

Саймон Холмс

Стек MEAN

Mongo
Express
Angular
Node



Getting **MEAN**

*with Mongo, Express,
Angular, and Node*

SIMON HOLMES



MANNING
SHELTER ISLAND

Саймон Холмс

Стек MEAN

**Mongo,
Express,
Angular,
Node**



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

2017

ББК 32.988.02-018
УДК 004.738.5
Х72

Холмс С.

Х72 Стек MEAN. Mongo, Express, Angular, Node. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02459-4

Обычно при веб-разработке на всех уровнях стека используются разные языки программирования. База данных MongoDB, фреймворки Express и AngularJS и технология Node.js вместе образуют стек MEAN — мощную платформу, на всех уровнях которой применяется всего один язык: JavaScript. Стек MEAN привлекателен для разработчиков и бизнеса благодаря простоте и экономичности, а конечные пользователи любят MEAN-приложения за их скорость и отзывчивость.

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

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с Manning. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1617292033 англ.
978-5-496-02459-4

© 2016 by Manning Publications Co.
© Перевод на русский язык ООО Издательство «Питер», 2017
© Издание на русском языке, оформление ООО Издательство «Питер», 2017
© Серия «Библиотека программиста», 2017

Краткое содержание

Предисловие	16
Благодарности	19
Об этой книге	21
Часть I. Задаем отправную точку	25
Глава 1. Знакомство с разработкой full-stack.....	27
Глава 2. Проектируем архитектуру на основе стека MEAN	53
Часть II. Создание веб-приложения на платформе Node	83
Глава 3. Создание и настройка проекта на стеке MEAN	85
Глава 4. Создание статического сайта с помощью Node и Express.....	117
Глава 5. Создание модели данных с помощью MongoDB и Mongoose	162
Глава 6. Создание API REST: делаем базу данных MongoDB доступной приложению	208
Глава 7. Потребление API REST: использование API из Express	256
Часть III. Добавление динамической клиентской части с помощью Angular	301
Глава 8. Добавление компонентов Angular в приложение Express	303
Глава 9. Создание одностраничного приложения с помощью Angular: фундамент	343
Глава 10. Создание одностраничного приложения с помощью Angular: следующий уровень	377

Часть IV. Управление аутентификацией и пользовательскими сеансами	425
Глава 11. Аутентификация пользователей, управление сеансами и обеспечение безопасности API.....	427
Приложения	477
Приложение А. Установка стека.....	478
Приложение Б. Установка и подготовка вспомогательного программного обеспечения	482
Приложение В. Разбираемся со всеми представлениями	486

Оглавление

Предисловие	16
Благодарности	19
Об этой книге	21
Дорожная карта.....	22
Стандарты оформления кода	22
Скачивание кода	23
Автор в Интернете	23
Об иллюстрации на обложке	24
Часть I. Задаем отправную точку	25
Глава 1. Знакомство с разработкой full-stack.....	27
1.1. Зачем изучать full-stack	28
1.1.1. Краткая история веб-разработки	28
1.1.2. Тенденция к разработке full-stack	30
1.1.3. Преимущества разработки full-stack.....	31
1.1.4. Почему именно стек MEAN.....	31
1.2. Знакомимся с Node.js: веб-сервер/веб-платформа.....	32
1.2.1. JavaScript: единый язык программирования для всего стека	32
1.2.2. Быстрая, производительная и масштабируемая платформа	33
1.2.3. Использование предварительно собранных пакетов с помощью npm	37
1.3. Знакомимся с Express: фреймворк	38
1.3.1. Упрощаем настройку сервера	38
1.3.2. Маршрутизация URL для ответов	38
1.3.3. Представления: ответы HTML	39
1.3.4. Запоминаем посетителей с помощью поддержки сессий.....	39

1.4. Знакомимся с MongoDB: база данных.....	39
1.4.1. Реляционные и документоориентированные базы данных	40
1.4.2. Документы MongoDB: хранилище данных JavaScript	40
1.4.3. Больше чем документоориентированная база данных	41
1.4.4. Для чего MongoDB не подходит	41
1.4.5. Использование Mongoose для моделирования данных и не только	42
1.5. Знакомимся с AngularJS: фреймворк клиентской части	43
1.5.1. jQuery и AngularJS	43
1.5.2. Двусторонняя привязка данных: работа с данными на странице	44
1.5.3. Использование AngularJS для загрузки новых страниц	45
1.5.4. Какова обратная сторона?	46
1.6. Вспомогательные утилиты	46
1.6.1. Применение Twitter Bootstrap для пользовательского интерфейса	47
1.6.2. Управление исходным кодом с помощью Git	48
1.6.3. Хостинг с помощью Heroku	49
1.7. Соединяем все вместе в реальном примере	49
1.7.1. Знакомство с примером приложения.....	49
1.7.2. Как компоненты стека MEAN работают вместе.....	51
1.8. Резюме	51
Глава 2. Проектируем архитектуру на основе стека MEAN	53
2.1. Традиционная архитектура стека MEAN	53
2.2. Выходим за рамки одностраничных приложений	55
2.2.1. Сложности со сканированием сайтов	55
2.2.2. Веб-аналитика и история браузера.....	56
2.2.3. Скорость начальной загрузки.....	57
2.2.4. SPA или не SPA, вот в чем вопрос	58
2.3. Разработка гибкой архитектуры MEAN.....	59
2.3.1. Требования к движку блога	59
2.3.2. Архитектура движка блога.....	61
2.3.3. Рекомендуемое решение: создание внутреннего API для слоя данных	64
2.4. Планирование реального приложения	66
2.4.1. Планирование приложения на высоком уровне.....	66
2.4.2. Проектирование архитектуры приложения	68
2.4.3. Конечный продукт	71
2.5. Разбиваем разработку на этапы	72
2.5.1. Этапы ускоренной разработки прототипа	72
2.5.2. Шаги создания Local	74

2.6. Аппаратная архитектура	79
2.6.1. Аппаратные средства для разработки приложений	79
2.6.2. Аппаратное обеспечение для промышленной эксплуатации	80
2.7. Резюме	82
Часть II. Создание веб-приложения на платформе Node	83
Глава 3. Создание и настройка проекта на стеке MEAN	85
3.1. Краткий обзор Express, Node и npm	87
3.1.1. Описание пакетов с помощью package.json	87
3.1.2. Установка зависимостей Node с помощью npm	89
3.2. Создание проекта Express	90
3.2.1. Установка отдельных частей	91
3.2.2. Проверка установленных пакетов	91
3.2.3. Создание каталога проекта	92
3.2.4. Конфигурирование установки Express	92
3.2.5. Создание проекта Express и его проверка	94
3.2.6. Перезапуск приложения	98
3.2.7. Автоматический перезапуск приложения с помощью NODEMON	98
3.3. Модифицируем Express для MVC	99
3.3.1. Обзор MVC с высоты птичьего полета	99
3.3.2. Изменение структуры каталогов	100
3.3.3. Новые каталоги views и routes	101
3.3.4. Отделение контроллеров от маршрутов	102
3.4. Импорт Bootstrap для скорости и адаптивности макетов	105
3.4.1. Скачивание Bootstrap и добавление его в приложение	106
3.4.2. Использование Bootstrap в приложении	107
3.5. Выводим приложение в Интернет с помощью Heroku	110
3.5.1. Установка и настройка Heroku	110
3.5.2. Запускаем сайт в Интернет с помощью Git	112
3.6. Резюме	116
Глава 4. Создание статического сайта с помощью Node и Express	117
4.1. Описание маршрутов в Express	119
4.1.1. Различные файлы контроллеров для разных логических наборов	120
4.1.2. Запрос файлов контроллера	121
4.1.3. Настройка маршрутов	121
4.2. Создание основных контроллеров	122
4.2.1. Настройка контроллеров	122
4.2.2. Тестирование контроллеров и маршрутов	123

4.3. Создание представлений.....	125
4.3.1. Краткий обзор Bootstrap	125
4.3.2. Настройка каркаса HTML с помощью шаблонов Jade и Bootstrap	127
4.3.3. Создание шаблона.....	132
4.4. Добавление остальных представлений.....	137
4.4.1. Страница Details (Подробности).....	138
4.4.2. Создаем страницу Add Review (Добавление отзыва)	141
4.4.3. Страница About (О нас)	143
4.5. Извлечение данных из представлений и их интеллектуализация.....	146
4.5.1. Как переместить данные из представления в контроллер	147
4.5.2. Работаем со сложными повторяющимися данными	150
4.5.3. Манипуляции данными и представлениями с помощью кода	154
4.5.4. Инструкции include и mixin для создания пригодных для повторного использования компонентов макета.....	155
4.5.5. Законченная домашняя страница	157
4.5.6. Модификация остальных представлений и контроллеров	159
4.6. Резюме	161
Глава 5. Создание модели данных с помощью MongoDB и Mongoose.....	162
5.1. Подключение приложения Express к MongoDB с помощью Mongoose	165
5.1.1. Добавление Mongoose в приложение	166
5.1.2. Добавление в приложение соединения с Mongoose.....	166
5.2. Зачем нужно моделировать данные	174
5.2.1. Что такое Mongoose и как он работает	176
5.2.2. Как Mongoose моделирует данные.....	177
5.2.3. Анализ пути схемы	177
5.3. Описание простых схем Mongoose	179
5.3.1. Основы настройки схемы.....	179
5.3.2. Использование географических данных в MongoDB и Mongoose	182
5.3.3. Создание более сложных схем с поддокументами	183
5.3.4. Окончательная схема	189
5.3.5. Компиляция схем MongoDB в модели	191
5.4. Использование командной оболочки MongoDB для создания базы данных MongoDB и добавления данных.....	192
5.4.1. Основы командной оболочки MongoDB	193
5.4.2. Создание базы данных MongoDB.....	194
5.5. Введение базы данных в промышленную эксплуатацию	198
5.5.1. Настройка MongoLab и получение URI базы данных	198
5.5.2. Помещение данных в базу данных	201
5.5.3. Заставляем приложение использовать правильную базу данных	203
5.6. Резюме	207

Глава 6. Создание API REST: делаем базу данных MongoDB доступной приложению	208
6.1. Правила API REST	209
6.1.1. URL запросов	210
6.1.2. Методы запроса	211
6.1.3. Ответы и коды состояния	214
6.2. Настройка API в Express	216
6.2.1. Создание маршрутов	217
6.2.2. Создание заглушек для контроллеров	220
6.2.3. Возврат JSON из запроса Express	220
6.2.4. Включение модели	221
6.2.5. Тестирование API	222
6.3. Методы GET: чтение данных из MongoDB	224
6.3.1. Поиск отдельного документа в MongoDB с помощью Mongoose	224
6.3.2. Поиск отдельного поддокумента на основе идентификаторов	229
6.3.3. Поиск нескольких документов с помощью геопространственных запросов	232
6.4. Методы POST: добавление данных в MongoDB	240
6.4.1. Создание новых документов в MongoDB	241
6.4.2. Проверка данных с помощью Mongoose	242
6.4.3. Создание новых поддокументов в MongoDB	243
6.5. Методы PUT: обновление данных в MongoDB	247
6.5.1. Использование Mongoose для обновления документа в MongoDB	247
6.5.2. Метод Mongoose save	248
6.5.3. Обновление существующего поддокумента в MongoDB	250
6.6. Метод DELETE: удаление данных из MongoDB	252
6.6.1. Удаление документов из MongoDB	252
6.6.2. Удаление поддокумента из MongoDB	253
6.7. Резюме	255
Глава 7. Потребление API REST: использование API из Express	256
7.1. Обращение к API из Express	257
7.1.1. Добавление в наш проект модуля request	258
7.1.2. Настройка параметров по умолчанию	258
7.1.3. Модуль request	259
7.2. Использование списков данных из API: домашняя страница Loc8r	261
7.2.1. Разделение обязанностей: перенос визуализации в поименованную функцию	261
7.2.2. Создание запроса API	262
7.2.3. Использование данных ответа API	263

7.2.4. Изменение данных перед их отображением: приводим в порядок расстояния.....	265
7.2.5. Перехват возвращаемых API ошибок	267
7.3. Получение от API отдельных документов: страница Details приложения Loc8r.....	273
7.3.1. Настройка URL и маршрутов для обращения к конкретным документам MongoDB	274
7.3.2. Разделение обязанностей: перемещаем визуализацию в поименованную функцию	275
7.3.3. Выполнение запросов к API с использованием уникального идентификатора из параметра URL.....	276
7.3.4. Передаем данные из API в представление	277
7.3.5. Отладка и исправление ошибок с представлением	278
7.3.6. Создание страниц ошибок в зависимости от статуса	281
7.4. Добавление данных в БД через API: добавляем отзывы Loc8r	284
7.4.1. Настройка маршрутизации и представлений	285
7.4.2. Отправка данных отзывов API методом POST.....	290
7.5. Защита целостности данных с помощью их проверок	292
7.5.1. Проверка на уровне схемы с помощью Mongoose	293
7.5.2. Проверка на уровне приложения с помощью Node и Express.....	297
7.5.3. Проверка в браузере с помощью jQuery	298
7.6. Резюме	300

Часть III. Добавление динамической клиентской части с помощью Angular

Глава 8. Добавление компонентов Angular в приложение Express.....	303
8.1. Настройка и запуск Angular	304
8.1.1. Открываем для себя двустороннюю привязку данных	305
8.1.2. Готовим все для настоящих достижений (и кода JavaScript)	308
8.2. Отображение и фильтрация списка для домашней страницы.....	312
8.2.1. Добавление Angular в приложение Express	312
8.2.2. Перемещение выдачи данных из Express в Angular	313
8.2.3. Фильтры Angular для форматирования данных.....	318
8.2.4. Директивы Angular для создания HTML-сниппетов.....	322
8.3. Получение данных из API	328
8.3.1. Сервисы для данных	328
8.3.2. Выполнение HTTP-запросов к API из Angular	330
8.3.3. Добавляем HTML-геолокацию для поиска местоположений, находящихся рядом с вами	333
8.4. Обеспечиваем надлежащую работу форм	340
8.5. Резюме	341

Глава 9. Создание одностраничного приложения с помощью Angular: фундамент	343
9.1. Подготовительные работы для Angular SPA	344
9.1.1. Создание каталога app_client для приложения на стороне клиента	345
9.1.2. Создание основного файла приложения SPA	345
9.1.3. Добавление основного файла приложения SPA в макет Jade	346
9.2. Переключение с Express-маршрутизации на Angular-маршрутизацию	346
9.2.1. Отключение маршрутизации Express	348
9.2.2. Добавляем в приложение ngRoute (angular-route)	350
9.3. Добавление первых представлений, контроллеров и сервисов	352
9.3.1. Создаем представление Angular	353
9.3.2. Добавляем контроллер к маршруту	355
9.3.3. Рекомендуемое решение для контроллера: использование синтаксиса controllerAs	358
9.3.4. Сервисы	361
9.3.5. Использование фильтров и директив	365
9.4. Улучшаем производительность браузера	369
9.4.1. Обертываем каждый файл в IIFE	369
9.4.2. Внедряем зависимости вручную для защиты от сокращения	371
9.4.3. Используем UglifyJS для сокращения и конкатенации сценариев	372
9.5. Резюме	376
Глава 10. Создание одностраничного приложения с помощью Angular: следующий уровень	377
10.1. Полнофункциональное SPA: перестаем основываться на серверном приложении	378
10.1.1. Создание изолированной HTML-страницы хоста	379
10.1.2. Создаем допускающие многократное использование директивы для каркаса страницы	381
10.1.3. Удаление # из URL	387
10.2. Добавление дополнительных страниц и динамическое внедрение HTML	389
10.2.1. Добавление в SPA нового маршрута и страницы	389
10.2.2. Создание фильтра для преобразования разрывов строк	393
10.2.3. Передача HTML в привязку Angular	395
10.3. Более сложные представления и параметры маршрутизации	397
10.3.1. Развертываем фреймворк страницы	398
10.3.2. Параметры URL в контроллерах и сервисах	400
10.3.3. Создаем представление для страницы Details (Подробности)	403
10.4. Использование компонентов AngularUI для создания модального всплывающего окна	407
10.4.1. Подготовка AngularUI	408
10.4.2. Добавление и использование обработчика нажатий	409

10.4.3. Создаем модальное окно Bootstrap с помощью AngularUI	411
10.4.4. Передача данных в модальное окно	414
10.4.5. Форма для отправки отзыва.....	417
10.5. Резюме	424

Часть IV. Управление аутентификацией

и пользовательскими сеансами	425
---	------------

Глава 11. Аутентификация пользователей, управление сеансами

и обеспечение безопасности API.....	427
11.1. Подход к аутентификации в стеке MEAN	428
11.1.1. Традиционный серверный подход.....	429
11.1.2. Подход с использованием всего стека MEAN	430
11.2. Создание схемы для пользователей в MongoDB	433
11.2.1. Одностороннее шифрование паролей: хеши и соль	433
11.2.2. Создание схемы Mongoose	434
11.2.3. Задание зашифрованных путей с помощью методов Mongoose	435
11.2.4. Проверяем введенный пароль	436
11.2.5. Генерация веб-маркера JSON	437
11.3. Создание API аутентификации с помощью Passport	440
11.3.1. Установка и конфигурация Passport	441
11.3.2. Создание конечных точек API для возврата веб-маркеров JSON	444
11.4. Защита конечных точек API	449
11.4.1. Добавление промежуточного ПО аутентификации в маршруты Express.....	450
11.4.2. Использование информации из JWT в контроллере	452
11.5. Создание сервиса аутентификации Angular	455
11.5.1. Управление сеансами пользователей в Angular.....	456
11.5.2. Предоставляем пользователям возможность регистрироваться, входить в приложение и выходить из него	457
11.5.3. Использование данных из JWT в сервисе Angular	459
11.6. Создание страниц регистрации и входа в приложение.....	461
11.6.1. Создаем страницу регистрации	461
11.6.2. Создаем страницу входа	465
11.7. Работаем с аутентификацией в приложении Angular	468
11.7.1. Меняем навигацию	468
11.7.2. Добавляем данные пользователя в отзыв	471
11.8. Резюме	475

Приложения	477
Приложение А. Установка стека	478
А.1. Установка Node и npm.....	478
А.1.1. Установка Node в Windows	478
А.1.2. Установка Node в Mac OS X.....	479
А.1.3. Установка Node в Linux.....	479
А.1.4. Проверяем установку путем сверки версий	479
А.2. Глобальная установка Express	479
А.3. Установка MongoDB.....	480
А.3.1. Установка MongoDB в Windows	480
А.3.2. Установка MongoDB в Mac OS X.....	480
А.3.3. Установка MongoDB в Linux.....	480
А.3.4. Запуск MongoDB в качестве сервиса	481
А.3.5. Проверка номера версии MongoDB	481
А.4. Установка Angular	481
Приложение Б. Установка и подготовка вспомогательного программного обеспечения	482
Б.1. Twitter Bootstrap.....	482
Б.1.1. Получение темы Amelia.....	483
Б.1.2. Чистка каталогов	484
Б.2. Установка Git	484
Б.3. Установка подходящего интерфейса командной строки	484
Б.4. Настройка Heroku	485
Б.4.1. Подписка на Heroku	485
Б.4.2. Установка набора инструментов Heroku	485
Б.4.3. Вход в Heroku с помощью терминала.....	485
Приложение В. Разбираемся со всеми представлениями	486
В.1. Перемещение данных из представлений в контроллеры	486
В.1.1. Страница Details (Подробности)	486
В.1.2. Страница Add Review (Добавление отзыва)	490
В.1.3. Страница About (О нас).....	491

Предисловие

В далеком 1995 году я впервые ощутил вкус веб-разработки, слепив воедино несколько страниц простого HTML в качестве части университетской курсовой работы. Это было лишь малой частью курса обучения, представлявшего собой смесь инженерии разработки программного обеспечения с теорией и практикой коммуникации, причем довольно необычную смесь. Я выучил основы разработки программного обеспечения, проектирования баз данных (БД) и программирования. Но помимо этого я узнал о важности аудитории и конечных пользователей и о методах взаимосвязи с ними, вербальных и не только.

