействие в объектной модели документов (нажатие ссылки пользователем и произошедший в результате переход на страницу). Добавим ссылку на `views/home.handlebars`:

```
<p>Возникли вопросы? Перейдите на страницу
<a href="/about" data-test-id="about">О нас</a>!</p>
```

Вас, вероятно, заинтересовал атрибут `data-test-id`. Для выполнения тестирования нам нужно каким-то образом идентифицировать ссылку, чтобы мы смогли (виртуально) на ней щелкнуть. Для этого мы могли бы воспользоваться классом CSS, но я предпочитаю оставлять классы для настройки стилей, а для автоматизации использовать атрибуты данных. Можно также было бы выполнить поиск текста «О нас», но подобный поиск в DOM оказался бы ненадежным и затратным. Кроме того, имеет смысл сделать запрос по параметру `href`, но в таком случае этот тест было бы сложно заставить провалиться, а нам это нужно в учебных целях.

Мы можем пойти дальше, запустить наше приложение и нашими же неуклюжими руками проверить, что функциональность работает в соответствии с ожиданиями, прежде чем перейти к чему-нибудь более автоматизированному.

Прежде чем мы перейдем к установке Puppeteer и напишем интеграционный тест, нужно преобразовать наше приложение так, чтобы оно импортировалось как модуль (на данный момент оно может быть запущено только напрямую). Способ, которым это можно сделать в Node, не совсем очевиден: нужно заменить внизу `meadowlark.js` вызов `app.listen` на следующее:

```
if(require.main === module) {
  app.listen(port, () => {
    console.log( `Express запущен на http://localhost:${port}` +
      '; нажмите Ctrl+C для завершения.' )
  })
} else {
  module.exports = app
}
```

Я не привожу здесь описание технической стороны происходящего, поскольку это несколько утомительно, но если вас это интересует, то после тщательного прочтения документации по модулям Node все станет ясно (http://bit.ly/32BDO3H). Важно знать, что при запуске файла JavaScript с помощью Node напрямую `require.main`

будет соответствовать глобальному модулю, в противном случае он будет импортирован из другого модуля.

Теперь, когда мы со всем этим разобрались, можно установить Puppeteer, который, по сути, является управляемой версией Chrome в режиме headless. *Headless* означает, что браузер может быть запущен без фактического отображения интерфейса пользователя на экране. Для установки Puppeteer выполните:

```
npm install --save-dev puppeteer
```

Мы также установим маленькую утилиту, которая находит и открывает порт. Это нужно, чтобы при тестировании не возникали ошибки, связанные с тем, что наше приложение не может быть запущено на запрашиваемом нами порте.

```
npm install --save-dev portfinder
```

Теперь мы можем написать интеграционный тест, который выполняет следующее.

1. Запускает сервер нашего приложения на незанятом порте.
2. Запускает браузер Chrome в режиме headless и открывает в нем страницу.
3. Переходит на домашнюю страницу нашего приложения.
4. Находит ссылку с `data-test-id="about"` и щелкает на ней.
5. Ожидает перехода по ссылке.
6. Проверяет, что мы находимся на странице /about.

Создайте каталог `integration-tests` (вы можете назвать его как вам угодно) и в нем — файл `basic-navigation.test.js` (`ch05/integration-tests/basic-navigation.test.js` в прилагаемом репозитории):

```
const portfinder = require('portfinder')
const puppeteer = require('puppeteer')

const app = require('../meadowlark.js')

let server = null
let port = null

beforeEach(async () => {
  port = await portfinder.getPortPromise()
  server = app.listen(port)
})

afterEach(() => {
  server.close()
})

test('домашняя страница ссылается на страницу Описание', async () => {
  const browser = await puppeteer.launch()
```

```
  const page = await browser.newPage()
  await page.goto(`http://localhost:${port}`)
  await Promise.all([
    page.waitForNavigation(),
    page.click('[data-test-id="about"]'),
  ])
  expect(page.url()).toBe(`http://localhost:${port}/about`)
  await browser.close()
})
```

Мы используем функции-хелперы Jest `beforeEach` и `afterEach` для запуска нашего сервера до каждого теста и для его остановки после каждого теста. На данный момент у нас есть только один тест, поэтому в происходящем будет смысл, когда мы добавим больше тестов. Вместо этого можно воспользоваться `beforeAll` и `afterAll`, чтобы не производить запуск и остановку сервера для каждого теста. Данные действия способны ускорить выполнение ваших тестов, но произойдет это за счет того, что перед каждым тестом не будет производиться очистка среды. То есть если один из ваших тестов производит изменения, которые влияют на результаты последующих, то вы вносите сложные в поддержке зависимости.

В нашем текущем тесте мы используем API из Puppeteer, что дает нам множество функций формирования запросов к DOM. Следует отметить, что почти все здесь является асинхронным, и мы широко используем `await`, дабы упростить чтение и написание тестов (почти все вызовы Puppeteer API возвращают промисы). Если вы не знакомы с `await`, то я рекомендую вам статью Тамаса Пироса по адресу http://bit.ly/2rEXU0d. Мы обертываем вместе щелчок по ссылке для вызова `Promise.all` и навигацию, чтобы согласно документации Puppeteer устранить условие гонки (race condition).

Функциональность API Puppeteer значительно шире, чем я мог бы затронуть в данной книге. К счастью, по нему есть прекрасная документация (http://bit.ly/2KctokI).

Тестирование является жизненно важным, но далеко не единственным инструментом в вашем распоряжении, гарантирующим качество проекта. Линтинг поможет вам изначально предотвратить возникновение ошибок.

Линтинг

Хороший линтер подобен второй паре глаз: он замечает то, что проходит мимо нашего внимания. Первым JavaScript-линтером был созданный Дугласом Крокфордом JSLint. В 2011 г. Антон Ковалев создал ответвление JSLint — JSHint. Ковалев обнаружил, что JSLint стал чересчур «категоричным», и захотел создать лучше настраиваемый и разрабатываемый всем сообществом JavaScript-линтер. После JSHint появился ставший самым популярным ESLint (https://eslint.org/) от Николаса Закаса (он победил с большим отрывом в обзоре *State of JavaScript 2017*

(http://bit.ly/2Q7w32O)). ESLint не только универсальный, но и самый активно поддерживаемый линтер, и я рекомендую именно его ввиду гибкости настройки.

ESLint может быть установлен для каждого отдельного проекта или глобально. Чтобы ненароком что-нибудь не нарушить, я стараюсь избегать глобальных установок. Например, если я установил ESLint глобально и часто его обновляю, то из-за критических изменений контроль кода в старых проектах успешно выполняться не будет и мне придется делать лишнюю работу по обновлению проекта.

Чтобы установить ESLint в вашем проекте, наберите:

```
npm install --save-dev eslint
```

ESLint запрашивает файл конфигурации, содержащий сведения о том, какие правила нужно использовать. На выполнение этого действия с нуля может уйти много времени, но, к счастью, ESLint предоставляет соответствующую утилиту. Запустите следующее из корневого каталога вашего проекта:

```
./node_modules/.bin/eslint --init
```

Если ESLint установлен глобально, то можно использовать просто eslint --init. Громоздкий путь ./node_modules/.bin требуется только для запуска напрямую локально установленных утилит. Скоро мы увидим, что необходимость в этом отпадает при добавлении утилит в раздел scripts файла package.json, что рекомендуется для часто используемых утилит. В любом случае конфигурацию ESLint нужно создавать только один раз на проект.

ESLint будет задавать вопросы, отвечая на которые в большинстве случаев нужно выбирать значения по умолчанию, однако на некоторых моментах следует повысить внимание.

- *Какой тип модулей используется в вашем проекте?*

  Поскольку мы используем Node (в отличие от кода, который будет запускаться в браузере), вам нужно выбрать CommonJS (require/exports). В вашем проекте также может быть JavaScript на стороне клиента, в этом случае вы можете захотеть настроить линтинг отдельно. Проще всего будет создать два отдельных проекта, но возможно и наличие множества конфигураций ESLint в одном проекте. За более подробной информацией обращайтесь к документации ESLint (https://eslint.org/).

- *Какой фреймворк используется в вашем проекте?*

  Если вы не видите на этом месте Express (а я на данный момент не вижу), то выбирайте Ничего из этого.

- *Где запускается ваш код?*

  Выберите Node.

Когда ESLint настроен, нам нужен подходящий способ его запуска. Добавьте следующее в раздел `scripts` вашего `package.json`:

```
"lint": "eslint meadowlark.js lib"
```

Обратите внимание: нам нужно явно указать ESLint, какие файлы и папки мы хотим проверять. Это аргумент в пользу того, чтобы размещать весь ваш исходный код в одном каталоге (обычно `src`).

Теперь запустите следующую команду и держите себя в руках:

```
npm run lint
```

Вероятно, вы увидите множество ошибок — это то, что обычно происходит при первом запуске ESLint. Однако если вы вручную писали тесты Jest, здесь будут связанные с ним ложные ошибки следующего вида:

```
 3:1   ошибка   'test' не определен     no-undef
 5:25  ошибка   'jest' не определен     no-undef
 7:3   ошибка   'expect' не определен   no-undef
 8:3   ошибка   'expect' не определен   no-undef
11:1   ошибка   'test' не определен     no-undef
13:25  ошибка   'jest' не определен     no-undef
15:3   ошибка   'expect' не определен   no-undef
```

ESLint, что вполне разумно, не признает нераспознанные глобальные переменные. Jest вводит глобальные переменные: `test`, `describe`, `jest` и `expect`. К счастью, эта проблема легко решается. Откройте в корневом каталоге вашего проекта файл `.eslintrc.js` (это конфигурация ESLint). Добавьте следующее в раздел `env`:

```
"jest": true,
```

Если вы снова запустите `npm run lint`, то увидите гораздо меньше ошибок.

Так что же делать с остальными ошибками? Я не могу давать вам конкретные указания, а лишь делюсь знаниями, полученными опытным путем. Вообще, ошибки линтинга возникают по одной из трех причин.

- Настоящая проблема, которую нужно решить. Она не всегда очевидна, в этом случае вам нужно обратиться к документации ESLint по конкретной ошибке.
- Правило, с которым вы не согласны, и вам просто нужно его отключить. Бóльшая часть правил ESLint является делом вкуса. Скоро я покажу, как отключать правила.
- Вы согласны с правилом, но есть случаи, когда при определенных обстоятельствах оно невыполнимо либо его применение является затратным. В таких ситуациях вы можете отключить правила для определенных строк в файле, пример этого мы также рассмотрим.

Если вы набирали примеры вручную, то должны увидеть следующие ошибки:

```
/Users/ethan/wdne2e-companion/ch05/meadowlark.js
  27:5 ошибка Неожиданное выражение в консоли no-console

/Users/ethan/wdne2e-companion/ch05/lib/handlers.js
  10:39 ошибка 'next' определен, но никогда не используется no-unused-vars
```

ESLint жалуется по поводу логирования в консоли, поскольку это не всегда лучший способ представления вывода в приложении. Оно может быть зашумленным и непоследовательным, и, в зависимости от того, как вы его запускаете на выполнение, вывод может оказаться скрытым. Однако в нашем случае давайте будем считать, что это нас не беспокоит и мы хотим отключить это правило. Откройте файл `.eslintrc`, найдите раздел `rules` (если его нет, то создайте его вверху экспортируемого объекта) и добавьте следующее правило:

```
"rules": {
  "no-console": "off",
},
```

Теперь при повторном запуске `npm run lint` мы увидим, что ошибки больше нет! Следующее немного сложнее.

Откройте `lib/handlers.js` и рассмотрите такую строку:

```
exports.serverError = (err, req, res, next) => res.render('500')
```

ESLint прав; мы предоставляем `next` в качестве аргумента, но ничего с ним не делаем (мы также ничего не делаем с `err` и `req`, но, в зависимости от того, как JavaScript обрабатывает аргументы функций, нам нужно сюда *что-нибудь* поместить, чтобы мы могли добраться до `res`, который *используем*).

У вас может возникнуть соблазн просто-напросто удалить аргумент `next`. Вы можете подумать: «Что в этом плохого?» В этом случае не будет ошибок выполнения и линтер будет счастлив, но небольшой вред все же будет причинен: пользовательский обработчик ошибок перестанет работать! (Если хотите убедиться в этом, сгенерируйте исключение в одном из ваших маршрутов и попробуйте посетить его, а затем удалите аргумент `next` из обработчика `serverError`.)

Express действует несколько тоньше: он использует количество передаваемых действительных аргументов, чтобы распознать обработчик ошибок. Без данного аргумента `next` — используете вы его или нет — Express больше не распознает его как обработчик ошибок.

Команда разработчиков Express сделала обработчик ошибок, несомненно, «по-умному», но умный код часто может сбивать с толку, быть малопонятным и его работу легко нарушить. Как бы я ни любил Express, считаю, что именно здесь команда разработчиков пошла неверным путем: они должны были найти менее специфический и более явный способ указания обработчика ошибок.

Мы не можем изменить код нашего обработчика, нам нужен обработчик ошибок, но нам нравится это правило и нет желания его отключать. Один из вариантов — просто жить с этим, но ошибки будут накапливаться и станут источником постоянного раздражения. В итоге само наличие линтера потеряет смысл. К счастью, мы можем исправить ситуацию, отключив правило для одной строки. Отредактируйте `lib/handlers.js`, добавив следующее около вашего обработчика ошибок:

```
// Express распознает обработчик ошибок по его четырем аргументам,
// поэтому нам нужно отключить правило no-unused-vars в ESLint.
/* eslint-disable no-unused-vars */
exports.serverError = (err, req, res, next) => res.render('500')
/* eslint-enable no-unused-vars */
```

На первых порах линтинг способен вас разочаровать — может показаться, что линтер постоянно ставит подножки. Но вы не должны стесняться отключать правила, которые вам не подходят. В конце концов, по мере того, как вы научитесь не допускать распространенные ошибки, для обнаружения которых и предназначен линтинг, разочарования сойдут на нет.

Польза тестирования и линтинга неоспорима, но ни от одного инструмента не будет толку, если его не использовать! Может показаться странным, что приходится тратить время и силы на то, чтобы писать модульные тесты и настраивать линтинг, но я видел, каково это бывает, особенно в обстановке с растущим напряжением. К счастью, есть способ всегда помнить об этих инструментах — непрерывная интеграция.

Непрерывная интеграция

Вот еще одна чрезвычайно полезная концепция QA: непрерывная интеграция (continuous integration, CI). Она особенно важна при работе в команде, но даже если вы работаете в одиночку, она поможет навести порядок.

По сути, CI запускает все ваши тесты или их часть всякий раз, когда вы добавляете код в репозиторий (вы можете контролировать, к каким веткам он применяется). Если все тесты проходят успешно, ничего особенного не происходит (в зависимости от настроек CI вы можете, например, получить электронное письмо со словами: «Отлично сделано!»).

Если же имеются ошибки, последствия обычно более заметные. Опять же это зависит от настроек CI, но, как правило, вся команда получает электронное письмо, сообщающее, что вы «испортили сборку». Если ответственный за интеграцию в вашей команде — садист, то начальник тоже может оказаться в списке рассылки! Я знаю команды, в которых включали световую сигнализацию и сирену, когда кто-то портил сборку. А в одном особенно креативном офисе маленькая роботизированная пусковая установка стреляла мягкими пенопластовыми пульками в провинившегося разработчика! Это сильный стимул для запуска набора инструментов по обеспечению качества до фиксации изменений.

Описание установки и настройки сервера CI выходит за рамки данной книги, однако посвященная обеспечению качества глава была бы неполна без его упоминания.

В настоящее время наиболее распространенный сервер CI для проектов Node — Travis CI (https://travis-ci.org/). Travis CI — решение, располагающееся на внешнем сервере, что может оказаться весьма привлекательным, поскольку освобождает вас от необходимости настраивать собственный сервер CI. Оно предлагает великолепную поддержку интеграции для использующих GitHub. Другой вариант — CircleCI (https://circleci.com/).

Если вы трудитесь над проектом в одиночку, возможно, сервер CI не принесет вам большой пользы, но, если работаете в команде или над проектом с открытым исходным кодом, я крайне рекомендую задуматься о настройке CI.

Резюме

В этой главе мы охватили множество вопросов, и я считаю, что все это поможет сформировать реальные навыки работы в любом фреймворке разработки. Размер экосистемы JavaScript ошеломляет, и если вы новичок, то можете не знать, с чего начать. Надеюсь, что эта глава указала вам верное направление.

Теперь, когда у нас есть некоторый опыт применения этих инструментов, обратим внимание на основные сведения об объектах запроса и ответа Node и Express, которые объединяют все, что происходит в приложениях Express.

6 Объекты запроса и ответа

В этой главе мы изучим важные сведения об объектах запроса и ответа, с которых начинается и которыми заканчивается практически все происходящее в приложениях Express. Когда вы создаете веб-сервер с помощью Express, бóльшая часть ваших действий начинается с объекта запроса и заканчивается объектом ответа.

Эти два объекта возникли в Node и были расширены Express. Перед тем как углубиться в то, что они нам предлагают, немного разберемся, как клиент (обычно браузер) запрашивает страницу у сервера и как сервер эту страницу возвращает.

Составные части URL

Мы постоянно сталкиваемся с URL, но при этом редко задумываемся об их составных частях. Рассмотрим три URL и их составные части (рис. 6.1).

- *Протокол*. Протокол определяет, как будет передаваться запрос. Мы будем иметь дело исключительно с `http` и `https`. Среди других распространенных протоколов — `file` и `ftp`.
- *Хост*. Хост идентифицирует сервер. Сервер, находящийся на вашей машине (localhost) или в локальной сети, может определяться одним словом или числовым IP-адресом. В Интернете имя хоста будет заканчиваться доменом верхнего уровня (top-level domain, TLD), например `.com` или `.net`. Помимо этого, в нем могут быть *поддомены*, предшествующие имени хоста. Очень распространенный поддомен — `www`, хотя он может быть каким угодно. Поддомены необязательны.
- *Порт*. У каждого сервера есть набор пронумерованных портов. Некоторые номера портов — особенные, например 80 и 443. Если вы опустите указание

порта, то для HTTP предполагается порт 80, а для HTTPS — 443. Вообще, если вы не используете порты 80 или 443, лучше выбирать номера портов больше 1023[1]. Широко распространена практика использования легких для запоминания номеров, таких как 3000, 8080 и 8088. С заданным портом может ассоциироваться только один сервер, и, хотя вы вольны выбирать из большого количества номеров, может возникнуть необходимость сменить номер порта, если он широко используется.

- *Путь*. Обычно среди всех частей URL приложение в первую очередь интересует именно путь (можно принимать решения на основе протокола, имени хоста и порта, но это плохая практика). Путь следует использовать для однозначной идентификации страниц или других ресурсов в вашем приложении.
- *Строка запроса*. Строка запроса — необязательная совокупность пар «имя/значение». Строка запроса начинается со знака вопроса (`?`), а пары «имя/значение» разделяются амперсандами (`&`). Как имена, так и значения должны быть подвергнуты *URL-кодированию*. Для выполнения этой процедуры JavaScript предоставляет встроенную функцию `encodeURIComponent`. Например, пробелы будут заменены знаками плюс (`+`), другие специальные символы — цифровыми ссылками на символы. Иногда строку запроса называют строкой поиска или просто поиском.
- *Фрагмент*. Фрагмент (или *хеш*) вообще не передается на сервер, он предназначен исключительно для использования браузером. В некоторых одностраничных приложениях фрагменты используются для управления навигацией в приложении. Первоначально единственным назначением фрагментов было заставить браузер отображать определенную часть документа, отмеченную тегом-якорем (например, `<a id="chapter06">`).

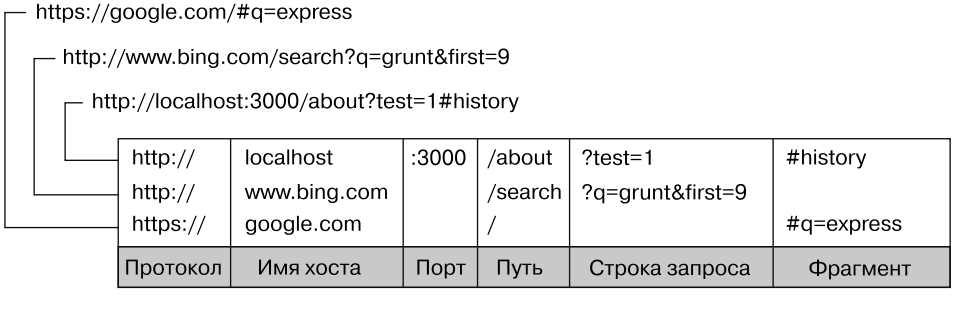

Рис. 6.1. URL и его составные части

[1] Порты 0–1023 — известные порты, зарезервированные для общих служб (https://ru.wikipedia.org/wiki/Список_портов_TCP_и_UDP).

Методы запросов HTTP

Протокол HTTP определяет набор *методов запроса* (часто называемых *глаголами HTTP*), используемых клиентом для связи с сервером. Несомненно, наиболее распространенные методы — `GET` и `POST`.

Когда вы набираете URL в браузере (или нажимаете на ссылку), браузер отправляет серверу HTTP-запрос `GET`. Наиболее важная информация, передаваемая серверу, — путь URL и строка запроса. Чтобы решить, как ответить, приложение использует сочетание метода, пути и строки запроса.

На сайте большинство ваших страниц будут отвечать на запрос `GET`. Запросы `POST` обычно предназначаются для отправки информации обратно серверу (при обработке форм, например). Запросы `POST` часто отвечают тем же HTML, что и в соответствующем запросе `GET`, после того как сервер обработает всю включенную в запрос информацию (например, данные формы). При связи с вашим сервером браузеры будут использовать исключительно методы `GET` и `POST`. Однако в Ajax-запросах вашего приложения могут использоваться любые глаголы HTTP. Например, метод HTTP под названием `DELETE` вполне может быть использован для вызова API, который производит удаление данных.

Работая с Node и Express, вы полностью ответственны за то, на обращения каких методов отвечаете. В Express вы, как правило, будете писать обработчики для конкретных методов.

Заголовки запроса

URL не единственная вещь, которая передается серверу, когда вы заходите на страницу. Ваш браузер отправляет массу невидимой информации при каждом посещении сайта. Я не говорю о персональной информации (хотя, если ваш браузер инфицирован вредоносным ПО, это может случиться). Браузер сообщает серверу, на каком языке он желает получить страницу (например, если вы скачиваете Chrome в Испании, он запросит испанскую версию посещаемых вами страниц, если она существует). Он также будет отправлять информацию об *«агенте пользователя»* (браузер, операционная система и аппаратное обеспечение) и другие фрагменты информации. Вся эта информация отправляется в виде заголовка запроса, что возможно благодаря свойству `headers` объекта запроса. Если вам интересно посмотреть, какую информацию отправляет ваш браузер, можете создать простейший маршрут Express для отображения этой информации (`ch06/00-echo-headers.js` в прилагаемом репозитории):

```
app.get('/headers', (req, res) => {
  res.type('text/plain')
  const headers = Object.entries(req.headers)
    .map(([key, value]) => `${key}: ${value}`)
  res.send(headers.join('\n'))
})
```

Заголовки ответа

Точно так же, как браузер отправляет скрытую информацию на сервер в виде заголовков запроса, так и сервер при ответе отправляет обратно информацию, отображаемую либо нет браузером. Обычно включаемая в заголовки ответа информация — это метаданные и сведения о сервере. Мы уже видели заголовок `Content-Type`, сообщающий браузеру, какой тип контента передается (HTML, изображение, CSS, JavaScript и т. п.). Обратите внимание, что браузер будет признавать заголовок `Content-Type` независимо от пути URL. Так что вы можете выдавать HTML по пути `/image.jpg` или изображение по пути `/text.html` (уважительных причин, чтобы так поступать, нет, просто важно понимать, что пути абстрактны и браузер использует `Content-Type` для определения того, как отображать контент). Помимо `Content-Type`, заголовки могут указывать, сжат ли ответ и какой вид кодировки используется. Заголовки ответа также могут содержать указание для браузера, долго ли он может кэшировать ресурс. Это важные соображения для оптимизации сайта, мы обсудим их подробнее в главе 17.

Заголовки ответа довольно часто содержат информацию о типе сервера, а иногда даже подробности об операционной системе. Недостаток возврата информации о сервере в том, что это дает хакерам возможность взломать защиту вашего сайта. Очень заботящиеся о своей безопасности серверы часто опускают эту информацию или посылают ложные сведения. Отключить заголовок Express по умолчанию `X-Powered-By` несложно (`ch06/01-disable-x-powered-by.js` в прилагаемом репозитории):

```
app.disable('x-powered-by')
```

Если вы хотите посмотреть на заголовки ответа, их можно найти в инструментах разработчика вашего браузера. Например, чтобы увидеть заголовки ответа в Chrome, сделайте следующее.

1. Откройте консоль JavaScript.
2. Щелкните на вкладке Network (Сеть).
3. Перезагрузите страницу.
4. Выберите HTML из списка запросов (он будет первым).
5. Щелкните на вкладке Headers (Заголовки) — и увидите все заголовки ответа.

Типы данных Интернета

Заголовок `Content-Type` исключительно важен: без него клиенту пришлось бы мучительно гадать, как отображать контент. Формат заголовка `Content-Type` — *тип данных Интернета*, состоящий из типа, подтипа и необязательных параметров. Например, `text/html; charset=UTF-8` определяет тип `text`, подтип `HTML` и схему кодирования символов UTF-8. Администрация адресного пространства Интернета

(Internet Assigned Numbers Authority, IANA) поддерживает официальный список типов данных Интернета (https://www.iana.org/assignments/media-types/media-types.xhtml). Вы часто можете слышать, как попеременно используются такие термины, как «тип содержимого», «тип данных Интернета» и «тип MIME». Многоцелевые расширения электронной почты в Интернете (Multipurpose Internet Mail Extensions, MIME) были предшественником типов данных Интернета и в основном являются их эквивалентом.

Тело запроса

В дополнение к заголовкам у запроса может быть *тело* (точно так же, как телом ответа является фактически возвращаемое содержимое). У обыкновенных запросов `GET` тело отсутствует, но у запросов `POST` оно, как правило, есть. Наиболее распространенный тип данных для тел запросов `POST` — `application/x-www-form-urlencoded`, представляющий собой закодированные пары «имя/значение», разделенные амперсандами (в сущности, тот же формат, что и у строки запроса). Если `POST` должен поддерживать загрузку файлов на сервер, используется более сложный тип данных — `multipart/form-data`. Наконец, запросы AJAX могут использовать для тела тип данных `application/json`. Больше о теле запроса мы узнаем в главе 8.

Объект запроса

Объект запроса (передается как первый параметр обработчика запросов, это значит, что вы можете дать ему такое название, какое захотите: обычно это имя `req` или `request`) начинает свое существование в качестве экземпляра класса `http.IncomingMessage` и является одним из основных объектов Node. Express добавляет ему дополнительную функциональность. Взглянем на наиболее полезные свойства и методы объекта запроса (все эти методы добавлены Express, за исключением `req.headers` и `req.url`, ведущих свое начало из Node).

- `req.params` — массив, содержащий *именованные параметры маршрутизации*. Мы узнаем об этом больше в главе 14.
- `req.query` — объект, содержащий параметры строки запроса (иногда их называют `GET`-параметрами) в виде пар «имя/значение».
- `req.body` — объект, содержащий параметры `POST`. Такое название он носит потому, что `POST`-параметры передаются в теле запроса, а не в URL, как параметры строки запроса. Чтобы получить доступ к `req.body`, вам понадобится промежуточное ПО, умеющее интерпретировать содержимое тела, о чем мы узнаем в главе 10.

- `req.route` — информация о текущем совпавшем маршруте. Полезна, главным образом, для отладки маршрутизации.
- `req.cookies/req.signedCookies` — объекты, содержащие значения cookies, передаваемые от клиента. См. главу 9.
- `req.headers` — заголовки запроса, полученные от клиента. Это объект, ключами которого являются названия заголовков, а значениями — значения заголовков. Следует отметить, что такое сопоставление связано с положенным в основу объектом `http.IncomingMessage`, поэтому вы не найдете этого в документации Express.
- `req.accepts(types)` — удобный метод для принятия решения о том, должен ли клиент принимать данный тип или типы (необязательный параметр `types` может быть одиночным типом MIME, например `application/json`, разделенным запятыми списком или массивом). Этот метод обычно интересен тем, кто пишет публичные API; он предполагает, что браузеры по умолчанию всегда принимают HTML.
- `req.ip` — IP-адрес клиента.
- `req.path` — путь запроса (без протокола, хоста, порта или строки запроса).
- `req.hostname` — удобный метод, возвращающий переданное клиентом имя хоста. Эта информация может быть поддельной и не должна использоваться из соображений безопасности.
- `req.xhr` — удобное свойство, возвращающее `true`, если запрос выполнен с помощью Ajax.
- `req.protocol` — протокол, использованный при совершении данного запроса (в нашем случае это будет или `http`, или `https`).
- `req.secure` — удобное свойство, возвращающее `true`, если соединение безопасное. Эквивалентно `req.protocol==='https'`.
- `req.url/req.originalUrl` — небольшая неточность в наименовании — эти свойства возвращают путь и строку запроса (они не включают протокол, хост или порт). `req.url` может быть переписан для нужд внутренней маршрутизации, но `req.originalUrl` разработан так, чтобы всегда хранить исходный путь и строку запроса.

Объект ответа

Объект ответа (передается как второй параметр обработчика запросов, это значит, что вы можете дать ему такое название, какое вам удобно: обычно это имя `res`, `resp` или `response`) начинает свое существование в качестве экземпляра класса `http.ServerResponse` и является одним из основных объектов Node. Express добавляет ему дополнительную функциональность. Взглянем на наиболее полезные свойства и методы объекта ответа (все эти методы добавлены Express).

- `res.status(code)` — устанавливает код состояния HTTP. По умолчанию в Express код состояния — 200 («OK»), так что вам придется использовать этот метод для возвращения состояния 404 («Не найдено»), 500 («Ошибка сервера») или любого другого кода состояния, который хотите применить. Для перенаправлений (коды состояния 301, 302, 303 и 307) предпочтительнее использовать метод `redirect`. Обратите внимание: `res.status` возвращает объект ответа, что означает наличие возможности соединять вызовы в цепочки: `res.status(404).send('Not found')`.
- `res.set(name, value)` — устанавливает заголовок ответа. Вряд ли в обычных условиях вы будете делать это вручную. Вы также можете установить множество заголовков одновременно, передав в качестве аргумента единичный объект, ключами которого являются имена заголовков, а значениями — значения заголовков.
- `res.cookie(name, value, [options])`, `res.clearCookie(name, [options])` — устанавливает или очищает cookies, которые будут храниться на клиенте. Для этого требуется поддержка промежуточного ПО, см. главу 9.
- `res.redirect([status], url)` — выполняет перенаправление браузера. Код перенаправления по умолчанию — 302 («Найдено»). В целом вам лучше минимизировать перенаправления, за исключением случая окончательного перемещения страницы, когда следует использовать код 301 («Перемещено навсегда»).
- `res.send(body)` — отправляет ответ клиенту. По умолчанию в Express используется тип содержимого `text/html`, так что, если вы хотите изменить его на `text/plain`, необходимо вызвать `res.type('text/plain')` перед вызовом `res.send`. Если тело — объект или массив, ответ будет отправлен в виде JSON (с установленным соответствующим типом содержимого), но, если вы хотите отправить JSON, я рекомендую делать это явным образом путем вызова `res.json`.
- `res.json(json)` — отправляет JSON клиенту.
- `res.jsonp(json)` — отправляет JSONP клиенту.
- `res.end()` — завершает соединение без отправки ответа. Чтобы узнать больше о различиях между `res.send`, `res.json` и `res.end`, смотрите статью Тамаса Пироса «Сравнение res.json(), res.send() и res.end() в Express» по адресу https://blog.fullstacktraining.com/res-json-vs-res-send-vs-res-end-in-express/.
- `res.type(type)` — удобный метод для установки заголовка `Content-Type`. Практически эквивалентен `res.set('Content-Type', type)`, за исключением того, что он также будет пытаться установить соответствие расширений файлов типам данных Интернета, если вы укажете строку без косой черты. Например, `Content-Type` в случае `res.type('txt')` будет `text/plain`. Есть области применения, где эта функциональность может быть полезной (например, автоматическая раздача различных мультимедийных файлов), но в целом вам лучше

избегать этого, предпочитая явным образом устанавливать правильный тип данных Интернета.

- `res.format(object)` — этот метод позволяет вам отправлять различное содержимое в зависимости от заголовка `Accept` запроса. Он в основном используется в разных API, и мы подробнее обсудим это в главе 15. Вот очень простой пример: `res.format({'text/plain': 'Привет!', 'text/html': '<b>Привет!</b>'})`.
- `res.attachment([filename]), res.download(path, [filename], [callback])` — оба этих метода устанавливают заголовок ответа `Content-Disposition` в значение `attachment`; это указывает браузеру, что необходимо загружать содержимое вместо его отображения. Вы можете задать `filename` в качестве подсказки браузеру. С помощью `res.download` можно задать файл для скачивания, в то время как `res.attachment` просто устанавливает заголовок и вам нужно будет отправить контент клиенту.
- `res.sendFile(path, [options], [callback])` — этот метод читает файл, заданный параметром `path`, и отправляет его содержимое клиенту. Он редко оказывается нужным — проще использовать промежуточное ПО `static` и разместить файлы, которые вы хотите сделать доступными клиенту, в каталоге `public`. Однако, если есть желание выдать другой ресурс с того же URL в зависимости от какого-либо условия, этот метод может оказаться полезным.
- `res.links(links)` — задает заголовок ответа `Links`. Это узкоспециализированный заголовок, редко используемый в большинстве приложений.
- `res.locals, res.render(view, [locals], callback)`. `res.locals` — объект, содержащий контекст *по умолчанию* для рендеринга представлений. `res.render` отображает представление, используя указанный в настройках шаблонизатор (не путайте параметр `locals` в `res.render` с `res.locals`: он перекрывает контекст в `res.locals`, но непереопределенный контекст по-прежнему будет доступен). Обратите внимание на то, что `res.render` по умолчанию будет приводить к коду состояния ответа 200; используйте `res.status` для указания других кодов состояния. Рендеринг представлений будет детальнее рассмотрен в главе 7.

Получение более подробной информации

Из-за прототипного наследования в JavaScript иной раз непросто понять, с чем вы имеете дело. Node предоставляет вам объекты, расширяемые Express, причем добавляемые вами пакеты также могут их расширять. Точное выявление того, что вам доступно, иногда может оказаться непростым делом. В целом я рекомендовал бы действовать наоборот: если вам нужна какая-то функциональность, сначала проверьте документацию по API Express (http://expressjs.com/api.html). API Express достаточно полный, и, вероятно, вы найдете там то, что ищете.

Если потребуется какая-то недокументированная информация, возможно, придется заглянуть в исходный код Express (https://github.com/expressjs/express). Я советую сделать это! Вы увидите, что все не так страшно, как кажется. Вот краткая инструкция по поиску в исходных текстах Express.

- `lib/application.js`. Главный интерфейс Express. Именно сюда следует заглядывать, если вы хотите понять схему подключения промежуточного ПО или подробности рендеринга представлений.
- `lib/express.js`. Относительно небольшой файл, предоставляющий функцию `createApplication` (при экспорте этого файла по умолчанию), которая создает экземпляр приложения Express.
- `lib/request.js`. Расширяет объект Node `http.IncomingMessage` для обеспечения устойчивости к ошибкам объекта запроса. Именно здесь следует искать информацию о свойствах и методах объекта запроса.
- `lib/response.js`. Расширяет объект Node `http.ServerResponse` для обеспечения возможностей объекта ответа. Именно здесь следует искать информацию о свойствах и методах объекта ответа.
- `lib/router/route.js`. Обеспечивает базовую поддержку маршрутизации. Хотя маршрутизация — важнейшая вещь для вашего приложения, длина этого файла — менее 230 строк. Вы увидите, что он весьма прост и изящен.

По мере углубления в исходный код Express вы, вероятно, захотите обратиться к документации по Node (https://nodejs.org/en/docs/), особенно к разделу о модуле HTTP.

Разбиваем на части

В этом разделе я сделал обзор объектов запроса и ответа — самого важного в приложениях Express. Однако вполне вероятно, что большую часть времени вы будете использовать лишь малую долю этой функциональности. Так что классифицируем функциональность по тому, насколько часто вы будете ее применять.

Рендеринг контента

При рендеринге контента чаще всего вы будете использовать `res.render`, который отображает представления в макетах, обеспечивая наилучшие характеристики. Иногда возникают ситуации, когда нужно по-быстрому написать страницу для тестирования, поэтому, если нужна просто тестовая страница, можете использовать `res.send`. Вы можете применить `req.query` для получения значений строки запроса, `req.session` — для получения значений сеансовых переменных или `req.cookie/req.signedCookies` — для получения cookies. При-

меры 6.1–6.8 демонстрируют наиболее распространенные задачи рендеринга контента.

Пример 6.1. Стандартное использование (ch06/02-basic-rendering.js)

```
// стандартное использование
app.get('/about', (req, res) => {
  res.render('about')
})
```

Пример 6.2. Отличные от 200 коды ответа (ch06/03-different-response-codes.js)

```
app.get('/error', (req, res) => {
  res.status(500)
  res.render('error')
})

// или в одну строку ...

app.get('/error', (req, res) => res.status(500).render('error'))
```

Пример 6.3. Передача контекста представлению, включая строку запроса, cookies и значения сеансовых переменных (ch06/04-view-with-content.js)

```
app.get('/greeting', (req, res) => {
  res.render('greeting', {
    message: 'Приветствую, уважаемый программист!',
    style: req.query.style,
    userid: req.cookies.userid,
    username: req.session.username
  })
})
```

Пример 6.4. Рендеринг представления без макета (ch06/05-view-without-layout.js)

```
// у следующего макета нет файла макета, так что
// views/no-layout.handlebars должен включать весь
// необходимый HTML
app.get('/no-layout', (req, res) =>
  res.render('no-layout', { layout: null })
)
```

Пример 6.5. Рендеринг представления с пользовательским макетом (ch06/06-custom-layout.js)

```
// будет использоваться файл макета
// views/layouts/custom.handlebars
app.get('/custom-layout', (req, res) =>
  res.render('custom-layout', { layout: 'custom' })
)
```

Пример 6.6. Рендеринг неформатированного текстового вывода (ch06/07-plaintext-output.js)

```
app.get('/text', (req, res) => {
  res.type('text/plain')
  res.send('это тест')
})
```

Пример 6.7. Добавление обработчика ошибок (ch06/08-error-handler.js)

```
// Это должно находиться ПОСЛЕ всех ваших маршрутов.
// Обратите внимание на то, что, даже если вам
// не нужна функция "next",
// она должна быть включена, чтобы Express
// распознал это как обработчик ошибок.
app.use((err, req, res, next) => {
  console.error('** ОШИБКА СЕРВЕРА: ' + err.message)
  res.status(500).render('08-error',
    { message: "Не стоило это нажимать!" })
})
```

Пример 6.8. Добавление обработчика кода состояния 404 (ch06/09-custom-404.js)

```
// Это должно находиться ПОСЛЕ всех ваших маршрутов
app.use((req, res) =>
  res.status(404).render('404')
)
```

Обработка форм

При обработке форм информация из них обычно находится в `req.body` (иногда в `req.query`). Вы можете использовать `req.xhr`, чтобы выяснить, был ли это Ajax-запрос или запрос браузера (подробнее мы рассмотрим это в главе 8) (примеры 6.9 и 6.10). Для приведенных ниже примеров нужно подключить промежуточное ПО для парсинга тела запроса:

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }))
```

Больше информации о парсере тела запроса — в главе 8.

Пример 6.9. Стандартная обработка формы (ch06/10-basic-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  console.log(`Получен контакт от ${req.body.name} <${req.body.email}>`)
  res.redirect(303, '10-thank-you')
})
```

Пример 6.10. Более устойчивая к ошибкам обработка формы (ch06/11-more-robust-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  try {
    // Здесь мы попытаемся сохранить контакт в базе данных
    // или воспользуемся другим способом хранения...
    // На данный момент мы просто сымитируем ошибку.
    if(req.body.simulateError) throw new Error("ошибка при сохранении
контакта!")
    console.log(`Получен контакт от ${req.body.name} <${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/thank-you'),
      'application/json': () => res.json({ success: true }),
    })
  } catch(err) {
    // Здесь мы будем обрабатывать все ошибки при сохранении
    console.error(`Ошибка при обработке контакта от ${req.body.name} ` +
      `<${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/contact-error'),
      'application/json': () => res.status(500).json({
      error: 'ошибка при сохранении информации о контакте' }),
    })
  }
})
```

Предоставление API

Когда вы предоставляете API, то аналогично обработке форм параметры обычно находятся в `req.query`, хотя можно также использовать `req.body`. Отличие случая с API в том, что вместо HTML вы будете возвращать JSON, XML или даже неформатированный текст и часто применять менее распространенные методы HTTP, такие как `PUT`, `POST` и `DELETE`. Предоставление API будет рассмотрено в главе 15. Примеры 6.11 и 6.12 используют следующий массив продуктов, который при обычных условиях будет извлекаться из базы данных:

```
var tours = [
  { id: 0, name: 'Худ-Ривер', price: 99.99 },
  { id: 1, name: 'Орегон Коуст', price: 149.95 },
];
```

Термин «конечная точка» (endpoint) часто используется для описания отдельной функции API.

Пример 6.11. Простая конечная точка GET, возвращающая только JSON (ch06/12-api.get.js)

```
app.get('/api/tours', (req, res) => res.json(tours))
```

Пример 6.12 использует метод `res.format` из Express для выполнения ответа в соответствии с предпочтениями клиента.

Пример 6.12. Простая конечная точка GET, возвращающая JSON, XML или текст (ch06/13-api-json-xml-text.js)

```
app.get('/api/tours', (req, res) => {
  const toursXml = '<?xml version="1.0"?><tours>' +
    tours.map(p =>
      `<tour price="${p.price}" id="${p.id}">${p.name}</tour>`
    ).join('') + '</tours>'
  const toursText = tours.map(p =>
      `${p.id}: ${p.name} (${p.price})`
    ).join('\n')
  res.format({
    'application/json': () => res.json(tours),
    'application/xml': () => res.type('application/xml').send(toursXml),
    'text/xml': () => res.type('text/xml').send(toursXml),
    'text/plain': () => res.type('text/plain').send(toursXml),
  })
})
```

В примере 6.13 конечная точка `PUT` меняет продукт и возвращает JSON. Параметры передаются в теле запроса (`":id"` в строке маршрута приказывает Express добавить свойство `id` к `req.params`).

Пример 6.13. Конечная точка PUT для изменения (ch06/14-api-put.js)

```
app.put('/api/tour/:id', (req, res) => {
  const p = tours.find(p => p.id === parseInt(req.params.id))
  if(!p) return res.status(404).json({ error: 'No such tour exists' })
  if(req.body.name) p.name = req.body.name
  if(req.body.price) p.price = req.body.price
  res.json({ success: true })
})
```

И наконец, пример 6.14 демонстрирует конечную точку `DELETE`.

Пример 6.14. Конечная точка DELETE для удаления (ch06/15-api-del.js)

```
app.delete('/api/tour/:id', (req, res) => {
  const idx = tours.findIndex(tour => tour.id === parseInt(req.params.id))
  if(idx < 0) return res.json({ error: 'Такого тура не существует.' })
  tours.splice(idx, 1)
  res.json({ success: true })
})
```

Резюме

Надеюсь, что небольшие примеры из этой главы помогли составить представление о широко используемой в приложениях Express функциональности. Эти примеры задумывались как краткое руководство, к которому впоследствии можно возвращаться.

В следующей главе мы подробно рассмотрим шаблонизацию, использованную в примерах рендеринга в этой главе.

7 Шаблонизация с помощью Handlebars

В этой главе будет рассмотрена *шаблонизация* — методика проектирования и форматирования контента, отображаемого пользователю. Шаблонизацию можно рассматривать как эволюционировавшее стандартное письмо: «Уважаемый [Имя Отчество]! С сожалением сообщаем Вам, что [Устаревшая технология] больше никем не используется, но шаблонизация живет и здравствует!» Чтобы отправить это письмо нескольким людям, вам нужно лишь изменить [Имя Отчество] и [Устаревшую технологию].

Этот процесс замены значений полей иногда называют интерполяцией, что в данном контексте является всего-навсего красивым названием предоставления недостающей информации.

Хотя такие фреймворки для разработки клиентской части, как React, Angular и Vue, быстро вытеснили шаблонизацию на стороне сервера, последняя все еще применяется, например, для создания сообщений электронной почты в формате HTML. К тому же как Angular, так и Vue используют для написания HTML подход, схожий с шаблонизацией, так что полученные вами знания о шаблонизации на стороне сервера применимы к этим фреймворкам для разработки клиентской части.

Если вы раньше работали с PHP, то можете удивиться, из-за чего весь этот ажиотаж: PHP — один из первых языков, который можно назвать языком шаблонизации. Практически все основные языки, подходящие для веб-разработки, включают тот или иной вид поддержки шаблонизации. Но в последнее время ситуация изменилась: *шаблонизатор* обычно не привязан к языку.

Так как же выглядит шаблонизация? Начнем с того, что шаблонизация замещает, и рассмотрим наиболее прямой и очевидный путь генерации одного языка из другого (конкретнее, мы будем генерировать HTML с помощью JavaScript):

```
document.write('<h1>Пожалуйста, не делайте так</h1>')
document.write('<p><span class="code">document.write</span> капризен,\n')
document.write('и его следует избегать в любом случае.</p>')
document.write('<p>Сегодняшняя дата: ' + new Date() + '.</p>')
```

Возможно, единственная причина, по которой это кажется очевидным, заключается в том, что именно так всегда учили программированию:

```
10 PRINT "Hello world!"
```

В императивных языках мы привыкли говорить: «Сделай это, затем то, а потом что-то еще». В некоторых случаях этот подход отлично работает. В нем нет ничего плохого, если у вас 500 строк JavaScript для выполнения сложного вычисления, результатом которого является одно число, и каждый шаг зависит от предыдущего. Но что, если дела обстоят с точностью до наоборот? У вас 500 строк HTML и три строки JavaScript. Имеет ли смысл писать `document.write` 500 раз? Отнюдь.

На самом деле все сводится к тому, что переключать контекст проблематично. Если вы пишете много кода JavaScript, вплетать в него HTML неудобно, это приводит к путанице. Обратный способ не так уж плох: мы привыкли писать JavaScript в блоках `<script>`, но надеюсь, что вы видите разницу: здесь все-таки есть переключение контекста и вы или пишете HTML, или в блоке `<script>` пишете JavaScript. Использование же JavaScript для генерации HTML чревато проблемами.

- Вам придется постоянно думать о том, какие символы необходимо экранировать и как это сделать.
- Использование JavaScript для генерации HTML, который, в свою очередь, сам содержит JavaScript, быстро сведет вас с ума.
- Вы обычно лишаетесь приятной подсветки синтаксиса и прочих удобных возможностей, отражающих специфику языка, которыми обладает ваш редактор.
- Становится гораздо сложнее заметить плохо сформированный HTML.
- Трудно визуально анализировать код.
- Другим людям может быть сложнее понимать ваш код.

Шаблонизация решает проблему, позволяя писать на целевом языке и при этом обеспечивая возможность вставлять динамические данные. Взгляните на предыдущий пример, переписанный в виде шаблона Mustache:

```
<h1>Намного лучше</h1>
<p>Никаких <span class="code">document.write</span> здесь!</p>
<p>Сегодняшняя дата {{today}}.</p>
```

Все, что нам осталось сделать, — обеспечить значение для `{{today}}`. Это и есть основа языков шаблонизации.

Нет абсолютных правил, кроме этого[1]

Я не говорю, что вам *никогда* не следует писать HTML в JavaScript, а только то, что этого следует избегать при любой возможности. В частности, это приемлемо в коде клиентской части, особенно если вы используете надежный фреймворк для ее разработки. Например, следующее вызвало бы с моей стороны немного комментариев:

```
document.querySelector('#error').innerHTML =
  'Случилось что-то <b>очень плохое!</b>'
```

Однако я бы намекнул, что настало время применить шаблон, если код постепенно видоизменится до вот такого:

```
document.querySelector('#error').innerHTML =
  '<div class="error"><h3>Error</h3>' +
  '<p>Случилось что-то <b><a href="/error-detail/' + errorNumber
  +'"> очень плохое.</a></b> ' +
  '<a href="/try-again">Попробуйте снова<a>, или ' +
  '<a href="/contact">обратитесь в техподдержку</a>.</p></div>')
```

Дело в том, что я посоветовал бы вам очень тщательно подумать, где нужно провести границу между HTML в строках и использованием шаблонов. Лично я допустил бы перекос в сторону шаблонов и избегал генерации HTML с помощью JavaScript во всех случаях, за исключением простейших.

Выбор шаблонизатора

Во вселенной Node у вас есть выбор из множества шаблонизаторов. Как же подобрать подходящий? Это непростой вопрос, существенно зависящий от того, что именно вам нужно. Вот некоторые критерии, которые следует принять во внимание.

- *Производительность.* Несомненно, вы хотите, чтобы шаблонизатор работал как можно быстрее, но нельзя, чтобы он замедлял работу вашего сайта.
- *Клиент, сервер или и то и другое?* Большинство шаблонизаторов доступны на стороне как сервера, так и клиента. Если вам необходимо использовать шаблоны и тут и там (и вы будете это делать), рекомендую выбрать шаблонизатор, производительность которого одинакова в обоих случаях.
- *Абстракция.* Вам хочется чего-то знакомого (вроде обычного HTML с добавлением фигурных скобок) или вы тайно ненавидите HTML и предпочли бы что-то без всех этих угловых скобок? Шаблонизация (особенно на стороне сервера) дает вам возможность выбора.

[1] Перефразируя моего друга Пола Инмана.

Это лишь некоторые из наиболее важных критериев выбора шаблонизатора. На данный момент альтернативные варианты шаблонизаторов уже достаточно зрелые, так что вы вряд ли ошибетесь с выбором.

Express позволяет использовать любой шаблонизатор, какой пожелаете, так что, если Handlebars вас не устраивает, вы без проблем сможете его заменить. Если хотите узнать, какие существуют варианты, можете попробовать вот эту забавную и удобную утилиту выбора шаблонизатора: http://bit.ly/2CExtK0 (для нее давно не было обновлений, но она все еще актуальна).

Прежде чем перейти к обсуждению Handlebars, взглянем на исключительно абстрактный шаблонизатор.

Pug: другой подход

В то время как большинство шаблонизаторов используют подход с сильной ориентацией на HTML, Pug выделяется тем, что абстрагирует вас от его подробностей. Стоит также отметить, что Pug — детище Ти Джея Головайчука, того самого, кто подарил нам Express. Неудивительно, что Pug отлично интегрируется с Express. Подход, используемый Pug, весьма благороден: в его основе лежит утверждение, что HTML — слишком перегруженный деталями и трудоемкий для написания вручную язык. Посмотрим, как выглядит шаблон Pug вместе с результирующим HTML (изначально взят с домашней страницы Pug (https://pugjs.org/) и слегка изменен для соответствия книжному формату):

```
doctype html
html(lang="ru")
  head
    title= pageTitle
    script.
      if (foo) {
        bar(1 + 5)
      }
  body

    h1 Pug
    #container
      if youAreUsingJadePug
        p Браво!
      else
        p Просто сделайте это!
      p.
        Pug — сжатый и простой язык
        шаблонизации с сильным
        акцентом на производительности
        и многочисленных возможностях.
```

```
<!DOCTYPE html>
<html lang="ru">
<head>
<title>Демонстрация Pug</title>
<script>
  if (foo) {
    bar(1 + 5)
  }
</script>
<body>
<h1>Pug</h1>
<div id="container">

<p>Браво!</p>


<p>
  Pug — сжатый и простой язык
  шаблонизации с сильным
  акцентом на производительности
  и многочисленных возможностях.
</p>
</body>
</html>
```

Pug, безусловно, сводит к минимуму набор на клавиатуре (больше никаких угловых скобок и закрывающих тегов). Вместо этого он опирается на структурированное расположение текста и определенные общепринятые правила, облегчая выражение идей. У Pug есть еще одно достоинство: теоретически, когда меняется сам язык HTML, вы можете просто перенастроить Pug на новую версию HTML, что обеспечит вашему контенту «защиту от будущего».

Как бы я ни восхищался философией Pug и изяществом его выполнения, я обнаружил, что не хочу абстрагироваться от подробностей HTML. Этот язык лежит в основе всего, что я как веб-разработчик делаю, и, если цена этого — износ клавиш с угловыми скобками на клавиатуре, так тому и быть. Множество разработчиков клиентской части, с которыми я обсуждал этот вопрос, думают аналогично, так что мир, возможно, пока еще просто не готов к Pug.

На этом мы расстаемся с Pug, больше его в этой книге вы не увидите. Однако, если вам нравится абстракция, у вас определенно не будет проблем при использовании Pug с Express. Существует масса ресурсов, которые вам в этом помогут.

Основы Handlebars

Handlebars — расширение Mustache, еще одного распространенного шаблонизатора. Я рекомендую Handlebars из-за его удобной интеграции с JavaScript (как в клиентской, так и в серверной части) и знакомого синтаксиса. На мой взгляд, он обеспечивает все правильные компромиссы, и именно на нем мы сосредоточимся в данной книге. Стоит заметить, что концепции, которые будут обсуждаться, легко применимы и к другим шаблонизаторам, так что вы будете вполне готовы к тому, чтобы попробовать другие шаблонизаторы, если Handlebars не придется по душе.

Ключ к пониманию шаблонизации — понимание концепции *контекста*. Когда вы отображаете шаблон, вы передаете шаблонизатору объект, называемый *объектом контекста*, что и обеспечивает работу подстановок.

Например, если мой объект контекста:

```
{ name: 'Лютик' },
```

а шаблон:

```
<p>Добро пожаловать, {{name}}!</p>,
```

то `{{name}}` будет заменено на `Лютик`. Что же произойдет, если вы хотите передать HTML-код шаблону? Например, если вместо этого наш контекст будет:

```
{ name: '<b>Лютик</b>' },
```

использование предыдущего шаблона приведет к выдаче `<p>Добро пожаловать, <b>Лютик<b></p>`, что, вероятно, совсем не то, чего вы хотели. Для ре-

шения этой проблемы просто используйте три фигурные скобки вместо двух: `{{{name}}}`.

Несмотря на то что мы решили не формировать HTML-код посредством JavaScript, возможность отключать экранирование HTML-кода с помощью тройных фигурных скобок имеет интересные варианты применения. Например, если вы создаете систему управления контентом (content management system, CMS) с помощью WYSIWYG-редактора (what you see is what you get — «что видишь, то и получишь»), вам, вероятно, захочется иметь возможность передавать HTML-код вашим представлениям. Кроме того, возможность отображать свойства из контекста без экранирования HTML-кода важна для макетов и секций, о чем мы скоро узнаем.

На рис. 7.1 мы видим, как механизм Handlebars использует контекст (представленный овалом) в сочетании с шаблоном для рендеринга HTML-кода.

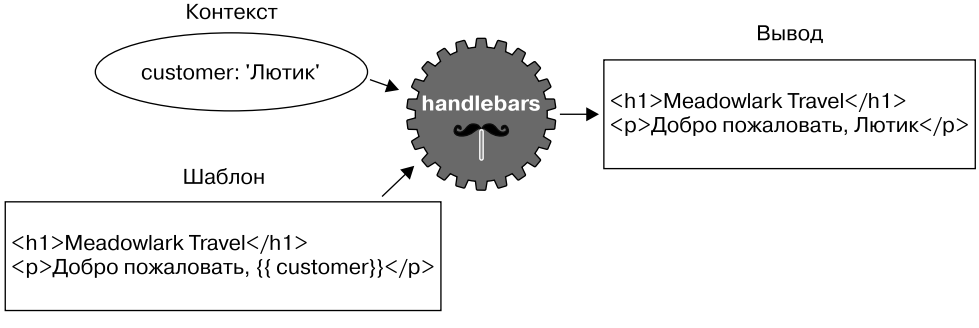

Рис. 7.1. Рендеринг HTML-кода с помощью Handlebars

Комментарии

Комментарии в Handlebars выглядят вот так: `{{! Здесь находится комментарий }}`. Важно понимать различие между комментариями в Handlebars и комментариями HTML. Рассмотрим следующий шаблон:

```
{{! Очень секретный комментарий }}
<!-- Не очень секретный комментарий -->
```

Если это серверный шаблон, очень секретный комментарий никогда не будет отправлен браузеру, в то время как не очень секретный комментарий будет виден, если пользователь заглянет в исходный код HTML. Вам следует предпочесть комментарии Handlebars всем остальным, раскрывающим подробности реализации или что-то еще, что вам не хотелось бы выставлять напоказ.

Блоки

Все усложняется, когда мы начинаем рассматривать *блоки*. Блоки обеспечивают управление последовательностью выполнения, условное выполнение и расширяемость. Рассмотрим следующий контекстный объект:

```
{
  currency: {
    name: 'Доллары США',
    abbrev: 'USD',
  },
  tours: [
    { name: 'Худ-Ривер', price: '$99.95' },
    { name: 'Орегон Коуст', price: '$159.95' },
  ],
  specialsUrl: '/january-specials',
  currencies: [ 'USD', 'GBP', 'BTC' ],
}
```

Теперь взглянем на шаблон, которому мы можем этот контекст передать:

```
<ul>
  {{#each tours}}
    {{! Я в новом блоке... и контекст изменился }}
    <li>
      {{name}} - {{price}}
      {{#if ../currencies}}
         ({{../../currency.abbrev}})
      {{/if}}
    </li>
  {{/each}}
</ul>
{{#unless currencies}}
  <p>Все цены в {{currency.name}}.</p>
{{/unless}}
{{#if specialsUrl}}
  {{! Я в новом блоке... но контекст вроде бы не изменился }}
  <p>Проверьте наши <a href="{{specialsUrl}}">специальные предложения!</p>
{{else}}
  <p>Просьба чаще пересматривать наши специальные предложения.</p>
{{/if}}
<p>
  {{#each currencies}}
    <a href="#" class="currency">{{.}}</a>
  {{else}}
    К сожалению, в настоящее время мы принимаем только {{currency.name}}.
  {{/each}}
</p>
```

В этом шаблоне происходит многое, так что разобьем его на составные части. Он начинается со вспомогательного элемента `each`, обеспечивающего итерацию по массиву. Важно понимать, что между `{{#each tours}}` и `{{/each tours}}` меняется контекст. На первом проходе он меняется на `{ name: 'Худ-Ривер', price: '$99.95' }`, а на втором проходе — на `{ name: 'Орегон Коуст', price, '$159.95' }`. Таким образом, внутри этого блока мы можем ссылаться на `{{name}}` и `{{price}}`. Однако если мы хотим обратиться к объекту `currency`, нам придется использовать `../`, чтобы получить доступ к *родительскому* контексту.

Если свойство контекста само по себе является объектом, мы можем обратиться к его свойствам как обычно, через точку, например: `{{currency.name}}`.

Как у `if`, так и у `each` может быть (необязательный) блок `else` (в случае `each` блок `else` будет выполняться при отсутствии элементов в массиве). Мы также использовали вспомогательный элемент `unless`, являющийся, по существу, противоположностью вспомогательного элемента `if`: он выполняется только в том случае, когда аргумент ложен.

Последняя вещь, которую хотелось бы отметить относительно этого шаблона: использование `{{.}}` в блоке `{{#each currencies}}`. `{{.}}` ссылается на текущий контекст; в данном случае текущий контекст — просто строка в массиве, которую мы хотим вывести на экран.

Обращение к текущему контексту через одиночную точку имеет и другое применение: оно позволяет различать вспомогательные функции (хелперы) (которые мы вскоре изучим) и свойства текущего контекста. Например, если у вас есть хелпер foo и свойство foo в текущем контексте, {{foo}} ссылается на хелпер, а {{./foo}} — на свойство.

Серверные шаблоны

Серверные шаблоны дают возможность отобразить HTML-код до его отправки клиенту. В отличие от шаблонизации на стороне клиента, где шаблоны доступны любопытному пользователю, знающему, как смотреть исходный код HTML, ваши пользователи никогда не увидят серверные шаблоны или объекты контекста, используемые для генерации окончательного HTML-кода.

Помимо скрытия подробностей реализации, серверные шаблоны поддерживают *кэширование* шаблонов, играющее важную роль в обеспечении производительности. Шаблонизатор кэширует скомпилированные шаблоны (перекомпилирует и кэширует заново только в случае изменения самого шаблона), что повышает производительность шаблонизированных представлений. По умолчанию кэширование представлений отключено в режиме разработки и активно в эксплуатационном

режиме. Явным образом активировать кэширование представлений, если захотите, можно следующим образом:

```
app.set('view cache', true)
```

Непосредственно «из коробки» Express поддерживает Pug, EJS и JSHTML. Мы уже обсуждали Pug, и я не стану рекомендовать использовать EJS или JSHTML (на мой взгляд, они оба недостаточно развиты в плане синтаксиса). Итак, нужно добавить пакет Node, который обеспечит поддержку Handlebars для Express:

```
npm install express-handlebars
```

Затем привяжем его к Express (`ch07/00/meadowlark.js` в прилагаемом репозитории):

```
const expressHandlebars = require('express-handlebars')
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
})
app.set('view engine', 'handlebars')
```

Пакет express-handlebars предполагает, что расширение шаблонов Handlebars будет .handlebars. Я уже привык к нему, но, если это расширение слишком длинное для вас, можете изменить его на распространенное .hbs при создании экземпляра express-handlebars: app.engine('handlebars','.hbs' })).

Представления и макеты

Представление обычно означает отдельную страницу вашего сайта (хотя оно может означать и загружаемую с помощью Ajax часть страницы, электронное письмо или что-то еще в том же роде). По умолчанию Express ищет представления в подкаталоге `views`. *Макет* — особая разновидность представления, по сути, шаблон для шаблонов. Макеты важны, поскольку у большинства (если не у всех) страниц вашего сайта будут практически одинаковые макеты. Например, у них должны быть элементы `<html>` и `<title>`, они обычно загружают одни и те же файлы CSS и т. д. Вряд ли вы захотите дублировать этот код на каждой странице, вот тут-то и пригодятся макеты. Взглянем на основу макета:

```
<!doctype>
<html>
  <head>
    <title>Meadowlark Travel</title>
    <link rel="stylesheet" href="/css/main.css">
  </head>
  <body>
```

```
    {{{body}}}
  </body>
</html>
```

Обратите внимание на текст внутри тега `<body>`: `{{{body}}}`. Благодаря ему шаблонизатор знает, где отобразить содержимое вашего представления. Важно использовать три фигурные скобки вместо двух, поскольку представление почти наверняка будет содержать HTML и мы не хотим, чтобы Handlebars пытался его экранировать. Замечу, что нет ограничений относительно размещения поля `{{{body}}}`. Например, если вы создаете адаптивный макет в Bootstrap, то, вероятно, поместите представление внутри контейнера `<div>`. Многие стандартные элементы страницы, такие как шапка (header) и подвал (footer), также чаще всего находятся в макетах, а не в представлениях. Вот пример:

```
<!-- ... -->
<body>
  <div class="container">
    <header>
      <div class="container">
        <h1>Meadowlark Travel</h1>
        <img src="/img/logo.png" alt="Логотип Meadowlark Travel">
      </div>
    </header>
    <div class="container">
      {{{body}}}
    </div>
    <footer>&copy; 2019 Meadowlark Travel</footer>
  </div>
</body>
```

На рис. 7.2 мы увидим, как шаблонизатор объединяет представление, макет и контекст. Самая важная вещь, которую проясняет эта схема, — порядок выполнения действий. Сначала, до макета, *отображается представление*. На первый взгляд это может показаться нелогичным: раз представление отображаться *внутри* макета, не должен ли макет отображается первым? Несмотря на то что технически это вполне возможно выполнить, есть определенные преимущества в обратном порядке действий. В частности, это позволяет самому представлению адаптировать макет, что вполне может пригодиться, в чем мы убедимся при обсуждении *секций* далее в этой главе.

Благодаря определенному порядку выполнения действий вы можете передать в представление свойство body, и оно будет правильно отображаться в представлении. Однако при рендеринге макета значение body будет перезаписано отображаемым представлением.

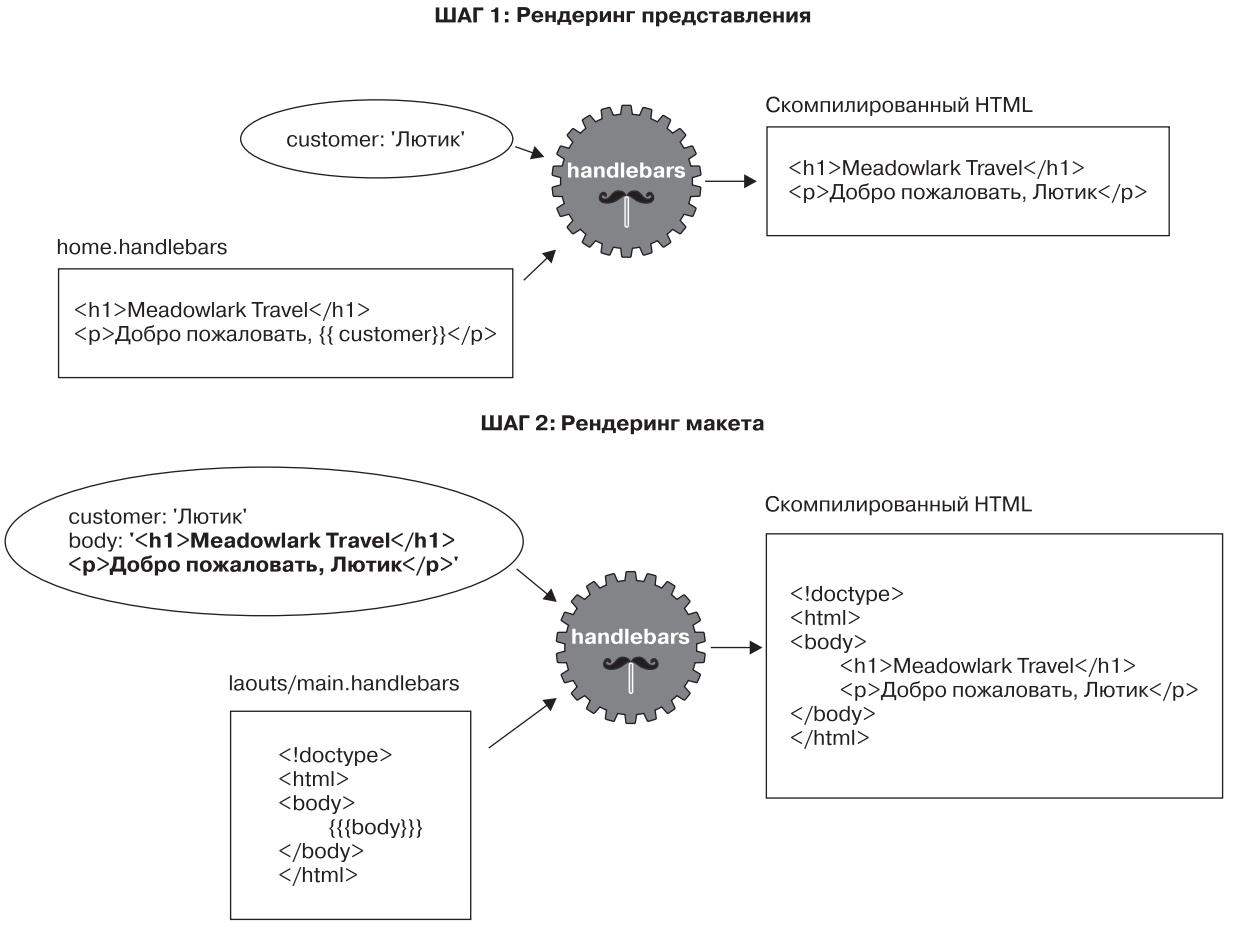

Рис. 7.2. Рендеринг представления с помощью макета

Использование (или неиспользование) макетов в Express

По всей вероятности, большинство ваших страниц (если не все) станут использовать один и тот же макет, так что нет смысла указывать макет каждый раз при визуализации страницы. Как вы видели, при создании шаблонизатора мы задали имя макета по умолчанию:

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
})
```

По умолчанию Express ищет представления в подкаталоге `views`, а макеты — в подкаталоге `layouts`. Так что, если у вас есть представление `views/foo.handlebars`, можете его визуализировать следующим образом:

```
app.get('/foo', (req, res) => res.render('foo'))
```

В качестве макета при этом будет использоваться `views/layouts/main.handlebars`. Если вообще не хотите использовать макет (а значит, вам придется держать весь шаблонный код в представлении), можете указать в контекстном объекте `layout: null`:

```
app.get('/foo', (req, res) => res.render('foo', { layout: null }))
```

А если хотите использовать другой шаблон, можете указать имя шаблона:

```
app.get('/foo', (req, res) => res.render('foo', { layout: 'microsite' }))
```

При этом будет визуализироваться представление с макетом `views/layouts/microsite.handlebars`.