В 1998 году для относящейся к теории и практике коммуникации части моего диплома мне понадобилась публикация для компании по моему выбору. Я решил написать рекламный проспект школы, где в то время преподавала мама, но сделать его в виде сайта. Это была исключительно клиентская часть. К счастью, у меня не сохранилось копии того кода, так как меня бросает в дрожь при одной мысли о нем. Мы использовали HTML с фреймами, макетами таблиц, внедряемыми стилями и легким налетом JavaScript. По сегодняшним стандартам код ужасен, но на тот момент он был вполне современным и даже более того. Я оказался первым человеком в университете, представившим сайт в качестве публикации. Мне даже пришлось рассказывать преподавателям, как открыть его в их браузере с дискеты, на которой я его принес! После получения оценки я продал сайт школе, которой он был посвящен. Тогда я подумал: «Возможно, у этой вашей веб-разработки есть будущее».

В последующие годы я использовал обе части своего университетского образования, работая в качестве «интернетчика» в одном из лондонских дизайнерских агентств. Поскольку это было дизайнерское агентство, взаимодействие пользователя с клиентской частью приложения играло главную роль. Но, разумеется, име-

лась и серверная часть, обеспечивавшая работу клиентской. Как единственный специалист, я совмещал обе роли классического full-stack-разработчика. В те времена практически не было распределения обязанностей. БД была тесно сцеплена с серверной частью. Логика серверной части, разметка и логика клиентской части были тесно связаны. Основной причиной этого было представление о проекте — сайте — как о единой сущности.

Многие из рекомендуемых в этой книге решений выстраданы долгими годами поиска. То, что сначала кажется безобидным, практически наверняка более простым, а иногда даже более разумным, в дальнейшем может привести к неприятным последствиям. Не позволяйте таким случаям выбивать вас из колеи и удерживать от дальнейшего продвижения. Ошибки неизбежны, и по крайней мере в данной сфере они — замечательный способ обучения. Говорят, что учатся на своих ошибках. Это правда, но вы будете всегда на шаг впереди других, если станете учиться и на чужих ошибках.

Ландшафт веб-разработки с годами изменился, но я по-прежнему активно занимаюсь созданием — или управлением созданием — полномасштабных сайтов и приложений. Я научился ценить искусство склеивания в одно целое приложений, созданных на основе различных технологий. Это действительно искусство: знание технологий и их возможностей — лишь часть задачи.

Когда я наткнулся на **Node.js**, я **сильно увлекся им и начал активно использовать** эту идею. До того мне часто приходилось переключать контекст с одного языка программирования на другой, и возможность сосредоточиться на овладении единым языком программирования была исключительно заманчивой. Я прикинул, что при правильном использовании он значительно ускорит разработку за счет снижения переключений языкового контекста. Экспериментируя с Node, я начал создавать собственный фреймворк MVC — до того момента, как узнал о существовании Express. Express решал многие проблемы и сложные задачи, с которыми я сталкивался при первых попытках изучения Node и его использования для создания сайта или веб-приложения. Его освоение во многих отношениях не было биномом Ньютона.

Естественно, почти каждое веб-приложение использует БД. Мне не хотелось обращаться к использовавшемуся мною ранее Microsoft SQL Server из-за чрезмерной для маленьких частных проектов стоимости. Небольшое исследование вопроса навело меня на ведущую базу данных NoSQL с открытым исходным кодом — MongoDB. Она работала с JavaScript естественным образом! Я был восхищен сильнее, чем следовало бы восхищаться базой данных. Но MongoDB отличалась от всех использовавшихся мною прежде БД: MongoDB — документоориентированная БД, и это в корне отличает ее от прочих и меняет ваш подход к проектированию баз данных. Мне пришлось перестраиваться на такой стиль мышления, но постепенно я с ней разобрался.

Не хватало лишь одного кирпичика. JavaScript в браузере теперь отвечает не только за расширение функциональности, но и за ее *создание* и управление логикой приложения. Из всех доступных вариантов я сразу же выбрал AngularJS. Когда я услышал, что Валерий Карпов из MongoDB придумал термин «стек MEAN», я понял: вот оно. Я осознал, что появился стек технологий нового поколения.

Я знал, что стек MEAN окажется мощным. Я знал, что стек MEAN окажется гибким. Я знал, что стек MEAN поразит воображение разработчиков. Каждая из его отдельных технологий великолепна, а уж когда они объединяются, получается нечто исключительное. Именно так появилась эта книга. Чтобы извлечь максимум из MEAN, нужно не просто знать технологии, но и понимать, как заставить их работать сообща.

Благодарности

Я начну с людей, которые очень много значат для меня, которые вдохновляют меня на движение вперед и в конечном счете придают всему смысл. Я говорю о моей жене Салли и дочерях Эри и Бэл. Все, что я делаю, начинается с этих трех дам и заканчивается ими.

Безусловно, я благодарен коллективу издательства Manning. Я понимаю, что в нем трудятся не только те, кого я сейчас упомяну персонально, так что, если вы принимали какое-либо участие в создании моей книги, спасибо вам! Вот те, с кем я лично имел дело.

С самого начала со мной работал Робин де Йонг, поспособствовавший запуску проекта и формированию очертаний книги. Огромная благодарность Берту Бэйтсу за его проницательность и за то, что он не стеснялся критиковать мои мнения на самых первых этапах. Это были забавные разговоры.

Движущей силой книги были мои редакторы: Сьюзи Питцен, Сюзанна Клайн и Карен Миллер. И конечно, мой научный редактор Мариус Бьютак. Спасибо им всем за острые глаза, замечательные идеи и положительные отзывы.

Два человека просто поразили меня количеством приложенных усилий и вниманием к деталям. Спасибо вам, Кевин Салливан и Джоди Аллен, за литературную редактуру и корректуру, за то, что всегда держали руку на пульсе при необходимости уложиться в исключительно сжатые сроки.

Последняя по порядку, но отнюдь не по значению из коллектива Manning — Кэндис Гилхули, следившая за продажами книги и державшая меня в курсе событий.

Издательство Manning может гордиться также своей программой предварительного доступа (Manning Early Access Program, MEAP) и интернет-форумом для авторов. Комментарии, исправления, идеи и отзывы читателей предварительной

версии оказались чрезвычайно полезными для улучшения качества этой книги. Я не буду называть имена всех, кто внес в это дело свой вклад. Вы себя и так знаете — спасибо вам!

Особая благодарность за отзывы и предложения следующим рецензентам, читавшим рукопись на различных стадиях ее создания: Андреа Тароччи, Энди Найту, Блэйку Холлу, Цинтии Пеппер, Дэвиду Молину, Дэнису Ндвиге, Дэвану Пэливэлу, Дугласу Данкану, Филипу Правика, Филиппо Венери, Франческо Бианчи, Хесусу Родригесу, Мэггу Мерксу, Рамбабу Роса и Уильяму И. Уилеру. А также Стивену Дженкинсу и Дипаку Вохре за техническую корректуру, выполненную перед самым выходом издания.

И еще я благодарен Тамасу Пиросу и Мареку Карвовски за то, что они терпели меня и ночные обсуждения со мной технических вопросов. Спасибо, ребята!

Об этой книге

JavaScript достиг зрелости. Благодаря ему теперь можно создать веб-приложение от начала до конца с помощью всего одного языка программирования. Стек MEAN включает в себя лучшие в своем классе технологии в данной области. В качестве БД вы получаете MongoDB, в качестве серверного фреймворка веб-приложений — Express, в качестве клиентского фреймворка — AngularJS, а в качестве серверной платформы — Node.

Эта книга познакомит вас с каждой из этих технологий, а также с тем, как заставить их работать вместе в качестве стека. На протяжении этой книги мы создаем работающее приложение, сосредотачиваясь по очереди на каждой технологии, наблюдая, как она вписывается в общую архитектуру приложения. Так что эта книга ориентирована на практику, на то, чтобы вы освоились со всеми перечисленными технологиями и их совместным использованием.

Через всю книгу красной нитью проходит понятие «рекомендуемое решение». Издание — своеобразный трамплин для создания с помощью стека MEAN замечательных приложений, так что в нем мы концентрируемся на выработке хороших привычек, умения делать все правильно и планировать все заранее.

Эта книга не учит тому, как использовать HTML, CSS или базовый JavaScript, — предполагается, что читатель уже знаком с ними. Она включает очень краткое изложение основ CSS-фреймворка Twitter Bootstrap. Кроме того, бонусом к ней идет отличное приложение по JavaScript — его теории, рекомендуемым решениям, советам и глюкам. Не помешает заглянуть туда пораньше. Это приложение можно найти в Интернете по адресу www.manning.com/books/getting-mean-with-mongo-express-angular-and-node.

Дорожная карта

Эта книга приглашает вас в путешествие по 11 главам.

Глава 1 рассматривает преимущества изучения разработки full-stack и анализирует компоненты стека MEAN.

Глава 2 основывается на приобретенном знании компонентов и обсуждает варианты их совместного использования для создания различных вещей.

Глава 3 знакомит с Express и помогает быстро освоить создание и настройку проекта MEAN.

Глава 4 дает более глубокие познания в Express посредством построения статической версии приложения.

Глава 5 использует уже имеющиеся знания о приложении и применяет MongoDB и Mongoose для проектирования и построения требующейся нам модели данных.

Глава 6 охватывает преимущества и процессы создания API данных. Мы создадим API REST с помощью Express, MongoDB и Mongoose.

Глава 7 связывает API REST с приложением путем его получения из нашего статического приложения Express.

Глава 8 является введением в стек AngularJS: мы увидим, как использовать его в создании компонентов для существующей веб-страницы, в том числе как обращаться к API REST для получения данных.

Глава 9 излагает основы создания одностраничных приложений с помощью Angular, демонстрируя разработку удобного в сопровождении модульного масштабируемого приложения.

Глава 10 базируется на материале главы 9, в ней продолжается разработка одностраничного приложения. При этом излагаются некоторые важные концепции и повышается сложность приложения Angular.

Глава 11 затрагивает все части стека MEAN, так как связана с добавлением в приложение аутентификации, позволяющей пользователям регистрироваться и заходить в приложение.

Стандарты оформления кода

Исходный код в листингах набран моноширинным шрифтом, чтобы можно было отличить его от обычного текста.

Имена методов и функций, названия свойств, элементов JSON и атрибутов в тексте, а также имена файлов и каталогов набраны таким шрифтом.

Гиперссылки и названия элементов интерфейса обозначены таким шрифтом.

Курсивом выделены новые термины и слова, которые автор хочет акцентировать.

В некоторых случаях исходный код пришлось переформатировать, чтобы он мог поместиться на странице. Изначально код и был написан с учетом ограничений по ширине страницы, но иногда вы можете обнаружить небольшие различия в форматировании между кодом, опубликованным в книге, и тем, который предназначен для скачивания. В редких случаях, когда длинные строки нельзя было переформатировать, не исказив их смысла, листинги в книге содержат маркеры продолжения строк.

Многие листинги сопровождаются комментариями к коду, раскрывающими важные идеи. Часто встречаются пронумерованные комментарии, которые подробнее поясняются в тексте рядом с кодом.

Скачивание кода

Исходный код создаваемого на протяжении книги приложения доступен на сайте издательства Manning: <http://www.manning.com/books/getting-mean-with-mongo-express-angular-and-node>. Его можно найти также на GitHub: <https://github.com/simonholmes/getting-MEAN>.

Для каждого из этапов создания приложения имеется отдельный каталог (ветвь на GitHub), обычно по состоянию на конец главы. Каталоги (ветви) не включают каталог `nodemodules` — такова рекомендуемая практика. Для запуска приложения, находящегося в любом из каталогов, вам понадобится проинсталлировать зависимости с помощью выполнения команды `npm install` в командной строке. В книге описывается, что это такое и почему необходимо это сделать.

Автор в Интернете

Приобретение *стека MEAN* дает вам право на бесплатный доступ к частному веб-форуму издательства Manning, где вы можете оставить замечания о книге, задать технические вопросы и получить помощь автора и других пользователей. Для того чтобы получить доступ к форуму и возможность подписаться на него, перейдите в браузере по ссылке <http://www.manning.com/books/getting-mean-with-mongo-express-angular-and-node>. На этой странице приведена информация о том, когда можно попасть на форум после регистрации и какие виды помощи доступны, а также размещены правила поведения на форуме.

Обязательства издательства Manning по отношению к читателям требуют предоставления места для содержательного диалога между ними, а также между читателями и автором. Эти обязательства не подразумевают какого-либо конкретного объема участия автора, чей вклад в работу форума остается добровольным

(и неоплачиваемым). Мы советуем вам задавать автору интересные и трудные вопросы, чтобы его интерес не угас.

Указанный форум и архивы предыдущих обсуждений будут доступны на сайте издательства столько, сколько будет в продаже книга.

Об иллюстрации на обложке

Рисунок на обложке данной книги носит название «Одежда константинопольской дамы ок. 1730 г.». Эта иллюстрация взята из четырехтомной книги Томаса Джеффериса «Сборник платья различных народов, старинных и современных», изданной в Лондоне между 1757 и 1772 годами. Титульная страница сообщает, что это раскрашенные вручную гравюры на меди с усилением цветности гуммиарабиком. Томаса Джеффериса (1719–1771) называли географом Его величества Георга III. Он был английским картографом и ведущим поставщиком карт своего времени. Джефферис гравировал и печатал карты для правительства и других официальных органов и издавал множество различных коммерческих карт и атласов, особенно Северной Америки. Его работа в качестве создателя карт пробудила интерес к особенностям одежды в тех странах, которые он исследовал и картографировал. Ее особенности блестяще отражены в его сборнике.

Восхищение дальними странами и путешествия для развлечения в конце XVIII века были относительно новым явлением, и подобные сборники были популярны, так как знакомили и туристов, и кабинетных путешественников с обитателями иных стран. Разнообразие рисунков в книгах Джеффериса живо говорит о неповторимости и индивидуальности народов мира каких-то 200 лет тому назад. Манера одеваться с той поры изменилась, и разнообразие различных регионов и стран, столь богатое в то время, постепенно исчезло. Сейчас зачастую сложно отличить обитателей одного континента от обитателей другого. Возможно, если попытаться смотреть на вещи оптимистически, становится очевидно: мы обменяли культурное и видимое разнообразие на более разностороннюю личную жизнь. Или жизнь, более разностороннюю и интересную интеллектуально и технически.

В наше время, когда непросто отличить одну книгу по компьютерам от другой, Manning отмечает изобретательность и предприимчивость компьютерного бизнеса обложками книг, основанными на богатом разнообразии быта различных мест двухвековой давности.

ЧАСТЬ I

Задаем отправную точку

Разработка full-stack при правильном подходе может принести отличные плоды. В приложении бывает немало «движущихся частей», и задача разработчика состоит в том, чтобы заставить их работать согласованно. В первую очередь изучите стандартные блоки, с которыми вам придется работать, и способы их комбинирования для достижения различных результатов.

Именно этому посвящена вся часть I книги. В главе 1 мы подробнее рассмотрим преимущества разработки full-stack и изучим компоненты стека MEAN. Основываясь на знании этих компонентов, в главе 2 мы обсудим их совместное использование для создания различных приложений.

К концу главы 3 вы будете хорошо понимать возможные программные и аппаратные архитектуры, основанные на стеке MEAN, а также схему приложения, которое мы будем создавать на протяжении всей книги.

Глава 1

Знакомство с разработкой full-stack

В этой главе:

- ❑ преимущества разработки full-stack;
- ❑ обзор компонентов стека MEAN;
- ❑ то, что делает стек MEAN столь интересным;
- ❑ предварительный обзор приложения, которое мы будем создавать на протяжении всей книги.

Если мы с вами похожи, то вам, наверное, не терпится скорее углубиться в какой-нибудь код и начать что-то создавать. Но сначала воспользуемся моментом, выясним, что же такое *разработка full-stack*, и взглянем на составные части стека, чтобы убедиться, что вы понимаете их все.

При упоминании разработки full-stack я подразумеваю разработку всех частей сайта или приложения. Full-stack начинается с БД и веб-сервера в прикладной части, центральной частью становится логика приложения и управления, а окончанием — пользовательский интерфейс в клиентской части.

Стек MEAN включает четыре основные технологии:

- ❑ **MongoDB** — база данных;
- ❑ **Express** — веб-фреймворк;
- ❑ **AngularJS** — фреймворк клиентской части;
- ❑ **Node.js** — веб-сервер,

а также ряд вспомогательных.

MongoDB существует с 2007 года и активно поддерживается компанией MongoDB Inc., ранее известной как 10gen.

Express выпущен в 2009 году Т. Дж. Головайчуком и с тех пор стал самым популярным фреймворком для Node.js. Его исходный код открыт, он активно развивается и поддерживается более чем 100 участниками проекта.

AngularJS относится к ПО с открытым исходным кодом и поддерживается корпорацией Google. Он существует с 2010 года и постоянно развивается и расширяется.

Node.js был создан в 2009 году, его разработка и поддержка спонсируются Joyent. Node.js использует в качестве ядра V8 — движок JavaScript с открытым исходным кодом компании Google.

1.1. Зачем изучать full-stack

Действительно, зачем изучать full-stack? Похоже, это огромный кусок работы! Ну да, это *действительно* огромный, но весьма полезный кусок работы. И не все в стеке MEAN так сложно, как вы можете подумать.

1.1.1. Краткая история веб-разработки

В давние времена, когда Интернет был совсем юным, люди не ждали многого от сайтов. Способ представления данных не был столь важен, гораздо больше всех заботило то, что происходило за кулисами. Обычно, если вы немного знали Perl и могли скомпоновать фрагмент HTML, то уже считались веб-разработчиком.

По мере распространения использования Интернета бизнес-компании начали сильнее интересоваться тем, как они в нем представлены. В сочетании с улучшенной поддержкой браузерами каскадных таблиц стилей (Cascading Style Sheets (CSS)) и JavaScript такой интерес привел к тому, что реализация клиентской части стала более сложной. Речь больше не шла о способности компоновать HTML — приходилось тратить время на CSS и JavaScript, убеждаясь в привлекательности внешнего вида сайтов и их правильном функционировании. И все это должно было работать в различных браузерах, значительно хуже совместимых, чем сейчас.

Именно тогда стали различать разработчиков клиентской и прикладной частей. Рост этого различия с течением времени демонстрирует рис. 1.1.

В то время как разработчики прикладной части концентрируют свое внимание на закулисной механике, разработчики клиентской части сосредоточены на обеспечении пользователю приятного опыта взаимодействия с сайтом. С течением времени требования к обоим лагерям выросли, что способствовало развитию этой тенденции. Разработчикам часто приходилось выбирать себе область компетенции и концентрироваться на ней.

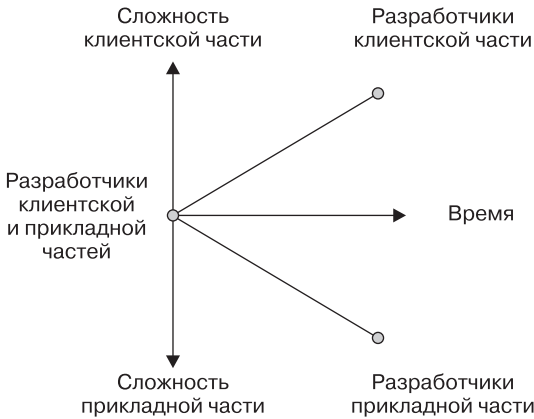


Рис. 1.1. Рост различий разработчиков клиентской и прикладной частей с течением времени

Как библиотеки и фреймворки помогают разработчикам

В 2000-е годы библиотеки и фреймворки стали популярными и широко применяются в большинстве распространенных языков программирования как на стороне клиента, так и на стороне сервера. Вспомните Dojo и jQuery для клиентского JavaScript, а также CodeIgniter для PHP и Ruby on Rails. Эти фреймворки были спроектированы для облегчения жизни разработчика, снижения входного порога. Хорошая библиотека или фреймворк уменьшают сложность разработки, ускоряя написание кода и снижая требования к наличию специальных знаний. Эта тенденция к упрощению привела к возрождению разработчиков full-stack, создававших как клиентскую часть, так и стоящую за ней логику приложения (рис. 1.2).

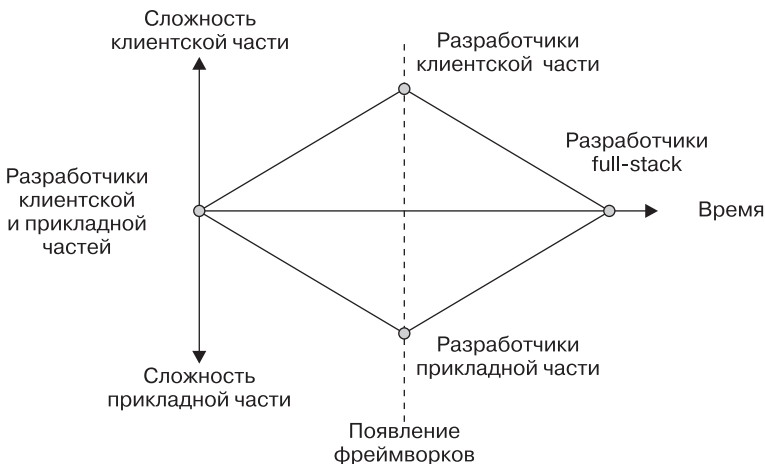


Рис. 1.2. Влияние фреймворков на отдельные блоки веб-разработки

Этот рисунок иллюстрирует скорее тенденцию, чем утверждение: «Все веб-разработчики должны быть разработчиками full-stack». Конечно, на протяжении всей истории Интернета существовали разработчики full-stack, а в перспективе, вероятнее всего, некоторые разработчики будут выбирать в качестве специализации клиентскую или серверную разработку. Я же стремлюсь показать, что благодаря использованию фреймворков и современных инструментов вам, чтобы быть хорошим веб-разработчиком, больше не надо выбирать одну из сторон.

Огромное преимущество использования подхода на основе фреймворков заключается в невероятной производительности в силу того, что можно рассмотреть приложение и его внутренние связи со всех сторон.

Продвижение кода приложения дальше по стеку

Продолжая тенденцию фреймворков, в последние несколько лет отмечаем развивающуюся тенденцию к вынесению логики приложения из серверной части в клиентскую. Можно рассматривать это как программирование прикладной части в клиентской. Наиболее популярные среди применяющих этот подход фреймворков JavaScript — AngularJS, Backbone и Ember.

Такое тесное сцепление кода приложения с клиентской частью на самом деле начинает размывать границу между теми, кто традиционно разрабатывает клиентскую и серверную части. Одна из причин популярности данного подхода — снижение благодаря этому нагрузки на сервер, а значит, снижение стоимости. Фактически при этом применяется привлечение необходимых приложению вычислительных мощностей за счет перенесения нагрузки на пользовательские браузеры.

Далее в книге я рассмотрю аргументы за и против этого подхода, включая то, когда уместно или неуместно применять одну из таких технологий.

1.1.2. Тенденция к разработке full-stack

Как уже говорилось, пути разработчиков клиентской и прикладной частей пересекаются и вполне можно быть профессионалом в обеих областях. Если вы фрилансер, консультант или трудитесь в небольшой команде, разносторонняя квалификация исключительно полезна, так как повышает вашу ценность для заказчиков. Умение разработать сайт или приложение от начала до конца позволяет вам полностью контролировать процесс и придает согласованность совместной работе различных частей, как если бы их не создавали по отдельности разные команды.

Если вы работаете в большой компании, то, вероятно, вам придется специализироваться (или по крайней мере сконцентрировать свои усилия) в одной области. Но, как правило, будет полезно знать, как созданные вами компоненты сочетаются с другими компонентами — это позволит лучше понимать требования и цели других команд и проекта в целом.

Наконец, строить приложения, основываясь на **full stack**, **весьма полезно**. В каждой его части есть свои задачи и проблемы, решение которых поддерживает интерес к разработке. Технологии и доступные на сегодняшний день инструменты усиливают эти ощущения и помогают создавать замечательные веб-приложения относительно легко и просто.

1.1.3. Преимущества разработки full-stack

Существует множество преимуществ изучения разработки full-stack. Для начинающих это, конечно, удовольствие от изучения нового и экспериментов с новыми технологиями. А также удовольствие от овладения чем-то совершенно особым и радостное возбуждение от возможности самостоятельно создать и запустить полномасштабное, ориентированное на работу с данными приложение.

Преимуществами работы в команде являются следующие:

- ❑ благодаря тому что вы понимаете, как устроены различные области и как они совместно функционируют, вы лучше видите более обширную картину;
- ❑ вы лучше понимаете, как действуют другие части команды и что им нужно для успешной работы;
- ❑ члены команды могут свободнее менять участок работы.

Дополнительные преимущества самостоятельной работы:

- ❑ вы можете самостоятельно создавать приложения от начала до конца, не завися от других людей;
- ❑ вы приобретаете больше навыков и имеете возможность предложить своим заказчикам большее число услуг.

В общем, можно многое сказать в пользу разработки full-stack. Большинство встречавшихся мне наиболее искусных разработчиков были разработчиками full-stack. Их всеобъемлющие знания и способность видеть более обширную картину давали им громадное преимущество перед другими.

1.1.4. Почему именно стек MEAN

Стек MEAN объединяет несколько лучших в своем классе современных веб-технологий в очень мощный и гибкий стек. Одна из лучших его черт: он не просто использует JavaScript в браузере — он использует JavaScript повсюду. С помощью стека MEAN вы можете программировать как клиентскую, так и серверную часть на одном и том же языке.

Основная технология, которая делает это возможным, — Node.js, внедряющая JavaScript в прикладную часть.

1.2. Знакомимся с Node.js: веб-сервер/веб-платформа

Node.js — это буква N в MEAN. То, что она стоит на последнем месте, отнюдь не значит, что она менее важна, чем прочие составляющие, — на самом деле это основа стека!

В двух словах: Node.js — программная платформа, позволяющая вам создать собственный веб-сервер и строить на нем веб-приложения. Node.js сам по себе не веб-сервер и не язык программирования. Он включает встроенную серверную библиотеку HTTP, следовательно, вам не нужно запускать отдельный веб-сервер, такой как Apache или Internet Information Services (IIS). В конечном счете вы получаете более полный контроль над работой своего веб-сервера за счет усложнения его установки и запуска, особенно в промышленной среде.

В случае PHP, например, вы можете легко найти виртуальный веб-хостинг, на котором запущен Apache, отправить туда файлы по FTP, и — вуаля! — ваш сайт работает. Это возможно, поскольку веб-хостинг заранее настроил Apache для вас и других пользователей. В случае Node.js все иначе, так как вы настраиваете сервер Node.js при создании приложения. Многие традиционные провайдеры веб-хостинга отстают в смысле поддержки Node.js, но несколько новых хостингов, работающих по принципу «платформа как услуга» (Platform as a Service (PaaS)), например Heroku, Nodejitsu и OpenShift, учитывают эту потребность. Подход к развертыванию промышленных сайтов на подобных PaaS-хостингах отличается от старой модели FTP, но если в нем разобраться, оказывается, что он довольно удобен. Мы будем развертывать сайт на Heroku в процессе продвижения по книге.

Альтернативный подход к хостингу приложения Node.js состоит в выполнении всей работы самостоятельно на выделенном сервере, на который можно установить все необходимое. Но описание администрирования промышленного сервера потребовало бы отдельной книги! Можно заменить любой из прочих компонентов альтернативной технологией, и это ни на что не повлияет, но отказ от Node.js приведет к изменению всего, что надстроено поверх него.

1.2.1. JavaScript: единый язык программирования для всего стека

Одна из главных причин популярности платформы Node.js такова: для нее программируют на знакомом большинству программистов языке — JavaScript. До сегодняшнего дня, если вы хотели быть разработчиком full-stack, вам нужно было в совершенстве знать по крайней мере два языка программирования: JavaScript для клиентской части и что-то еще, например PHP или Ruby, — для серверной.

ВТОРЖЕНИЕ MICROSOFT В СФЕРУ СЕРВЕРНОГО JAVASCRIPT

В конце 1990-х Microsoft выпустила технологию активных серверных страниц (Active Server Pages, ныне известную как классические ASP). ASP можно было писать как на VBScript, так и на JavaScript, но версия JavaScript не стала особенно популярной. В основном это произошло потому, что на тот момент многим был привычен Visual Basic, на который похож VBScript. Большинство книг и веб-ресурсов были посвящены VBScript, так что он быстро стал стандартным языком классических ASP.

С выходом Node.js вы получили возможность использовать уже имеющиеся знания при программировании на серверной стороне. Изучение языка программирования — один из самых сложных этапов изучения новой технологии, но если вы уже немного знакомы с JavaScript, то оказываетесь на шаг впереди других!

Конечно, для освоения Node.js характерна определенная кривая обучения, даже если вы уже опытный JavaScript-разработчик клиентской части. Проблемы и затруднения при серверном программировании отличаются от таковых при программировании клиентской части, но вы столкнулись бы с ними вне зависимости от используемой технологии. При разработке клиентской части вам будет важно убедиться, что все работает в множестве различных браузеров на различных устройствах. В ходе работы над серверной частью вас станут больше заботить поток выполнения кода, отсутствие задержек и экономный расход системных ресурсов.

1.2.2. Быстрая, производительная и масштабируемая платформа

Еще одна причина популярности платформы Node.js заключается в том, что при правильном программировании она работает очень быстро и чрезвычайно эффективно расходует системные ресурсы, что позволяет приложениям Node.js обслуживать больше пользователей при меньших системных ресурсах, чем большинству других широко распространенных серверных технологий. Так что владельцам компаний тоже нравится идея использования Node.js, ведь она может снизить текущие расходы даже при широкомасштабном применении.

За счет чего это происходит? Node.js использует мало системных ресурсов в силу своей однопоточности, тогда как традиционные веб-серверы многопоточны. Посмотрим, что это значит, начав с традиционного многопоточного подхода.

Традиционный многопоточный веб-сервер

Большинство современных широко распространенных веб-серверов, включая Apache и IIS, многопоточны. Это значит, что каждый новый посетитель (или сеанс) получает новый поток и связанный с этим потоком объем оперативной памяти, зачастую примерно 8 Мбайт.

Если говорить об аналогии из реального мира, представим себе двух человек, собирающихся в банк с различными целями. В многопоточной модели каждый из них пошел бы к отдельному банковскому клерку, который бы работал с их запросами, как показано на рис. 1.3.

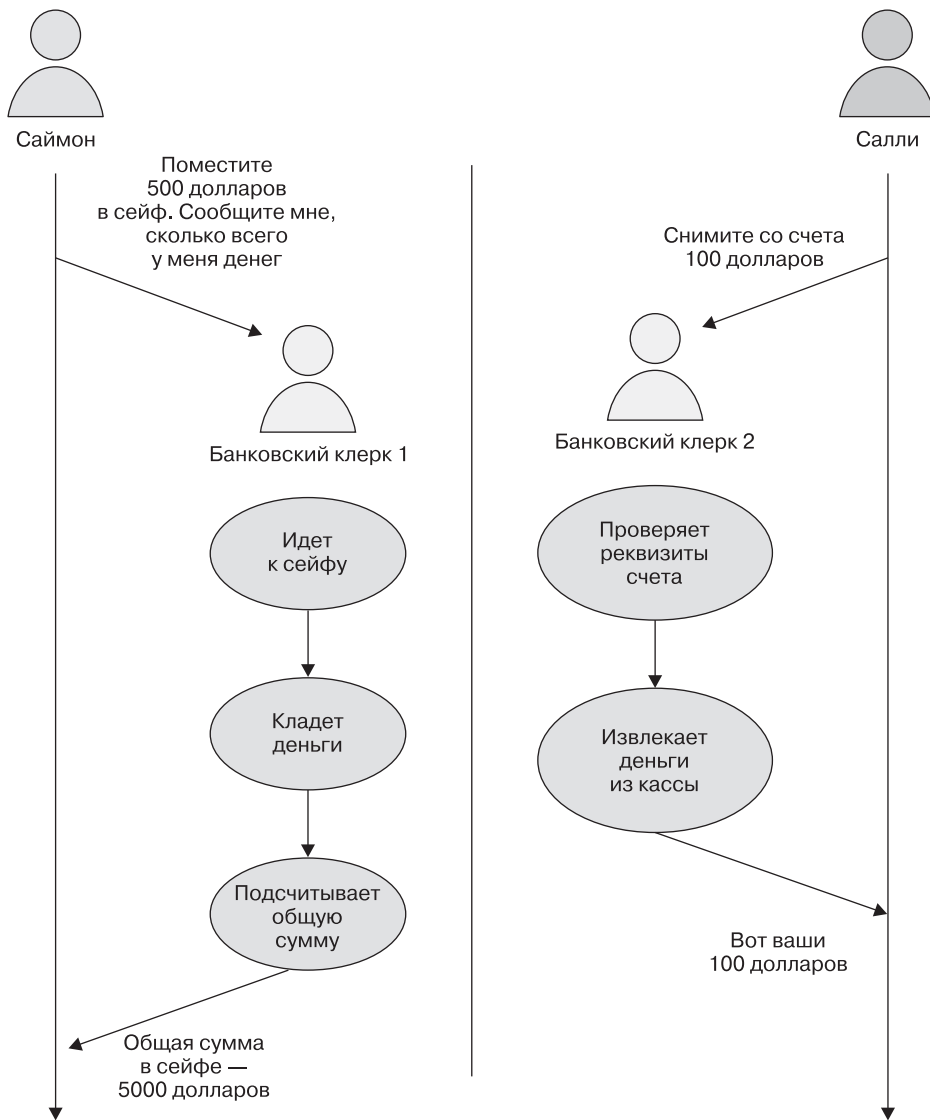


Рис. 1.3. Пример многопоточного подхода: посетители используют отдельные ресурсы. Посетители и их выделенные ресурсы не знают друг о друге и не контактируют с другими посетителями и их ресурсами

На рис. 1.3 видно, что Саймон идет к банковскому клерку 1, а Салли — к банковскому клерку 2. Никто из них не знает друг о друге и никак на другого не влияет. Банковский клерк 1 на протяжении всей транзакции работает с Саймоном и больше ни с кем, то же самое происходит у банковского клерка 2 и Салли.

Такой подход работает идеально до тех пор, пока у вас имеется достаточно банковских клерков для обслуживания клиентов. Когда банк загружен работой и клиентов становится больше, чем сотрудников, обслуживание начинает замедляться и клиентам приходится ждать вызова. Банки обычно не слишком сильно волнуются насчет этого и, похоже, рады видеть очереди, но для сайтов это не так. Если сайт возвращает вам ответ слишком медленно, вы, скорее всего, уйдете с него и никогда не вернетесь.

Именно поэтому зачастую создаются веб-серверы с большим запасом мощности и оперативной памяти, которые 90 % времени не используются полностью. Аппаратное обеспечение устанавливается с расчетом на громадный пик трафика. Это все равно как если бы банк переехал в большее здание и нанял на полный рабочий день 50 дополнительных клерков, поскольку в середине дня у них много работы.

Наверняка существует лучший способ — способ, который бы лучше масштабировался. Здесь и вступает в игру однопоточный подход.

Однопоточный веб-сервер

Сервер Node.js — однопоточный и работает не так, как многопоточный. Вместо того чтобы предоставлять каждому посетителю отдельный поток и отдельный набор ресурсов, он всех посетителей соединяет с одним и тем же потоком. Посетитель и поток взаимодействуют только при необходимости — когда посетитель что-либо запрашивает или поток отвечает на запрос.

Возвращаясь к аналогии с банковскими служащими, в данном случае будет только один клерк, работающий со всеми посетителями. Но вместо того, чтобы отходить и выполнять все запросы от начала до конца, он поручает все требующие много времени задачи персоналу резервного офиса и занимается следующим запросом. Работу такого подхода иллюстрирует рис. 1.4, используя те же два запроса из многопоточного примера.

При однопоточном подходе, показанном на рис. 1.4, Салли и Саймон подают свои запросы одному и тому же сотруднику банка. Но вместо того, чтобы целиком работать с одним из них, клерк принимает первый запрос и передает его лицу, которое лучше всего может его обработать, прежде чем принять следующий запрос и поступить аналогичным образом. Когда клерку говорят, что задание выполнено, он передает результат непосредственно запросившему его посетителю.

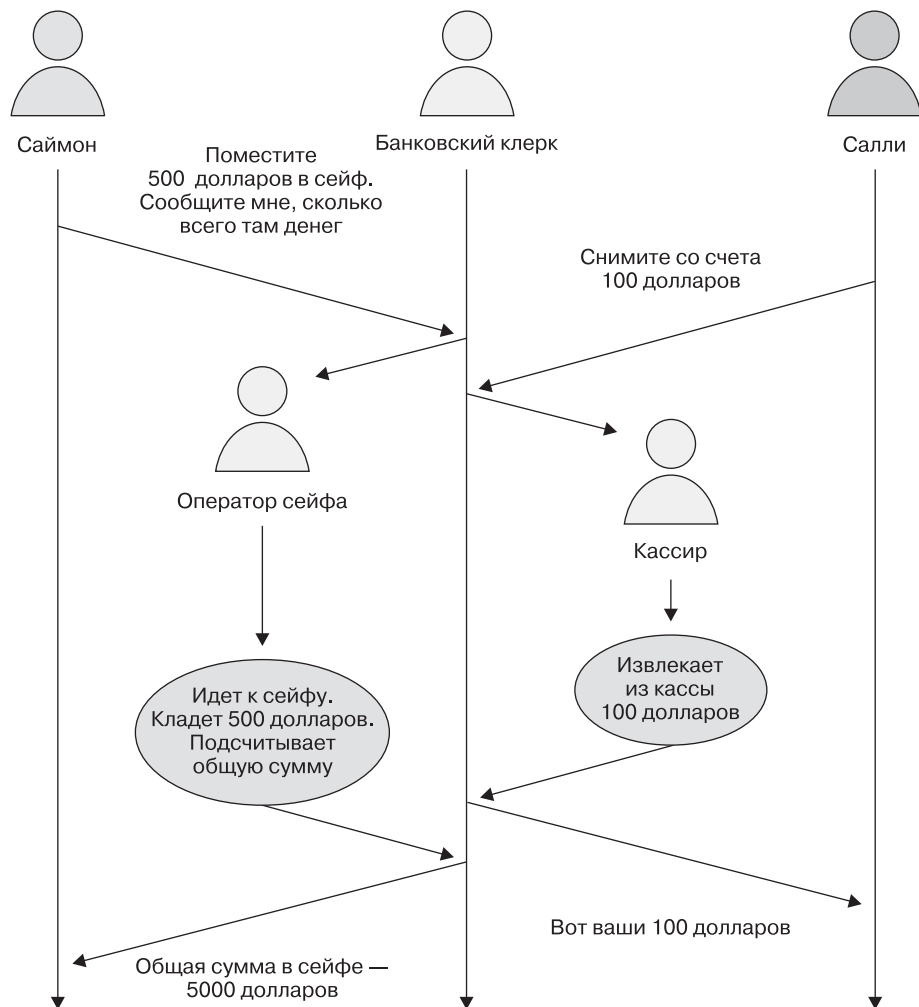


Рис. 1.4. Пример однопоточного подхода: посетители используют один основной ресурс. Он должен быть хорошо организован, чтобы один посетитель не мешал другим

БЛОКИРУЮЩИЙ И НЕБЛОКИРУЮЩИЙ КОД

При однопоточной модели важно помнить, что все ваши пользователи используют один и тот же основной процесс. Чтобы сохранить плавное течение потока выполнения, необходимо убедиться в отсутствии в коде потенциальных блокировок одной операцией другой, что может стать причиной задержки. Примером такого блокирования может послужить хождение банковского клерка к сейфу, чтобы положить туда деньги Саймона, во время которого Салли пришлось бы ждать его возвращения, чтобы подать свой запрос.

Аналогично, если ваш основной процесс отвечает за чтение всех отдельных статических файлов, таких как CSS, JavaScript и изображения, он не сможет обрабатывать никакие другие запросы, блокируя, таким образом, поток выполнения. Другая распространенная потенциально блокирующая задача — это взаимодействие с БД. Если процесс обращается к БД каждый раз, когда от него этого требуют, выполняет ли он поиск или сохранение данных, он не сможет делать ничего другого.