Помните, что чем больше у вас шаблонов, тем проще должен быть ваш макет HTML. В то же время это может оправдаться при наличии у вас страниц, сформированных по иному макету. В этом вопросе вам придется найти правильное соотношение для своих проектов.

Секции

В основе одного метода, который я позаимствовал у замечательного шаблонизатора *Razor*, созданного компанией Microsoft, лежит идея *секций*. Макеты отлично работают, если каждое из представлений спокойно помещается в отдельный элемент макета, но что произойдет, если представлению понадобится внедриться в другие части макета? Распространенный пример — представление, которому требуется добавить что-либо в элемент `<head>` или вставить `<script>`, который из соображений производительности иногда оказывается последним элементом макета.

Ни у Handlebars, ни у `express-handlebars` нет встроенного способа сделать это. К счастью, хелперы Handlebars сильно облегчают эту задачу. При создании объекта

Handlebars мы добавим хелпер `section` (`ch07/01/meadowlark.js` в прилагаемом репозитории):

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
  helpers: {
    section: function(name, options) {
      if(!this._sections) this._sections = {}
      this._sections[name] = options.fn(this)
      return null
    },
  },
}))
```

Теперь мы можем использовать в представлении хелпер `section`. Добавим представление (`views/jquery-test.handlebars`) для включения чего-либо в `<head>` и скрипт:

```
{{#section 'head'}}
  <!-- Мы хотим, чтобы Google игнорировал эту страницу -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Тестовая страница</h1>
<p>Тестируем что-нибудь связанное со скриптом.</p>

{{#section 'scripts'}}
  <script>
    document.querySelector('body')
      .insertAdjacentHTML('beforeEnd', '<small>(scripting works!)</small>')
  </script>
{{/section}}
```

А теперь можем разместить в нашем макете секции так же, как размещаем `{{{body}}}`:

```
{{#section 'head'}}
  <!-- Мы хотим, чтобы Google игнорировал эту страницу -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Тестовая страница</h1>
<p>Тестируем что-нибудь связанное со скриптом.</p>

{{#section 'scripts'}}
  <script>
    const div = document.createElement('div')
    div.appendChild(document.createTextNode('(scripting works!)'))
    document.querySelector('body').appendChild(div)
  </script>
{{/section}}
```

Частичные шаблоны

Очень часто вам будут встречаться компоненты, которые вы захотите повторно использовать на различных страницах (в кругах разработчиков клиентской части их иногда называют *виджетами*). Один из способов добиться этого с помощью шаблонов — использовать *частичные шаблоны* (они не отображают целое представление или целую страницу, отсюда и название). Допустим, нам нужен компонент Current Weather, отображающий текущие погодные условия в Портленде, Бенде и Манзаните[1]. Мы хотим сделать этот компонент пригодным для повторного использования, чтобы иметь возможность поместить его на любую нужную нам страницу. Поэтому будем использовать частичный шаблон. Сначала создадим файл частичного шаблона `views/partials/weather.handlebars`:

```
<div class="weatherWidget">
  {{#each partials.weatherContext}}
    <div class="location">
      <h3>{{location.name}}</h3>
      <a href="{{location.forecastUrl}}">
        <img src="{{iconUrl}}" alt="{{weather}}">
        {{weather}}, {{temp}}
      </a>
    </div>
  {{/each}}
  <small>Источник:
    <a href="https://www.weather.gov/documentation/services-web-api">
    National Weather Service</a></small>
</div>
```

Обратите внимание, что имя области видимости контекста начинается с `partials.weatherContext`. Поскольку нам нужна возможность использовать частичный шаблон на любой странице, передавать контекст для каждого представления неудобно, так что вместо этого мы используем объект `res.locals`, доступный для каждого представления. Но, поскольку нам не хотелось бы пересекаться с контекстом, задаваемым отдельными представлениями, мы поместили весь частичный контекст в объект `partials`.

express-handlebars позволяет передавать частичные шаблоны как часть контекста. Например, если вы добавите partials.foo = "Шаблон!" в ваш контекст, вы сможете визуализировать этот частичный шаблон с помощью {{> foo}}. При таком использовании любые файлы представлений .handlebars будут переопределены, и именно поэтому выше мы использовали partials.weatherContext вместо partials.weather, которое переопределило бы views/partials/weather.handlebars.

[1] Города в штате Орегон. — *Примеч. пер.*

В главе 19 мы увидим, как можно извлечь информацию о текущей погоде из общедоступного API National Weather Service. Пока же просто будем использовать фиктивные данные, возвращаемые функцией, которую назовем `getWeatherData`.

Мы хотим, чтобы в этом примере данные о погоде были доступны любому представлению, и наилучший механизм в этом случае — промежуточное ПО (больше о нем мы поговорим в главе 10). Промежуточное ПО будет передавать данные о погоде в объект `res.locals.partials`, благодаря которому они будут доступны для частичных шаблонов.

Мы поместим промежуточное ПО в отдельный файл `lib/middleware/weather.js` (`ch07/01/lib/middleware/weather.js` в прилагаемом репозитории) для упрощения его тестирования.

```
const getWeatherData = () => Promise.resolve([
  {
    location: {
      name: 'Портленд',
      coordinates: { lat: 45.5154586, lng: -122.6793461 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/112,103/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,40?size=medium',
    weather: 'Вероятны грозы и ливни',
    temp: '59 F',
  },
  {
    location: {
      name: 'Бенд',
      coordinates: { lat: 44.0581728, lng: -121.3153096 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PDT/34,40/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra_sct,50?size=medium',
    weather: 'Местами грозы и ливни',
    temp: '51 F',
  },
  {
    location: {
      name: 'Манзанита',
      coordinates: { lat: 45.7184398, lng: -123.9351354 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/73,120/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,90?size=medium',
    weather: 'Грозы и ливни',
    temp: '55 F',
  },
])

const weatherMiddleware = async (req, res, next) => {
  if(!res.locals.partials) res.locals.partials = {}
```

```
  res.locals.partials.weatherContext = await getWeatherData()
  next()
}

module.exports = weatherMiddleware
```

Теперь, когда все настроено, нам осталось сделать лишь одно — использовать частичный шаблон в представлении. Например, чтобы поместить наш виджет на домашнюю страницу, отредактируем `views/home.handlebars`:

```
<h2>Домашняя страница</h2>
{{> weather}}
```

Синтаксис вида `{{> partial_name}}` описывает то, как вы включаете частичный шаблон в представление: `express-handlebars` будет знать, что нужно искать представление `partial_name.handlebars` (или `weather.handlebars` в нашем примере) в `views/partials`.

express-handlebars поддерживает подкаталоги, так что, если у вас много частичных шаблонов, можете их упорядочить. Например, если у вас есть частичные шаблоны социальных медиа, можете разместить их в каталоге views/partials/social и включить с помощью {{> social/facebook}}, {{> social/twitter}} и т. д.

Совершенствование шаблонов

В основе вашего сайта лежат шаблоны. Хорошая структура шаблонов сократит время разработки, повысит единообразие сайта и уменьшит число мест, в которых могут скрываться различные странности макетов. Но, чтобы получить этот эффект, вам придется провести некоторое время за тщательным проектированием шаблонов. Определить, сколько требуется шаблонов, — настоящее искусство; в общем случае чем меньше, тем лучше, однако существует точка снижения эффективности, зависящая от однородности страниц. Шаблоны также являются первой линией защиты от проблем межбраузерной совместимости и правильности HTML. Они должны быть спроектированы с любовью и сопровождаться кем-то, кто хорошо разбирается в разработке клиентской части. Отличное место для начала этой деятельности, особенно если вы новичок, HTML5 Boilerplate (http://html5boilerplate.com/). В предыдущих примерах мы использовали минимальный шаблон HTML5, чтобы соответствовать книжному формату, но для нашего настоящего проекта будем применять HTML5 Boilerplate.

Другое популярное место, с которого стоит начать работу с шаблонами, — темы производства сторонних разработчиков. На таких сайтах, как Themeforest (http://bit.ly/34Tdkfj) и WrapBootstrap (https://wrapbootstrap.com/), можно найти сотни готовых для использования тем, которые можно применять в качестве отправной точки

для своих шаблонов. Использование тем сторонних разработчиков начинается с переименования основного файла (обычно `index.html`) в `main.handlebars` (или дайте файлу макета другое название) и помещения всех ресурсов (CSS, JavaScript, изображений) в используемый вами для статических файлов каталог `public`. Затем нужно отредактировать файл шаблона и решить, куда вы хотите вставить выражение `{{{body}}}`.

В зависимости от элементов шаблона вы можете захотеть переместить некоторые из них в частичные шаблоны. Отличный пример этого — «герой» (большой баннер, разработанный специально для привлечения внимания пользователя). Если «герой» должен отображаться на каждой странице (вероятно, не лучший вариант), вы, наверное, оставите его в файле шаблона. Если он показывается только на одной странице (обычно домашней), то его лучше поместить только в это представление. Если он показывается на нескольких (но не на всех) страницах, то можно рассмотреть возможность его помещения в частичный шаблон. Выбор за вами, и в этом состоит искусство создания оригинального привлекательного сайта.

Резюме

Мы увидели, как шаблонизация может облегчить написание, чтение и сопровождение вашего кода. Благодаря шаблонам больше не нужно мучительно собирать HTML-код из строк JavaScript: мы можем писать его в любимом редакторе и делать его динамическим с помощью лаконичного и легкого для чтения языка шаблонизации.

Теперь, когда мы узнали, как форматировать отображаемый контент, обратим внимание на то, как с помощью HTML-форм передать данные в нашу систему.

8 Обработка форм

Наиболее распространенным способом сбора информации у ваших пользователей является применение *HTML-форм*. Независимо от того, позволяете ли вы браузеру отправить форму обычным путем, используете Ajax или сложные элементы управления на клиентской стороне, базовым механизмом по-прежнему остается HTML-форма. В этой главе мы обсудим различные способы обработки форм, проверки в них данных, а также загрузки файлов.

Отправка данных с клиентской стороны на сервер

У вас есть два способа отправить данные с клиентской стороны на сервер: через строку запроса и через тело запроса. Как правило, если вы используете строку запроса, вы создаете GET-запрос, а если используете тело — то POST-запрос. (Протокол HTTP не препятствует тому, чтобы вы делали это каким-то другим путем, но в этом нет смысла и здесь лучше придерживаться стандартной практики.)

Есть распространенный стереотип, что POST-запрос безопасен, а GET — нет. В действительности они оба безопасны, если вы используете протокол HTTPS, если же вы его не применяете, ни один из них безопасным не является. Если вы не используете HTTPS, злоумышленник может просмотреть данные тела POST-запроса так же легко, как увидеть строку GET-запроса. Однако, если вы применяете GET-запросы, ваши пользователи будут видеть все введенные ими данные (включая скрытые поля) в строке запроса, что неопрятно и некрасиво. Кроме того, браузеры часто накладывают ограничения на длину строки (размер тела ничем не ограничен). По этой причине я обычно рекомендую использовать POST для отправки форм.

HTML-формы

Эта книга фокусируется на серверной стороне, но очень важно иметь определенное базовое понимание того, как создаются HTML-формы. Вот простой пример:

```
<form action="/process" method="POST" >
  <input type="hidden" name="hush" val="Скрытое, но не секретное!" >
  <div>
    <label for="fieldColor" >Ваш любимый цвет: </label>
    <input type="text" id="fieldColor" name="color" >
  </div>
  <div>
    <button type="submit" >Отправить</button>
  </div>
</form>
```

Обратите внимание, что метод определяется явным образом как `POST` в теге `<form>`; если вы этого не делаете, по умолчанию используется `GET`. Атрибут `action` (действие) определяет URL, который получит форму, когда она будет отправлена. Если вы опускаете это поле, она будет отправлена по тому же URL, с которого была загружена. Я рекомендую всегда заполнять атрибут `action`, даже если вы используете Ajax, это поможет предотвратить потерю данных (для получения дополнительной информации см. главу 22).

С точки зрения сервера важный атрибут в полях `<input>` — это атрибут `name`: именно по нему сервер определяет поле. Очень важно понимать, что имя атрибута отличается от атрибута `id`, который следует использовать только для стилизации и обеспечения должного функционирования клиентской стороны (на сервер он не передается).

Обратите внимание на скрытое поле: оно не будет отображаться браузером. Однако вам не следует использовать его для секретной или уязвимой информации: все, что нужно сделать пользователю, — это открыть исходный код страницы, и он увидит там это скрытое поле.

HTML не ограничивает вас в создании большого количества форм на одной и той же странице (это было неудачное ограничение на некоторых ранних серверных фреймворках — ASP, я на тебя смотрю!). Я рекомендую хранить ваши формы в логически согласованном виде: форма должна содержать все поля, которые вы хотели бы отправить за раз (опциональные и пустые поля вполне допустимы), и ничего из того, чего вы отправлять не хотите. Если у вас на странице предполагаются два разных действия, используйте две разные формы. Вы можете использовать одну большую форму и определять, какое действие предпринимать, основываясь на том, какую кнопку нажал пользователь. Но, во-первых, это большая морока, а во-вторых, это не всегда удобно для пользователей с ограниченными физическими возможностями из-за способа отображения форм браузерами для таких людей.

Когда пользователь отправляет форму, выполняется запрос на URL `/process` и значения полей отправляются на сервер в теле запроса.

Кодирование

Когда ваша форма отправлена (либо браузером, либо посредством Ajax), она должна быть каким-нибудь способом закодирована. Если вы не указываете метод кодирования явно, по умолчанию используется `application/x-wwwform-urlencoded` (это попросту длинное название для типа данных «закодированный URL» — URL encoded). Это основное и простое в использовании кодирование, которое поддерживается стандартной версией Express.

Если вам нужно загружать файлы на сервер, задача становится куда более сложной. Нет простого пути отправки файлов с использованием кодирования URL, так что вам придется использовать тип кодирования `multipart/form-data`, который не обрабатывается Express напрямую.

Различные подходы к обработке форм

Если вы не используете Ajax, то можете только отправить форму посредством браузера, что вызовет перезагрузку страницы. Однако как страница перезагрузится, остается на ваше усмотрение. Есть два фактора, которые вы рассматриваете при обработке формы: каким путем обрабатывать форму и какой ответ будет отправлен браузеру.

Если ваша форма использует `method="POST"` (что рекомендуется), то обычно применяется один и тот же путь как для отображения формы, так и для ее обработки: их можно различить, поскольку сначала использовался GET-запрос, а затем POST-запрос. При использовании этого подхода вы можете пропустить атрибут `action` в этой форме.

Второй вариант — применение отдельного пути для обработки формы. Например, если ваша страница контактов использует путь `/contact`, вы можете задействовать путь `/process-contact` для обработки формы (посредством указания `action="/process-contact"`). Если вы применяете этот подход, то можете отправить форму посредством GET-запроса (я не рекомендую это делать — она без всякой необходимости показывает поля вашей формы в URL). Использование отдельной конечной точки для отправки форм может быть предпочтительным, если у вас есть много URL, которые применяют тот же способ отправки (например, у вас может быть поле для регистрации через электронную почту на многих страницах сайта).

Какой бы путь ни использовался в форме, вы должны решить, какой ответ отправлять браузеру. Вот варианты, которые можно задействовать.

- *Прямой ответ HTML*. После обработки формы вы можете отправить HTML (например, представление) напрямую браузеру. Однако использовать этот способ не рекомендуется, так как при этом будет выводиться предупреждение, если пользователь попробует перезагрузить страницу. Кроме того, он препятствует использованию закладок и кнопки Back (Назад).

- *Переадресация 302*. Такой подход применяют многие, однако это неправильное использование кода ответа 302 (Found (Найдено)). В HTTP 1.1 добавлен код ответа 303 (See Other (Смотреть другое)), использовать который предпочтительнее. Применяйте переадресацию 303, если у вас нет необходимости поддерживать браузеры, выпущенные до 1996 г.
- *Переадресация 303*. Код ответа 303 (See Other (Смотреть другое)) был добавлен в HTTP 1.1 для того, чтобы переадресация 302 не использовалась не по назначению. Спецификация HTTP отдельно указывает, что браузер должен использовать GET-запрос вслед за переадресацией 303 независимо от первоначального метода. Это рекомендуемый метод для ответа на запрос отправки формы.

Поскольку рекомендуется, чтобы вы отвечали на отправленную форму переадресацией 303, возникает вопрос: куда именно осуществлять переадресацию? Ответ таков: по вашему усмотрению. Вот наиболее распространенные подходы.

- *Перенаправление в случае успеха или неуспеха на специальные страницы*. Этот метод предполагает, что вы создадите специальные URL для соответствующих сообщений об успешном или неуспешном результате. Например, если пользователь подписывается на вашу рассылку по электронной почте, но при регистрации произошла ошибка в базе данных, вы можете перенаправлять на адрес `/error/database`. Если электронный адрес пользователя неправильный, можете перенаправлять на `/error/invalid-email`, а если все прошло успешно — на `/promo-email/thank-you`. Одним из преимуществ данного метода является то, что это очень удобно для аналитики: количество посещений вашей страницы `/promo-email/thank-you` должно практически совпадать с количеством людей, подписавшихся на ваши рекламные рассылки. Кроме того, этот метод довольно просто реализовать. Однако и у него есть свои недостатки. Он предполагает, что вы должны выделять URL для каждого из таких случаев, что означает необходимость дизайна, наполнения и поддержки каждой из этих страниц. Второй недостаток состоит в том, что с точки зрения удобства пользователя этот подход может быть неоптимальным: хотя пользователям и нравится, когда их благодарят, но им необходимо возвращаться назад на ту страницу, где они были перед отправкой формы, или переходить туда, куда они хотят направиться дальше. Пока что мы будем использовать этот подход, а к уведомлениям перейдем в главе 9.
- *Перенаправление на исходную страницу с уведомлением*. Для небольших форм, разбросанных по всему сайту (например, для указания адреса электронной почты), лучше не менять ход навигации, чтобы пользователю было удобнее. Таким образом, обеспечьте возможность отправить электронный адрес, не уходя со страницы. Одним из способов, позволяющих сделать это, является, конечно же, Ajax, но если вы не хотите использовать Ajax (или желаете задействовать альтернативный механизм для большего удобства пользователя), то можете перенаправлять обратно на ту же страницу, с которой пользователь отправил

форму. Простейший способ сделать это — применение в форме скрытого поля, которое заполняется текущим URL. Поскольку вы хотите, чтобы пользователь получил какую-то обратную связь после отправки формы, используйте уведомление.

- *Перенаправление на новое место с уведомлением*. Как правило, большие формы занимают целую страницу и нет смысла оставаться на той же странице после отправки формы. В этом случае вы должны понять, куда пользователь наверняка хотел бы пойти, и перенаправить его туда. Например, если вы создаете интерфейс админки и у вас есть форма для создания нового турпакета, разумно было бы ожидать от пользователя, что после отправки формы он захочет перейти к собственному списку туристических пакетов. Однако вам все равно следовало бы использовать уведомление для того, чтобы сообщить пользователю о результате отправки формы.

Если вы используете Ajax, я рекомендую отдельный URL. Заманчиво начинать обработчики Ajax с префикса (например, `/ajax/enter`), однако я не одобряю данный подход — это привязывает детали реализации к URL. Кроме того, как мы увидим позже, обработчик Ajax должен уметь обрабатывать и формы, отправленные браузером.

Обработка форм посредством Express

Если вы используете GET-запрос для обработки формы, поля будут доступны как объект `req.query`. Например, если у вас есть поле ввода запросов HTML с атрибутом имени `email`, его значение будет отправлено обработчику как `req.query.email`. Этот подход настолько прост, что здесь больше и рассказывать нечего.

Если вы применяете POST (что я рекомендую), то должны использовать промежуточное ПО, чтобы интерпретировать URL-закодированное тело. Прежде всего установите пакет body-parser (`npm install body-parser`), затем добавьте его (`ch08/meadowlark.js` в прилагаемом репозитории):

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
```

Добавив `body-parser`, вы увидите, что `req.body` станет доступным для вас, таким образом, доступными станут и поля формы. Обратите внимание, что `req.body` не препятствует использованию строки запросов. Пойдем дальше и добавим к Meadowlark Travel форму, позволяющую пользователю подписаться на почтовую рассылку. Для демонстрации будем применять строку запросов, скрытое поле и видимые поля во `/views/newsletter-signup.handlebars`:

```
<h2>Подпишитесь на нашу рассылку для получения новостей и специальных
предложений! </h2>
<form class="form-horizontal" role="form"
      action="/newsletter-signup/process?form=newsletter" method="POST">
```

```
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label" >Имя</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
      id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label" >
    Электронный адрес</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
      id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-default" >
      Зарегистрироваться</button>
    </div>
  </div>
</form>
```

Обратите внимание, что мы используем стили Bootstrap и будем их применять дальше в этой книге. Если вы не знакомы с Bootstrap, обратитесь к документации Bootstrap (http://getbootstrap.com/).

Мы уже подключили `body-parser`, и теперь нам нужно добавить обработчики для страницы подписки на рассылки, функции обработки и страницы с благодарностью (`ch08/lib/handlers.js` в прилагаемом репозитории):

```
exports.newsletterSignup = (req, res) => {
  // Мы изучим CSRF позже... сейчас мы лишь
  // вводим фиктивное значение.
  res.render('newsletter', { csrf: 'Здесь находится токен CSRF' });
})
exports.newsletterSignupProcess = (req, res) => {
  console.log('Форма (из строки запроса): ' + req.query.form)
  console.log('Токен CSRF (из скрытого поля формы): ' + req.body._csrf)
  console.log('Имя (из видимого поля формы): ' + req.body.name)
  console.log('E-mail (из видимого поля формы): ' + req.body.email)
  res.redirect(303, '/newsletter-signup/thank-you')
}
exports.newsletterSignupThankYou = (req, res) =>
  res.render('newsletter-signup-thank-you')
```

Создайте файл `views/newsletter-signup-thank-you.handlebars`, если вы этого еще не сделали.

Затем мы привяжем обработчики к нашему приложению (`ch08/meadowlark.js` в прилагаемом репозитории):

```
app.get('/newsletter-signup', handlers.newsletterSignup)
app.post('/newsletter-signup/process', handlers.newsletterSignupProcess)
app.get('/newsletter-signup/thank-you', handlers.newsletterSignupThankYou)
```

Вот и все, что нужно было сделать. Обратите внимание, что в обработчике мы перенаправляем на представление Thank you. Мы могли бы передать представление сюда, но, если так сделать, поле URL в браузере посетителя останется `/process`, что может сбить с толку; перенаправление решает эту проблему.

Важно, чтобы в этом случае вы использовали перенаправление 303 (или 302), а не 301. Перенаправление 301 — постоянное, это значит, что ваш браузер может кэшировать место перенаправления. Если вы уже использовали перенаправление 301 и пробуете отправить форму еще раз, браузер может полностью проигнорировать обработчик /process и перейти напрямую на /thank-you, поскольку он полагает, что это перенаправление должно быть постоянным. В то же время перенаправление 303 говорит вашему браузеру: «Да, ваш запрос действителен и вы можете найти ответ здесь», и, таким образом, он не кэширует место перенаправления.

В большинстве фреймворков разработки клиентской части данные формы отправляются в формате JSON с помощью API `fetch`, который мы рассмотрим далее. Тем не менее понимание того, как браузеры обрабатывают отправку формы по умолчанию, будет плюсом, ведь вам будут встречаться реализованные таким образом формы.

Обратим внимание на отправку форм с fetch.

Отправка данных формы с помощью fetch

Применение API fetch для отправки данных формы, закодированных в формате JSON, — более современный подход, дающий больше контроля над взаимодействиями клиента и сервера и позволяющий реже обновлять страницы.

Поскольку мы больше не обращаемся к серверу с запросами по принципу «туда и обратно», нам не нужно беспокоиться о перенаправлениях и многочисленных пользовательских URL (у нас по-прежнему есть отдельный URL для обработки самой формы), поэтому мы помещаем все, что относится к подписке на рассылки, под одним URL с названием `/newsletter`.

Начнем с кода клиентской части. Содержимое самой HTML-формы изменять не нужно (поля и макет остаются прежними), но нам не требуется указывать `action`

или method, и мы обернем нашу форму в элемент-контейнер , который упростит отображение сообщения Спасибо:

```
<div id="newsletterSignupFormContainer">
  <form class="form-horizontal role="form" id="newsletterSignupForm">
    <!-- остальное содержимое формы не меняется... -->
  </form>
</div>
```

Затем мы напишем скрипт, который перехватывает событие отправки формы и отменяет его (с помощью Event#preventDefault), так что мы можем обработать форму самостоятельно (ch08/views/newsletter.handlebars в прилагаемом репозитории):

```
<script>
  document.getElementById('newsletterSignupForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const form = evt.target
      const body = JSON.stringify({
      _csrf: form.elements._csrf.value,
      name: form.elements.name.value,
      email: form.elements.email.value,
    })
    const headers = { 'Content-Type': 'application/json' }
    const container = document.getElementById('newsletterSignupFormContainer')
    fetch('/api/newsletter-signup', { method: 'post', body, headers })
    .then(resp => {
      if(resp.status < 200 || resp.status >= 300)
      throw new Error(`Запрос отклонен со статусом ${resp.status}`)
      return resp.json()
    })
    .then(json => {
      container.innerHTML = '<b>Спасибо, что подписались!</b>'
    })
    .catch(err => {
      container.innerHTML = `<b>Извините, возникли проблемы при подписке.` +
      `Пожалуйста, <a href="/newsletter">попробуйте еще раз</a>`
    })
  })
</script>
```

Прежде чем указать две наши конечные точки, убедимся, что в файле сервера мы добавим промежуточный обработчик для парсинга тела запроса в формате JSON.

```
app.use(bodyParser.json())
//...
app.get('/newsletter', handlers.newsletter)
app.post('/api/newsletter-signup', handlers.api.newsletterSignup)
```

Обратите внимание: мы добавляем конечную точку для обработки формы к URL, начинающемуся с api. Этот прием обычно применяется, чтобы провести

различие между пользовательскими (браузерными) конечными точками и конечными точками API, которые должны быть доступны с помощью `fetch`.

Теперь добавим эти конечные точки в файл `lib/handlers.js`:

```
exports.newsletter = (req, res) => {
  // Мы изучим CSRF позже... сейчас мы лишь
  // вводим фиктивное значение.
  res.render('newsletter', { csrf: 'Здесь находится токен CSRF' })
}
exports.api = {
  newsletterSignup: (req, res) => {
    console.log('Токен CSRF (из скрытого поля формы): ' + req.body._csrf)
    console.log('Имя (из видимого поля формы): ' + req.body.name)
    console.log('Email (из видимого поля формы): ' + req.body.email)
    res.send({ result: 'success' })
  },
}
```

В этом примере мы предполагаем, что все Ajax-запросы будут искать JSON, но нет требований, что Ajax должен использовать JSON для передачи информации (по сути, Ajax — это акроним, в котором X означает XML). Этот подход очень дружественен JavaScript, поскольку JavaScript — эксперт в обработке JSON. Если вы делаете конечные точки публично доступными или знаете, что ваши Ajax-запросы могут использовать что-то другое, отличное от JSON, вы должны вернуть соответствующий ответ исключительно на основе заголовка Accepts, который мы можем с легкостью получить посредством вспомогательного метода req.access (http://bit.ly/33Syx92). Если ответ основывается только на заголовке Accepts, вы также можете обратить внимание на res.format — удобный и вспомогательный метод, позволяющий высылать ответ сервера в зависимости от того, чего ожидает клиент. Если вы так делаете, то должны установить свойства dataType или Accepts, когда осуществляете Ajax-запросы с помощью JavaScript.

Загрузка файлов на сервер

Я уже упоминал, что отправка файлов таит в себе уйму сложностей. К счастью, есть несколько замечательных проектов, которые облегчают работу с загрузкой файлов.

Существует четыре популярные опции для обработки multipart/form-data: Busboy, Multiparty, Formidable и Multer. Я пользовался ими всеми, и все они хорошо себя показали, но я считаю, что Multiparty лучше поддерживается, поэтому здесь мы будем использовать ее.

Создадим форму загрузки файла на сервер для фотоконкурса Meadowlark Travel vacation (`views/contest/vacation-photo.handlebars`):

```
<h2>Конкурс отпускных фотографий</h2>

<form class="form-horizontal" role="form"
```

```
      enctype="multipart/form-data" method="POST"
      action="/contest/vacation-photo/{year}/{month}">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Имя</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
      id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2
        control-label">Адрес электронной почты</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
      id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldPhoto" class="col-sm-2 control-label">
    Фотография из отпуска</label>
    <div class="col-sm-4">
      <input type="file" class="form-control" required accept="image/*"
      id="fieldPhoto" name="photo">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">
      Отправить</button>
    </div>
  </div>
</form>
```

Обратите внимание, что мы указываем `enctype="multipart/form-data"` для разрешения загрузки файлов на сервер. Мы также ограничиваем типы файлов, которые могут быть загружены, посредством использования атрибута `accept` (опционально).

Теперь нужно создать обработчики маршрутов, но у нас возникло небольшое затруднение. Мы хотим сохранить возможность легко тестировать обработчики маршрутов, что сложно сделать из-за обработки multipart/form-data (так же, как и в случае использования ПО промежуточной обработки для различных типов кодирования тела запроса до того, как доберемся до обработчиков). Поскольку мы не хотим самостоятельно тестировать многокомпонентное декодирование форм (можем считать, что multiparty сделала это как надо), то оставляем наши обработчики «чистыми», передавая им уже обработанную информацию. Так как мы еще не знаем, как это выглядит, то начнем с промежуточного ПО для Express в `meadowlark.js`.

```
const multiparty = require('multiparty')

app.post('/contest/vacation-photo/:year/:month', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if(err) return res.status(500).send({ error: err.message })
    handlers.vacationPhotoContestProcess(req, res, fields, files)
  })
})
```

Мы используем метод `parse` из Multiparty для разбора данных запроса на файлы и поля данных. Он сохраняет файлы во временный каталог на сервере, и эта информация возвращается в массиве `files`.

Сейчас нам нужно передать дополнительную информацию нашему (тестируемому) обработчику маршрута: поля (которые, ввиду того что мы используем другой парсер тела запроса, не будут находиться в `req.body`, как в предыдущем примере) и сведения о полученных нами файлах. Теперь, когда мы знаем, как это выглядит, можем написать обработчик маршрута.

```
exports.vacationPhotoContestProcess = (req, res, fields, files) => {
  console.log('данные поля: ', fields)
  console.log('файлы: ', files)
  res.redirect(303, '/contest/vacation-photo-thank-you')
}
```

Год и месяц указаны как параметры путей, это вы изучите в главе 14. Продолжим — запустим это и изучим логи в консоли. Вы увидите, что поля форм будут показаны, как вы и предполагали, в виде объекта со свойствами, соответствующими именам ваших полей. Объект `files` содержит больше данных, но он относительно простой. Для каждого загруженного файла вы увидите свойства, описывающие его размер, путь, по которому он был загружен (обычно это случайное имя во временном каталоге), и первоначальное название, то есть название того файла, который был загружен пользователем на сервер (просто имя файла, а не полный путь из соображений безопасности и приватности).

Вы можете делать с этим файлом что захотите: можете сохранить его в базе данных, скопировать в постоянное место размещения либо загрузить в облачную систему хранения файлов. Помните, что, если вы рассчитываете на локальное хранилище, приложение нельзя будет должным образом расширить, поэтому данный вариант не лучший для облачного хранения данных. Мы вернемся к этому примеру в главе 13.

Загрузка файлов посредством fetch

К счастью, использовать fetch для загрузки файлов почти то же самое, что дать браузеру сделать это. Трудности при загрузке файлов связаны с кодировками, но этим у нас занимается промежуточное ПО.

Рассмотрим скрипт JavaScript для отправки содержимого формы с помощью fetch:

```
<script>
  document.getElementById('vacationPhotoContestForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const body = new FormData(evt.target)
      const container =
          document.getElementById('vacationPhotoContestFormContainer')
      const url = '/api/vacation-photo-contest/{{year}}/{{month}}'
      fetch(url, { method: 'post', body })
      .then(resp => {
        if(resp.status < 200 || resp.status >= 300)
        throw new Error(`Запрос отклонен со статусом ${resp.status}`)
        return resp.json()
      })
      .then(json => {
        container.innerHTML = '<b>Спасибо за загрузку фото!</b>'
      })
      .catch(err => {
        container.innerHTML = `<b>Извините, при загрузке возникли проблемы.` +
        `Пожалуйста, <a href="/newsletter">попробуйте еще раз</a>`
      })
    })
</script>
```

Важно отметить, что здесь мы преобразуем элемент формы в объект `FormData`, который может быть напрямую принят fetch как тело запроса. Вот и все! Поскольку кодировка тела запроса такая же, как и при отправке стандартными средствами браузера, то обработчик практически такой же. Нам нужно просто вернуть ответ в формате JSON вместо перенаправления:

```
exports.api.vacationPhotoContest = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.send({ result: 'success' })
}
```

Усовершенствование интерфейса пользователя для загрузки файлов

Надо сказать, что встроенный в браузер управляющий элемент для загрузки файлов `<input>` не очень хорош с точки зрения интерфейса пользователя. Вы, вероятно, видели куда более привлекательные интерфейсы с применением технологии перетаскивания объектов и кнопки для загрузки файлов.

Хорошая новость в том, что изученные вами методики применимы практически ко всем популярным «навороченным» компонентам для загрузки файлов. В конце

концов, под красивой оболочкой большинства из них кроется один и тот же механизм загрузки.

Ниже приведены некоторые из наиболее популярных клиентских приложений для загрузки файлов.

- jQuery File Upload (http://bit.ly/2Qbcd6I).
- Uppy (http://bit.ly/2rEFWeb) (его преимущество в поддержке загрузки файлов из различных популярных источников).
- file-upload-with-preview (http://bit.ly/2X5fS7F) (дает полный контроль; у вас есть доступ к массиву объектов файлов, который можно использовать для создания объекта `FormData` для `fetch`).

Резюме

В этой главе вы познакомились со множеством методов обработки форм. Мы исследовали как традиционный способ, которым браузеры обрабатывают формы (браузер делает к серверу запрос POST с содержимым формы, и ответ с сервера, обычно перенаправленный, отображается), так и набирающий распространение подход, при котором отправка формы стандартными средствами браузера предотвращается и мы выполняем обработку с помощью fetch.

Мы узнали о широко распространенных способах кодирования форм.

- `application/x-www-form-urlencoded` — применяемое по умолчанию кодирование, которым легко пользоваться, обычно связаное с традиционной обработкой форм.
- `application/json` — кодирование, получившее распространение для отправки данных (не файлов) с помощью fetch.
- `multipart/form-data` — кодирование, использующееся при передаче файлов.

Теперь, зная, как загрузить пользовательские данные на сервер, обратимся к *cookie-файлам* и *сеансам*, которые также служат для синхронизации сервера и клиента.

9 Cookie-файлы и сеансы

В этой главе вы узнаете, как пользоваться cookie-файлами и сеансами, чтобы улучшить взаимодействие с пользователями путем запоминания их предпочтений при переходе с одной страницы на другую или между сеансами браузера.

HTTP — протокол *без сохранения состояния*. Это означает, что, когда вы загружаете страницу в своем браузере и затем переходите на другую страницу того же сайта, ни у сервера, ни у браузера нет никакой внутренней возможности знать, что это тот же браузер посещает тот же сайт. Другими словами, Интернет работает таким образом, что *каждый HTTP-запрос содержит всю информацию, необходимую серверу для удовлетворения этого запроса*.

Однако есть проблема: если бы на этом все заканчивалось, мы никогда бы не смогли войти ни на один сайт. Потоковое видео не работало бы. Сайты забывали бы ваши настройки при переходе с одной страницы на другую. Так что обязан существовать способ формирования состояния поверх HTTP, и тут в кадре появляются cookie-файлы и сеансы.

Cookie-файлы, к сожалению, заработали дурную славу из-за тех неблаговидных целей, для которых использовались. Это печально, поскольку cookie-файлы на самом деле очень важны для функционирования современного Интернета (хотя HTML5 предоставил некоторые новые возможности, например локальные хранилища, которые можно использовать для тех же целей).

Идея cookie-файла проста: сервер отправляет фрагмент информации, который браузер хранит на протяжении настраиваемого промежутка времени. Содержание этого фрагмента информации полностью зависит от сервера. Часто это просто уникальный идентификационный номер (ID), соответствующий конкретному браузеру, так что поддерживается определенная имитация сохранения состояния.

Есть несколько важных фактов о cookie-файлах, которые вы должны знать.

- *Cookie-файлы не тайна для пользователя*. Все cookie-файлы, отправляемые сервером клиенту, доступны последнему для просмотра. Нет причин, по которым вы не могли бы отправить что-либо в зашифрованном виде для защиты содержимого, но необходимость в этом возникает редко (по крайней мере если вы не занимаетесь чем-то неблаговидным!). Подписанные cookie-файлы, о которых

мы немного поговорим далее, могут обфусцировать (то есть затруднить понимание) содержимое cookie-файла, но это не даст никакой криптографической защиты от любопытных глаз.

- *Пользователь может удалить или запретить cookie-файлы.* Пользователи полностью контролируют cookie-файлы, а браузеры позволяют удалять их скопом или по отдельности. Но, если вы не замышляете что-то нехорошее, у вас нет причин делать это, однако такая возможность удобна при тестировании. Пользователи также могут запретить cookie-файлы, что создает больше проблем, поскольку лишь простейшие веб-приложения могут обходиться без cookie-файлов.
- *Стандартные cookie-файлы могут быть подделаны.* Всякий раз, когда браузер выполняет запрос к вашему серверу со связанным cookie-файлом и вы слепо доверяете содержимому этого cookie-файла, вы превращаетесь в мишень для атаки. Верхом безрассудства было бы, например, выполнять содержащийся в cookie-файле код. Используйте подписанные cookie-файлы, чтобы наверняка знать, что они не подделаны.
- *Cookie-файлы могут использоваться для атак.* В последние годы появилась категория атак, называемых межсайтовым скриптингом (cross-site scripting, XSS). Один из методов XSS включает зловредный JavaScript, меняющий содержимое cookie-файлов. Это еще одна причина не доверять содержимому возвращаемых вашему серверу cookie-файлов. В этой ситуации помогает использование подписанных cookie-файлов (вмешательство будет заметно в подписанном cookie-файле, и неважно, кто его изменил: пользователь или зловредный JavaScript), кроме того, есть настройка, указывающая, что cookie-файлы должен изменять только сервер. Такие cookie-файлы, возможно, годятся не для всех случаев, но они, безусловно, безопаснее.
- *Пользователи заметят, если вы станете злоупотреблять cookie-файлами.* Пользователей будет раздражать, если вы станете устанавливать множество cookie-файлов на их компьютеры или хранить там массу данных. Лучше этого избегать. Старайтесь сводить использование cookie-файлов к минимуму.
- *Сеансы предпочтительнее cookie-файлов.* В большинстве случаев для сохранения состояния вы можете использовать *сеансы*. Это не только разумно, но и удобно, так как вам не нужно беспокоиться о возможном неправильном использовании хранилищ ваших пользователей, а также безопасно. Конечно, сеансы используют cookie-файлы, но в этом случае всю грязную работу за вас выполнит Express.

Cookie-файлы не магия: когда сервер хочет сохранить на клиенте cookie-файл, он отправляет заголовок Set-Cookie, содержащий пары «имя/значение»; а когда клиент отправляет запрос серверу, от которого он получил cookie-файлы, он отправляет многочисленные заголовки запроса Cookie, содержащие значения cookie-файлов.

Внешнее хранение данных доступа

Для безопасности использования cookie-файлов необходим так называемый *секрет cookie*. Секрет cookie-файла представляет собой строку, известную серверу и используемую для шифрования защищенных cookie-файлов перед их отправкой клиенту. Это не пароль, который необходимо помнить, так что он может быть просто случайной строкой. Для генерации секрета cookie я обычно использую случайное число или генератор случайных паролей (http://bit.ly/2QcjuDb), на создание которого вдохновил комикс xkcd.

Внешнее хранение данных доступа к сторонним ресурсам, таких как секрет cookie-файла, пароли к базам данных и токены для доступа к API (Twitter, Facebook и т. п.), является распространенной практикой. Это не только облегчает сопровождение (путем упрощения нахождения и обновления данных доступа), но и позволяет исключить файл с данными доступа из системы контроля версий, что особенно критично для репозиториев с открытым исходных кодом, размещаемых в GitHub или других общедоступных репозиториях исходного кода.

Для этого мы будем хранить данные доступа в файле JSON. Создайте файл `.credentials.development.json`:

```
{
  "cookieSecret": "...здесь находится ваш секрет cookie-файла"
}
```

Это файл данных доступа, которым мы будем пользоваться в процессе разработки. Таким образом, вы можете использовать разные данные доступа для разных окружений: окружения конечных пользователей (production), тестирования (test) или других, что очень удобно.

Чтобы упростить управление зависимостями при росте нашего приложения, добавим слой абстракций над файлом данных доступа. Наша версия будет очень простой. Создайте файл `config.js`:

```
const env = process.env.NODE_ENV || 'development'
const credentials = require(`./.credentials.${env}`)
module.exports = { credentials }
```

Теперь, чтобы случайно не добавить этот файл в наш репозиторий, внесите `credentials.js` в ваш файл `.gitignore`:

```
const { credentials } = require('./config')
```

В дальнейшем мы будем использовать этот же файл для хранения других данных доступа, но пока нам достаточно лишь нашего секрета cookie-файла.

Если вы используете прилагаемый к книге репозиторий, вам придется создать собственный файл данных доступа, так как он не включен в репозиторий.

Cookie-файлы в Express

Прежде чем устанавливать в своем приложении cookie-файлы и обращаться к ним, вам необходимо подключить промежуточное ПО `cookie-parser`. Сначала выполните `npm install cookie-parser`, затем (`ch09/meadowlark.js` в прилагаемом репозитории):

```
const cookieParser = require('cookie-parser')
app.use(cookieParser(credentials.cookieSecret))
```

Как только вы это сделали, можете устанавливать cookie-файлы или подписанные cookie-файлы везде, где у вас есть доступ к объекту ответа:

```
res.cookie('monster', 'нам-нам')
res.cookie('signed_monster', 'нам-нам', { signed: true })
```

Подписанные cookie-файлы имеют приоритет перед неподписанными. Если вы назовете ваш подписанный cookie-файл signed_monster, у вас не может быть неподписанного cookie-файла с таким же названием (он вернется как undefined).

Чтобы извлечь значение cookie-файла (если оно есть), отправленного с клиента, просто обратитесь к свойствам `cookie` или `signedCookie` объекта запроса:

```
const monster = req.cookies.monster
const signedMonster = req.signedCookies.signed_monster
```

Вы можете использовать в качестве имени cookie любую строку, какую пожелаете. Например, мы могли применить 'signed monster' вместо 'signed_monster', но тогда пришлось бы использовать скобочную нотацию для извлечения cookie-файла: req.signedCookies['signed monster']. По этой причине я рекомендую применять имена cookie-файлов без специальных символов.

Для удаления cookie-файла используйте `req.clearCookie`:

```
res.clearCookie('monster')
```

При установке cookie-файла вы можете указать следующие опции:

- `domain` — управляет доменами, с которыми связан cookie-файл, что позволяет привязывать cookie-файлы к конкретным поддоменам. Обратите внимание, что вы не можете установить cookie-файл для домена, отличного от того, на котором работает ваш сервер: он просто не будет выполнять какие-либо действия;
- `path` — управляет путем, на который распространяется действие данного cookie-файла. Обратите внимание, что в путях предполагается неявный символ

подстановки (wildcard) в конце: если вы используете путь / (по умолчанию), он будет распространяться на все страницы вашего сайта. Если используете путь /foo, он будет распространяться на пути /foo, /foo/bar и т. д.;

- maxAge — определяет, сколько времени (в миллисекундах) клиент должен хранить cookie-файл до его удаления. Если вы опустите эту опцию, cookie-файл будет удален при закрытии браузера (можете также указать дату окончания срока действия cookie-файла с помощью опции expires, но синтаксис при этом удручающий. Я рекомендую использовать maxAge);
- secure — указывает, что данный cookie-файл будет отправляться только через защищенное (HTTPS) соединение;
- httpOnly — установка этому параметру значения true указывает, что cookie-файл будет изменяться только сервером. То есть JavaScript на стороне клиента не может его изменять. Это помогает предотвращать XSS-атаки;
- signed — установите значение true, чтобы подписать данный cookie-файл, делая его доступным в res.signedCookies вместо res.cookies. Поддельные подписанные cookie-файлы будут отклонены сервером, а значение cookie-файла возвращено к первоначальному значению.

Просмотр cookie-файлов

Вероятно, в рамках тестирования вам понадобится способ просматривать cookie-файлы в вашей системе. У большинства браузеров имеется возможность просматривать cookie-файлы и хранимые ими значения. Например, в Chrome откройте инструменты разработчика и выберите вкладку Application (Приложение). В дереве слева вы увидите пункт Cookies. Разверните его — и увидите в списке сайт, который просматриваете в текущий момент. Нажмите на него — перед вами появятся все связанные с этим сайтом cookie-файлы. Вы можете также щелкнуть правой кнопкой мыши на домене для очистки всех cookie-файлов или на отдельном cookie-файле для его удаления.

Сеансы

Сеансы — всего лишь более удобный инструмент для сохранения состояния. Для реализации сеансов необходимо *что-нибудь* хранить на клиенте, в противном случае сервер не сможет распознать, что следующий запрос выполняет тот же клиент. Обычный способ для этого — cookie-файл, содержащий уникальный идентификатор. Сервер будет использовать этот идентификатор для извлечения информации о соответствующем сеансе.

Cookie-файлы — не единственный способ достижения этой цели. Во время пика паники по поводу cookie-файлов, когда процветало злоупотребление ими, многие пользователи просто отключали cookie-файлы. Тогда были изобретены другие методы сохранения состояния, такие как добавление к URL сеансовой информации. Эти методики оказались сложными, неэффективными, выглядели неряшливо, и лучше пусть они остаются в прошлом. HTML5 предоставляет другую возможность для сеансов — *локальное хранилище*, которое имеет преимущество перед cookie-файлами, если вам нужно хранить большое количество данных. За более подробной информацией обращайтесь к документации MDN для `Window.localStorage` (https://mzl.la/2CDrGo4).

Вообще, существует два способа реализации сеансов: хранить все в cookie-файле или хранить в cookie-файле только уникальный идентификатор, а все остальное — на сервере. Первый способ называется «сеансы на основе cookie-файлов» и является не самым удачным вариантом использования cookie. Как бы то ни было, он означает хранение всего вносимого вами в сеанс в браузере клиента, а такой подход я никак не могу рекомендовать. Я посоветовал бы этот подход, только если вы собираетесь хранить лишь небольшой фрагмент информации, не возражаете против доступа к нему пользователя и уверены, что с течением времени такая система не выйдет из-под контроля. Если вы хотите использовать этот подход, взгляните на промежуточное ПО `cookie-session` (http://bit.ly/2qNv9h6).

Хранилища в памяти

Если вы склоняетесь к хранению сеансовой информации на сервере, что я и советую делать, вам потребуется место для хранения. Простейший вариант — сеансы в памяти. Их легко настраивать, однако у них есть колоссальный недостаток: при перезагрузке сервера (а во время работы с данной книгой вы будете делать это многократно!) ваша сеансовая информация пропадет. Хуже того, при масштабировании до нескольких серверов (см. главу 12) обслуживать запрос каждый раз может другой сервер: сеансовая информация иногда будет в наличии, а иногда — нет. Очевидно, что это совершенно неприемлемо при реальной эксплуатации, однако вполне достаточно для нужд разработки и тестирования. Мы узнаем, как организовать постоянное хранение сеансовой информации, в главе 13.

Сначала установим `express-session` (`npm install express-session`), затем, после подключения парсера cookie-файлов, подключим `express-session` (`ch09/meadowalrk.js` в репозитории, прилагаемом к книге):

```
const expressSession = require('express-session')
// Убедитесь, что вы подключили
// промежуточное ПО cookie
// до промежуточного ПО session!
app.use(expressSession({
```

```
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
}))
```

Промежуточное ПО `express-session` принимает конфигурационный объект со следующими опциями.

- `resave` — заставляет сеанс заново сохраняться в хранилище, даже если запрос не менялся. Предпочтительнее устанавливать этот параметр в `false` (для получения дополнительной информации см. документацию по `express-session`).
- `saveUninitialized` — установка этого параметра в `true` приводит к сохранению новых (неинициализированных) сеансов в хранилище, даже если они не менялись. Предпочтительнее (и даже необходимо, если вам требуется получить разрешение пользователя перед установкой cookie-файла) устанавливать этот параметр в `false` (см. документацию по `express-session` для получения дополнительной информации).
- `secret` — ключ (или ключи), используемый для подписания cookie-файла идентификатора сеанса. Может быть тем же ключом, что и применяемый для `cookie-parser`.
- `key` — имя cookie-файла, в котором будет храниться уникальный идентификатор сеанса. По умолчанию `connect.sid`.
- `store` — экземпляр сеансового хранилища. По умолчанию это экземпляр `MemoryStore`, что вполне подходит для наших текущих целей. В главе 13 мы рассмотрим, как использовать в качестве хранилища базу данных.
- `cookie` — настройки для cookie-файла сеанса (`path`, `domain`, `secure` и т. д.). Применяются стандартные значения по умолчанию для cookie-файлов.

Использование сеансов

Как только вы настроите сеансы, их использование станет элементарным: просто воспользуйтесь свойствами переменной `session` объекта запроса:

```
req.session.userName = 'Anonymous'
const colorScheme = req.session.colorScheme || 'dark'
```

Обратите внимание, что при работе с сеансами нам не нужно использовать объект запроса для чтения значения и объект ответа для установки значения — все это выполняется через объект запроса (у объекта ответа нет свойства `session`). Чтобы удалить сеанс, можно использовать оператор JavaScript `delete`:

```
req.session.userName = null // Этот оператор устанавливает
                            // 'userName' в значение null, но не удаляет.

delete req.session.colorScheme // А этот удаляет 'colorScheme'.
```

Использование сеансов для реализации уведомлений

Уведомления — просто средство обеспечения обратной связи с пользователями таким способом, который не мешал бы их навигации. Простейший метод реализации уведомлений — использование сеансов (можно также использовать строку запроса, но при этом возникнут более уродливые URL, а уведомления начнут попадать в закладки браузера, чего, вероятно, вам не хотелось бы). Сначала настроим HTML. Мы будем использовать уведомления (alert) Bootstrap, так что убедитесь, что Bootstrap у вас подключен (см. документацию по основам Bootstrap: http://bit.ly/36YxeYf). Вы можете подключить файлы Bootstrap CSS и JavaScript в вашем основном шаблоне, пример этого есть в репозитории, прилагаемом к книге. В файле шаблона, где-нибудь в заметном месте (обычно непосредственно после шапки сайта), поместите следующее:

```
{{#if flash}}
  <div class="alert alert-dismissible alert-{{flash.type}}">
    <button type="button" class="close"
      data-dismiss="alert" aria-hidden="true">&times;<button>
    <strong>{{flash.intro}}</strong> {{{flash.message}}}
  </div>
{{/if}}
```

Обратите внимание, что мы используем три фигурные скобки для `flash.message` — это позволит использовать простой HTML в наших сообщениях (возможно, нам захочется выделить какие-то слова или включить гиперссылки). Теперь подключим необходимое промежуточное ПО для добавления в контекст объекта `flash`, если таковой имеется в сеансе. Сразу после однократного отображения уведомления желательно удалить его из сеанса, чтобы оно не отображалось при следующем запросе. Мы создадим промежуточное ПО, которое будет проверять сеанс на наличие уведомлений и, если они есть, передавать их в объект `res.locals`, делая доступными для представлений. Мы поместим это промежуточное ПО в файл `lib/middleware/flash.js`:

```
module.exports = (req, res, next) => {
  // Если имеется уведомление,
  // переместим его в контекст, а затем удалим.
  res.locals.flash = req.session.flash
  delete req.session.flash
  next()
})
```

В файле `meadowalrk.js` до каких-либо маршрутов представления подключим промежуточное ПО уведомлений:

```
const flashMiddleware = require('./lib/middleware/flash')
app.use(flashMiddleware)
```

Теперь посмотрим, как использовать уведомления в реальности. Представьте, что мы подписываем пользователей на новостную рассылку и хотим перенаправить их в архив рассылки после того, как они подпишутся. Обработчик форм при этом мог бы выглядеть примерно так:

```
// Немного измененная версия официального регулярного выражения
// W3C HTML5 для электронной почты:
// https://html.spec.whatwg.org/multipage/forms.htmlvalid-e-mail-address.
const VALID_EMAIL_REGEX = new RegExp('^[a-zA-Z0-9.!#$%&\'*+\/=?^_`{|}~-]+@' +
  '[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?' +
  '(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)+$');

app.post('/newsletter', function(req, res){
  const name = req.body.name || '', email = req.body.email || ''
  // Проверка вводимых данных
  if(VALID_EMAIL_REGEX.test(email)) {
    req.session.flash = {
      type: 'danger',
      intro: 'Ошибка проверки!',
      message: 'Введенный вами адрес электронной почты
      некорректен.',
    }
    return res.redirect(303, '/newsletter/archive');
  }
  // NewsletterSignup – пример объекта, который вы могли бы
  // создать; поскольку все реализации различаются, оставляю
  // написание этих зависящих от конкретного проекта интерфейсов
  // на ваше усмотрение. Это просто демонстрация того, как типичная
  // реализация на основе Express может выглядеть в вашем проекте.
  new NewsletterSignup({ name, email }).save((err) => {
    if(err) {
      req.session.flash = {
        type: 'danger',
        intro: 'Ошибка базы данных!',
        message: 'Произошла ошибка базы данных.
        Пожалуйста, попробуйте позже',
      }
      return res.redirect(303, '/newsletter/archive')
    }
    req.session.flash = {
      type: 'success',
      intro: 'Спасибо!',
      message: 'Вы были подписаны на информационный бюллетень.',
    };
    return res.redirect(303, '/newsletter/archive')
  })
})
```

Обратите внимание, что мы отличаем ошибки проверки вводимых данных от ошибки базы данных. Запомните: если мы выполняем проверку вводимых данных

в клиентской части, ее также следует выполнять в серверной части, поскольку пользователи со злыми намерениями проверку в клиентской части могут обойти.

Здорово иметь реализованный механизм уведомлений на сайте, даже если он не подходит для всех случаев (например, уведомления не всегда подходят для мастеров с несколькими формами или потоками подсчета стоимости в корзине для виртуальных покупок). Кроме того, их очень удобно использовать во время разработки в качестве способа обеспечения обратной связи, даже если потом вы замените их на другой механизм. Добавление поддержки для уведомлений — то, что я делаю при настройке сайта в первую очередь, и мы будем использовать этот метод на протяжении всей книги.

Поскольку уведомление перемещается из сеанса в res.locals.flash в промежуточном ПО, вам придется выполнять перенаправление, чтобы оно отобразилось. Если вы хотите отображать уведомление без перенаправления, устанавливайте значение res.locals.flash вместо req.session.flash.

В примере в этой главе применяется браузерная отправка форм с перенаправлением, так как управление интерфейсом пользователя с помощью сеансов обычно не используется в приложениях с отправкой форм через Ajax. При этом событии (отправке формы) желательно указывать любые ошибки в объекте формата JSON, который возвращает обработчик формы, а клиентский код будет модифицировать DOM на сервере, чтобы сообщения об ошибках отображались динамически. Это не значит, что для приложений с рендерингом на стороне клиента сеансы бесполезны, но они редко используются для этих целей.

Для чего использовать сеансы

Сеансы удобны, когда вам необходимо сохранить предпочтения пользователя, относящиеся к нескольким страницам. Чаще всего сеансы используются для предоставления информации об аутентификации пользователя: вы входите в систему — и создается сеанс. После этого вам не нужно выполнять вход в систему всякий раз, когда вы перезагружаете страницу. Сеансы могут быть полезны даже без учетных записей пользователя. Для сайтов вполне обычно запоминать, какая сортировка вам нравится или какой формат представления даты вы предпочитаете, — и все это без необходимости входить в систему.

Несмотря на то что я советую отдавать предпочтение сеансам, а не cookie-файлам, важно понимать, как работают последние (в частности, потому, что они делают возможным функционирование сеансов). Это поможет вам в вопросах диагностики, а также облегчит процесс понимания механизмов безопасности и защиты персональной информации вашим приложением.

Резюме

Понимание cookie-файлов и сеансов помогает лучше разобраться в том, как веб-приложения поддерживают видимость сохранения состояния, когда базовый протокол HTTP этого не подразумевает. Мы познакомились с методиками работы с cookie-файлами и сеансами, позволяющими контролировать взаимодействие пользователя с вашим сайтом.

Кроме того, мы писали промежуточное ПО, не вдаваясь при этом в подробные пояснения. В следующей главе мы вплотную займемся промежуточным ПО и изучим все, что к нему относится!

10 Промежуточное ПО

Мы уже немного соприкасались с промежуточным ПО: использовали существующее промежуточное ПО (`body-parser`, `cookie-parser`, `static` и `express-session` как минимум) и даже писали свое собственное (для добавления данных о погоде в контекст шаблона, настройки уведомлений и обработчика состояния 404). Но что же такое промежуточное ПО?

На понятийном уровне *промежуточное ПО* — инструмент для инкапсуляции функциональности, особенно той, что работает с HTTP-запросом к вашему приложению. На деле это просто функция, принимающая три аргумента: объект запроса, объект ответа и функцию `next`, о которой мы вскоре поговорим (существует также разновидность функции для обработки ошибок, которая принимает четыре аргумента, она будет описана в конце этой главы).

Промежуточное ПО выполняется способом, называемым *конвейерной обработкой*. Представьте обычную трубу, по которой течет вода. Вода закачивается через один конец трубы, проходит через манометры и клапаны, прежде чем попадает по назначению. Важный нюанс этой аналогии в том, что *очередность имеет значение*: если вы поместите манометр перед клапаном, воздействие будет отличаться от случая, когда манометр расположен после. Аналогично, если имеется клапан, который что-то впрыскивает в воду, все, что ниже по течению от этого клапана, будет содержать добавляемый ингредиент. В приложениях Express вставка промежуточного ПО в конвейер осуществляется путем вызова `app.use`.

До версии Express 4.0 организация конвейера осложнялась необходимостью явного подключения *маршрутизатора*. В зависимости от места его подключения маршруты могли оказаться скомпонованными беспорядочно, что делало очередность конвейера менее очевидной из-за смешивания обработчиков маршрутов и промежуточного ПО. В Express 4.0 обработчики маршрутов и промежуточное ПО вызываются в том порядке, в котором они подключены, что делает очередность более понятной.

Последним промежуточным ПО в вашем конвейере становится универсальный обработчик для любого запроса, не подходящего ни для одного прочего маршрута.

Это общепринятая практика. Такое промежуточное ПО обычно возвращает код состояния 404 («Не найдено»).

Так как же запрос завершается в конвейере? Это определяется функцией `next`, передаваемой каждому промежуточному ПО: если вы не вызовете `next()`, запрос завершится по окончании работы данного промежуточного ПО.

Принципы работы промежуточного ПО

Гибкость мышления относительно промежуточного ПО и обработчиков маршрутов — ключ к пониманию того, как работает Express. Вот несколько вещей, которые вам следует иметь в виду.

- Обработчики маршрутов (`app.get`, `app.post` и т. д., взятые вместе, они часто называются `app.METHOD`) могут рассматриваться как промежуточное ПО, обрабатывающее только конкретный глагол HTTP (`GET`, `POST` и т. д.). И наоборот, промежуточное ПО можно рассматривать как обработчик маршрутов, обрабатывающий все глаголы HTTP (что, по существу, эквивалентно `app.all`, обрабатывающему все глаголы HTTP; для экзотических глаголов вроде `PURGE` могут быть небольшие отличия, но для обычных глаголов разницы не будет).
- В качестве первого параметра обработчикам маршрутов требуется путь. Если вы хотите, чтобы он подходил для любого маршрута, просто используйте `/*`. Промежуточное ПО в качестве первого параметра также может принимать путь, но это необязательный параметр (его отсутствие будет означать любой путь, как если бы вы указали `*`).
- Обработчики маршрутов и промежуточное ПО принимают функцию обратного вызова с двумя, тремя или четырьмя параметрами (формально параметров может быть ноль или один, но практического применения эти варианты не находят). Если параметров два или три, первые два — объекты запроса и ответа, а третий — функция `next`. Если параметров четыре — промежуточное ПО становится *обрабатывающим ошибки*, а первый параметр — объектом ошибки, за которым следуют объекты запроса, ответа и объект `next`.
- Если вы не вызываете `next()`, конвейер будет завершен и обработчики маршрутов или промежуточного ПО больше выполняться не будут. Если вы не вызываете `next()`, то должны отправить ответ клиенту (`res.send`, `res.json`, `res.render` и т. п.). Если этого не сделать, соединение зависнет и в конечном счете будет разорвано из-за превышения лимита ожидания.
- Если вы вызываете `next()`, то, как правило, отправлять ответ клиенту нецелесообразно. Если вы его отправите, то будут выполнены промежуточное ПО или обработчики маршрутов, находящиеся дальше в конвейере, но любые ответы, отправляемые ими клиенту, будут проигнорированы.

Примеры промежуточного ПО

Если хотите увидеть все это в действии, попробуем применить простейшее промежуточное ПО (`ch10/00-simple-middleware.js` в прилагаемом репозитории):

```
app.use((req, res, next) => {
  console.log('обработка запроса для ${req.url}....')
  next()
})

app.use((req, res, next) => {
  console.log('завершаем запрос')
  res.send('Спасибо за игру!')
  // Обратите внимание, что мы здесь не вызываем next()...
  // Запрос на этом завершается.
})

app.use((req, res, next) => {
  console.log('Упс, меня никогда не вызовут!')
})
```

Здесь мы видим три примера промежуточного ПО. Первое просто отправляет в консоль сообщение, перед тем как передать запрос следующему промежуточному ПО в конвейере посредством вызова `next()`. Затем следующее промежуточное ПО непосредственно обрабатывает запрос. Обратите внимание: если мы опустим здесь вызов `res.send`, то ответ никогда не будет возвращен клиенту. В конечном счете соединение с клиентом будет разорвано из-за превышения лимита времени. Последнее промежуточное ПО никогда не будет выполнено, поскольку все запросы завершены в предыдущих промежуточных ПО.