Поэтому, чтобы однопоточный подход работал, вы должны убедиться в том, что ваш код — неблокирующий. Добиться этого можно, сделав выполнение всех блокирующих операций асинхронным, что предотвратит блокирование ими потока выполнения основного процесса.

Несмотря на то что банковский клерк всего один, никто из посетителей не догадывается о наличии других и ни на кого из них не влияют запросы другого. Такой подход означает, что банку не требуется большое количество постоянно работающих клерков. Конечно, эта модель масштабируется не безгранично, но она более эффективна, чем многопоточная. Вы можете сделать больше, расходуя меньше ресурсов. Но это не означает, что вам никогда не понадобится добавлять ресурсы.

Данный подход оказалось возможно реализовать в Node.js благодаря асинхронным возможностям JavaScript. Вы увидите их в действии далее в этой книге, но если плаваете в теории, загляните в приложение, размещенное в Интернете по адресу <https://www.manning.com/books/getting-mean-with-mongo-express-angular-and-node>, особенно в раздел об обратных вызовах.

1.2.3. Использование предварительно собранных пакетов с помощью npm

npm — система управления пакетами, устанавливаемая вместе с Node.js. Она позволяет вам загружать модули (или пакеты) Node.js для расширения функциональности своего приложения. На момент написания книги через npm доступно более 46 000 пакетов, так что можете представить, какие глубокие знания и какой огромный опыт вы можете сделать частью приложения.

Пакеты npm сильно различаются по предлагаемым возможностям. Мы будем использовать некоторые из них на протяжении всей книги, чтобы получить инфраструктуру приложений и драйвер базы данных с поддержкой схемы. Другие примеры включают вспомогательные библиотеки, такие как *Underscore*, такие фреймворки тестирования, как *Mocha*, и прочие утилиты, например *Colors*, которая добавляет поддержку различных цветов в журнальные сообщения командной оболочки Node.js.

Мы взглянем на npm и на то, как она работает, внимательнее, когда начнем создавать приложение в главе 3.

Как вы могли видеть, Node.js исключительно мощен и гибок, но он не особенно полезен при попытке создания сайта или приложения. Для этого был создан Express. Express устанавливается посредством `npm`.

1.3. Знакомимся с Express: фреймворк

Express — это буква E в MEAN. Поскольку Node.js — платформа, она не ограничивает то, как ее следует настраивать и использовать. Это одна из ее сильных сторон. Но при создании сайтов и веб-приложений каждый раз приходится выполнять несколько довольно распространенных задач. Express — фреймворк веб-приложений для Node.js, спроектированный для выполнения этих задач проверенным и повторяющимся способом.

1.3.1. Упрощаем настройку сервера

Как уже отмечалось, Node.js — это платформа, а не сервер. Это дает вам возможность проявить при настройке сервера свои творческие способности и заставить его выполнять то, что другие веб-серверы делать не способны. Но это же делает более трудным создание и запуск простого сайта.

Express делает эти сложности несущественными благодаря настройке веб-сервера для прослушивания входящих запросов и возврата соответствующих ответов. Дополнительно он задает структуру каталогов. Один из этих каталогов настроен для обработки статистических файлов неблокирующим образом: последнее, что бы вы хотели для своего приложения, — необходимость ждать, пока кто-то еще запрашивает файл CSS! Вы можете сконфигурировать эту возможность самостоятельно в Node.js, но Express делает это за вас.

1.3.2. Маршрутизация URL для ответов

Замечательная особенность Express — по-настоящему простой интерфейс для маршрутизации входящего URL к соответствующему фрагменту кода. Неважно, будет ли этот код выдавать статическую веб-страницу, выполнять чтение из БД или записывать в нее, — интерфейс прост и единообразен.

Здесь Express фактически уменьшил сложность выполнения всего этого в Node.js, ускоряя тем самым написание кода и облегчая его поддержку.

1.3.3. Представления: ответы HTML

Вполне возможно, что вам понадобится отвечать на много запросов к вашему приложению путем отправки браузеру HTML-кода. Думаю, вас уже не удивит, что Express упрощает эту задачу по сравнению с нативным Node.js.

Express обеспечивает поддержку множества различных шаблонизаторов, упрощающих создание страниц HTML с помощью повторно используемых компонентов, а также данных из вашего приложения. Express собирает их вместе и выдает браузеру в виде HTML-кода.

1.3.4. Запоминаем посетителей с помощью поддержки сеансов

Будучи однопоточным, Node.js не запоминает посетителей между запросами. У него нет отдельной области оперативной памяти, выделенной только для вас, он просто видит ряд запросов HTTP. HTTP — протокол без сохранения состояния, так что в нем отсутствует понятие состояния сеанса. В данных условиях в Node.js сложно создать персонализированную информацию или иметь защищенную область, в которой пользователь мог бы входить на сайт, — в этом нет особого смысла, если сайт забывает, кто вы такой, на каждой следующей странице. Задать эту возможность, конечно, можно, но вам придется запрограммировать ее самостоятельно.

Вы ни за что не поверите: у Express имеется решение и этой задачи! Express позволяет использовать *сеансы*, так что вы можете идентифицировать отдельных посетителей на протяжении нескольких запросов и страниц.

Будучи надстройкой над Node.js, Express помогает вам и предоставляет надежную отправную точку для создания веб-приложений. Он снимает множество сложностей и повторяющихся задач, о которых большинство из нас не должно или не хочет беспокоиться. Мы просто хотим создавать веб-приложения.

1.4. Знакомимся с MongoDB: база данных

Способность хранить и использовать данные жизненно важна для большинства приложений. Выбранной для стека MEAN базой данных является MongoDB — буква M в MEAN. MongoDB исключительно хорошо подходит для данного стека. Подобно Node.js, она славится своей быстротой работы и масштабируемостью.

1.4.1. Реляционные и документоориентированные базы данных

Если раньше вы использовали реляционные БД или даже просто электронные таблицы, вам наверняка привычны столбцы и строки. Обычно столбец задает имя и тип данных, а каждая строка представляет собой отдельную сущность (см. пример в табл. 1.1).

Таблица 1.1. Как строки и столбцы выглядят в реляционной БД

firstName (имя)	middleName (второе имя)	lastName (фамилия)	maidenName (девичья фамилия)	Nickname (псевдоним)
Саймон	Дэвид	Холмс		Сай
Салли	Джун	Панайоту		
Ребекка		Норман	Холмс	Бек

MongoDB *не такая!* MongoDB — документоориентированная БД. Строки в ней по-прежнему присутствуют, но столбцов как таковых нет. Вместо столбцов, описывающих, что должно быть в строке, каждая строка представляет собой документ, который содержит как данные, так и их описание. Пример описания коллекции документов см. в табл. 1.2 (расположение в виде колонок выполнено для удобства чтения — это не столбцы).

Таблица 1.2. Каждый документ в документоориентированной БД содержит как данные, так и их описание и располагается без соблюдения определенного порядка

firstName: "Саймон"	middleName: "Дэвид"	lastName: "Холмс"	nickname: "Сай"
lastName: "Панайоту"	middleName: "Джун"	firstName: "Салли"	
maidenName: "Холмс"	firstName: "Ребекка"	lastName: "Норман"	nickname: "Бек"

Такой менее структурированный подход означает, что коллекции документов могут содержать самые разнообразные данные. Давайте взглянем на пример документа, чтобы вы лучше поняли, что я имею в виду.

1.4.2. Документы MongoDB: хранилище данных JavaScript

MongoDB хранит документы в виде BSON — двоичной JSON (JavaScript Serialized Object Notation — сериализованная нотация объектов JavaScript). Если вы плохо знакомы с JSON, пусть вас это пока что не беспокоит — посмотрите соответствующий раздел в приложении к книге, которое можно найти в Интернете. Если корот-

ко, JSON — применяемый в JavaScript способ хранения данных, поэтому MongoDB так хорошо вписывается в ориентированный на JavaScript стек MEAN!

Следующий фрагмент кода демонстрирует простейший пример документа MongoDB:

```
{
  "firstName" : "Саймон",
  "lastName"  : "Холмс",
  "_id"       : ObjectId("52279effc62ca8b0c1000007")
}
```

Даже если вы недостаточно хорошо знаете JSON, то все равно, вероятно, видите, что этот документ содержит мои имя и фамилию — Саймон Холмс! Так что вместо хранения набора данных, соответствующего множеству столбцов, документ хранит пары «название/значение». Поэтому документ полезен сам по себе, так как он и описывает и определяет данные.

Небольшое замечание насчет `_id`. Вы наверняка заметили запись `_id` среди названий в предыдущем примере документа MongoDB. Запись `_id` — уникальный идентификатор, который MongoDB присваивает каждому новому документу при создании.

Мы рассмотрим документы MongoDB подробнее в главе 5, когда начнем добавлять в наше приложение данные.

1.4.3. Больше чем документоориентированная база данных

MongoDB отличается от многих других документоориентированных БД поддержкой вторичной индексации и способностью работать с запросами, обладающими широкими возможностями. Это значит, что вы можете создавать индексы не только по полю уникального идентификатора, а также что запросы к проиндексированным полям выполняются намного быстрее. Вы можете также создавать весьма сложные запросы к базе данных MongoDB — не на уровне огромных команд SQL с соединениями повсюду, но с возможностями, которых достаточно для выполнения большинства пользовательских сценариев.

По мере построения приложения на протяжении данной книги мы еще поразвлекаемся со всем этим, и вы оцените возможности MongoDB.

1.4.4. Для чего MongoDB не подходит

MongoDB — не транзакционная БД и не должна использоваться в качестве таковой. Транзакционная база данных может выполнять несколько отдельных операций как одну транзакцию. Если какая-либо из операций в транзакции завершается неудачей, то вся транзакция тоже завершается неудачей и ни одна из операций не выполняется. MongoDB работает *не так*. MongoDB рассматривает каждую операцию

отдельно, и если какая-либо одна завершается неудачей, то завершается неудачей только она, а остальные операции будут выполняться.

Это имеет большое значение при обновлении нескольких коллекций документов одновременно. Например, если вы создаете корзину для виртуальных покупок, вам нужно быть уверенными, что платеж выполнен и зафиксирован, а также что заказ отмечен как подтвержденный и предназначенный для обработки. Вам вряд ли хочется учитывать вероятность того, что покупатель мог оплатить заказ, который ваша система считает все еще находящимся в корзине. Так что эти две операции должны быть связаны в одну *транзакцию*. Структура вашей БД может предоставлять возможность их выполнения в одной коллекции, или можно запрограммировать в логике приложения механизм отката и «страховочные сетки» на случай сбоя, или же можно использовать транзакционную БД.

1.4.5. Использование Mongoose для моделирования данных и не только

Гибкость MongoDB относительно хранимых в документах данных — отличное качество для БД. Но большинство приложений требуют для своих данных определенной структуры. Обратите внимание на то, что структуры требуют приложения, а не база данных. Так где же будет разумнее всего определить структуру данных вашего приложения? В самом приложении!

С этой целью компания, разработавшая MongoDB, создала Mongoose. По их собственным словам, Mongoose обеспечивает «изящное моделирование объектов MongoDB для Node.js» (<http://mongoosejs.com/>).

Что такое моделирование данных

Моделирование данных в контексте Mongoose и MongoDB — это определение того, какие данные *могут*, а какие — *должны* присутствовать в документе. При хранении информации пользователя вам может понадобиться возможность сохранить имя, фамилию, адрес электронной почты и номер телефона. Но *необходимы* вам только имя и адрес электронной почты, причем адрес электронной почты должен быть неповторяющимся. Эта информация задается в схеме, которая используется как основа для модели данных.

Какие еще возможности предоставляет Mongoose

Помимо моделирования данных, Mongoose добавляет к возможностям MongoDB множество возможностей, полезных при создании веб-приложений. Mongoose облегчает управление подключениями к базе данных MongoDB, а также сохранение и чтение данных. Позднее мы будем все это использовать. А также обсудим, как

благодаря Mongoose вы обретаете возможность добавления проверки данных на уровне схемы, гарантируя, что в БД можно будет сохранять только корректные данные.

MongoDB — замечательный вариант БД для большинства веб-приложений, поскольку она обеспечивает равновесие между скоростью чисто документоориентированных БД и возможностями реляционных БД. То, что данные фактически хранятся в JSON, делает ее идеальным хранилищем данных для стека MEAN.

1.5. Знакомимся с AngularJS: фреймворк клиентской части

AngularJS — это буква A в MEAN. Говоря упрощенно, AngularJS — это фреймворк JavaScript для работы с данными непосредственно в клиентской части.

Можно использовать Node.js, Express и MongoDB для создания полнофункционального, ориентированного на работу с данными веб-приложения. И на протяжении всей книги мы именно этим и будем заниматься. Но можно еще и положить вишенку на торт, добавив к стеку AngularJS.

Традиционно выполняют всю обработку данных и логику приложения на сервере, который затем передает HTML-код браузеру. AngularJS дает возможность перенести часть этой обработки и логики на браузер, оставляя серверу лишь передачу данных из БД. Мы рассмотрим это при обсуждении двусторонней привязки данных, но сначала нужно разобраться, похож ли AngularJS на jQuery, наиболее популярную JavaScript-библиотеку клиентской части.

1.5.1. jQuery и AngularJS

Если вы знакомы с jQuery, то, возможно, задумывались, работает ли AngularJS схожим образом. Если коротко, то нет, не совсем. jQuery, как правило, добавляется на страницу для обеспечения интерактивности уже после того, как HTML-код отправлен на браузер, а объектная модель документов (Document Object Model (DOM)) полностью загружена. AngularJS подключается на шаг раньше и помогает создавать HTML-код на основе имеющихся данных.

Кроме того, jQuery — библиотека, и в ней имеется набор возможностей, которые вы можете использовать. AngularJS — то, что называется «*упрямый*» (*opinionated*) фреймворк. Имеется в виду, что он навязывает вам свое мнение о том, как его следует использовать.

Как уже упоминалось, AngularJS помогает создавать HTML-код на основе имеющихся данных, но делает не только это. Он также немедленно обновляет

HTML-код при изменении данных, причем может и обновлять данные при изменении HTML-кода. Это называется двусторонней привязкой данных, краткий обзор которой сейчас и будет создан.

1.5.2. Двусторонняя привязка данных: работа с данными на странице

Чтобы лучше понять смысл двусторонней привязки данных, взглянем сначала на традиционный подход — одностороннюю привязку данных. Односторонняя привязка данных — это то, на что вы нацелены при изучении использования Node.js, Express и MongoDB. Node.js получает данные от MongoDB, затем Express использует шаблон для компиляции этих данных в HTML-код, который затем отправляется серверу (рис. 1.5).

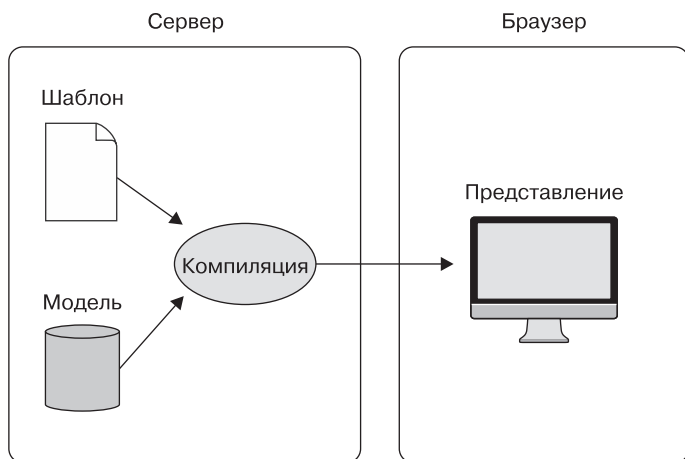


Рис. 1.5. Односторонняя привязка данных — шаблон и модель собираются вместе на сервере перед отправкой браузеру

Подобная однонаправленная модель — фундамент большинства ориентированных на работу с БД сайтов. В этой модели большая часть тяжелой работы выполняется на сервере, а браузеру остаются лишь формирование HTML и выполнение интерактивного JavaScript.

Двусторонняя привязка данных — нечто иное. Здесь шаблон и данные отправляются браузеру независимо друг от друга. Браузер сам компилирует шаблон в представление и данные — в модель. Главное отличие в том, что представление — динамическое. Представление привязано к модели, так что при изменениях модели представление тоже немедленно меняется. Двусторонняя привязка данных показана на рис. 1.6.

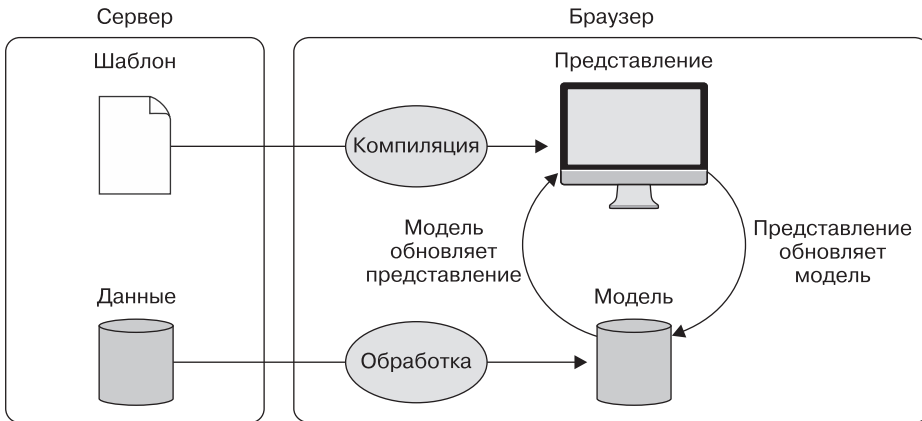


Рис. 1.6. Двусторонняя привязка данных — представление и модель обрабатываются в браузере и связываются, причем каждый немедленно обновляет другого

Во время чтения части III этой книги вы попробуете все это в действии. И вы не будете разочарованы.

1.5.3. Использование AngularJS для загрузки новых страниц

AngularJS был специально спроектирован с расчетом на функциональность, называемую *одностраничными приложениями* (single-page application (SPA)). Фактически SPA — это приложение, выполняющее все внутри браузера и никогда не производящее полной перезагрузки страницы. Это значит, что вся логика приложения, обработка данных, пользовательский поток выполнения и отправка шаблонов могут быть организованы в браузере.

Рассмотрим Gmail. Это пример SPA. На странице, помимо разнообразных наборов данных, показываются различные представления, но сама страница никогда не перезагружается.

Такой подход способен существенно снизить количество необходимых вам ресурсов сервера, причем фактически производится краудсорсинг требуемых приложению вычислительных мощностей. Браузер каждого пользователя выполняет тяжелую работу, а ваш сервер, по сути, просто выдает статические файлы и данные по запросу.

Подобный подход также улучшает механизм взаимодействия с пользователем. Как только приложение загружено, количество обращений к серверу уменьшается, что снижает вероятность задержек.

Все это звучит замечательно, но ведь наверняка за это придется заплатить? Иначе почему не все веб-приложения основаны на AngularJS?

1.5.4. Какова обратная сторона?

Несмотря на множество достоинств, AngularJS не годится для всех сайтов без исключения. Библиотеки клиентской части, такие как jQuery, лучше подходят для постепенного расширения. Идея в том, что ваш сайт будет отлично работать без JavaScript, а подключение JavaScript лишь улучшает взаимодействие с пользователем. С AngularJS и, конечно, любым другим фреймворком SPA дела обстоят иначе. AngularJS использует JavaScript для формирования визуализируемого HTML-кода из шаблонов и данных, так что, если ваш браузер не поддерживает JavaScript или в коде есть ошибка, сайт вообще не будет работать.

Такая зависимость от JavaScript при создании страницы вызывает проблемы с поисковыми системами. Когда поисковая система сканирует сайт своим поисковым агентом, она не запускает никакого JavaScript, а при использовании AngularJS единственное, что у вас имеется до выполнения JavaScript, — шаблон с сервера. Если нужно, чтобы поисковые системы индексировали ваши контент и данные, а не шаблоны, то следует подумать, подходит ли AngularJS для данного проекта.

Существуют способы справиться с этой проблемой — говоря кратко, нужно, чтобы ваш сервер выводил скомпилированное содержимое наряду с AngularJS. Но если нет *необходимости* это делать, то я рекомендовал бы так не поступать.