Теперь рассмотрим более сложный и полный пример (файл `route-example.js` в прилагаемом репозитории):

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('\n\nВСЕГДА')
  next()
})

app.get('/a', (req, res) => {
  console.log('/a: маршрут завершен')
  res.send('a')
})
app.get('/a', (req, res) => {
  console.log('/a: никогда не вызывается')
})
```

```
app.get('/b', (req, res, next) => {
  console.log('/b: маршрут не завершен')
  next()
})
app.use((req, res, next) => {
  console.log('ИНОГДА')
  next()
})
app.get('/b', (req, res, next) => {
  console.log('/b (часть 2): сгенерирована ошибка')
  throw new Error('b не выполнено')
})
app.use('/b', (err, req, res, next) => {
  console.log('/b ошибка обнаружена и передана далее');
  next(err)
})
app.get('/c', (err, req) => {
  console.log('/c: сгенерирована ошибка')
  throw new Error('c не выполнено')
})

app.use('/c', (err, req, res, next) => {
  console.log('/c: ошибка обнаружена, но не передана
  далее')
  next()
})

app.use((err, req, res, next) => {
  console.log('обнаружена необработанная ошибка: ' +
  err.message)
  res.send('500 – ошибка сервера')
})

app.use((req, res) => {
  console.log('маршрут не обработан')
  res.send('404 – не найдено')
})

const port = process.env.PORT || 3000
app.listen(port, () => console.log(`Express запущен на http://
localhost:${port}` +
  '; для завершения нажмите Ctrl+C.'))
```

Перед тем как выполнить этот пример, представьте себе его результат. Каковы будут маршруты? Что увидит клиент? Что будет выведено в консоль? Если вы сумели правильно ответить на все три вопроса, значит, поняли, как работать с маршрутами в Express. Обратите особое внимание на различие между запросом к `/b` и запросом к `/c` — в обоих случаях имеется ошибка, но одна приводит к коду состояния 404, а другая — к коду состояния 500.

Обратите внимание, что промежуточное ПО *должно* быть функцией. Помните, что в JavaScript возврат функции из функции — обычное и довольно распространенное явление. Например, вы увидите, что `express.static` — функция, и мы фактически вызываем ее, так что она должна возвращать другую функцию. Рассмотрите следующее:

```
app.use(express.static)        // Это не будет работать так, как ожидается.
console.log(express.static())  // Будет выводить в консоль "function",
                               // указывая, что express.static – функция,
                               // которая сама возвращает функцию.
```

Следует отметить, что модуль может экспортировать функцию, которая, в свою очередь, может использоваться непосредственно в качестве промежуточного ПО. Например, вот модуль `lib/tourRequiresWaiver.js` (пакетные предложения Meadowlark Travel по скалолазанию требуют документа, освобождающего фирму от ответственности):

```
module.exports = (req,res,next) => {
  const { cart } = req.session
  if(!cart) return next()
  if(cart.items.some(item => item.product.requiresWaiver)) {
    cart.warnings.push('Один или более выбранных вами' +
      'туров требуют документа об отказе от ответственности.)
  }
  next()
}
```

Мы можем подключить это промежуточное ПО, например, вот так (`ch10/02-item-waiver.example.js` в прилагаемом репозитории):

```
const requiresWaiver = require('./lib/tourRequiresWaiver')
app.use(requiresWaiver)
```

Чаще всего, однако, вы бы экспортировали объект, свойства которого являются промежуточным ПО. Например, поместим весь код подтверждения корзины для виртуальных покупок в `lib/cartValidation.js`:

```
module.exports = {

  resetValidation(req, res, next) {
    const { cart } = req.session
    if(cart) cart.warnings = cart.errors = []
    next()
  },

  checkWaivers(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.product.requiresWaiver)) {
```

```
      cart.warnings.push('Один или более выбранных вами' +
      'туров требуют документа об отказе от ответственности.')
    }
    next()
  },

  checkGuestCounts(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.guests > item.product.maxGuests )) {
      cart.errors.push('В одном или более из выбранных вами ' +
      'туров недостаточно мест для выбранного вами ' +
      'количества гостей')
    }
  next()
  },
}
```

После этого можно подключить промежуточное ПО следующим образом:

```
const cartValidation = require('./lib/cartValidation')

app.use(cartValidation.resetValidation)
app.use(cartValidation.checkWaivers)
app.use(cartValidation.checkGuestCounts)
```

В предыдущем примере у нас было промежуточное ПО, довольно рано прерывающее свое выполнение с помощью оператора return next(). Express не предполагает, что промежуточное ПО будет возвращать значение (и не производит никаких действий с возвращаемыми значениями), так что это всего лишь более краткий способ записи next(); return.

Распространенное промежуточное ПО

Хотя на npm есть тысячи проектов промежуточного ПО, только некоторые из них получили широкое распространение и являются основными и лишь единицы из их числа можно обнаружить в каждом нетривиальном проекте Express. Некоторое промежуточное ПО было настолько общепринятым, что даже попало в пакет Express, но затем было перемещено в отдельные пакеты. В пакете Express осталось только промежуточное ПО `static`.

В этом списке я попытался перечислить наиболее распространенное промежуточное ПО!

- `basicauth-middleware` — обеспечивает базовую (basic) схему авторизации. Имейте в виду, что базовая авторизация предоставляет лишь минимальную

защиту, поэтому лучше использовать базовую авторизацию *только* поверх HTTPS (в противном случае имена пользователей и пароли будут передаваться открытым текстом). Используйте базовую авторизацию лишь в тех случаях, когда требуется что-то очень быстрое и простое и вы применяете HTTPS.

- `body-parser` — производит парсинг тела HTTP-запроса. Предоставляет промежуточное ПО для парсинга тела запроса в кодировке URL, JSON и др.
- `busboy`, `multiparty`, `formidable`, `multer` — все эти варианты промежуточного ПО производят парсинг тела запроса, закодированного с помощью `multipart/form-data`.
- `compression` — сжимает данные ответа с помощью gzip или deflate. Это отличная вещь, за которую ваши пользователи скажут спасибо, особенно те, у кого медленное или мобильное подключение к Интернету. Его желательно подключать как можно раньше, до любого промежуточного ПО, которое может отправить ответ. Единственное, что я советую подключать до `compress`, — это отладочное или журналирующее промежуточное ПО, которое не отправляет ответы. Следует отметить, что большинство сред промышленной эксплуатации не нуждаются в этом промежуточном ПО, так как сжатие производится прокси-серверами типа NGINX.
- `cookie-parser` — обеспечивает поддержку cookie-файлов (см. главу 9).
- `cookie-session` — обеспечивает поддержку хранения сеансовой информации в cookie-файлах. В целом я не рекомендую такой подход к сеансам. Это промежуточное ПО должно подключаться после `cookie-parser` (см. главу 9).
- `express-session` — обеспечивает поддержку сеансов на основе идентификатора сеанса, хранимого в cookie-файле. По умолчанию используется хранилище в памяти, не подходящее для промышленной эксплуатации, но для использования можно настроить хранилище на основе базы данных (см. главы 9 и 13).
- `csurf` — обеспечивает защиту от атак типа «межсайтовая подделка запроса» (cross-site request forgery, CSRF). Использует сеансы, так что должно быть подключено после промежуточного ПО `express-session`. К сожалению, простое подключение этого промежуточного ПО автоматически не дает защиты от CSRF-атак (см. главу 18 для получения более подробной информации).
- `serve-index` — предоставляет поддержку перечисления каталогов для статических файлов. Это промежуточное ПО может понадобиться только в том случае, если возникнет необходимость в перечислении каталогов.
- `errorhandler` — отправляет трассировку стека и сообщения об ошибке на клиент. Я не рекомендую подключение этого промежуточного ПО на промышленном сервере, так как оно раскрывает подробности реализации,

что может неблагоприятно повлиять на безопасность или защиту персональной информации (см. главу 20 для получения более подробной информации).

- `serve-favicon` — выдает favicon (пиктограмму, появляющуюся в полосе заголовка браузера). Не является жизненно необходимым: вы можете просто поместить `favicon.ico` в корневой каталог вашего каталога для статических файлов, но это промежуточное ПО может повысить производительность. Если вы его используете, помните, что его нужно подключать в самом верху стека промежуточного ПО. Оно позволяет также использовать отличающееся от `favicon.ico` имя файла.
- `morgan` — обеспечивает поддержку автоматического логирования: все запросы будут записываться в логи (см. главу 20 для получения более подробной информации).
- `method-override` — обеспечивает поддержку заголовка запроса `x-http-method-override`, позволяющего браузерам подделывать использование HTTP-методов, отличных от `GET` или `POST`. Может быть полезным для отладки. Требуется, только если вы пишете API.
- `response-time` — добавляет в ответ заголовок `X-Response-Time`, содержащий время ответа в миллисекундах. Обычно это промежуточное ПО не требуется, разве что вы выполняете настройку производительности.
- `static` — обеспечивает поддержку выдачи статических (общедоступных) файлов. Вы можете подключать это промежуточное ПО неоднократно, указывая различные каталоги (см. главу 17 для получения более подробной информации).
- `vhost` — виртуальные хосты (vhosts) — термин, заимствованный у Apache, — упрощают управление поддоменами в Express (см. главу 14 для получения более подробной информации).

Промежуточное ПО сторонних производителей

В настоящий момент не существует хранилища или предметного указателя промежуточного ПО сторонних производителей. Тем не менее практически все промежуточное ПО Express доступно в npm, так что если вы выполните поиск в npm по ключевым словам Express и middleware, то получите неплохой список. В официальной документации Express также есть полезный список промежуточного ПО (http://bit.ly/36UrbnL).

Резюме

Из этой главы мы узнали, что такое промежуточное ПО, как самостоятельно его писать и как оно обрабатывается в качестве части приложения Express. Если вы стали воспринимать Express просто как коллекцию промежуточного ПО, то вы начали понимать Express! Даже обработчики маршрутов, которые мы до сих пор использовали, — это просто специализированное промежуточное ПО.

В следующей главе мы рассмотрим другую широко распространенную потребность инфраструктуры: отправку электронной почты (будем надеяться, что и для этого есть какое-нибудь промежуточное ПО).

11 Отправка электронной почты

Одно из главных средств связи вашего сайта с окружающим миром — электронная почта. Отправка электронной почты — функциональная возможность, в которой важно все, начиная с регистрации пользователей и инструкций по восстановлению забытого пароля до рекламных писем и уведомлений о проблемах. В этой главе вы узнаете, как улучшить взаимодействие с пользователями посредством форматирования и отправки электронных писем с помощью Node и Express.

Ни у Node, ни у Express нет встроенной возможности отправки электронной почты, так что нам придется использовать сторонний модуль. Я рекомендую великолепный пакет Nodemailer, созданный Андрисом Рейнманом. Перед тем как углубиться в настройку Nodemailer, разберемся с основными понятиями электронной почты.

SMTP, MSA и MTA

Всеобщий язык отправки сообщений электронной почты — *простой протокол электронной почты* (Simple Mail Transfer Protocol, SMTP). SMTP можно использовать для отправки электронной почты непосредственно на почтовый сервер получателя, но это плохая идея: если вы не являетесь доверенным отправителем вроде Google или Yahoo!, высока вероятность того, что ваше письмо попадет прямиком в папку для спама. Лучше использовать *агент отправки почты* (Mail Submission Agent, MSA), который будет доставлять электронную почту по доверенным каналам, снижая вероятность того, что ваше письмо будет помечено как спам. Вдобавок, чтобы гарантировать доставку вашего письма, агенты отправки почты умеют справляться с неприятностями вроде временных перебоев в обслуживании и возвращенных писем. Последний член этого уравнения — почтовый сервер (Mail Transfer Agent, MTA) — сервис, фактически отправляющий письмо конечному адресату. В рамках данной книги *MSA*, *MTA* и *SMTP-сервер*, по сути, эквивалентные понятия.

Итак, вам понадобится доступ к MSA. Вы можете начать работу с использования распространенных бесплатных сервисов электронной почты, таких как Gmail,

Outlook или Yahoo!, но они уже не столь дружественны к автоматизированной рассылке электронных писем, как прежде (попытка предотвратить злоупотребление). К счастью, у вас есть возможность выбора между двумя прекрасными сервисами электронной почты с опцией экономии трафика: SendGrid (https://sendgrid.com/) и Mailgun (https://www.mailgun.com/). Я пользовался обоими сервисами, и оба мне понравились. В примерах в этой книге будет применяться SendGrid.

Если вы работаете в организации, у нее может быть свой MSA, так что свяжитесь с вашим отделом IT и спросите, есть ли в организации ретранслятор SMTP для автоматизированной рассылки сообщений электронной почты.

Если вы используете SendGrid или Mailgun, то пришло время настроить ваш аккаунт. Для SendGrid потребуется создать ключ API, который будет вашим паролем для SMTP.

Получение сообщений электронной почты

Большинству сайтов нужна только возможность *отправлять* сообщения электронной почты, например инструкции по восстановлению пароля или рекламные письма. Однако некоторым приложениям нужно также получать сообщения электронной почты. Хороший пример — система отслеживания проблемных вопросов, отправляющая письмо, когда кто-то обновляет обсуждение проблемы: если вы отвечаете на это письмо, обсуждение автоматически дополняется вашим ответом.

К сожалению, получение сообщений электронной почты гораздо сложнее и в данной книге рассматриваться не будет. Если вам необходима такая функциональность, переложите поддержку почтового ящика на провайдера электронной почты и получите периодический доступ к нему через агент IMAP, такой как imap-simple (http://bit.ly/2qQK0r5).

Заголовки сообщений электронной почты

Сообщение электронной почты состоит из двух частей: заголовка и тела (весьма схоже с HTTP-запросом). *Заголовок* содержит информацию о письме: от кого оно, кому адресовано, дату получения, тему и т. д. Заголовки — то, что пользователь обычно видит в приложении электронной почты, но на самом деле их число гораздо больше. Основная часть почтовых программ позволяет посмотреть заголовки. Если вы никогда этого не делали, рекомендую попробовать. Заголовки предоставляют всю информацию о том, как письмо добралось до вас: в заголовке перечислены каждый сервер и MTA, через которые прошло письмо.

Людей часто удивляет, что некоторые заголовки, такие как адрес От, могут быть заданы отправителем произвольным образом. То, что вы задаете адрес От, отличающийся от учетной записи, с которой отправляете письмо, часто называют *спуфингом*. Ничто не мешает вам отправить письмо с адреса Билла Гейтса billg@microsoft.com. Я не советую этого делать, а просто довожу до вашего сведения, что вы можете

установить в некоторых заголовках любые значения, какие только захотите. Иногда есть уважительные причины так поступать, но злоупотреблять этим не стоит.

Как бы то ни было, у сообщения электронной почты, которое вы отправляете, *должен* быть адрес От. Иногда это обстоятельство вызывает проблемы при автоматизированной рассылке писем, поэтому часто можно видеть письма с обратными адресами типа «Не отвечайте на это письмо» (do-not-reply@meadowlarktravel.com). Станете ли вы использовать этот подход, или автоматизированные рассылки будут приходить с адреса вроде Meadowlark Travel info@meadowlarktravel.com — дело ваше, но, если вы решите использовать второй подход, будьте готовы отвечать на письма, которые придут на адрес info@meadowlarktravel.com.

Форматы сообщений электронной почты

Когда Интернет только появился, все сообщения электронной почты представляли собой обычный текст в кодировке ASCII. С тех пор мир сильно изменился, и люди хотят отправлять письма на различных языках и делать нетривиальные вещи вроде вставки форматированного текста, изображений и вложений. Здесь дело начинает принимать иной оборот: форматы и кодировки сообщений электронной почты — жуткая смесь методов и стандартов.

К счастью, нам не нужно решать эти проблемы — Nodemailer сделает это за нас. Единственное, что нужно знать, — это то, что письмо может быть или неформатированным текстом (Unicode), или HTML.

Практически все современные приложения электронной почты поддерживают письма в формате HTML, так что в целом форматирование писем электронной почты в HTML не должно создавать проблем. Тем не менее попадаются текстовые пуристы, избегающие использования писем в формате HTML, так что я советую всегда включать как текстовый, так и HTML-вариант письма. Если вас не устраивает необходимость писать и текстовый, и HTML-варианты, Nodemailer поддерживает сокращенную форму вызова для автоматической генерации текстовой версии из HTML.

Сообщения электронной почты в формате HTML

Сообщения электронной почты в формате HTML — тема, которой хватило бы на целую книгу. К сожалению, это не так просто, как писать HTML, подобный тому, который вы могли бы писать для вашего сайта: большинство почтовых программ поддерживают лишь небольшое подмножество HTML. По большей части вам придется писать HTML так, как если бы на дворе все еще был 1996 г. Одним словом, не слишком приятное дело. В частности, вам придется вернуться к использованию таблиц для макетов (звучит грустная музыка).

Если вы уже сталкивались с проблемами совместимости браузеров с HTML, то знаете, какой головной болью это может обернуться. Проблемы совместимости

электронной почты — гораздо хуже. К счастью, существует несколько вещей, которые могут помочь с этим.

Во-первых, я призываю вас прочитать замечательную статью о написании писем в формате HTML (http://bit.ly/33CsaXs) из базы знаний MailChimp. В ней прекрасно описаны азы и объяснено, на что обратить внимание при написании писем в формате HTML.

Во-вторых, рекомендую то, что сэкономит массу времени: HTML Email Boilerplate (http://bit.ly/2qJ1XIe). По сути, это исключительно хорошо написанный и тщательно протестированный шаблон для писем в формате HTML.

В-третьих, тестирование. Допустим, вы разобрались в том, как писать сообщения в формате HTML, и уже используете HTML Email Boilerplate, однако тестирование — единственный способ убедиться, что ваше письмо не «взорвется» на Lotus Notes 7 (да, его все еще используют). Хотелось бы инсталлировать 30 различных почтовых программ для тестирования одного письма? К счастью, существует отличный сервис Litmu (http://bit.ly/2NI6JPo), который сделает все за вас. Он недешевый, тарифные планы стартуют от 100 долларов в месяц, но, если вы отправляете много рекламных писем, такая цена себя оправдает.

В то же время, если вы используете довольно простое форматирование, в дорогом сервисе тестирования, таком как Litmus, нет необходимости. Если вы ограничитесь заголовками, жирным/курсивным шрифтом, горизонтальными линейками и ссылками на изображения, то вам нечего опасаться.

Nodemailer

Вначале нам необходимо установить пакет Nodemailer:

```
npm install nodemailer
```

Далее подключим пакет `nodemailer` и создадим экземпляр Nodemailer (транспорт, говоря языком Nodemailer):

```
const nodemailer = require('nodemailer')

const mailTransport = nodemailer.createTransport({

  auth: {
    user: credentials.sendgrid.user,
    pass: credentials.sendgrid.password,
  }
})
```

Замечу, что мы используем модуль данных доступа, который настроили в главе 9. Вам понадобится модифицировать файл `credentials.js` следующим образом:

```
{
  "cookieSecret": "здесь находится ваш секрет cookie-файла",
  "sendgrid": {
```

```
    "user": "ваше имя пользователя sendgrid",
    "password": "ваш пароль sendgrid"
  }
}
```

Общепринятые параметры конфигурации для SMTP включают порт, тип аутентификации и параметры TLS. Однако большинство основных почтовых сервисов применяют параметры по умолчанию. Чтобы узнать, какие параметры нужно использовать, ознакомьтесь с документацией вашего почтового сервиса (введите поиск по словам «отправка сообщений через SMTP», «настройка SMTP» или «пересылка через SMTP»). Если у вас возникли трудности при отправке сообщений через SMTP, возможно, вам нужно проверить параметры; полный перечень поддерживаемых параметров смотрите в документации Nodemailer (https://nodemailer.com/smtp).

Если вы выполняете задания в соответствии с прилагаемым к книге репозиторием, то обнаружите, что в файле данных доступа нет никаких настроек. Ранее многие пользователи обращались ко мне с вопросом, почему файл пустой или отсутствует. Я намеренно не предоставляю действительные данные доступа по той же причине, по которой и вы должны быть осторожны с вашими данными доступа! Я доверяю вам, дорогой читатель, но не настолько, чтобы дать вам пароль от ящика моей электронной почты!

Отправка писем

Теперь, когда у нас есть экземпляр почтового транспорта, можем отправлять письма. Начнем с очень простого примера, в котором текстовое сообщение отправляется единственному адресату (`ch11/00-smtp.js` в прилагаемом репозитории):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joecustomer@gmail.com',
    subject: 'Ваш тур Meadowlark Travel',
    text: 'Спасибо за заказ поездки в Meadowlark Travel. ' +
    'Мы ждем Вас с нетерпением!',
  })
  console.log('письмо успешно отправлено: ', result)
} catch(err) {
  console.log('невозможно отправить письмо: ' + err.message)
}
```

В примерах кода в этом параграфе я использую фиктивные адреса электронной почты типа joecustomer@gmail.com. Вам, вероятно, придется заменить их на принадлежащий вам адрес, чтобы проверить, что происходит. С другой стороны, на бедный joecustomer@gmail.com придет куча бессмысленных писем!

Вы могли заметить, что здесь мы обрабатываем ошибки, но важно понимать, что отсутствие ошибок не говорит об успешной доставке сообщения электронной почты *адресату*. Параметр обратного вызова `error` будет устанавливаться в случае проблем только при обмене сообщениями с MSA, такими как сетевая ошибка или ошибка аутентификации. Если MSA не сумел доставить письмо (например, по причине неправильного адреса электронной почты или неизвестной учетной записи пользователя), вам нужно будет проверить действия в вашем аккаунте на вашем почтовом сервисе, что можно сделать как из интерфейса администратора, так и через API.

Если необходимо, чтобы ваша система автоматически определяла, успешно ли было доставлено письмо, воспользуйтесь API вашего почтового сервиса. За более подробной информацией обращайтесь к документации по API вашего почтового сервиса.

Отправка писем нескольким адресатам

Nodemail поддерживает отправку почты нескольким адресатам путем отделения адресатов друг от друга запятыми (`ch11/01-multiple-recipients.js` в прилагаемом репозитории):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' + 'fred@hotmail.com',
    subject: 'Ваш тур Meadowlark Travel',
    text: 'Спасибо за заказ поездки в Meadowlark Travel. ' +
    'Мы ждем Вас с нетерпением!',
  })
  console.log('письмо успешно отправлено: ', result)
} catch(err) {
  console.log('невозможно отправить письмо: ' + err.message)
}
```

Обратите внимание, что в этом примере мы вперемешку использовали чистые адреса электронной почты (joe@gmail.com) и адреса, в которых указано имя получателя (Джейн Клиент jane@yahoo.com). Такой синтаксис допустим.

При отправке электронной почты многим адресатам нужно внимательно следить за тем, чтобы не превысить ограничения вашего MSA. SendGrid, например, ограничивает количество получателей одного письма цифрой 1000. Если вы делаете массовые рассылки, вероятно, вам захочется отправлять много писем со множеством получателей в каждом (`ch11/02-many-recipients.js` в прилагаемом репозитории):

```
// largeRecipientList — массив адресов электронной почты.
const recipientLimit = 100
const batches = largeRecipientList.reduce((batches, r) => {
  const lastBatch = batches[batches.length - 1]
  if(lastBatch.length < recipientLimit)
    lastBatch.push(r)
```

```
  else
    batches.push([r])
  return batches
}, [[]])
try {
  const results = await Promise.all(batches.map(batch =>
    mailTransport.sendMail({
      from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
      to: batch.join(', '),
      subject: 'Специальная цена на туристический пакет "Худ-Ривер"!',
      text: 'Закажите поездку по живописной реке прямо сейчас!',
    })
  ))
  console.log(results)
} catch(err) {
  console.log('по крайней мере один пакет писем ' +
              'не был доставлен: ' + err.message)
}
```

Рекомендуемые варианты для массовых рассылок

Безусловно, можно делать массовые рассылки с помощью Nodemailer и соответствующего MSA, но вам следует хорошо подумать, прежде чем пойти по этому пути. Осознание своей ответственности при организации кампании по рассылке электронной почты предполагает предоставление людям способа отказаться от получения ваших рекламных писем, и это отнюдь не простая задача. Умножьте это на количество поддерживаемых вами списков рассылки (у вас может быть, например, еженедельный информационный бюллетень и кампания со специальными уведомлениями). В этой области лучше не изобретать велосипед. Такие сервисы, как Emma (https://myemma.com/), MailChimp (http://mailchimp.com/) и Campaign Monitor (http://www.campaignmonitor.com/), предоставляют вам все необходимые возможности, включая отличные инструменты для наблюдения за успешностью кампании по рассылке. Их расценки вполне доступны, и я очень рекомендую использовать их для рассылки рекламных писем, информационных бюллетеней и т. д.

Отправка писем в формате HTML

До сих пор мы отправляли сообщения электронной почты в виде неформатированного текста, но в наше время большинство людей ожидает чего-то более симпатичного. Nodemailer позволяет отправлять в одном письме как текстовую, так и HTML-версию, оставляя почтовой программе выбор, какую версию отображать (обычно HTML) (`ch11/03-html-email.js` в прилагаемом репозитории):

```
const result = await mailTransport.sendMail({
  from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
  to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
  'fred@hotmail.com',
```

```
  subject: 'Ваш тур от Meadowlark Travel',
  html: '<h1>Meadowlark Travel</h1>\n<p>Спасибо за заказ ' +
  поездки в Meadowlark Travel. ' +
  '<b>Мы ждем Вас с нетерпением!</b>',
  text: 'Спасибо за заказ поездки в Meadowlark Travel. ' +
  'Мы ждем Вас с нетерпением!',
})
```

Такой подход требует массы усилий. Особенно это касается ситуации, когда лишь некоторые из ваших пользователей предпочитают чисто текстовые письма. Если вы хотите сэкономить время, можете писать письма в формате HTML и автоматически генерировать текст из HTML с помощью пакета, аналогичного html-to-formatted-text (http://bit.ly/34RX8Lq). Просто имейте в виду, что качество не будет таким же высоким, как при написании текста вручную; HTML не всегда можно преобразовать чисто.

Изображения в письмах в формате HTML

Вставлять изображения в письма в формате HTML можно, однако я этого не одобряю, поскольку они раздувают сообщения электронной почты. Вместо этого вам стоит обеспечить доступ к изображениям на своем сервере, которые вы бы хотели использовать в письмах, и ссылаться на них в письмах соответствующим образом[1].

Лучше всего отвести в папке со статическими ресурсами специальное место для изображений из сообщений электронной почты. При этом стоит держать отдельно ресурсы, которые вы используете и на своем сайте, и в электронной почте. Это снижает вероятность нежелательного влияния на макет писем.

Добавим ресурсы электронной почты в проект сайта Meadowlark Travel. Создайте в каталоге `public` подкаталог `email`. Вы можете поместить туда `logo.png` и любые другие изображения, которые хотели бы использовать в письмах. Теперь можете использовать эти изображения в своей электронной почте напрямую:

```
<img src="//meadowlarktravel.com/email/logo.png"
  alt="Meadowlark Travel Logo">
```

Понятно, что не следует использовать localhost при рассылке сообщений электронной почты другим людям; у них, вероятно, даже нет работающего сервера, тем более на порте 3000! В зависимости от почтовой программы вы можете использовать localhost в своих письмах для целей тестирования, но за пределами вашего компьютера это работать не будет. В главе 17 мы обсудим некоторые способы оптимизации процесса перехода от разработки к промышленной эксплуатации.

[1] Следует учитывать, что многие почтовые клиенты из соображений безопасности не позволяют загружать изображения. По этой причине в письме желательно указывать ссылку на его веб-версию или делать письма такими, чтобы они были читабельными даже при отсутствии изображений. — *Примеч. пер.*

Использование представлений для отправки писем в формате HTML

До сих пор мы помещали HTML в строки внутри JavaScript, но этой практики лучше избегать. Пока что наш HTML был довольно прост, но взгляните на HTML Email Boilerplate (http://bit.ly/2qJ1XIe): хочется ли вам вставлять весь этот шаблонный код в строку? Конечно, нет!

К счастью, мы можем использовать для этой цели представления. Рассмотрим пример письма «Спасибо за заказ поездки в Meadowlark Travel», который немного расширим. Допустим, у нас имеется корзина для виртуальных покупок, содержащая информацию о заказе. Этот объект будет сохраняться в сеансе. Пусть последний шаг в процессе заказа — форма, которую обрабатывает `/cart/checkout`, отправляющий письмо с подтверждением. Начнем с создания представления для страницы «Спасибо вам», `views/cart-thank-you.handlebars`:

```
<p>Спасибо за заказ поездки в Meadowlark Travel, {{cart.billing.name}}!</p>
<p>Ваш номер бронирования: {{cart.number}}, на адрес {{cart.billing.email}}
было отправлено письмо в качестве подтверждения.</p>
```

Затем создаем для письма шаблон. Скачайте HTML Email Boilerplate и поместите его в `views/email/cart-thank-you.handlebars`. Отредактируйте файл и измените тело следующим образом:

```
<table cellpadding="0" cellspacing="0" border="0" id="backgroundTable">
  <tr>
    <td valign="top">
      <table cellpadding="0" cellspacing="0" border="0" align="center">
        <tr>
          <td width="200" valign="top"><img class="image_fix"
              src="//placehold.it/100x100"
              alt="Meadowlark Travel" title="Meadowlark Travel"
              width="180" height="220" /></td>
        </tr>
        <tr>
          <td width="200" valign="top"><p>
              Спасибо за заказ поездки
              В Meadowlark Travel, {{cart.billing.name}}.</p>
              <p> Ваш номер бронирования: {{cart.number}}.
              </p></td>
        </tr>
        <tr>
          <td width="200" valign="top">Проблемы
              с бронированием? Свяжитесь с Meadowlark Travel по номеру
              <span class="mobile_link">555-555-0123</span>.</td>
        </tr>
      </table>
   </td>
  </tr>
</table>
```

Поскольку вы не можете использовать адреса localhost в письмах, то, если ваш сайт еще не в режиме промышленной эксплуатации, для графики можно использовать сервис-заполнитель. Например, http://placehold.it/100x100 динамически выдает квадратное изображение со стороной 100 пикселов, которое вы можете использовать. Этот метод весьма часто используется для изображений, предназначенных для обозначения местоположения (for placement only, FPO) и для компоновки макетов.

Теперь мы можем создать маршрут для страницы «Спасибо вам» нашей корзины (`ch11/04-rendering-html-email.js` в прилагаемом репозитории):

```
app.post('/cart/checkout', (req, res, next) => {
  const cart = req.session.cart
  if(!cart) next(new Error('Корзина не существует.'))
  const name = req.body.name || '', email = req.body.email || ''
  // Проверка вводимых данных.
  if(!email.match(VALID_EMAIL_REGEX))
    return res.next(new Error('Некорректный адрес ' +
                              'электронной почты.'))
  // Присваиваем случайный идентификатор корзины;
  // при обычных условиях мы бы использовали
  // здесь идентификатор из БД.
  cart.number = Math.random().toString().replace(/^0\.0*/, '')
  cart.billing = {
    name: name,
    email: email,
  }
  res.render('email/cart-thank-you', { layout: null, cart: cart },
    (err,html) => {
      console.log('визуализируемое сообщение: ', html)
      if( err ) console.log('ошибка в шаблоне письма');
      mailTransport.sendMail({
        from: '"Meadowlark Travel": info@meadowlarktravel.com',
        to: cart.billing.email,
        subject: 'Спасибо за заказ поездки в ' +
        'Meadowlark Travel',
        html: html,
        text: htmlToFormattedText(html),
      })
      .then(info => {
        console.log('sent! ', info)
        res.render('cart-thank-you', { cart: cart })
      })
      .catch(err => {
        console.error('Не могу отправить ' +
        'подтверждение: ' + err.message)
      })
    }
  )
})
```

Обратите внимание на то, что мы вызываем `res.render` дважды. В обычных условиях вы вызываете его только один раз (второй вызов отобразит результаты первого вызова). Однако в данном случае мы обходим обычный процесс рендеринга во время первого вызова: заметьте, что мы передаем параметром функцию обратного вызова. Выполнение этого действия препятствует рендерингу результатов представления в браузере. Взамен функция обратного вызова получает отображенное представление в параметре `html`: все, что нам остается сделать, — взять этот отображенный HTML-код и отправить письмо! Мы указываем `layout: null`, чтобы избежать использования нашего файла макета, поскольку он весь содержится в шаблоне письма (другой возможный подход — создать отдельный файл макета для сообщений электронной почты и использовать его вместо этого). Наконец, мы снова вызываем `res.render`. На этот раз результаты будут, как полагается, отображены в HTML-ответе.

Инкапсуляция функциональности электронной почты

Если в вашем сайте электронная почта используется в значительном объеме, вы можете инкапсулировать ее функциональность. Предположим, что ваш сайт всегда должен отправлять письма от одного и того же отправителя (Meadowlark Travel, info@meadowlarktravel.com) и при этом они должны быть в формате HTML с автоматически сгенерированным текстом. Создадим модуль `lib/email.js` (`ch11/lib/email.js` в прилагаемом репозитории):

```
const nodemailer = require('nodemailer')
const htmlToFormattedText = require('html-to-formatted-text')

module.exports = credentials => {

  const mailTransport = nodemailer.createTransport({
    host: 'smtp.sendgrid.net',
    auth: {
      user: credentials.sendgrid.user,
      pass: credentials.sendgrid.password,
    },
  })

  const from = '"Meadowlark Travel" <info@meadowlarktravel.com>'
  const errorRecipient = 'youremail@gmail.com'

  return {
    send: (to, subject, html) =>
    mailTransport.sendMail({
      from,
      to,
      subject,
```

```
      html,
      text: htmlToFormattedText(html),
    }),
  }

}
```

Теперь все, что нам нужно сделать, чтобы отправить письмо (`ch11/05-email-library.js` в прилагаемом репозитории):

```
const emailService = require('./lib/email')(credentials)

emailService.send('joecustomer@gmail.com',
  'Сегодня распродажа туров Худ-Ривер!',
  'Налетайте на них, пока не остыли!')
```

Резюме

В этой главе вы изучили основы отправки электронных сообщений в Интернете. Если вы выполняли задания вручную, то настроили бесплатный сервис электронной почты (скорее всего, это SendGrid или Mailgun) и использовали его для отправки как текстового, так и HTML-варианта писем. Вы узнали, как механизм рендеринга шаблонов, который мы применяли для рендеринга HTML-кода в приложениях Express, может быть использован в случае с сообщениями электронной почты в формате HTML.

Сообщения электронной почты остаются важным способом взаимодействия с пользователями. Постарайтесь не злоупотреблять этой мощью! У вас, наверное, как и у меня, ящик переполнен сообщениями автоматизированной рассылки, которые вы игнорируете. Чем меньше таких сообщений, тем лучше. Ваше приложение может отправлять письма пользователям по уважительным причинам и из соображений полезности, но вам всегда следует спрашивать себя: «*Действительно* ли мои пользователи хотят получать эти письма? Может быть, есть другой способ донести эту информацию?»

Мы рассмотрели базовую инфраструктуру, и теперь нам нужно создавать приложения. Мы потратим немного времени на обсуждение последующего ввода в эксплуатацию нашего приложения и обсудим, как сделать это успешно.

12 Промышленная эксплуатация

Несмотря на то что на данной стадии обсуждение вопросов промышленной эксплуатации может показаться преждевременным, вы сэкономите немало времени и защитите себя от головной боли в будущем, если начнете думать об этом уже сейчас. День запуска наступит так быстро, что и опомниться не успеете.

В этой главе вы узнаете о поддержке, предоставляемой Express для различных сред выполнения, способах масштабирования вашего сайта и о том, как осуществлять мониторинг его состояния. Вы увидите, как можно смоделировать условия для тестирования и разработки, а также как выполнить нагрузочное тестирование, чтобы выявить эксплуатационные проблемы до их возникновения.

Среды выполнения

Express поддерживает концепцию *сред выполнения* — инструментов для запуска приложения в режиме эксплуатации (production), разработки (development), тестирования (test). В действительности у вас может быть столько сред выполнения, сколько пожелаете. Например, могут быть промежуточная среда (staging) и тренировочная среда (training). Однако не забывайте о том, что разработка, эксплуатация и тестирование — стандартные среды: и Express, и стороннее промежуточное ПО часто принимают решения, опираясь именно на них. Другими словами, если у вас есть промежуточная среда, для нее нельзя организовать автоматическое наследование свойств среды эксплуатации. Поэтому я рекомендую придерживаться стандартных сред эксплуатации, разработки и тестирования.

Несмотря на то что существует возможность задания среды выполнения с помощью вызова `app.set('env', 'production')`, делать этого не стоит, поскольку ваше приложение всегда будет запускаться в одной и той же среде независимо от ситуации. Хуже того, оно может начать работать в одной среде, а затем переключиться на другую.

Предпочтительнее указывать среду выполнения с помощью переменной среды `NODE_ENV`. Изменим наше приложение для выдачи отчета о режиме, в котором оно работает, путем вызова `app.get('env')`:

```
const port = process.env.PORT || 3000
  app.listen(port, () => console.log(`Express запущено в режиме ` +
  `${app.get('env')} на http://localhost:${port}` +
  `; нажмите Ctrl+C для завершения.`))
```

Если вы сейчас запустите свой сервер, то увидите, что работаете в режиме разработки. Этот режим будет использоваться по умолчанию, если вы не укажете другой вариант. Попробуем переключить его в режим эксплуатации:

```
$ export NODE_ENV=production
$ node meadowlark.js
```

Если вы пользуетесь Unix/BSD, существует удобный синтаксис, позволяющий менять среду только на время действия команды:

```
$ NODE_ENV=production node meadowlark.js
```

При этом сервер будет запущен в режиме эксплуатации, но, когда его выполнение завершится, значение переменной `NODE_ENV` окажется неизмененным. Мне очень нравится этот способ: снижается вероятность того, что я случайно присвою переменным среды не те значения.

Если вы запускаете Express в режиме эксплуатации, то можете столкнуться с предупреждением о компонентах, непригодных для использования в этом режиме. Если вы следили за примерами из данной книги, то видели, что connect.session использует хранилище в памяти, не подходящее для среды эксплуатации. Как только в главе 13 мы перейдем к использованию хранилища на основе базы данных, это предупреждение исчезнет.

Отдельные конфигурации для различных сред

Само переключение среды выполнения дает немного, разве что Express в режиме эксплуатации будет генерировать больше предупреждений в консоли (например, сообщит вам об устаревших модулях, которые в дальнейшем будут удалены). Помимо этого, режим эксплуатации по умолчанию включает кэширование представлений (см. главу 7).

Среда выполнения — это прежде всего инструмент, позволяющий принимать решения о том, как приложение должно себя вести в различных средах. Будьте внимательны и старайтесь сводить к минимуму различия между средами разработки, тестирования и эксплуатации. То есть используйте эту возможность умеренно. При существенных отличиях среды разработки или тестирования от среды эксплуатации увеличивается вероятность различий в поведении при эксплуатации — верный путь к повышению количества ошибок (или числа

труднообнаруживаемых ошибок). При этом некоторые различия неизбежны: например, если ваше приложение в значительной мере завязано на базе данных, вы вряд ли захотите испортить промышленную базу данных во время разработки, так что у вас появится хороший кандидат на роль отдельной конфигурации. Другая сфера, в которой напрашивается использование отдельной конфигурации, — более подробное логирование. Существует немало действий, которые стоило бы записать в логи во время разработки, но при этом они не требуют логирования в процессе эксплуатации.

Добавим логирование для сервера. Дело в том, что при эксплуатации и разработке поведение должно быть разным. Для разработки мы можем оставить поведение по умолчанию, но для эксплуатации нам нужно, чтобы логи записывались в файл. Мы будем использовать `morgan` (`npm install morgan`) — самое распространенное промежуточное ПО для логирования (`ch12/00-logging.js` в прилагаемом репозитории):

```
const morgan = require('morgan')
const fs = require('fs')

switch(app.get('env')){
  case 'development':
    app.use(require('morgan')('dev'));
    break;
  case 'production':
    const stream = fs.createWriteStream(__dirname + '/access.log',
      { flags: 'a' })
    app.use(morgan('combined', { stream }))
    break
}
```

Если вы запустите сервер как обычно (`node meadowlark.js`) и посетите ваш сайт, то увидите, что действия логируются в консоли. Чтобы посмотреть на поведение приложения в режиме эксплуатации, запустите его с `NODE_ENV=production`. Теперь при посещении приложения в терминале вы не заметите никаких действий (что, вероятно, желательно для промышленного сервера), но все они записываются в лог в комбинированном формате Apache Combined Log Format (http://bit.ly/2NGC592).

Мы сделали это, создав поток для записи с добавлением элементов в конце (`{ flags: a }`); и передав его в настройки morgan. У morgan есть много опций; чтобы посмотреть их все, обратитесь к документации по morgan (http://bit.ly/32H5wMr).

В предыдущем примере мы использовали __dirname для хранения лога запроса в подкаталоге самого проекта. Если захотите так сделать, вам придется добавить log в файл .gitignore. В качестве другого варианта можете использовать более Unix-подобный подход и сохранять логи в подкаталоге /var/log, как по умолчанию делает Apache.

Еще раз подчеркну, что конфигурации для различных сред следует выбирать максимально продуманно. Всегда помните, что, когда сайт находится в промышленной эксплуатации, его экземпляры будут (или должны) работать в режиме `production`. Когда бы вы ни пожелали внести изменение, относящееся к разработке, всегда следует сначала обдумать, к каким последствиям это может привести при эксплуатации. Более устойчивый к ошибкам пример отдельной конфигурации для режима эксплуатации мы рассмотрим в главе 13.

Запуск процесса в Node

До сих пор ваше приложение запускалось путем вызова напрямую командой `node` (например, `node meadowlark.js`). Это подходит для разработки и тестирования, но чревато проблемами в процессе эксплуатации. В частности, не предусмотрена защита в случае, если в работе приложения произойдет сбой или его выполнение прервется. Эту проблему может решить устойчивый к ошибкам менеджер процессов.

Некоторые хостинг-решения сами предоставляют менеджер процессов. Это значит, что поставщик услуг хостинга даст вам параметр конфигурации, который нужно указать в файле приложения, и веб-сервис будет управлять процессами.

Но если вам нужно управлять процессами самостоятельно, вы можете выбрать среди двух популярных менеджеров процессов:

- Forever (https://github.com/foreversd/forever);
- PM2 (https://github.com/Unitech/pm2).

Поскольку среды промышленной эксплуатации могут широко варьироваться, мы не будем вдаваться в подробности установки и конфигурации менеджера процессов. Как по Forever, так и по PM2 есть прекрасная документация, вы можете установить и использовать их на вашей рабочей машине, чтобы изучить настройку.

Я пользовался обоими менеджерами, и у меня не сложились определенные предпочтения. Forever немного проще и его легче начать использовать, а PM2 предоставляет больше функций.

Если вы не хотите тратить много времени на эксперименты с менеджерами процессов, рекомендую попробовать Forever. Вам нужно сделать лишь два шага. Сначала установите Forever:

```
npm install -g forever
```

Затем запустите ваше приложение с Forever из корневого каталога вашего приложения:

```
forever start meadowlark.js
```

Теперь ваше приложение выполняется и будет выполняться, даже если вы закроете окно терминала! Вы можете перезапустить процесс с помощью `forever restart meadowlark.js` и остановить посредством `forever stop meadowlark.js`.

Начать работать с PM2 немного сложнее, но это стоит того, если вам нужен собственный менеджер процессов для эксплуатации.

Масштабируем сайт

Масштабирование бывает двух типов: вертикальное и горизонтальное. *Вертикальное* масштабирование предполагает повышение мощности серверов: более быстрые CPU, лучшая архитектура, больше ядер, больше памяти и т. п. *Горизонтальное* же подразумевает обычное увеличение числа серверов. По мере роста популярности облачных вычислений и повсеместного распространения виртуализации значимость вычислительной мощности сервера постоянно уменьшается, так что горизонтальное масштабирование, как правило, наиболее эффективный в плане затрат метод масштабирования сайтов в соответствии с их потребностями.

При разработке сайтов для Node вам всегда следует предусматривать возможность горизонтального масштабирования. Даже если приложение совсем крошечное (возможно, это обычное приложение во внутренней сети организации, у которого всегда ограниченный круг пользователей) и предположительно никогда не будет нуждаться в масштабировании, эту привычку стоит выработать. В конце концов, возможно, ваш следующий проект Node окажется новым Twitter и масштабирование будет жизненно необходимо. К счастью, Node предоставляет отличную поддержку горизонтального масштабирования, а написание приложения благодаря этому становится безболезненным.

При создании сайта, ориентированного на горизонтальное масштабирование, очень важно помнить о персистентности данных. Если вы привыкли полагаться на файловое хранилище, *забудьте об этом прямо сейчас*!

Моя первая встреча с этой проблемой была поистине катастрофической. Один из наших заказчиков запустил интернет-конкурс. Под него было разработано веб-приложение для оповещения первых 50 победителей о том, что они получат приз. С этим конкретным заказчиком мы не могли спокойно использовать базу данных из-за определенных корпоративных IT-ограничений, так что практически все сохранение было реализовано путем записи неструктурированных файлов. Как только в файл записывалось 50 победителей, больше никто не получал оповещение о том, что победил. Проблема заключалась в балансировке нагрузки на сервер: половина запросов обрабатывалась одним сервером, половина — другим. Один сервер оповестил 50 человек о том, что они выиграли, и то же самое сделал второй сервер. К счастью, призы были небольшими (флисовые пледы), а не какие-нибудь iPad, так что заказчик согласился на дополнительные траты и вручил 100 призов вместо 50 (я предложил заплатить за 50 дополнительных пледов из своего кармана, так как это была моя ошибка, но он благородно отказался).

Мораль этой истории в том, что, если у вас нет файловой системы, доступной для *всех* ваших серверов, не полагайтесь на локальную файловую систему в вопросе сохраняемости. Исключениями являются данные, предназначенные только для чтения, и резервные копии. Например, я обычно делаю резервную копию от-

правленных данных формы в локальном неструктурированном файле на случай обрыва соединения с базой данных. В случае перебоя в обслуживании базы данных обойти все серверы и собрать файлы довольно хлопотно, но по крайней мере ничего не пропадет.

Горизонтальное масштабирование с помощью кластеров приложений

Node сам по себе поддерживает *кластеры приложений* — простую, рассчитанную на один сервер форму горизонтального масштабирования. С помощью кластеров приложений вы можете создать независимый сервер для каждого ядра (CPU) в системе (превышение количества серверов над количеством ядер не улучшит производительность вашего приложения). Кластеры приложений хороши по двум причинам: во-первых, они помогают максимизировать производительность конкретного сервера (аппаратного или виртуальной машины), во-вторых, это простой способ провести параллельное тестирование приложения.

Добавим поддержку кластеризации в наш сайт. Обычно все это делают в главном файле приложения, но мы создадим второй файл приложения. Он будет выполнять приложение в кластере, используя некластеризованный файл приложения, который мы постоянно использовали ранее. Чтобы это стало возможным, придется сначала внести небольшие изменения в `meadowlark.js` (упрощенный пример см. в `ch12/01-server.js` в прилагаемом репозитории):

```
function startServer(port) {
  app.listen(port, function() {
    console.log(`Express запущен в режиме ${app.get('env')} ` +
      `на http://localhost:${port}` +
      `; нажмите Ctrl+C для завершения.`)
  })
}

if(require.main === module) {
  // Приложение запускается непосредственно;
  // запускаем сервер приложения.
  startServer(process.env.PORT || 3000)
} else {
  // Приложение импортируется как модуль с помощью "require":
  // экспортируем функцию для создания сервера.
  module.exports = startServer
}
```

Как вы помните из главы 5, `if require.main === module` означает, что скрипт был запущен на выполнение напрямую, иначе он был бы вызван с помощью `require` из другого скрипта.

Далее создаем новый скрипт `meadowlark_cluster.js` (упрощенный пример см. в `ch12/01-server.js` в прилагаемом репозитории):

```
const cluster = require('cluster')

function startWorker() {
  const worker = cluster.fork()
  console.log(`КЛАСТЕР: Исполнитель ${worker.id} запущен`)
}

if(cluster.isMaster){

  require('os').cpus().forEach(startWorker)

  // Записываем в лог все отключившиеся исполнители;
  // если исполнитель отключается, он должен затем
  // завершить работу, так что мы подождем
  // события завершения работы для порождения
  // нового исполнителя ему на замену.
  cluster.on('disconnect', worker => console.log(
    `КЛАСТЕР: Исполнитель ${worker.id} отключился от кластера.`
  ))

  // Когда исполнитель завершает работу,
  // создаем исполнитель ему на замену.
  cluster.on('exit', (worker, code, signal) => {
    console.log(
      `КЛАСТЕР: Исполнитель ${worker.id} завершил работу ` +
      `с кодом завершения ${code} (${signal})`
    )
    startWorker()
  })

} else {

  const port = process.env.PORT || 3000
  // Запускаем наше приложение на исполнителе;
  // см. meadowlark.js.
  require('./meadowlark.js')(port)

}
```

В процессе выполнения этот JavaScript будет находиться в контексте основного приложения (когда он запускается непосредственно командой `node meadowlark_cluster.js`) или в контексте исполнителя, когда его выполняет кластерная система Node. Контекст выполнения определяется свойствами `cluster.isMaster` и `cluster.isWorker`. Когда мы выполняем этот скрипт, он выполняется в привилегированном режиме и мы запускаем исполнитель с помощью `cluster.fork` для каждого CPU в системе. Кроме того, мы заново порождаем все завершившие работу исполнители, выполняя прослушивание на предмет событий `exit` от исполнителей.

Наконец, в выражении `else` мы обрабатываем случай запуска на исполнителе. Поскольку мы настроили `meadowlark.js` на использование в качестве модуля, просто импортируем его и немедленно вызываем (вспомните, мы экспортировали его как функцию, запускающую сервер).

Теперь запустим новый кластеризованный сервер:

```
node meadowlark-cluster.js
```

Обратите внимание: если вы применяете виртуализацию (например, VirtualBox от компании Oracle), вам может потребоваться настройка VM для использования нескольких процессоров. По умолчанию виртуальные машины часто имеют только один процессор.

Если вы работаете в многоядерной системе, то должны увидеть, что запущено некоторое количество исполнителей. Если хотите увидеть доказательства того, что разные исполнители обрабатывают разные запросы, добавьте перед маршрутами следующее промежуточное ПО:

```
const cluster = require('cluster')

app.use((req, res, next) => {
  if(cluster.isWorker)
    console.log(`Исполнитель ${cluster.worker.id} получил запрос`)
  next()
})
```

Теперь вы можете подключиться к своему приложению с помощью браузера. Перезагрузив страницу несколько раз, вы увидите, как получаете новый исполнитель из пула при каждом запросе. (У вас может не получиться: Node рассчитан на обработку большого количества соединений и простой перезагрузки браузера не всегда достаточно, чтобы нагрузить его в полной мере. Позже мы изучим стресс-тестирование, и вы сможете лучше рассмотреть кластер в действии.)

Обработка неперехваченных исключений

В асинхронном мире Node неперехваченные исключения особенно важны. Начнем с простого примера, не доставляющего особых хлопот (я призываю вас следить за этими примерами):

```
app.get('/fail', (req, res) => {
  throw new Error('Нет!')
})
```

Express, выполняя обработчики маршрутов, помещает их в блок `try/catch`, что на самом деле нельзя считать неперехваченным исключением. Это не вызовет особых проблем: Express будет записывать в логи исключения на стороне

сервера, и посетитель сайта получит уродливый дамп стека вызовов. Тем не менее на стабильность вашего сервера это не повлияет, и остальные запросы продолжат выдаваться без ошибок. Чтобы обеспечить выдачу красивой страницы с сообщением об ошибке, создадим файл `views/500.handlebars` и добавим обработчик ошибки после всех маршрутов:

```
app.use((err, req, res, next) => {
  console.error(err.message, err.stack)
  app.status(500).render('500')
})
```

Демонстрация страницы с сообщением об ошибке в стиле вашего сайта — всегда хорошая практика: при появлении ошибок это не только создает впечатление профессионального подхода к делу, но и позволяет предпринять какие-то действия. Например, данный обработчик ошибок мог бы послужить неплохим местом для оповещения команды разработчиков о том, что произошла ошибка. К сожалению, это помогает только в случае исключений, которые Express может перехватить. Попробуем кое-что похуже:

```
app.get('/epic-fail', (req, res) => {
  process.nextTick(() =>
    throw new Error('Бабах!')
  )
})
```

Попытайтесь выполнить это. Результат окажется намного катастрофичнее: весь ваш сервер упадет. Помимо того что пользователь не получит сообщения об ошибке, сервер прекратит работу и запросы обрабатываться не будут. Это происходит потому, что `setTimeout` выполняется *асинхронно*: выполнение функции с исключением откладывается до момента бездействия Node. Проблема в том, что, когда Node оказывается не занят и находит возможность выполнить эту функцию, у него уже отсутствует контекст соответствующего запроса и не остается другого выхода, кроме как бесцеремонно остановить весь сервер по причине неопределенного состояния (Node не известны ни назначение данной функции, ни ее вызывающая функция, так что у него нет оснований полагать, что какие-либо последующие функции будут работать правильно).

Вызов функции process.nextTick похож на вызов функции setTimeout с параметром 0. Я применял ее здесь для демонстрационных целей — это совсем не то, что вы стали бы использовать в коде на стороне сервера в обычной ситуации. Однако в следующих главах мы будем иметь дело со многими выполняемыми асинхронно вещами, такими как доступ к базе данных и файловой системе, сетевой доступ и другие, которые могут быть подвержены данной проблеме.

Существуют действия, которые мы можем предпринять для обработки неперехваченных исключений, но *если Node не может установить стабильность вашего приложения, то и вы не сможете*. Другими словами, если имеется неперехваченное исключение, единственный выход — остановить сервер. Лучшее, что можно сделать в этом случае, — остановить его так мягко, как только возможно, и иметь в наличии механизм обработки отказа. Простейший механизм обработки отказа — использование кластера. Если ваше приложение работает в кластеризованном режиме и один из исполнителей останавливается, основное приложение порождает другой исполнитель на замену (вам даже не нужно много исполнителей — кластера с одним исполнителем вполне достаточно, хотя обработка отказа при этом будет происходить несколько медленнее).

Каким образом, зная все это, мы можем остановить сервер как можно мягче в случае появления неперехваченного исключения? В Node это происходит с помощью события `uncaughtException`. (У Node также есть механизм под названием *«домены»*, но он устарел и использовать его не стоит.)

```
process.on('uncaughtException', err => {
  console.error('НЕПЕРЕХВАЧЕННОЕ ИСКЛЮЧЕНИЕ\n', err.stack);
  // Выполните здесь всю необходимую очистку данных…
  // Закройте соединения с базой данных и т. д.
  process.exit(1)
})
```

Не стоит надеяться, что в вашем приложении никогда не будет неперехваченных исключений. У вас всегда должен быть механизм для записи исключения и оповещения о нем. К этому нужно относиться серьезно. Попробуйте определить, почему произошло исключение, чтобы вы могли исправить ситуацию. Такие сервисы, как Sentry (https://sentry.io), Rollbar (https://rollbar.com/), Airbrake (https://airbrake.io/) и New Relic (https://newrelic.com/), являются отличным способом записи такого типа ошибок для их анализа. Например, чтобы воспользоваться Sentry, нужно сначала зарегистрировать бесплатный аккаунт, после чего вы получите имя источника данных (data source name, DSN) и затем сможете модифицировать ваш обработчик событий:

```
const Sentry = require('@sentry/node')
Sentry.init({ dsn: '** YOUR DSN GOES HERE **' })

process.on('uncaughtException', err => {
  // Произведите здесь всю необходимую очистку данных…
  // Закройте соединения с базой данных и т. д.
  Sentry.captureException(err)
  process.exit(1)
})
```

Горизонтальное масштабирование с несколькими серверами

Хотя горизонтальное масштабирование с использованием кластеризации может максимизировать производительность отдельного сервера, что будет, если вам нужен не один сервер, а больше? Здесь все несколько усложняется. Чтобы добиться подобной разновидности параллелизма, вам потребуется *прокси-сервер* (его часто называют *обратным прокси-сервером*, чтобы отличать от прокси-серверов, широко используемых для доступа к внешним сетям, но я считаю подобную терминологию ненужной и запутывающей, так что буду называть его просто прокси).

Двумя весьма популярными вариантами являются NGINX (произносится «энджин экс») (https://www.nginx.com/) и HAProxy (http://www.haproxy.org/). Серверы NGINX, в частности, растут как грибы после дождя. Я недавно выполнял сравнительный анализ для своей компании и обнаружил, что 80 % наших конкурентов используют NGINX. Как NGINX, так и HAProxy — надежные высокопроизводительные прокси-серверы, подходящие даже для самых требовательных приложений. (Если вам необходимы доказательства, задумайтесь о том, что Netflix, отвечающий ни много ни мало за 15 % *всего трафика Интернета*, использует NGINX.)

Существуют также некоторые основанные на Node прокси-серверы поменьше, такие как `node-http-proxy` (bit.ly/34RWyNN). Это неплохие варианты для разработки или для случая, когда ваши требования невелики. Я бы порекомендовал для эксплуатации все-таки использовать NGINX или HAProxy (оба бесплатны, хотя предоставляют платную техническую поддержку).

Установка и настройка прокси-сервера выходит за рамки данной книги, но это не так сложно, как может показаться (особенно если вы используете `node-http-proxy` или другой легковесный прокси). Использование кластеров дает некоторые гарантии готовности нашего сайта к горизонтальному масштабированию.

Если вы настроили прокси-сервер, не забудьте сообщить Express, что используете прокси и что ему можно доверять:

```
app.enable('trust proxy')
```

Выполнение этого шага гарантирует, что `req.ip`, `req.protocol` и `req.secure` будут отображать подробности соединения между *клиентом и прокси*, а не клиентом и вашим приложением. Помимо этого, в массиве `req.ips` будут находиться исходный IP клиента, а также доменные имена или адреса всех промежуточных прокси.

Мониторинг сайта

Мониторинг сайта — одно из важнейших (и чаще всего игнорируемых) мероприятий по контролю качества, которые вы можете предпринять. Хуже, чем бодрствовать в три часа утра, ремонтируя легший сайт, может быть только ночной звонок от начальника, разгневанного тем, что сайт перестал работать (или, что вообще

ужасно, прийти утром на работу и осознать, что ваш клиент только что потерял 10 000 долларов на продажах, ибо сайт не работал всю ночь и этого никто не заметил).

Со сбоями сложно что-то сделать — они так же неизбежны, как смерть и налоги. Однако, если и есть что-то, что поможет вам убедить начальника и клиентов в своей компетентности, так это то, что вы всегда будете оповещены о сбоях раньше их.

Сторонние мониторы работоспособности

Наличие монитора работоспособности на сервере вашего сайта столь же действенно, как и наличие дымового пожарного извещателя в доме, где никто не живет. Возможно, он сумеет перехватить ошибки в случае сбоя отдельных страниц, но, если из строя выйдет весь сервер, он также может сломаться, даже не отправив вам сигнала SOS. Именно поэтому вашей первой линией защиты должны быть сторонние мониторы работоспособности. UptimeRobot (http://uptimerobot.com/) бесплатен вплоть до 50 используемых мониторов и прост в настройке. Предупреждения могут быть отправлены по электронной почте, через СМС (текстовое сообщение), Twitter или Slack (среди прочих). Вы можете следить за кодами возврата для отдельных страниц (любой код, отличный от 200, считается ошибкой) или проверять наличие/отсутствие ключевого слова на странице. Помните, что использование монитора ключевых слов может повлиять на получаемую вами аналитику (в большинстве аналитических сервисов вы можете исключить трафик от мониторов работоспособности).

Если же вам требуется больше возможностей, существуют другие, более дорогостоящие сервисы, например Pingdom (http://pingdom.com/) и Site24x7 (http://www.site24x7.com/).

Стресс-тестирование

Стрессовое (или *нагрузочное*) *тестирование* разработано, дабы вы были уверены в том, что ваш сервер не ляжет под нагрузкой в сотни или тысячи одновременных запросов. Это еще одна непростая область, которая может стать темой для целой книги: стрессовое тестирование может быть сколь угодно запутанным, а насколько именно, зависит от природы вашего проекта. Если у вас есть основания полагать, что сайт может оказаться чрезвычайно популярным, возможно, вам стоит уделить больше внимания стрессовому тестированию.

Добавим простой тест с использованием Artillery (https://artillery.io/). Сначала установите Artillery, выполнив `npm install -g artillery`; затем отредактируйте файл `package.json` и добавьте следующее в раздел `scripts`:

```
"scripts": {
  "stress": "artillery quick --count 10 -n 20 http://localhost:3000/"
}
```

Это эмуляция десяти виртуальных пользователей (`--count 10`), каждый из которых будет отправлять по 20 запросов (`-n 20`) к вашему серверу.

Удостоверьтесь, что ваше приложение запускается (например, в отдельном окне терминала), и затем выполните `npm run stress`. Вы увидите статистику, подобную этой:

```
Начальная фаза 0, длительность: 1с @ 16:43:37(-0700) 2019-04-14
Отчет @ 16:43:38(-0700) 2019-04-14
Продолжительность: 1 секунда
  Запущено скриптов: 10
  Выполнено скриптов: 10
  Выполнено запросов: 200
  В течение секунды отправлено запросов (RPS): 147.06
  Величина задержки:
    минимальная: 1.8
    максимальная: 10.3
    средняя: 2.5
    p95: 4.2
    p99: 5.4
  Коды:
    200: 200

Все виртуальные пользователи закончили работу
Итоговый отчет @ 16:43:38(-0700) 2019-04-14
  Запущено скриптов: 10
  Выполнено скриптов: 10
  Выполнено запросов: 200
  В течение секунды отправлено запросов (RPS): 145.99
  Величина задержки:
    минимальная: 1.8
    максимальная: 10.3
    средняя: 2.5
    p95: 4.2
    p99: 5.4
  Статистика скриптов:
    0: 10 (100%)
  Коды:
    200: 200
```

Этот тест был запущен на моем рабочем ноутбуке. Как видите, Express потребовалось менее 10,3 миллисекунды на обслуживание запросов и 99 % из них было обслужено менее чем за 5,4 миллисекунды. Я не могу точно сказать, на какие числа вам следует обращать внимание, но, чтобы приложение было быстрым, суммарное время соединений должно быть менее 50 миллисекунд. (Не забывайте, что это всего лишь время, которое сервер затрачивает на доставку данных клиенту; клиент должен еще их отобразить, а это продолжительный процесс, таким образом, чем меньше времени занимает передача данных, тем лучше.)

Если вы будете регулярно проводить стрессовое тестирование вашего приложения и сравнивать результаты с образцом, то сможете выявлять проблемы. Если вы только что закончили писать какой-нибудь функционал и увидели, что время соединения увеличилось втрое, то, вероятно, захотите настроить производительность этого функционала!

Резюме

Надеюсь, прочитав эту главу, вы поняли, о каких вещах следует подумать перед запуском приложения в эксплуатацию. Здесь приведено много деталей, касающихся промышленно эксплуатируемых приложений, и, хотя вы не можете предугадать все форс-мажоры, которые могут возникнуть при запуске, чем более предусмотрительными вы будете, тем лучше. Как говорил Луи Пастер, «в области наблюдения удача сопутствует только подготовленному уму».

13 Персистентность данных

Всем сайтам и веб-приложениям, кроме простейших, требуется *сохраняемость* (persistence), то есть какой-то способ более постоянного хранения данных, чем энергозависимая память, чтобы данные не пропали при сбоях сервера, перебоях с электричеством, обновлениях и переездах. В данной главе мы обсудим возможные варианты хранения данных и рассмотрим как документоориентированные, так и реляционные базы данных. Прежде чем приступить к базам данных, начнем с базовой формы персистентности — хранения данных в файловой системе.

Хранение данных в файловой системе

Один из способов достижения сохраняемости — простое хранение данных в так называемых неструктурированных файлах (*неструктурированные* они потому, что в таких файлах нет внутренней структуры, это обычная последовательность байтов). Node позволяет хранить данные в файловой системе посредством модуля `fs` (file system — «файловая система»).

У хранения данных в файловой системе есть свои недостатки. В частности, оно плохо масштабируется. В тот момент, когда потребуется более одного сервера для обработки возросшего количества трафика, вы столкнетесь с проблемами хранения данных в файловой системе. Исключением может стать ситуация, когда все ваши серверы имеют доступ к общей файловой системе. Поскольку в неструктурированных файлах нет внутренней структуры, вся тяжесть работ по нахождению, сортировке и фильтрации данных ляжет на приложение. В силу этих причин для хранения данных лучше использовать базы данных, а не файловые системы. Единственное исключение — хранение двоичных файлов, таких как изображения, звуковые или видеофайлы. Хотя многие базы данных умеют работать с этим типом данных, они редко делают это эффективнее файловой системы (хотя информация о двоичных файлах обычно хранится в базе данных для обеспечения возможности поиска, сортировки и фильтрации).

Если нужно хранить двоичные данные, помните о наличии у хранилища на основе файловой системы проблем с масштабированием. Если у вашего хостинг-

сервиса нет доступа к совместно используемой файловой системе (а обычно так и бывает), стоит рассмотреть возможность хранения двоичных файлов в базе данных (чаще всего это требует настройки, чтобы база данных не начала сильно тормозить вплоть до полной остановки) или в облачном хранилище, таком как Amazon S3 или Microsoft Azure Storage.

Теперь, когда мы закончили с предупреждениями, взглянем на имеющуюся у Node поддержку файловой системы. Обратимся снова к отпускному фотоконкурсу из главы 8. Заменим в файле приложения обработчик формы (`ch13/00-mongodb/lib/handlers.js` в прилагаемом репозитории):

```
const pathUtils = require('path')
const fs = require('fs')

// Создаем каталог для хранения отпускных фото
// (если он еще не существует).
const dataDir = pathUtils.resolve(__dirname, '..', 'data')
const vacationPhotosDir = pathUtils.join(dataDir, 'vacation-photos')
if(!fs.existsSync(dataDir)) fs.mkdirSync(dataDir)
if(!fs.existsSync(vacationPhotosDir)) fs.mkdirSync(vacationPhotosDir)

function saveContestEntry(contestName, email, year, month, photoPath){
  // TODO... это будет добавлено позже
}

// Эти основанные на промисах версии функций
// файловой системы понадобятся нам позже.
const { promisify } = require('util')
const mkdir = promisify(fs.mkdir)
const rename = promisify(fs.rename)

exports.api.vacationPhotoContest = async (req, res, fields, files) => {
  const photo = files.photo[0]
  const dir = vacationPhotosDir + '/' + Date.now()
  const path = dir + '/' + photo.originalFilename
  await mkdir(dir)
  await rename(photo.path, path)
  saveContestEntry('vacation-photo', fields.email,
    req.params.year, req.params.month, path)
  res.send({ result: 'success' })
}
```

Здесь происходит множество различных событий, поэтому рассмотрим все по частям. Сначала создаем каталог для хранения загруженных на сервер файлов (если его еще нет). Вероятно, вы захотите добавить каталог `data` в файл `.gitignore`, чтобы случайно не внести загруженные на сервер файлы в репозиторий. Как вы помните из главы 8, мы обрабатываем действительную загрузку файлов в `meadowlark.js` и вызываем обработчик с уже декодированными файлами. То, что мы получаем, — это объект (`files`), содержащий информацию о загруженных файлах. В целях предотвращения противоречий мы не можем просто

использовать имена файлов, заданные пользователем (на тот случай, когда два пользователя загрузят `portland.jpg`). Во избежание этой проблемы создаем уникальный каталог на основе временной отметки; весьма маловероятно, что оба пользователя загрузят `portland.jpg` в одну и ту же миллисекунду! Затем мы переименовываем (перемещаем) загруженный файл (файловый процессор даст ему временное имя, которое мы можем получить из свойства `path`) в созданное нами имя.

Наконец, нам нужен способ для связи загруженных пользователями файлов с их адресами электронной почты (а также месяцем и годом отправки). Мы можем закодировать эту информацию в имени файла или каталога, но лучше хранить ее в базе данных. Поскольку мы еще не научились это делать, то инкапсулируем эту функциональность в функции `vacationPhotoContest` и закончим данную функцию позже в этой главе.

Ни в коем случае не доверяйте тому, что загружает пользователь, поскольку это может оказаться способом атаки на ваш сайт. Например, злонамеренный пользователь может легко создать вредоносный исполняемый файл, изменить его расширение на .jpg и загрузить в качестве первого шага атаки (в надежде найти какой-то способ запустить его позже). Аналогично мы рискуем, называя файл в соответствии со свойством name, предоставляемым браузером; кто-нибудь может злоупотребить этим, вставив в имя файла специальные символы. Чтобы сделать код максимально защищенным, нам следовало бы дать файлу случайное имя, оставив только расширение и убедившись, что оно состоит лишь из буквенно-цифровых символов.

Несмотря на недостатки, хранение данных в файловой системе часто используется для промежуточных файловых хранилищ, поэтому полезно уметь пользоваться библиотекой Node для файловой системы. Впрочем, чтобы избежать недостатков хранилищ с файловой системой, обратимся к хранению данных в облаке.

Хранение данных в облаке

Облачные хранилища становятся все более популярными, и я советую вам воспользоваться одним из этих недорогих и устойчивых к ошибкам сервисов.

Намерившись использовать облачный сервис, нужно проделать определенную предварительную работу. Очевидно, необходимо создать аккаунт, но вы должны понимать, как облачный сервер аутентифицирует приложение и некоторую базовую терминологию (например, AWS называет свой механизм файловых хранилищ бакетами, тогда как Azure использует для них название *«контейнеры»*). Подробная информация об этом выходит за пределы данной книги, но по ней есть хорошая документация.

- AWS: основы Node.js (https://amzn.to/2CCYk9s).
- Azure для разработчиков JavaScript и Node.js (http://bit.ly/2NEkTku).

Хорошая новость в том, что как только вы завершите эту первоначальную конфигурацию, использовать облачные хранилища будет очень легко. Вот пример того, как просто сохранить файл в аккаунте Amazon S3:

```
const filename = 'customerUpload.jpg'

s3.putObject({
  Bucket: 'uploads',
  Key: filename,
  Body: fs.readFileSync(__dirname + '/tmp/ + filename),
})
```

Смотрите документацию AWS SDK (http://aws.amazon.com/sdkfornodejs) для получения более подробной информации.

И пример того, как выполнить то же самое с помощью Microsoft Azure:

```
const filename = 'customerUpload.jpg'

const blobService = azure.createBlobService()
blobService.createBlockBlobFromFile('uploads', filename, __dirname +
  '/tmp/' + filename)
```

Смотрите документацию Microsoft Azure (http://bit.ly/2Kd3rRK) для получения более подробной информации.

Теперь, когда мы узнали несколько методов хранения файлов, рассмотрим хранение структурированных данных в базах данных.

Хранение данных в базе данных

Всем, кроме простейших сайтов и веб-приложений, требуется база данных. Даже если большая часть ваших данных — двоичные и вы используете общую файловую систему или облачное хранилище, вероятно, вам понадобится база данных для каталогизации этих двоичных данных.

Традиционно под *базой данных* понимается *реляционная система управления базами данных, РСУБД* (relational database management system, RDBMS). Реляционные базы данных, такие как Oracle, MySQL, PostgreSQL или SQL Server, основываются на формальной теории баз данных и десятилетиях исследований. На сегодня это вполне зрелая технология, и высокая производительность этих баз данных несомненна. Однако теперь мы можем позволить себе более широкую трактовку понятия базы данных. В последние годы в моду вошли NoSQL-базы данных, оспаривающие расстановку вещей в области хранения данных для Интернета.

Было бы глупо утверждать, что NoSQL-базы данных чем-то лучше реляционных, но у них действительно есть определенные преимущества перед реляционными и наоборот. Хотя интегрировать реляционную базу данных с приложениями Node довольно легко, существуют NoSQL-базы данных, которые будто специально созданы для Node.

Два наиболее распространенных типа NoSQL-баз данных: *документоориентированные* базы данных и базы данных *«ключ — значение»*. Документоориентированные базы лучше подходят для хранения объектов, что делает их естественным дополнением Node и JavaScript. Базы данных «ключ — значение», как понятно из названия, исключительно просты и представляют собой отличный выбор для приложений, схемы данных которых соответствуют парам «ключ — значение».

Мне кажется, что документоориентированные базы данных — оптимальный компромисс между ограничениями реляционных баз данных и простотой баз данных «ключ — значение», поэтому мы будем использовать их для наших примеров. MongoDB — ведущая документоориентированная база данных, общепризнанная и очень надежная.

Во втором примере мы будем использовать PostgreSQL — популярную и устойчивую к ошибкам РСУБД с открытым исходным кодом.

Замечания относительно производительности

Простота баз данных NoSQL — палка о двух концах. Тщательное проектирование реляционной базы данных может оказаться сложной задачей, но результат оправдает ожидания: вы получите базу данных с великолепной производительностью. Не обманывайте себя, думая, что из-за того, что базы данных NoSQL, как правило, проще реляционных, их настройка на максимальную производительность не представляет никакого труда.

Для достижения высокой производительности реляционные базы данных обычно полагаются на жесткие структуры данных и десятилетия исследований в области оптимизации. В то же время базы данных NoSQL учли распределенную природу сети Интернет и, подобно Node, сосредоточились на параллелизме для масштабирования производительности (реляционные базы данных также поддерживают параллелизм, но он обычно приберегается для наиболее требовательных приложений).

Проектирование баз данных для масштабирования и высокой производительности — обширная сложная тема, выходящая за пределы данной книги. Если вашему приложению требуется высокий уровень производительности базы данных, рекомендую начать с книги Кристины Ходоров и Майкла Диролфа *MongoDB: The Definitive Guide*[1] (http://bit.ly/Mongo_DB_Guide).

Абстрагирование слоя базы данных

В этой книге мы продемонстрируем использование двух разных баз данных (а если точнее, двух существенно отличающихся архитектур баз данных) в процессе реализации одной и той же функциональности. Хотя цель этой книги — описание двух популярных вариантов архитектур баз данных, мы рассмотрим реальный сценарий: смену основного компонента веб-приложения в процессе проектирования. Это может произойти по многим причинам. Как правило, все сводится к тому, что

[1] *Chodorow K., Dirolf M.* MongoDB: The Definitive Guide. — O'Reilly, 2019.

другая технология будет менее затратной или позволит реализовать необходимую функциональность быстрее.

Имеет смысл по возможности *абстрагировать* выбранные вами технологии, что подразумевает написание своего рода слоя API, обобщающего эти технологии. Если сделать это правильно, то снизятся затраты на отказ от компонента, о котором идет речь. Тем не менее за это все равно придется платить: слой абстракций — вещь, которую нужно писать и поддерживать.

К счастью, наш слой абстракций будет очень маленьким, поскольку в рамках этой книги нужно поддерживать только несколько функций. На данный момент функции следующие.

- Возврат списка действующих отпускных туров из базы данных.
- Хранение адресов электронной почты пользователей, которые хотят получать оповещения о начале определенных туров в течение сезона.

Кажется, что все это достаточно просто, однако здесь нужно учитывать много деталей. Что представляет собой отпускной тур? Всегда ли мы хотим получать оповещения обо всех турах из базы данных, или нам необходима возможность применять фильтры или разбивку на страницы? Как мы идентифицируем туры? И так далее.

В рамках нашей книги слой абстракций будет простым. Мы поместим его в файл `db.js`, из которого будут экспортироваться два метода, для начала реализованные как макеты:

```
module.exports = {
  getVacations: async (options = {}) => {
    // Начальные данные отпускного тура:
    const vacations = [
      {
        name: 'Однодневный тур в Худ-Ривер',
        slug: 'hood-river-day-trip',
        category: 'Однодневный тур',
        sku: 'HR199',
        description: 'Проведите день в плавании по реке ' +
          'Колумбия и насладитесь сваренным ' +
          'по традиционным рецептам пивом в Худ-Ривер!',
        location: {
          // мы будем использовать это для
          // геокодирования позже.
          search: 'Худ-Ривер, Орегон, США',
        },
        price: 99.95,
        tags: ['однодневный тур', 'худ-ривер', 'плавание',
               'виндсерфинг', 'пивоварни'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
      }
    ]
```

```
    // Если определена опция "available" (доступно),
    // возвращать только соответствующие туры.
    if(options.available !== undefined)
    return vacations.filter(({ available }) => available === options.available)
    return vacations
  },
  addVacationInSeasonListener: async (email, sku) => {
    // Мы только притворимся, что делаем это...
    // Поскольку это асинхронная функция, автоматически
    // будет возвращен новый промис,
    // исполняющийся со значением undefined.
  },
}
```

Этим определяется ожидание того, какой должна быть реализация нашей базы данных с точки зрения приложения, и все, что нам нужно, — это спроектировать базы данных в соответствии с этим интерфейсом. Заметьте, что мы вводим понятие «доступности» отпускного тура. Это необходимо, чтобы в определенный момент на некоторое время можно было легко заблокировать туры, а не удалять их из базы данных. Пример использования — сообщение о том, что номера B&B («кровать и завтрак») будут недоступны в течение нескольких месяцев в связи с перепланировкой. Мы не смешиваем происходящее с понятием сезонности, потому что можем перечислить на веб-сайте несезонные отпускные туры, ведь многим людям нравится планировать заранее.

Мы также включаем некоторую очень общую информацию о местоположении; более подробно рассмотрим это в главе 19.

Теперь, когда мы заложили основы абстракции слоя базы данных, взглянем, как можно реализовать хранение данных с помощью MongoDB.

Установка и настройка MongoDB

Степень сложности установки и настройки экземпляра MongoDB зависит от используемой вами операционной системы. Поэтому мы вообще обойдем эту проблему путем использования замечательного бесплатного хостинг-сервиса MongoDB — mLab.

mLab не единственный доступный сервис MongoDB. На данный момент компания MongoDB предлагает через свой продукт MongoDB Atlas (https://www.mongodb.com/) бесплатный и низкозатратный хостинг баз данных. Однако при промышленной эксплуатации не стоит использовать бесплатные учетные записи. Как mLab, так и MongoDB Atlas предлагают подходящие для промышленной эксплуатации планы, так что вам следует изучить их расценки, прежде чем выбирать. Будет проще оставаться на том же хостинг-сервисе при переходе к эксплуатации.

Начать работать с mLab несложно: просто перейдите по адресу https://mlab.com и нажмите на Sign Up (Зарегистрироваться). Заполните регистрационную форму,

войдите на сайт — и окажетесь на главной странице. Под надписью Databases увидите сообщение: no databases at this time (в настоящее время базы данных отсутствуют). Нажмите Create new (Создать новую) — и будете перемещены на страницу с опциями для новой базы данных. Первым делом нужно выбрать провайдера облачных услуг. Для бесплатной (песочница) учетной записи выбор практически не имеет значения, хотя вам стоит остановиться на центре обработки данных, расположенном рядом с вами (но не каждый центр обработки данных будет предоставлять учетные записи-песочницы). Выберите SANDBOX (Песочница) и регион. Затем выберите имя базы данных и нажмите на Submit Order (Отправить заказ) (это все еще заказ, хотя и бесплатный!). Вы вернетесь к списку баз данных, и через несколько секунд ваша база данных будет готова к использованию.

Настройка базы данных — это лишь полдела. Теперь нужно узнать, как получить к ней доступ с помощью Node, и тут на сцену выходит Mongoose.

Mongoose

Несмотря на то что для MongoDB (http://bit.ly/2Kfw0hE) существует низкоуровневый драйвер, вы, вероятно, пожелаете использовать объектно-документное отображение (object document mapper, ODM). Официально поддерживаемым ODM MongoDB является Mongoose.

Одно из преимуществ JavaScript — исключительная гибкость его объектной модели. Если вам нужно добавить в объект свойство или метод, вы просто добавляете его, не беспокоясь о необходимости модифицировать класс. К сожалению, такая разновидность гибкости способна негативно сказаться на ваших базах данных, поскольку они могут становиться фрагментированными и трудно оптимизируемыми. Mongoose пытается найти компромисс: он представляет *схемы* и *модели* (взятые вместе, схемы и модели подобны классам в обычном объектно-ориентированном программировании). Схемы довольно гибки, но тем не менее обеспечивают необходимую для вашей базы данных структуру.

Прежде чем начать, нужно установить модуль Mongoose:

```
npm install mongoose
```

Затем добавляем данные доступа к нашей базе данных в файл `.credentials.development.json`:

```
  "mongo": {
    "connectionString": "your_dev_connection_string"
  }
}
```

Строку подключения вы найдете на странице базы данных в mLab. На главной странице щелкните на соответствующей базе данных. Появится блок с URI подключения к MongoDB (он начинается с mongodb://). Вам также понадобится пользователь для базы данных. Чтобы создать его, щелкните на Users (Пользователи), а затем на Add database user (Добавить пользователя базы данных).

Обратите внимание, что можно определить второй набор данных доступа для эксплуатации, создав файл `.credentials.production.js` с `NODE_ENV=production`; это нужно будет сделать, когда придет время запустить проект!

Теперь, когда мы завершили настройку, давайте создадим настоящее подключение к базе данных и сделаем что-нибудь полезное!

Подключение к базе данных с помощью Mongoose

Начнем с создания подключения к нашей базе данных. Мы поместим в `db.js` код инициализации базы данных и фиктивный API, созданный нами ранее (`ch13/00-mongodb/db.js` в прилагаемом репозитории):

```
const mongoose = require('mongoose')
const { connectionString } = credentials.mongo
if(!connectionString) {
  console.error('Отсутствует строка подключения к MongoDB!')
  process.exit(1)
}

mongoose.connect(connectionString)
const db = mongoose.connection
db.on('error' err => {
  console.error('Ошибка MongoDB: ' + err.message)
  process.exit(1)
})
db.once('open', () => console.log('Установлено соединение с MongoDB'))

module.exports = {
  getVacations: async () => {
    //...возврат начальных данных отпускного тура.
  },
  addVacationInSeasonListener: async (email, sku) => {
    //...ничего не делать.
  },
}
```

Любой файл, которому нужно получить доступ к базе данных, может просто импортировать `db.js`. Тем не менее мы хотим, чтобы инициализация произошла немедленно, до того как нам понадобится API, поэтому возьмем это на себя и произведем импорт из `meadowlark.js` (где нам ничего не нужно делать с API):

```
require('./db')
```

Подключение к базе данных установлено, пришло время задуматься о том, как мы собираемся структурировать данные, передаваемые в базу данных и выгружаемые из нее.

Создание схем и моделей

Создадим базу отпускных туров для Meadowlark Travel. Начнем с описания схемы и создания на ее основе модели. Создайте файл `models/vacation.js` (`ch13/00-mongodb/models/vacation.js` в прилагаемом репозитории):

```
const mongoose = require('mongoose')

const vacationSchema = mongoose.Schema({
  name: String,
  slug: String,
  category: String,
  sku: String,
  description: String,
  location: {
    search: String,
    coordinates: {
      lat: Number,
      lng: Number,
    },
  },
  price: Number,
  tags: [String],
  inSeason: Boolean,
  available: Boolean,
  requiresWaiver: Boolean,
  maximumGuests: Number,
  notes: String,
  packagesSold: Number,
})

const Vacation = mongoose.model('Vacation', vacationSchema)
module.exports = Vacation
```

Этот код объявляет свойства, составляющие нашу модель отпуска, и типы этих свойств. Как видите, здесь есть несколько свойств со строковым типом данных, несколько численных и два булевых свойства, а также массив строк, обозначенный `[String]`. На этой стадии мы также определяем методы, принадлежащие схеме. У каждого продукта есть единица хранения (stock keeping unit, SKU); хотя мы и не рассматриваем отпускные туры как нечто хранящееся на складе, концепция SKU стандартна для бухгалтерского учета даже тогда, когда продаются нематериальные товары.

Как только у нас появляется схема, создаем модель с помощью `mongoose.model`: на данной стадии `Vacation` весьма напоминает класс в обычном объектно-ориентированном программировании. Обратите внимание, что нам необходимо определить методы, прежде чем создадим нашу модель.

Из-за природы чисел с плавающей запятой вам следует быть внимательными при финансовых вычислениях в JavaScript. Запись цен в центах, а не в долларах помогает, однако не устраняет проблему целиком. Мы не будем углубляться в эту тему ради скромных целей нашего сайта агентства путешествий. Но если в вашем приложении используются очень большие или очень маленькие денежные суммы (например, дробные части центов для объемов торгов и количества открытых позиций), рассмотрите такие библиотеки, как currency.js (https://currency.js.org/) или decimal.js-light (http://bit.ly/2X6kbQ5). Помимо этого, можно использовать встроенный в JavaScript объект BigInt (https://mzl.la/2Xhs45r), доступный в Node 10 (с ограниченной браузерной поддержкой на момент написания этих строк).

Экспортируем созданный Mongoose объект модели `Vacation`. Мы можем непосредственно использовать данную модель, но это сделает напрасными наши усилия по предоставлению абстрактного слоя базы данных. Поэтому выбираем импорт только из файла `db.js` и разрешаем остальной части приложения использовать его методы. Добавьте модель `Vacation` в `db.js`:

```
const Vacation = require('./models/vacation')
```

Мы определили все структуры, но база данных не вызывает большого интереса, поскольку в ней на самом деле ничего нет. Сделаем ее полезной, внеся в нее какие-нибудь начальные данные.

Определение начальных данных

В нашей базе данных пока еще нет отпускных туров, так что для начала добавим парочку. Со временем, возможно, у вас появится необходимость управлять продуктами, но в рамках данной книги мы просто выполним все это в коде (`ch13/00-mongodb/db.js` в прилагаемом репозитории):

```
Vacation.find((err, vacations) => {
  if(err) return cosole.error(err)
  if(vacations.length) return

  new Vacation({
    name: 'Однодневный тур в Худ-Ривер',
    slug: 'hood-river-day-trip',
    category: 'Однодневный тур',
    sku: 'HR199',
    description: 'Проведите день в плавании по реке ' +
      'Колумбия и насладитесь сваренным ' +
      'по традиционным рецептам пивом в Худ-Ривер!',
    location: {
      search: 'Худ-Ривер, Орегон, США',
    },
    price: 99.95,
```

```
    tags: ['однодневный тур', 'худ-ривер', 'плавание', 'виндсерфинг',
           'пивоварни'],
    inSeason: true,
    maximumGuests: 16,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
    name: 'Отдых в Орегон Коуст',
    slug: 'oregon-coast-getaway',
    category: 'Отдых на выходных',
    sku: 'OC39',
    description: 'Насладитесь океанским воздухом ' +
       'и причудливыми прибрежными городками!',
    location: {
      search: 'Кэннон Бич, Орегон, США',
    },
    price: 269.95,
    tags: ['отдых на выходных', 'орегон коуст',
           'прогулки по пляжу'],
    inSeason: false,
    maximumGuests: 8,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
    name: 'Скалолазание в Бенде',
    slug: 'rock-climbing-in-bend',
    category: 'Приключение',
    sku: 'B99',
    description: 'Пощекочите себе нервы горным ' +
      'восхождением на пустынной возвышенности.',
    location: {
      search: 'Бенд, Орегон, США',
    },
    price: 289.95,
    tags: ['отдых на выходных', 'бенд', 'пустынная
            возвышенность', 'скалолазание'],
    inSeason: true,
    requiresWaiver: true,
    maximumGuests: 4,
    available: false,
    packagesSold: 0,
    notes: 'Гид по данному туру в настоящий момент ' +
           'восстанавливается после лыжной травмы.',
  }).save()
})
```

Здесь используются два метода Mongoose. Первый, `find` (искать), выполняет ровно то, о чем говорит его название. В данном случае он находит все экземпляры `Vacation` в базе данных и выполняет обратный вызов с этим списком. Мы так поступаем, поскольку не желаем продолжать чтение первоначально заданных отпускных туров: если в базе данных уже есть отпускные туры, то первичное ее заполнение было выполнено и мы спокойно можем идти дальше. Однако при первом своем выполнении `find` вернет пустой список, так что мы переходим к созданию двух отпускных туров, а затем вызываем для них метод `save`, сохраняющий новые объекты в базе данных.

Теперь, когда в базе данных есть данные, пришло время получить их обратно!

Извлечение данных

Мы рассмотрели метод `find`, используемый для отображения списка отпускных туров. Теперь же хотим передать функции `find` параметр, на основе которого будет выполняться фильтрация данных, а именно отобразятся только доступные в настоящий момент отпускные туры.

Создайте представление для страницы с продуктами, `views/vacations.handlebars`:

```
<h1>Отпускные туры</h1>
{{#each vacations}}
  <div class="vacation">
    <h3>{{name}}</h3>
    <p>{{description}}</p>
    {{#if inSeason}}
      <span class="price">{{price}}</span>
      <a href="/cart/add?sku={{sku}}" class="btn btn-default">Купить сейчас!</a>
    {{else}}
      <span class="outOfSeason">К сожалению, в настоящий момент несезон
          для этого тура.
      {{! Страница "сообщите мне, когда наступит сезон для этого тура"
          станет нашей следующей задачей. }}
      <a href="/notify-me-when-in-season?sku={{sku}}">Сообщите мне, когда
          наступит сезон для этого тура.</a>
    {{/if}}
  </div>
{{/each}}
```

Теперь мы можем создать обработчики маршрутов, которые свяжут все это вместе. В файле `lib/handlers.js` (не забудьте импортировать `../db`) мы создадим обработчик:

```
exports.listVacations = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  const context = {
    vacations: vacations.map(vacation => ({
```

```
        sku: vacation.sku,
        name: vacation.name,
        description: vacation.description,
        price: '$' + vacation.price.toFixed(2),
        inSeason: vacation.inSeason,
    }))
  }
  res.render('vacations', context);
}
```

Мы добавим маршрут для вызова обработчика в файл `meadowlark.js`:

```
app.get('/vacations', handlers.listVacations)
```

Если вы запустите выполнение этого примера, то увидите лишь отпускные туры из нашей фиктивно реализованной базы данных. Это объясняется тем, что мы инициализировали базу данных и внесли начальные данные, но не заменили фиктивную реализацию на настоящую. Так давайте сделаем это. Откройте файл `db.js` и преобразуйте `getVacations`:

```
module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    //...
  },
}
```

Все достаточно просто — в одну строку! Это можно объяснить тем, что Mongoose выполняет много сложной работы за нас и мы спроектировали наш API так, как работает Mongoose. Когда позже адаптируете все под PostgreSQL, увидите, что придется немного больше потрудиться.

Проницательный читатель может обеспокоиться тем, что слой абстракции базы данных мало заботится о своей нейтральности с точки зрения технологий. Например, разработчик прочтет этот код и увидит, что отпускной модели можно передать опции Mongoose, и тогда приложение будет использовать функции, специфические для Mongoose, что сделает трудным переключение между базами данных. В наших силах сделать несколько шагов для предотвращения этого. Вместо того чтобы просто передавать все в Mongoose, мы можем найти специфические опции и обрабатывать их явно, ясно показывая, что любая реализация должна предоставлять эти опции. Но в данном примере мы закроем на это глаза и оставим код простым.

В основном вышеприведенный код должен казаться вам знакомым, но могут появиться и неожиданные моменты. Например, может показаться странным наш способ подготовки контекста представления для списка туров. Почему мы преобразовываем возвращенные из базы данных продукты в практически идентичный

объект? Одна из причин в том, что мы хотим правильно отображать отформатированную цену, поэтому приходится конвертировать ее в форматированную строку.

Можно сократить время, потраченное на набор, написав следующее:

```
const context = {
  vacations: products.map(vacations => {
    vacation.price = '$' + vacation.price.toFixed(2)
    return vacation
  })
}
```

Это определенно сэкономило бы несколько строк кода, но, как подсказывает опыт, существуют веские причины не передавать представлениям непреобразованные объекты базы данных. При этом представление получает массу, возможно, ненужных ему свойств в несовместимых с ним форматах. Наш пример пока довольно прост, но, когда он станет сложнее, вы, вероятно, захотите еще больше подогнать под конкретный случай передаваемые представлениям данные. Это увеличивает вероятность случайного раскрытия конфиденциальной информации или информации, которая может подорвать безопасность вашего сайта. По этим причинам я рекомендую преобразовывать возвращаемых из БД данных и передавать в представление только то, что требуется (при необходимости преобразовывая, как мы поступили с `price`).

В отдельных вариантах архитектуры MVC появляется третий компонент, называемый моделью представления. По сути, модель представления очищает модель (или модели) и преобразует ее наиболее подходящим для отображения в представлении образом. Фактически то, что мы здесь делаем, — это создание модели представления на лету.

Мы прошли долгий путь. Мы успешно используем базу данных для хранения информации об отпускных турах, но в базах данных мало проку, если мы не можем их обновлять. Обратим внимание на данный аспект взаимодействия с базами данных.

Добавление данных

Мы уже рассматривали, как можно добавлять (добавляли данные, когда первоначально задавали набор отпускных туров) и обновлять (обновляли количество проданных туристических пакетов при бронировании тура) данные, но теперь взглянем на несколько более сложный скрипт, подчеркивающий гибкость документоориентированных баз данных.

Когда для тура несезон, мы отображаем ссылку, призывающую покупателя получить оповещение о наступлении сезона. Осуществим привязку этой функциональности. Для начала создадим схему и модель (`models/vacationInSeasonListener.js`):

```
const mongoose = require('mongoose')

const vacationInSeasonListenerSchema = mongoose.Schema({
  email: String,
  skus: [String],
})
const VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
  vacationInSeasonListenerSchema)

module.exports = VacationInSeasonListener
```

Далее мы создадим представление `views/notify-me-when-in-season.handlebars`:

```
<div class="formContainer">
  <form class="form-horizontal newsletterForm" role="form"
        action="/notify-me-when-in-season" method="POST">
    <input type="hidden" name="sku" value="{{sku}}">
    <div class="form-group">
      <label for="fieldEmail" class="col-sm-2 control-label">
         Электронная почта
      </label>
      <div class="col-sm-4">
        <input type="email" class="form-control" required
          id="fieldEmail" name="email">
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-4">
        <button type="submit" class="btn btn-default">Отправить</button>
      </div>
    </div>
  </form>
</div>
```

И наконец, обработчики маршрутов:

```
exports.notifyWhenInSeasonForm = (req, res) =>
  res.render('notify-me-when-in-season', { sku: req.query.sku })

exports.notifyWhenInSeasonProcess = (req, res) => {
  const { email, sku } = req.body
  await db.addVacationInSeasonListener(email, sku)
  return res.redirect(303, '/vacations')
}
```

Наконец мы добавляем реальную реализацию в `db.js`:

```
const VacationInSeasonListener = require('./models/vacationInSeasonListener')

module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    await VacationInSeasonListener.updateOne(
      { email },
      { $push: { skus: sku } },
      { upsert: true }
    )
  },
}
```

Что за чудеса? Как мы можем обновлять запись в наборе `VacationInSeasonListener` до того, как он начнет существовать? Секрет заключается в удобном механизме Mongoose, носящем название `upsert` (комбинация английских слов update — «обновить» и insert — «вставить»). По сути, если записи с заданным адресом электронной почты не существует, она будет создана. Если же запись есть, она обновится. Затем мы используем магическую переменную `$push`, чтобы указать, что мы бы хотели добавить значение в массив.

Приведенный код не предотвращает добавления в запись нескольких SKU, если пользователь многократно заполняет форму. Когда наступит сезон для тура и мы будем искать пользователей, желающих получить оповещение, нам придется действовать аккуратно, чтобы не оповестить их несколько раз.

PostgreSQL

Объектно-ориентированные базы данных типа MongoDB прекрасны тем, что их можно быстро начать использовать. Но если вы хотите создать устойчивое к ошибкам приложение, то для структурирования объектно-ориентированной базы данных придется приложить столько же усилий — или даже больше, — как и на планирование традиционной реляционной базы данных. Более того, у вас может быть опыт использования реляционных баз данных, или вполне вероятно, что в вашем распоряжении есть реляционная база данных, к которой вы хотите подключиться.

К счастью, в экосистеме JavaScript найдется надежная поддержка для любой реляционной базы данных и при желании (или в случае необходимости) вы можете использовать реляционную базу данных без каких-либо проблем.

По аналогии с ODM, который мы использовали для MongoDB, для реляционных баз данных доступны инструменты объектно-реляционного отображения (object-relational mapping, ORM). Однако, поскольку большинство читателей,

интересующихся этой темой, уже, вероятно, знакомы с реляционными базами данных и SQL, мы будем пользоваться непосредственно клиентом Node PostgreSQL.

Как и с MongoDB, будем использовать бесплатный онлайн-сервис PostgreSQL. Если чувствуете уверенность, можете установить и настроить собственную базу данных PostgreSQL. Все, что изменится, — это строка подключения. Если вы пользуетесь собственным экземпляром PostgreSQL, удостоверьтесь, что это версия 9.4 или более поздняя, так как в версии 9.4 был введен тип данных JSON, который нам понадобится (на момент написания этих строк я использую версию 11.3).

Есть много вариантов PostgreSQL, доступных онлайн; в этом примере я буду пользоваться ElephantSQL (https://www.elephantsql.com/). Нет ничего проще, чем начать его использовать: создайте аккаунт (для авторизации подойдет ваша учетная запись в GitHub) и щелкните на Create New Instance (Создать новый экземпляр). Вам нужно только дать ему (экземпляру) название (например, `meadowlark`) и выбрать план (вы можете воспользоваться бесплатным планом). Вам также нужно указать регион (попытайтесь выбрать тот, который максимально близок к вам). Как только все настроено, вы найдете раздел Details (Подробности), где приведена информация о вашем экземпляре. Скопируйте URL (строку подключения), включающий имя пользователя, пароль и местоположение экземпляра — все в одной строке.

Поместите эту строку в файл `.credentials.development.json`:

```
"postgres": {
  "connectionString": "your_dev_connection_string"
}
```

Единственное различие между объектно-ориентированными базами данных и РСУБД в том, что при определении схемы последних проводится больше предварительных работ и до добавления и извлечения данных для создания схемы используется язык структурированных запросов SQL. В соответствии с данной парадигмой мы сделаем это отдельным шагом самостоятельно, вместо того чтобы позволить ODM или ORM сделать это за нас, как произошло в случае с MongoDB.

Мы можем создать сценарии SQL и использовать клиент командной строки для выполнения сценариев определения данных, которые создадут наши таблицы. Или можно выполнить эту работу в JavaScript с помощью клиентского API PostgreSQL, но за отдельный шаг, который выполняется только один раз. Поскольку данная книга — о Node и Express, мы сделаем последнее.

Сначала установите клиентскую библиотеку `pg` (`npm install pg`). Затем создайте файл `db-init.js`, который будет запускаться только для инициализации базы данных и отличается от файла `db.js`, используемого при каждом запуске сервера (`ch13/01-postgres/db.js` в прилагаемом репозитории):

```
const { credentials } = require('./config')

const { Client } = require('pg')
```

```
const { connectionString } = credentials.postgres
const client = new Client({ connectionString })

const createScript = `
  CREATE TABLE IF NOT EXISTS vacations (
    name varchar(200) NOT NULL,
    slug varchar(200) NOT NULL UNIQUE,
    category varchar(50),
    sku varchar(20),
    description text,
    location_search varchar(100) NOT NULL,
    location_lat double precision,
    location_lng double precision,
    price money,
    tags jsonb,
    in_season boolean,
    available boolean,
    requires_waiver boolean,
    maximum_guests integer,
    notes text,
    packages_sold integer
  );
`

const getVacationCount = async client => {
  const { rows } = await client.query('SELECT COUNT(*) FROM VACATIONS')
  return Number(rows[0].count)
}

const seedVacations = async client => {
  const sql = `
    INSERT INTO vacations(
      name,
      slug,
      category,
      sku,
      description,
      location_search,
      price,
      tags,
      in_season,
      available,
      requires_waiver,
      maximum_guests,
      notes,
      packages_sold
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14)
  `
```

```
  await client.query(sql, [
    'Однодневный тур в Худ-Ривер',
    'hood-river-day-trip',
    'Однодневный тур',
    'HR199',
    'Проведите день в плавании по реке Колумбия и насладитесь сваренным по
    традиционным рецептам пивом в Худ-Ривер!',
    'Худ-Ривер, Орегон, США',
    99.95,
    `["однодневный тур", "худ-ривер", "плавание", "виндсерфинг", "пивоварни"]`,
    true,
    true,
    false,
    16,
    null,
    0,
  ])
  // Здесь можно воспользоваться тем же шаблоном
  // для добавления информации о других турах…
}

client.connect().then(async () => {
  try {
    console.log('создание схемы базы данных')
    await client.query(createScript)
    const vacationCount = await getVacationCount(client)
    if(vacationCount === 0) {
      console.log('внесение начальных данных отпускных туров')
      await seedVacations(client)
    }
  } catch(err) {
    console.log('ОШИБКА: невозможно инициализировать базу данных')
    console.log(err.message)
  } finally {
    client.end()
  }
})
```

Начнем с конца файла. Мы вызываем метод `connect()` на объекте клиента базы данных (`client`), который устанавливает подключение и возвращает промис. После разрешения этого промиса можно производить действия с базой данных.

Первым делом мы вызываем `client.query(createScript)`, который создаст таблицу `vacations`. Если мы посмотрим на `createScript`, то увидим, что это описание данных языком SQL. Углубленное рассмотрение SQL выходит за границы данной книги, но если вы читаете этот раздел, то предполагается, что у вас есть хотя бы смутное представление о SQL. Вы могли заметить, что для названий полей используется змеиная нотация (snake_case) вместо верблюжьей (camelCase).

То есть `inSeason` стало `in_season`. Хотя в PostgreSQL возможно использование верблюжьей нотации для именования структур, вам нужно заключать в кавычки любые идентификаторы, которые начинаются с заглавных букв, что доставляет больше хлопот, чем пользы. Мы вернемся к этому немного позже.

Как видите, нам уже нужно хорошенько обдумать нашу схему. Насколько длинным может быть название тура? (Здесь мы ограничиваемся произвольным числом 200 символов.) Какой длины могут быть названия категорий и SKU? Обратите внимание: для цен мы используем тип `money` из PostgreSQL и сделали `slug` первичным ключом (вместо добавления отдельного ID).

Если вы уже знакомы с реляционными базами данных, то для вас в этой простой схеме не будет ничего нового. Однако вам может броситься в глаза то, как мы поступили с данными в колонке `tags`.

При традиционном проектировании баз данных мы, вероятно, создали бы новую таблицу для отношений между отпускными турами и тегами (это называется *нормализацией*). Здесь также можно было бы сделать подобное, но именно в данном случае можно пойти на компромисс между традиционным проектированием реляционных баз данных и способом, принятым в JavaScript. Если бы у нас было две таблицы (например, `vacations` и `vacation_tags`), нам пришлось бы запрашивать данные из обеих, чтобы создать один объект, содержащий всю информацию об отпускном туре, как это было в примере для MongoDB. Эта дополнительная сложность может повлиять на производительность, но давайте предположим, что этого не случится и мы просто хотим иметь возможность быстро определять теги для конкретного тура. Можно сделать текстовое поле и разделить теги запятыми, но тогда придется анализировать теги, и в этом случае PostgreSQL предоставляет лучший способ — с помощью типов данных JSON. Скоро мы увидим, что, определяя их как JSON (jsonb, двоичное представление с, как правило, более высокой производительностью), можем хранить их как массив JavaScript и на выходе получим массив JavaScript, как это было в MongoDB.

Наконец, мы задаем начальные данные в базе данных с помощью той же базовой концепции, как и прежде: если таблица `vacations` пустая, то добавляем какие-нибудь начальные данные, иначе будем считать, что это уже сделано.

Вы заметите, что добавлять данные не так удобно, как это было с MongoDB. Существуют разные способы решения этой проблемы, но в этом примере я хочу использовать SQL явно. Мы могли бы написать функцию, чтобы сделать операторы ввода более естественными, или задействовать ORM (подробнее об этом — позже). Но на данный момент SQL делает свою работу, и это должно быть удобно для всех, кто уже знаком с SQL.

Несмотря на то что данный скрипт спроектирован по принципу разового запуска для инициализации и внесения начальных данных в базу данных, он написан так, что его безопасно запускать несколько раз. Мы включили опцию `IF NOT EXISTS` и проверяем, пуста ли таблица `vacations`, перед тем как добавить начальные данные.

Теперь мы можем запустить этот скрипт, чтобы инициализировать базу данных:

```
$ node db-init.js
```

База данных настроена, и можно написать код для ее использования на нашем веб-сайте.

Серверы баз данных обычно могут обрабатывать только ограниченное число подключений за раз. Поэтому веб-серверы, как правило, реализуют стратегию под названием *«организация пула подключений»* (connection pooling), чтобы компенсировать издержки на установление подключения с риском оставить соединения открытыми на слишком длительное время и «завалить» сервер. К счастью, все эти подробности обрабатываются клиентом PostgreSQL Node.

На этот раз мы воспользуемся немного другой стратегией для файла `db.js`. Теперь файл, который мы ранее подключали лишь с целью установления соединения с базой данных, будет возвращать написанный нами API, который скроет подробности коммуникации с базой.

Нам также нужно принять решение о модели `vacation`. Вспомните, когда мы создали модель, мы использовали змеиную нотацию для схемы базы данных, но во всем коде JavaScript — верблюжья нотация. У нас есть три варианта.

- Рефакторинг нашей схемы для использования верблюжьей нотации. Это сделает SQL уродливее, поскольку нужно помнить о том, чтобы брать в кавычки названия свойств.
- Использование змеиной нотации в JavaScript. Это далеко от идеала, ведь мы любим стандарты (правда?).
- Использование змеиной нотации на стороне базы данных и преобразование в верблюжью нотацию на стороне JavaScript. Это потребует больше работы, но SQL и JavaScript останутся безупречными.

К счастью, третий вариант можно реализовать автоматически. Мы можем написать собственную функцию для этого преобразования, но воспользуемся популярной библиотекой утилит Lodash, что чрезвычайно легко. Просто запустите `npm install lodash` для ее установки.

На данный момент потребности нашей базы данных очень скромны. Все, что нам нужно, — обратиться ко всем доступным пакетам туров, поэтому файл `db.js` будет выглядеть так (`ch13/01-postgres/db.js` в прилагаемом репозитории):

```
const { Pool } = require('pg')
const _ = require('lodash')

const { credentials } = require('./config')

const { connectionString } = credentials.postgres
const pool = new Pool({ connectionString })
```

```
module.exports = {
  getVacations: async () => {
    const { rows } = await pool.query('SELECT * FROM VACATIONS')
    return rows.map(row => {
      const vacation = _.mapKeys(row, (v, k) => _.camelCase(k))
      vacation.price = parseFloat(vacation.price.replace(/^\$/, ''))
      vacation.location = {
        search: vacation.locationSearch,
        coordinates: {
          lat: vacation.locationLat,
          lng: vacation.locationLng,
        },
      }
      return vacation
    })
  }
}
```

Коротко и ясно! Мы экспортируем единственный метод `getVacations`, который делает то, о чем говорилось. Он также использует функции `mapKeys` и `camelCase` из Lodash для преобразования свойств базы данных в верблюжью нотацию.

Единственное, что следует отметить, нужно быть аккуратными с атрибутом `price` (цена). Библиотека `pg` преобразовала тип `money` из PostgreSQL в уже отформатированную строку. И не без причины: как мы уже говорили, в JavaScript только недавно была добавлена поддержка численных типов для вычислений с произвольной точностью (`BigInt`), но еще нет адаптера PostgreSQL, который может этим воспользоваться (и этот тип данных не всегда самый эффективный). Мы могли бы изменить схему базы данных, чтобы использовать численный тип вместо типа `money`, но нам не стоит позволять тем выборам, которые мы сделали для клиентской части, определять схему. Мы также могли бы иметь дело с предварительно форматированными строками, возвращаемыми из `pg`, но тогда пришлось бы изменить весь существующий код, который полагается на то, что `price` — это число. Более того, этот подход не позволит производить численные вычисления в клиентской части (такие как суммирование цен на позиции в вашей корзине). В силу всех этих причин делаем выбор в пользу преобразования строки в число при извлечении из базы данных.

Кроме этого, преобразуем информацию о местоположении — которая хранится в таблице «плоско» — в более JavaScript-подобную структуру. Это делается только для того, чтобы добиться одинакового поведения с примером для MongoDB; мы можем использовать структурированные данные, как они есть (или преобразовать пример с MongoDB так, чтобы там была неформатированная структура).

Последнее, что нам нужно изучить, чтобы работать с PostgreSQL, — обновление данных, так что давайте реализуем функцию слушателя «сезонных туров».

Добавление данных

Мы будем использовать пример со слушателем «сезонных туров» так же, как и пример с MongoDB. Начнем с добавления следующего определения данных в строку createScript в `db-init.js`:

```
CREATE TABLE IF NOT EXISTS vacation_in_season_listeners (
  email varchar(200) NOT NULL,
  sku varchar(20) NOT NULL,
  PRIMARY KEY (email, sku)
);
```

Помните, что мы постарались написать `db-init.js` так, чтобы его можно было запустить в любой момент. Так что мы можем просто запустить его снова, чтобы создать таблицу `vacation_in_season_listeners`.

Теперь можно преобразовать `db.js`, чтобы включить метод для обновления этой таблицы:

```
module.exports = {
  //...
  addVacationInSeasonListener: async (email, sku) => {
    await pool.query(
      'INSERT INTO vacation_in_season_listeners (email, sku) ' +
      'VALUES ($1, $2) ' +
      'ON CONFLICT DO NOTHING',
      [email, sku]
    )
  },
}
```

Условие PostgreSQL `ON CONFLICT` фактически обеспечивает обновления и вставки. В данном случае, если точная комбинация адреса электронной почты и SKU уже присутствует, пользователь, которого нужно уведомить, уже подписался, поэтому нам ничего не нужно делать. Если бы в этой таблице были другие столбцы (например, дата последней регистрации), мы могли бы использовать более сложное условие `ON CONFLICT` (за более подробной информацией обращайтесь к документации PostgreSQL по `INSERT`). Заметьте, что это поведение также зависит от способа определения таблицы. Мы сделали `email` и `SKU` составными первичными ключами, что означает, что они не могут содержать повторяющиеся значения, что, в свою очередь, требует применинения условия `ON CONFLICT` (иначе в результате выполнения команды `INSERT` произойдет ошибка, когда пользователь во второй раз попытается подписаться на уведомление об одном и том же отпускном туре).

Теперь мы увидели полный пример подключения двух типов баз данных, объектно-ориентированной базы данных и РСУБД. Ясно, что функции базы данных

всегда одинаковые: хранение, извлечение и обновление данных непротиворечивым и масштабируемым образом. По этой причине мы смогли создать слой абстракций, чтобы можно было выбирать разные технологии баз данных. Последнее, для чего может понадобиться база данных, — это устойчивое хранение сеансов, на что я намекнул в главе 9.

Использование баз данных для хранения сеансов

Как обсуждалось в главе 9, использование хранилищ в памяти для данных сеансов не подходит для промышленной эксплуатации. К счастью, хранилищем сеансов может легко послужить база данных.

Хотя мы можем использовать существующие базы данных MongoDB или PostgreSQL, полнофункциональная база данных для хранения сеансов — явный перебор. В этом случае прекрасно подойдет база данных «ключ — значение». В момент написания этих строк наиболее популярными базами данных «ключ — значение» для хранилищ сеансов были Redis (https://redis.io/) и Memcached (https://memcached.org/). Чтобы оставаться верными другим примерам в этой главе, будем использовать бесплатный онлайн-сервис для предоставления базы данных Redis.

Для начала перейдите на сайт Redis Labs (https://redislabs.com/), создайте учетную запись, а затем бесплатную подписку и план. Для плана выберите Cache (Кэш) и назовите базу данных; остальные настройки можно оставить по умолчанию.

Вы попадете на страницу View Database (Показать базу данных) и, поскольку критическую информацию не заполнишь за пару секунд, сохраняйте спокойствие. Вам нужны поля Endpoint (Конечная точка) и Redis Password (Пароль Redis) под Access Control &Security (Контроль доступа и безопасность) (по умолчанию это скрыто, но будет показано после нажатия расположенной рядом маленькой кнопки). Возьмите эти данные и поместите их в файл `.credentials.development.json`:

```
"redis": {
  "url": "redis://:<ВАШ ПАРОЛЬ>@<ВАША КОНЕЧНАЯ ТОЧКА>"
}
```

Обратите внимание на немного странный URL: обычно перед двоеточием, что предшествует вашему паролю, находилось имя пользователя, но Redis позволяет подключаться с одним только паролем. Тем не менее двоеточие, разделяющее имя пользователя и пароль, все еще требуется.

Воспользуемся пакетом `connect-redis`, чтобы предоставить хранилище сеансов Redis. После установки (`npm install connect-redis`) его можно настроить в основном файле приложения. Мы по-прежнему используем `express-session`, но теперь передаем ему новое свойство `store`, которое настраивает его для использования базы данных. Заметьте, что нам нужно передать `expressSession` в функцию, возвращаемую из `connect-redis`, чтобы получить конструктор: эта странная особен-

ность хранилищ сеансов весьма распространена (`ch13/00-mongodb/meadowlark.js` или `ch13/01-postgres/meadowlark.js` в прилагаемом репозитории):

```
const expressSession = require('express-session')
const RedisStore = require('connect-redis')(expressSession)

app.use(cookieParser(credentials.cookieSecret))
app.use(expressSession({
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
  store: new RedisStore({
    url: credentials.redis.url,
    logErrors: true, // настоятельно рекомендуется!
  }),
}))
```

Постараемся использовать наше новоиспеченное хранилище сеансов для чего-то полезного. Представим, что нам нужна возможность отображать цены на туры в разных валютах. Более того, мы хотим, чтобы сайт запоминал предпочитаемые пользователями валюты.

Начнем с добавления окна справочника валют внизу страницы с отпускными турами:

```
<hr>
<p>Валюта:
  <a href="/set-currency/USD" class="currency {{currencyUSD}}">USD</a> |
  <a href="/set-currency/GBP" class="currency {{currencyGBP}}">GBP</a> |
  <a href="/set-currency/BTC" class="currency {{currencyBTC}}">BTC</a>
</p>
```

Теперь напишем немного CSS-кода (его можно встроить прямо в разметку `views/layouts/main.handlebars` или добавить ссылку на CSS-файл в каталоге `public`):

```
</p>a.currency {
  text-decoration: none;
}
.currency.selected {
  font-weight: bold;
  font-size: 150%;
}
```

Напоследок добавим обработчик маршрута для установки валюты и преобразуем обработчик маршрута для `/vacations`, чтобы цены отображались в текущей валюте (`ch13/00-mongodb/lib/handlers.js` или `ch13/01-postgres/lib/handlers.js` в прилагаемом репозитории):

```
exports.setCurrency = (req, res) => {
  req.session.currency = req.params.currency
  return res.redirect(303, '/vacations')
}
```

```
function convertFromUSD(value, currency) {
  switch(currency) {
    case 'USD': return value * 1
    case 'GBP': return value * 0.79
    case 'BTC': return value * 0.000078
    default: return NaN
  }
}

exports.listVacations = (req, res) => {
  Vacation.find({ available: true }, (err, vacations) => {
    const currency = req.session.currency || 'USD'
    const context = {
      currency: currency,
      vacations: vacations.map(vacation => {
        return {
          sku: vacation.sku,
          name: vacation.name,
          description: vacation.description,
          inSeason: vacation.inSeason,
          price: convertFromUSD(vacation.price, currency),
          qty: vacation.qty,
        }
      })
    }
    switch(currency){
      case 'USD': context.currencyUSD = 'selected'; break
      case 'GBP': context.currencyGBP = 'selected'; break
      case 'BTC': context.currencyBTC = 'selected'; break
    }
    res.render('vacations', context)
  })
}
```

Нам также нужно добавить маршрут с обработчиком установки валюты в файле `meadowlark.js`:

```
app.get('/set-currency/:currency', handlers.setCurrency)
```

Это, конечно, не самый лучший способ конвертации валют. Хотелось бы воспользоваться сторонним API, чтобы быть уверенными в актуальности курсов валют. Но того, что есть, достаточно для демонстрационных целей. Теперь вы можете переключаться между разными валютами и — забегите вперед и попробуйте — остановить, а также перезапустить сервер. Вы обнаружите, что выбранная вами валюта осталась запомненной! Если вы очистите cookie, предпочитаемая валюта будет забыта. Вы заметите, что красивое форматирование валюты исчезло; теперь восстановить это будет сложнее, и я оставлю данную задачу как упражнение для читателя.

Другое упражнение — сделать маршрут `set-currency` универсальным, чтобы от него было больше пользы. На текущий момент он всегда будет перенаправлять вас на страницу с турами, но что, если вы захотите воспользоваться им на странице корзины покупок? Посмотрим, может, вы додумаетесь до одного-двух способов решения этой задачи.

Если вы взглянете на базу данных, то найдете новую коллекцию `sessions`. Исследовав ее, обнаружите документ с вашим сеансовым ID (свойство `sid`) и предпочитаемой валютой.

Резюме

Определенно в этой главе мы рассмотрели много важного. Наличие базы данных делает полезными большинство веб-приложений. Проектирование и настройка баз данных — тема обширная, но я надеюсь, что дал вам основные инструменты, которые пригодятся для связывания двух типов баз данных и перемещения данных.

Теперь, зная эти основы, снова рассмотрим маршрутизацию и ее важную роль в веб-приложениях.

14 Маршрутизация

Маршрутизация — один из важнейших аспектов сайта или веб-сервиса. К счастью, маршрутизация в Express отличается простотой, гибкостью и устойчивостью к ошибкам. *Маршрутизация* — механизм, с помощью которого запросы (в соответствии с заданными URL и методом HTTP) находят путь к обрабатывающему их коду. Как я уже отмечал, маршрутизация обычно основывается на файлах и очень проста. К примеру, если вы помещаете на ваш сайт файл `foo/about.html`, то сможете получить доступ к нему через браузер по адресу `/foo/about.html`. Просто, но негибко. На случай, если вы не заметили, буквы HTML в URL стали признаком старомодности.

Перед тем как углубиться в технические стороны маршрутизации с помощью Express, следует обсудить понятие *информационной архитектуры* (information architecture, IA). Идея IA связана с понятийной организацией вашего контента. Наличие открытой (но не слишком усложненной) информационной архитектуры до того, как вы начнете обдумывать маршрутизацию, принесет огромную пользу в будущем.

Один из наиболее здравых и неустаревающих очерков на тему информационной архитектуры написан Тимом Бернерсом-Ли, человеком, который фактически изобрел Интернет. Вы можете и должны прочитать его не откладывая: http://www.w3.org/Provider/Style/URI.html. Очерк был написан в 1998 г. На минутку заглянем в него: не так уж много было написано в 1998 г. того, что ничуть не потеряло своей актуальности сейчас.

Согласно этому очерку, нам предлагается взять на себя огромную ответственность: «В обязанности веб-мастера входит выделение URI, которые должны оставаться актуальными на протяжении 2, 20 или даже 200 лет. Это требует внимательности, организованности и преданности делу» (Тим Бернерс-Ли).

Мне кажется, что, если бы для занятий веб-проектированием требовалась лицензия, как для других видов инженерной деятельности, мы бы приносили клятву в этом. (Внимательного читателя указанной статьи может позабавить тот факт, что ее URL заканчивается на `.html`.)

Можно провести аналогию (которую, увы, могут не понять молодые читатели), представив, что ваша любимая библиотека каждые два года полностью реорганизует универсальную десятичную классификацию (УДК). Зайдя однажды в би-

блиотеку, вы просто не сможете ничего найти. Именно это и происходит, когда вы перепроектируете структуру URL.

Серьезно обдумайте ваши URL: будут ли они по-прежнему иметь смысл через 20 лет (200 лет, возможно, перебор: кто знает, будем ли мы вообще использовать URL к тому времени. Тем не менее я восхищаюсь теми, кто заглядывает так далеко в будущее). Хорошо продумайте схему организации контента. Распределяйте все по логическим категориям и постарайтесь не загонять себя в глухой угол. Это одновременно наука и искусство.

И возможно, самое главное: работайте над проектированием своих URL вместе с другими людьми. Даже если вы лучший информационный архитектор в округе, вы можете удивиться, узнав, с каких разных точек зрения на один и тот же контент смотрят разные люди. Я не утверждаю, что вы должны создать информационную архитектуру, имеющую смысл с *любых* позиций, поскольку это нереально, но возможность увидеть проблему с нескольких точек зрения принесет новые идеи и выявит недостатки в вашей собственной информационной архитектуре.

Вот несколько советов, которые помогут вам создать долговечную информационную архитектуру.

- *Никогда не раскрывайте технические подробности в своих URL.* Случалось ли вам когда-нибудь заходить на сайт, замечать, что его URL заканчивается на `.asp`, и делать вывод, что этот сайт безнадежно устарел? Запомните: были времена, когда ASP считалась передовой технологией. Мне больно об этом говорить, но точно так же придет время упадка и JavaScript, и JSON, и Node, и Express. Будем надеяться, что это произойдет через много-много плодотворных для них лет, но время редко благосклонно к технологиям.
- *Избегайте бессмысленной информации в своих URL.* Тщательно обдумывайте каждое слово в URL. Если оно ничего не означает, выкиньте его. Например, меня всегда раздражает, когда сайты используют слово home в URL. Ваш корневой URL и есть ваша домашняя страница. Не нужно дополнительно делать URL, подобные `/home/directions` и `/home/contact`.
- *Не используйте без надобности длинные URL.* При прочих равных условиях короткий URL лучше длинного. Однако не пытайтесь сделать URL коротким в ущерб его понятности или SEO. Аббревиатуры заманчивы, но их нужно хорошо обдумывать: они должны быть общепринятыми и повсеместно распространенными, чтобы вы увековечили их в URL.
- *Будьте последовательны в использовании разделителей слов.* Распространенная практика — разделение слов дефисом и чуть менее распространенная — с помощью подчеркиваний. Дефисы обычно считаются более эстетически привлекательными, чем знаки подчеркивания, и большинство экспертов по SEO советуют использовать именно их. Но, что бы вы ни выбрали — дефисы или знаки подчеркивания, — будьте последовательны в их использовании.
- *Никогда не используйте пробел или непечатаемые символы.* Использовать пробел в URL не рекомендуется. Обычно он просто преобразуется в знак плюс (`+`),

что приводит к путанице. Очевидно, что следует избегать использования непечатаемых символов, и я бы рекомендовал не использовать любые символы, кроме буквенных, цифровых, тире и подчеркиваний. В момент разработки подобное может казаться интересным решением, но интересные решения, как правило, не выдерживают испытания временем. Понятно, что, если ваш сайт предназначен не для англоязычного круга пользователей, можете использовать отличные от латиницы символы (с помощью URL-кодов), хотя это может создать проблему при локализации сайта.

- *Используйте для своих URL нижний регистр*. Это правило часто вызывает споры: есть люди, считающие, что смешанный регистр в URL не только допустим, но и предпочтителен. Я не хочу начинать войну по этому поводу, замечу лишь, что преимущество нижнего регистра — в возможности его автоматической генерации. Если вам когда-то приходилось проходить по всему сайту, вручную придавая удобоваримый вид тысячам ссылок или выполняя строковые сравнения, вы оцените этот довод по достоинству. Лично я считаю URL в нижнем регистре более привлекательными, но решение, конечно, за вами.

Маршруты и SEO

Если вы хотите, чтобы ваш сайт легко находился в поисковых системах (а большинство людей хотят), необходимо подумать о поисковой оптимизации и о том, как URL на нее повлияют. В частности, если имеются отдельные особо важные ключевые слова — *и если это имеет смысл*, — рассмотрите возможность сделать их частями URL. Например, Meadowlark Travel предлагает несколько отпускных туров в «Орегон Коуст». Чтобы гарантировать высокий рейтинг в поисковых системах, мы используем «Орегон Коуст» (Oregon Coast) в заголовке, теле и описаниях метатегов, а URL начинается с `/vacations/oregon-coast`. Отпускной туристический пакет «Манзанита» можно найти по адресу `/vacations/oregon-coast/manzanita`. Если ради сокращения URL мы используем просто `/vacations/manzanita`, то потеряем важную SEO.

Однако не поддавайтесь искушению бездумно запихать в URL ключевые слова, пытаясь улучшить рейтинги, — это не сработает. Например, изменение URL туристического пакета «Манзаниты» на `/vacations/oregon-coast-portland-and-hood-river/oregon-coast/manzanita` в попытке лишний раз упомянуть «Орегон Коуст» (Oregon Coast), одновременно задействовав ключевые слова «Портланд» (Portland) и «Худ-Ривер» (Hood River), ошибочная стратегия. Она не считается с правилами хорошей информационной архитектуры и наверняка приведет к провальным результатам.

Поддомены

Как и путь, поддомены — еще одна часть URL, часто используемая для маршрутизации запросов. Лучше использовать поддомены для существенно различающихся частей вашего приложения, например REST API (http://api.meadowlarktravel.com) или

административного интерфейса (http://admin.meadowlarktravel.com). Иногда поддомены используются по техническим причинам. Например, если нам пришлось сделать блог на основе Wordpress, в то время как оставшаяся часть сайта использует Express, удобнее использовать адрес http://blog.meadowlarktravel.com (а еще лучше задействовать прокси-сервер, например NGINX). Обычно декомпозиция контента с помощью поддоменов приводит к определенным последствиям с точки зрения SEO. Именно поэтому в большинстве случаев их стоит приберечь для областей сайта, для которых SEO не важна, например, административный интерфейс и API. Помните об этом и перед использованием поддомена для контента, для которого важна поисковая оптимизация, убедитесь, что нет другого варианта.

Механизм маршрутизации в Express по умолчанию не учитывает поддомены: `app.get(/about)` будет обрабатывать запросы для http://meadowlarktravel.com/about, http://www.meadowlarktravel.com/about и http://admin.meadowlarktravel.com/about. Если вы хотите обрабатывать адреса с добавлением поддоменов отдельно, можете использовать пакет `vhost` (от англ. virtual host — виртуальный хост, ведущий свою историю от часто применяемого для обработки поддоменов механизма Apache). Сначала установим пакет (`npm install vhost`). Чтобы тестировать маршрутизацию на основе доменов на вашей рабочей машине, нужно каким-то образом получить фиктивные имена доменов. К счастью, для этого предназначен файл `hosts`. На машинах с операционными системами macOS и Linux его можно найти в `/etc/hosts`, с Windows — в `c:\windows\system32\drivers\etc\hosts`. Добавьте следующее в файл `hosts` (для его редактирования вам понадобятся права администратора):

```
127.0.0.1 admin.meadowlark.local
127.0.0.1 meadowlark.local
```

Это говорит вашему компьютеру, что meadowlark.local и admin.meadowlark.local следует обрабатывать как обычные домены сети Интернет, но сопоставлять им локальный хост (127.0.0.1). Мы используем высокоуровневый домен .local, чтобы избежать путаницы (вы можете использовать `.com` или любой другой домен в Интернете, но он переопределит реальный домен, что плохо).

Затем вы можете воспользоваться промежуточным ПО `vhost` для маршрутизации с учетом доменов (`ch14/00-subdomains.js` в прилагаемом репозитории):

```
// Создаем поддомен "admin"... это должно находиться
// до всех остальных ваших маршрутов.
var admin = express.Router()
app.use(vhost('admin.*', admin))

// Создаем маршруты для "admin"; это можно разместить в любом месте.
admin.get('*', (req, res) => res.send('Добро пожаловать, администратор!'))

// Обычные маршруты.
app.get('*', (req, res) => res.send('Добро пожаловать, пользователь!'))
```

По сути, `express.Router()` создает новый экземпляр маршрутизатора Express. Работать с этим экземпляром можно так же, как и с исходным (`app`): вы можете добавлять маршруты и промежуточное ПО точно так же, как добавляли бы в `app`.

Однако он ничего не будет делать до тех пор, пока вы не добавите его в app. Мы добавим его с помощью vhost, который свяжет данный экземпляр маршрутизатора с соответствующим поддоменом.

express.Router также можно использовать для декомпозиции маршрутов, так что за раз вы можете скомпоновать много обработчиков запросов. Подробную информацию см. в документации по маршрутизации в Express (http://bit.ly/2X8VC59).

Обработчики маршрутов — промежуточное ПО

Мы уже видели элементарную маршрутизацию — установление соответствия с заданным путем. Но что же на самом деле *выполняет* app.get(\'/foo',…)? Как мы видели в главе 10, это просто специализированное промежуточное ПО, вплоть до передачи метода next. Посмотрим на пример посложнее (ch14/01-fifty-fifty.js в прилагаемом репозитории):

```
app.get('/fifty-fifty', (req, res, next) => {
  if(Math.random() < 0.5) return next()
  res.send('иногда это')
})
app.get('/fifty-fifty', (req,res) => {
  res.send('а иногда вот это')
})
```

В предыдущем примере у нас есть два обработчика одного и того же маршрута. При обычных условиях использовался бы первый, но в данном случае первый будет пропускаться примерно в половине эпизодов, предоставляя возможность выполниться второму. Нам даже не требуется использовать app.get дважды: в одном вызове app.get можно использовать столько обработчиков, сколько нужно. Вот пример с приблизительно одинаковой вероятностью трех различных ответов (ch14/02-red-green-blue.js в прилагаемом репозитории):

```
app.get('/rgb',
  (req, res, next) => {
    // Около трети запросов получат ответ "красный".
    if(Math.random() < 0.33) return next()
    res.send('красный')
  },
  function(req,res, next){
    // Половина остальных двух третей (то есть вторая треть)
    // запросов получат ответ "зеленый".
    if(Math.random() < 0.5) return next()
    res.send('зеленый')
  },
  function(req,res){
    // И последняя треть запросов получат ответ "синий".
```

```
    res.send('синий')
  },
)
```

Хотя на первый взгляд это и может показаться не особо полезным, но позволяет вам создавать функции широкого применения, пригодные для использования в любом из ваших маршрутов. Например, пусть у нас имеется механизм отображения специальных предложений на определенных страницах. Специальные предложения часто меняются, причем отображаются не на каждой странице. Мы можем создать функцию для внедрения специальных предложений в свойство `res.locals` (которое вы, наверное, помните из главы 7) (`ch14/03-specials.js` в прилагаемом репозитории):

```
async function specials(req, res, next) {
  res.locals.specials = getSpecialsFromDatabase()
  next()
}

app.get('/page-with-specials', specials, (req, res) =>
  res.render('page-with-specials')
)
```

С помощью этого подхода можно реализовать механизм авторизации. Пускай код авторизации пользователя устанавливает сеансовую переменную `req.session.authorized`. Для создания пригодного для многократного применения фильтра авторизации можно использовать следующее (`ch14/04-authorizer.js` в прилагаемом репозитории):

```
function authorize(req, res, next){
  if(req.session.authorized) return next()
  res.render('not-authorized')
}

app.get('/public', () => res.render('public'))

app.get('/secret', authorize, () => res.render('secret'))
```

Пути маршрутов и регулярные выражения

Когда вы задаете путь (например, `/foo`) в своем маршруте, Express в конечном счете преобразует его в регулярное выражение. В путях маршрутов можно использовать некоторые метасимволы регулярных выражений: `+`, `?`, `*`, `(` и `)`. Рассмотрим несколько примеров. Допустим, вы хотели бы, чтобы URL `/user` и `/username` обрабатывались с помощью одного маршрута:

```
app.get('/user(name)?', (req, res) => res.render('user'));
```

Один из моих любимых креативных сайтов — к сожалению, прекративший свое существование — http://khaaan.com. Все, что там было, — это изображение всеми

любимого капитана Хан Соло из «Звездных войн» с его культовым поясом. Бесполезно, но всегда вызывало улыбку. Допустим, мы хотим сделать собственную страницу KHAAAAAAAAN, но не хотели бы заставлять пользователей помнить, что в названии две, три или десять букв «а». Следующий код решает эту задачу:

```
app.get('/khaa+n', (req, res) => res.render('khaaan'))
```

Однако не все обычные метасимволы регулярных выражений имеют смысл в путях маршрутов — только вышеперечисленные. Это важно, поскольку точка — обычный метасимвол регулярных выражений со значением «любой символ» и может использоваться в маршрутах без экранирования.

Наконец, если вам действительно нужны все возможности регулярных выражений для вашего маршрута, поддерживается даже вот такое:

```
app.get(/crazy|mad(ness)?|lunacy/, (req,res) =>
  res.render('madness')
)
```

Я пока не находил веских причин для использования метасимволов регулярных выражений в путях маршрутов, не говоря уже о полных регулярных выражениях, однако не помешает знать о существовании подобной функциональности.

Параметры маршрутов

В то время как регулярные выражения вряд ли окажутся в числе ежедневно используемого вами инструментария Express, параметры маршрутов вы, вероятнее всего, будете использовать довольно часто. Вкратце: это способ сделать часть вашего маршрута переменным параметром. Допустим, на сайте нужна отдельная страница для каждого сотрудника. У нас есть база данных сотрудников с биографиями и фотографиями. По мере роста компании добавление нового маршрута для каждого сотрудника становится все более громоздким. Посмотрим, как в этом могут помочь параметры маршрута (`ch14/05-staff.js` в прилагаемом репозитории):

```
const staff = {
  mitch: { name: "Митч",
    bio: 'Митч — человек, который прикроет вашу спину ' +
    'во время драки в баре.' },
  madeline: { name: "Мадлен", bio: 'Мадлен — наш специалист по Орегону.' },
  walt: { name: "Уолт", bio: 'Уолт — наш специалист по пансионату Орегон
Коуст.' },
}

app.get('/staff/:name', (req, res, next) => {
  const info = staff[req.params.name]
  if(!info) return next() // В конечном счете передаст
                          // управление обработчику кода 404.
  res.render('05-staffer', info)
})
```

Обратите внимание на то, как мы использовали `:name` в нашем маршруте. Это будет соответствовать любой строке, не включающей косую черту, и поместит ее в объект `req.params` с ключом `name`. Данную возможность мы будем часто применять, особенно при создании REST API. В маршруте может быть несколько параметров. Например, если мы хотим разбить список сотрудников по городам, можно использовать следующее:

```
const staff = {
  portland: {
    mitch: { name: "Митч", bio: 'Митч – человек, который прикроет вашу спину.' },
    madeline: { name: "Мадлен",bio: 'Мадлен – наш специалист по Орегону.' },
  },
  bend: {
    walt: { name: "Улолт",
    bio: 'Уолт – наш специалист по пансионату Орегон Коуст.' },
  },
}

app.get('/staff/:city/:name', (req, res, next) => {
  const cityStaff = staff[req.params.city]
  if(!cityStaff) return next() // Если город не опознан, управление
                               // передастся обработчику ошибки 404.
  const info = cityStaff[req.params.name]
  if(!info) return next() // Если работник не опознан, управление
                          // передастся обработчику ошибки 404.
  res.render('staffer', info)
})
```

Организация маршрутов

Вам уже, наверное, ясно, что определять все маршруты в главном файле приложения — слишком несуразное решение. Кроме того, что со временем этот файл начнет расти, данный способ разделения функциональности не самый лучший, поскольку в нем и так уже много чего выполняется. У простого сайта может быть десяток или около того маршрутов, но у более крупного их количество может измеряться сотнями.

Так как же организовать маршруты? Точнее, как бы вы *хотели* их организовать? В организации маршрутов вас ограничивает не Express, а только ваша фантазия.

Я рассмотрю некоторые популярные способы обработки маршрутов в следующих разделах, но в целом рекомендую придерживаться четырех руководящих принципов принятия решения о способе организации маршрутов.

- *Используйте именованные функции для обработчиков маршрутов.* Написание обработчиков маршрутов путем определения функции прямо на месте подходит для небольших приложений или создания прототипа, но по мере роста сайта такой способ окажется неудобным.

- *Маршруты не должны выглядеть загадочно.* Этот принцип умышленно сформулирован расплывчато, поскольку большой сложный сайт может потребовать более сложной схемы организации, чем десятистраничный сайт. На одном конце спектра — расположение всех маршрутов вашего сайта в одном файле, так что вы всегда будете знать, где их искать. Для более крупных сайтов такая схема нежелательна, поэтому вам придется разбить маршруты по функциональным областям. Однако даже тогда нужно следить за тем, чтобы было понятно, где искать конкретный маршрут. Когда вам понадобится исправить что-либо, последнее, чего вы захотите, — провести час, выясняя, где именно обрабатывается маршрут. У меня был проект ASP.NET MVC, который в этом отношении оказался просто ужасен. Маршруты обрабатывались как минимум в десяти различных местах не только нелогично и непоследовательно, но часто даже внутренне противоречиво. Несмотря на то что я был хорошо знаком с этим (очень большим) сайтом, мне постоянно приходилось проводить немало времени, выискивая, где же обрабатываются некоторые URL.
- *Организация маршрутов должна быть расширяемой.* Если у вас на текущий момент 20 или 30 маршрутов, определить их все в одном файле, вероятно, вполне нормально. Но что будет через три года, когда у вас будет 200 маршрутов? Это вполне может произойти. Какой бы метод вы ни выбрали, следует удостовериться, что у вас есть свободное пространство для роста.
- *Не игнорируйте автоматические обработчики маршрутов на основе представлений.* Если ваш сайт состоит из множества статических страниц и у него фиксированные URL, все ваши маршруты будут заканчиваться примерно так: `app.get('/static/thing', (req, res) => res.render(\'static/thing'))`. Чтобы снизить количество ненужного повторения кода, обдумайте возможность использования автоматических обработчиков маршрутов на основе представлений. Этот подход описан далее в этой главе и может применяться вместе с пользовательскими маршрутами.

Объявление маршрутов в модуле

Первый шаг организации маршрутов — размещение их в собственном модуле. Существует несколько способов сделать это. Один из них — когда ваш модуль возвращает массив объектов, содержащих свойства `method` и `handler`. Затем вы можете описать маршруты в файле своего приложения следующим образом:

```
const routes = require('./routes.js')

routes.forEach(route => app[route.method](route.handler))
```

У этого способа есть свои достоинства, и он может быть приспособлен для динамического хранения маршрутов, например, в базе данных или в файле JSON. Однако, если такая функциональность вам не нужна, советую передавать в модуль

экземпляр app и поручить ему добавление маршрутов. Именно такой подход мы будем использовать в нашем примере. Создадим файл routes.js и переместим в него все наши существующие маршруты:

```
module.exports = app => {

  app.get('/', (req,res) => app.render('home'))

  //...