Самое простое, что вы можете сделать, — использовать AngularJS для одного и не использовать для другого. Нет ничего плохого в том, чтобы в своем проекте использовать AngularJS выборочно. Например, у вас могут иметься насыщенное данными интерактивное приложение или раздел сайта, для создания которых AngularJS подходит идеально. У вас также могут быть блог или какие-то связанные с вашим приложением маркетинговые страницы. Их не обязательно создавать с помощью AngularJS, и, вполне возможно, лучше будет выдавать их с сервера традиционным способом. Таким образом, часть вашего сайта выдается Node.js, Express и MongoDB, а в другой части работает AngularJS.

Такой гибкий подход — одна из самых сильных сторон стека MEAN. С помощью одного стека вы можете достичь множества различных целей.

1.6. Вспомогательные утилиты

Стек MEAN дает вам все, что нужно для создания насыщенных данными интерактивных веб-приложений, но вам могут оказаться полезными несколько дополнительных технологий. Вы можете использовать Twitter Bootstrap для создания хорошего пользовательского интерфейса, Git — для управления кодом и Heroku — для того, чтобы обеспечить размещение приложения на реальном URL. В следующих главах мы рассмотрим вопрос включения этих технологий в стек MEAN. Здесь же я просто кратко опишу, чем каждая из них может оказаться вам полезна.

1.6.1. Применение Twitter Bootstrap для пользовательского интерфейса

В книге мы будем использовать Twitter Bootstrap для создания адаптивного интерфейса с минимальными усилиями. Для стека он неважен, и если вы создаете какой-то особый проект или приложение из существующего HTML-кода, то, возможно, не захотите его включать. Но мы планируем создавать приложение по принципу ускоренной разработки программ, переходя от идеи к приложению без учета влияния внешних факторов.

Bootstrap — фреймворк клиентской части, очень полезный при создании отличного пользовательского интерфейса. Среди обеспечиваемых Bootstrap возможностей — адаптивная система сеток, стили по умолчанию для многих компонентов интерфейса и способность менять внешний вид с помощью тем.

Адаптивная система сеток

При использовании адаптивного макета вы создаете отдельную страницу HTML, которая ведет себя по-разному на различных устройствах. Это происходит скорее в результате определения разрешения экрана, чем из-за выяснения того, какое устройство используется. Bootstrap ориентируется в своих макетах на четыре контрольные точки по ширине в пикселах, приблизительно соответствующие телефонам, планшетам, ноутбукам и внешним мониторам. Так что если вы посвятите немного времени обдумыванию структуры вашего HTML-кода и классов CSS, то сможете использовать один HTML-код для выдачи одного и того же контента в различных макетах, подходящих для различных размеров экрана.

Классы CSS и компоненты HTML

Bootstrap поставляется с набором встроенных классов CSS, полезных при создании удобных визуальных компонентов. В их числе заголовки страниц, контейнеры экстренных сообщений, метки, бейджи, стилизованные списки... и этот список можно продолжить! Они многое продумали, и это действительно помогает быстро создавать приложения без лишних трат времени на макет HTML и стили CSS.

В данной книге не ставилась цель обучить вас использованию Bootstrap, но я буду обращать ваше внимание на различные возможности его применения.

Добавляем темы — совсем другие ощущения

Bootstrap по умолчанию предоставляет варианты оформления интерфейса, которые обеспечивают действительно неплохую точку отсчета. Его возможности используются настолько часто, что ваш сайт может в итоге оказаться похожим на чей-то еще. К счастью, можно скачивать темы для Bootstrap, чтобы придать вашему приложению

оригинальный вид. Скачивание темы не сложнее замены файла CSS Bootstrap новым. В этой книге для создания нашего приложения будем использовать бесплатную тему, но на множестве сайтов в Интернете можно купить превосходные темы, которые позволят вашему сайту производить неизгладимое впечатление.

1.6.2. Управление исходным кодом с помощью Git

Сохранять код на своем компьютере или сетевом диске, конечно, замечательно, но так сохраняется только последняя версия. При этом получить доступ к ней можете только вы или кто-то из вашей сети.

Git — распределенная система контроля версий и управления исходным кодом. Это значит, что несколько человек могут работать с одной базой исходных текстов одновременно с различных компьютеров и из различных сетей. Они могут работать совместно, сохраняя и записывая все изменения. Также возможно при необходимости откатиться к предыдущему состоянию.

Как использовать Git

Git обычно используют из командной строки, хотя существуют GUI для Windows и Mac. В данной книге мы будем использовать операторы командной строки для необходимых нам команд. Git обладает широкими возможностями, так что мы здесь затронем его лишь очень поверхностно, но все, что мы будем делать, будет прокомментировано.

При обычной настройке Git имеется локальный репозиторий на вашей машине и удаленный централизованный основной репозиторий, размещенный, например, на GitHub или BitBucket. Вы можете извлекать данные из удаленного репозитория в локальный или отправлять из локального в удаленный. Все это очень просто выполняется из командной строки, причем как у GitHub, так и у BitBucket имеется веб-интерфейс, так что можно наглядно отслеживать все зафиксированные изменения.

Для чего нам Git

В книге мы собираемся работать с Git по двум причинам.

Во-первых, исходный код примера приложения в этой книге будет храниться на GitHub, для различных промежуточных этапов разработки будут созданы отдельные ветви. У нас будет возможность клонировать основной репозиторий или отдельные его ветви, чтобы использовать соответствующий код.

Во-вторых, мы будем применять Git в качестве метода развертывания нашего приложения на действующий веб-сервер, чтобы его мог увидеть весь мир. Для хостинга используем Heroku.

1.6.3. Хостинг с помощью Heroku

Хостинг приложений Node.js может оказаться непростой задачей, но не обязательно. Многие провайдеры виртуального хостинга не знают об интересе публики к Node.js. Некоторые установят его для вас при необходимости, так что вы сможете запустить приложения, но в целом серверы не настроены в соответствии со специфическими потребностями Node.js. Для успешного запуска приложения Node.js вам нужен или сервер, сконфигурированный с расчетом на него, или провайдер PaaS, нацеленный именно на хостинг Node.js.

В книге выберем второй вариант. Мы собираемся использовать Heroku (<http://www.heroku.com/>) в качестве провайдера хостинга. **Heroku — один из ведущих провайдеров хостинга для приложений Node.js**, и у него имеется великолепный бесплатный пакет, которым мы воспользуемся.

Приложения на Heroku, по сути дела, представляют собой репозиторий Git, что делает процесс публикации чрезвычайно простым. Как только все настроено, вы можете опубликовать свое приложение в промышленной среде с помощью одной-единственной команды:

```
$ git push heroku master
```

Я же говорил вам, что ничего сложного тут не должно быть.

1.7. Соединяем все вместе в реальном примере

Как уже неоднократно упоминалось, в нашей книге мы будем создавать на основе стека MEAN работающее приложение. Так вы изучите основы каждой из технологий, равно как и их совместную работу.

1.7.1. Знакомство с примером приложения

Так что же мы на самом деле будем создавать в ходе чтения этой книги? Мы будем создавать приложение под названием Loc8r. Оно будет выводить список ближайших мест с Wi-Fi, куда можно пойти и поработать. Оно также отображает наличие удобств, время начала работы, рейтинг и схему проезда. Пользователи смогут войти на сайт, выставить оценки и отправить отзывы.

Данное приложение основывается на реальных примерах из практики. Сами по себе приложения, основанные на определении местоположения, не новы и существуют в нескольких различных формах. Foursquare и функция Facebook «Места поблизости» перечисляют все находящееся рядом, что только могут. UrbanSpoon помогает находить поблизости места, где можно поесть, предоставляя пользователю

возможность выбора по ценовой категории и типу кухни. Даже в приложениях компаний Starbucks и McDonald's имеются разделы, помогающие пользователям найти ближайшее заведение.

Настоящие или сфабрикованные данные?

Ладно, в этой книге мы планируем сфабриковать данные для Loc8r, но вы можете подобрать данные, собрать их методом краудсорсинга или использовать внешний источник, если захотите. При использовании подхода ускоренной разработки вы часто будете сталкиваться с тем, что фабрикация данных для первой, необщедоступной версии вашего приложения значительно ускоряет процесс.

Конечный продукт

Здесь для Loc8r мы будем использовать все слои стека MEAN, включая Twitter Bootstrap для создания адаптивного макета. Несколько скриншотов того, что мы будем создавать в этой книге, демонстрирует рис. 1.7.

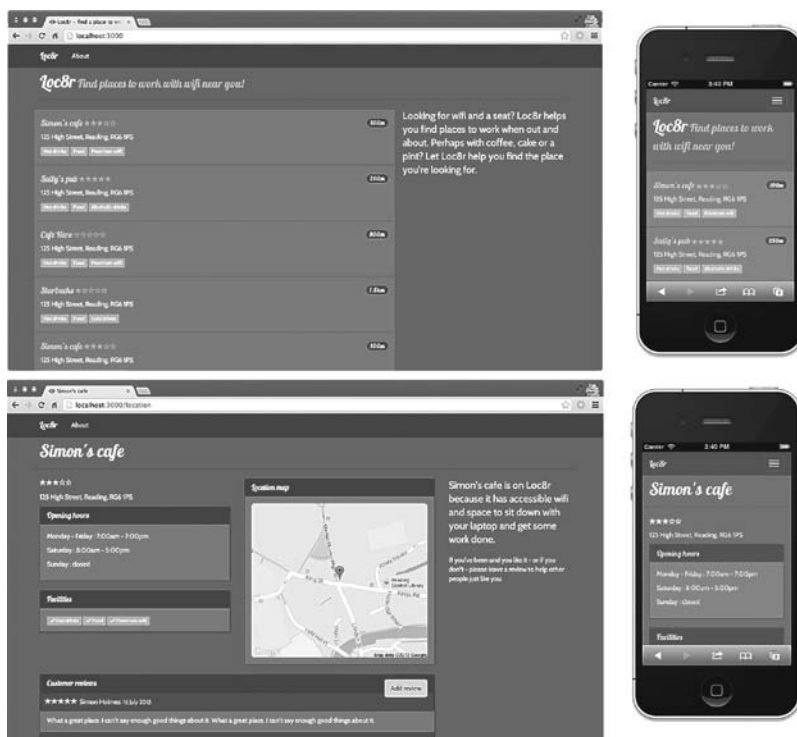


Рис. 1.7. Loc8r — приложение, которое мы будем создавать на протяжении всей книги. Оно по-разному отображается на различных устройствах, выводя список мест и подробную информацию о каждом из них и предоставляя возможность войти и оставить отзывы

1.7.2. Как компоненты стека MEAN работают вместе

К тому времени, как вы закончите чтение этой книги, у вас будет работающее на стеке MEAN приложение, повсеместно использующее JavaScript. MongoDB хранит данные в виде двоичной JSON, которые через Mongoose доступны как JSON. Фреймворк Express располагается поверх Node.js, весь код в котором написан на JavaScript. На клиентской стороне располагается AngularJS — опять-таки JavaScript. Их соединения и ход выполнения демонстрирует рис. 1.8.

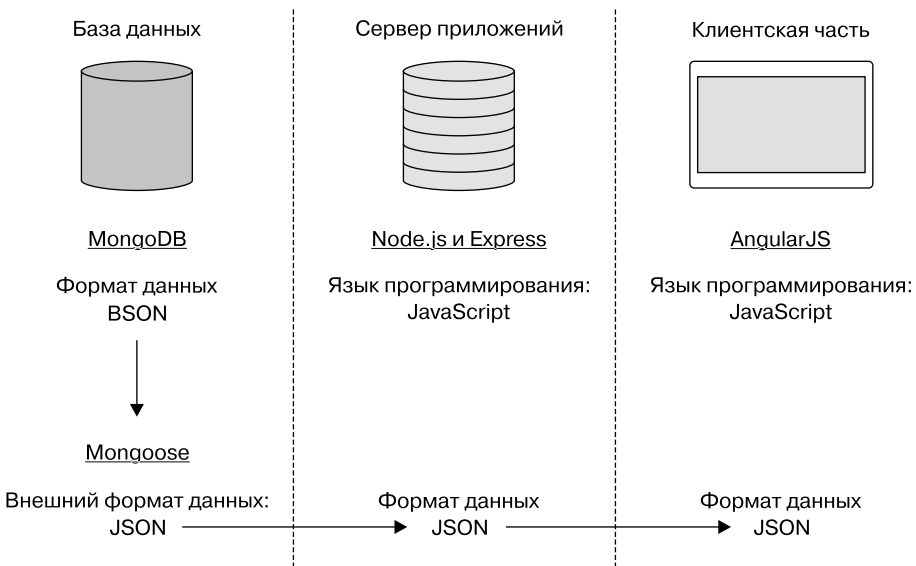


Рис. 1.8. JavaScript — общий язык программирования для всего стека MEAN, а JSON — общий формат данных

Мы рассмотрим различные виды архитектуры стека MEAN и способ создания Loс8г в главе 2.

1.8. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Различные технологии, из которых состоит стек MEAN.
- ❑ MongoDB в качестве слоя базы данных.
- ❑ Совместную работу Node.js и Express для обеспечения слоя сервера приложений.

- ❑ AngularJS, обеспечивающий замечательный слой привязки данных для клиентской части.
- ❑ Совместную работу всех компонентов стека MEAN.
- ❑ Несколько путей возможного расширения стека MEAN дополнительными технологиями.

Поскольку JavaScript играет в стеке центральную роль, пожалуйста, загляните в приложение D (доступное в Интернете), содержащее памятку по подводным камням и рекомендуемым решениям JavaScript.

В главе 2 мы обсудим, насколько стек MEAN гибок и как можно спроектировать его архитектуру по-разному для различных сценариев применения.

Глава 2

Проектируем архитектуру на основе стека MEAN

В этой главе:

- ❑ знакомство с обычной архитектурой стека MEAN;
- ❑ соображения по поводу одностраничных приложений;
- ❑ обзор альтернативных архитектур стека MEAN;
- ❑ проектирование архитектуры для реального приложения;
- ❑ планирование построения приложения на основе нашего проекта архитектуры.

В главе 1 мы рассматривали компоненты стека MEAN и их совместную работу. В данной главе более подробно рассмотрим, как они работают вместе.

Начнем с рассмотрения того, что некоторые люди считают *единственно возможной* архитектурой стека MEAN, особенно когда сталкиваются с ним впервые. С помощью нескольких примеров мы изучим, почему вам может понадобиться использовать другую архитектуру, а также поменять местами части стека. MEAN — очень мощный стек, его можно применять для решения разнообразных задач... если проявить немного творческой смекалки при проектировании решений.

2.1. Традиционная архитектура стека MEAN

Распространенный способ проектирования приложения на основе стека MEAN — использование API REST (representational state transfer — передача состояния представления) для поставки данных одностраничному приложению. Этот API обычно создается с помощью MongoDB, Express и Node.js, при построенном на

AngularJS SPA. Такой подход особенно популярен у разработчиков, перешедших на стек MEAN с AngularJS и ищущих стек, который предоставляет быстрый адаптивный API. Основы этого подхода и поток данных иллюстрирует рис. 2.1.

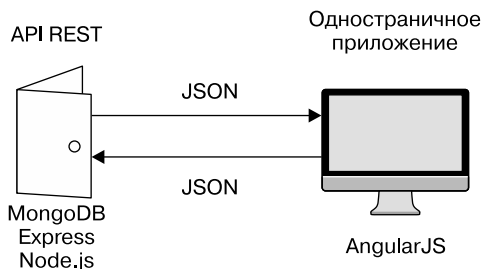


Рис. 2.1. Традиционный подход к архитектуре на основе стека MEAN, использующий MongoDB, Express и Node.js для создания API REST, поставляющего данные JSON запущенному на браузере одностраничному приложению AngularJS

ЧТО ТАКОЕ API REST

REST расшифровывается как «передача состояния представления» (REpresentational State Transfer) и представляет собой скорее архитектурный стиль, чем жесткий протокол. В REST отсутствует сохранение состояния, он ничего не знает о состоянии или истории действий текущего пользователя.

API — аббревиатура словосочетания application program interface — интерфейс программирования приложений, который дает возможность приложениям общаться друг с другом.

Таким образом, API REST — интерфейс без сохранения состояния к вашему приложению. В случае применения стека MEAN API REST используется для создания интерфейса без сохранения состояния к БД, предоставляя другим приложениям способ работы с данными.

На рис. 2.1 изображена отличная схема, даже идеальная, если у вас применяется (или вы хотите создать) SPA в качестве пользовательского интерфейса. AngularJS спроектирован с ориентацией на создание SPA, получение данных из API REST, равно как и отправку их обратно. MongoDB, Express и Node.js также обладают огромными возможностями по созданию API, используя JSON во всем стеке, включая и саму БД.

Именно поэтому многие люди приходят к стеку MEAN в поисках ответа на вопрос: «Я создал приложение на AngularJS, откуда я теперь смогу получить данные?»

Подобная архитектура замечательна в случае SPA, но что, если у вас его нет или вы не хотите его использовать? Если это единственная рассматриваемая вами архитектура на основе стека MEAN, то вы можете почувствовать себя в тупике и на-

чать искать решение в других местах. Но стек MEAN чрезвычайно гибок. Все четыре компонента обладают обширными возможностями и могут многое вам предложить.

2.2. Выходим за рамки одностраничных приложений

Программирование SPA на AngularJS подобно поездке на Porsche с опущенным верхом по прибрежному шоссе. И то и другое изумительно. Оба — многофункциональные, быстрые, классные, динамичные и обладают очень, очень широкими возможностями. И весьма вероятно, что и то и другое — огромный шаг вперед по сравнению с тем, что вы делали раньше.

Но иногда они не подходят для ваших задач. Если вам необходимо уложить доски для серфинга и выехать вместе с семьей на недельку в отпуск, спортивная машина доставит вам немало трудностей. Как ни замечательна она, в этом случае вам понадобится что-то иное. То же самое со SPA. Да, создавать их на AngularJS замечательно, но иногда это не лучшее решение вашей задачи.

Давайте бегло взглянем на кое-что касающееся SPA, что стоит иметь в виду при проектировании решения и выяснении того, подходит полный SPA для вашего проекта или нет. Данный раздел ни в коем случае не направлен против SPA. Одностраничные приложения обычно предоставляют потрясающий пользовательский интерфейс, при этом снижается нагрузка на ваши серверы и, следовательно, расходы на хостинг. В разделах 2.3.1 и 2.3.2 мы рассмотрим хороший и плохой пользовательские сценарии для одностраничных приложений и на самом деле создадим одно к концу книги!

2.2.1. Сложности со сканированием сайтов

Приложения JavaScript создают немало сложностей для поисковых систем с точки зрения сканирования и индексации сайтов. Большинство поисковых систем просматривают HTML-содержимое страницы, но не скачивают и не выполняют JavaScript. Те же, которые делают это, фактически сканируют созданный JavaScript контент далеко не так хорошо, как выдаваемый сервером. Если весь ваш контент выдается через JavaScript-приложение, вы не будете знать, какая его часть будет проиндексирована.

Сопутствующий недостаток: автоматически создаваемые превью страниц из социальных сетей, таких как Facebook, LinkedIn и Pinterest, работают не очень хорошо. Это происходит еще и потому, что они обращаются к HTML-коду соответствующей страницы и пытаются извлечь подходящие изображения и текст. Подобно поисковым системам, они не выполняют имеющийся на странице JavaScript-код, так что выдаваемый JavaScript контент не виден.

Придание SPA видимости сканируемости

Существует пара обходных маневров, позволяющих придать вашему сайту видимость сканируемости. Оба включают создание отдельных HTML-страниц, зеркально отображающих содержимое вашего SPA. Вы можете заставить свой сервер создавать основанную на HTML версию сайта и выдавать ее поисковым агентам, также можно использовать автономный браузер, такой как PhantomJS, для запуска вашего JavaScript-приложения и вывода результирующего HTML.

Оба этих подхода требуют немалых усилий и могут вызвать головную боль у службы техподдержки, если ваш сайт крупный и сложный. Имеются здесь и потенциальные подводные камни с точки зрения оптимизации поисковых систем (search engine optimization (SEO)). Если ваш сгенерированный сервером HTML-код покажется поисковому агенту слишком отличным от контента SPA, то ваш сайт может быть оштрафован. Запуск PhantomJS для вывода HTML может замедлить время отклика ваших страниц, а за это поисковые системы, в частности Google, будут ставить вас дальше в списке выдачи.

Имеет ли это значение?

Имеет это значение или нет, зависит от того, что вы хотите создать. Если основной план развития того, что вы создаете, связан с трафиком поисковых систем или соцсетями, то вам следует уделить сканируемости немалое внимание. Если же вы создаете что-либо маленькое, которое и должно будет остаться маленьким, то можете воспользоваться вышеупомянутыми обходными путями, которые при больших масштабах потребовали бы значительных усилий.

В то же время если вы создаете приложение, которому не требуется слишком большая SEO, или если вы *хотите*, чтобы ваш сайт плохо поддавался сканированию, то беспокоиться об этом смысла нет. Это может оказаться даже определенным преимуществом.

2.2.2. Веб-аналитика и история браузера

Аналитические инструменты, такие как Google Analytics, в значительной степени зависят от полной загрузки страниц в браузер, инициируемой изменением URL. SPA подобным образом не работают. Именно поэтому они и называются *одностраничными приложениями*!

После загрузки первой страницы все загрузки последующих страниц и изменения контента обрабатываются внутри приложения. Так что браузер никогда не начнет загрузку новой страницы, в историю браузера ничего нового не попадет, и ваш аналитический пакет понятия не будет иметь о том, кто что делает на вашем сайте.

Добавление загрузок страниц к SPA

Можно добавить события загрузки страниц к SPA с помощью API историй HTML5, это поможет вам интегрировать в сайт аналитические инструменты. Сложность заключается в управлении им и обеспечении правильного отслеживания всего происходящего на сайте, что включает проверку пропущенных отчетов и дублирующихся записей.

Хорошая новость заключается в том, что вам не нужно создавать все с нуля. Существует несколько доступных в Интернете вариантов интеграции аналитических инструментов для AngularJS с открытым исходным кодом. Вам по-прежнему нужно будет внедрить их в приложение и убедиться, что все работает правильно, но не нужно будет делать все с самого начала.

Серьезная ли это проблема?

Масштабы данной проблемы зависят от вашей потребности в гарантированно точной аналитике. Если вы хотели бы наблюдать за тенденциями в потоках и действиях посетителей, то интеграция, вероятно, покажется вам несложной. Чем больше подробностей и гарантированной точности вам нужно, тем больше работы придется выполнить при разработке и тестировании. Хотя, вероятно, намного легче просто включить код вашей веб-аналитики в каждую страницу сгенерированного сервером сайта, интеграция аналитики вряд ли окажется единственной причиной, по которой вам захочется выбрать вариант не одностраничного приложения.

2.2.3. Скорость начальной загрузки

SPA отличаются более медленной загрузкой первой страницы, чем серверные приложения. Это происходит потому, что при первой загрузке приходится загружать фреймворк и код приложения, прежде чем визуализировать в браузере нужное представление в виде HTML. Серверному приложению нужно всего лишь отправить требуемый HTML-код браузеру, что уменьшает время ожидания и время загрузки.

Ускоряем загрузку страницы

Существуют определенные способы ускорения начальной загрузки SPA, такие как довольно тяжеловесный подход кеширования и отложенной загрузки модулей по мере их необходимости. Но вы никогда не сможете избежать загрузки фреймворка, а также по крайней мере части кода приложения и, вероятно, необходимости обратиться к API за данными до отображения чего-либо в браузере.

Стоит ли волноваться насчет скорости?

Отвечая на вопрос, следует ли вам беспокоиться по поводу скорости начальной загрузки страницы, опять-таки скажу: «Зависит от обстоятельств». Это зависит от того, что именно вы создаете и сколько людей будет взаимодействовать с данным продуктом.

Вспомните Gmail. Gmail представляет собой SPA и требует для загрузки немало времени. Правда, это обычно всего лишь пара секунд, но в наше время в Интернете все нетерпеливы и ожидают немедленного отклика. При этом люди не против подождать загрузки Gmail, ведь в процессе работы он ведет себя адаптивно и быстро. А раз уж вы зашли, то часто остаетесь в нем на какое-то время.

Но если речь идет о блоге, извлекающем информацию из поисковых систем и других внешних источников, не хотелось бы несколько секунд ждать загрузки первой страницы. Посетители в таком случае сочтут, что ваш сайт не работает или работает слишком медленно, и нажмут кнопку Назад, прежде чем у вас появится шанс показать им ваш контент. Готов поспорить, вам это знакомо, ведь вы и сами не раз поступали точно так же!

2.2.4. SPA или не SPA, вот в чем вопрос

Напомню, что это не была тренировка в критике SPA, мы просто воспользовались случаем, чтобы подумать о вещах, которые часто откладывают в сторону до тех пор, пока не окажется слишком поздно. Три вопроса, касающихся возможности сканирования, интеграции веб-аналитики и скорости загрузки страницы, не были предназначены для формулировки четких рекомендаций относительно того, когда создавать SPA, а когда — делать что-то иное. Их целью было лишь дать вам пищу для размышлений.

Может оказаться, что ни одна из этих вещей не является проблемой для вашего проекта и что SPA определенно подходящий вариант. Но если вы видите, что каждый вопрос заставляет вас задуматься и, похоже, вам понадобится ввести в приложение обходной маневр для каждого из них, то SPA, наверное, не лучший вариант.

Если же ваш случай где-то посередине, то вам нужно определить для себя, что наиболее важно и, следовательно, что окажется наилучшим решением для проекта. В качестве эмпирического правила: если ваше решение с самого начала содержит множество обходных маневров, то, вероятно, лучше его пересмотреть.

Даже если вы решили, что SPA вам не подходит, это не значит, что вы не можете использовать стек MEAN. Пойдем дальше и посмотрим на другой вариант архитектуры.

2.3. Разработка гибкой архитектуры MEAN

Если работа с AngularJS похожа на обладание Porsche, то остальная часть стека соответствует наличию в гараже дополнительного Audi RS6. Множество людей обратят внимание на спортивную машину перед домом и не заметят автомобиль-универсал в гараже. Но если все-таки зайти в гараж и полюбопытствовать, то окажется, что под его капотом двигатель Lamborghini V10. Этот автомобиль-универсал может предложить гораздо больше, чем кажется на первый взгляд!

Использование MongoDB, Express и Node.js вместе только для создания API REST подобно использованию Audi RS6 лишь для того, чтобы возить ребенка в школу. И они и он обладают отличными возможностями и выполняют эту работу очень хорошо, но при этом способны на большее.

Мы уже кратко обсуждали, что можно получить с помощью этих технологий, в главе 1, но вот несколько отправных точек.

- MongoDB может хранить и обрабатывать потоковым образом двоичную информацию.
- Node.js особенно хорош для подключений в режиме реального времени с помощью веб-сокетов.
- Express — фреймворк веб-приложений со встроенным применением шаблонов, маршрутизацией и управлением сеансами.

Это далеко не полный список, и я определенно не смогу рассмотреть все возможности всех этих технологий в данной книге. Мне бы понадобилось несколько книг для этого! Все, что я могу сделать, — привести простой пример и показать, как можно соединить воедино части стека MEAN для проектирования наилучшего решения.

2.3.1. Требования к движку блога

Взглянем на уже знакомую нам идею движка блога и посмотрим, как можно наилучшим образом спроектировать стек MEAN для его создания.

Движок блога обычно включает две части. Это ориентированная на публику интерфейсная часть, выдающая читателям статьи, которые, хочется надеяться, будут перепродаваться и распространяться по Интернету. У движка блога имеется также интерфейс администратора, в который владельцы блогов входят для написания новых статей и управления своими блогами. Некоторые из основных характеристик этих двух частей демонстрирует рис. 2.2.

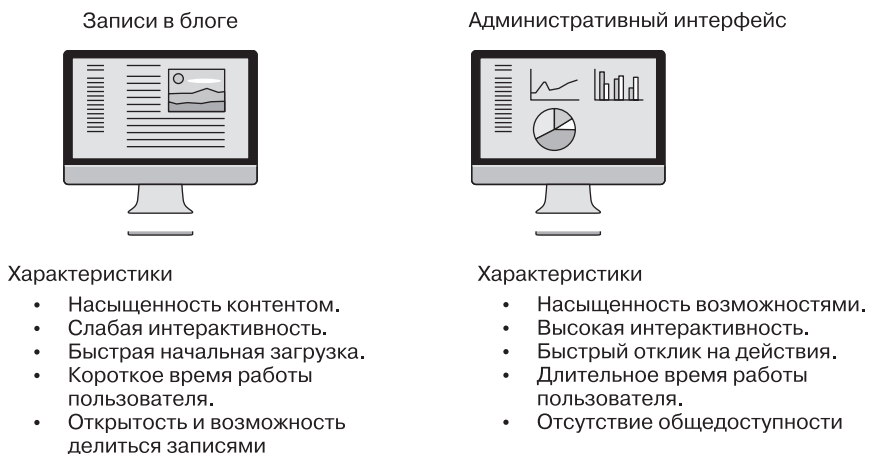


Рис. 2.2. Конфликтующие характеристики двух частей движка блога — записей в блоге, ориентированных на публику, и необщедоступного административного интерфейса

Глядя на списки на рис. 2.2, легко увидеть высокий уровень конфликтования между характеристиками этих двух частей. Вам нужна насыщенная контентом и слабоинтерактивная среда для статей блогов и богатая возможностями высокоинтерактивная среда для административного интерфейса. Статьи блога должны быть быстрозагружаемыми — для снижения показателя отказов, тогда как административная область должна быстро реагировать на действия пользователя и вводимые им данные. Наконец, пользователи обычно задерживаются на записи в блоге ненадолго, но могут захотеть поделиться ею с другими, тогда как административный интерфейс необщедоступен и отдельный пользователь может работать в нем длительное время.

Принимая во внимание обсуждавшиеся ранее потенциальные проблемы со SPA и глядя на характеристики записей в блоге, вы можете обнаружить немало совпадений. Вполне вероятно, что с учетом этого вы решите не использовать SPA для выдачи своих записей в блоге читателям. В то же время SPA отлично подходит для административного интерфейса.

Так что же делать? Возможно, самым важным будет удержание читателей блога — если их впечатления от вашего блога были негативными, они просто не вернуться и не будут делиться записями. Если блог не привлекает, то блогер просто прекратит в него писать или перейдет на другую платформу. Опять-таки медленный и плохо реагирующий административный интерфейс станет причиной того, что владельцы блогов начнут «убегать с корабля». Как можно добиться того, чтобы все были довольны, а движок блога был при деле?

2.3.2. Архитектура движка блога

Ответ заключается в том, что не надо искать одно решение на все случаи жизни. Пусть у вас будет два приложения. У вас есть ориентированный на публику контент, который нужно доставлять непосредственно с сервера, и интерактивный необщедоступный административный интерфейс, который вы хотите выполнить в виде SPA. Давайте рассмотрим каждое из этих приложений по отдельности, начав с административного интерфейса.

Административный интерфейс: SPA на AngularJS

Мы уже обсуждали, что для административного интерфейса идеально подходило бы SPA, построенное на основе AngularJS. Так что архитектура этой части движка будет выглядеть очень знакомо: API REST, построенный на MongoDB, Express и Node.js, с одностраничным приложением на AngularJS в интерфейсной части. Рисунок 2.3 демонстрирует, как это будет выглядеть.

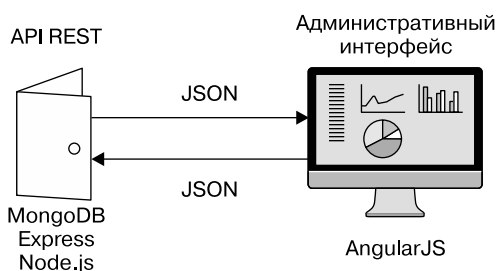


Рис. 2.3. Знакомая картина: административный интерфейс представляет собой SPA на основе AngularJS, использующее API REST, построенное на MongoDB, Express и Node.js

Ничего особенно нового на рис. 2.3 вы не увидите. Все приложение построено на AngularJS и выполняется в браузере, а между приложением AngularJS и API REST туда и обратно передаются данные JSON.

Записи в блоге: что делать?

Если задуматься о записях в блоге, то все оказывается несколько сложнее.

Представьте себе стек MEAN только как одностраничное приложение на AngularJS, обращающееся к API REST, и вы окажетесь в дураках. Вы можете в любом случае создать интерфейсную часть в виде SPA, раз хотите использовать JavaScript и стек MEAN. Но это не лучшее решение. Вы можете счесть, что стек MEAN просто не подходит для данного случая, и выбрать другой стек технологий. Но вам ведь этого не надо! Вам нужен просто сплошной JavaScript.

Так что взглянем на стек MEAN повнимательнее и подумаем обо всех его компонентах. Вы знаете, что Express — фреймворк веб-приложений. Вы знаете, что Express может использовать шаблонизаторы для создания HTML-кода на сервере. Вы знаете, что Express может использовать маршрутизацию URL и паттерны MVC. Вам стоит задуматься: а не в Express ли заключается ответ?

Записи в блоге: используем Express

В данном сценарии с блогом доставка HTML-кода и контента непосредственно с сервера — именно то, что вам нужно. Express выполняет это особенно хорошо, даже с самого начала обеспечивает выбор шаблонизаторов. HTML-содержимое потребует данные из БД, так что вы снова используете для этого API REST (подробнее мы рассмотрим, почему это наилучший подход, в следующем подразделе). Основу такой архитектуры демонстрирует рис. 2.4.

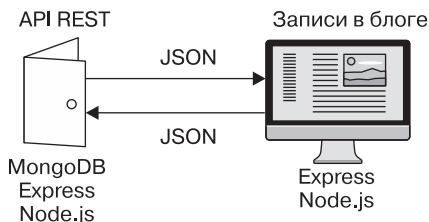


Рис. 2.4. Архитектура доставки HTML-кода непосредственно с сервера: спереди приложения Express и Node.js, взаимодействующие с API REST, построенным на MongoDB, Express и Node.js

Вы получаете подход, при котором можете использовать стек MEAN или по крайней мере его часть для доставки основанного на базе данных контента непосредственно от сервера к браузеру. Но этим дело не ограничивается. Стек MEAN еще более гибок.

Записи в блоге: используем стек полнее

Вы видели приложение Express, выдающее контент блога посетителям. Если вам нужно, чтобы посетители могли заходить на сайт, возможно, для добавления комментариев к статьям, вам необходимо отслеживать сеансы пользователей. Для этого можно использовать MongoDB вместе с вашим приложением на Express.

Можно также рядом с вашими сообщениями в блоге динамически отображать какие-либо данные, например связанные сообщения или поисковое поле с автодополнением, опережающим ввод с клавиатуры. Их можно реализовать с помощью AngularJS. Помните: AngularJS предназначен не только для SPA, его можно использовать также для добавления каких-нибудь интерактивных данных на страницу, которая иначе была бы статической. Рисунок 2.5 показывает, как эти необязательные части MEAN добавляются в архитектуру компонента блога.

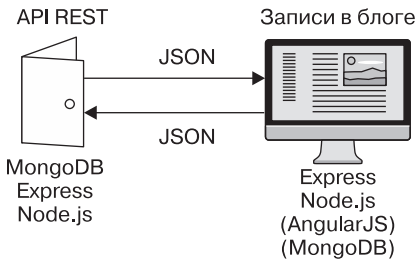


Рис. 2.5. Добавление параметров использования AngularJS и MongoDB как части интерфейсного аспекта движка блога, выдающего записи посетителям

Теперь вы можете организовать взаимодействие приложения, выдающего посетителям контент и основанного на полном стеке MEAN, с вашим API REST.

Движок блога: гибридная архитектура

На этой стадии у нас имеется два отдельных приложения, использующих API REST. Если немного спланировать, это может быть общий API REST, применяемый обеими частями приложения.

Целостность архитектуры с одним API REST, взаимодействующим с двумя приложениями клиентской части, демонстрирует рис. 2.6.

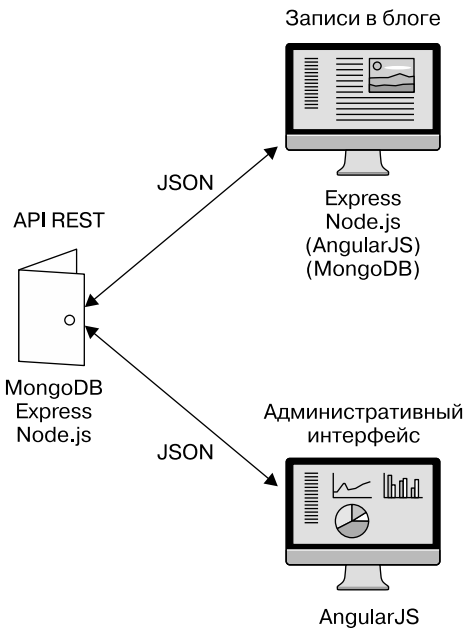


Рис. 2.6. Гибридная архитектура стека MEAN: ради обеспечения наиболее подходящего решения один API REST поставляет данные двум ориентированным на взаимодействие с пользователями приложениям, основанным на различных частях стека MEAN

Это простой пример демонстрации возможности соединения разнообразных частей стека MEAN в различные архитектуры для удовлетворения требований, которые предъявляет к вам проект. Возможные варианты ограничиваются лишь вашим знанием компонентов и изобретательностью при их сочетании. Не существует какой-то одной правильной архитектуры для стека MEAN.

2.3.3. Рекомендуемое решение: создание внутреннего API для слоя данных

Вероятно, вы обратили внимание на то, что каждый вариант архитектуры включает API, создающий прослойку над данными, и предоставляет основному приложению возможность взаимодействовать с БД. Для этого имеются свои причины.

Если вы собирались начать с построения приложения на Node.js и Express, выдавая HTML-содержимое непосредственно с сервера, было бы очень удобно обращаться к БД прямо из кода приложения Node.js. В краткосрочной перспективе такой подход кажется удобным. Но в долгосрочной перспективе он оказывается более сложным, так как тесно сцепляет ваши данные с кодом приложения таким способом, при котором более ничто не может использовать этот код.

Другой вариант: создать собственный API, способный непосредственно обращаться к БД и выводить нужные вам данные. При этом приложение Node.js может взаимодействовать с этим API, не обращаясь напрямую к БД. Эти два варианта сравниваются на рис. 2.7.

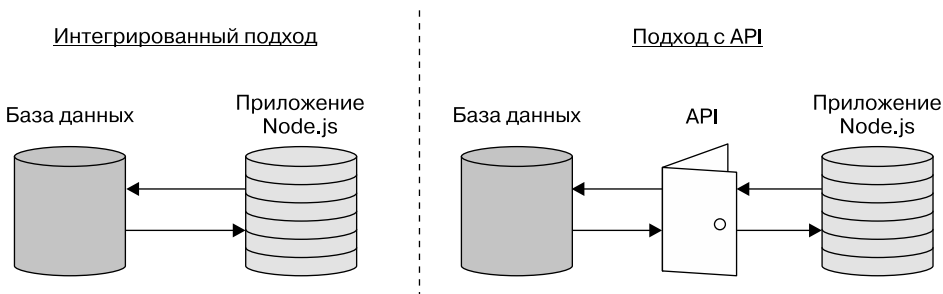


Рис. 2.7. Краткосрочная перспектива интеграции данных в приложение Node.js. Приложение Node.js может напрямую обращаться к БД, или можно создать API, который будет взаимодействовать с БД, при этом приложение будет обращаться только к этому API

Глядя на рис. 2.7, вы можете удивиться: зачем тратить силы на создание API, который будет просто занимать место между вашим приложением и БД? Может, это просто дополнительная работа? На данном этапе — да, это требует дополни-

тельной работы, но мы взглянем чуть дальше. Что если вам захочется потом использовать свои данные в нативном мобильном приложении? Или, например, в клиентской части на AngularJS?