}
```

Если мы просто вырежем их и вставим, то столкнемся с определенными проблемами. Например, если у нас есть встроенные обработчики маршрутов, использующие переменные и методы, которые недоступны в новом контексте, то связи будут нарушены. Мы можем добавить требуемые импорты, но воздержимся от этого. В ближайшем будущем мы собираемся переместить обработчики в их собственный модуль, тогда и решим данную проблему.

Так как мы подключим маршруты? Очень легко: в meadowlark.js, просто импортируем наши маршруты:

```
require('./routes')(app)
```

Либо мы можем сделать это более явно — добавить именованный импорт (который мы назвали addRoutes, чтобы лучше отразить его природу как функции; при желании можно назвать файл так же):

```
const addRoutes = require('./routes')

addRoutes(app)
```

Логическая группировка обработчиков

Чтобы придерживаться первого руководящего принципа (использование именованных функций для обработчиков маршрутов), нам понадобится место, куда можно поместить эти обработчики. Один довольно-таки экстремальный вариант — отдельный JavaScript-файл для каждого обработчика. Мне сложно представить ситуацию, когда данный подход был бы оправдан. Лучше каким-то образом сгруппировать вместе связанную между собой функциональность. Это не только упрощает применение совместно используемой функциональности, но и облегчает внесение изменений в связанные методы.

Пока сгруппируем нашу функциональность в следующие отдельные файлы: handlers/main.js, куда поместим обработчик домашней страницы, обработчик страницы О нас и вообще любой обработчик, которому не найдется другого логического места; handlers/vacations.js, куда попадут относящиеся к отпускным турам обработчики и т. д.

Рассмотрим файл handlers/main.js:

```
const fortune = require('../lib/fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) => {
  const fortune = fortune.getFortune()
  res.render('about', { fortune })
}

//...
```

Теперь изменим файл routes.js, чтобы воспользоваться этим:

```
const main = require('./handlers/main')

module.exports = function(app) {

  app.get('/', main.home)
  app.get('/about', main.about)
  //...

}
```

Это удовлетворяет всем нашим руководящим принципам. Файл /routes.js исключительно понятен. В нем с ходу легко разобрать, какие маршруты существуют в вашем сайте и где они обрабатываются. Мы также оставили себе массу свободного пространства для роста. Можем сгруппировать связанную функциональность в таком количестве различных файлов, какое нам нужно. И если routes.js когда-нибудь станет слишком громоздким, мы можем снова использовать тот же прием и передать объект app другому модулю, который, в свою очередь, зарегистрирует больше маршрутов (хотя это начинает попахивать переусложнением — убедитесь, что столь запутанный подход в вашем случае действительно обоснован).

Автоматический рендеринг представлений

Если вы когда-либо испытывали ностальгию по старым добрым дням, когда можно было просто поместить файл HTML в каталог и — вуаля! — ваш сайт выдавал его, то вы не одиноки. Если ваш сайт перегружен контентом без особой функциональности, добавление маршрута для каждого представления будет казаться ненужной морокой. К счастью, существует способ обойти эту проблему.

Допустим, вы просто хотите добавить файл views/foo.handlebars и каким-то чудесным образом сделать его доступным по пути /foo. Взглянем, как это можно сделать. В файле приложения прямо перед обработчиком кода 404 добавим следующее промежуточное ПО (ch14/06-auto-views.js в прилагаемом репозитории):

```
const autoViews = {}
const fs = require('fs')
```

```
const { promisify } = require('util')
const fileExists = promisify(fs.exists)

app.use(async (req, res, next) => {
  const path = req.path.toLowerCase()e();
  // Проверка кэша; если он там есть, визуализируем представление.
  if(autoViews[path]) return res.render(autoViews[path])
  // Если его нет в кэше, проверяем наличие
  // подходящего файла .handlebars
  if(await fileExists(__dirname + '/views' + path + '.handlebars')) {{
    autoViews[path] = path.replace(/^\//, '')
    return res.render(autoViews[path])
  }
  // Представление не найдено; переходим к обработчику кода 404.
  next()
})
```

Теперь можем просто добавить файл `.handlebars` в каталог `view`, и он как по мановению волшебной палочки будет отображаться по соответствующему пути. Обратите внимание, что обычные маршруты будут обходить этот механизм (поскольку мы разместили автоматический обработчик представления после всех маршрутов), так что, если у вас есть маршрут, отображающий другое представление для маршрута `/foo`, он будет иметь преимущество.

Нужно отметить, что при этом подходе вы столкнетесь с проблемами, если удалите посещенное представление; оно могло быть добавлено в объект `autoViews`, и следующие представления будут пытаться его визуализировать даже в случае удаления, что вызовет ошибку. Проблему можно решить путем обертывания в блок `try/catch` и удаления этого представления из `autoViews` при обнаружении ошибки; данное усовершенствование я оставляю читателю в качестве упражнения.

Резюме

Маршрутизация — важная часть вашего проекта. Существует множество возможных подходов к организации обработчиков маршрутов, так что не стесняйтесь экспериментировать и находить методы, подходящие для вас и вашего проекта. Я призываю вас выбирать явные методы, которые легко отслеживать. Маршрутизация, по большому счету, — это то, что связывает внешний мир (клиент, обычно браузер) и отвечающий на стороне сервера код. Сложность отображения этой связи затруднит отслеживание потока информации в вашем приложении, что станет помехой как при разработке, так и при отладке.

15 REST API и JSON

Хотя мы познакомились с несколькими примерами REST API в главе 8, до сих пор наша парадигма звучала так: «обработать данные на стороне сервера и отправить форматированный HTML клиенту». Все чаще этот режим выполнения операций перестает быть режимом по умолчанию для веб-приложений. Их сменили одностраничные приложения (SPA), изначально получающие свой HTML- и CSS-код в одном статическом пакете, а в дальнейшем — неструктурированные данные в формате JSON и напрямую манипулирующие HTML. Точно так же сообщение посредством размещения форм сменилось прямым взаимодействием с сервером с помощью HTTP-запросов к API.

Так что пришло время обратить внимание на использование Express для предоставления конечных точек API вместо предварительно форматированных HTML. Это пригодится в главе 16, когда мы продемонстрируем возможности API по динамическому рендерингу приложений.

В этой главе мы ограничимся лишь предоставлением интерфейса для работы с HTML: написанием остальных частей приложения займемся в главе 16. Вместо этого сосредоточимся на API, обеспечивающем доступ к базе данных отпускных туров и поддержку регистрации слушателей для «несезонных туров».

Веб-сервис — обобщенный термин, означающий любой интерфейс программирования приложений (API), доступный по протоколу HTTP. Идея веб-сервисов давно витала в воздухе, но до недавнего времени реализующие их технологии были слишком консервативными, сложными и запутанными. Системы, использующие такие технологии (например, SOAP и WSDL), существуют до сих пор, и есть пакеты Node, с помощью которых можно наладить взаимодействие с ними. Впрочем, описывать это я не стану, мы сосредоточимся на предоставлении так называемых RESTful-сервисов, сопряжение с которыми гораздо проще.

Акроним REST означает «передача состояния представления», а странная фраза RESTful используется как прилагательное для описания веб-сервисов, соответствующих принципам REST. Формальное определение REST довольно сложное, оно переполнено теорией вычислительных машин и систем, но суть его в том, что REST — соединение между клиентом и сервером без сохранения состояния. Формальное определение REST также указывает на то, что сервис может кэшироваться и сервисы могут быть многослойными (то есть под используемым вами REST API могут быть другие REST API).

С практической точки зрения из-за ограничений HTTP сложно создать RESTful API: например, вам пришлось бы сильно постараться, чтобы сохранять состояние. Так что наша работа по большей части уже сделана за нас.

JSON и XML

Чтобы обеспечить функционирование API, необходим общий язык для общения. Часть средств связи нам навязана: для соединения с сервером мы должны использовать методы HTTP. Но за исключением этого мы свободны в выборе языка данных. Традиционно для этого используется XML, по-прежнему имеющий немалое значение как язык разметки. Хотя XML не особенно сложен, Дуглас Крокфорд заметил, что нелишним было бы создать что-то более простое. Так на свет появилась нотация объектов JavaScript (JavaScript Object Notation, JSON). Помимо исключительной дружественности по отношению к JavaScript (хотя она никоим образом не проприетарна — это просто удобный формат для парсинга на любом языке), она может похвастаться большим (по сравнению с XML) удобством при написании вручную.

Я предпочитаю JSON, а не XML для большинства приложений из-за лучшей поддержки JavaScript и более простого и сжатого формата. Советую сосредоточиться на JSON и обеспечивать поддержку XML только в том случае, когда существующая система требует XML для связи с вашим приложением.

Наш API

Мы детально спроектируем наш API, прежде чем начать его реализовывать. В дополнение к перечислению туров и подписке на уведомления о «несезонных турах» добавим конечную точку «удалить тур». Поскольку это публичный API, мы не будем на самом деле удалять отпускной тур, а просто отметим его как тур, для которого было «запрошено удаление», и оставим все на рассмотрение администратора. Например, вы можете использовать эту незащищенную конечную точку, чтобы

дать возможность поставщикам сделать запрос об удалении туров, который позже рассмотрит администратор. Вот конечные точки вашего API.

`GET /api/vacations`

Извлекает отпускные туры.

`GET /api/vacation/:sku`

Возвращает тур по его SKU.

`POST /api/vacation/:sku/notify-when-in-season`

Принимает адрес электронной почты как параметр строки запроса и добавляет для указанного тура слушателя, ожидающего уведомления.

`DELETE /api/vacation/:sku`

Запрашивает удаление тура; принимает адрес электронной почты (принадлежащий человеку, запросившему удаление) и примечание в качестве параметров строки запроса.

Доступно большое количество HTTP-глаголов. Наиболее распространенные — GET и POST, за которыми следуют DELETE и PUT. Глагол POST, как правило, используется для создания, а PUT — для обновления (или изменения) чего-либо. Значение этих слов никак не указывает на то, чем отличается применение данных глаголов, так что вы, возможно, захотите использовать разные пути, чтобы различать эти две операции, во избежание недоразумений. Если вы хотите узнать больше о HTTP-глаголах, рекомендую начать со статьи Тамаса Пироса (http://bit.ly/32L4QWt).

Существует множество способов описания API. Здесь мы используем сочетание методов и путей HTTP для распознавания вызовов API и смесь параметров строки и тела запроса для передачи данных. Как вариант, мы могли использовать различные пути (например, `/api/vacations/delete`) с одним и тем же методом HTTP[1]. К тому же мы могли передавать данные единообразно. Например, можно было передавать всю необходимую для извлечения параметров информацию в URL вместо использования строки запроса `DEL /api/vacation/:id/:email/:notes`. Чтобы избежать чрезмерно длинных URL, рекомендую использовать тело запроса для передачи больших блоков данных (например, примечания о запросе на удаление).

[1] Если ваш клиент не может использовать различные методы HTTP, см. пакет http://bit.ly/2O7nr9E, позволяющий подделывать разные HTTP-методы.

Для API в JSON принято популярное и высокоавторитетное соглашение с креативным названием JSON:API. На мой взгляд, оно многословное, в нем много повторений, но я считаю, что наличие несовершенного стандарта куда лучше, чем его отсутствие. Хотя мы не используем JSON:API в этой книге, вы изучите все необходимое, чтобы освоить принятые в JSON:API соглашения. Более подробную информацию см. на домашней странице JSON:API (https://jsonapi.org/).

Выдача отчета об ошибках API

Выдача отчета об ошибках в интерфейсах программирования приложений HTTP обычно выполняется посредством кодов состояния HTTP. Если запрос возвращает 200 (OK), клиент знает, что тот был успешен. Если запрос возвращает 500 (Внутренняя ошибка сервера), то он потерпел неудачу. В большинстве приложений, однако, далеко не все может (или должно) классифицироваться строго как успех или неудача. Например, что будет, если вы запрашиваете что-либо по ID, но данный ID не существует? Это не ошибка сервера: клиент запросил что-то несуществующее. В целом ошибки можно сгруппировать в следующие категории.

- *Катастрофические ошибки.* Ошибки, приводящие к нестабильному или неопределенному состоянию сервера. Обычно они происходят в результате необработанного исключения. Единственный безопасный способ восстановления после катастрофической ошибки — перезагрузка сервера. В идеале любые ожидающие обработки запросы при этом должны получить код ответа 500, но, если сбой достаточно серьезен, сервер может оказаться не в состоянии ответить и запросы завершатся из-за превышения лимита времени.
- *Устранимые ошибки сервера.* Устранимые ошибки сервера не требуют его перезагрузки или каких-либо других решительных действий. Такая ошибка — результат неожиданной сбойной ситуации на сервере (например, недоступности подключения к базе данных). Проблема при этом может быть постоянной или кратковременной. Код ответа 500 вполне подходит в подобной ситуации.
- *Ошибка клиента.* Ошибка клиента — результат совершенной клиентом ошибки, подразумевающей, как правило, пропуск или ввод некорректных параметров. Использовать код ответа 500 при этом неуместно: в конце концов, сбоя сервера не было. Все работает нормально, просто клиент использует API неправильно. У вас есть несколько вариантов ответных действий: вы можете вернуть код состояния 200 и описать ошибку в теле ответа или дополнительно попытаться описать ошибку с помощью соответствующего кода состояния HTTP. Рекомендую использовать второй подход. Наиболее подходящие коды ответа в данном случае: 404 (Не найдено), 400 (Неверный запрос) и 401 (Не авторизован). Кроме них, тело ответа должно содержать подробности ошибки. Если

вы хотите пойти дальше, сообщение об ошибке может даже содержать ссылку на документацию. Обратите внимание: если пользователь запрашивает список чего-либо, а возвращать нечего, это не сбойная ситуация — вполне допустимо просто вернуть пустой список.

В нашем приложении будем использовать сочетание кодов ответа HTTP и сообщений об ошибках в теле.

Совместное использование ресурсов между разными источниками

Если вы публикуете API, то, вероятно, хотели бы сделать его доступным для других. Это приведет к появлению *межсайтовых HTTP-запросов*. Межсайтовые HTTP-запросы использовались во многих атаках, из-за чего их применение было ограничено в соответствии с *принципом одинакового источника*, накладывающим ограничения на места, откуда могут быть загружены скрипты. А именно, должны совпадать протокол, домен и порт. Это делает невозможным использование вашего API другими сайтами, и тут-то на помощь приходит совместное использование ресурсов между разными источниками (cross-origin resource sharing, CORS). CORS позволяет вам в отдельных случаях убирать эти ограничения вплоть до указания списка конкретных доменов, которым разрешен доступ к скрипту. CORS реализуется посредством заголовка `Access-Control-Allow-Origin`. Простейший способ реализовать его в приложении Express состоит в использовании пакета `cors` (`npm install cors`). Чтобы активизировать CORS в вашем приложении, используйте:

```
const cors = require('cors')

app.use(cors())
```

Поскольку ограничение API одним источником не случайно, а предназначено для предотвращения атак, я советую применять CORS только при необходимости. В данной ситуации мы хотим сделать доступным весь наш API (но только API), так что ограничим CORS путями, начинающимися с `/api`:

```
const cors = require('cors')

app.use('/api', cors())
```

Смотрите документацию к пакету `cors` (https://github.com/expressjs/cors) для получения информации о расширенном использовании CORS.

Наши тесты

При использовании отличных от GET глаголов HTTP тестирование может оказаться проблематичным, поскольку браузеры могут инициировать только запросы типа GET (и запросы POST для форм). Существуют пути обхода этого ограничения,

например замечательное приложение Postman (https://www.getpostman.com/). Однако, независимо от того, используете вы подобную утилиту или нет, автоматизированные тесты не помешают. Прежде чем мы будем писать тесты для API, нам потребуется способ *вызова* REST API. Для этого используем пакет Node под названием `node-fetch`, который дублирует браузерный API fetch:

```
npm install --save-dev node-fetch@2.6.0
```

Поместим тесты для тех вызовов API, которые собираемся реализовать, в файл `tests/api/api.test.js` (`ch15/test/api/api.test.js` в прилагаемом репозитории):

```
const fetch = require('node-fetch')

const baseUrl = 'http://localhost:3000'

const _fetch = async (method, path, body) => {
  body = typeof body === 'string' ? body : JSON.stringify(body)
  const headers = { 'Content-Type': 'application/json' }
  const res = await fetch(baseUrl + path, { method, body, headers })
  if(res.status < 200 || res.status > 299)
    throw new Error(`API вернул состояние ${res.status}`)
  return res.json()
}

describe('API tests', () => {

  test('GET /api/vacations', async () => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    expect(vacation0.name).toMatch(/\w/)
    expect(typeof vacation0.price).toBe('number')
  })
  test('GET /api/vacation/:sku', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    const vacation = await _fetch('get', '/api/vacation/' + vacation0.sku)
    expect(vacation.name).toBe(vacation0.name)
  })

  test('POST /api/vacation/:sku/notify-when-in-season', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    // На данный момент все, что можно сделать, – удостовериться,
    // что HTTP-запрос был успешным.
    await _fetch('post', `/api/vacation/${vacation0.sku}/notify-when-in-season`,
    { email: 'test@meadowlarktravel.com' })
  })
```

```
  test('DELETE /api/vacation/:id', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    // На данный момент все, что можно сделать, — удостовериться,
    // что HTTP-запрос был успешным.
    await _fetch('delete', `/api/vacation/${vacation0.sku}`)
  })
})
```

Наш набор тестов запускается с функции-хелпера `_fetch`, которая производит некоторые организационные работы. Она закодирует тело в формате JSON, если это не было сделано ранее, добавит требуемые заголовки и выбросит соответствующую ошибку, если код состояния запроса не 200.

У нас есть по одному тесту для каждой конечной точки API. Я не утверждаю, что эти тесты устойчивы к ошибкам или исчерпывающие; даже для такого простого API можно (и нужно), чтобы для каждой конечной точки было по нескольку тестов. То, что приведено здесь, — это по большей части отправная точка, иллюстрирующая методы тестирования API.

Следует упомянуть несколько важных особенностей этих тестов. Во-первых, мы считаем, что API был запущен на порте 3000. Более надежный набор тестов нашел бы открытый порт, запустил на нем API, что было бы частью его настройки, и остановил бы его по завершении всех тестов. Во-вторых, этот тест предполагает, что API уже были предоставлены данные. Например, первый тест ожидает, что будет хотя бы один отпускной тур с названием и ценой. У вас может не быть возможности сделать такое предположение в реальном приложении (например, вы можете начать, когда данных нет, или протестировать допустимость отсутствия данных). Опять же в более надежном фреймворке тестирования был бы способ добавления и повторного внесения начальных данных в API, чтобы вы каждый раз могли начать с известного состояния. К примеру, у вас могут быть скрипты, которые настраивают базу данных, вносят в нее начальные данные, подключают к ней API, а затем удаляют ее при каждом запуске тестов. Как мы видели в главе 5, тестирование — это большая и сложная тема, и здесь мы можем лишь слегка ее коснуться.

Первый тест предназначен для конечной точки GET `/api/vacations`. Он охватывает все отпускные туры, проверяет на наличие хотя бы одного тура, проверяет, есть ли у первого тура название и цена. Мы могли бы, по всей видимости, протестировать другие свойства данных. В качестве упражнения подумайте, какие свойства важнее всего протестировать.

Второй тест покрывает конечную точку GET `/api/vacation/:sku`. Поскольку у нас нет единообразных тестовых данных, мы начинаем с получения всех отпускных туров и для тестирования этой конечной точки берем SKU первого тура.

Два последних теста охватывают конечные точки POST `/api/vacation/:sku/notify-when-in-season` и DELETE `/api/vacation/:sku`. К сожалению, наши текущие API и фреймворк тестирования мало что могут сделать для подтверждения того, что эти конечные точки делают то, что должны. Поэтому мы вызываем их по умолчанию и считаем, что если ошибок не возникло, то API все делает правильно. Если мы захотим сделать эти тесты более надежными, то должны либо добавить конечные точки, которые позволяют проверить действия (например, конечная точка, определяющая, был ли на конкретный отпускной тур зарегистрирован данный адрес электронной почты), либо дать тестам какую-нибудь лазейку для доступа к базе данных.

Если вы теперь запустите тесты, то они провалятся из-за превышения лимита времени, ибо мы не реализовали наш API и даже не запустили сервер. Так что давайте сделаем это!

Использование Express для предоставления API

Express вполне может обеспечить предоставление API. Доступны различные модули npm, которые предоставляют полезную функциональность (для примера см. `connect-rest` и `json-api`). Но я считаю, что стандартная версия Express прекрасно справится с этим, и мы будем придерживаться реализации с помощью только Express.

Начнем с создания обработчиков в `lib/handlers.js` (можно создать отдельный файл, например, `lib/api.js`, но пока не будем усложнять):

```
exports.getVacationsApi = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  res.json(vacations)
}

exports.getVacationBySkuApi = async (req, res) => {
  const vacation = await db.getVacationBySku(req.params.sku)
  res.json(vacation)
}

exports.addVacationInSeasonListenerApi = async (req, res) => {
  await db.addVacationInSeasonListener(req.params.sku, req.body.email)
  res.json({ message: 'выполнено успешно' })
}

exports.requestDeleteVacationApi = async (req, res) => {
  const { email, notes } = req.body
  res.status(500).json({ message: 'еще не реализовано' })
}
```

Затем мы подключим API в файле `meadowlark.js`:

```
app.get('/api/vacations', handlers.getVacationsApi)
app.get('/api/vacation/:sku', handlers.getVacationBySkuApi)
app.post('/api/vacation/:sku/notify-when-in-season',
  handlers.addVacationInSeasonListenerApi)
app.delete('/api/vacation/:sku', handlers.requestDeleteVacationApi)
```

Пока не происходит ничего необычного. Обратите внимание, что мы применяем слой абстракции базы данных, поэтому неважно, используем ли мы реализацию для MongoDB или для PostgreSQL (несмотря на небольшую неувязку — зависимость дополнительных полей от реализации, что при необходимости можно устранить).

Я оставлю `requestDeleteVacationsApi` в качестве упражнения для читателя, поскольку данную функциональность можно реализовать множеством разных способов. Самый простой — преобразовать схему отпускных туров так, чтобы в ней были поля «запрошено удаление», для которых обновляются адреса электронной почты и примечания при вызове API. Более сложный подход — самостоятельная таблица типа очереди на модерацию, в которую отдельно записываются запросы на удаление со ссылкой на соответствующий тур; лучше, если она сама будет передаваться администратору.

Если вы корректно установили Jest в главе 5, можете запустить `npm test` — и тесты API сработают (Jest будет искать все файлы, заканчивающиеся на `.test.js`). Вы увидите, что три теста прошли, а один провалился — для незавершенного DELETE `/api/vacation/:sku`.

Резюме

Надеюсь, после этой главы вы задались вопросом: «И это все?» На данный момент вы, вероятно, понимаете, что прямое назначение Express — отвечать на HTTP-запросы. Какие это запросы и как на них отвечать — все по вашему усмотрению. Должен ли быть ответом HTML-код? Или CSS-код? Неформатированный текст? JSON? Все это легко сделать с Express. Типом ответа может быть даже двоичный файл. Например, можно с легкостью динамически создать и вернуть изображения. В этом смысле API — всего лишь один из множества способов, которыми может ответить Express.

В следующей главе мы построим одностраничное приложение, чтобы использовать этот API, и повторим то, что сделали в прошлой главе, но другим способом.

16 Одностраничные приложения

Термин *«одностраничное приложение» (single-page application, SPA)* ошибочен, или по крайней мере в нем смешаны два значения слова «страница». С точки зрения пользователя, SPA могут содержать (и обычно содержат) разные страницы: домашнюю, страницу Туры, страницу О нас и т. д. Фактически вы могли бы создать традиционное приложение на стороне сервера и SPA, но для пользователя они оказались бы неразличимыми.

Понятие «одностраничное» больше отсылает к тому, где и как создавался HTML, чем к взаимодействию с пользователем. В SPA сервер доставляет отдельный пакет HTML, когда пользователь впервые загружает приложение[1], и все изменения в интерфейсе пользователя (который для него может выглядеть как несколько страниц) являются результатом управления DOM со стороны JavaScript в ответ на действия пользователя и события в сети.

SPA по-прежнему нужно часто взаимодействовать с сервером, но HTML обычно отправляется как часть первого запроса. Потом сервер и клиент обмениваются лишь данными в формате JSON и статическими ресурсами.

Чтобы понять, почему сейчас предпочтение отдается именно такому подходу к разработке веб-приложений, нужен небольшой экскурс в историю.

Краткая история разработки веб-приложений

В течение последнего десятилетия подход к веб-разработке претерпел значительные изменения, но единственное, что остается неизменным, — это компоненты, задействованные на веб-сайте или в веб-приложении. А именно:

- HTML и объектная модель документа (DOM);
- JavaScript;

---

[1] Для увеличения производительности пакет может быть разделен на части, которые загружаются при необходимости (это называется ленивой загрузкой), но принцип тот же.

- CSS;
- статические ресурсы (обычно это мультимедийные файлы: изображения, видеофайлы и т. д.).

Эти компоненты, объединенные браузером, обеспечивают взаимодействие с пользователем. Однако приблизительно в 2012 г. принципы, на которых основано это взаимодействие, стали кардинально меняться. В наши дни в веб-разработке преобладает парадигма *одностраничных приложений*, или SPA.

Для того, чтобы понять суть SPA, нужно знать, с чем их сравнивать, поэтому вернемся в 1998 г., время, когда появилось первое упоминание о Web 2.0 (до появления jQuery остается восемь лет).

В 1998 г. основным методом выполнения веб-приложений была отправка веб-сервером HTML, CSS, JavaScript и мультимедийных ресурсов *в ответ на каждый запрос*. Представьте, что вы смотрите телевизор и хотите переключить канал. В метафорическом смысле вам нужно выбросить ваш телевизор, купить новый, притащить его домой и настроить — все это лишь для того, чтобы переключить канал (то есть перейти на другую страницу даже на том же самом сайте).

В 1999 г. в попытке описать богатство взаимодействия с веб-сайтами в обращение был введен термин Web 2.0. В период между 1999 и 2012 гг. наблюдался технологический прогресс, подготовивший почву для SPA.

Рассудительные веб-разработчики стали осознавать, что если они хотят сохранить своих пользователей, то издержки на отправку всего веб-сайта всякий раз, когда пользователь пожелает (в метафорическом смысле) переключить канал, неприемлемы. Они поняли, что далеко не каждое изменение в приложении требует информации с сервера, а также то, что не каждому изменению, требующему информации с сервера, нужна пересылка целого приложения, порой достаточно лишь небольшого фрагмента.

В период с 1999 по 2012 г. страницы оставались обычными страницами: при заходе на веб-сайт вы получали HTML-код, CSS и статические ресурсы. При переходе на другую страницу вы получали другой HTML-код, иные статические ресурсы, а иногда даже другой CSS. Однако на каждой странице сама страница могла быть изменена в результате взаимодействия с пользователем, и вместо того, чтобы просить у сервера целое приложение, JavaScript менял непосредственно DOM. Если информацию нужно было доставлять с сервера, то она отправлялась в форматах XML и JSON без всего сопровождающего HTML-кода. При этом интерпретация данных и соответствующее изменение интерфейса пользователя оставались на усмотрение JavaScript. В 2006 г. был представлен jQuery, значительно снизивший нагрузку на управление DOM и обработку запросов по сети.

Многие из этих изменений были обусловлены растущей мощностью компьютеров и, как следствие, браузеров. Веб-разработчики приходили к мысли, что все больше работ, направленных на то, чтобы сделать веб-сайт или веб-приложение красивее, могут выполняться на компьютере пользователя, а не на стороне сервера.

Эти перемены в подходе достигли апогея, когда в конце 2000-х гг. появились смартфоны. Теперь не только браузеры могли больше, но и люди хотели получать доступ к веб-приложениям по беспроводной сети. Издержки на отправку данных неожиданно возросли, и сведение к минимуму пересылки данных по сети, а также выполнение браузером максимального количества работ стало задачей номер один.

К 2012 г. общепринятой практикой стало отправлять как можно меньше информации по сети и производить как можно больше действий в браузере. Словно бульон, породивший первую форму жизни, эта богатая среда обеспечила условия для естественной эволюции одностраничного приложения.

Идея достаточно простая: для любого веб-приложения HTML, JavaScript и CSS (если они есть) передаются только один раз. Как только браузер получил HTML, JavaScript производит все изменения в DOM, заставляя пользователей думать, что они перешли на другую страницу. Серверу больше не нужно отправлять другой HTML, когда вы, например, переходите с домашней страницы на страницу Туры.

Сервер по-прежнему участвует в процессе: он отвечает за предоставление актуальных данных и является «единственным источником истины» в многопользовательских приложениях. Но в архитектуре SPA он больше не отвечает за то, каким приложение предстанет перед пользователем. Теперь это задача JavaScript и фреймворков.

Хотя Angular обычно считается первым фреймворком SPA, есть и другие не менее значимые: React, Vue и Ember.

Если вы новичок в разработке, то SPA может быть единственной архитектурой, с которой вы знакомы, а все вышесказанное для вас — просто интересная история. Если же вы опытный специалист, то данные перемены могут показаться вам резкими и сбивающими с толку. В какую бы группу вы ни попали, в этой главе я помогу вам понять, как веб-приложения доставляются в виде SPA и какова в этом роль Express.

Эта история имеет отношение к Express, поскольку роль сервера менялась в ходе изменений в технологиях веб-разработки. Когда вышло первое издание этой книги, Express повсеместно использовался для обслуживания многостраничных приложений (наряду с API, которые поддерживали Web 2.0-подобную функциональность). Теперь же он почти полностью используется для SPA, серверов разработки и API, отражая переменчивую природу веб-разработки.

Интересно, что все еще существуют веские доводы в пользу того, чтобы веб-приложения могли обслуживать какие-нибудь специфические страницы (вместо общих страниц, которые будут повторно форматироваться браузером). Может показаться, что это замкнутый круг или что мы отказываемся от достижений SPA, но эта техника лучше отражает архитектуру SPA. Эта техника, называемая рендерингом на стороне сервера (server-side rendering, SSR), позволяет серверам использовать для ускорения загрузки первой страницы тот же код, который браузеры применяют для создания отдельных страниц. Главное здесь то, что сервер не должен

много думать: он просто использует для генерации определенной страницы те же методы, что и браузер. Такой вид SSR обычно выполняется для ускорения загрузки первой страницы и для поддержки поисковой оптимизации. Это более продвинутая тема, которую мы не будем здесь рассматривать, но о ней следует знать.

Теперь, когда мы немного разобрались с тем, как и почему одностраничные приложения появились на свет, давайте взглянем на доступные сегодня фреймворки SPA.

Технологии SPA

Сегодня существует множество технологий SPA. Рассмотрим некоторые из них.

- *React*. На данный момент React — царь горы SPA, на одном склоне которой расположился бывший лидер (Angular), а на другом — амбициозный захватчик (Vue). В 2018 г. React обошел Angular по статистике использования. React — это библиотека с открытым исходным кодом, но изначально он задумывался как проект Facebook, и Facebook все еще поддерживает его. Мы будем использовать React для рефакторинга Meadowlark Travel.
- *Angular*. Angular от Google — первый SPA, по мнению большинства. Он приобрел широкую популярность, но в итоге был свергнут с престола React. В конце 2014 г. стало известно о выходе второй версии Angular. Она значительно отличалась от первой и вызвала отчуждение у многих старых пользователей, а новых отпугнула. Я считаю, что эти изменения (хотя, возможно, и необходимые) способствовали тому, что React наконец обставил Angular. Другая причина в том, что Angular — более крупный фреймворк, чем React. У этого есть и плюсы и минусы: Angular предоставляет более законченную архитектуру для построения полных приложений, а также имеет четкий путь в реализации определенных вещей, тогда как фреймворки типа React и Vue оставляют больше возможностей для свободы и творчества. Независимо от того, какой подход лучше, большие фреймворки громоздкие и медленно развиваются, что позволило React стать передовым.
- *Vue.js*. Это выскочка, бросивший вызов React, и порождение ума одного разработчика — Эвана Ю. За невероятно короткое время приобрел впечатляющее количество сторонников, которые его обожают, но ему все еще далеко до популярности React. У меня есть опыт работы с Vue, и я ценю его понятную документацию и легковесный подход, но мне больше по душе архитектура и философия React.
- *Ember*. Как и Angular, Ember предлагает исчерпывающий фреймворк приложений. У него есть большое и активное сообщество, и пусть он не такой новаторский, как React или Vue, он понятен и функционален. Я предпочитаю легковесные фреймворки, поэтому придерживаюсь React.
- *Polymer*. У меня нет опыта использования Polymer, но этот фреймворк поддерживает Google, что повышает доверие. Многим интересно, что может предложить Polymer, но я не видел большого числа желающих начать его использовать.

Если вы ищете надежный, не требующий настройки фреймворк и ничего не имеете против установленного набора правил, то вам следует рассмотреть Angular или Ember. Если у вас есть потребность в творческом самовыражении и инновациях, то я рекомендую React и Vue. Я еще не знаю, к какой категории отнести Polymer, но к нему стоит присмотреться.

Теперь, когда мы познакомились с игроками, приступим к рефакторингу Meadowlark Travel и преобразуем его в SPA с помощью React!

Создание приложения React

Самый простой способ начать работу с приложением React — воспользоваться утилитой create-react-app (CRA), создающей весь шаблонный код и инструменты разработчика, а также предоставляющей минимальное начальное приложение, которое вы затем можете достроить. Более того, create-react-app будет следить за актуальностью своей конфигурации, так что вы сможете сфокусироваться на создании вашего приложения вместо того, чтобы заниматься инструментарием фреймворка. Если вы когда-нибудь дойдете до того, что вам придется настроить комплект инструментальных средств, то можете «извлечь» ваше приложение: вы потеряете возможность поддерживать актуальность инструментария CRA, но получите полный контроль над конфигурацией.

В отличие от того, что мы делали до сих пор, когда все программные продукты располагались бок о бок с приложением Express, SPA лучше рассматривать как полностью отдельные и независимые приложения. В связи с этим у нас будет не один, а два корневых каталога приложения. Во избежание разночтений я буду обращаться к папке, в которой находится приложение Express, как к корневому каталогу сервера, а к папке с приложением React — как к корневому каталогу клиента. Корневой каталог приложения — место обитания обеих этих папок.

Перейдите в корневой каталог приложения и создайте папку `server`, где будет располагаться сервер Express. Не создавайте папку для клиентского приложения: CRA сделает это за вас.

Прежде чем запускать CRA, нужно установить Yarn (https://yarnpkg.com/). Yarn — такой же менеджер пакетов, как и npm. Он по большей части замещает npm. Yarn необязателен для разработки в React, но де-факто является стандартом, и игнорировать его использование — то же самое, что плыть против течения. Использование Yarn и npm несколько различается, но единственная разница, которую вы заметите, — то, что вы запускаете `yarn add` вместо `npm install`. Чтобы установить Yarn, просто следуйте инструкциям по установке Yarn (http://bit.ly/2xHZ2Cx).

После установки Yarn запустите следующее из каталога приложения:

```
yarn create react-app client
```

Теперь перейдите в каталог клиента и наберите `yarn start`. Через несколько секунд откроется новое окно браузера с запущенным приложением React!

Не останавливайтесь и оставьте запущенным окно терминала. В CRA есть поддержка горячей перезагрузки (hot reloading), так что, когда вы вносите изменения в ваш исходный код, он быстро скомпилируется и браузер автоматически перезапустится. Как только вы привыкнете, то не сможете без этого жить.

Основы React

По React есть прекрасная документация, которую я не буду здесь воспроизводить. Если вы новичок в React, начните с учебника «Введение в React» (http://bit.ly/36VdKUq) и продолжите с руководством по основным концепциям (http://bit.ly/2KgT939).

Вы увидите, что React строится вокруг компонентов — базовых элементов React. Все, что видит пользователь и с чем он взаимодействует, как правило, является компонентом React. Давайте посмотрим на `client/src/App.js` (содержимое вашего файла может слегка отличаться — CRA со временем вносит изменения):

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Отредактируйте <code>src/App.js</code> и сохраните для перезагрузки.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Одна из основных концепций React — интерфейс пользователя генерируется функциями. Как мы уже видели, простейший компонент React — обычная функция, возвращающая HTML. Возможно, вы смотрите на это и думаете, что это некорректный JavaScript; это выглядит как смесь с HTML! В реальности все немного

сложнее. По умолчанию React поддерживает расширенную версию JavaScript под названием JSX. JSX позволяет писать код, который выглядит как HTML. Но на самом деле это не HTML; он создает элементы React, предназначенные для сопоставления элементам DOM.

В принципе, вы можете рассматривать их как HTML-код. Здесь `App` — это функция, которая будет выполнять рендеринг HTML-кода, в соответствии с возвращаемым JSX.

Нужно отметить несколько моментов: поскольку JSX близок к HTML, но не является им, то между ними есть несколько тонких различий. Как вы могли заметить, мы используем `className` вместо `class`, так как `class` — зарезервированное слово в JavaScript.

Чтобы определить HTML, достаточно указать элемент HTML в любом месте, где ожидается выражение. Вы также можете вернуться к принятому в JavaScript использованию фигурных скобок внутри HTML. Например:

```
const value = Math.floor(Math.random()*6) + 1
const html = <div>Вам выпало {value}!</div>
```

В этом примере HTML начинается с `<div>`, а фигурные скобки вокруг `value` — это возврат к JavaScript для предоставления числа, хранящегося в `value`. Мы можем просто встроить вычисления:

```
const html = <div>Вам выпало {Math.floor(Math.random()*6) + 1}!</div>
```

Любое корректное выражение JavaScript можно поместить внутри фигурных скобок в JSX, включая элементы HTML! Распространенным случаем применения этого является рендеринг списков:

```
const colors = ['красный', 'зеленый', 'синий']
const html = (
  <ul>
    {colors.map(color =>
      <li key={color}>{color}</li>
    )}
  </ul>
)
```

Учитывая этот пример, стоит кое-что упомянуть. Во-первых, обратите внимание на то, что мы произвели сопоставление цветам элементов списка `<li>` с цветами. Это критично: JSX работает только путем вычисления выражений. Поэтому `<ul>` должен содержать либо выражение, либо массив выражений. Если вы замените `map` на `forEach`, то обнаружите, что элементы списка `<li>` не отображаются. Во-вторых, заметьте, что элементы `<li>` получают свойство `key`: это уступка в пользу производительности. Чтобы React знал, когда повторно отображать элементы в массиве, каждому элементу нужен уникальный ключ. Поскольку элементы нашего массива уникальные, мы просто воспользовались этим значением, но общепринятым

является использование ID или (если ничто другое не доступно) индекса элемента массива.

Предлагаю вам поэкспериментировать с примерами на JSX в `client/src/App.js`, прежде чем двигаться дальше. Если вы оставили запущенным yarn (`yarn start`), то при каждом сохранении изменений они будут автоматически отражаться в браузере, что ускорит процесс обучения.

Прежде чем завершить знакомство с основами React, нужно затронуть еще одну тему — концепцию состояния. У каждого компонента может быть собственное состояние, что, по сути, означает «связанные с компонентом данные, которые могут изменяться». Прекрасный пример — корзина покупок. В состоянии компонента корзины покупок содержится список элементов; когда вы добавляете элементы в корзину и удаляете их из нее, состояние компонента изменяется. Эта концепция может показаться чрезмерно простой или очевидной, но большинство нюансов создания приложения React сводится именно к эффективному проектированию и управлению состояниями компонентов. Мы увидим пример состояния, когда возьмемся за страницу Туры.

Давайте пойдем дальше и создадим домашнюю страницу Meadowlark Travel.

Домашняя страница

Помните из представлений Handlebars, что у нас есть файл с шаблоном, определяющим первоначальный внешний вид сайта. Для начала сфокусируемся на содержимом тега `<body>` (за исключением скриптов):

```
<div class="container">
  <header>
    <h1>Meadowlark Travel</h1>
    <a href="/"><img src="/img/logo.png" alt="Логотип Meadowlark Travel"></a>
  </header>
  {{{body}}}
</div>
```

Это все просто перестроить в компонент React. Во-первых, мы скопируем наш собственный логотип в каталог `client/src`. Почему не в каталог `public`? Небольшие или повсеместно используемые графические элементы эффективнее встроить в пакет JavaScript, а упаковщик (bundler), который вы получили вместе с CRA, сделает разумный выбор. Пример приложения, полученный от CRA, помещает свой логотип прямо в каталог `client/src`, но мне все-таки нравится собирать ресурсы изображений в подкаталоге, так что разместите наш логотип (`logo.png`) в `client/src/img/logo.png`.

Остался лишь один нюанс — что делать с `{{{body}}}`? В наших представлениях это то место, где будут отображаться другие представления — контент конкретной страницы, на которой вы находитесь. Мы можем продублировать эту идею в React.

Поскольку весь контент отображается в виде компонентов, то мы просто будем отображать здесь другой компонент. Начнем с пустого компонента Home и вскоре его достроим:

```
import React from 'react'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (<i>coming soon</i>)
}

function App() {
  return (
    <div className="container">
      <header>
        <h1>Meadowlark Travel</h1>
        <img src={logo} alt="Логотип Meadowlark" />
      </header>
      <Home />
    </div>
  )
}

export default App
```

Мы используем тот же подход для CSS, что и пример CRA: можно просто создать файл CSS и импортировать его. Таким образом, есть возможность редактировать этот файл и применять нужные стили. Мы не будем усложнять этот пример, поскольку в том, как мы стилизуем HTML с помощью CSS, ничего не изменилось, у нас есть все привычные инструменты.

CRA настраивает линтинг, и, выполняя примеры на протяжении этой главы, вы, возможно, будете видеть предупреждения (как в выводе CRA в терминале, так и в консоли JavaScript вашего браузера). Это происходит потому, что мы добавляем все постепенно; к концу главы не должно быть никаких предупреждений. Если они есть, проверьте, не пропустили ли вы какой-нибудь шаг! Вы также можете свериться с прилагаемым к книге репозиторием.

Маршрутизация

Основная концепция, которую мы изучали в главе 14, не изменилась: мы, как и раньше, используем путь URL, чтобы определить, какую часть интерфейса видит пользователь. Разница лишь в том, что происходящее обрабатывается клиентским приложением. Клиентское приложение отвечает за изменение интерфейса

пользователя согласно маршруту: если для навигации требуются новые или обновленные данные с сервера, то клиентское приложение само запрашивает эти данные у сервера.

Есть много способов маршрутизации в приложениях React и множество радикальных мнений об этом. Тем не менее для маршрутизации есть основная библиотека — React Router (http://bit.ly/32GvAXK). Мне многое не нравится в React Router, но она настолько распространена, что вы обязательно с ней столкнетесь. Более того, всегда хорошо, когда наготове есть что-то базовое, что можно взять и запустить. Исходя из этих двух причин, мы и будем использовать React Router в нашем проекте.

Начнем с установки версии React Router для DOM (для мобильной разработки предусмотрена версия для React Native):

```
yarn add react-router-dom
```

Теперь подключим маршрутизатор и добавим страницы О нас и Не найдено. По щелчку на логотипе сайта будет осуществляться переход на домашнюю страницу.

```
import React from 'react'
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (
    <div>
      <h2>Добро пожаловать на Meadowlark Travel</h2>
      <p>Посетите страницу "<Link to="/about">О нас</Link>"!</p>
    </div>
  )
}

function About() {
  return (<i>Скоро появится</i>)
}
function NotFound() {
  return (<i>Не найдено</i>)
}

function App() {
  return (
    <Router>
      <div className="container">
```

```
        <header>
          <h1>Meadowlark Travel</h1>
          <Link to="/"><img src={logo} alt="Логотип Meadowlark Travel" /></Link>
        </header>
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/about" exact component={About} />
          <Route component={NotFound} />
        </Switch>
      </div>
    </Router>
  )
}

export default App
```

Первое, что нужно отметить: мы оборачиваем все наше приложение в компонент `<Router>`. Как вы могли предположить, это позволяет осуществлять маршрутизацию. Внутри `<Router>` можно использовать `<Route>`, чтобы при определенных условиях отображать компонент, основываясь на пути URL. Мы поместили маршруты для нашего контента в компонент `<Switch>`: это гарантирует, что только один из содержащихся в нем компонентов отображается.

Маршрутизация с помощью Express и React Router немного различается. В Express мы можем отобразить страницу в соответствии с первым успешным совпадением (или страницу 404, если ничего нельзя найти). С React Router путь — это указание на то, какую комбинацию компонентов нужно отобразить. Это более гибкий способ, чем маршрутизация с помощью Express. По этой причине маршруты в React Router по умолчанию ведут себя так, будто заканчиваются знаком «звездочка» (`*`). То есть маршрут `/` по умолчанию будет совпадать с любой страницей, поскольку все они начинаются с наклонной черты. Поэтому мы используем свойство `exact`, чтобы заставить этот маршрут вести себя как маршрут в Express. Без свойства `exact` маршрут `/about` аналогично будет совпадать с `/about/contact`, а это отнюдь не то, что нужно. Для маршрутизации вашего контента вы, вероятно, захотите, чтобы все ваши маршруты (за исключением маршрута `Не найдено`) содержали `exact`. Иначе придется распределить их внутри `<Switch>`, чтобы они шли в правильном порядке.

Второй момент, достойный упоминания, — использование `<Link>`. Вы можете спросить, почему бы просто не использовать теги `<a>`. Дело в том, что если над ними дополнительно не потрудиться, то браузер будет старательно обрабатывать теги `<a>` как «переход куда-то» даже в пределах одного и того же сайта. В результате к серверу будет отправлен новый HTTP-запрос, HTML с CSS загрузятся заново и SPA потеряет смысл. Система будет функционировать и при загрузке страницы React Router все сделает правильно, но вызов ненужных сетевых запросов повлияет на скорость и эффективность работы. Важно понимать эту

разницу, демонстрирующую природу одностраничных приложений. В качестве эксперимента создайте два элемента навигации, один с использованием `<Link>`, а второй — с `<a>`:

```
<Link to="/">Домашняя страница (SPA)</Link>
<a href="/">Домашняя страница (перезагрузка)</Link>
```

Затем откройте ваши инструменты разработчика и вкладку Network (Сеть), очистите трафик и нажмите Preserve log (Сохранить лог) (в Chrome). Теперь щелкните на ссылке Home (SPA) (Домашняя страница (SPA)) и обратите внимание, что здесь вообще нет сетевого трафика. Щелкните на ссылке Home (reload) (Домашняя страница (перезагрузка)) и понаблюдайте за сетевым трафиком. В этом, говоря кратко, проявляется природа одностраничных приложений.

Страница Туры: графический дизайн

До сих пор мы занимались строительством чисто клиентского приложения. Когда же дело дойдет до Express? Наш сервер по-прежнему единственный источник истины. В частности, на нем хранится база данных с отпускными турами, которые мы хотим отображать на нашем сайте. К счастью, мы уже проделали бóльшую часть работы в главе 15: мы предоставили API, который будет возвращать туры в формате JSON, готовые для использования в приложении React.

Прежде чем связать две эти вещи, давайте все-таки забежим вперед и создадим страницу Туры. Пока нет туров, которые можно было бы отобразить, но пусть это вас не останавливает.

В предыдущем разделе мы включили все страницы с контентом в `client/src/App.js`, что, по большому счету, плохо: было бы лучше, если бы каждый элемент оставался отдельным. Так что мы потратим время для выделения Vacations в отдельный компонент. Создайте файл `client/src/Vacations.js`:

```
import React, { useState, useEffect } from 'react'
import { Link } from 'react-router-dom'

function Vacations() {
  const [vacations, setVacations] = useState([])
  return (
    <>
      <h2>Туры</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
```

```
      </div>
    </>
  )
}

export default Vacations
```

Пока все довольно просто: мы возвращаем `<div>`, содержащий дополнительные элементы `<div>`, каждый из которых представляет отпускной тур. Так откуда берется переменная `vacations`? В этом примере мы используем новейшую функцию React под названием React hooks (хуки). До появления хуков, если компоненту требовалось свое собственное состояние (в данном случае список туров), вам нужно было реализовать класс. Они позволяют иметь компоненты, основанные на функциях, со своим состоянием. Мы вызываем `useState` в функции `Vacations` для инициализации состояния. Обратите внимание: мы передаем `useState` пустой массив — начальное значение состояния `vacations` (вскоре я расскажу, как можно его заполнить). `useState` возвращает массив, содержащий само значение состояния (`vacations`), и способ обновления состояния (`setVacations`).

Вы можете спросить, почему нельзя непосредственно преобразовать `vacations`: это просто массив, так не можем ли мы вызвать `push` для добавления в него туров? Можем, но это сведет на нет саму цель системы управления состояниями React, обеспечивающую непротиворечивость, производительность и взаимодействие между компонентами.

Вас могло заинтересовать то, что выглядит как пустой компонент (`<>...</>`) вокруг туров. Это *фрагмент* (http://bit.ly/2ryneVj). Он нужен, поскольку каждый компонент должен отобразить один элемент. В нашем случае есть два элемента: `<h2>` и `<div>`. Фрагмент просто предоставляет прозрачный корневой элемент, чтобы поместить в него эти два элемента и отобразить их как один.

Добавим компонент `Vacations` в приложение, хотя у нас еще нет туров, чтобы их показать. Сначала импортируйте страницу с турами в `client/src/App.js`:

```
import Vacations from './Vacations'
```

Теперь все, что нам нужно сделать, — создать для него маршрут в компоненте маршрутизатора `<Switch>`:

```
<Switch>
    <Route path="/" exact component={Home} />
    <Route path="/about" exact component={About} />
    <Route path="/vacations" exact component={Vacations} />
    <Route component={NotFound} />
</Switch>
```

Сохраните это; ваше приложение должно автоматически перезагрузиться — и вы сможете перейти на страницу `/vacations`, хотя на ней пока нет ничего интересного. Теперь, когда у нас есть основная инфраструктура клиента, обратим внимание на взаимодействие с Express.

Страница Туры: интеграция сервера

Мы уже сделали большую часть работы для страницы Туры; у нас есть конечная точка API для получения туров из базы данных и возвращения их в формате JSON. Теперь нам нужно понять, как заставить взаимодействовать серверную и клиентскую части.

Можем начать с того, что делали в главе 15; туда не нужно ничего добавлять, но мы можем отбросить некоторые вещи, в которых отпала необходимость. Можно удалить следующее.

- Поддержку Handlebars и представлений (мы оставим промежуточное ПО static по причинам, которые рассмотрим позже).
- Cookie и сеансы (наш SPA по-прежнему может использовать cookie, но помощь сервера ему больше не нужна и мы рассматриваем сеансы в совершенно другом свете).
- Все маршруты, которые отображают представления (мы, конечно же, оставляем маршруты API).

Наш сервер стал значительно проще. Что мы теперь будем с ним делать? Во-первых, нужно обратить внимание на тот факт, что мы используем порт 3000, а сервер разработки CRA по умолчанию также использует порт 3000. Можно изменить оба, но я предлагаю поменять порт Express (это случайный выбор). Я обычно применяю 3033 лишь потому, что мне нравится это число. Помните, что мы установили порт по умолчанию в `meadowlark.js`, так что нам нужно поменять его:

```
const port = process.env.PORT || 3033
```

Мы, конечно, могли воспользоваться переменной среды, чтобы его контролировать, но, поскольку планируем часто использовать его вместе с сервером разработки SPA, можем изменить код.

Теперь, когда оба сервера запущены, они могут взаимодействовать. Но как? В приложении React мы можем сделать что-то вроде:

```
fetch('http://localhost:3033/api/vacations')
```

Проблема этого подхода в том, что мы собираемся делать запросы такого рода по всему приложению и теперь везде будем вставлять `http://localhost:3033`, что не будет работать при промышленной эксплуатации и, возможно, не сработает на компьютере вашего коллеги, поскольку могут понадобиться другие порты или для сервера тестирования потребуется иной порт и т. д. и т. п. Применять такой подход — напрашиваться на проблемы с конфигурацией. Да, вы можете хранить базовый URL как переменную, которую вы повсеместно используете, но есть способ лучше.

В идеальных с точки зрения вашего приложения условиях все размещается в одном и том же месте: одни и те же протокол, хост и порт для получения HTML, статические ресурсы и API. Это многое упрощает и обеспечивает непротиворе-

чивость вашего исходного кода. Если все приходит из одного и того же места, вы можете запросто опустить протокол, хост и порт и просто вызвать `fetch(/api/vacations)`. Это хороший подход и, к счастью, легко реализуемый!

В конфигурации для CRA есть поддержка прокси, которая позволяет передавать веб-запросы вашему API. Отредактируйте файл `client/package.json` и добавьте следующее:

```
"proxy": "http://localhost:3033",
```

Неважно, где вы это добавите. Я обычно вставляю код между `private` и `dependencies`, поскольку мне нравится видеть эту строку в начале файла. Теперь (до тех пор пока сервер Express запущен на порте 3033) сервер разработки CRA будет передавать запросы к API через сервер Express.

Когда настройка выполнена, используем `effect` (еще один хук React) для получения и обновления данных отпускных туров. Ниже целиком приведен компонент Vacations с хуком `useEffect`:

```
function Vacations() {
  // Настройка состояния
  const [vacations, setVacations] = useState([])

  // Извлечение начальных данных
  useEffect(() => {
    fetch('/api/vacations')
      .then(res => res.json())
      .then(setVacations)
  }, [])
  return (
    <>
      <h2>Туры</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
      </div>
    </>
  )
}
```

Как и прежде, `useState` настраивает компонент состояния так, что в нем есть массив `vacations` и сопутствующий метод для добавления туров. Теперь мы добавили `useEffect`, который вызывает API, чтобы вернуть туры, а затем асинхронно вызывает метод для добавления туров. Обратите внимание, что мы передаем пустой массив в качестве второго аргумента `useEffect`; это сообщает React, что данное

действие должно выполняться однократно при подключении компонента. На первый взгляд такой способ подачи сигнала может показаться странным, но как только вы узнаете больше о хуках, поймете, что все достаточно логично. Более подробную информацию ищите в документации по React hooks (http://bit.ly/34MGSeK).

Методы-перехватчики были добавлены относительно недавно — в версии 16.8 в феврале 2019 г., — поэтому, даже если у вас есть опыт работы с React, хуки вам могут быть незнакомы. Я твердо уверен, что hooks — великолепное нововведение в архитектуре React. Да, сначала они могут показаться чужеродными, но позже вы поймете, что они действительно упрощают компоненты и устраняют некоторые широко распространенные сложные ошибки, связанные с состояниями.

Теперь, когда мы узнали, как получать данные с сервера, рассмотрим, как их отправить обратно.

Отправка информации серверу

У нас уже есть конечная точка API для внесения изменений на сервере; также у нас есть конечная точка для отправки сообщений электронной почты о том, что для тура снова настал сезон. Преобразуем компонент `Vacations`, чтобы выводилась форма для подписки на несезонные туры. Создадим два новых компонента в истинных традициях React: разобьем представление отдельного тура на компоненты `Vacation` и `NotifyWhenInSeason`. Это можно было бы сделать в одном компоненте, но рекомендуемый подход к разработке в React — это множество целевых компонентов вместо гигантских многоцелевых компонентов (тем не менее ради краткости мы не станем помещать эти компоненты в отдельные файлы, я оставляю это в качестве упражнения).

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

function NotifyWhenInSeason({ sku }) {
  return (
    <>
      <i>Уведомить меня, когда начнется сезон на этот тур:</i>
      <input type="email" placeholder="(ваш email)" />
      <button>OK</button>
    </>
  )
}

function Vacation({ vacation }) {
  return (
    <div key={vacation.sku}>
      <h3>{vacation.name}</h3>
      <p>{vacation.description}</p>
      <span className="price">{vacation.price}</span>
      {!vacation.inSeason &&
```

```
        <div>
          <p><i>Этот тур временно вне сезона.</i></p>
          <NotifyWhenInSeason sky={vacation.sku} />
        </div>
      }
    </div>
  )
}

function Vacations() {
  const [vacations, setVacations] = useState([])
  useEffect(() => {
    fetch('/api/vacations')
    .then(res => res.json())
    .then(setVacations)
  }, [])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <Vacation key={vacation.sku} vacation={vacation} />
        )}
      </div>
    </>
  )
}

export default Vacations
```

Теперь, если у вас есть какие-нибудь туры, для которых `inSeason` имеет значение `false` (а у вас они есть, если только вы не изменили базу данных или скрипты инициализации), обновите форму. Сейчас мы подключим кнопку, чтобы выполнять вызов API. Преобразуйте `NotifyWhenInSeason`:

```
function NotifyWhenInSeason({ sku }) {
  const [registeredEmail, setRegisteredEmail] = useState(null)
  const [email, setEmail] = useState('')
  function onSubmit(event) {
    fetch(`/api/vacation/${sku}/notify-when-in-season`, {
      method: 'POST',
      body: JSON.stringify({ email }),
      headers: { 'Content-Type': 'application/json' },
    })
    .then(res => {
      if(res.status < 200 || res.status > 299)
      return alert('Возникли проблемы… пожалуйста, попробуйте еще раз.')
      setRegisteredEmail(email)
    })
```

```
    event.preventDefault()
  }
  if(registeredEmail) return (
    <i>Вам придет уведомление на {registeredEmail}, когда
    начнется сезон на этот тур!</i>
  )
  return (
    <form onSubmit={onSubmit}>
      <i>Уведомить меня, когда начнется сезон на этот тур:</i>
      <input
        type="email"
        placeholder="(ваш email)"
        value={email}
        onChange={({ target: { value } }) => setEmail(value)}
      />
      <button type="submit">OK</button>
    </form>
  )
}
```

Здесь компонент отслеживает две разные величины: адрес электронной почты, когда пользователь его набирает, и окончательное значение, получаемое после того, как была нажата кнопка OK. Первое — это метод, известный как *управляемые компоненты*, вы можете узнать о них больше из документации по формам React (http://bit.ly/2X9P9qh). Второе отслеживается для того, чтобы знать, когда пользователь произвел действие — нажатие кнопки OK, — и иметь возможность соответственно изменить интерфейс пользователя. Мы могли бы использовать простое булево значение «зарегистрировано», но это позволяет нашему UI напомнить пользователям адрес их электронной почты.

Нам следует еще немного поработать над передачей информации API: нужно определить метод (POST), закодировать тело в формат JSON и указать тип контента.

Стоит отметить, что мы принимаем решение о том, какой интерфейс пользователя вернуть. Если пользователь уже зарегистрирован, мы возвращаем простое сообщение, а если нет, отобразим форму. Этот шаблон весьма распространен в React.

Уф! Кажется слишком много труда и очень мало функциональности, причем она достаточно сырая. Обработка ошибок, возникающих при вызове API, работает, но она не слишком-то дружественна к пользователю. К тому же компонент будет помнить, на какие туры мы подписались, но только пока мы находимся на странице. Если мы перейдем куда-нибудь и вернемся назад, то снова увидим форму.

Мы можем предпринять некоторые действия, чтобы сделать этот код более приемлемым. Для начала можем написать обертку API, которая будет выполнять грязную работу по кодированию вывода и установлению ошибок; это, несомненно, принесет свои плоды, когда возрастет количество используемых нами конечных

точек API. Есть много популярных фреймворков обработки форм для React, значительно облегчающих этот тяжкий труд.

Решить задачу запоминания туров, на которые подписался пользователь, несколько сложнее. От чего действительно была бы польза — это от возможности объектам для туров получить доступ к информации (был ли зарегистрирован пользователь). Однако наш целевой компонент ничего не знает о туре; у него есть только SKU. Тема следующего раздела, указывающая на решение этой проблемы, — *управление состоянием*.

Управление состоянием

Большая часть архитектурной работы по планированию и проектированию приложений React касается управления состоянием (state management), но не отдельных компонентов, а того, как они обмениваются состояниями и согласовывают их. В нашем примере приложения происходит обмен состояниями: компонент `Vacations` передает объект `vacation` в компонент `Vacation`, а тот, в свою очередь, передает SKU тура слушателю `NotifyWhenInSeason`. Но до сих пор поток информации лишь спускался вниз по дереву; что произойдет, когда информация должна будет вернуться *вверх*?

Наиболее распространенный подход — передача функций, отвечающих за обновление состояния. Например, у компонента `Vacations` может быть функция для преобразования тура, которую он сможет передать `Vacation`, а тот, в свою очередь, может передать ее `NotifyWhenInSeason`. Когда `NotifyWhenInSeason` вызывает ее, чтобы преобразовать тур, на вершине дерева `Vacations` распознает, что произошли изменения, и это приводит к повторному рендерингу как его самого, так и всех его потомков.

Кажется, что все это утомительно и сложно, порой так и есть, но существуют методы, способные помочь. Они настолько разные и порой такие сложные, что мы не сможем полностью их здесь охватить (тем более эта книга не о React), но я могу указать вам на дополнительную литературу.

- *Redux*. Это первое, что приходит на ум, когда возникает необходимость во всеобъемлющем управлении состоянием в приложениях React. Он был первой официальной архитектурой управления состоянием и до сих пор невероятно популярен. Его концепция крайне проста, но я все равно предпочитаю именно этот фреймворк. Даже если вы не выберете Redux, я рекомендую вам посмотреть бесплатные видеоуроки от его создателя Дэна Абрамова (https://egghead.io/courses/getting-started-with-redux).
- *MobX*. Появился после Redux. Приобрел впечатляющее число сторонников за короткий срок и, вероятно, является вторым по популярности контейнером состояний после Redux. MobX, безусловно, может дать код, который легче читать, но я по-прежнему считаю, что преимущество Redux в том, что он предоставляет

хороший фреймворк при масштабировании вашего приложения, несмотря на большое количество стереотипного кода.

- *Apollo*. Сам по себе не является библиотекой управления состоянием, но часто используется в этом качестве. По сути, это клиентский интерфейс для GraphQL (https://graphql.org/) — альтернативы REST API, — который предлагает значительную интеграцию с React. Если вы используете GraphQL (или интересуетесь им), то вам определенно стоит взглянуть на Apollo.
- *React Context*. Сам React начал с предоставления Context API, который теперь уже встроен в него. Он отчасти выполняет те же функции, что и Redux, но с меньшим количеством шаблонного кода. Однако мне кажется, что React Context не такой надежный и по мере роста приложения Redux становится лучшим выбором.

Если вы только начинаете разработку на React, то сперва можете игнорировать сложность управления состояниями во всем приложении, но очень скоро поймете необходимость более организованного способа управлять состоянием. Когда вы достигнете этой точки, то захотите взглянуть на некоторые из этих вариантов и выберете тот, который вам подходит.

Варианты развертывания

До сих пор мы пользовались встроенным сервером разработки CRA, который является лучшим с точки зрения разработки, и я рекомендую придерживаться его. Однако, когда наступает время развертывания, этот выбор перестает быть подходящим. К счастью, CRA поставляется со встроенным скриптом, создающим оптимизированный под эксплуатацию пакет, после чего у вас появляется широкий выбор. Когда вы будете готовы создать пакет развертывания, просто запустите команду `yarn build` — и будет создан каталог `build`. Все ресурсы в папке `build` статические и могут быть развернуты где угодно.

Мой текущий выбор в области развертывания — поместить сборку CRA в бакет AWS S3 с включенным статическим хостингом веб-сайтов (https://amzn.to/3736fuT). Это далеко не единственный вариант: любой крупный поставщик облачных услуг и CDN предлагает что-нибудь подобное.

В этой конфигурации нам нужно создать маршрутизацию, чтобы вызовы API направлялись на сервер Express и статические файлы нашей сборки были доступны на CDN. Для развертывания с помощью AWS я использую AWS CloudFront (https://amzn.to/2KglZRb), чтобы выполнить эту маршрутизацию; статические ресурсы выдаются из упомянутого выше бакета S3, а запросы API направляются либо на экземпляр EC2 в сервере Express, либо на Lambda.

Другой вариант — позволить Express делать все. Его преимущество — способность объединить целое приложение на одном сервере, что облегчает развертывание и упрощает управление. Возможно, он неидеален с точки зрения масшта-

бирования или производительности, но это прекрасный выбор для маленьких приложений.

Для того чтобы ваше приложение полностью обслуживалось Express, просто скопируйте содержимое папки `build`, созданной при запуске `yarn build`, в каталог `public` в вашем приложении Express. До тех пор пока ваше промежуточное ПО `static` привязано, файл `index.html` будет раздаваться автоматически, и это все, что вам нужно.

Попробуйте: если сервер Express все еще запущен на порте 3033, то вы сможете посетить http://localhost:3033 — такое же приложение, как и то, что предоставляет сервер разработки CRA!

Если вы не понимаете, как работает сервер разработки CRA, то он использует пакет webpack-dev-server, под капотом которого применяется Express! То есть все в конце концов опять возвращается к Express!

Резюме

В этой главе мы лишь поверхностно затронули React и связанные с ним технологии. Если вы хотите углубиться в его изучение, то книга *Learning React*[1] — прекрасное руководство для старта. Кроме того, в этой книге описано управление состоянием с помощью Redux (хотя на текущий момент в ней не охвачены методы-перехватчики) и есть исчерпывающая хорошо изложенная официальная документация по React.

SPA определенно изменили понимание того, как доставлять веб-приложения, и позволили значительно улучшить производительность, особенно на мобильных устройствах. Несмотря на то что Express был написан, когда большая часть HTML-кода отображалась на стороне сервера, он не устарел. Как раз наоборот, необходимость предоставлять API одностраничным приложениям дала Express новую жизнь!

После прочтения этой главы вам должно быть понятно, что суть осталась прежней: данные пересылаются туда и обратно между сервером и клиентом. Изменилась лишь природа этих данных, и нужно привыкнуть к изменению HTML посредством динамических манипуляций DOM.

[1] *Бэнкс А., Порселло Е.* React и Redux. Функциональная веб-разработка. — СПб.: Питер, 2019.

17 Статический контент

Термин *«статический контент»* относится к тем ресурсам, предоставляемым приложением, которые не меняются от запроса к запросу. Вот основные претенденты на эту роль.

- *Мультимедийные файлы*. Изображения, видео- и аудиофайлы. Конечно, можно генерировать файлы изображений на лету (равно как видео- и аудиофайлы, хотя это менее распространенная практика), но большая часть мультимедийных ресурсов — статические.
- *HTML*. Если наше приложение использует представления для динамического рендеринга HTML, то это, как правило, не подпадает под определение статического HTML (хотя ради повышения производительности вы можете генерировать HTML динамически, кэшировать его и выдавать как статический ресурс). Одностраничные приложения, как мы уже убедились, обычно отправляют клиенту один статический HTML-файл, что является самой распространенной причиной обработки HTML как статических ресурсов. Следует отметить, что требовать от клиента использовать расширение `.html`, — не слишком современный подход, так что на сегодня большинство серверов выдают статические HTML-ресурсы без этого расширения (поэтому `/foo` и `/foo.html` вернут одно и то же содержимое).
- *CSS*. Даже если вы используете язык, расширяющий CSS, такой как LESS, Sass или Stylus, браузеру понадобится обычный CSS[1], являющийся статическим ресурсом.
- *JavaScript*. То, что JavaScript выполняется на сервере, не говорит о недопустимости существования клиентского JavaScript. Клиентский JavaScript рассматри-

[1] С помощью определенных трюков JavaScript в браузере можно использовать некомпилированный LESS. Однако такой подход может привести к негативным последствиям в плане производительности, так что я не советую его применять.

вается как статический ресурс. Конечно, тут все менее определенно: а что, если имеется общий код, который мы хотим использовать как на стороне клиента, так и на стороне сервера? Существует ряд способов решения этой проблемы, но отправляемый клиенту JavaScript обычно статический.

- *Двоичные загружаемые файлы.* Это универсальная категория — различные файлы в форматах PDF, ZIP, документы Word, инсталляционные пакеты и т. п.

Если вы всего лишь создаете API, у вас может не быть статических ресурсов. В таком случае можете пропустить данную главу.

Вопросы производительности

Способ обработки статических ресурсов существенно влияет на реальную производительность вашего сайта, особенно если в нем много мультимедийных ресурсов. Два основных фактора, улучшающих производительность: *снижение числа запросов* и *уменьшение размера содержимого*.

Из них снижение числа запросов более критично, особенно для мобильных приложений (накладные расходы на выполнение HTTP-запроса намного выше в сотовых сетях). Снизить число запросов можно двумя способами: объединением ресурсов и кэшированием в браузере.

Объединение ресурсов — преимущественно задача архитектуры и клиентской части: маленькие изображения необходимо по максимуму объединить в один спрайт. После этого с помощью CSS можно задать смещение и размер, чтобы отображалась лишь нужная часть изображения. Я настоятельно рекомендую использовать для создания спрайтов бесплатный сервис SpritePad (http://bit.ly/33GYvwm). Он делает создание спрайтов элементарным, а также генерирует для вас CSS. Нет ничего проще. Вам, вероятно, будет достаточно бесплатной функциональности SpritePad, но, если окажется, что нужно создавать множество спрайтов, подойдут премиальные предложения.

С помощью кэширования в браузере можно уменьшить количество запросов HTTP. Это возможно за счет хранения наиболее часто используемых статических ресурсов в браузере клиента. Хотя браузеры стараются максимально автоматизировать кэширование, никакой магии в этом нет: существует множество вещей, которые вы можете и должны сделать, чтобы браузер смог кэшировать ваши ресурсы.

Наконец, мы можем увеличить производительность путем уменьшения размера статических ресурсов. Некоторые из предназначенных для этого алгоритмов работают *без потерь* (уменьшение размера достигается без потери каких-либо

данных), а некоторые — *с потерями* (уменьшение размера достигается за счет ухудшения качества статических ресурсов). Методы, работающие без потерь, включают минимизацию JavaScript и CSS, а также оптимизацию изображений PNG. Методы, работающие с потерями, включают увеличение степени сжатия JPEG и видеофайлов. Далее в данной главе мы обсудим минимизацию и сборку (также уменьшающую число HTTP-запросов).

Уменьшение количества HTTP-запросов со временем будет не столь важным, так как HTTP/2 становится более обыденным. Одно из главнейших улучшений в HTTP/2 — многократное использование запросов и ответов, что снижает расходы на параллельную загрузку множества ресурсов. За дополнительной информацией обращайтесь к статье Ильи Григорика «Введение в HTTP/2» (http://bit.ly/34TXhxR).

Сети доставки контента

Когда вы переводите свой сайт в эксплуатацию, статические ресурсы должны быть выложены *где-то* в Интернете. Возможно, вы привыкли выкладывать их на том же сервере, где генерируется весь ваш динамический HTML. В нашем примере до сих пор также использовался данный подход: запускаемый командой `node meadowlark.js` сервер Node/Express раздает как все виды HTML, так и статические ресурсы. Однако, если хотите оптимизировать производительность вашего сайта (или заложить эту возможность на будущее), вам понадобится возможность выкладывать статические ресурсы в *сети доставки контента* (content delivery network, CDN). CDN — сервер, оптимизированный для доставки статических ресурсов. Он использует специальные заголовки (о которых мы скоро узнаем больше), включающие кэширование в браузере.

Помимо этого, CDN может включать *географическую оптимизацию* (часто называемую edge caching), то есть статическое содержимое будет доставляться с ближайшего к вашему клиенту сервера. Хотя Интернет очень быстр (конечно, он работает не со скоростью света, но с близкой к ней), данные будут доставляться еще быстрее с расстояния сотни километров, чем тысячи. Экономия времени в каждом отдельном случае незначительна, но, если умножить ее на количество пользователей, запросов и ресурсов, она быстро приобретет внушительные размеры.

На большую часть ваших статических ресурсов будут ссылаться в представлениях HTML (элементы `<link>` ссылаются на CSS-файлы, `<script>` — на файлы JavaScript, теги `<img>` будут ссылаться на изображения, также имеются теги внедрения мультимедийных файлов). Широко распространена практика статических ссылок в CSS, обычно в свойстве `background-image`. И наконец, на статические ре-

сурсы иногда ссылаются в JavaScript, например, код JavaScript может динамически менять или вставлять теги `<img>` или свойства `background-image`.

Не стоит беспокоиться о совместном использовании ресурсов разными доменами (CORS) при применении CDN. Загружаемые в HTML внешние ресурсы не подчиняются правилам CORS: вам требуется активизировать CORS только для ресурсов, загружаемых через Ajax (см. главу 15).

Проектирование для CDN

То, как вы будете использовать CDN, зависит от архитектуры вашего сайта. Большинство сетей доставки контента позволяют устанавливать правила маршрутизации, чтобы определить, куда отправлять входящие запросы. Хотя вы можете установить довольно сложные правила маршрутизации, обычно все сводится к отправке запросов статических ресурсов в одно место (как правило, предоставляемое вашей CDN) и запросов динамических конечных точек (динамических страниц или конечных точек API) — в другое.

Выбор и конфигурация CDN — обширная тема, которую я не буду здесь раскрывать, но дам базовые сведения, что поможет настроить выбранную вами CDN.

Наиболее простой подход к созданию структуры вашего приложения — сделать динамические и статические ресурсы легко различимыми. Это позволит максимально упростить правила маршрутизации CDN. Хотя этого можно достичь с помощью поддоменов (например, динамические ресурсы выдаются с meadowlark.com, а статические — с static.meadowlark.com), данный подход связан с дополнительными трудностями и усложняет локальную разработку. Более простой способ — использование путей запросов: например, все, что начинается с `/public/`, — это статические ресурсы, а все остальное — динамические. Подход может отличаться, если вы генерируете содержимое с помощью Express или используете Express с целью предоставления API для одностраничного приложения.

Веб-сайт с рендерингом на стороне сервера

Если вы используете Express для рендеринга динамического HTML (проще говоря, все, что начинается со `/static/`), это статические ресурсы, а все остальное —динамические. При таком подходе все ваши (динамически генерируемые) URL будут такими, какими вы хотите их видеть (если только они не начинаются со `/static/`, конечно же!), а все ваши статические ресурсы будут иметь префикс `/static/`:

```
<img src="/static/img/meadowlark-logo-1.png" alt="Meadowlark Logo">
Добро пожаловать в <a href="/about">Meadowlark Travel</a>.
```

До сих пор в этой книге мы использовали промежуточное ПО `static`, как если бы все статические ресурсы выкладывались в корневом каталоге. Таким образом, помещая статический ресурс `foo.png` в папку `public`, мы ссылаемся на него по пути URL `/foo.png`, а не `/static/foo.png`. Конечно, можно создать подкаталог `static` внутри существующего каталога `public`, и URL для `/public/static/foo.png` будет `/static/foo.png`, но это не очень разумно. К счастью, промежуточное ПО `static` позволяет нам избежать этого — достаточно указать другой путь при вызове `app.use`:

```
app.use('/static', express.static('public'))
```

Теперь в среде разработки мы можем использовать ту же структуру URL, что и при эксплуатации. Если содержимое папки `public` и CDN синхронизировано, можно ссылаться на статические ресурсы в обоих местах и без проблем переключаться между разработкой и эксплуатацией.

При настройке маршрутизации для CDN (вам нужно будет обратиться к документации по вашей CDN) маршрутизация будет выглядеть следующим образом.

Путь URL	**Пункт назначения/источник маршрутизации**
/static/*	Статическое файловое хранилище CDN
/* (все остальное)	Сервер Node/Express, прокси или балансировщик нагрузки

Одностраничные приложения

Одностраничные приложения, как правило, являются противоположностью вебсайтов с рендерингом на стороне сервера: серверу будет передаваться только API (например, любой запрос, начинающийся с `/api`), все остальное передается в статическое файловое хранилище.

В главе 16 вы увидели, что можно создать сборку для эксплуатации вашего приложения, в которую войдут все статические ресурсы, загружаемые в CDN. Затем нужно будет только удостовериться, что настройка маршрутизации вашего API корректна. Таким образом, у вас будет следующая маршрутизация.

Путь URL	**Пункт назначения/источник маршрутизации**
/api/*	Сервер Node/Express, прокси или балансировщик нагрузки
/* (все остальное)	Статическое файловое хранилище CDN

Узнав, как можно структурировать приложение, чтобы без проблем перейти от разработки к эксплуатации, обратимся к тому, что на самом деле происходит при кэшировании и как это оптимизирует производительность.

Кэширование статических ресурсов

Независимо от того, что вы используете для раздачи статических ресурсов — Express или CDN, стоит разобраться в заголовках HTTP-запросов, которые браузер использует для определения, когда и как кэшировать статические ресурсы.

- `Expires/Cache-Control`. Эти два заголовка информируют ваш браузер о максимальном количестве времени, в течение которого ресурс может храниться в кэше. Браузер воспринимает их всерьез: если они приказывают ему хранить что-либо в течение месяца, он попросту не станет загружать это заново на протяжении месяца, до тех пор пока оно остается в кэше. Важно понимать, что браузер может удалить изображение из кэша до истечения срока по причинам, которые вы не в состоянии контролировать. Например, пользователь может очистить кэш вручную или браузер может удалить ваш ресурс, чтобы освободить место для чаще посещаемых пользователем ресурсов. Вам необходим только один из этих заголовков. Expires поддерживается более широко, так что предпочтительнее использовать его. Если ресурс находится в кэше и срок его хранения еще не истек, браузер вообще не выполнит запрос `GET`, что улучшит производительность, особенно на мобильных устройствах.
- `Last-Modified/ETag`. Обеспечивают своего рода контроль версий: если браузеру необходимо извлечь ресурс, он проверит эти теги *до* загрузки содержимого. Запрос `GET` к серверу все же будет выполнен, но, если возвращаемые в этих заголовках значения продемонстрируют браузеру, что ресурс не менялся, к загрузке файла браузер не перейдет. Как можно догадаться по имени, `Last-Modified` позволяет вам задать дату последнего изменения ресурса. А `ETag` дает возможность использовать произвольную строку, обычно строку с версией или хеш содержимого.

При выдаче статических ресурсов следует использовать заголовок `Expires` и либо `Last-Modified`, либо `ETag`. Встроенное в Express промежуточное ПО `static` устанавливает `Cache-Control`, но не обрабатывает ни `Last-Modified`, ни `ETag`. Поэтому для разработки оно подходит, а для эксплуатации — нет.

Если вы решили выкладывать свои статические ресурсы в CDN, такие как Amazon CloudFront, Microsoft Azure, Fastly, Cloudflare, Akamai или StackPath, то получите бонус: большинство таких деталей будут обработаны за вас. Вы сможете произвести точную настройку, хотя настройки по умолчанию в любом из этих сервисов также хороши.

Изменение статического содержимого

Кэширование значительно улучшает производительность вашего сайта, но не без последствий. В частности, при изменении любого из статических ресурсов клиенты могут не увидеть изменений до тех пор, пока не истечет срок хранения

закэшированных версий в браузере. Google рекомендует кэшировать на срок один месяц, а лучше — один год. Представьте пользователя, который заходит на ваш сайт каждый день через один и тот же браузер: он может увидеть ваши обновления только через год!

Очевидно, что такая ситуация нежелательна, но вы не можете приказать своим пользователям очистить кэш. Решение этой проблемы заключается в *запрете кэширования (cache busting)*. Этот прием даст вам контроль над тем, когда браузер должен перезагружать ресурс. При этом методе к имени файла просто добавляется какая-либо информация о его версии. Когда вы обновляете ресурс, имя ресурса меняется и браузер узнает о необходимости его скачать. Как правило, это равносильно контролю версий ресурса (`main.2.css` или `main.css?version=2`) или добавлению какого-нибудь хеша (`main.e16b7e149dccfcc399e025e0c454bf77.css`). Какой бы метод вы ни использовали, при обновлении ресурса его название меняется и браузер знает, что его нужно загрузить.

С мультимедийными ресурсами можно поступить аналогично. Возьмем наш логотип (`/static/img/meadowlark_logo.png`). Если мы выкладываем его в CDN для максимального увеличения производительности, задавая длительность периода хранения один год, а затем меняем логотип, пользователи могут не увидеть изменений на протяжении года. Однако, если мы переименуем логотип в `/static/img/meadowlark_logo-1.png` и отразим это изменение в HTML, браузеру придется скачать его, поскольку он кажется новым ресурсом.

Если вы остановились на фреймворке одностраничных приложений, таком как `create-react-app` или аналогичном, то на этапе построения будет создана сборка готовых к эксплуатации ресурсов с добавлением к ним хешей.

Если вы начинаете с нуля, то, вероятно, захотите взглянуть на сборщики (это то, что находится под капотом фреймворков одностраничных приложений). JavaScript, CSS и некоторые другие типы статических ресурсов будут объединены в как можно меньшее количество сборок, и результат будет предельно минимизирован. Настройка сборки — обширная тема, но, к счастью, по ней есть много хорошей документации. Ниже приведены наиболее популярные сборщики, доступные на данный момент.

- *Webpack* (https://webpack.js.org/). Один из первых сборщиков, достигших настоящего подъема. У него до сих пор много сторонников. Он очень сложный, и за эту сложность приходится платить: кривая обучаемости крутая. Однако данный упаковщик хорош для обучения азам.
- *Parcel* (https://parceljs.org/). Появился недавно и наделал много шума. Он чрезвычайно хорошо задокументирован, очень быстрый, и, главное, у него самая короткая кривая обучаемости. Он подходит, если нужно сделать работу быстро и без хлопот.
- *Rollup* (https://rollupjs.org/). Находится где-то между Webpack и Parcel. Как Webpack, он надежен и многофункционален. Однако начать работать с ним проще, чем с Webpack, но не так легко, как с Parcel.

Резюме

При всей кажущейся простоте статические ресурсы доставляют массу хлопот. Однако они составляют основную массу передаваемых вашим посетителям данных, так что потраченное на их оптимизацию время окупится с лихвой.

Действенное решение для статических ресурсов, не упоминавшееся ранее, — выложить статические ресурсы в CDN и всегда использовать в представлениях и CSS полный URL к ресурсу. У этого подхода есть преимущество: он очень прост, но, если когда-нибудь вы пожелаете провести недельный хакатон в лесной хижине, где нет доступа к Интернету, у вас будут проблемы!

Тщательно продуманные сборка и минимизация — еще одна сфера, в которой вы можете сэкономить время, если для вашего приложения выигрыш в производительности не оправдывает приложенных усилий. В частности, если на вашем сайте только один или два JavaScript-файла, а все CSS находятся в одном файле, можно вообще отказаться от сборки, но реальные приложения имеют тенденцию расти со временем.

Какой бы метод вы ни выбрали для раздачи статических ресурсов, я советую выкладывать их отдельно, лучше всего в CDN. Если вам кажется, что это хлопотно, уверяю: это совсем не так сложно, как кажется. Особенно если вы предварительно потратите немного времени на систему развертывания так, что развертывание статических ресурсов в одно местоположение, а приложения — в другое будет автоматическим.

Если вас беспокоит стоимость хостинга в CDN, призываю взглянуть на суммы, которые вы сейчас платите за хостинг. Большинство провайдеров хостинга берут большие деньги за трафик, даже если вы об этом не знаете. Однако, если внезапно ваш сайт оказался упомянут на Slashdot и вы испытали на себе слешдот-эффект, вам может прийти совершенно неожиданный счет за услуги хостинга. CDN-хостинг обычно настраивается таким образом, что вы платите только за то, что используете. Приведу пример: сайт для местной компании средних размеров, которым я управляю, использует примерно 20 Гбайт трафика в месяц, при этом плата за размещение статических ресурсов (а это весьма насыщенный медиафайлами сайт) составляет лишь пару долларов в месяц.

Получаемые за счет выкладывания статических ресурсов в CDN выгоды существенны, а стоимость и неудобства от этого минимальны, так что я решительно советую вам выбрать этот путь.

18 Безопасность

Для большинства современных сайтов и приложений есть определенные требования к безопасности. Если вы разрешаете людям входить в систему или храните *информацию, позволяющую установить личность* (ИПУЛ, от англ. personally identifiable information, PII), то захотите обеспечить определенную безопасность для вашего сайта. В этой главе мы обсудим безопасный HTTP (HTTPS), обеспечивающий фундамент, на котором можно построить безопасный сайт, а также механизмы аутентификации, при этом особенно сфокусируемся на аутентификации с использованием сторонних сервисов.

Безопасность — большая тема, она сама по себе могла бы стать поводом для написания книги. По этой причине в нашем издании в центре внимания будет использование существующих моделей аутентификации. Написание собственного модуля аутентификации, конечно же, возможно, но это большое и сложное занятие. Более того, есть веские причины предпочесть сторонние подходы для входа в систему, которые мы обсудим позже в этой главе.

HTTPS

Первый шаг к предоставлению безопасных сервисов — это использование HTTPS. Природа Интернета позволяет перехватить пакеты, передающиеся между клиентами и серверами. HTTPS шифрует эти пакеты, делая доступ к передаваемой информации экстремально сложной задачей. (Я говорю, что это очень сложно, а не невозможно, потому что нет такого понятия, как идеальная безопасность. Однако протокол HTTPS рассматривается как довольно безопасный, например, для банкинга, корпоративной безопасности или здравоохранения.)

Вы можете рассматривать HTTPS как фундамент для обеспечения безопасности своего сайта. Он не обеспечивает аутентификацию, но лежит в ее основе. Например, ваша система аутентификации включает передачу пароля. Если он передается незашифрованным, то никакая дальнейшая сложная аутентификация не обеспечит

безопасность вашей системы. Безопасность прочна ровно настолько, насколько прочным является ее самое слабое звено, и первым звеном в этой цепочке следует считать сетевой протокол.

Протокол HTTPS основывается на том, что у сервера имеется *сертификат открытого ключа* (public key certificate), иногда называемый сертификатом SSL. Современный стандартный формат для сертификатов SSL называется X.509. Идея сертификатов состоит в том, что есть *центры сертификации* (certification authorities, CA), которые их выпускают. Центр сертификации создает *корневые сертификаты* (trusted root certificates), доступные производителям браузеров. Браузеры запускают эти корневые сертификаты во время установки, что налаживает цепочку сертификатов между центром сертификации и браузером. Для того чтобы эта цепочка работала, ваш сервер должен использовать сертификат, выпущенный центром сертификации.

Таким образом, для предоставления HTTPS-соединения вам необходимо получить сертификат в центре сертификации. Что для этого нужно? В целом, вы можете сгенерировать собственный сертификат, получить его у бесплатного центра сертификации или же купить у коммерческого центра сертификации.

Создание собственного сертификата

Создать собственный сертификат несложно, но такой способ подходит лишь для целей разработки и тестирования (и возможно, для интранета). В силу иерархической природы, установленной центрами сертификации, браузеры доверяют только сертификатам, изданным известными центрами сертификации (и вероятно, это не вы). Если ваш сайт использует сертификат, выпущенный неизвестным вашему браузеру центром сертификации, то браузер будет предупреждать вас тревожными фразами о том, что вы устанавливаете соединение с неизвестным, а потому не вызывающим доверия объектом. Для разработки и тестирования это нормально: вы и ваша команда знаете, что создали собственный сертификат, и ожидаете такого же поведения от браузеров. Если бы вы запустили подобный сайт для большого количества конечных потребителей, они бы массово с него уходили.

Если вы контролируете раздачу и установку браузеров, то можете автоматически установить свой корневой сертификат во время установки браузера. Это защитит пользователей данного браузера от предупреждающего сообщения во время соединения с вашим сайтом. Установка такого ключа, однако, непроста и применима лишь к той обстановке, в которой вы контролируете используемые браузеры. Если у вас нет веских оснований применять именно такой подход, то делать этого не стоит, поскольку он может вызывать много проблем.

Для создания собственного сертификата вам понадобится реализация OpenSSL. Таблица 18.1 показывает, как получить эту реализацию.

Таблица 18.1. Получение реализации для разных платформ

Платформа	Инструкции
macOS	brew install openssl
Ubuntu, Debian	sudo apt-get install openssl
Другие Linux	Скачайте с сайта http://www.openssl.org/source; распакуйте tar-архив и следуйте инструкциям
Windows	Скачайте с сайта http://gnuwin32.sourceforge.net/packages/openssl.htm

Если вы пользователь Windows, вам придется указать месторасположение файла конфигурации OpenSSL, и для этого из-за особенностей имен путей в Windows вам могут понадобиться определенные навыки. Наиболее надежный способ: найти местонахождение файла openssl.cnf (обычно он в каталоге share при инсталляции) и, прежде чем запустить команду openssl, установить переменную среды OPENSSL_CNF: SET OPENSSL_CONF=openssl.cnf.

После того как установите OpenSSL, можете создать приватный ключ и публичный сертификат:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem
        -out meadowlark.crt
```

Вас спросят о некоторых деталях, таких как код страны, город и штат, полное доменное имя (fully qualified domain name, FQDN, также называемое *общим именем* или *полностью определенным именем хоста*) и адрес электронной почты. Поскольку этот сертификат будет использоваться для разработки и тестирования, предоставленные вами значения не особо важны (фактически они опциональны, но если вы их пропустите, браузер будет рассматривать сертификат как еще более подозрительный). Общее имя (FQDN) — это то, что использует браузер для определения домена. Если вы, например, используете `localhost`, то можете применить его в качестве вашего FQDN. Либо можете использовать IP-адрес сервера или имя сервера, если оно доступно. Если указанное в сертификате общее имя и используемое в URL доменное имя не совпадают, шифрование по-прежнему будет работать, но ваш браузер предупредит вас о несоответствии.

Если вам интересны детали этой команды, можете прочитать о них на странице документации OpenSSL. Стоит отметить, что у опции `nodes` нет ничего общего с Node, равно как и с множественным числом от слова nodes («узлы»): на самом деле оно означает no DES, и здесь имеется в виду, что приватный ключ не зашифрован с использованием алгоритма для симметричного шифрования DES.

Результатом этой команды будут два файла, `meadowlark.pem` и `meadowlark.crt`. Файл PEM (Privacy-enhanced Electronic Mail, электронная почта с усовершенствованной защитой) — это ваш приватный ключ, он не должен быть доступен клиентской стороне. Файл CRT — это самоподписанный сертификат, который будет передаваться браузеру для установки защищенного соединения.

В качестве альтернативы можете получить бесплатные самоподписанные сертификаты, например, на сайте http://bit.ly/354ClEL.

Использование бесплатного сертификата

HTTPS базируется на доверии, и реальность такова, что самым простым путем завоевания доверия в Интернете является его покупка. Впрочем, не стоит думать, что это все — поддержка торговцев воздухом: установка инфраструктуры безопасности, гарантия сертификатов и поддержание отношений с производителями браузеров — дорогое удовольствие.

Покупка сертификата не единственный законный вариант использования готовых сертификатов. Let's Encrypt, бесплатный автоматизированный центр сертификации, основанный на открытом исходном коде, также неплох. Если вы не вложились в инфраструктуру, предполагающую бесплатные или недорогие сертификаты как часть вашего хостинга (например, AWS), Let's Encrypt (https://letsencrypt.org/) — прекрасный выбор. Единственный его недостаток в том, что максимальный срок действия сертификатов — 90 дней. Этот изъян компенсируется тем, что сертификаты легко обновляются автоматически и Let's Encrypt рекомендует автоматическое обновление каждые 60 дней, чтобы гарантировать, что срок действия сертификатов не истек.

Все крупные производители сертификатов, такие как Comodo и Symantec, предлагают бесплатные пробные сертификаты на период от 30 до 90 дней. Это действенный вариант, если вы хотите протестировать коммерческий сертификат, но нужно будет купить сертификат до окончания пробного периода, если вы захотите убедиться в постоянстве сервиса.

Покупка сертификата

В настоящее время 90 % от примерно 50 корневых сертификатов, распространяемых с крупными браузерами, принадлежат четырем компаниям: Symantec (купившей VeriSign), Comodo Group, Go Daddy и GlobalSign. Покупка напрямую у центра сертификации может влететь в копейку: стоимость сертификата стартует от 300 долларов в год (правда, некоторые предлагают сертификаты менее чем за 100 долларов в год). Более дешевый вариант — обратиться к реселлеру. У него SSL-сертификат можно приобрести всего за 10 долларов в год или даже меньше.

Важно понять, почему вы платите 10, 150 или 300 долларов (а то и больше) за сертификат. Первый важный момент: нет никакой разницы в уровне шифрования,

предложенной сертификатами за 10 и 1500 долларов. Разумеется, продавцы дорогих сертификатов предпочли бы, чтобы вы этого не знали, поэтому их маркетинговые отделы всеми силами стараются скрыть этот факт.

Если вы решили воспользоваться услугами коммерческого поставщика сертификатов, то я рекомендую учесть при выборе следующие три фактора.

- *Поддержка пользователей.* Если у вас возникают проблемы с сертификатом, будь это браузерная поддержка (допустим, клиенты сообщают вам, что их браузер отмечает ваш сертификат как не заслуживающий доверия), проблемы с установкой или трудности с продлением, вы, несомненно, оцените хорошую поддержку пользователей. Это одна из причин купить более дорогой сертификат. Часто хостинг-провайдеры перепродают сертификаты, и, по моему опыту, они предоставляют более высокий уровень поддержки пользователей, поскольку хотят удержать вас и как клиента хостинга.
- *Однодоменные, мультиподдоменные, мультидоменные и групповые сертификаты.* Однодоменные сертификаты обычно самые недорогие. Из-за этого их не стоит считать плохими, но помните: если вы покупаете сертификат на meadowlarktravel.com, то он не будет работать на www.meadowlarktravel.com и наоборот. По этой причине я стараюсь не покупать однодоменные сертификаты, хотя они идеальны для бюджетных решений (вы всегда можете установить перенаправление на нужный домен). Мультиподдоменные сертификаты хороши тем, что один сертификат, который вы купите, покроет meadowlarktravel.com, www.meadowlark.com, blog.meadowlarktravel.com, shop.meadowlarktravel.com и т. д. Обратная сторона медали — нужно знать заранее, какие поддомены вы хотите использовать.

  Если вы знаете, что будете добавлять или использовать разные поддомены в течение года (для которых понадобится поддержка HTTPS), рассмотрите возможность покупки группового сертификата. Это дороже, но такие сертификаты будут работать на любом поддомене и вам не придется указывать, какие это будут поддомены.

  Наконец, есть мультидоменные сертификаты, которые, как и групповые, стоят дороже. Эти сертификаты поддерживают множество доменов, так что, например, у вас может быть meadowlarktravel.com, meadowlarktravel.us, meadowlarktravel.com и варианты с www.
- *Сертификаты с проверкой домена, компании и расширенной проверкой.* Есть три вида сертификатов: домена, компании и с расширенной проверкой. Сертификаты домена, как понятно из названия, обеспечивают уверенность в том, что вы имеете дело с *доменом*, в котором, на ваш взгляд, вы находитесь. Сертификаты компании предоставляют определенные гарантии реальной организации, с которой вы имеете дело. Их получить сложнее: обычно в этом случае требуется собрать немало бумаг. Вы должны предоставить выписку из федерального или регионального реестра о названии вашей компании, физические адреса и другие документы. Разные поставщики сертификатов будут требовать разные бумаги,

поэтому уточните, что конкретно ваш поставщик запрашивает для получения такого сертификата. Наконец, есть сертификаты с *расширенной проверкой*, это своего рода роллс-ройс в мире SSL-сертификатов. Как и сертификаты компании, они проверяют наличие организации, а также требуют высочайших стандартов подтверждения и даже могут запросить дорогостоящий аудит для проверки методов, используемых в целях обеспечения безопасности данных (хотя это, кажется, происходит все реже и реже). Их стоимость — как минимум 150 долларов на один домен.

Я рекомендую либо не такие дорогие сертификаты с проверкой домена, либо сертификаты с расширенной проверкой. Сертификаты компании, проверяющие существование вашей организации, не отображаются браузерами как-то иначе, так что, по моему опыту, если пользователь не исследует сертификат самостоятельно (что редкость), то очевидная разница между ними и сертификатами с проверкой домена будет незаметна. А вот сертификаты с расширенной проверкой обычно отображают для пользователей определенные сведения, говорящие о том, что они имеют дело с законным бизнесом (например, адресная строка отображается зеленым цветом, а название организации показано рядом со значком SSL).

Если вы когда-то имели дело с сертификатами SSL, то могли удивиться, почему я не упомянул гарантию сертификата. Я опустил этот ценовой дифференциатор, поскольку, по сути, это страховка от того, что практически невозможно. Идея такова: если кто-то финансово пострадает из-за трансакции, совершенной на вашем сайте, и сможет *доказать, что это произошло из-за недостаточного шифрования*, гарантия покроет ваши убытки. Если ваше приложение включает финансовые трансакции, то риск того, что кто-то подаст против вас судебный иск из-за финансовых потерь, достаточно высок. Но вероятность того, что это случилось из-за недостаточного шифрования, практически равна нулю. Если бы я искал причину финансовых потерь компании, связанных с онлайн-сервисами, то последнее, что бы сделал, — это доказывал, что SSL-шифрование было взломано. Если вы выбираете среди двух сертификатов, разница между которыми только в цене и страховом покрытии, покупайте более дешевый вариант.

Процесс покупки сертификата начинается с создания приватного ключа (так же как мы делали ранее с самоподписанным сертификатом). Затем вы создаете *запрос подписи сертификата* (certificate signing request, CSR), который будет загружен в течение процесса покупки сертификата (эмитент сертификата предоставит инструкцию, как это сделать). Обратите внимание, что у эмитента сертификата нет доступа к вашему приватному ключу, равно как ваш приватный ключ не передается через Интернет, что обеспечивает его безопасность. Затем эмитент отправляет вам сертификат, у которого будет расширение `.crt`, `.cer` или `.der` (сертификат будет в формате DER — Distinguished Encoding Rules, отсюда и менее распространенное расширение `.der`). Вы также получите любые сертификаты в цепочке сертификатов. Передача этого сертификата по электронной почте безопасна, поскольку он не будет работать без созданного вами приватного ключа.

Разрешение HTTPS для вашего приложения в Express

Вы можете изменить приложение Express так, чтобы ваш сайт выдавался посредством HTTPS. На практике и в эксплуатации это встречается крайне редко. Почему — узнаем в следующем разделе. Однако для продвинутых приложений, тестирования и просто для себя полезно понимать, как использовать HTTPS.

Когда у вас есть ваш приватный ключ и сертификат, использовать его в приложении несложно. Вернемся к тому, как мы создавали сервер:

```
app.listen(app.get('port'), () => {
  console.log(`Express запущен в режиме ${app.get('env')} ` +
    `на порте+ ${app.get('port')}.`)
})
```

Переключение на HTTPS довольно простое. Я рекомендую разместить ваш приватный ключ и сертификат SSL в подкаталоге, называемом `ssl` (хотя часто их хранят в корневом каталоге проекта). Затем просто используйте модуль `https` вместо `http` и передайте объект `options` методу `createServer`:

```
const https = require('https')
const fs = require('fs') // Обычно в начале файла.

// ...остальные настройки приложения.
const options = {
    key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem'),
    cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt'),
}

const port = process.env.PORT || 3000
https.createServer(options, app).listen(port, () => {
    console.log(`Express запущен в режиме ${app.get('env')} ` +
        `на порте + ${port}.`)
})
```

Это все, что нужно. Если вы по-прежнему запускаете сервер на порте 3000, можете подключиться к https://localhost:3000. Если попытаетесь подключиться к http://localhost:3000, он просто отключится по тайм-ауту.

Примечание о портах

Знали вы или нет, но во время посещения сайта вы *всегда* подключаетесь к определенному порту, даже если это не указано в URL. Если вы не указываете порт, для HTTP будет использоваться порт 80. Собственно говоря, большинство браузеров попросту не отображают номер порта, если вы явно укажете порт 80. Например, зайдите на http://www.apple.com:80, и, по всей вероятности, когда страница будет загружаться, браузер попросту уберет `:80`. Он по-прежнему будет подключаться к порту 80 — это произойдет по умолчанию.

Аналогично стандартный порт для HTTPS — 443. Поведение браузера в этом случае аналогичное: если вы подключаетесь к https://www.google.com:443, большинство браузеров просто не будут отображать `:443`, но это именно тот порт, через который они подключаются.

Если вы не используете порт 80 для HTTP или 443 для HTTPS, нужно явно указать порт и протокол для корректного подключения. Не стоит запускать HTTP и HTTPS на одном порте (технически это возможно, но нет ни одной разумной причины делать это, к тому же реализация данного процесса очень сложная).

Если вы хотите запустить HTTP-приложение на порте 80 или приложение HTTPS на порте 443, когда нет надобности указывать порт явно, рассмотрите две вещи. Первая: во многих системах по умолчанию есть веб-сервер, запущенный на порте 80. Вторая: в большинстве операционных систем порты 1–1023 требуют повышенных привилегий для открытия. Например, в Linux или macOS, если вы попытаетесь запустить свое приложение на порте 80, запуск завершится с ошибкой `EACCES`. Для запуска на порте 80 или 443 (или любом другом порте меньше 1024) вам придется повысить свои привилегии посредством использования команды `sudo`. Если у вас нет прав администратора, то запустить сервер напрямую на порте 80 или 443 не выйдет.

Если вы не управляете собственными серверами, у вас, вероятно, нет корневого доступа к учетной записи вашего хостинга: так что же произойдет, когда вы захотите запустить на порте 80 или 443? В целом, у хостинг-провайдеров есть нечто наподобие прокси-сервиса, который работает с повышенными привилегиями. Он передаст запросы вашему приложению, работающему на непривилегированном порте. Подробнее об этом мы поговорим в следующем разделе.

HTTPS и прокси

Как мы уже видели, HTTPS очень просто использовать с Express. Однако если вы пожелаете масштабировать свой сайт для обработки большего количества трафика, то захотите использовать прокси-сервер, такой как NGINX (см. главу 12). Если ваш сайт запущен в общей среде хостинга, то там наверняка есть прокси-сервер, который будет маршрутизировать запросы к вашему приложению.

Если вы используете прокси-сервер, то клиент (браузер пользователя) будет общаться с *прокси-сервером*, а не с вашим сервером. Прокси-сервер, в свою очередь, скорее всего, будет связываться с вашим приложением посредством обычного HTTP, поскольку ваше приложение и прокси-сервер будут запущены вместе в защищенной сети. Вы можете услышать фразу, что HTTPS *прерывается* на прокси-сервере или прокси-сервер выполняет завершение SSL.

В большинстве случаев, когда вы или ваш хостинг-провайдер корректно сконфигурировали прокси-сервер для обработки запросов HTTPS, вам не понадобится делать какую-то дополнительную работу. Наличие исключений из этого правила

будет зависеть от того, нужно ли вашему приложению обрабатывать как безопасные, так и небезопасные запросы.

Есть три решения этой проблемы. Первая — сконфигурировать ваш прокси-сервер для перенаправления всего HTTP-трафика на HTTPS, по существу вынуждая все коммуникации с вашим приложением вести через HTTPS. Данный подход становится все более распространенным, поскольку это довольно простое решение проблемы.

Второй подход — как-то передавать протокол, используемый на стороне связи «клиент — прокси», на сервер. Обычный способ — передача через заголовок `X-Forwarded-Proto`. Например, для установки этого заголовка в NGINX:

```
proxy_set_header X-Forwarded-Proto $scheme;
```

Затем в своем приложении вы можете протестировать, был ли протокол HTTPS:

```
app.get('/', (req, res) => {
  // Следующее, по существу, эквивалентно: if(req.secure).
  if(req.headers['x-forwarded-proto']==='https') {
    res.send('линия безопасна');
  } else {
    res.send('вы не защищены!');
  }
})
```

В NGINX есть отдельный блок конфигурации server для HTTP и HTTPS. Если вы не установите X-Forwarded-Protocol в блоке конфигурации, соответствующем HTTP, вы, таким образом, подвергаетесь опасности подмены заголовка клиентом и обмана вашего приложения, которое будет считать соединение безопасным, даже если на самом деле это не так. Если вы используете этот подход, обязательно убедитесь в том, что всегда устанавливаете заголовок X-Forwarded-Protocol.

Когда вы используете прокси, Express обеспечивает некоторые удобные свойства, которые делают прокси более «прозрачным» (его как будто нет, а предоставляемые им преимущества сохранились). Чтобы воспользоваться этим, укажите Express доверять прокси посредством `app.enable('trust proxy')`. Когда вы это сделаете, `req.protocol`, `req.secure` и `req.ip` будут относиться к соединению клиента к прокси, а не к вашему приложению.

Межсайтовая подделка запроса

Атаки межсайтовой подделки запроса (Cross-Site Request Forgery, CSRF) пользуются тем, что пользователи обычно доверяют своему браузеру и посещают множество сайтов в течение одной и той же сессии. В ходе атаки скрипт CSRF на сайте злоумышленника отправляет запросы другому сайту: если вы залогинены на

другом сайте, сайт злоумышленника может успешно получить доступ к безопасным данным с другого сайта.

Для предотвращения атак CSRF у вас должен быть способ убедиться, что запрос пришел от вашего сайта. Это можно сделать с помощью передачи уникального токена браузера. Когда браузер отправляет форму, сервер проверяет токены, чтобы убедиться, что они совпадают. Промежуточное ПО `csurf` само обрабатывает создание и проверку токена. Все, что вы должны сделать, — убедиться в том, что токен включен в запросы серверу. Установите промежуточное ПО `csurf` (`npm install csurf`), затем подключите его и добавьте токен к `res.locals`. Убедитесь, что промежуточное ПО `csurf` подключается после `body-parser`, `cookie-parser` и `express-session`:

```
// Это нужно вставить после body-parser,
// cookie-parser и express-session.
const csrf = require('csurf')

app.use(csrf({ cookie: true }))
app.use((req, res, next) => {
  res.locals._csrfToken = req.csrfToken()
  next()
})
```

Промежуточное ПО `csurf` добавляет метод `csurfToken` к объекту запроса. Нам не нужно назначать его для `res.locals` — мы могли бы просто передать `req.csurfToken()` явно каждому представлению, которое в этом нуждается.

Обратите внимание: несмотря на то что сам пакет называется csurf, в названиях большинства переменных и методов присутствует csrf без u. В этом легко запутаться, так что будьте внимательны с гласными!

Сейчас на всех ваших формах (и в вызовах AJAX) нужно предоставить поле с именем `_csrf`, которое должно совпадать со сгенерированным токеном. Посмотрим, как бы мы добавили это к одной из наших форм:

```
<form action="/newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{{_csrfToken}}">
  Name: <input type="text" name="name"><br>
  Email: <input type="email" name="email"><br>
  <button type="submit">Отправить</button>
</form>
```

Промежуточное ПО `csurf` будет обрабатывать все остальное: если поле содержит поля, но нет действительного поля `_csrf`, оно вызовет ошибку (убедитесь, что у вас есть обработка ошибок в промежуточном ПО!). Теперь удалите скрытое поле и посмотрите, что получится.

Если у вас есть API, вряд ли вам захочется, чтобы промежуточное ПО csurf с ним взаимодействовало. Если хотите ограничить доступ к вашему API с остальных сайтов, нужно посмотреть функциональность API key в таких библиотеках, как connect-rest. Для предотвращения взаимодействия csurf с промежуточным ПО подключите его до того, как подключите csurf.

Аутентификация

Аутентификация — большая и сложная тема. К сожалению, это еще и важная часть большинства нетривиальных веб-приложений. Опираясь на свой жизненный опыт, я обязан вас предупредить: *не пытайтесь делать это самостоятельно*. Если на вашей визитке не написано «Эксперт по безопасности», вы наверняка не готовы к сложному анализу, задействованному в разработке системы безопасной аутентификации.

Я не говорю, что вам не стоит даже пробовать понять систему безопасности, использованную в вашем приложении. Я лишь рекомендую не пытаться строить ее самостоятельно. Изучайте открытый исходный код техник аутентификации, которые я вам собираюсь порекомендовать. Это определенно даст некоторое понимание того, почему вы не захотите браться за эту задачу без посторонней помощи!

Аутентификация или авторизация

Эти два термина очень часто используются как взаимозаменяемые, однако между ними есть разница. *Аутентификация* относится к проверке подлинности пользователя, то есть позволяет убедиться, что он именно тот, за кого себя выдает. *Авторизация* относится к определению того, к чему пользователь может получить доступ. Например, покупатели могут быть авторизованы для доступа к своей учетной записи, тогда как сотрудник Meadowlark Travel может быть авторизирован для доступа к учетной записи другого пользователя или к заметкам продавца.

Зачастую для аутентификации используется аббревиатура authN, а для авторизации — authZ.

Обычно (но не всегда) сначала выполняется аутентификация, а затем устанавливается авторизация. Авторизация может быть простой (авторизирован/не авторизирован), широкой (пользователь/администратор) или гибкой, определяющей привилегии чтения, записи, удаления и обновления в зависимости от разных типов учетных записей. Сложность системы авторизации зависит от типа приложения, которое вы пишете.

Поскольку авторизация зависит от особенностей вашего приложения, в этой книге я приведу лишь приблизительный сценарий, используя очень широкую схему аутентификации (покупатель/сотрудник).

Проблема с паролями

Суть проблемы с паролями в том, что любая система безопасности настолько сильна, насколько сильно ее самое слабое звено. В случае использования паролей пользователю нужно их придумывать — это и есть слабейшее звено. Людям присуща пресловутая привычка придумывать откровенно небезопасные пароли. Я пишу эти строки в 2018 г., когда анализ дыр в системе безопасности выявил наиболее популярный пароль — 12345, на втором месте — password. Даже в 2018 г., когда осознание вопросов безопасности вышло на новый, более высокий уровень, люди по-прежнему выдумывают вопиюще плохие пароли. Если политика паролей требует использовать в них, например, заглавную букву, цифру и знак препинания, это часто приводит к такому результату — Password1!.

Даже анализ списка распространенных паролей не решает проблему. Люди начинают записывать свои высококачественные пароли в записных книжках, хранить их в незашифрованном виде в файлах на компьютерах или отправлять себе по электронной почте.

Эту проблему вы как разработчик приложения вряд ли сможете решить. Однако в ваших силах сделать то, что поспособствует созданию более безопасных паролей. Одним из таких решений может быть перекладывание ответственности при аутентификации на сторонний сервис. Второе решение — сделать вход в систему дружественным к сервисам менеджмента паролей, таким как 1Password, Bitwarden и LastPass.

Сторонняя аутентификация

Сторонняя аутентификация основана на том, что практически у каждого в Интернете есть учетная запись как минимум на одном крупном сервисе, таком как Google, Facebook, Twitter или LinkedIn. Все эти сервисы предоставляют механизм аутентификации и идентификации ваших пользователей через их сервисы.

Стороннюю аутентификацию часто называют федеративной или делегированной аутентификацией. Эти термины в основном взаимозаменяемые, хотя федеративная аутентификация обычно ассоциируется с языком разметки декларации безопасности (security assertion markup language, SAML) и OpenID, а делегированная часто ассоциируется с OAuth.

У сторонней аутентификации есть три основных преимущества. Во-первых, ваше бремя аутентификации облегчено. Вам не нужно беспокоиться об аутентификации

индивидуальных пользователей. Все, о чем вам придется думать, — это о взаимодействии с доверенным сторонним сервисом. Во-вторых, сторонняя аутентификация снижает *усталость от паролей* — стресс, ассоциируемый с наличием слишком большого количества учетных записей. Я использую LastPass (http://lastpass.com/) и вот сейчас проверил свой склад паролей: у меня их почти 400. Как у профессионала в области технологий, у меня эта цифра наверняка больше, чем у среднестатистического интернет-пользователя. Однако даже у обычного интернет-пользователя десятки, а то и сотни аккаунтов. В-третьих, сторонняя аутентификация делается *«без шума и пыли»*: она позволяет пользователям начать использовать сайт быстрее с помощью уже имеющихся данных доступа. Если пользователи видят, что должны создать *еще одну* пару «имя пользователя и пароль», они часто просто уходят.

Если вы не сторонник менеджера паролей, вероятно, будете использовать один и тот же пароль для большинства сайтов. У многих людей есть «безопасный» пароль, который используется для банкинга и чего-то подобного, и «небезопасный» — для всех прочих мест. Проблема такого подхода в том, что, если хотя бы у *одного* из этих сайтов есть дыра в системе безопасности и пароль станет известен, хакеры начнут пытаться использовать тот же пароль в остальных сервисах. Это как класть все яйца в одну корзину.

У сторонней аутентификации есть и отрицательные моменты. В это трудно поверить, но все-таки *есть* люди, у которых нет аккаунта в Google, Facebook, Twitter и LinkedIn. Далее, часть людей, у которых *есть* такие аккаунты, проявляет особую осторожность и желает сохранить приватность, поэтому не хочет использовать эти данные доступа для регистрации на вашем сайте. Многие сайты решают эту проблему рекомендацией применять существующие учетные записи, но те, у кого их нет, или те, кто не хочет использовать их для доступа к вашему сервису, могут сделать для него новую учетную запись.

Хранение пользователей в вашей базе данных

Независимо от того, полагаетесь вы на сторонний сервис для аутентификации ваших пользователей или нет, вы можете хранить записи пользователей в своей базе данных. Например, если вы используете Facebook для аутентификации, это только проверяет личность пользователя. Если же нужно сохранять особые для этого пользователя настройки, вы не будете применять Facebook — вам придется хранить информацию о пользователе в собственной базе данных. Кроме того, вы наверняка пожелаете ассоциировать адрес электронной почты с этим пользователем, а он может не захотеть применять тот же почтовый ящик, что и для Facebook (или какого-то другого стороннего сервиса, который вы используете). Наконец, хранение информации в собственной базе данных позволяет вам выполнять аутентификацию самим, если вы хотите иметь такую опцию.

Так что создадим для наших пользователей модель `models/user.js`:

```
const mongoose = require('mongoose')

const userSchema = mongoose.Schema({
  authId: String,
  name: String,
  email: String,
  role: String,
  created: Date,
})

const User = mongoose.model('User', userSchema)
module.exports = User
```

Произведем преобразование `db.js`, применяя соответствующие абстракции (если вы пользуетесь PostgreSQL, то привязку этих абстракций я оставлю в качестве упражнения):

```
const User = require('./models/user')
module.exports = {
  //...
  getUserById: async id => User.findById(id),
  getUserByAuthId: async authId => User.findOne({ authId }),
  addUser: async data => new User(data).save(),
}
```

Напомню, что у каждого объекта в MongoDB есть уникальный идентификатор, сохраненный в свойстве `_id`. Однако этот идентификатор контролируется MongoDB и нам нужен какой-то способ сопоставить запись пользователя со сторонним идентификатором, поэтому у нас есть собственное свойство идентификатора, названное `authId`. Поскольку мы будем использовать несколько стратегий аутентификации, для предотвращения столкновений этот идентификатор будет представлять собой комбинацию имени стратегии и стороннего идентификатора. Например, у пользователя Facebook это может быть `authId facebook:525764102`, тогда как у пользователя Twitter — `authId twitter:376841763`.

В нашем примере будем использовать две роли: «покупатель» и «сотрудник».

Аутентификация или регистрация и пользовательский опыт

Аутентификация — проверка личности пользователя посредством либо доверенного стороннего сервиса, либо данных доступа, предоставляемых пользователю (таких как имя пользователя и пароль). Регистрация — это процесс, в результате которого пользователь получает учетную запись на вашем сайте (с нашей точки зрения, регистрация — это создание для этого пользователя записи в базе данных).

Когда пользователи заходят на ваш сайт впервые, им должно быть ясно, что они регистрируются. Используя стороннюю систему аутентификации, мы могли бы зарегистрировать их без их ведома, если они успешно аутентифицируются через сторонний сервис. Но это плохая практика, так как пользователи должны понимать, что они регистрируются на вашем сайте (независимо от того, проходят они стороннюю аутентификацию или нет), и иметь ясный механизм отмены членства.

Одна из часто возникающих ситуаций, которую вам следует рассмотреть, — это путаница сторонних сервисов. Пользователь, который зарегистрировался в январе с использованием Facebook, а затем вернулся в июле и столкнулся с предложением войти в систему через Google, Facebook, Twitter или LinkedIn, давно мог забыть, посредством какого сервиса регистрировался первый раз. Это одна из ловушек сторонней аутентификации, и здесь вы фактически бессильны. Это еще один повод запросить у пользователей адрес электронной почты: таким образом вы даете им возможность найти учетную запись по адресу электронной почты и отправить на него письмо, указав при этом, какой сервис использовался при аутентификации.

Если вы хорошо знаете, какие соцсети предпочитают ваши пользователи, можете воспользоваться главным сервисом аутентификации. Например, если вы уверены, что у большинства ваших пользователей есть учетная запись Facebook, можете сделать большую кнопку, на которой написано: «Войти с Facebook». Затем, используя маленькие кнопки или просто текстовые ссылки, написать: «Войти с Google, Twitter или LinkedIn». Этот подход поможет сократить количество случаев путаницы сторонних сервисов.

Passport

Passport — очень популярный и надежный модуль аутентификации для Node/Express. Он не связан с каким-либо механизмом аутентификации, скорее, основан на подключаемых *стратегиях* аутентификации (включая локальную стратегию, если вы не хотите использовать стороннюю аутентификацию). Поток аутентификационной информации может быть непосильным для восприятия, так что начнем с одного механизма аутентификации, а другие добавим позже.

Важно понять одну деталь: со сторонней аутентификацией ваше приложение *никогда не получит пароль*. Все обрабатывается полностью сторонним сервисом. Это хорошо, поскольку бремя обеспечения безопасности и хранения паролей ложится не на вас, а на этот сторонний сервис[1].

Таким образом, весь процесс полагается на перенаправления (по крайней мере должен, если ваше приложение не получает пароли пользователей от сторонних сервисов). Возможно, поначалу вас мог смутить тот факт, что, передав *локальные*

[1] Маловероятно, что сторонний сервис хранит пароли. Пароль можно проверить, сохранив «соленый» хеш, то есть трансформированный в одну сторону пароль. То есть, когда вы генерируете хеш из пароля, вы не можете восстановить пароль. «Соление» хеша обеспечивает дополнительную защиту от определенных видов атак.

URL-адреса стороннему сервису, можно по-прежнему успешно проходить аутентификацию (в конце концов, сторонний сервер, обрабатывающий ваш запрос, не знает о вашем *локальном* адресе). Это работает, поскольку сторонний сервис инструктирует *ваш браузер* о перенаправлении, а браузер находится внутри вашей сети, таким образом, ему доступно перенаправление к локальным адресам.

Базовый процесс изображен на рис. 18.1. Эта схема показывает важный поток функциональности, давая понять, что аутентификация происходит на стороннем сайте. Наслаждайтесь простотой схемы — дальше будет гораздо сложнее.

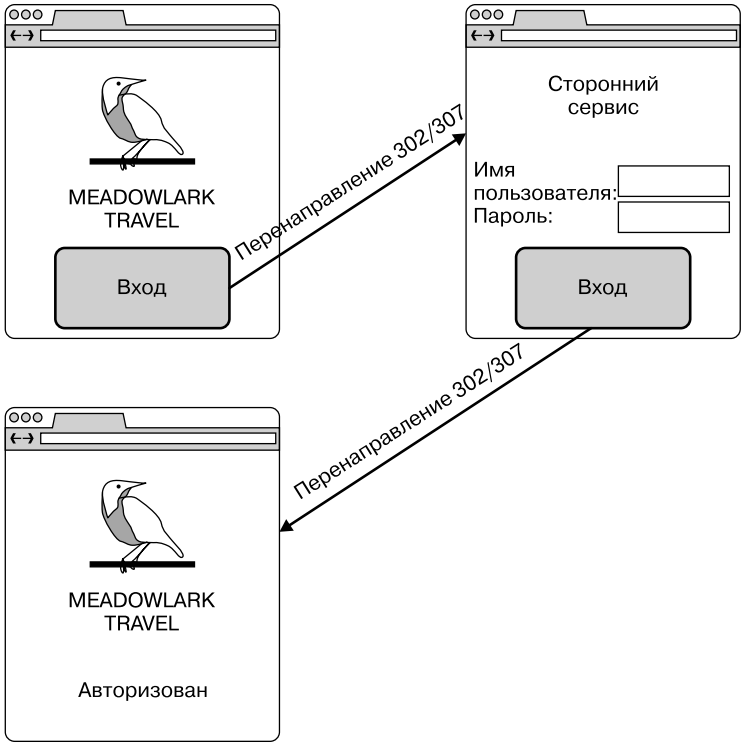

Рис. 18.1. Процесс сторонней аутентификации

Когда вы используете Passport, то выполняете четыре шага, за которые отвечает ваше приложение. Рассмотрим более детально отображение хода сторонней аутентификации (рис. 18.2).

Для простоты используем Meadowlark Travel в роли вашего приложения и Facebook — для механизма сторонней аутентификации. Рисунок 18.2 показывает, как пользователь заходит со страницы входа на безопасную страницу Настройки пользователя (страница Настройки пользователя здесь используется исключительно в целях иллюстрации — это может быть любая требующая аутентификации страница сайта).

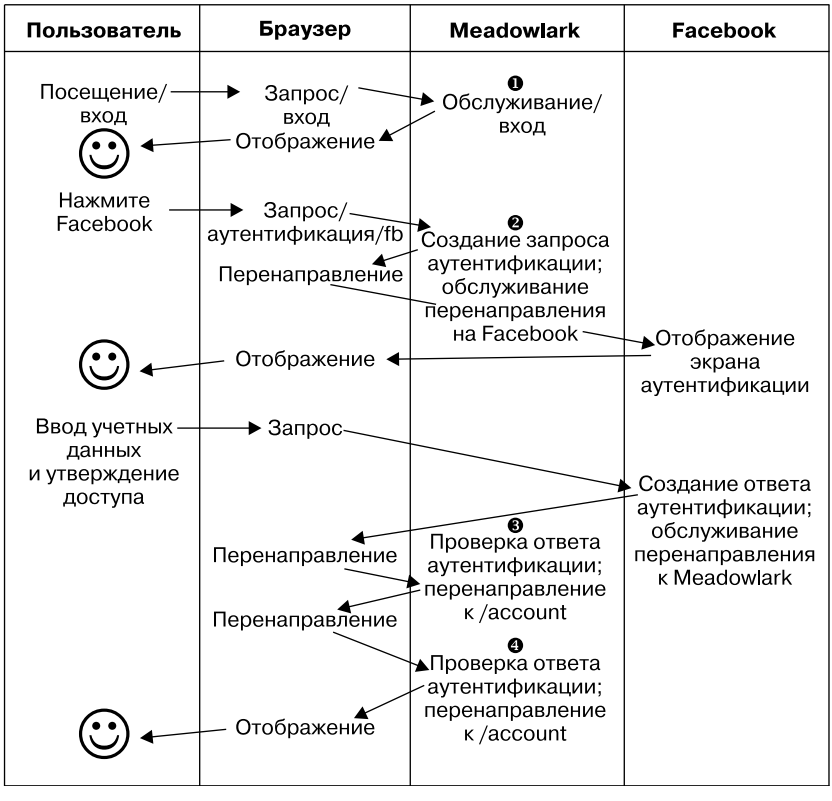

Рис. 18.2. Детальное отображение процесса сторонней аутентификации

Схема демонстрирует детали, о которых вы обычно не думаете, но их очень важно понимать в данном контексте. В частности, когда вы посещаете URL, вы не делаете запрос на сервере — это работа браузера. При этом браузер может выполнять три вещи: делать HTTP-запрос, отображать ответ и осуществлять перенаправление, в котором, в свою очередь, может быть другое перенаправление.

В колонке Meadowlark вы видите четыре шага, за которые отвечает ваше приложение. К счастью, мы используем Passport (и подключаемые стратегии) для реализации деталей этих шагов, в противном случае эта глава была бы гораздо длиннее.

Прежде чем углубиться в детали реализации, рассмотрим шаги чуть подробнее.

- *Страница входа в систему*. Страница входа — место, где пользователь может выбрать метод входа. Если вы используете стороннюю аутентификацию, это просто кнопка или ссылка. Если вы применяете локальную аутентификацию, то будут видны поля для ввода имени пользователя и пароля. Если пользователь попытается получить доступ к URL, требующему аутентификации (такому как `/account` в нашем примере), не будучи залогиненным, то, возможно, это будет

та страница, на которую вы захотите перенаправлять (альтернативный путь — можете перенаправлять на страницу Не авторизирован со ссылкой на страницу входа в систему).

- *Создать запрос аутентификации.* На этом шаге вы создаете запрос, который будет отправлен стороннему сервису через перенаправление. Детали этого запроса сложны и зависят от стратегии аутентификации. Здесь всю тяжелую работу выполняют Passport и плагин стратегии. Запрос аутентификации включает защиту от атаки посредника (man in the middle) и от других методов, которыми может воспользоваться злоумышленник. Обычно запрос аутентификации непродолжительный, поэтому вы не можете его сохранить и надеяться, что будете использовать впоследствии: это помогает предотвратить атаки, ограничивая время, в течение которого злоумышленник может атаковать. В этом месте вы также можете запрашивать дополнительную информацию от механизма сторонней авторизации. Например, довольно часто запрашивается имя пользователя и, возможно, адрес электронной почты. Учтите, что чем больше информации вы запрашиваете у пользователя, тем меньше вероятность авторизации им вашего приложения.
- *Проверить ответ аутентификации.* Предположим, пользователь авторизовал ваше приложение и вы возвращаетесь с подтвержденным ответом аутентификации от стороннего сервиса, что подтверждает личность вашего пользователя. Еще раз: детали этой проверки сложны и будут обрабатываться Passport и плагином стратегии. Когда ответ аутентификации указывает, что пользователь не авторизован (были введены неправильные данные доступа, или ваше приложение не было авторизовано пользователем), вам следует выполнить перенаправление на соответствующую страницу (либо обратно на страницу входа, либо на страницу Не авторизован или Не удалось авторизовать). В ответ на аутентификацию будут включены идентификатор для пользователя, уникальный для *конкретного стороннего сервиса*, а также любые другие данные, которые вы запрашивали на шаге 2. Для осуществления шага 4 мы должны «запомнить», что пользователь авторизован. Простой способ сделать это — установить переменную сессии, содержащую идентификатор пользователя, который указывает, что эта сессия уже авторизована (можно использовать и cookie, хотя я рекомендую применять сессии).
- *Проверить авторизацию.* На шаге 3 мы записали идентификатор пользователя в эту сессию. Его наличие позволяет получить объект пользователя из базы данных, содержащий информацию о том, какие функции этот пользователь авторизован выполнять. Таким образом, нет необходимости выполнять стороннюю аутентификацию для каждого запроса. Эта задача проста, и для ее решения нам больше не нужен Passport — у нас есть собственный объект пользователя, содержащий наши личные правила аутентификации. Если этот объект недоступен, значит, запрос не авторизован и мы можем перенаправлять пользователя на страницу входа или Не авторизован.

Использование Passport для аутентификации — трудоемкий процесс. Тем не менее аутентификация является важной частью нашего приложения, и было бы разумно потратить время на то, чтобы сделать ее правильно. Есть проекты, такие как LockIt, которые пытаются предоставить практически готовое к употреблению решение. Другой набирающий популярность вариант — Auth0 (https://auth0.com/). Он надежен, но не так прост в установке, как LockIt (http://bit.ly/lock_it). Однако, если вы хотите использовать LockIt, Auth0 или аналогичные решения максимально эффективно, вам нужно разобраться в деталях аутентификации и авторизации, для чего, собственно, эта глава и была написана. Кроме того, если вы когда-нибудь заходите модифицировать решения аутентификации в соответствии со специфическими требованиями, Passport послужит отличной стартовой площадкой.

Установка Passport

Чтобы не усложнять, начнем с одного провайдера аутентификации. Выберем Facebook. Перед тем как установить Passport и стратегию Facebook, придется выполнить небольшое конфигурирование в самом Facebook. Для аутентификации в Facebook нам понадобится *приложение Facebook*. Если у вас уже есть подходящее приложение Facebook, можете использовать его либо создайте новое специально для аутентификации. Если возможно, используйте официальную учетную запись Facebook своей организации для создания вашего приложения. Таким образом, если вы работаете в Meadowlark Travel, используйте учетную запись Facebook компании Meadowlark Travel для создания приложения (вы всегда можете добавить свою личную учетную запись как администратора приложения для простоты администрирования). В целях тестирования можно использовать и личную учетную запись Facebook, но использование ее для готового продукта непрофессионально и, вполне вероятно, покажется вашим пользователям подозрительным шагом.

Детали администрирования приложением Facebook часто меняются, поэтому я опущу подробности. Обратитесь к документации разработчика Facebook (http://bit.ly/372bc7c), если вам нужно получить подробную информацию о создании и администрировании вашего приложения.

Для разработки и тестирования вам необходимо связать доменное имя разработки/тестирования с приложением. Facebook позволяет использовать localhost (с номерами портов), что отлично подходит для тестирования. В качестве альтернативы можно указать локальный IP-адрес, что будет полезно, если вы используете виртуальный сервер или другой сервер в своей сети для тестирования. Здесь важно, что URL, который вы вводите в браузере для тестирования приложения (например, http://localhost:3000), связан с приложением Facebook. В настоящее время вы можете связать с приложением Facebook только один домен. Если нужно связать больше, понадобится создать несколько приложений (например, это может быть Meadowlark Dev, Meadowlark Test и Meadowlark Staging, а конечное приложение может называться просто Meadowlark Travel).

Когда вы сконфигурируете приложение, вам понадобятся его уникальный идентификатор и секретный ключ, которые вы найдете на странице управления приложениями Facebook для этого приложения.

Вас может расстроить сообщение от Facebook вроде «Данный URL не разрешен конфигурацией приложения». Это значит, что имя хоста и порт в URL обратного вызова не совпадают с теми, которые вы настроили в своем приложении. Если посмотрите на URL в адресной строке вашего браузера, то увидите закодированный URL, который должен дать ключ к разгадке. Например, если я использую 192.168.0.103:3443 и получаю такое сообщение, я смотрю на адрес URL. Если вижу в строке запроса redirect_uri=https3A%2F2F192.68.0.103%3A3443%2Fauth%2Ffacebook%2Fcallback, то могу быстро определить ошибку: в имени хоста я использовал 68 вместо 168.

Теперь установим Passport и стратегию аутентификации Facebook:

```
npm install passport passport-facebook
```

Далее должен быть многострочный код аутентификации (особенно если поддерживаются множественные стратегии), но мы не хотим загромождать им `meadowlark.js`. Вместо этого создадим модуль, называемый `lib/auth.js`. Это будет большой файл, так что разобьем работу на части (полный пример см. в разделе `ch18` прилагаемого репозитория). Начнем с импорта и двух методов, требуемых Passport, `serializeUser` и `deserializeUser`:

```
const passport = require('passport')
const FacebookStrategy = require('passport-facebook').Strategy

const db = require('../db')

passport.serializeUser((user, done) => done(null, user._id))

passport.deserializeUser((id, done) => {
  db.getUserById(id)
    .then(user => done(null, user))
    .catch(err => done(err, null))
})
```

Passport использует `serializeUser` и `deserializeUser` для установки соответствия запросов аутентификации пользователя, позволяя вам использовать любой метод хранения. В данном случае мы собираемся сохранять только идентификатор в базе данных (свойство `_id`) в этой сессии. Используемый нами способ осуществляет сериализацию и десериализацию немного неправильно: мы просто храним идентификатор пользователя в сессии. Затем при необходимости можем получить объект `user` с помощью поиска этого идентификатора в базе данных.

После того как эти два метода будут реализованы, пока сессия активна и пользователь успешно прошел аутентификацию, `req.session.passport.user` будет соответствовать объекту `user` как полученному из базы данных.

Далее нужно выбрать, что экспортировать. Для включения функциональности Passport нужно провести два отдельных мероприятия: инициализировать Passport и зарегистрировать маршруты, которые будут обрабатывать аутентификацию и перенаправленные обратные вызовы от наших сервисов сторонней аутентификации. Мы не будем объединять их в одну функцию, поскольку в главном файле приложения можем захотеть выбрать момент, когда Passport будет включен в цепочку промежуточного ПО (помните, что, когда вы добавляете промежуточное ПО, порядок очень важен). Так что вместо того, чтобы наш модуль экспортировал функцию, которая делает любую из этих вещей, мы собираемся сделать так, чтобы он возвращал функцию, которая возвращает объект, в котором есть нужные нам методы. Почему бы не вернуть просто объект, чтобы начать с него? Потому что нам нужно приготовить некоторые конфигурационные значения. Кроме того, поскольку мы должны связать промежуточное ПО Passport с нашим приложением, функция будет простым способом передать объект в приложение Express:

```
module.exports = (app, options) => {
  // Если перенаправления для успеха и неуспеха не определены,
  // установите разумные значения по умолчанию.
  if(!options.successRedirect) options.successRedirect = '/account'
  if(!options.failureRedirect) options.failureRedirect = '/login'
  return {
    init: function() { /* TODO */ },
    registerRoutes: function() { /* TODO */ },
  }
}
```

Прежде чем вникать в детали методов `init` и `registerRoutes`, посмотрим, как мы будем использовать этот модуль (надеюсь, это немного прояснит дело с возвратом функции, которая возвращает объект):

```
const createAuth = require('./lib/auth')

// …другие настройки приложения.

const auth = createAuth(app, {
  // baseUrl опционален; по умолчанию будет
  // использоваться localhost, если вы пропустите его;
  // имеет смысл установить его, если вы не
  // работаете на своей локальной машине. Например,
  // если вы используете вспомогательный сервер,
  // можете установить в переменной среды BASE_URL
  // https://staging.meadowlark.com.
  baseUrl: process.env.BASE_URL,
  providers: credentials.authProviders,
  successRedirect: '/account',
```

```
  failureRedirect: '/unauthorized',
})

// В auth.init() подключается промежуточное ПО Passport.
auth.init()

// Теперь мы можем указать наши маршруты аутентификации.
auth.registerRoutes()
```

Обратите внимание, что в дополнение к указанию путей перенаправления при успехе и неуспехе мы также указываем свойство, именуемое `providers`, которое реализовали в файле `credentials.js` (см. главу 13). Нам понадобится добавить свойство `authProviders` в `.credentials.development.json`:

```
"authProviders": {
  "facebook": {
    "appId": "your_app_id",
    "appSecret": "your_app_secret"
  }
}
```

Еще одна причина объединения кода аутентификации в модуле, подобном этому, в том, что мы можем использовать его для других проектов; на самом деле уже есть некоторые проекты аутентификации, которые, по сути, делают то, что мы делаем здесь. Тем не менее очень важно понимать детали того, что происходит, так что, если в конечном итоге вы будете использовать модуль, написанный кем-то другим, это поможет вам понять все, что происходит в вашем процессе аутентификации.

Теперь посмотрим на метод `init` (на этом месте в `auth.js` было `"TODO"`):

```
init: function() {
  var config = options.providers

  // Конфигурирование стратегии Facebook.
  passport.use(new FacebookStrategy({
    clientID: config.facebook.appId,
    clientSecret: config.facebook.appSecret,
    callbackURL: (options.baseUrl || '') + '/auth/facebook/callback',
  }, function(accessToken, refreshToken, profile, done){
    const authId = 'facebook:' + profile.id
    db.getUserByAuthId(authId)
      .then(user => {
        if(user) return done(null, user)
        db.addUser({
          authId: authId,
          name: profile.displayName,
          created: new Date(),
```