Вряд ли вам захочется оказаться в положении, когда нужно будет создавать для каждого из них отдельные, хотя и похожие интерфейсы. Этого можно избежать, если заранее создать выводящий нужные вам данные API. Если у вас под рукой есть API при интеграции слоя данных в ваше приложение, можно просто сделать так, чтобы приложение ссылалось на этот API. И неважно, основано ли приложение на Node.js, AngularJS или iOS. Причем API не должен быть общедоступным, главное, чтобы к нему могли обращаться вы. Сравнение двух подходов для приложений Node.js, AngularJS или iOS, использующих один и тот же источник данных, демонстрирует рис. 2.8.

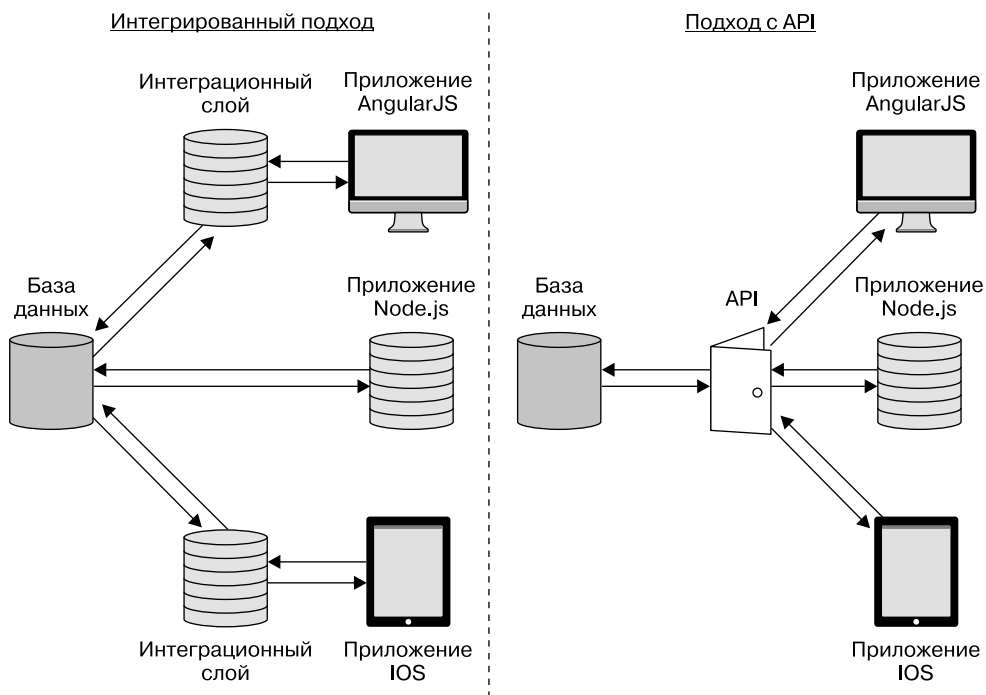


Рис. 2.8. Долгосрочная перспектива интеграции данных в ваше приложение Node.js, а также дополнительные приложения AngularJS и iOS. Интегрированный подход стал фрагментированным, а подход с API — прост и удобен в сопровождении

Как видно из рис. 2.8, ранее более простой интегрированный подход теперь оказывается фрагментированным и сложным. Приходится контролировать и сопровождать три отдельные интеграции данных чтобы сохранить согласованность,

любые изменения нужно будет выполнять в нескольких местах. Если же у вас один API, то можно не беспокоиться обо всем этом. Так что, приложив поначалу немного больше усилий, можно сделать свою дальнейшую жизнь намного проще. Мы рассмотрим создание внутренних API в главе 6.

2.4. Планирование реального приложения

Как я упоминал в главе 1, на протяжении этой книги мы будем создавать работающее приложение `Loc8r`, основанное на стеке MEAN. `Loc8r` будет выводить список расположенных поблизости мест с Wi-Fi, куда можно пойти, чтобы поработать. Пользователи в нем смогут также отправить свои оценки и отзывы.

Для демонстрационного приложения мы сфабрикуем данные таким образом, чтобы можно было легко и удобно его протестировать. Итак, приступим к планированию.

2.4.1. Планирование приложения на высоком уровне

Сначала обдумаем, какие экраны понадобятся в нашем приложении. Сосредоточим внимание на отдельных представлениях страниц на экране и путях пользователей. Сделаем это на очень общем уровне, не особо заботясь о деталях того, что находится на каждой странице. Хорошей идеей будет зарисовать этот этап на бумаге или доске для визуализации приложения в целом. Это будет полезно также для систематизации экранов по наборам и потокам и послужит хорошим ориентиром, когда наступит время создавать приложение. Поскольку со страницами не связаны никакие данные и за ними нет логики приложения, можно легко и свободно добавлять и удалять составные части, изменять, что и где отображать, и даже менять количество страниц. Вполне вероятно, что с первого раза не получится правильно, главное — начать и повторять процесс, улучшая все до тех пор, пока вы не будете удовлетворены отдельными страницами и общими путями движения пользователей.

Планирование экранов

Приступим. Как уже говорилось, наша цель следующая: приложение `Loc8r` будет выводить список расположенных поблизости мест с Wi-Fi, куда можно пойти, чтобы поработать. Оно также отображает наличие удобств, время начала работы, рейтинг и схему проезда. Пользователи получают возможность отправлять свои оценки и отзывы.

Из этого можно сделать вывод о том, что нам потребуются:

- ❑ экран со списком близлежащих мест с Wi-Fi;
- ❑ экран, выводящий подробности относительно конкретного места;
- ❑ экран для добавления отзыва о месте.

Вероятно, нам понадобится также сообщить посетителям, для чего предназначено `Loc8r` и откуда оно взялось, так что не помешает добавить в список еще одну форму — экран для информации `About` (О нас).

Распределение экранов по наборам

Далее нам нужно взять список экранов и упорядочить их в соответствии с тем, как они относятся друг к другу. Например, первые три экрана в списке связаны с местом расположения. Страница `About` (О нас) ни с чем не связана, так что ее можно отнести к смешанному набору `Others` (Другие). Зарисовка схемы даст нам что-то вроде рис. 2.9.

Место расположения



Другие

Страница
О нас



Рис. 2.9. Распределение отдельных экранов приложения по логическим наборам

Подобный быстрый набросок — первый этап планирования, и нам нужно выполнить его, прежде чем начать думать об архитектуре. Этот этап дает нам возможность взглянуть на основные страницы, а заодно обдумать пути движения пользователей. Рисунок 2.9, например, демонстрирует также основной путь пользователя в наборе `Locations` (Место расположения): переход со страницы `List` (Список) на страницу `Details` (Подробности), а затем в форму для добавления отзыва.

2.4.2. Проектирование архитектуры приложения

На первый взгляд `Loc8r` довольно простое приложение всего с несколькими экранами. Но нам все равно необходимо обдумать его архитектуру, так как мы собираемся передавать данные из БД в браузер, позволять пользователям взаимодействовать с данными и обеспечить возможность обратной отправки данных в БД.

Начнем с API

Поскольку приложение собирается использовать БД и передавать данные, разумно начать проектировать архитектуру с той части, которая определено будет нам необходима. Отправной пункт — API REST, созданный с помощью `Express` и `Node.js` для обеспечения возможности взаимодействия с базой данных `MongoDB`, демонстрирует рис. 2.10.

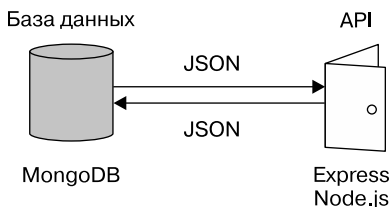


Рис. 2.10. Начинаем с обычного API REST, используя `MongoDB`, `Express` и `Node.js`

Как уже обсуждалось в этой главе, создание API для обмена данными предопределено и является основным пунктом архитектуры. Так что обратимся к более интересному и сложному вопросу: каким образом мы будем проектировать архитектуру самого приложения?

Варианты архитектуры приложения

На этой стадии нам нужно взглянуть на то, чего конкретно требует наше приложение, и решить, как следует объединять части стека `MEAN` с целью создания наилучшего решения. Необходимо ли нам что-то специфическое от `MongoDB`, `Express`, `AngularJS` и `Node.js`, такое, что может существенно повлиять на наши решения? Хотим ли мы, чтобы HTML-содержимое выдавалось непосредственно с сервера, или SPA будет лучшим вариантом?

Что касается `Loc8r`, то каких-либо необычных или специальных требований у него нет, а желаемая степень удобства сканирования поисковыми системами зависит от бизнес-плана по развитию. Если цель состоит в привлечении натурального трафика от поисковых систем, то да, возможность сканирования необходима. Если же цель заключается в продвижении приложения как такового, то видимость его для поисковых систем менее важна.

Возвращаясь к примеру с блогом, можно сразу же наметить три возможные архитектуры приложения, как показано на рис. 2.11.

1. Приложение на Node.js и Express.
2. Приложение на Node.js и Express с дополнениями на AngularJS для интерактивности.
3. SPA на AngularJS.

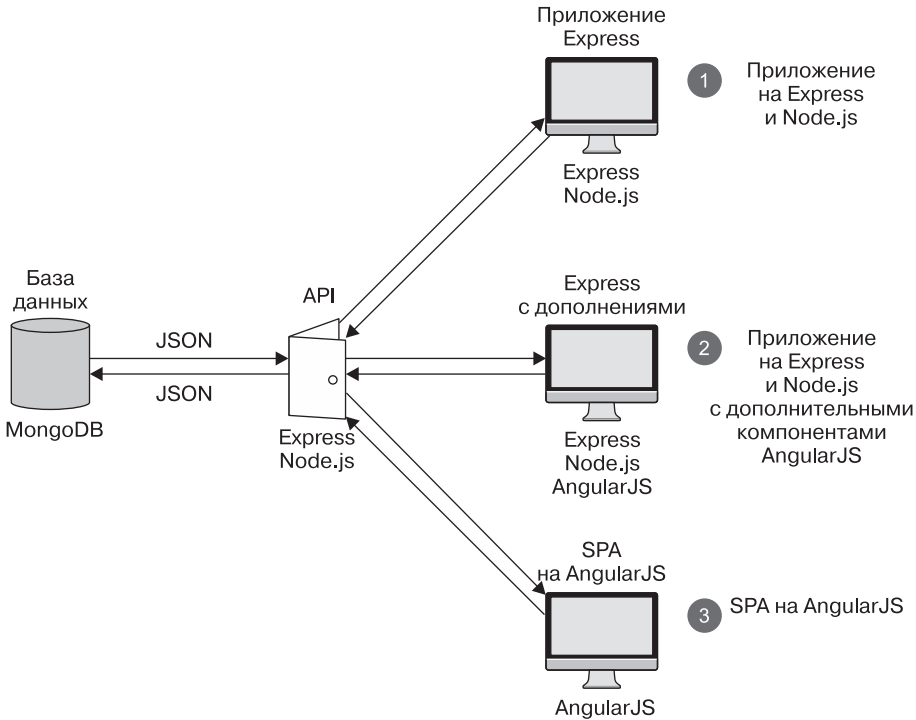


Рис. 2.11. Три варианта построения приложения Los8r, начиная с серверного приложения на Node.js и Express и заканчивая полностью клиентским SPA на AngularJS

Какой же из этих трех вариантов оптимален для Los8r?

Выбор архитектуры приложения

Каких-либо особых бизнес-требований, которые заставляли бы нас предпочесть один из вариантов архитектуры другому, отсутствуют. Впрочем, это несущественно, поскольку мы собираемся реализовать в книге все три варианта. Построение всех трех архитектур позволит нам изучить работу каждого из подходов и даст возможность взглянуть по очереди на каждую из технологий, создавая приложение слой за слоем.

Мы будем создавать эти архитектуры в порядке, показанном на рис. 2.11, начиная с приложения на Node.js и Express. Затем перейдем к добавлению некоторого количества кода на AngularJS, прежде чем переделать его в SPA на AngularJS. Хотя при обычных условиях вы не станете разрабатывать сайт подобным образом, это даст вам отличную возможность изучить все тонкости стека MEAN. В разделе 2.5 мы обсудим этот подход вкратце и пройдемся по плану более детально.

Закключаем все в проект Express

Схемы архитектуры, которые мы видели до сих пор, подразумевали наличие у нас отдельных приложений Express для API и логики приложения. Это вполне возможно и хорошо подходит для большого проекта. Если мы ожидаем больших объемов трафика, то даже можем захотеть разместить главное приложение и API на отдельных серверах. Дополнительная выгода от этого — возможность задавать для каждого из серверов и приложений особые настройки, лучше приспособленные для их индивидуальных нужд.

Другой способ более прост, при его реализации все содержится в одном проекте Express. В этом случае нам нужно думать о размещении и развертывании только одного приложения и управлении лишь одним набором исходного кода. Именно так мы и поступим с `Lo8gr`, используя один проект Express, содержащий несколько подприложений. Данный подход проиллюстрирован на рис. 2.12.

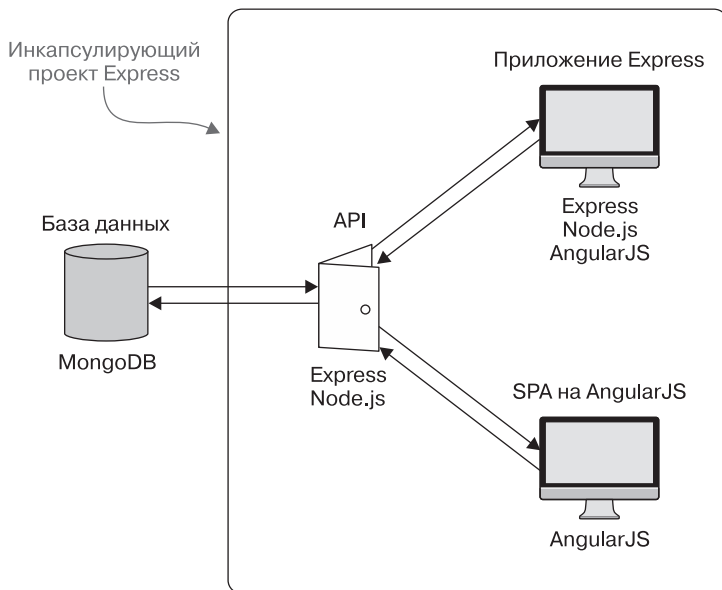


Рис. 2.12. Архитектура приложения с API и логикой приложения, содержащимися в одном проекте Express

При компоновке приложения таким образом важно хорошо организовать код, чтобы различные части приложения хранились отдельно от других. Это облегчит не только сопровождение кода, но и разбиение его в будущем на отдельные проекты, если вы сочтете это необходимым. Такова основная тема, к которой мы будем возвращаться на протяжении всей книги.

2.4.3. Конечный продукт

Как можно увидеть, мы будем использовать для создания Loc8r все слои стека MEAN. А также включим сюда Twitter Bootstrap для облегчения создания адаптивного макета. Скриншоты приложения, которое мы будем создавать на протяжении данной книги, демонстрируются на рис. 2.13.

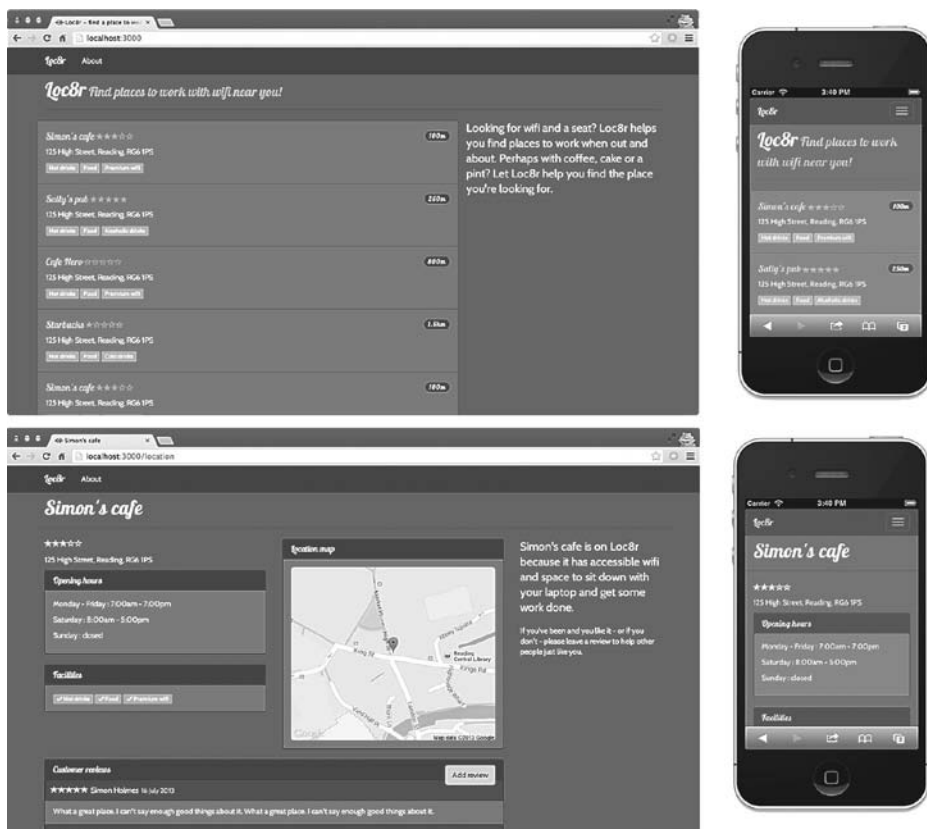


Рис. 2.13. Loc8r — приложение, которое мы собираемся создавать на протяжении данной книги. Оно будет отображаться по-разному на различных устройствах, выводя список мест и подробностей о каждом из них и предоставляя посетителям возможность войти и оставить отзывы

2.5. Разбиваем разработку на этапы

В этой книге мы ставим перед собой две цели:

- ❑ создать приложение на основе стека MEAN;
- ❑ по ходу дела изучить различные слои стека.

Мы подходим к проекту так, как я сам сделал бы при ускоренной разработке, но с несколькими поправками, чтобы вы лучше рассмотрели стек в целом. Начнем с пяти этапов ускоренной разработки прототипа, а затем посмотрим на использование этого подхода для построения `Loosr` слой за слоем, в процессе работы сосредотачиваясь на различных технологиях.

2.5.1. Этапы ускоренной разработки прототипа

Давайте разобьем процесс на несколько этапов, концентрируя усилия на одной части за раз, что увеличит наши шансы на успех. Подобный подход отлично позволяет претворять идеи в жизнь.

Этап 1. Создание статического сайта

Первый этап заключается в создании статической версии приложения, по сути представляющей собой набор HTML-страниц. Цели этого этапа:

- ❑ быстро придумать макет;
- ❑ убедиться, что пути движения пользователей логичны.

На данном этапе нас не волнуют вопросы базы данных или показуха в пользовательском интерфейсе. Все, что мы хотим, — создать работающий макет основных экранов и путей, которые пользователь будет проходить в приложении.

Этап 2. Проектирование модели данных и создание БД

Как только мы получили устраивающий нас работающий статический прототип, следующей задачей становится поиск в статическом приложении любых жестко зашитых данных и помещение их в БД. Цели этого этапа:

- ❑ описать модель данных¹, отражающую требования приложения;
- ❑ создать БД для работы с моделью.

¹ Автор использует понятие «модель данных» в смысле, приближенном к смыслу выражения «модель базы данных», хотя большинство специалистов сейчас понимают под ней инструмент, а не конечный результат. — *Примеч. пер.*

Первая цель — описание модели данных. Взглянем еще раз на наш проект с высоты птичьего полета и попробуем ответить на вопросы: данные о каких объектах нам требуются, как эти объекты взаимосвязаны и какие данные в них содержатся?

Если бы мы реализовывали этот этап до создания статического прототипа, то имели бы дело с абстрактными понятиями и идеями. Теперь, когда у нас есть прототип, мы видим, что происходит на различных страницах и где какие данные нужны. Внезапно этот этап значительно упрощается. Сами того не замечая, мы выполнили всю тяжелую часть анализа при создании статического прототипа.

Этап 3. Создание API данных

После выполнения этапов 1 и 2 в нашем распоряжении оказывается, с одной стороны, статический сайт, а с другой — база данных. На этом этапе и следующем за ним мы выполним очевидные шаги по их связыванию между собой.

Цель этапа 3 — создать API REST, который позволил бы нашему приложению взаимодействовать с БД.