```
          role: 'customer',
        })
        .then(user => done(null, user))
        .catch(err => done(err, null))
      })
      .catch(err => {
        if(err) return done(err, null);
      })
}))

app.use(passport.initialize())
  app.use(passport.sessionp))
},
```

Это довольно плотный кусок кода, но в действительности бóльшая его часть — просто шаблонный код Passport. Важный фрагмент находится внутри функции, которая будет передана экземпляру `FacebookStrategy`. Когда эта функция вызывается (после успешного прохождения пользователем аутентификации), параметр `profile` содержит информацию о пользователе Facebook. Самое главное: он включает идентификатор Facebook — это то, что мы будем использовать, чтобы связать учетную запись Facebook с нашим собственным объектом `user`. Обратите внимание, что мы размещаем свойство `authID` в пространстве имен посредством добавления префикса `facebook:`. Это предотвратит возможность совпадения идентификатора Facebook с идентификатором Twitter или Google, какой бы мизерной ни была вероятность подобного совпадения (также это позволяет исследовать модели пользователя, чтобы увидеть, какой метод аутентификации он использует; иногда это полезно). Если в базе данных уже содержится запись для этого идентификатора в пространстве имен, мы просто возвращаем ее (когда вызывается `serializeUser`, который помещает идентификатор пользователя в сессию). Если запись пользователя не вернулась, мы создаем новый объект `user` и сохраняем его в базе данных.

Последнее, что мы должны сделать, — создать метод `registerRoutes` (не волнуйтесь, это намного короче):

```
registerRoutes: () => {
  app.get('/auth/facebook', function(req, res, next){
    if(req.query.redirect) req.session.authRedirect = req.query.redirect
    passport.authenticate('facebook')(req, res, next)
  })
  app.get('/auth/facebook/callback', passport.authenticate('facebook',
    { failureRedirect: options.failureRedirect }),
    (req, res) => {
      // Мы сюда попадаем только при успешной аутентификации.
      const redirect = req.session.authRedirect
      if(redirect) delete req.session.authRedirect
      res.redirect(303, redirect || options.successRedirect)
    }
  )
},
```

Теперь у нас есть путь `/auth/facebook`; посещение его автоматически перенаправит посетителя на страницу аутентификации Facebook (это сделано посредством `passport.authenticate('facebook')`, см. второй шаг на рис. 18.1). Обратите внимание, что мы проверяем, есть ли параметр строки запросов `redirect`; если он есть, сохраняем его в сессии. Так мы можем автоматически перенаправлять к месту назначения после завершения аутентификации. После того как пользователь авторизован через Facebook, браузер будет перенаправлен обратно на ваш сайт, в частности к пути `/auth/facebook/callback` (с опциональной строкой запроса `redirect`, где пользователь был изначально).

В строке запроса также есть токены аутентификации, которые проверяются Passport. Если проверка прошла неуспешно, Passport перенаправит браузер на `options.failureRedirect`. Если проверка прошла успешно, Passport вызовет `next`, то есть то место, куда возвращается ваше приложение. Обратите внимание, как промежуточное ПО подключено к обработчику `/auth/facebook/callback`: `passport.authenticate` вызывается первым. Если он вызывает `next`, управление переходит к вашей функции, которая затем перенаправляет либо в исходное место, либо на `options.successRedirect`, если не был указан параметр строки запроса `redirect`.

Опускание параметра строки запроса redirect способно упростить маршруты аутентификации, которые могут быть витиеватыми, если у вас лишь один URL, требующий аутентификации. Наличие такого функционала удобно и улучшает взаимодействие с пользователем. Вне всякого сомнения, вы с этим уже сталкивались: находили нужную страницу и вам нужно было войти в систему. Вы делали это, после чего перенаправлялись на страницу по умолчанию и снова переходили на свою страницу. Не самый лучший пользовательский опыт.

Магия Passport во время этого процесса — сохранение пользователя (в нашем случае просто идентификатора пользователя в базе данных) в сессию. Это хорошо, поскольку браузер выполняет *перенаправление*, что является другим запросом HTTP: не получая этой информации в сессии, мы не имели бы возможности узнать, что пользователь был аутентифицирован! Как только аутентификация прошла успешно, будет установлен `req.session.passport.user`, и, таким образом, будущие запросы будут знать, что этот пользователь прошел аутентификацию.

Посмотрим на обработчик `/account` и увидим, как он проверяет, что пользователь прошел аутентификацию (этот обработчик маршрута будет в главном файле приложения или в отдельном модуле маршрутизации, не в `/lib/auth.js`):

```
app.get('/account', (req, res) => {
  if(!req.user)
    return res.redirect(303, '/unauthorized')
  res.render('account', { username: req.user.name })
})
```

```
// Нам также нужна страница 'Не авторизирован'.
app.get('/unauthorized', (req, res) => {
  res.status(403).render('unauthorized')
})
// И способ выхода из аккаунта.
app.get('/logout', (req, res) => {
  req.logout()
  res.redirect('/')
})
```

Сейчас только прошедшие аутентификацию пользователи увидят страницу учетной записи; все остальные будут перенаправлены на страницу Не авторизован.

Авторизация на основе ролей

Пока мы технически не делаем авторизацию, лишь различаем авторизованных и неавторизованных пользователей. Тем не менее допустим, что мы хотим, чтобы только покупатели видели свои учетные записи (у сотрудников будет совершенно другое представление для просмотра информации учетной записи пользователя).

Помните, что в одном маршруте у вас может быть несколько функций, вызывающихся в определенном порядке. Создадим функцию `customersOnly`, которая пропустит только покупателей:

```
const customerOnly = (req, res, next) => {
  if(req.user && req.user.role === 'customer') return next()
  // Мы хотим, чтобы при посещении страниц только
  // для покупателей требовался логин.
  res.redirect(303, '/unauthorized')
}
```

Создадим также функцию `employeeOnly`, которая будет работать немного по-другому. Скажем, у нас есть путь `/sales`, который мы хотим сделать доступным только для сотрудников. Кроме того, мы не хотим, чтобы все прочие знали о его существовании, даже если наткнутся на него случайно. Если потенциальный злоумышленник пойдет по пути `/sales`, он увидит страницу Не авторизован, а это уже небольшая информация, которая сделает атаку проще (лишь из-за того, что известно, что такая страница существует). Таким образом, для обеспечения небольшой дополнительной безопасности мы хотим, чтобы не сотрудники, посещая страницу `/sales`, видели обычную страницу 404, которая не позволит потенциальным злоумышленникам работать с ней:

```
const employeeOnly = (req, res, next) => {
  if(req.user && req.user.role === 'employee') return next()
  // Мы хотим, чтобы неуспех авторизации посещения страниц только
  // для сотрудников был скрытым, чтобы потенциальные хакеры
  // не смогли даже узнать, что такая страница существует.
  next('route')
}
```

Вызов `next('route')` не просто выполнит следующий обработчик в маршруте — он пропустит этот маршрут в целом. Если предположить, что нет дальнейшего маршрута, который будет обрабатывать `/account`, это в конечном итоге приведет к обработчику 404, давая нам желаемый результат.

Вот как просто использовать эти функции:

```
// Маршруты покупателя

app.get('/account', customerOnly, (req, res) => {
  res.render('account', { username: req.user.name })
})
app.get('/account/order-history', customerOnly, (req, res) => {
  res.render('account/order-history')
})
app.get('/account/email-prefs', customerOnly, (req, res) => {
res.render('account/email-prefs')
})

// Маршруты сотрудника
app.get('/sales', employeeOnly, (req, res) => {
  res.render('sales')
})
```

Авторизация на основе ролей может быть настолько простой или сложной, как вы того захотите. Например, вы пожелаете разрешить несколько ролей. Можете использовать следующие функцию и маршрут:

```
const allow = roles => (req, res, next) => {
  if(req.user && roles.split(',').indexOf(req.user.role)!==-1) return next()
  res.redirect(303, '/unauthorized')
}
```

Надеюсь, этот пример продемонстрирует, насколько креативными вы можете быть с авторизацией на основе ролей. Вы можете даже авторизовать на основе других свойств, таких как время, которое пользователь затратил на работу с вашим сервисом, или число отпускных туров, которые пользователь забронировал с использованием вашего сервиса.

Добавление дополнительных поставщиков аутентификации

Теперь, когда основная конструкция готова и работает, добавление дополнительных поставщиков аутентификации — дело довольно простое. Скажем, мы хотим проводить аутентификацию посредством Google. Прежде чем начнете добавлять код, нужно создать проект вашей учетной записи Google.

Зайдите в вашу консоль разработчиков Google (http://bit.ly/2KcY1X0) и выберите проект на панели навигации (если у вас еще нет проекта, нажмите New Project (Создать проект) и следуйте инструкциям). После того как проект выбран, выберите Enable

APIs and Services (Включить API и сервисы) и включите Cloud Identity API. Нажмите на Credentials (Данные доступа), а затем Create Credentials (Создать данные доступа), выберите OAuth client ID (Идентификатор клиента OAuth) и далее — Web application (Веб-приложение). Введите соответствующие URL для вашего приложения: для тестирования можно использовать http://localhost:3000 для авторизованных источников и http://localhost:3000/auth/google/callback для авторизованных URI перенаправления.

После того как вы все сделаете на стороне Google, запустите `npm install passport-google-oauth20` и добавьте следующий код в `lib/auth.js`:

```
// Конфигурация стратегии Google
passport.use(new GoogleStrategy({
  clientID: config.google.clientID,
  clientSecret: config.google.clientSecret,
  callbackURL: (options.baseUrl || '') + '/auth/google/callback',
}, (token, tokenSecret, profile, done) => {
  const authId = 'google:' + profile.id
  db.getUserByAuthId(authId)
    .then(user => {
      if(user) return done(null, user)
      db.addUser({
        authId: authId,
        name: profile.displayName,
        created: new Date(),
        role: 'customer',
      })
      .then(user => done(null, user))
      .catch(err => done(err, null))
    })
    .catch(err => {
      console.log('упс, произошла ошибка: ', err.message)
      if(err) return done(err, null);
    })
}))
```

А этот код — в метод `registerRoutes`:

```
// Регистрируем маршруты Google
app.get('/auth/google', (req, res, next) => {
  if(req.query.redirect) req.session.authRedirect = req.query.redirect
  passport.authenticate('google', { scope: ['profile'] })(req, res, next)
})
app.get('/auth/google/callback', passport.authenticate('google',
  { failureRedirect: options.failureRedirect }),
  function(req, res){(req, res) => {
    // Мы сюда попадаем только при успешной
    // аутентификации
    const redirect = req.session.authRedirect
    if(redirect) delete req.session.authRedirect
    res.redirect(303, req.query.redirect || options.successRedirect)
  }
)
```

Резюме

Поздравляю, вы осилили самую сложную главу! Жаль, что такие важные вещи, как аутентификация и авторизация, настолько сложны, но в реальном мире, изобилующем угрозами безопасности, эта сложность неизбежна. К счастью, такие проекты, как Passport (и отличные схемы аутентификации, базирующиеся на нем), несколько облегчают нашу работу. Тем не менее я призываю вас не стараться как можно быстрее разобраться с этой областью в своем приложении: усердие в сфере безопасности сделает из вас порядочного гражданина Интернета. Пользователи, возможно, никогда вас не поблагодарят, но горе тем владельцам приложений, которые позволяют скомпрометировать пользовательские данные из-за низкого уровня безопасности!

19 Интеграция со сторонними API

В последнее время мало успешных сайтов, которые были бы полностью автономны. Для того чтобы заинтересовать существующих и найти новых пользователей, интеграция с социальными сетями обязательна. Для указания местонахождения магазинов и мест оказания прочих услуг огромное значение имеет использование сервисов геолокации и отображения на карте. И это еще не все: многие организации осознают, что предоставление API помогает расширить перечень услуг и сделать их более полезными.

В этой главе мы обсудим две наиболее актуальные потребности интеграции: социальные медиа и геолокацию.

Социальные медиа

Социальные медиа — отличный способ продвинуть продукт или сервис: возможность делиться вашим контентом в социальных медиа очень важна для пользователей. В момент, когда я пишу эту главу, доминирующие социальные сети — Facebook, Twitter, Instagram и YouTube. У сайтов вроде Pinterest и Flickr есть свое место, но их аудитория, как правило, небольшая и более специфическая (например, если ваш сайт связан с созданием изделий способом «сделай сам», то вы наверняка захотите поддерживать Pinterest). Хотите верьте, хотите нет, но я предсказываю, что MySpace еще вернется. Они замечательно передизайнили сайт, и, что стоит отметить, он сделан на Node.

Плагины социальных медиа и производительность сайта

Большая часть интеграции социальных медиа — дело клиентской части. Вы ссылаетесь на соответствующие файлы JavaScript на вашей странице, и они включают как входящий контент (например, три верхних поста с вашей страницы на Facebook), так и исходящий (например, можно сделать твит о странице, на которой вы на-

ходитесь). Зачастую это простой способ интеграции социальных медиа, но у него тоже есть нюансы: я видел, как страницы загружаются вдвое, а то и втрое дольше из-за дополнительных HTTP-запросов. Если производительность страницы для вас важна (а она должна быть важной, особенно если вы ориентируетесь на пользователей мобильных устройств), нужно тщательно продумать интеграцию социальных медиа.

При этом код, позволяющий включить кнопку Facebook Нравится или кнопку Tweet, использует cookie браузера, чтобы отправить сообщение от имени пользователя. Перемещение этого функционала на серверную сторону будет непростым, а в некоторых случаях и вовсе невозможным. Так что, если вам нужен именно этот функционал, наилучшим решением может быть подключение соответствующей сторонней библиотеки, даже несмотря на то, что это способно сказаться на производительности страницы.

Поиск твитов

Предположим, мы хотим указать топ-10 твитов, содержащих хештеги #Oregon и #travel. Для этого можно использовать компонент клиентской части, но он будет задействовать дополнительные запросы HTTP. Кроме того, если мы сделаем это на серверной стороне, у нас появится возможность кэширования твитов для повышения производительности. В этом случае также можно отправлять в черный список твиты недоброжелателей, тогда как на стороне клиента сделать это гораздо сложнее.

Twitter, как и Facebook, позволяет создавать *приложения*. Хотя это немного неподходящая формулировка: приложение Twitter ничего не делает (в буквальном смысле). Это, скорее, набор данных доступа, которые вы можете использовать для создания реального приложения на вашем сайте. Простейший и наиболее универсальный способ получения доступа на Twitter API — создать приложение и использовать его для получения токена доступа.

Создайте приложение Twitter, зайдя на http://dev.twitter.com. Удостоверьтесь, что вы осуществили вход, и щелкните на своем имени пользователя на панели навигации, а затем — на Apps (Приложения). Нажмите Create an app (Создать новое приложение) и следуйте инструкциям. Когда появится приложение, вы увидите, что у вас теперь есть *пользовательский ключ API* и *секретный ключ API*. Секретный ключ API, на что указывает название, нужно держать в тайне: никогда не включайте его в ответы, отправляемые на сторону клиента. Если кто-то извне получит доступ к этому ключу, он сможет создавать запросы от имени вашего приложения. Это может привести к печальным последствиям, если их использование будет злонамеренным.

Теперь у нас есть пользовательский ключ API и секретный ключ, и мы можем общаться с Twitter REST API.

Чтобы сохранять код в аккуратном виде, поместим наш код Twitter в модуль, названный `lib/twitter.js`:

```
const https = require('https')

module.exports = twitterOptions => {

  return {
    search: async (query, count) => {
      // то, что нужно сделать
    }
  }
}
```

Этот шаблон наверняка кажется вам знакомым. Наш модуль экспортирует функцию, в которую вызывающая сторона передает объект конфигурации. Возвращается объект, содержащий методы. Таким образом, мы можем добавить функционал в наш модуль. Сейчас мы предоставляем только метод `search`. Вот как будем использовать библиотеку:

```
const twitter = require('./lib/twitter')({
  consumerApiKey: credentials.twitter.consumerApiKey,
  apiSecretKey: credentials.twitter.apiSecretKey,
})

const tweets = await twitter.search('#Oregon #travel', 10)
// Твиты будут в result.statuses
```

(Не забывайте внести свойство `twitter` с `consumerApiKey` и `apiSecretKey` в файл `.credentials.development.json`.)

Прежде чем реализовать метод `search`, мы должны предоставить определенную функциональность для аутентификации самих себя в Twitter. Процесс прост: используем HTTPS для запроса токена доступа, базирующийся на наших пользовательском ключе и пользовательском секрете. Мы должны сделать это только раз: токены у Twitter даются бессрочно (впрочем, можете аннулировать их вручную). Поскольку мы не хотим запрашивать токен доступа каждый раз, мы его закэшируем для дальнейшего использования.

Способ, посредством которого мы построили модуль, позволяет создать приватную функциональность, недоступную вызывающей стороне. В частности, единственная функция, которая доступна вызывающей стороне, — это `module.exports`. Поскольку мы возвращаем функцию, вызывающей стороне будет доступна только она. Вызов функции в результате дает объект, и только свойства этого объекта доступны вызывающей стороне. Итак, мы собираемся создать переменную `accessToken`, которую будем использовать для кэширования нашего токена доступа, и функцию `getAccessToken`, которая этот токен получит. При первом запуске она

отправляет запрос Twitter API для получения токена доступа. Последующие вызовы просто возвращают значение `accessToken`:

```
const https = require('https')

module.exports = function(twitterOptions){

    // Эта переменная будет невидимой вне этого модуля
    let accessToken = null

    // Эта функция будет невидимой за пределами этого модуля
    const getAccessToken = async () => {
        if(accessToken) return accessToken
        // То, что нужно сделать: получение токена доступа
    }

    return {
        search: async (query, count) => {
            // То, что нужно сделать
        }
    }
}
```

Мы помечаем `getAccessToken` как асинхронный, так как нам может понадобиться HTTP-запрос к Twitter API (если в кэше нет токена). Теперь, когда мы установили базовую структуру, реализуем `getAccessToken`:

```
const getAccessToken = async () => {
  if(accessToken) return accessToken

  const bearerToken = Buffer(
    encodeURIComponent(twitterOptions.consumerApiKey) + ':' +
    encodeURIComponent(twitterOptions.apiSecretKey)
  ).toString('base64')

  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'POST',
    path: '/oauth2/token?grant_type=client_credentials',
    headers: {
      'Authorization': 'Basic ' + bearerToken,
    },
  }

  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
```

```
      res.on('end', () => {
        const auth = JSON.parse(data)
        if(auth.token_type !== 'bearer')
          return reject(new Error('Авторизация в Twitter не прошла.'))
        accessToken = auth.access_token
        return resolve(accessToken)
      })
    }).end()
  )
}
```

Подробности построения этого вызова доступны на странице документации разработчика Twitter: http://bit.ly/2KcJ4EA. По сути, мы должны сделать токен предъявителя (bearer token), представляющий собой base64-кодированную комбинацию пользовательского ключа и пользовательского секрета. После того как мы создадим токен, можем вызвать `/oauth2/token` API с заголовком `Authorization`, содержащим токен предъявителя для запроса токена доступа. Заметьте, что мы должны использовать HTTPS: если вы попробуете сделать этот вызов посредством HTTP, то передадите свой секрет в незашифрованном виде, а API попросту отключит вас.

После того как получим полный ответ от API (мы ожидаем событие `end` потока ответа), можем разобрать JSON, убедиться, что тип токена `bearer`, и дальше идти своим путем. Мы кэшируем токен доступа, затем вызываем функцию обратного вызова.

Теперь, раз у нас есть механизм получения токена доступа, можно делать вызовы API. Реализуем наш метод поиска:

```
search: async (query, count) => {
  const accessToken = await getAccessToken()
  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/search/tweets.json?q=' +
      encodeURIComponent(query) +
      '&count=' + (count || 10),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  )
},
```

Отображение твитов

Сейчас у нас есть возможность искать твиты. Как мы будем отображать их на нашем сайте? По большому счету, это зависит от вас, но есть несколько вещей, которые мы должны рассмотреть. Twitter заинтересован в том, чтобы его данные использовались в соответствующем стиле. У него есть требования (http://bit.ly/32ET4N2), согласно которым при показе твита должно быть включено отображение определенных функциональных элементов.

В требованиях возможны некоторые маневры (например, если вы отображаете твиты на устройстве, не поддерживающем изображения, нет надобности добавлять аватар), но по большей части конечный итог должен быть максимально похож на то, как выглядит встроенный твит. Над этим придется хорошо поработать, но есть способ все упростить. Правда, он включает в себя ссылки на библиотеки виджетов Twitter, а это и есть тот самый HTTP-запрос, которого мы пытаемся избежать.

Если вам нужно отображать твиты, лучше использовать библиотеку виджетов Twitter, несмотря на то что это добавит дополнительный HTTP-запрос. Для более сложного использования API вам придется обратиться к REST API со стороны сервера, поэтому, вероятно, вы будете использовать REST API в связке со скриптами веб-интерфейса.

Продолжим рассматривать наш пример: мы хотим отображать десять последних твитов, в которых упоминаются хештеги #Oregon и #travel. Будем использовать REST API для поиска твитов и библиотеку виджетов Twitter для их отображения. Поскольку мы не хотим сталкиваться с ограничением в использовании (и к тому же замедлять работу сервера), будем кэшировать твиты и HTML для их отображения за 15 минут.

Начнем с модификации библиотеки Twitter, добавив метод `embed`, который получает HTML для отображения твита. Обратите внимание: для создания строки запроса из объекта мы используем библиотеку npm под названием `querystringify`, так что не забудьте установить ее командой `npm install querystringify` и импортировать (`const qs = require( 'querystringify ')`), а затем добавить следующую функцию в раздел экспорта `lib/twitter.js`:

```
embed: async (url, options = {}) => {
  options.url = url
  const accessToken = await getAccessToken()
  const requestOptions = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/statuses/oembed.json?' + qs.stringify(options),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) =>
```

```
    https.request(requestOptions, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  )
},
```

Теперь мы готовы искать и кэшировать твиты. В главном файле нашего приложения создадим функцию `getTopTweets`:

```
const twitterClient = createTwitterClient(credentials.twitter)

const getTopTweets = ((twitterClient, search) => {
  const topTweets = {
    count: 10,
    lastRefreshed: 0,
    refreshInterval: 15 * 60 * 1000,
    mtweets: [],
  }
  return async () => {
    if(Date.now() > topTweets.lastRefreshed + topTweets.refreshInterval) {
      const tweets =
        await twitterClient.search('#Oregon #travel', topTweets.count)
        const formattedTweets = await Promise.all(
        tweets.statuses.map(async ({ id_str, user }) => {
          const url = `https://twitter.com/${user.id_str}/statuses/${id_str}`
          const embeddedTweet =
            await twitterClient.embed(url, { omit_script: 1 })
          return embeddedTweet.html
        })
      )
      topTweets.lastRefreshed = Date.now()
      topTweets.tweets = formattedTweets
    }
    return topTweets.tweets
  }
})(twitterClient, '#Oregon #travel')
```

По сути, функция `getTopTweets` предназначена не только для поиска твитов по указанному хештегу, но и для кэширования этих твитов на какой-то разумный период времени. Заметьте: мы создали немедленно вызываемую функцию (immediately invoked function expression, IIFE). Это связано с тем, что мы хотим, чтобы `topTweets` безопасно кэшировалась внутри замыкания, дабы не повредить его. Асинхронная функция, возвращаемая из функции IIFE, при необходимости обновляет кэш и затем возвращает его содержимое.

Наконец, создадим представление `views/social.handlebars` для присутствующих социальных медиа (которая на данный момент содержит только выбранные твиты):

```
<h2>Oregon Travel в социальных сетях</h2>

<script id="twitter-wjs" type="text/javascript"
  async defer src="//platform.twitter.com/widgets.js"></script>

{{{tweets}}}
```

И маршрут для их обработки:

```
app.get('/social', async (req, res) => {
    res.render('social', { tweets: await getTopTweets() })
})
```

Обратите внимание, что мы ссылаемся на внешний скрипт `widgets.js` в Twitter. Он будет форматировать вложенные твиты на данной странице и предоставлять им функциональность. API oembed по умолчанию будет включать ссылку на этот скрипт в HTML, но поскольку мы отображаем десять твитов, то ссылаться на этот скрипт будем в девять раз больше, чем нужно! Помня об этом, при вызове API oembed мы передаем опцию `{ omit_script: 1 }`. Затем нам нужно где-то предоставить доступ к тому, что мы сделали в представлении. Попробуйте удалить этот скрипт из представления. Вы по-прежнему будете видеть твиты, но у них не будет форматирования и функциональности.

Теперь у нас прекрасно налажена связь с социальными медиа! Перейдем к другому важному приложению — к отображению карт!

Геокодирование

Геокодирование — процесс получения адреса или названия места (Блетчли-парк, Шервуд Драйв, Блетчли, Милтон Кейнс MK3 6EB, Великобритания) и преобразования его в географические координаты (широта — 51,997 659 7, долгота — 0,740 686 3). Если предполагается, что ваше приложение будет делать какие-то географические расчеты — расстояния или направления — или отображать карту, вам понадобятся географические координаты.

Возможно, вы привыкли видеть географические координаты, выраженные в градусах, минутах и секундах (ГМС). API геокодера и сервисы отображения карт используют числа с плавающей точкой и для широты, и для долготы. Если вам нужно отображать ГМС координаты, смотрите https://ru.wikipedia.org/wiki/Преобразование_геодезических_систем_координат.

Геокодирование с Google

И Google, и Bing предлагают превосходные REST-сервисы для геокодирования. В нашем примере используем Google, но у Bing сервис очень похож.

Без добавления платежного аккаунта к вашей учетной записи Google вы будете ограничены одним запросом геокодирования в день, что значительно замедлит тестирование! В этой книге везде, где только возможно, я старался не рекомендовать сервисы, которыми нельзя воспользоваться бесплатно хотя бы для разработки. Я попробовал некоторые бесплатные сервисы геокодирования и понял, что с точки зрения использования они значительно уступают платным, поэтому продолжаю рекомендовать Google geocoding. Однако на момент написания этих строк Google предоставляет месячный кредит 200 долларов для разработки, и вам потребуется сделать 40 000 запросов, чтобы его истратить! Если хотите попробовать, перейдите в консоль Google (http://bit.ly/2KcY1X0), выберите в главном меню пункт Billing (Оплата) и введите вашу платежную информацию.

После настройки оплаты вам понадобится ключ API для Google API geocoding. Перейдите в консоль, выберите ваш проект на панели навигации и щелкните на списке API. Если в вашем списке включенных API нет geocoding API, найдите его в списке дополнительных API и добавьте. Большинство Google API используют одни и те же данные доступа, так что щелкните на меню навигации слева вверху и вернитесь к вашей панели. Щелкните на Credentials (Данные доступа) и создайте новый ключ API, если у вас еще нет подходящего. Следует отметить, что ключи API могут иметь ограничения для защиты от злоумышленников, поэтому убедитесь, что ваш ключ API можно использовать из вашего приложения. Если вам нужен один ключ для разработки, вы можете ограничить его вашим IP-адресом (если вы его не знаете, можете просто спросить Google: «Какой у меня IP-адрес?»).

Как только вы получите ключ API, добавьте его в `.credentials.development.json`:

```
"google": {
  "apiKey": "<YOUR API KEY>"
}
```

Затем создайте модуль `lib/geocode.js`:

```
const https = require('https')
const { credentials } = require('../config')

module.exports = async query => {

  const options = {
    hostname: 'maps.googleapis.com',
    path: '/maps/api/geocode/json?address=' +
      encodeURIComponent(query) + '&key=' +
      credentials.google.apiKey,
  }
```

```
  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => {
        data = JSON.parse(data)
        if(!data.results.length)
          return reject(new Error(`Результаты по запросу "${query}" не найдены`))
        resolve(data.results[0].geometry.location)
      })
    }).end()
  )
}
```

Теперь у нас есть функция, которая свяжется с Google API для геокодирования адреса. Если адрес не найден (или причина неудачи в чем-то другом), будет возвращена ошибка. API может вернуть несколько адресов. Например, если вы ищете 10 Main street без указания города, штата или почтового кода, он вернет десятки результатов. Наша реализация просто выберет первый из этого списка. API возвращает много информации, но все, что нам нужно в настоящий момент, — это координаты. Вы можете легко изменить интерфейс, чтобы возвращать больше информации. Смотрите документацию Google (http://bit.ly/2O4EE3t) для получения более подробных сведений о данных, которые возвращает API.

Ограничения использования

На момент написания книги Google придерживался ограничения 5000 запросов за 100 секунд, чтобы избежать злоупотребления. Google API также требует, чтобы вы использовали на своем сайте Google Maps. То есть если вы используете сервис Google для геокодирования данных, то не можете отображать эту информацию на карте Bing, не нарушив при этом условия предоставления услуг. В целом это не очень обременительное ограничение, поскольку вы вряд ли будете использовать геокодирование, не показывая при этом месторасположение на карте. Но если вам карты Bing нравятся больше, чем Google, или наоборот, вы должны помнить об условиях предоставления услуг и использовать соответствующий API.

Геокодирование ваших данных

У нас есть хорошая база отпускных туров по Орегону, и мы можем отобразить карту, где булавками отметим места, посещение которых предполагается в рамках туров. Здесь в игру вступает геокодирование.

Кроме того, у нас есть сведения о турах в базе данных, и у каждого тура есть строка поиска местоположения, которая будет работать с геокодированием, но у нас еще нет координат.

Теперь вопрос в том, где и как мы будем производить геокодирование. В широком смысле у нас есть три варианта.

- Геокодирование в момент добавления тура в базу данных. Это, вероятно, прекрасный выбор, когда в системе есть интерфейс администратора, позволяющий поставщикам услуг динамически добавлять туры в базу данных. Поскольку до такой функциональности у нас дело не дойдет, мы откажемся от данного варианта.
- Геокодирование при извлечении тура из базы данных. При этом подходе при каждом получении туров из базы данных будет производиться проверка: если у какого-то из них не окажется координат, мы определим их геокодированием. Звучит заманчиво, и этот подход, возможно, самый простой из трех, но у него есть существенные недостатки. Во-первых, производительность: если вы добавили 100 новых туров в базу данных, первому человеку, который просмотрит список туров, придется ждать, пока выполнятся все запросы геокодирования и результаты окажутся в базе данных. Кроме того, легко можно представить ситуацию, когда модуль нагрузочного тестирования вносит тысячу туров в базу данных и затем выполняет тысячу запросов. Поскольку все они запускаются параллельно, каждый из тысячи этих запросов вызовет тысячу запросов геокодирования, поскольку данные еще не были добавлены в базу, что даст миллион запросов геокодирования и счет на 5000 долларов от Google! Так что вычеркиваем этот вариант из списка.
- Скрипт для поиска туров, у которых нет данных о координатах, и их геокодирования. Этот подход — лучшее решение для нашего текущего приложения. В рамках разработки мы один раз заполняем базу данных отпускных туров, и у нас еще нет интерфейса администратора. Кроме того, если позже мы решим его добавить, то проблем не будет: фактически мы можем просто запустить этот процесс после внесения нового тура и он будет работать.

Сначала нужно добавить способ обновления существующих отпускных туров в файл `db.js` (мы также добавим метод для закрытия подключения к базе данных, что пригодится в скриптах):

```
module.exports = {
  //...
  updateVacationBySku: async (sku, data) => Vacation.updateOne({ sku }, data),
  close: () => mongoose.connection.close(),
},
```

Затем мы можем написать скрипт `db-geocode.js`:

```
const db = require('./db')
const geocode = require('./lib/geocode')
```

```
const geocodeVacations = async () => {
  const vacations = await db.getVacations()
  const vacationsWithoutCoordinates = vacations.filter(({ location }) =>
    !location.coordinates || typeof location.coordinates.lat !== 'number')
  console.log(`геокодирование ${vacationsWithoutCoordinates.length} ` +
    `из ${vacations.length} туров:`)
  return Promise.all(vacationsWithoutCoordinates.map(async ({ sku,
                     location }) => {
    const { search } = location
    if(typeof search !== 'string' || !/\w/.test(search))
      return console.log(` SKU ${sku} НЕ ВЫПОЛНЕН: отсутствует location.search`)
    try {
      const coordinates = await geocode(search)
      await db.updateVacationBySku(sku, { location: { search, coordinates } })
    console.log(` SKU ${sku} SUCCEEDED: ${coordinates.lat}, ${coordinates.lng}`)
    } catch(err) {
      return console.log(` SKU {sku} FAILED: ${err.message}`)
    }
  }))
}

geocodeVacations()
  .then(() => {
    console.log('ВЫПОЛНЕНО')
    db.close()
  })
  .catch(err => {
    console.error('ОШИБКА: ' + err.message)
    db.close()
  })
```

После запуска этого скрипта (`node db-geocode.js`) вы должны увидеть, что все ваши туры были успешно геокодированы! Теперь, когда мы располагаем этой информацией, научимся показывать ее на карте.

Отображение карты

Притом что отображение туров на карте относится к работе клиентской части, было бы обидно зайти так далеко и не увидеть плодов своей работы. Так что немного отойдем от серверной части нашей книги и посмотрим, как отобразить геокодированных дилеров на карте.

Мы уже создали ключ Google API для выполнения геокодирования, но нам все еще нужно включить API для карт. Перейдите в консоль Google, щелкните на списке API, найдите Maps JavaScript API и включите его, если еще этого не сделали.

Теперь можно создать представление для показа карт с ключевыми точками туров `views/vacations-map.handlebars`. Начнем с отображения карты, а затем поработаем над добавлением точек:

```
<div id="map" style="width: 100%; height: 60vh;"></div>
<script>
  let map = undefined
  async function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      // Приблизительный географический центр Орегона.
      center: { lat: 44.0978126, lng: -120.0963654 },
      // Такой коэффициент масштабирования
      // позволяет покрыть большую часть штата.
      zoom: 7,
    })
  }
</script>
<script src="https://maps.googleapis.com/maps/api/js?key={{googleApiKey}}
  &callback=initMap" async defer></script>
```

Пришло время поместить на карту булавки, соответствующие точкам-локациям. В главе 15 мы создали конечную точку `/api/vacations`, которая теперь содержит данные геокодирования. Мы воспользуемся ею для получения локаций и приколем булавки к карте. Преобразуйте функцию `initMap` в `views/vacations-map.handlebars.js`:

```
async function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    // Приблизительный географический центр Орегона.
    center: { lat: 44.0978126, lng: -120.0963654 },
    // Такой коэффициент масштабирования
    // позволяет покрыть большую часть штата.
    zoom: 7,
  })
  const vacations = await fetch('/api/vacations').then(res => res.json())
  vacations.forEach(({ name, location }) => {
    const marker = new google.maps.Marker({
      position: location.coordinates,
      map,
      title: name,
    })
  })
}
```

Теперь у нас есть карта, на которой отмечены ключевые точки наших туров! Эту страницу можно несколько улучшить: например, можно привязать булавку к странице со сведениями о туре, так чтобы при нажатии на нее осуществлялся переход на страницу тура. Мы можем также реализовать пользовательские булавки или подсказки: у Google Maps API много функций, о которых можно узнать из официальной документации Google (https://developers.google.com/maps/documentation/javascript/tutorial).

Метеоданные

Помните виджет «Текущая погода» из главы 7? Подключим его с актуальными данными! Мы будем использовать американский сервис National Weather Service (NWS) API для получения информации о прогнозе погоды. Как и в случае с Twitter и использования геокодирования мы будем кэшировать прогнозы, чтобы предотвратить передачу каждого обращения к нашему сайту в NWS (из-за чего можно попасть в черный список, если наш сайт станет популярным). Создайте сайт `lib/weather.js`:

```
const https = require('https')
const { URL } = require('url')

const _fetch = url => new Promise((resolve, reject) => {
  const { hostname, pathname, search } = new URL(url)
  const options = {
    hostname,
    path: pathname + search,
    headers: {
      'User-Agent': 'Meadowlark Travel'
    },
  }
  https.get(options, res => {
    let data = ''
    res.on('data', chunk => data += chunk)
    res.on('end', () => resolve(JSON.parse(data)))
  }).end()
})

module.exports = locations => {

  const cache = {
    refreshFrequency: 15 * 60 * 1000,
    lastRefreshed: 0,
    refreshing: false,
    forecasts: locations.map(location => ({ location })),
  }

  const updateForecast = async forecast => {
    if(!forecast.url) {
      const { lat, lng } = forecast.location.coordinates
      const path = `/points/${lat.toFixed(4)},${lng.toFixed(4)}`
      const points = await _fetch('https://api.weather.gov' + path)
      forecast.url = points.properties.forecast
    }
    const { properties: { periods } } = await _fetch(forecast.url)
    const currentPeriod = periods[0]
    Object.assign(forecast, {
```

```
      iconUrl: currentPeriod.icon,
      weather: currentPeriod.shortForecast,
      temp: currentPeriod.temperature + ' ' + currentPeriod.temperatureUnit,
    })
    return forecast
  }

  const getForecasts = async () => {
    if(Date.now() > cache.lastRefreshed + cache.refreshFrequency) {
      console.log('updating cache')
      cache.refreshing = true
      cache.forecasts = await Promise.all(cache.forecasts.map(updateForecast))
      cache.refreshing = false
    }
    return cache.forecasts
  }

  return getForecasts

}
```

Вы наверняка заметили, что нам надоело напрямую использовать встроенную в Node библиотеку https. Вместо этого мы создали сервисную функцию `_fetch`, чтобы функциональность для погоды стала более читаемой. Вы также могли заметить, что мы устанавливаем для заголовка `User-Agent` значение `Meadowlark Travel`. Это особенность API погоды NWS: ему требуется строка для `User-Agent`. Они обещают, что заменят это ключом API, но на данный момент нам просто нужно предоставить это значение.

Получение метеоданных из NWS API происходит в два этапа. Есть конечная точка API под названием `points`, которая принимает широту и долготу (с четырьмя знаками после запятой) и возвращает информацию об этом месте, включая соответствующий URL для получения прогноза погоды. Как только у нас будут такие URL для всех наборов координат, нам больше не придется их получать. Для обновления прогноза нужно будет лишь сделать запрос на этот URL.

Стоит отметить, что прогноз погоды возвращает гораздо больше данных, чем нам нужно; мы можем использовать это для реализации куда более сложного функционала. В частности, URL прогноза погоды возвращает массив периодов, первым элементом которого является текущий период (например, день или вечер). Далее идут периоды за эту и следующую неделю. Вы можете взглянуть на данные в массиве `periods`, чтобы увидеть, какие из них доступны.

Стоит отметить, что в кэше есть логическое свойство под названием `refreshing`. Оно необходимо, поскольку обновление кэша требует определенного времени и производится асинхронно. Если до завершения первого обновления придет много запросов, то работа по обновлению кэша остановится. При этом ничего не пострадает, но вы будете делать больше вызовов API, чем нужно. Эта логическая переменная — просто метка, которая сообщает любым дополнительным запросам: «Мы работаем над этим».

Все это мы разработали как замену для фиктивной функции, созданной в главе 7. Нам нужно лишь открыть `lib/middleware/weather.js` и заменить функцию `getWeatherData`:

```
const weatherData = require('../weather')

const getWeatherData = weatherData([
  {
    name: 'Портленд',
    coordinates: { lat: 45.5154586, lng: -122.6793461 },
  },
  {
    name: 'Бенд',
    coordinates: { lat: 44.0581728, lng: -121.3153096 },
  },
  {
    name: 'Манзанита',
    coordinates: { lat: 45.7184398, lng: -123.9351354 },
  },
])
```

Теперь в нашем виджете есть реальные метеоданные!

Резюме

Мы лишь поверхностно прошлись по тому, что связано с интеграцией API сторонних сервисов. Куда бы вы ни посмотрели, везде появляются API, предлагающие всевозможные данные (даже город Портленд сделал ряд публичных данных доступными через REST API). Охватить самую малую часть доступных для вас API — сложная задача, но в этой главе я изложил основы того, что вы должны знать об использовании этих API: `http.request`, `https.request` и парсинг JSON.

Теперь в нашем арсенале есть обширные знания. Мы рассмотрели много вопросов! Однако что же происходит, когда что-то пошло не так? В следующей главе мы обсудим методы отладки, которые помогут, когда что-то не получается.

20 Отладка

В англоязычной литературе принято пользоваться термином *debugging*, что можно перевести как «устранение ошибок», и это, пожалуй, не самое удачное понятие, поскольку оно ассоциируется именно с ошибками. Дело в том, что debugging — это деятельность, которой вы занимаетесь все время, независимо от того, реализуете ли новую функцию, изучаете, как что работает, или действительно исправляете ошибку. Пожалуй, в этой ситуации оптимальным был бы термин exploring — «исследование», но в англоязычной литературе традиционно придерживаются термина debugging, имеющего устоявшееся значение. Мы же будем пользоваться термином «отладка».

Отладка — навык, которым часто пренебрегают. Кажется, что он должен быть врожденным у большинства программистов и они с первых лет жизни должны знать, как это делается. Возможно, преподаватели информатики и авторы книг действительно рассматривают отладку как очевидное умение, поскольку ее обычно не замечают.

На самом деле отладка — это навык, которому можно научить. Он крайне важен, поскольку с его помощью программисты приходят к пониманию не только фреймворка, в котором работают, но и собственного кода, а также кода команды. В этой главе мы обсудим некоторые инструменты и техники, которые можно эффективно использовать для отладки приложений в Node и Express.

Первый принцип отладки

Как следует из названия, отладка часто связана с поиском и устранением дефектов. Прежде чем мы поговорим об инструментах, рассмотрим некоторые общие принципы отладки.

Сколько раз я говорил вам: отбросьте все невозможное, то, что останется, и будет ответом, каким бы невероятным он ни казался.

Сэр Артур Конан Дойл

Первый и важнейший принцип отладки — процесс *исключения всего лишнего*. Современные компьютерные системы невероятно сложны, и если бы вам понадобилось держать всю систему в голове и искать источник одной проблемы как иголку в стоге сена, то вы наверняка даже не знали бы, с чего начать. Всякий раз, когда вы сталкиваетесь с неочевидной проблемой, *самой первой мыслью* должна быть: «Что я могу *исключить* как источник проблемы?» Чем больше вы исключите, тем меньше останется мест, которые придется посмотреть.

Исключение может принимать различные формы. Вот некоторые распространенные примеры.

- Систематическое комментирование или отключение блоков кода.
- Написание кода, который покроется модульными тестами; модульные тесты сами по себе предоставляют фреймворк для исключения.
- Анализ сетевого трафика для определения того, на чьей стороне проблема: клиента или сервера.
- Тестирование другой части системы, у которой есть сходство с данной.
- Использование входных данных, которые работали прежде, и изменение их, до тех пор пока не обнаружится проблема.
- Использование контроля версий для перемещения вперед и назад во времени, до тех пор пока не исчезнет проблема. Вы можете изолировать ее как частное изменение (более подробную информацию смотрите на https://git-scm.com/docs/git-bisect).
- Создание заглушек для функциональности, чтобы исключить сложные подсистемы.

Однако исключение — это не панацея. Часто проблема возникает из-за сложного взаимодействия между несколькими компонентами. Она может уйти, при этом ее не удастся свести к какому-то одному компоненту. Но даже в этой ситуации, когда проблему сложно локализовать, исключение может помочь сузить ее.

Исключение — наиболее успешная практика, когда все действия осторожны и методичны. Очень легко что-то упустить, если вы бессмысленно устраняете компоненты, не учитывая, как они влияют на всю систему. Сыграйте в игру с самим собой: когда вы рассматриваете компонент для исключения, обдумайте, как его отсутствие повлияет на систему. Это подскажет, чего стоит ожидать и действительно ли удаление компонента принесет пользу.

Воспользуйтесь REPL и консолью

И Node, и ваш браузер предлагают вам *цикл чтения — вычисления — печати* (read — eval — print loop, REPL) — это, как правило, способ написать JavaScript в интерактивном режиме. Вы набираете некий JavaScript, нажимаете Enter и сразу же видите результат. Отличный способ, чтобы поиграть, и зачастую самый быстрый и интуитивно понятный метод найти ошибку в мелких кусочках кода.

Все, что вам нужно сделать в браузере, — это запустить консоль JavaScript, и у вас будет REPL. Все, что нужно сделать в Node, — это набрать `node` без каких-либо аргументов, и вы войдете в режим REPL. Вы можете запрашивать пакеты, создавать переменные и функции или делать что-то еще, что обычно делаете в своем коде (кроме создания пакетов: в REPL нет адекватного способа сделать это).

Логирование в консоль (использование `console.log`) — это тоже ваш друг. Пожалуй, это техника сырой отладки, но очень уж легкая (и понять просто, и реализовать). Вызов `console.log` в Node отобразит содержимое объекта в удобном для чтения формате, так что вы легко сможете найти проблему. Только имейте в виду, что некоторые объекты настолько велики, что их логирование в консоль даст слишком много выходных данных, а поиск в логе полезной информации займет много времени. Например, попробуйте `console.log(req)` в одном из своих обработчиков пути.

Использование встроенного отладчика Node

У Node есть встроенный отладчик, с помощью которого вы сможете последовательно пройти через приложение, как если бы прошлись совместно с интерпретатором JavaScript. Все, что вам нужно для начала отладки приложения, — это использовать аргумент `inspect`:

```
node inspect meadowlark.js
```

Сделав это, вы сразу же заметите несколько вещей. Так, в консоли вы увидите `URL` — это произошло потому, что отладчик Node работает посредством создания собственного веб-сервера, позволяющего контролировать выполнение отлаживаемого приложения. Прямо сейчас это вас вряд ли впечатлит, но полезность такого подхода станет очевидной, когда мы обсудим инспекторы.

Когда вы находитесь в отладчике консоли, можете набрать `help` для получения списка команд. Команды, которые вы будете использовать чаще всего, — это `n` (next), `s` (step in) и `o` (step out). Команда `n` позволит сделать шаг над текущей строкой: отладчик ее выполнит, но, если инструкция вызывает другие функции, они будут выполнены, прежде чем контроль к вам вернется. `s` в противовес ей шагнет в текущую строку: если эта строка вызывает другие функции, вы сможете пройти их пошагово. `o` позволяет выйти из текущей выполняемой функции (обратите внимание, что «вход» и «выход» касаются только *функций*, они не входят и не выходят из блоков `if` и `for` или других операторов управления).

У командной строки отладчика есть больше функциональных возможностей, но могу предположить, что вы не захотите часто ее использовать. Командная строка отлично подходит для многих вещей, но отладку нельзя назвать одной из них. Хорошо, когда она доступна в крайнем случае, например, если у вас только SSH-доступ к серверу или на вашем сервере не установлен графический интерфейс. В большинстве случаев вы будете пользоваться инспектором с графическим интерфейсом.

Инспекторы Node

Как уже отмечалось, вряд ли вы захотите использовать отладчик с командной строкой. Если только в крайнем случае. Node обеспечивает отладку через веб-сервис, и это дает альтернативные варианты.

Проще всего воспользоваться Chrome, который применяет такой же интерфейс отладчика, что и при отладке кода клиентской части. Если вы уже знакомы с отладчиком Chrome, то будете уверенно чувствовать себя. С ним легко начать работать. Запустите ваше приложение с опцией `--inspect` (это не то же самое, что упомянутый выше аргумент `inspect`):

```
node --inspect meadowlark.js
```

Теперь начинается самое интересное: введите `chrome://inspect` в адресной строке браузера. Вы увидите страницу DevTools, затем в разделе Devices (Устройства) нажмите на Open dedicated DevTools for Node (Открыть специальный набор инструментов DevTools для отладки Node). Откроется окно отладчика (рис. 20.1).

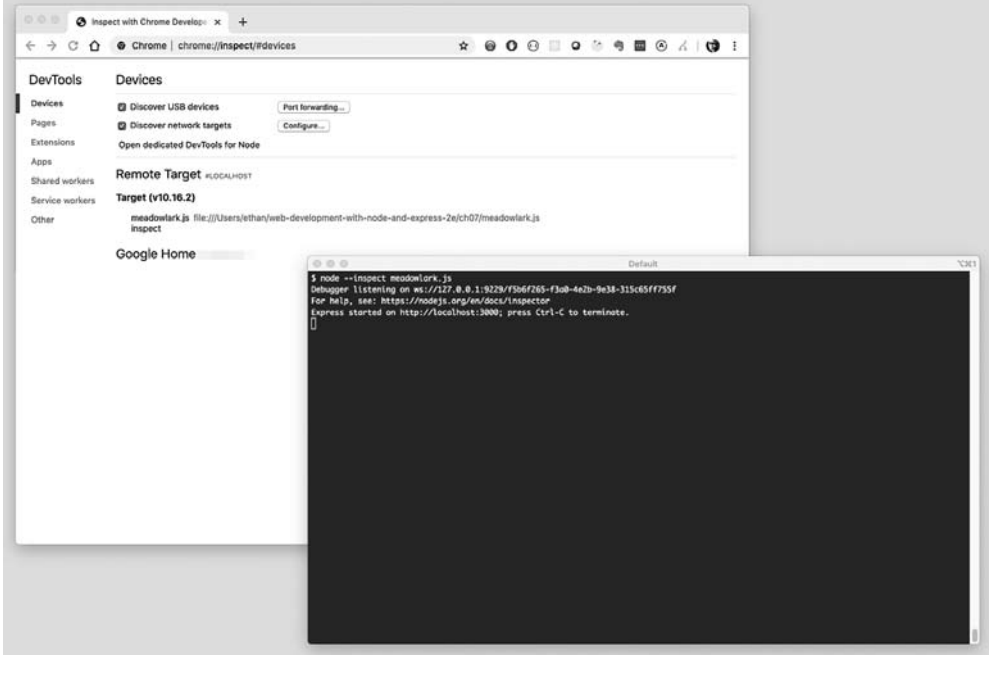

Рис. 20.1

Щелкните на вкладке Sources (Исходники) и далее на левой панели — на `Node.js`, чтобы развернуть, затем — на `file://`. Вы увидите папку, в которой находится ваше приложение; развернув эту панель, вы обнаружите все ваши исходники JavaScript

(вам будут показаны только файлы с JavaScript, а иногда файлы в формате JSON, если они где-нибудь используются). Здесь, щелкнув на любом файле, вы можете увидеть его исходник и установить точки останова (рис. 20.2).

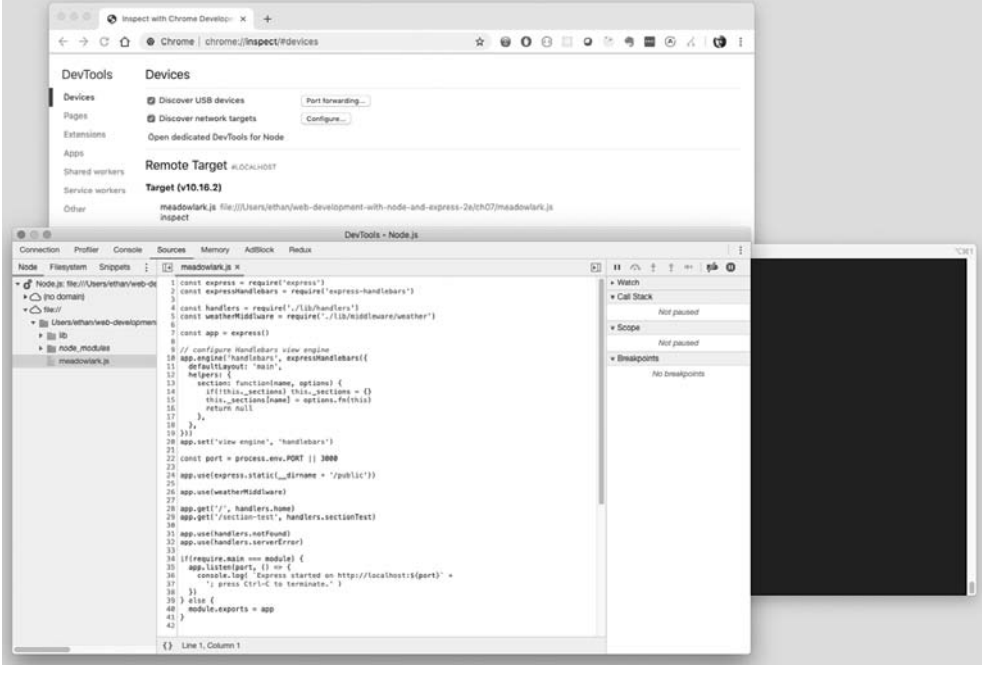

Рис. 20.2

В отличие от ситуации с командной строкой отладчика ваше приложение уже запущено: все промежуточное ПО подключено и приложение прослушивает порт. Как мы пошагово проходим наш код? Простейший способ — добавить *точку останова* (breakpoint). Это сообщит отладчику, что пора остановить выполнение на определенной строке, и далее вы пошагово сможете проходить код.

Все, что нужно, чтобы установить точку останова, — открыть файл с исходным кодом из `file://` и щелкнуть на номере строки (в левой колонке). В результате появится небольшая голубая стрелка, указывающая, что на этой строке есть точка останова (для отключения щелкните на ней еще раз). Теперь установите точку останова в одном из обработчиков маршрута. Далее в другом окне браузера перейдите по этому маршруту. Если вы используете Chrome, браузер автоматически переключится на окно отладчика, тогда как обычный браузер просто крутится, поскольку сервер был приостановлен и не отвечает на этот запрос.

В окне отладчика вы сможете пройти программу пошагово в более визуально ясной форме, чем при использовании командной строки. Вы увидите, что строка, на которой вы поставили точку останова, подсвечивается голубым цветом. Это значит,

текущая строка — строка выполнения. Отсюда вы можете получить доступ к ряду команд, как делали это из командной строки отладчика. Как и в командной строке отладчика, здесь доступны следующие действия.

- *Возобновить выполнение скрипта (F8)*. Это просто «пусть работает дальше»; вы больше не будете проходить код пошагово, во всяком случае, пока не остановитесь у другой точки останова. Обычно вы будете использовать этот прием, когда увидите все, что хотели увидеть, или захотите пропустить дальнейшее пошаговое прохождение до следующей точки останова.
- *Пройти над вызовом функции (F10)*. Если текущая строка вызывает функцию, отладчик не будет заходить в нее. То есть функция будет выполнена, а отладчик перейдет к следующей строке после вызова функции. Вы будете использовать эту возможность, когда окажетесь на вызове функции, детали которой вам сейчас неинтересны.
- *Войти в вызываемую функцию (F11)*. Эта манипуляция приведет вас в вызываемую функцию, ничего не скрывая. Если это единственное действие, которым вы пользуетесь, то увидите абсолютно все, что выполняется. На первых порах это кажется занятным, но после того, как вы позанимаетесь этим битый час, будете очень благодарны Node и Express за то, что они делают для вас!
- *Выйти из текущей функции (Shift+F11)*. Это действие выполнит остаток функции, в которой вы находитесь, и возобновит отладку на строке, следующей за строкой, вызвавшей данную функцию. Чаще всего вы будете использовать эту команду, когда либо случайно войдете в функцию, либо уже увидите в ней все, что вам нужно.

В дополнение ко всем командам управления у вас есть доступ к консоли — эта консоль выполняется в *текущем контексте вашего приложения*. Так что можете инспектировать переменные и даже менять их или вызывать функции. Это может быть невероятно удобно для отработки действительно простых вещей, а может создать путаницу, так что я не рекомендую вам слишком многое менять в запущенном приложении — очень легко потеряться.

Справа у вас есть некоторые полезные данные. Наверху находится область *наблюдения за выражениями* (watch expressions) — это выражения JavaScript, которые вы можете определить, и они будут обновляться в режиме реального времени, когда вы пошагово проходите приложение. Например, если есть конкретная переменная, изменение которой вы хотите отслеживать, можете ее там ввести.

Ниже области наблюдения за выражениями находится *стек вызовов*, который показывает, как вы пришли туда, где находитесь. То есть функция, в которой вы находитесь, была вызвана определенной функцией, и та функция тоже была вызвана какой-то функцией. Стек вызовов показывает список всех этих функций. В асинхронном мире Node стек вызовов не всегда просто разобрать и понять, особенно когда задействованы анонимные функции. Верхняя запись в этом списке — это то место, в котором вы находитесь сейчас. Прямо под ней — функция,

вызвавшая функцию, в которой вы сейчас находитесь, и т. д. Если вы щелкнете на любой записи в этом списке, то волшебным образом попадете в этот контекст: все ваши наблюдения и контекст консоли будут теперь в этом контексте.

Ниже стека вызовов находится область видимости переменных. Как следует из названия, это переменные, которые сейчас в области видимости (включая переменные в родительской области видимости, также видимые для нас). Этот раздел часто предоставляет много информации о ключевых переменных, которые на первый взгляд неинтересны. Если у вас много переменных, этот список станет громоздким и вы захотите ограничиться лишь теми переменными, которые вам интересны, в разделе наблюдения за выражениями.

Есть здесь и список всех точек останова, что на самом деле лишь ведение учета: это удобно, если вы занимаетесь отладкой сложной проблемы и у вас установлено много точек останова. Нажатие на такую точку перебросит вас напрямую в нужное место (но это не поменяет контекст, как нажатие на что-то в стеке вызовов; в этом и смысл, поскольку не каждая точка останова будет представлять активный контекст, в отличие от стека вызовов).

Иногда вам нужно провести отладку запуска приложения (например, когда вы подключаете промежуточное ПО к Express). Если запустить отладчик так, как мы это делали, он запустится мгновенно и явно до того, как мы успеем установить точку останова. К счастью, есть способ обойти эту проблему. Все, что нам нужно, — это указать `--inspect-brk` вместо `--inspect`:

```
node --inspect-brk meadowlark.js
```

Отладчик остановится на самой первой строке вашего приложения, и затем вы сможете пошагово пройти или установить точку останова там, где считаете нужным.

Chrome далеко не единственный вариант при выборе инспектора. В частности, если вы пользуетесь Visual Studio Code, то его встроенный отладчик прекрасно работает. Вместо того чтобы запустить ваше приложение с опциями `--inspect` или `--inspect-brk`, щелкните на значке Debug (Отладка) в боковом меню Visual Studio Code (перечеркнутый жучок). Вверху боковой панели инструментов вы увидите значок с зубчатым колесом, после щелчка на котором откроются настройки отладки. Единственная настройка, которая должна вас интересовать, — это `program` (программа); удостоверьтесь, что она указывает на вашу точку входа (например, `meadowlark.js`).

Вам может потребоваться установка текущего рабочего каталога или cwd, например, если вы открыли Visual Studio Code в родительской папке meadowlark.js (это то же самое, что перейти по команде cd в нужную папку до запуска node meadowlark.js).

После того как все настроено, нажмите зеленую стрелку Play (Запуск) на панели отладки — и ваш отладчик запущен. Интерфейс немного отличается от Chrome, но если вы пользуетесь Visual Studio Code, то вы, скорее всего, чувствуете себя как дома. Более подробную информацию ищите по адресу http://bit.ly/2pb7JBV.

Отладка асинхронных функций

Разработчика, впервые попробовавшего отладку асинхронного кода, ждет разочарование. Рассмотрим, например, такой код:

```
1 console.log('Ты скажи, барашек наш,');
2 fs.readFile('Не стриги меня пока.', function(err, data){
3         console.log('Сколько шерсти ты нам дашь?');
4         console.log(data);
5 })
6 console.log('Дам я шерсти три мешка.')
```

Если вы новичок в асинхронном программировании, то ожидаете увидеть следующее:

```
Ты скажи, барашек наш,
Сколько шерсти ты нам дашь?
Не стриги меня пока.
Дам я шерсти три мешка.
```

Но в действительности увидите такое:

```
Ты скажи, барашек наш,
Дам я шерсти три мешка.
Сколько шерсти ты нам дашь?
Не стриги меня пока.
```

Если вы запутались, отладка вряд ли поможет разобраться. Вы начинаете со строки 1, затем проходите шаг, что приводит к строке 2. После входите, ожидая, что войдете в функцию и окажетесь на строке 3, но в действительности оказываетесь на строке 5! Это потому, что `fs.readFile` выполняет функцию лишь тогда, *когда завершит чтение файла*, чего не произойдет, пока ваше приложение простаивает. Так что вы проходите над строкой 5 и попадаете на строку 6, затем продолжаете попытки войти, но так и не добираетесь до строки 3 (в конечном итоге доберетесь, но это может занять немало времени).

Если вы хотите провести отладку строк 3 или 4, все, что вам нужно сделать, — это поставить точку останова на строке 3 и запустить отладчик в режиме выполнения. Когда файл будет прочитан и функция вызвана, вы остановитесь на этой строке и, надеюсь, все будет ясно.

Отладка Express

Если вам, как и мне, приходилось видеть множество технически сложных фреймворков, идея пошагового прохождения исходного кода может показаться безумием или пыткой. Исследование исходного кода Express не детская задачка, но разобраться в этом *действительно* может каждый, кто хорошо понимает JavaScript и Node. А иногда, когда вы сталкиваетесь с определенными проблемами в своем коде, отладку этих проблем можно выполнить посредством изучения исходного кода самого Express или промежуточного ПО.

В этом разделе приведен краткий обзор исходного кода Express, чтобы вы смогли заняться более эффективной отладкой своих приложений Express. Для каждой его части я дам имя файла, относящегося к корневому каталогу Express (у вас он находится в каталоге `node_modules/express`), и имя функции. Я не буду использовать номера строк, поскольку они могут отличаться в зависимости от используемой версии Express.

- *Создание приложения Express* (`lib/express.js`, `function createApplication()`). Здесь приложение Express начинает свою жизнь. Это функция, запускаемая при вызове `const app = express()` в коде.
- *Инициализация приложения Express* (`lib/application.js`, `app.defaultConfiguration`). Это то место, где инициализируется Express. Здесь можно увидеть все значения по умолчанию, с которыми Express начинает работу. Необходимость ставить здесь точку останова возникает очень редко, но полезно хотя бы раз пройти через выполнение этой функции, чтобы почувствовать настройки Express по умолчанию.
- *Добавление промежуточного ПО* (`lib/application.js`, `app.use`). Эта функция вызывается каждый раз, когда промежуточное ПО подключается к Express (независимо от того, кто это явно делает: вы сами, Express или стороннее приложение). При кажущейся простоте необходимо приложить некоторые усилия, чтобы понять, как работает эта функция. Иногда полезно установить здесь точку останова (при запуске приложения понадобится использовать параметр `--debug-brk`, иначе все промежуточное ПО будет добавлено уже после установки точки останова), но это может удивить: вы ахнете, когда увидите, как много промежуточного ПО подключено к Express в обычном приложении.
- *Отобразить представление* (`lib/application.js`, `app.render`). Это еще одна довольно содержательная функция. Полезной она будет, если вам необходимо отладить сложные проблемы, связанные с представлением. Если вы пошагово проходите эту функцию, то увидите, как выбирается и вызывается механизм представления.
- *Запрос расширений* (`lib/request.js`). Вы, вероятно, будете удивлены, какой это небольшой и простой для понимания файл. Большинство методов, которые

Express добавляет к объектам запросов, — очень простые и удобные функции. Редко необходимо пошагово проходить этот код и ставить точки останова в силу его простоты. Однако иногда полезно посмотреть на этот код, если вы хотите понять, как работают некоторые удобные методы Express.

- *Отправить ответ* (`lib/response.js`, `res.send`). Почти не имеет значения, как именно вы создаете ответ — `.send`, `.render`, `.json` или `.jsonp`, — он все равно будет добавлен к этой функции (исключение — `.sendFile`). Так что это удобное место, чтобы поставить точку останова, потому что она должна быть вызвана в каждом ответе. Затем вы можете использовать стек вызовов, чтобы посмотреть, как вы сюда попали. Это может помочь с выяснением места, где возникла проблема.
- *Расширения объекта ответа* (`lib/response.js`). Функция `res.rend` может быть довольно содержательной, но большинство других методов в объекте ответа достаточно просты. Иногда полезно ставить точки останова на этих функциях, чтобы увидеть, как ваше приложение отвечает на запрос.
- *Промежуточное ПО для предоставления статических файлов* (`node_modules/serve-static/index.js`, функция `staticMiddleware`). В целом, если статические файлы не делают то, что вы от них ожидаете, проблема может заключаться в маршрутизации, а не в промежуточном ПО `static`, над которым маршрутизация имеет приоритет. Так что, если у вас есть файл `public/test.jpg` и маршрут `/test.jpg`, промежуточное ПО `static` никогда не будет вызвано в силу наличия маршрута. Однако, если вам нужно определить специфику того, как по-разному устанавливаются заголовки для статических файлов, полезно будет пройтись по промежуточному ПО для статических файлов.

Вы будете ломать голову, выясняя, где все это промежуточное ПО, поскольку в Express его очень мало (промежуточное ПО `static` и маршрутизатор — это исключения).

Точно так же, как при решении сложной проблемы полезно углубиться в исходный код Express, может возникнуть необходимость взглянуть на исходники промежуточного ПО. Их слишком много, чтобы перечислить все, но я хотел бы упомянуть три компонента, имеющих ключевое значение для понимания происходящего в приложении Express.

- *Промежуточное ПО для управления сеансами* (`node_modules/express-session/index.js`, функция `session`). Здесь содержится простой код, реализующий работу сеансов. Вы можете установить точку останова в этой функции, если есть проблемы, связанные с сеансами. Имейте в виду, что предоставление механизма хранения для промежуточного ПО `session` остается на ваше усмотрение.
- *Промежуточное ПО для обеспечения логирования* (`node_modules/morgan/index.js`, функция `logger`). Промежуточное ПО `logger` выполняется с целью помочь в отладке, а не для того, чтобы заниматься самоотладкой. Тем не менее в работе

по логированию есть некоторые тонкости, которые можно выявить лишь посредством пошагового прохождения промежуточного ПО `logger` один или два раза. Проделав это впервые, я ликовал. Потом научился использовать логирование куда эффективнее в своих приложениях, так что рекомендую пройтись по промежуточному ПО хотя бы раз.

- *Парсер URL-кодированного тела* (`node_modules/body-parser/index.js`, функция `urlencoded`). То, как парсируются тела запросов, зачастую является загадкой. На самом деле это несложно. Пошаговое прохождение промежуточного ПО поможет понять способ, посредством которого работают HTTP-запросы. Вне учебных целей вы вряд ли будете часто заходить в это промежуточное ПО для отладки.

Резюме

В этой книге мы немало говорили о промежуточном ПО. Я не могу перечислить все ориентиры, которые понадобятся вам при изучении внутреннего устройства Express, но надеюсь, что изложенные моменты приоткроют некоторые тайны Express и подстегнут вас исследовать исходный код фреймворка, когда это нужно. Промежуточное ПО существенно различается не только по качеству, но и по доступности: одно промежуточное ПО невероятно трудно понять, другое достаточно примитивно. В любом случае не бойтесь экспериментировать: если что-то окажется слишком сложным, вы можете это бросить (если, конечно, у вас нет в этом насущной необходимости), а если нет — можете что-то и выучить.

21 Ввод в эксплуатацию

Сегодня великий день: несколько недель или месяцев вы трудились над своим детищем и сейчас ваш сайт или сервис готов к запуску. Это так просто, как щелкнуть выключателем? Или все-таки нет?

Из этой главы (которую вам на самом деле следовало бы прочесть за несколько *недель* до запуска, а не за день) вы узнаете о некоторых моментах регистрации домена, доступных услугах хостинга, методах перемещения из промежуточной среды в среду продукта, техниках развертывания и тех вещах, которые вы должны рассмотреть при выборе сервисов, используемых в промышленной эксплуатации.

Регистрация домена и хостинг

Многие люди часто не могут понять, в чем разница между *регистрацией домена* и *хостингом*. Если вы читаете эту книгу, то наверняка не относитесь к их числу, но я уверен, что вы знаете таких людей, и среди них могут быть, например, ваши клиенты или менеджеры.

Каждый сайт и сервис в Интернете могут быть идентифицированы посредством *адреса интернет-протокола (IP-адреса)*, а иногда и нескольких. Людям эти номера запомнить непросто (ситуация будет лишь ухудшаться с грядущим распространением IPv6), но вашему компьютеру эти числа нужны для отображения веб-страницы. Вот здесь и понадобится *доменное имя*. Оно устанавливает соответствие между удобным человеку именем (например, google.com) и IP-адресом (74.125.239.13 или 2601:1c2:1902:5b38:c256:27ff:fe70:47d1).

Аналогией в реальном мире может быть разница между названием компании и физическим адресом. Доменное имя подобно названию компании (Apple), а IP-адрес — физическому адресу (Apple Park, 1, Купертино, Калифорния, США, 95014). Если вам нужно сесть в автомобиль и приехать в штаб-квартиру Apple, вы должны знать физический адрес. К счастью, если вы знаете название компании, то, вероятно, сможете выяснить и физический адрес. Другая причина, по которой

эта абстракция будет полезна: организация может переехать (получив новый физический адрес), но люди по-прежнему смогут ее найти (действительно, Apple физически переместил свою штаб-квартиру в период между выходами первого и второго изданий этой книги).

Хостинг описывает реальные компьютеры, на которых запущен ваш сайт. Продолжая аналогию, его можно сравнить с домами, которые вы увидите, придя по физическому адресу. Людей часто сбивает с толку, что у регистрации домена мало общего с хостингом, ведь редко бывает так, что вы покупаете домен и платите за хостинг одной и той же компании (подобно тому как вы покупаете землю у одного человека и платите другому, чтобы он построил дом и обслуживал его).

Конечно, можно разместить сайт и без доменного имени, но это непродуманное решение: IP-адреса совсем не годятся для рыночного продукта! Обычно, когда вы покупаете хостинг, вам автоматически присваивается поддомен (о них расскажу чуть позже), который можно рассматривать как что-то среднее между пригодным для рынка именем домена и IP-адресом (например, `ec2-54-201-235-192.us-west-2.compute.amazonaws.com`).

После того как у вас появится домен и вы запустите сайт в эксплуатацию, он будет доступен по многим URL, например:

- http://meadowlarktravel.com/;
- http://www.meadowlarktravel.com/;
- http://ec2-54-201-235-192.us-west-2.compute.amazonaws.com/;
- http://54.201.235.192/.

Благодаря соответствиям, установленным между доменами и IP-адресами, все эти адреса указывают на один и тот же сайт. После того как запрос приходит на ваш сайт, некоторые действия можно выполнить именно на основе используемого URL. Например, если кто-то заходит на ваш сайт по IP-адресу, вы можете автоматически перенаправить его на доменное имя, хотя это редкость (чаще перенаправляют с http://meadowlarktravel.com/ на http://www.meadowlarktravel.com/).

Большинство регистраторов доменов предлагают услуги хостинга (или являются партнерами компаний, занимающихся этим). Я никогда не считал хостинг у регистраторов каким-то привлекательным вариантом (за исключением AWS) и рекомендую разделять доменную регистрацию и хостинг из соображений безопасности и гибкости.

Система доменных имен

Система доменных имен (Domain Name System, DNS) отвечает за установку соответствия между именами доменов и IP-адресами. Система довольно сложная, но с DNS связаны некоторые вещи, которые вы должны знать, как владелец сайта.

Безопасность

Вы должны хорошо понимать *ценность доменных имен*. Если хакер захочет нарушить безопасность вашего хостинга и взять его под свой контроль, а вы при этом сохраните контроль над доменом, то сможете получить новый хостинг и перенаправить на него домен. Если же опасности подвергся *домен*, то проблема куда серьезнее. Ваша репутация связана с вашим доменом, и хорошие доменные имена тщательно охраняются. В мире всегда найдутся те, кто будет активно пытаться завладеть вашим доменом, особенно если он короткий или запоминающийся, потому что его можно продать, разрушить вашу репутацию или шантажировать вас. Таким образом, *для обеспечения безопасности домена вам необходимо принять очень серьезные меры*. Возможно, они должны быть даже серьезнее тех, к которым вы прибегали для обеспечения безопасности данных (конечно, это зависит от данных). Я видел, как люди тратят время и деньги на безопасность хостинга, но при этом соглашаются на самую дешевую и сомнительную регистрацию домена, какую только могут найти. Не повторяйте таких ошибок. (К счастью, качественная регистрация домена не так уж и дорого стоит.)

Учитывая важность защиты домена, при регистрации вы должны использовать все возможные средства обезопасить его. Как минимум стоит использовать надежные уникальные пароли и придерживаться надлежащей гигиены паролей (не записывайте их на стикере, прикрепленном к монитору). Пользуйтесь услугами регистратора, предлагающего двухфакторную аутентификацию. Не бойтесь задавать регистратору острые вопросы о том, что требуется для разрешения изменений в вашей учетной записи. Регистраторы, которые я могу порекомендовать, — AWS Route 53, Name.com и Namecheap.com. Все три предлагают двухфакторную аутентификацию, и, по моему опыту, у всех хорошая поддержка, а онлайновые панели управления простые и надежные.

Когда вы зарегистрируете домен, вам нужно сообщить сторонний адрес электронной почты, связанный с этим доменом (то есть если вы регистрируете meadowlarktravel.com, то не можете использовать admin@meadowlarktravel.com как регистрационный адрес электронной почты). Так как любая система безопасности надежна настолько, насколько надежно ее слабейшее звено, вы должны использовать адрес электронной почты с максимальной безопасностью. Довольно часто используют учетную запись Gmail или Outlook, и если вы пойдете тем же путем, то должны использовать те же стандарты безопасности, что и при регистрации самого домена (хорошая гигиена пароля и двухфакторная аутентификация).

Домены верхнего уровня

То, чем заканчивается ваш домен (например, .com или .net), называется *доменом верхнего уровня* (top-level-domain, TLD). Есть два типа доменов верхнего уровня: коды стран и общие. Домены верхнего уровня в виде кода страны (например, .us, .es и .uk) предназначены для обеспечения географической классификации. Однако есть некоторые ограничения для тех, кто может приобрести эти домены верхнего

уровня (в конце концов, Интернет — действительно глобальная сеть), так что они часто используются для «остроумных» доменов, таких как placehold.it и goo.gl.

Общие домены верхнего уровня (general TLD, gTLD) включают знакомые нам .com, .net, .gov, .fed, .mil и .edu. Каждый может приобрести свободный домен .com или .net, но для других упомянутых доменов есть определенные ограничения (табл. 21.1).

Таблица 21.1. Ограничения gTLD

Домен верхнего уровня	Дополнительная информация
.gov, .fed	https://www.dotgov.gov
.edu	http://net.educause.edu/edudomain
.mil	Военнослужащие или контрагенты должны связаться с их IT-отделом либо информационным центром

Корпорация по управлению доменными именами и IP-адресами (Internet Corporation for Assigned Names and Numbers, ICANN) отвечает за управление доменами верхнего уровня, хотя и делегирует значительную часть фактического администрирования другим организациям. Недавно ICANN разрешила использовать много новых глобальных доменов верхнего уровня, таких как .agency, .florist, .recipes и даже .ninja. В обозримом будущем .com наверняка останется премиальным доменом верхнего уровня и получить домен в нем будет сложнее всего. Люди, которым посчастливилось (и проявившие исключительную проницательность) купить домен в .com в годы становления Интернета, получили огромные выплаты за лучшие домены (например, Facebook купил fb.com за баснословную сумму — 8,5 миллиона долларов).

Учитывая определенный дефицит доменов .com, люди переходят к альтернативным доменам верхнего уровня или используют .com.us в попытке получить домен, полностью отражающий сущность их компании. При выборе домена нужно рассмотреть, как он будет использоваться. Если вы планируете заниматься преимущественно электронным маркетингом, когда пользователи чаще будут нажимать на ссылку, а не вводить доменное имя вручную, то вам, вероятно, нужно сосредоточиться на получении броского или осмысленного домена, а не короткого. Если вы фокусируетесь на печатной рекламе или у вас есть основания полагать, что люди будут вводить ваш URL вручную на своих устройствах, то лучше рассмотреть альтернативные домены верхнего уровня, где вы сможете получить более короткое доменное имя. Обычная практика — владение двумя доменами: коротким, что прост в наборе, и подлиннее, более подходящим для маркетинга.

Поддомены

Домен верхнего уровня идет после вашего домена, а поддомен — до него. До сих пор наиболее распространенным поддоменом является www. Я никогда его не любил. В конце концов, вы сидите за компьютером, *используете* Всемирную паутину (World Wide Web), и я уверен, что не запутаетесь, если в адресе не будет www, на-

поминающего вам, что вы делаете. По этой причине рекомендую не использовать поддомен для вашего основного домена: http://meadowlarktravel.com/ вместо http://www.meadowlarktravel.com/. Это короче и не захламляет адресную строку, а благодаря перенаправлениям нет опасности потерять посетителей, автоматически запускающих все с www.

Поддомены используются и в других целях. Я иногда встречаю такие названия, как blogs.meadowlarktravel.com, api.meadowlarktravel.com и m.meadowlarktravel.com (для мобильной версии сайтов). Часто они появляются по техническим причинам: поддомен проще использовать, если, например, ваш блог использует совершенно другой сервер, чем весь остальной сайт. Впрочем, хороший прокси может перенаправлять локальный трафик, основываясь или на поддомене, или на пути, так что при выборе того, что вы будете использовать — поддомен или путь, фокусируйтесь скорее на контенте, чем на технологии (как сказал Тим Бернерс-Ли, «URL выражает вашу информационную, а не техническую архитектуру»).

Я рекомендую использовать поддомены для разделения ощутимо разных частей сайта или сервиса. Например, удачной идеей для использования поддомена было бы размещение вашего API на api.meadowlarktravel.com. Микросайты (их внешний вид отличается от вида остального сайта, они используются, как правило, для выделения какого-то отдельного продукта или предмета) — тоже хорошие кандидаты для поддоменов. Другое разумное использование поддоменов — отделение интерфейса администратора от публичного интерфейса (admin.meadowlarktravel.com, только для сотрудников).

Доменный регистратор, если вы не укажете иное, будет перенаправлять ваш трафик на сервер независимо от поддомена. От сервера (или прокси) зависит, какие действия нужно совершать по отношению к поддомену.

Сервер имен

Сервер имен — это «клей», позволяющий доменным именам работать. Это то, что вам придется предоставить, когда вы устанавливаете хостинг для своего сайта. Как правило, все довольно просто, поскольку хостинг сделает бóльшую часть работы за вас. Например, мы выбираем хостинг meadowlarktravel.com на DigitalOcean. При настройке учетной записи хостинга на DigitalOcean вам дадут имена серверов имен DigitalOcean (их несколько, на всякий случай). DigitalOcean, как и большинство хостинг-провайдеров, дают своим серверам имена ns1.digitalocean.com, ns2.digitalocean.com и т. д. Перейдите к вашему доменному регистратору, установите имена серверов для домена, который вы хотите хостить, — и все готово.

В этом случае способ установки соответствия будет таким.

1. Посетитель сайта переходит на http://meadowlarktravel.com/.
2. Браузер отправляет запрос на сетевую систему компьютера.
3. Сетевая система компьютера, которой интернет-провайдер дал IP-адрес и DNS-сервер, просит DNS распознать meadowlarktravel.com.

4. DNS-распознаватель знает, что meadowlarktravel.com обрабатывается ns1.digitalocean.com, так что он просит ns1.digitalocean.com дать IP-адрес meadowlarktravel.com.
5. Сервер в ns1.digitalocean.com получает запрос, распознает, что meadowlarktravel.com — это действительно активный аккаунт, и возвращает соответствующий IP-адрес.

Это наиболее распространенный способ установить соответствие домена, но не единственный. Поскольку у сервера (или прокси), который фактически обслуживает ваш сайт, есть IP-адрес, вы можете убрать посредника, зарегистрировав IP-адрес с распознавателями DNS (это эффективно убирает посредника в виде сервера имен ns1.digitalocean.com в предыдущем примере). Для того чтобы этот подход работал, хостинг должен назначить вам статический IP-адрес. Как правило, хостинг-провайдеры дают своим серверам динамический IP-адрес. Это значит, что он может меняться без предварительного уведомления, и в таком случае эта схема не будет эффективно работать. Иногда статический IP-адрес стоит больше динамического — проконсультируйтесь со своим хостинг-провайдером.

Если вы хотите установить соответствие домена вашему сайту напрямую, минуя серверы имен вашего хостера, вам добавят *запись* либо A, либо CNAME. Запись типа A устанавливает прямое соответствие между доменным именем и IP-адресом, а CNAME — соответствие одного доменного имени другому. Записи CNAME обычно менее гибкие, так что записи типа A более предпочтительны.

Если вы используете AWS для сервера имен, то у него, кроме A и CNAME, есть еще запись alias, которая дает много преимуществ при указании ее сервисам с хостингом на AWS. За более подробной информацией обращайтесь к документации AWS (https://amzn.to/2pUuDhv).

Независимо от того, какую технику вы используете, карта соответствия доменных имен обычно жестко кэшируется. Это значит, что при изменении записи домена прикрепление его к новому серверу может занять до 48 часов. Учтите еще, что это зависит от местоположения: если вы видите, что домен работает в Лос-Анджелесе, то клиент в Нью-Йорке сможет увидеть, что домен прикреплен к предыдущему серверу. По моему опыту, 24 часов обычно хватает для распознавания домена на всей континентальной части США, а на международном уровне это занимает до 48 часов.

Если вам нужно, чтобы что-то начало работать точно в определенное время, не полагайтесь на изменения в DNS. Лучше поменяйте сервер, чтобы он перенаправлял на сайт или страницу Скоро будет, и внесите изменения в DNS заблаговременно, до фактического перехода. Затем в назначенный момент вы можете переключить сайт-заглушку на нужный сайт, и посетители увидят изменения сразу, независимо от того, в какой части света они находятся.

Хостинг

Выбор хостинга огромен. Node был запущен с размахом, и отовсюду поступали предложения удовлетворить потребности Node в хостинге. Выбор хостинг-провайдера зависит от ваших потребностей. Если вы полагаете, что ваш сайт будет следующим Amazon или Twitter, у вас будет другой набор проблем, чем при создании сайта для местного клуба коллекционеров марок.

Традиционный или облачный хостинг?

«Облако» — один из наиболее туманных технических терминов, возникших за последние годы. Это модное слово употребляют, когда хотят сказать «Интернет» или «часть Интернета». Но этот термин не совсем бесполезен. Хостинг в облаке в определенной степени подразумевает превращение компьютерных ресурсов в товар массового потребления, хотя это и не оговорено в техническом определении. То есть мы больше не рассматриваем сервер как некий физический объект — это просто распределенный ресурс, находящийся где-то в облаке. Я, конечно, упрощаю: компьютерные ресурсы различаются (в том числе по цене) в зависимости от объема памяти, количества процессоров и пр. Но разница, которую я хочу особенно подчеркнуть, в том, что вы даже можете не знать, на каком сервере хостится ваше приложение: на физическом или в облаке, и не поймете, что его перенесли с одного сервера на другой.

У облачного хостинга высокая степень *виртуализации*. То есть сервер (-ы), на котором (-ых) запущено ваше приложение, — обычно не физическая, а виртуальная машина, но она работает на физическом сервере. Облачный хостинг не придумал эту идею, но она стала его синонимом.

Хотя у облачного хостинга скромное происхождение, на данный момент это гораздо больше, чем просто распределенный сервер. Основные поставщики облачного хостинга предлагают много инфраструктурных сервисов, которые (теоретически) облегчают бремя поддержки и предоставляют высокую степень масштабируемости. Эти сервисы включают хранение в базе данных, файловые хранилища, сетевые очереди, аутентификацию, обработку видео, телекоммуникационные сервисы, движки искусственного интеллекта и многое другое.

Облачный хостинг может немного насторожить. Не знать ничего о том, на какой реальной физической машине запущен ваш сервер, верить, что его не затронут другие серверы, работающие на том же компьютере, — все это кажется каким-то странным. Однако в реальности ничего не изменилось: кто-то по-прежнему обслуживает физическое и сетевое оборудование, на котором работают ваши веб-приложения. Изменилось лишь то, что они больше не связаны с аппаратным обеспечением.

Я считаю, что традиционный хостинг (за неимением лучшего термина назовем его так) в конечном счете исчезнет. Это не говорит о том, что хостинговые компании уйдут из бизнеса (хотя некоторые, конечно, уйдут — это неизбежно), — они просто начнут сами предлагать облачный хостинг.

XaaS

Изучая облачный хостинг, вы столкнетесь с сокращениями SaaS, PaaS, IaaS и FaaS.

- *Программное обеспечение как сервис (Software as a Service, SaaS)*. SaaS обычно описывает предоставляемое вам программное обеспечение (сайты, приложения): вы просто его используете. Пример — «Google Документы» или Dropbox.
- *Платформа как сервис (Platform as a Service, PaaS)*. PaaS предоставляет всю инфраструктуру (операционные системы, сети). Все, что вам нужно, — написать ваши приложения. Граница между PaaS и IaaS зачастую размыта (да и вы как разработчик иногда будете сомневаться, где что), но в целом это та модель сервиса, которую мы обсуждали в данной книге. Если вы запускаете сайт или веб-сервис, PaaS — это то, что вам нужно.
- *Инфраструктура как сервис (Infrastructure as a Service, IaaS)*. IaaS дает вам больше гибкости, но у нее есть своя цена. Все, что вы получите, — это виртуальные машины и соединяющая их основная сеть. Вы будете отвечать за установку и поддержку операционных систем, баз данных и сетевых политик. Если вам не нужен такой контроль над своей средой, вы наверняка захотите использовать PaaS. (Обратите внимание, что PaaS позволяет *выбрать* операционную систему и сетевую конфигурацию — вам не нужно делать это самостоятельно.)
- *Функции как сервис (Functions as a Service, FaaS)*. FaaS описывает такие предложения, как AWS Lambda, Google Functions и Azure Functions, предоставляя способ запуска отдельных функций в облаке без необходимости самостоятельно настраивать среду исполнения. Это лежит в основе того, что принято называть бессерверной (serverless) архитектурой.

Гиганты хостинга

Компании, занятые обслуживанием Интернета (или по крайней мере те, что инвестировали значительные средства в управление Интернетом), поняли, что с превращением компьютерных ресурсов в товар массового потребления у них появился продукт, который они могут продать. Microsoft, Amazon и Google предлагают сервисы облачного вычисления, и они весьма неплохи.

Эти три сервиса стоят примерно одинаково: если потребности вашего хостинга скромны, то разница в цене будет минимальной. Если потребности в пропускной способности или хранилище огромны, сервисы нужно оценивать более тщательно, поскольку разница в стоимости может существенно возрасти.

Хотя Microsoft редко приходит на ум, когда мы рассматриваем платформы с открытым исходным кодом, я бы не упускал из виду Azure. Эта устоявшаяся и надежная платформа, к тому же Microsoft из кожи вон лез, чтобы сделать ее дружественной не только Node, но и сообществу с открытым исходным кодом. Microsoft предлагает месячный пробный период использования Azure, и это хороший способ понять, отвечает ли этот сервис вашим потребностям. Если вы рассма-

триваете вариант из большой тройки, я рекомендую бесплатный пробный период для оценки Azure. Microsoft предлагает Node API для всех его крупных сервисов, включая облачный хостинг хранения данных. В дополнение к отличному хостингу Node Azure предлагает отличную систему облачного хранения данных (с JavaScript API), а также хорошую поддержку MongoDB.

Amazon предлагает наиболее полный набор ресурсов, включая SMS (текстовые сообщения), облачное хранение, сервисы электронной почты, сервисы оплаты (электронная коммерция), DNS и т. д. Кроме того, у Amazon есть бесплатный пакет, так что их очень просто оценить.

Облачная платформа Google прошла долгий путь развития и теперь предлагает надежный хостинг Node и ожидаемо великолепную интеграцию с их собственными сервисами (в частности, эффективные сопоставление, аутентификацию и поиск).

В дополнение к большой тройке стоит упомянуть Heroku (https://www.heroku.com/), который уже некоторое время обслуживает желающих получить быстрый хостинг и гибкие приложения Node. Мне также повезло с DigitalOcean (https://www.digitalocean.com/), который в большей мере нацелен на предоставление контейнеров и ограниченного числа сервисов весьма дружественным пользователю способом.

Небольшие хостинговые компании

У небольших компаний, предоставляющих услуги хостинга (в англоязычной литературе их иногда называют boutique («лавочки») за неимением лучшего термина), может не быть такой инфраструктуры или ресурсов, как у Microsoft, Amazon или Google, но это не значит, что они не предлагают что-то хорошее.

Поскольку небольшие хостинговые компании не могут конкурировать с точки зрения инфраструктуры, они обычно сосредоточены на обслуживании клиентов и поддержке. Если вам нужна хорошая поддержка, рассмотрите их. Если у вас есть хостинг-провайдер, которым вы довольны, не стесняйтесь спрашивать, предлагает ли он (или планирует ли предложить) хостинг Node.

Развертывание

Меня до сих пор поражает, что в 2019 г. все еще есть люди, использующие FTP для развертывания своих приложений. Если вы так делаете, пожалуйста, прекратите. FTP совсем не безопасен. Вы передаете в незашифрованном виде не только все файлы, но также *имя пользователя и пароль*. Если ваш хостинг-провайдер не предлагает других вариантов, найдите нового. Если у вас действительно нет выбора, убедитесь, что используете уникальный пароль, который не применяете больше нигде.

Как минимум вы должны использовать SFTP или FTPS (не путать), но вам действительно стоит рассмотреть сервис *непрерывного развертывания* (continuous delivery, CD).

Идея непрерывного развертывания в том, что вы всегда ненамного (на несколько недель или даже дней) отстаете от выпущенной версии. CD обычно используется

наряду с *непрерывной интеграцией* (continuous integration, CI) — автоматизированным процессом интеграции работы разработчиков и тестирования.

По большому счету, чем больше вы автоматизируете процесс, тем проще будет вашим разработчикам. Только вообразите: вы производите слияние изменений и автоматически получаете сведения, что успешно прошли модульные тесты, затем интеграционные тесты, а после видите ваши изменения онлайн за считаные минуты! Это великая цель, но вам придется немало потрудиться, чтобы все это установить, плюс в течение долгого времени нужна будет поддержка.

Хотя сами шаги одни и те же (прогонка модульных тестов, прогонка интеграционных тестов, развертывание на сервере обкатки, развертывание на промышленных серверах), процесс настройки конвейеров CI/CD (этот термин будет часто встречаться при обсуждении CI/CD) существенно различается.

Взгляните на некоторые доступные для CI/CD варианты и выберите тот, который вам подходит.

- AWS CodePipeline (https://amzn.to/2CzTQAo). Если у вас хостинг на AWS, CodePipeline должен быть первым в вашем списке, поскольку это самый простой путь к CI/CD. Он весьма надежен, но, как мне кажется, не так дружественен к пользователям, как некоторые другие варианты.
- Microsoft Azure Web Apps (http://bit.ly/2CEsSI0). Если у вас хостинг на Azure, вам подойдет Web Apps (заметили тенденцию?). У меня нет большого опыта взаимодействия с этим сервисом, но к нему тепло относятся в сообществе.
- Travis CI (https://travis-ci.org/). Появился довольно давно, у него есть большая база верных пользователей и хорошая документация.
- Semaphore (https://semaphoreci.com/). Его легко установить и настроить, но он предоставляет мало функций, а его базовые (дешевые) планы — медленные.
- Google Cloud Build (http://bit.ly/2NGuIys). Его я еще не пробовал, но он кажется надежным и, как это было с CodePipeline и Azure Web Apps, вероятно, является лучшим выбором, если вы используете хостинг на Google Cloud.
- CircleCI (https://circleci.com/). Еще один CI, который вышел достаточно давно и к которому по-прежнему относятся с любовью.
- Jenkins (https://jenkins.io/). Очередной обладатель обширного сообщества. По моему опыту, он не соответствует современным методам развертывания, как и некоторые другие приведенные здесь варианты. Однако недавно вышла его новая версия, что звучит многообещающе.

Сервисы CI/CD автоматизируют созданные вами действия. Но вам по-прежнему нужно писать код, определять схему версионирования, писать высококачественные модульные и интеграционные тесты и способ их запуска, а также понимать инфраструктуру развертывания. Примеры из этой книги могут быть легко автоматизированы: почти все можно развернуть на одном сервере с запущенным экземпляром Node. Однако при росте вашей инфраструктуры сложность конвейера CI/CD также возрастет.

Роль Git в развертывании

Сильнейшим (и слабейшим) качеством Git является его гибкость. Он может быть адаптирован к любому мыслимому рабочему процессу. Я рекомендую создать одну или несколько веток *специально для развертывания*. Например, у вас есть ветка `production` и ветка `staging`. То, как вы используете эти ветки, в значительной степени зависит от особенностей рабочего процесса.

Один из популярных подходов — ход от `master` к `staging` и оттуда к `production`. После того как какие-то изменения в `master` уже готовы ко вводу в эксплуатацию, вы можете слить их со `staging`. Когда они будут одобрены на `staging`, вы сливаете их с `production`. В этом есть логика, но я не люблю беспорядок, который при этом создается (слияния, слияния везде). Кроме того, если у вас много функциональных возможностей, которые нужно обкатать и продвигать на производственный сервер в разном порядке, это может быстро привести к беспорядку.

Полагаю, что лучше всего слить `master` со `staging` и, когда вы уже будете готовы к вводу в эксплуатацию с изменениями, слить `master` с `production`. В этом случае `staging` и `production` будут меньше связаны: вы можете даже сделать несколько промежуточных веток для эксперимента с различным функционалом перед вводом в эксплуатацию (и вы можете слить вещи, отличающиеся от тех, которые находятся в `master`). Только тогда, когда что-то будет одобрено для производственной версии, вы это сливаете в `production`.

А что, если нужно отменить изменения? Здесь могут возникнуть сложности. Есть много техник для отмены изменений, таких как инверсия коммита для отката до предшествующих коммитов (`git revert`), но эти техники сложны и могут привести к возникновению проблем. Обычным способом является создание меток (например, `git tag v1.2.0` на вашей ветке `production`) при каждом развертывании. Если вам понадобится вернуться к определенной версии, то в вашем распоряжении всегда есть эти метки.

В конце концов, на усмотрение ваше и команды остается то, какой рабочий процесс использовать с Git. Куда важнее выбора рабочего процесса будет постоянство, с которым вы станете его использовать, а также связанные с ним обучение и общение.

Мы уже обсуждали полезность хранения ваших бинарных активов (мультимедиа и документов) отдельно от кода. Развертывание с помощью Git предлагает еще один стимул для использования этого подхода. Если в вашем репозитории есть 4 Гбайт мультимедийных данных, им понадобится целая вечность для клонирования и у вас будет ненужная копия всех данных для каждого рабочего сервера.

Развертывание с помощью Git вручную

Если вы еще не готовы приступить к установке CI/CD, можете начать с развертывания вручную с помощью Git. Преимуществом этого подхода является то, что вы привыкнете к этапам развертывания и проблемам, связанным с ним, что сослужит хорошую службу на шаге автоматизации.

Для каждого сервера, на котором вы хотите развернуться, нужно будет клонировать репозиторий, создать ветку `production`, затем установить инфраструктуру, необходимую для запуска/перезапуска приложения (что будет зависеть от выбора платформы). Когда вы обновите ветку `production`, придется перейти на каждый сервер, запустить `git pull --ff-only`, запустить `npm install --production`, после чего перезапустить приложение. Если развертка происходит нечасто и у вас немного серверов, это не будет слишком сложно, но, если вы обновляете часто, такой подход быстро устареет и вы захотите найти какой-то способ автоматизировать систему.

Аргумент --ff-only к git pull позволяет выполнить скачивание изменений в режиме перемотки (fast-forward), предотвращая автоматическое слияние или перебазирование. Если вы знаете, что скачивание осуществится в таком режиме, можете спокойно пропустить его. Однако, если у вас это войдет в привычку, вы никогда не вызовете случайно слияние или перебазирование!

То, что вы делаете здесь, воспроизводит вашу работу при разработке, за тем исключением, что вы делаете это на удаленном сервере. При ручном процессе всегда есть риск человеческих ошибок, и я рекомендую использовать этот подход лишь как ступеньку на пути к более автоматизированному развертыванию.

Резюме

Развертывание вашего сайта (особенно в первый раз) должно быть захватывающим событием. Оно должно сопровождаться аплодисментами и брызгами шампанского, но чаще всего его спутники — пот, проклятия и бессонные ночи. Я видел предостаточно случаев, когда сайты запускались в три часа ночи раздраженной и уставшей командой. К счастью, ситуация меняется, отчасти благодаря облачному развертыванию.

Неважно, какую стратегию развертывания вы выберете. Куда важнее начать развертывание как можно раньше, до того, как сайт начнет эксплуатироваться. Вам не нужно подключать домен. Если перед днем запуска вы уже несколько раз разворачивали сайт на рабочем сервере, шансы на успешное развертывание увеличиваются в разы. В идеале сайт должен функционировать на рабочем сервере задолго до запуска. Все, что вам нужно будет сделать в этом случае, — переключиться со старого сайта на новый.

22 Поддержка

Вы запустили сайт! Поздравляю, больше не нужно думать о нем. Что-что? Вы по-прежнему *продолжаете* это делать? Ну, в таком случае читайте книгу дальше.

Я могу вспомнить лишь пару эпизодов в своей карьере, и то они были скорее исключениями, когда, закончив разработку сайта, я больше к нему даже не прикасался (если такое и случалось, то происходило это, как правило, потому, что кто-то другой делал данную работу, а не из-за того, что в этой работе не было необходимости). Запуск сайта — это скорее жизнь, чем смерть. После того как сайт запущен, вы прикованы к аналитике, с нетерпением ждете реакции клиента, просыпаетесь в три часа ночи, чтобы проверить, что ваш сайт по-прежнему работает. Это ваше детище, ваш ребенок.

Замысел сайта, дизайн сайта, создание сайта. Все это можно планировать и воплощать в жизнь до смерти. Но вот что обычно делается быстро и окончательно, так это *планирование поддержки* сайта. В этой главе приведены некоторые советы по этой части.

Принципы поддержки

Имейте многолетний план

Меня всегда удивляет, когда клиент соглашается с ценой создания сайта, но при этом не обсуждает момент, как долго этот сайт будет существовать. Мой опыт подсказывает, что, если вы хорошо делаете работу, клиенты охотно за нее платят. А вот что клиенты не ценят, так это неприятные сюрпризы. Например, когда им через три года сообщают, что сайт нужно переделать, а они надеялись, что он просуществует в нынешнем виде лет пять.

Интернет развивается быстро. Даже если вы создали сайт с использованием передовых технологий, какие только можно найти, уже через два года он будет выглядеть как скрипучий реликт. Конечно, он может протянуть и лет семь, устаревая постепенно, но элегантно (и это встречается куда реже!).

Прогнозировать время жизни сайта — это и искусство, и умение продавать, и наука. Последнее подразумевает способность выполнять то, что умеют все ученые, но не в состоянии сделать большинство веб-разработчиков — вести записи. Представьте, что вы ведете учет каждого сайта, который ваша команда когда-либо запускала: здесь история запросов в поддержку, перечень всех неудач, список использовавшихся технологий, фиксация затраченного времени. На это все, конечно, влияют многие факторы, начиная от принимавших участие в разработке членов команды и заканчивая экономической ситуацией и постоянно меняющимися технологиями. Однако из этого вовсе не следует, что на основе таких данных нельзя выявить важные тенденции. Вы можете обнаружить, что для вашей команды лучше работают определенные подходы в разработке или какие-то конкретные платформы и технологии. Я могу с уверенностью гарантировать, что вы обнаружите корреляцию между прокрастинацией и дефектами: чем дольше вы откладываете обновление проблемной инфраструктуры или платформы, тем хуже для вас. Наличие хорошей системы учета ошибок и сохранение подробных записей позволит вам представить клиенту более совершенную (более реалистичную) картину того, каким может быть жизненный цикл вашего проекта.

Умение продавать, конечно, сводится к деньгам. Если клиент может себе позволить полное переписывание сайта каждые три года, то вряд ли он будет страдать от устаревания инфраструктуры (хотя могут возникнуть и другие проблемы). В то же время есть клиенты, которые стремятся каждый доллар тратить лишь на самое необходимое. Они хотят, чтобы сайт жил пять или семь лет (знаю сайты, которые протянули еще дольше, но полагаю, что семь лет — это максимальное реалистичное ожидаемое время жизни сайта, у которого есть хоть какая-то надежда быть полезным в течение этого времени). Вы несете ответственность перед обоими типами клиентов, ведь они пришли к вам со своими проблемами. С клиентов, у которых много денег, не берите плату только за то, что они у них есть: используйте дополнительные средства, чтобы сделать для них что-то экстраординарное. Клиентам, у которых жесткий бюджет, вы должны предложить творческий способ разработки их сайта, чтобы увеличить продолжительность его жизни, учитывая постоянно меняющиеся технологии. Обе крайности чреваты проблемами, но их можно решить. Важно то, что вы знаете, чего хотят клиенты.

Наконец, есть искусство постановки вопроса. Это то, что помогает понять, как много клиент способен себе позволить и где вы можете честно убедить его потратить больше денег, дабы он получил то, что ему нужно. Важны также искусство понимания технологий будущего и умение предсказать, какие из них полностью устареют через пять лет, а позиции каких останутся актуальны.

Конечно, нет способа предсказать что-либо с абсолютной уверенностью. Вы можете сделать ставку не на ту технологию, кадровые перестановки способны полностью поменять техническую культуру вашей организации, а поставщики технологий могут уйти из бизнеса (хотя, как правило, в мире с открытым исходным кодом это не такая уж и большая проблема). Технология, которая, как вы думали, будет надежной в течение всего жизненного цикла вашего продукта, может оказаться

преходящей, и реконструкцию придется провести раньше, чем планировалось. В то же время иногда идеальная команда приходит в нужное время с полностью подходящей технологией и создается то, что переживет все допустимые ожидания. Однако ничто из этой неопределенности не должно удерживать вас от создания плана: лучше, когда есть хоть какой-то план, чем плыть без руля и ветрил.

Вы наверняка поняли, что, на мой взгляд, JavaScript и Node — технологии, которые будут актуальны еще долгое время. Сообщество Node живое. Оно полно энтузиазма и основывается на языке, который *победил*. Самое главное, пожалуй, что JavaScript — это мультипарадигмальный язык. Объектно-ориентированный, функциональный, процедурный, асинхронный — все это в нем есть и делает JavaScript привлекательной платформой для разработчиков из самых разных областей, что в значительной степени отвечает за темпы инноваций в экосистеме JavaScript.

Используйте контроль версий

Наверное, это очевидные вещи, но речь не о том, чтобы просто *использовать* контроль версий, а еще и о том, чтобы использовать его *хорошо*. Почему вы применяете контроль версий? Поймите причины и убедитесь, что ваш инструментарий хорошо их поддерживает. Есть множество поводов использовать контроль версий, но наиболее важный, по моему мнению, — это атрибуция, точное знание того, какое изменение, когда и кем сделано, чтобы при необходимости можно было получить дополнительную информацию. Контроль версий — это один из важнейших инструментов для понимания истории наших проектов и того, как мы работаем вместе как команда.

Используйте систему отслеживания ошибок

Системы отслеживания ошибок берут начало в искусстве разработки. Невозможно полностью понять проект без систематического фиксирования его истории. Вы наверняка слышали одно из определений безумия: «делание одного и того же снова и снова в ожидании разных результатов» (эти слова часто приписываются Альберту Эйнштейну, что сомнительно). Действительно, это кажется сумасшествием — повторять ошибки снова и снова, но как вы можете их избежать, если не знаете, где и как ошибаетесь?

Записывайте все: каждый дефект из отчета клиента, каждую неточность, которую вы обнаружили до того, как ее увидел клиент, каждую жалобу, каждый вопрос, каждую небольшую похвалу. Фиксируйте, сколько времени заняло исправление, кто исправлял, какие коммиты Git использовались, кто одобрил исправление. Искусство здесь в том, чтобы найти инструменты, позволяющие облегчить рабочий процесс. Плохая система отслеживания ошибок может оказаться утомительной и непродуктивной, что еще хуже, чем бесполезная система. Хорошая система отслеживания ошибок подбросит новые идеи для вашего бизнеса, поможет команде и клиентам.

Соблюдайте гигиену

Я не имею в виду чистку зубов, хотя и об этом не стоит забывать. Речь о контроле версий, тестировании, анализе кода и отслеживании ошибок. Инструменты, которые вы применяете, полезны только тогда, когда используются корректно. Анализ кода может быть отличным способом поддержания гигиены, поскольку способен касаться *всего*, начиная с обсуждения использования системы отслеживания ошибок, в которой появился запрос, и заканчивая тестами, необходимыми для проверки исправлений, и комментариями в коммите контроля версий.

Данные, которые вы получаете от вашей системы отслеживания ошибок, должны периодически просматриваться и обсуждаться с командой. Исходя из них, вы можете получить представление о том, что работает, а что — нет. Вы удивитесь тому, что обнаружите.

Не откладывайте

Бороться с институциональной прокрастинацией очень тяжело. Обычно кажется, что в этом нет ничего плохого: вы замечаете, что ваша команда тратит уйму времени на недельные обновления, которые можно значительно улучшить за счет небольшого рефакторинга. Каждую неделю вы откладываете рефакторинг и, таким образом, еженедельно платите цену неэффективности[1]. Хуже того, эта цена может расти со временем.

Отличный пример — неуспех в обновлении зависимостей программного обеспечения. Поскольку ПО стареет, а члены команды меняются, сложно найти людей, которые помнят (или хотя бы понимают) старое ПО. Сообщество поддержки начинает испаряться, и это приводит к тому, что технология устаревает, а вы не можете получить поддержку. Такую ситуацию часто описывают как *технический долг*, и это реальная вещь. Вы должны избегать любых проволочек, однако понимание того, сколько проживет сайт, может повлиять на решение об устранении технического долга, возникшего в текущей версии: если вы собираетесь переписать весь сайт, вряд ли есть необходимость делать это.

Регулярно контролируйте качество

Для каждого из сайтов вы должны вести *документированный* регулярный контроль качества. Он должен включать проверку ссылок, валидацию HTML и CSS, а также прогон тестов. Ключевое слово здесь — «документированный». Если элементы, составляющие контроль качества, не документированы, вы непременно что-то упустите. Документированный список проверок для каждого сайта не только поможет предотвратить пропуск проверок, но и снизит порог вхождения в проект для новых членов команды. Проверку качества может выполнить и не-

[1] Эмпирическое правило Майка Уилсона из компании Fuel: «Если вы что-то делаете третий раз, найдите время автоматизировать это».

технический специалист из команды. Это не только придаст вашему (возможно) нетехническому менеджеру уверенности в достойной работе команды, но и позволит вам распределить ответственность за контроль качества, если у вас нет специального отдела по контролю качества. В зависимости от ваших отношений с клиентом вы можете поделиться с ним своим списком проверок контроля качества (или его частью) — это хороший способ напомнить ему, за что он платит деньги и что вы работаете в его интересах.

Для регулярной проверки качества я рекомендую использовать Google Webmaster Tools (http://bit.ly/2qH3Y7L) и Bing Webmaster Tools (https://binged.it/2qPwF2c). Их просто установить, и они покажут вам, как ваш сайт видят крупные поисковые системы. Они также предупредят вас о любых проблемах с файлом `robots.txt` и HTML, влияющих на хорошие результаты поиска, проблемах с безопасностью и пр.

Отслеживайте аналитику

Если вы не используете аналитику на своем сайте, вам нужно начать делать это прямо сейчас. Это позволит оценить популярность сайта, а также увидеть, как пользователи с ним работают. Google Analytics (GA) — отличный (и бесплатный!) сервис, и даже если вы используете другие сервисы для отслеживания аналитики, вряд ли есть причина не включать GA в ваш сайт.

Наблюдая за аналитикой, вы можете выявить тонкие проблемы с пользовательским взаимодействием. Есть какие-то страницы, не получающие столько трафика, сколько вы ожидали? Это может указывать на проблему с навигацией или акциями, а также на недоработки с сфере SEO. У вас большие показатели отказов? Это может говорить о том, что контент на страницах нуждается в улучшении (люди заходят на сайт через поиск, но когда заходят — видят, что это не то, что они искали). Ваш список проверок GA должен идти в ногу со списком проверок контроля качества или даже быть частью последнего. Это должен быть живой документ — в течение жизни вашего сайта вы или ваш клиент можете менять приоритеты, определяя то, что наиболее важно.

Оптимизируйте производительность

Исследования одно за другим показывают, что производительность сайта сильно влияет на трафик. Мир быстро меняется, и люди хотят получать контент максимально оперативно, особенно на мобильных платформах. Принцип номер один в настройке производительности таков: *сначала профилирование, затем оптимизация*. Под профилированием понимается определение того, что на самом деле замедляет работу вашего сайта. Если вы проводите дни, ускоряя отображение контента, тогда как реальная проблема в плагинах социальных медиа, то вы понапрасну тратите время и деньги.

Google PageSpeed Insights (http://bit.ly/2Qa3l15) — отличный способ измерения производительности вашего сайта (сейчас данные PageSpeed записываются в Google Analytics, так что вы можете отслеживать тенденции производительности).

Это не только даст вам общую оценку мобильной и «настольной» производительности, но и составит приоритетные предложения по поводу того, как можно улучшить производительность.

Если у вас нет проблем с производительностью, то, возможно, нет необходимости в ее периодической проверке (мониторинга существенных изменений производительности посредством Google Analytics должно быть достаточно). Однако приятно смотреть на ощутимое повышение трафика, когда вы увеличили производительность.

Уделяйте первостепенное внимание отслеживанию потенциальных покупателей

В мире Интернета самый ясный сигнал, свидетельствующий об интересе посетителей к вашему продукту или услуге, — это дать вам свою контактную информацию. Вы должны отнестись к ней максимально внимательно. Любая форма, в которую вводятся адрес электронной почты или номер телефона, *всегда* должна быть протестирована в рамках списка проверок контроля качества. Кроме того, обязательно должно быть предусмотрено резервное копирование, когда вы собираете эту информацию. Худшее, что вы можете сделать по отношению к потенциальному покупателю, — это собрать контактную информацию и затем потерять ее.

Поскольку отслеживание потенциальных покупателей крайне важно для успеха вашего сайта, я рекомендую придерживаться следующих пяти принципов сбора информации.

Обеспечьте запасной вариант для случая, когда не работает JavaScript. Сбор информации о покупателях посредством Ajax неплох — он часто предлагает лучший пользовательский опыт. Однако если JavaScript по каким-то причинам не работает (пользователь отключил его, или скрипт на вашем сайте содержит ошибку, из-за которой форма Ajax работает некорректно), то отправка формы должна работать в любом случае. Лучший способ проверить это — отключить JavaScript и использовать форму. Ничего страшного, если алгоритм взаимодействия с пользователем неидеален: важно то, что данные о пользователе не утеряны. Для реализации этого у вас *всегда* должен быть действительный и работающий параметр `action` в теге `<form>`, даже если вы применяете Ajax.

Если вы используете Ajax, получайте URL из параметра action формы. Это не всегда нужно, но полезно, поскольку помогает не забыть параметр `action` в тегах `<form>`. Если вы привязываете AJAX к успешной отправке без JavaScript, потерять данные покупателя намного сложнее. Например, ваша форма может быть `<form action="/submit/email" method="POST">`; затем в коде приложения Ajax вы получите из DOM параметр `action` для формы и будете использовать его в коде сохранения данных приложения Ajax.

Обеспечьте как минимум один уровень резервного копирования. Вы, вероятно, захотите сохранить данные о потенциальных покупателях в базе данных или на внешнем сервисе, например Campaign Monitor. Но что, если ваша база данных развалится, или Campaign Monitor перестанет быть доступным, или возникнут

проблемы с сетью? Вы по-прежнему не хотите потерять этого потенциального покупателя? Распространенный способ резервного копирования — отправить письмо по электронной почте в дополнение к сохранению потенциального покупателя. Если вы решите применить этот подход, то используйте не персональный, а общий адрес электронной почты (например, dev@meadowlarktravel.com): резервное копирование будет неидеальным, если вы отправите письмо кому-то и этот человек покинет вашу компанию. Вы можете также хранить данные потенциального покупателя в резервной базе данных или даже в файле CSV. Однако в случае сбоя основного хранилища данных должен быть какой-то механизм оповещения, сообщающий вам об этом. Сбор резервных данных — это первая половина сражения, вторая половина — знание о факте сбоя и принятие соответствующих мер.

В случае сбоя системы хранения данных проинформируйте об этом пользователя. Допустим, у вас три уровня резервного копирования. Основное хранилище — Campaign Monitor, и, если оно дало сбой, вы выполняете резервирование в файл CSV и отправляете письмо на dev@meadowlarktravel.com. Когда *все* эти каналы не работают, пользователь должен получить сообщение вроде «Извините, мы сейчас испытываем технические трудности. Пожалуйста, попробуйте снова или свяжитесь с support@meadowlarktravel.com».

Проверьте положительное подтверждение, а не отсутствие ошибки. Довольно часто в случае неудачи обработчик Ajax возвращает объект со свойством `err`, тогда у кода на клиентской стороне есть что-то вроде следующего: `if(data.err){ /* проинформировать пользователя о неудаче */ } else { /* поблагодарить пользователя за успешное подтверждение формы */ }`. Избегайте такого подхода. Нет ничего плохого в установке свойства `err`, но, если в обработчике Ajax есть ошибка, ведущая к возврату сервером кода ответа 500 или ответа, не являющегося JSON, *это приведет к незаметному неудачному завершению.* Пользовательский ввод пропадет в никуда, и пользователь не поймет, что произошло. Вместо этого предоставьте свойство `success` для успешного подтверждения формы (даже если основное хранилище не работает, если пользовательская информация была записана хоть чем-то, должно быть возвращено `success`). Тогда код на клиентской стороне станет таким: `if(data.success){ /* поблагодарить пользователя за успешное подтверждение формы */ } else { /* проинформировать пользователя о неудаче \*/ }`.

Предотвратите незаметные случаи неудачи

Поскольку разработчики всегда торопятся, ошибки записываются такими способами, что они никогда не будут проверяться. Это может быть лог, таблица в базе данных, лог консоли на стороне клиента или почтовое сообщение, которое приходит на мертвый адрес. Конечный результат один и тот же: *у вашего сайта есть проблемы с качеством, которые продолжают оставаться незамеченными.*

Защита номер один против этой проблемы — *предоставление простого стандартного метода для логирования ошибок.* Документируйте их. Не усложняйте процесс. Убедитесь, что каждый разработчик, работающий с вашим проектом, знает, как это делается. Процесс может быть сведен просто к предоставлению

функции `meadowlarkLog` (функция `log` часто используется другими пакетами). Неважно, что делает эта функция: сохраняет данные в базу данных, обычный файл, отправляет их по электронной почте или комбинирует эти способы. Главное, что это стандарт. Кроме того, это позволит вам улучшить механизм логирования (например, обычные файлы не столь полезны, когда вы масштабируете сервер, так что можете модифицировать функцию `meadowlarkLog`, чтобы она вместо этого сохраняла данные в базу данных). После того как механизм логирования станет рабочим и документированным и каждый в вашей команде будет знать об этом, добавьте проверку логов в свой список проверок контроля качества и обеспечьте инструкции, рассказывающие о том, как это сделать.

Повторное использование и рефакторинг кода

Одна и та же трагедия, с которой я постоянно сталкиваюсь, — это привычка изобретать велосипед. Обычно это касается каких-то мелочей, которые проще переписать, чем разыскивать в каком-то проекте, что вы делали несколько месяцев назад. Но все эти крохи со временем суммируются. Хуже того, это бросает вызов хорошему тестировщику: вы, вероятно, не собираетесь писать тесты для всех этих небольших фрагментов (а если и собираетесь, то из-за того, что не используете существующий код, потратите в два раза больше времени). Каждый фрагмент, делающий одно и то же, может содержать разные ошибки. Это плохая привычка.

Разработка с Node и Express предлагает хорошие способы борьбы с проблемой. Node принес в сферу разработки пространство имен (посредством Modules) и пакеты (посредством npm), а Express — концепцию промежуточного ПО. С этим инструментарием гораздо легче разрабатывать повторно используемый код.

Приватный реестр npm

Реестры npm — отличное место для хранения общего кода, в конце концов, это именно то, для чего npm и был разработан. В дополнение к простому хранению вы получаете контроль версий и удобный способ включения этих пакетов в другие проекты.

Однако в бочке меда есть и ложка дегтя: если вы не работаете в организации с полностью открытым исходным кодом, то можете не захотеть сделать npm-пакеты для всего повторно используемого кода. (Могут существовать и другие причины, кроме защиты интеллектуальной собственности: ваши пакеты могут быть настолько специфичными для организации или проекта, что нет смысла делать их доступными для публичного реестра.)

Один из способов обрабатывать это — использовать *приватные реестры npm*. На данный момент npm предлагает Orgs, который позволяет публиковать приватные пакеты и дает разработчикам платные логины с возможностью доступа к этим приватным пакетам. За более подробной информацией по npm Orgs обращайтесь к списку продуктов npm (https://www.npmjs.com/products).

Промежуточное ПО

Как вы увидели в этой книге, написание промежуточного ПО — это не что-то большое, страшное и сложное. Эту манипуляцию мы проделали десятки раз, и спустя какое-то время вы будете писать такое ПО, даже не задумываясь. Следующий шаг — поместить повторно используемое промежуточное ПО в пакет и затем в реестр npm.

Если вы обнаружите, что ваше промежуточное ПО слишком специфично для проекта, чтобы добавлять его в пакет для повторного употребления, рассмотрите рефакторинг промежуточного ПО и конфигурирование для более общего использования. Помните, что вы можете передать объекты конфигурации в промежуточное ПО для того, чтобы сделать его более полезным в целом ряде ситуаций. Вот перечень наиболее распространенных способов поместить промежуточное ПО в модуль Node. Предполагается, что вы используете эти модули как пакет и этот пакет называется `meadowlark-stuff`.

Модуль предоставляет функцию промежуточного ПО напрямую

Используйте этот метод, если промежуточному ПО не нужен объект конфигурации:

```
module.exports = (req, res, next) => {
  // Промежуточное ПО будет здесь... не забудьте вызвать
  // next() или next('route'), если промежуточное ПО
  // не должно быть конечной точкой.
  next()
}
```

Для использования этого промежуточного ПО:

```
const stuff = require('meadowlark-stuff')

app. use(stuff)
```

Модуль предоставляет функцию, возвращающую промежуточное ПО

Используйте этот метод, если вашему промежуточному ПО нужен объект конфигурации или другая информация:

```
module.exports = config => {
  // Обычно создают объект конфигурации,
  // если он не был передан.
  if(! config) config = {}

    return (req, res, next) => {
      // Промежуточное ПО будет здесь... не забудьте вызывать
      // next() или next('route'), если промежуточное ПО не должно
      // быть конечной точкой.
      next()
  }
}
```

Для использования этого промежуточного ПО:

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })

app. use(stuff)
```

Модуль предоставляет объект, содержащий промежуточное ПО

Используйте этот метод, если хотите предоставить несколько блоков связанного промежуточного ПО:

```
module.exports = config => {
  // Обычно создают объект конфигурации,
  // если он не был передан.
  if(! config) config = {}

  return {
    m1: (req, res, next) => {
      // Промежуточное ПО будет здесь... не забудьте вызывать
      // next() или next('route'), если промежуточное ПО не должно
      // быть конечной точкой.
      next()
    },
    m2: (req, res, next) => {
      next()
    },
  }
}
```

Для использования этого промежуточного ПО:

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })

app. use(stuff. m1)
app. use(stuff. m2)
```

Резюме

В процессе создания сайта основной упор чаще всего делается на запуск, и на то есть причина — это очень волнующий момент. Тем не менее клиент, который еще недавно был в восторге от только что созданного сайта, может быстро превратиться в недовольного покупателя, если нет качественной поддержки сайта. Подход к обслуживанию сайта с той же щепетильностью, что и к его запуску, обеспечит вам опыт, который поможет удержать клиента.

23 Дополнительные ресурсы

В этой книге приведен полный обзор процесса создания сайтов с Express. Мы изучили основы, но это по-прежнему лишь небольшая часть доступных вам пакетов, техник и фреймворков. В этой главе мы обсудим, где вы можете найти дополнительные ресурсы.

Онлайн-документация

По документации JavaScript, CSS и HTML сеть разработчиков Mozilla (Mozilla Developer Network, MDN) (https://developer.mozilla.org/) не имеет равных. Если мне нужна документация по JavaScript, я либо ищу напрямую в MDN, либо добавляю mdn в поисковый запрос. В противном случае в результатах поиска неизбежно появляется w3schools. Конечно, тот, кто управляет продвижением w3schools, — гений, но я рекомендую избегать этого сайта, поскольку документация там недостаточно хороша.

На MDN очень хорошая справочная информация об HTML, но если вы новичок в HTML5 (и даже если нет), то должны прочитать книгу «Погружение в HTML5»[1] Майка Пилгрима (http://diveintohtml5.info/). WHATWG поддерживает отличный живой стандарт спецификации HTML5 (http://developers.whatwg.org/). Обычно я обращаюсь к ней в первую очередь с вопросами по HTML, на которые сложно ответить. Наконец, официальные спецификации по HTML и CSS находятся на сайте W3C (http://www.w3.org/). Это сухие, трудные для чтения документы, но порой они оказываются единственным ресурсом для решения очень сложных проблем.

JavaScript придерживается спецификации языка ECMAScript ECMA-262 (http://bit.ly/ECMA-262_specs). Для отслеживания доступности функций JavaScript в Node (и разных браузерах) смотрите отличное руководство (http://bit.ly/36SoK53), которое ведет @kangax.

---

[1] *Pilgrim M.* Dive Into HTML5. — O'Reilly Media, 2009.

Документация Node (https://nodejs.org/en/docs) очень хороша и всеобъемлюща и должна выбираться первой в качестве авторитетной документации о модулях Node, таких как `http`, `https` и `fs`. Документация Express (https://expressjs.com/) довольно хорошая, но не столь всеобъемлющая, как того хотелось бы. Документация npm (https://docs.npmjs.com/) всеобъемлющая и полезная.

Периодические издания

Есть три бесплатных периодических издания, на которые вы обязательно должны подписаться и читать еженедельно:

- JavaScript Weekly (http://javascriptweekly.com/);
- Node Weekly (http://nodeweekly.com/);
- HTML5 Weekly (http://html5weekly.com/).

Они будут информировать вас о последних новостях и появляющихся сервисах, блогах и учебных пособиях.

Stack Overflow

Вероятно, вы уже использовали Stack Overflow (SO): с момента создания в 2008 г. он стал доминирующим онлайн-ресурсом, где можно получить ответы на вопросы по JavaScript, Node и Express (а также любой другой технологии, рассмотренной в этой книге). Stack Overflow — это сайт с вопросами и ответами, поддерживаемый сообществом и базирующийся на репутации. Модель репутации — это то, что обусловливает качество сайта и его постоянный успех. Пользователи зарабатывают репутацию благодаря тому, что за их вопросы и ответы голосуют, или за принятые ответы. Для того чтобы задать вопрос, не нужна репутация, а регистрация здесь бесплатна. Однако есть вещи, которые вы можете сделать для увеличения вероятности того, что на ваш вопрос дадут адекватный ответ, и мы их обсудим в этом разделе.

В Stack Overflow репутация — это валюта, и, хотя есть люди, которые действительно желают вам помочь, возможность повысить репутацию — это вишенка на торте, которая мотивирует дать хороший ответ. На SO есть множество действительно умных людей, и они соревнуются за предоставление первого и/или лучшего ответа на ваш вопрос (к счастью, есть серьезное препятствие для тех, кто стремится дать быстрый, но неверный ответ). Вот то, что вы можете сделать для увеличения вероятности получения правильного ответа на свой вопрос.

- *Будьте информированным пользователем SO*. Пройдите тур по SO, затем прочитайте раздел «Как задать хороший вопрос?». Если хотите, можете прочитать всю справочную документацию — вы заработаете значок!
- *Не задавайте вопросы, на которые уже дан ответ*. Проявите должное усердие и попытайтесь выяснить, не задавал ли кто-нибудь еще этот вопрос. Если вы

задаете вопрос, ответ на который просто найти на SO, ваш вопрос быстро закроют, как дубликат, и люди часто будут голосовать против вас, что отрицательно скажется на вашей репутации.

- *Не просите людей написать код за вас.* Если вы просто спросите: «Как мне сделать то-то и то-то?» — быстро увидите, что против вашего вопроса проголосовало немало людей и его закрыли. Сообщество SO ожидает, что вы приложите определенные усилия для решения проблемы, прежде чем обратитесь за помощью к SO. Опишите в своем вопросе, что вы уже пробовали сделать и почему это не работает.
- *Задавайте один вопрос за один раз.* Очень тяжело ответить на вопрос, где спрашивается: «Как сделать это, затем то, потом еще одну вещь и как лучше поступить в такой-то ситуации?» Задавать такие вопросы не рекомендуется.
- *Подготовьте очень короткий пример своей проблемы.* Я отвечаю на многие вопросы SO, но почти всегда пропускаю те, в которых содержится три страницы (или более!) кода. Вставка файла на 5000 строк в вопрос для SO — плохая практика, если вы хотите получить ответ на вопрос (но люди постоянно это делают). Таким способом вы лишь уменьшаете вероятность получения хорошего ответа. Да и сам процесс удаления лишнего может привести вас к решению проблемы (после этого вам не придется даже задавать вопрос на SO). Подготовка минимального примера полезна для отработки навыков отладки и способности мыслить критически. Она делает вас хорошим гражданином SO.
- *Изучите Markdown.* Stack Overflow использует Markdown для форматирования вопросов и ответов. У хорошо отформатированного вопроса больше шансов получить ответ, так что потратьте определенное время, чтобы выучить этот полезный и все чаще используемый язык разметки (http://bit.ly/2CB1L0a).
- *Принимайте ответы и голосуйте за них.* Если кто-то удовлетворительно отвечает на ваш вопрос, проголосуйте за ответ и примите его. Это повысит репутацию отвечающего, а репутация — это то, что движет SO. Если полезных ответов несколько, выберите тот, который считаете лучшим, но проголосуйте за остальные, которые считаете полезными.
- *Если вы решили проблему до того, как это сделал кто-то другой, ответьте на свой вопрос сами.* SO — это ресурс сообщества: если проблема есть у вас, возможно, она возникла у кого-то еще. Если вы ее решили, ответьте на свой вопрос — будьте полезными другим.

Если вам нравится помогать сообществу, попробуйте отвечать на вопросы сами: это весело, полезно и может дать больше преимуществ, чем произвольный подсчет репутации. Если у вас есть вопрос, на который вы не получили адекватного ответа в течение двух дней, можете начать *премировать* за вопрос, используя свою репутацию. Репутация будет снята с вашего счета, и она невозвращаема. Если кто-то даст удовлетворительный ответ на ваш вопрос и вы его примете, он получит премию. Но для раздачи премий у вас должно быть как минимум 50 репутационных очков.

Конечно, вы можете заработать репутацию, задавая качественные вопросы, но обычно быстрее ее получить, давая качественные ответы.

У ответов на вопросы есть еще одно преимущество — это хороший способ учиться. Я вообще чувствую, что больше учусь, отвечая на вопросы других, чем когда другие отвечают на вопросы, заданные мной. Если вы хотите действительно тщательно изучить технологию — выучите основы, а затем пытайтесь отвечать на вопросы в SO. Сначала вас будут постоянно «выбивать» другие люди, которые уже являются экспертами, но в скором времени вы обнаружите, что вы и *есть* один из этих экспертов.

Наконец, без колебаний используйте свою репутацию для дальнейшей карьеры. Хорошая репутация достойна того, чтобы добавить ее в свое резюме. Это сработало и в моем случае. Сейчас, когда я сам интервьюирую разработчиков, меня всегда впечатляет хорошая репутация в SO (хорошей я считаю репутацию свыше 3000, пятизначная репутация — это *великолепно*). Хорошая репутация говорит мне, что этот человек не просто компетентен в своей области, но еще и коммуникабельный и готов помочь.

Содействие развитию Express

Express и Connect — проекты с открытым исходным кодом, так что любой может отправить *запрос на принятие изменений* — pull requests (здесь используется жаргон GitHub для обозначения сделанных вами изменений, которые вы хотели бы включить в проект). Это не так легко: разработчики в этих проектах — профи и высшая власть. Я не отговариваю вас от участия, но предупреждаю, что вы должны приложить немало усилий, чтобы стать успешным соучастником проекта, и к этому необходимо отнестись серьезно.

По самому процессу сотрудничества есть хорошая документация на домашней странице Express (http://bit.ly/2q7WD0X). Механизм включает создание ответвления (fork) проекта в своей учетной записи GitHub, клонирование этого ветвления, внесение изменений, отправку их обратно на GitHub и создание pull request, который будет просмотрен кем-то из проекта. Если изменения незначительны или подразумевают исправление ошибок, вы можете просто отправить pull request. Если же пытаетесь сделать что-то крупное, то должны связаться с кем-нибудь из ведущих разработчиков и обсудить свое участие. Вы же не хотите потратить часы или дни на разработку сложного функционала лишь для того, чтобы впоследствии узнать, что это не вяжется с видением сопровождающего проект или уже выполняется кем-то другим?

Еще один способ внести свой вклад (косвенно) в разработку Express и Connect — публиковать пакеты npm, особенно промежуточное ПО. Публикация вашего промежуточного ПО не потребует утверждения, но это не значит, что вы можете небрежно загромождать реестр npm низкокачественным промежуточным ПО.

Планирование, тестирование, реализация и документация — и только потом ваше промежуточное ПО становится успешным.

Если вы публикуете свои пакеты, то вот минимальный перечень того, что у вас должно быть.

- *Имя пакета*. Вы можете назвать пакет как угодно, но имя должно быть таким, какое еще никто не использовал. Иногда это сложно. В пакетах npm пространство имен не ограничено учетной записью, так что за имя вы конкурируете глобально. Если вы пишете промежуточное ПО, то учтите: общепринято добавлять к вашему имени пакета префикс `connect-` или `express-`. Броские имена пакетов, не имеющие непосредственного отношения к тому, что они делают, — это прекрасно, но лучше давать пакету имя, которое намекает на его функции (лучший пример броского и при этом подходящего имени пакета — это `zombie` для эмуляции браузера, работающего без графического интерфейса).
- *Описание пакета*. Описание пакета должно быть кратким, сжатым и наглядным. Это одно из основных полей, которое индексируется, когда люди ищут пакеты, так что лучше здесь писать наглядно, а не умно (место для демонстрации ума и чувства юмора есть в вашей документации, так что не беспокойтесь).
- *Автор/соучастники*. Расскажите о себе.
- *Лицензия (-и)*. Этим часто пренебрегают, и нет ничего более ужасного, чем встретить пакет без лицензии. Такой шаг заставляет задуматься, стоит ли использовать его в своем проекте. Не делайте так. Лицензия MIT (http://bit.ly/mit_license) — самый простой выбор, если вы не хотите устанавливать ограничения на использование своего кода. Если вы хотите, чтобы исходный код был открытым и оставался таковым, другой популярный выбор — лицензия GPL (http://bit.ly/gpl_license). Мудро также было бы включить файл лицензии в корневой каталог вашего проекта (он должен начинаться с `LICENSE`). Для максимального охвата — двойная лицензия с MIT и GPL. Для примера этого в `package.json` и файлах `LICENSE` смотрите мой пакет `connect-bundle` (http://bit.ly/connect-bundle).
- *Версия*. Для работы системы версий вам необходимо вести версии пакетов. Обратите внимание, что контроль версий npm производится отдельно от номеров коммитов в репозитории: вы можете обновлять репозиторий, когда хотите, но это не повлияет на то, что увидят люди, когда будут использовать npm для установки вашего пакета. Для того чтобы изменения отражались в реестре, нужно увеличить номер версии и переопубликовать пакет.
- *Зависимости*. Вам нужно постараться, чтобы быть консервативными в отношении зависимостей в своих пакетах. Я не советую постоянно изобретать велосипед, но зависимости увеличивают размер и сложность лицензирования пакета. Как минимум вы должны убедиться в том, что перечислили только нужные зависимости.

- *Ключевые слова*. Как и описание, ключевые слова — это важные метаданные, используемые для поиска вашего пакета, так что выберите соответствующие ключевые слова.
- *Репозиторий*. У вас он должен быть. Наиболее распространенный — GitHub, но можно использовать и другой.
- *README.md*. Стандартный формат документации и для GitHub, и для npm — Markdown (http://bit.ly/33IxnwS). Это простой, вики-подобный синтаксис, который вы можете легко использовать. Качество документации жизненно важно, если вы хотите, чтобы ваш пакет использовали. Если я захожу на страницу npm и там нет документации, я обычно пропускаю это без дополнительных исследований. Как минимум вы должны описать основное применение (с примерами). Даже лучше, чтобы все опции были документированы. А если вы опишете, как запустить тесты, это будет даже больше, чем от вас ждут.

Готовы опубликовать свой пакет? Процесс публикации пакета довольно прост. Зарегистрируйте бесплатную учетную запись npm, затем выполните следующие действия.

- Наберите `npm adduser` и войдите в систему с вашими данными доступа npm.
- Наберите `npm publish` для публикации пакета.

Вот и все! Вы, вероятно, захотите создать проект с нуля и протестировать ваш пакет с использованием `npm install`.

Резюме

Я искренне надеюсь, что эта книга дала вам все нужное для того, чтобы начать работу с этим замечательным стеком технологий. Никогда раньше я не чувствовал себя таким воодушевленным новой технологией (невзирая на необычность главного героя, которым является JavaScript) и надеюсь, что мне удалось это передать. Хотя я профессионально создаю сайты много лет, чувствую, что именно благодаря Node и Express стал глубже понимать, как работает Интернет. Полагаю, что эта технология действительно углубляет понимание, а не скрывает детали и в то же время обеспечивает основу для быстрого и эффективного создания сайтов.

Если вы новичок в веб-разработке или просто в Node и Express, приветствую вас в рядах разработчиков JavaScript. С нетерпением жду встречи с вами в группах пользователей, на конференциях и, самое главное, встречи с тем, что вы создали.

Об иллюстрации на обложке

Изображенные на обложке птицы — черный жаворонок (*Melanocorypha yeltoniensis*) и белокрылый жаворонок (*Melanocorypha leucopter*). Обе птицы перелетные и часто залетают далеко от их привычных мест обитания в степях Казахстана и Центральной России. Самцы черных жаворонков не только размножаются, но и зимуют в казахских степях, в то время как самки улетают на юг. Белокрылые жаворонки, с другой стороны, в зимний период улетают далеко на запад и север от Черного моря. Но глобальный ареал обитания этих птиц еще шире: в Европе гнездится от четверти до половины общего числа белокрылых жаворонков и от 5 % до четверти общей численности черных жаворонков.

Черные жаворонки получили такое название за черное оперение, покрывающее почти все тело самцов этого вида. Самки же похожи на самцов лишь черными лапками и черными перьями на нижней стороне крыльев. Остальное тело самок покрыто светло- и темно-серыми перьями.

Белокрылые жаворонки отличаются своеобразным узором из черных, белых и бурых перьев на крыльях. Пепельно-белое брюшко дополняют серые полоски на спине. Самцы внешне отличаются от самок только рыжевато-бурой верхушкой головы.

Песня как черных, так и белокрылых жаворонков представляет собой почти непрерывный поток красивых свистов, трелей и журчания. Оба вида птиц во взрослом возрасте едят насекомых и семена, вьют гнезда на земле. Черные жаворонки были замечены в переноске в гнезда навоза для постройки стен или создания своеобразной подстилки, хотя причина такого поведения до конца не выяснена.

Многие из тех видов животных, которые изображены на обложках книг от издательства O'Reilly, находятся под угрозой исчезновения, хотя каждый из них является важной частью нашего мира.

И. Браун

Веб-разработка с применением Node и Express. Полноценное использование стека JavaScript

2-е издание

Перевел с английского *К. Синица*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Ю. Имбро*
Литературный редактор	*М. Куклис*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 21.01.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 350. Заказ 0000.