Этап 4. Привязка БД к приложению

При переходе к данному этапу мы имели статическое приложение и API, предоставляющее интерфейс к БД. Цель этапа — сделать так, чтобы приложение взаимодействовало с API.

После завершения этого этапа приложение практически не изменится, но из БД будут поступать данные, так что наше приложение превратится в приложение, ориентированное на работу с данными!

Этап 5. Расширение приложения

Этот этап всецело посвящен снабжению приложения дополнительной функциональностью. Можно добавить системы аутентификации, проверки данных или методы отображения сообщений об ошибках пользователю. Он может также включать добавление дополнительной интерактивности в клиентской части или усиление бизнес-логики в самом приложении. Итак, фактически цели данного этапа:

- ❑ добавить заключительные штрихи к нашему приложению;
- ❑ довести приложение до готовности к реальному использованию.

С обеспечиваемой этими пятью этапами великолепной методологией можно приступить к созданию нового проекта. Давайте рассмотрим, как именно мы будем следовать им при построении Loc8r.

2.5.2. Шаги создания Loc8r

При построении Loc8r на протяжении этой книги мы преследуем две цели. Во-первых, конечно, мы хотим создать работающее приложение на основе стека MEAN. Во-вторых, хотим изучить различные технологии, понять, как их использовать и как сочетать разными способами.

Так что на протяжении этой книги мы будем придерживаться пяти этапов разработки, но с некоторыми отклонениями, чтобы увидеть в действии весь стек. Прежде чем подробно рассмотреть дальнейшие шаги, вкратце напомним предлагаемую архитектуру (рис. 2.14).

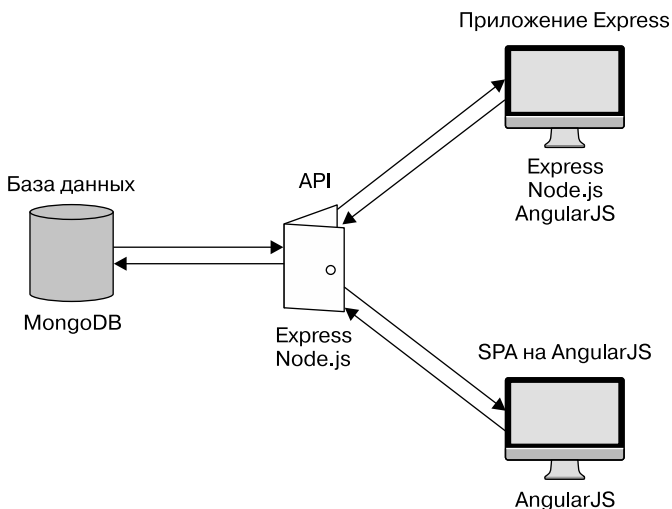


Рис. 2.14. Предлагаемая архитектура для Loc8r в том виде, в котором мы будем ее реализовывать в книге

Шаг 1. Создание статического сайта

Мы начнем с этапа 1 и создадим статический сайт. Я советую делать это для любого приложения или сайта, так как при этом можно многое узнать, затратив сравнительно небольшие усилия. При создании статического сайта уместно будет ориентироваться на будущее и иметь в виду, какой должна быть итоговая архитектура. Я уже описал архитектуру Loc8r, как показано на рис. 2.14.

На основе этой архитектуры создадим статическое приложение на Node и Express, используя его в качестве отправной точки работы со стеком MEAN. Этот шаг в нашем процессе выделен (рис. 2.15) как первая часть разработки предлагаемой архитектуры.

Данный шаг рассматривается подробнее в главах 3 и 4.

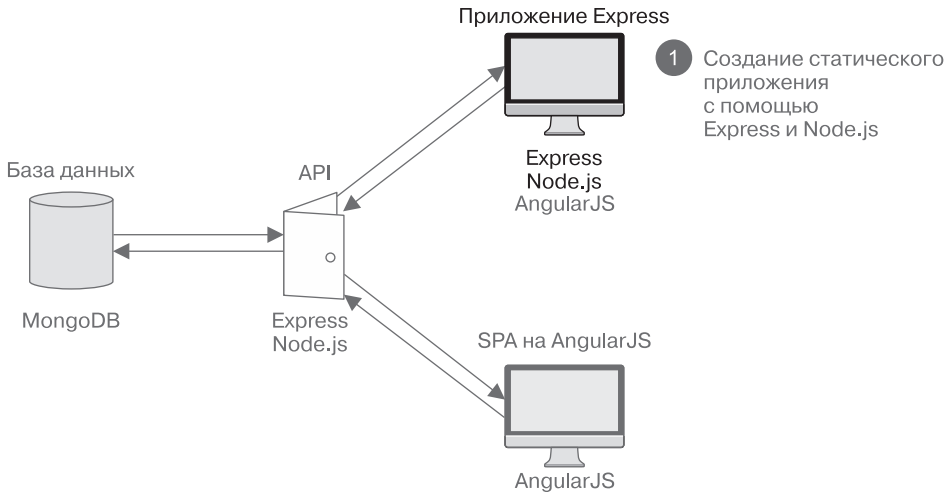


Рис. 2.15. Отправная точка работы над приложением — создание пользовательского интерфейса на основе Express и Node.js

Шаг 2. Проектирование модели данных и создание БД

Продолжим следовать этапам разработки и перейдем к этапу 2 — создадим БД и спроектируем модель данных. Опять-таки этот шаг, вероятно, понадобится любому приложению, но вы извлечете из него гораздо больше, если сначала сделаете шаг 1.

Дополнения, вносимые этим шагом в общую картину построения архитектуры приложения, иллюстрирует рис. 2.16.

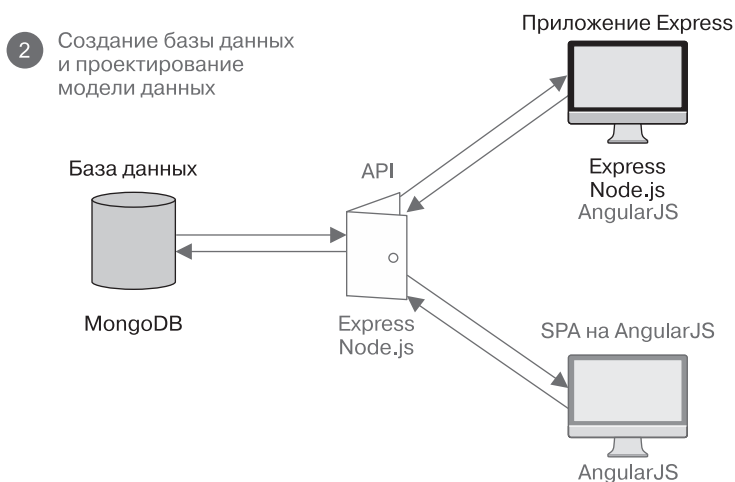


Рис. 2.16. После того как статический сайт создан, мы воспользуемся собранной информацией для проектирования модели данных и создания базы данных MongoDB

В стеке MEAN на этом шаге будем использовать MongoDB, существенно опираясь при моделировании данных на Mongoose. Сами модели данных будут при этом описаны в приложении Express. Этот шаг будет рассмотрен подробнее в главе 5.

Шаг 3. Построение API REST

После создания БД и описания моделей данных нам понадобится создать API REST таким образом, чтобы получить возможность взаимодействовать с данными посредством обращений через Интернет. Практически любому ориентированному на работу с данными приложению будет полезно наличие API, так что и этот шаг вам нужно будет сделать в большинстве проектов.

На рис. 2.17 вы можете увидеть, как именно этот шаг вписывается в процесс построения проекта в целом.

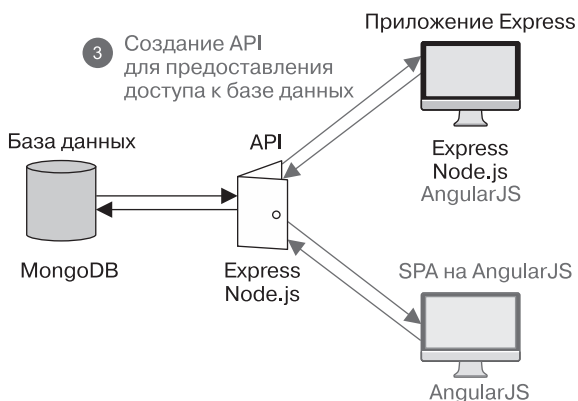


Рис. 2.17. Использование Express и Node.js для создания API, предоставляющего методы для взаимодействия с БД

В стеке MEAN этот шаг обычно выполняется с помощью Node.js и Express с изрядной долей помощи со стороны Mongoose. Мы будем чаще использовать Mongoose для сопряжения с MongoDB, чем работать с MongoDB напрямую. Этот шаг будет рассмотрен подробнее в главе 6.

Шаг 4. Использование API из приложения

Этот шаг соответствует этапу 4 процесса разработки. Именно сейчас Localhost начинает оживать. Статическое приложение из шага 1 будет модернизировано для использования API REST из шага 3 для взаимодействия с БД, созданной на шаге 2 (рис. 2.18).

Чтобы изучить все части стека и различные способы их применения, мы воспользуемся Express и Node.js для выполнения обращений к API. Если в реальном сценарии вы планируете создать большую часть приложения на AngularJS, то можно взамен этого привязать вашу БД к AngularJS. Мы рассмотрим это подробнее в главах 8–10.

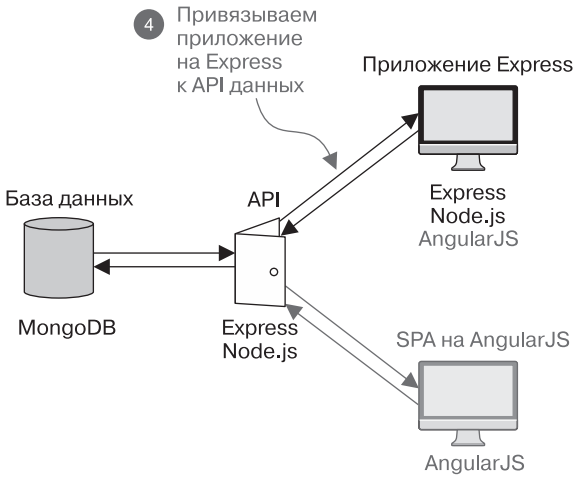


Рис. 2.18. Модернизируем статическое приложение Express, привязывая его к API данных и тем самым потенциально превращая приложение в ориентированное на работу с БД

Большую часть этого шага, как вы увидите в главе 7, мы выполним на Node.js и Express.

Шаг 5. Украшаем приложение

Шаг 5 соответствует этапу 5 процесса разработки, на котором мы добавляем к приложению дополнительные штрихи. Воспользуемся этим шагом, чтобы взглянуть на AngularJS, и увидим, как можно интегрировать компоненты AngularJS в приложение Express.

Это дополнение к архитектуре проекта отмечено на рис. 2.19.

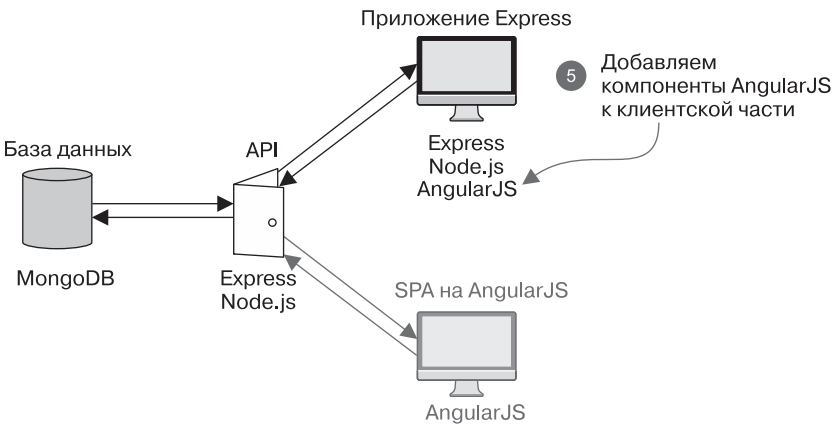


Рис. 2.19. Один из способов использования AngularJS в основанном на MEAN приложении — добавление компонентов к клиентской части в приложении Express

Этот шаг всецело посвящен знакомству с AngularJS и его использованию. Для данной цели мы, вероятнее всего, поменяем некоторые из наших настроек Node.js и Express. Этот шаг будет рассмотрен подробнее в главе 8.

Шаг 6. Переделываем код в SPA на AngularJS

На шаге 6 мы собираемся в корне изменить архитектуру путем замены приложения Express и переноса всей его логики в SPA, использующее AngularJS. В отличие от всех предыдущих шагов эти действия будут скорее не создавать что-то новое на основе предыдущих шагов, а замещать старое.

При обычном построении такой шаг — разработать приложение на Express, а затем переделать его на AngularJS — был бы странным, но для учебного процесса он отлично подходит. Мы сможем при этом сосредоточиться на AngularJS, так как уже знаем, что должно делать приложение, и у нас имеется готовый API данных.

Влияние этого изменения на архитектуру в целом демонстрирует рис. 2.20.

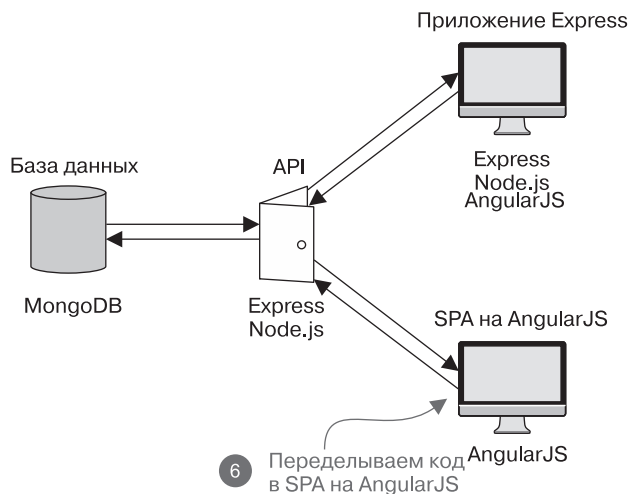


Рис. 2.20. В сущности, переписываем приложение в SPA на AngularJS

Опять-таки этот шаг сконцентрирован на AngularJS и будет рассмотрен в главах 9 и 10.

Шаг 7. Добавляем аутентификацию

На шаге 7 мы добавим в приложение функцию аутентификации и предоставим пользователям возможность зарегистрироваться и войти на сайт, а также посмотрим, как применять данные пользователей во время того, как они работают с приложением. Мы будем основываться на всем, что сделали до сих пор, и добавим аутентификацию к SPA на AngularJS. В качестве части этого мы сохраним инфор-

мацию о пользователе в БД и обезопасим некоторые конечные точки, чтобы с ним могли работать только аутентифицированные пользователи.

Рисунок 2.21 демонстрирует, с чем мы будем иметь дело с точки зрения архитектуры.

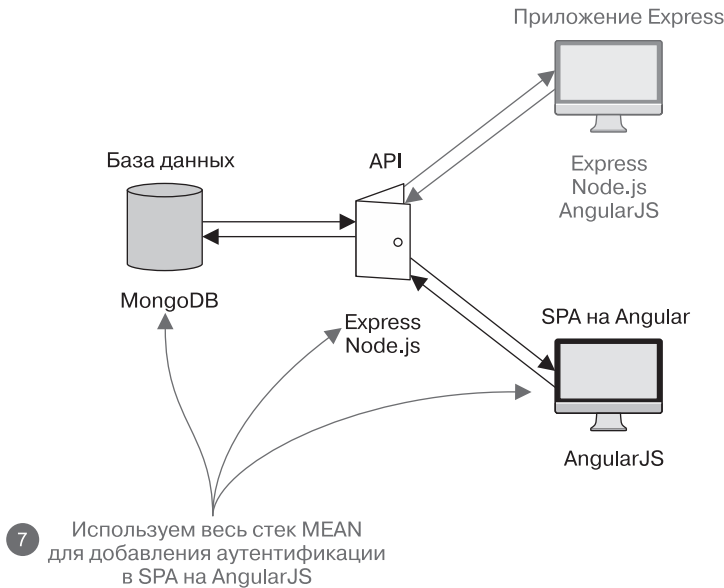


Рис. 2.21. Используем весь стек MEAN для добавления аутентификации в SPA на AngularJS

На этом шаге будем работать со всеми технологиями стека MEAN, который рассматривается в главе 11.

2.6. Аппаратная архитектура

Никакое обсуждение архитектуры не было бы полным без раздела об аппаратном обеспечении. Вы уже видели, каким образом можно свести воедино все программное обеспечение и компоненты кода, но какое аппаратное обеспечение понадобится для работы всего этого?

2.6.1. Аппаратные средства для разработки приложений

Хорошая новость заключается в том, что не потребуется ничего особенного для работы стека. Для разработки приложения на стеке MEAN достаточно обычного ноутбука или даже виртуальной машины (VM). Все компоненты стека можно установить в Windows, MacOS X и большинстве дистрибутивов Linux.

Я успешно разрабатывал приложения на ноутбуках под управлением Windows и MacOS X, а также на виртуальных машинах Ubuntu. Я предпочитаю родную среду разработки на OS X, но знаю многих приверженцев использования виртуальных машин с Linux.

Если есть локальная сеть и несколько различных серверов, можно распределить выполнение приложения по ним. Например, можно использовать одну машину как сервер БД, другую — для API REST, а третью — для самого кода главного приложения. До тех пор пока серверы могут взаимодействовать друг с другом, это не представляет собой проблемы.

2.6.2. Аппаратное обеспечение для промышленной эксплуатации

Подход к промышленной аппаратной архитектуре не сильно отличается от подхода к стандартному аппаратному обеспечению для разработки. Основное отличие — промышленное аппаратное обеспечение обычно более мощное и в нем открыт доступ в Интернет, для того чтобы можно было получать запросы от широкой аудитории.

Начальный размер

Вполне допустимо разместить и выполнять все части вашего приложения на одном и том же сервере. Основная схема приведена на рис. 2.22.

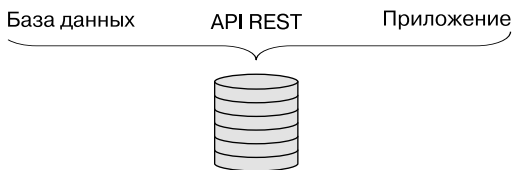


Рис. 2.22. Простейшая аппаратная архитектура — все располагается на одном сервере

Такая архитектура подходит для приложений с небольшими объемами трафика, но обычно не рекомендуется при увеличении приложения, чтобы приложению и БД не приходилось бороться за одни и те же ресурсы.

Рост приложения: отдельный сервер БД

Часто БД одной из первых переносится на отдельный сервер. Так что теперь у вас будет два сервера: один для приложения и второй — для БД. Этот подход проиллюстрирован на рис. 2.23.

Это весьма распространенная модель, особенно если вы решили использовать для своего хостинга провайдера платформы как услуги (PaaS). В книге мы будем применять этот подход.

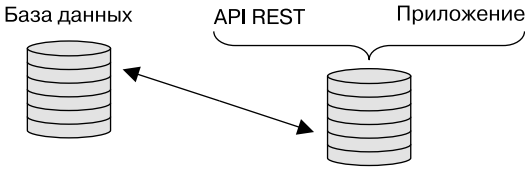


Рис. 2.23. Распространенный подход к аппаратной архитектуре: один сервер для выполнения кода приложения и API и второй — для БД

Масштабируемся

Аналогично тому, что мы говорили по поводу аппаратного обеспечения для разработки, можно выделить отдельные серверы для различных частей: сервер БД, сервер API и сервер приложения. Это даст возможность обрабатывать большие объемы трафика за счет распределения нагрузки по трем серверам, как показано на рис. 2.24.

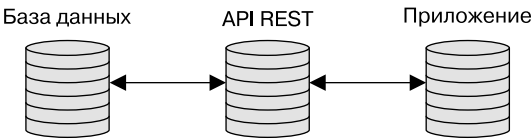


Рис. 2.24. Разделенная архитектура с использованием трех серверов: по одному для БД, API и выполнения кода приложения

Но этим дело не заканчивается. Если ваш трафик слишком велик для трех серверов, можно использовать несколько экземпляров, или кластеров, этих серверов (рис. 2.25).

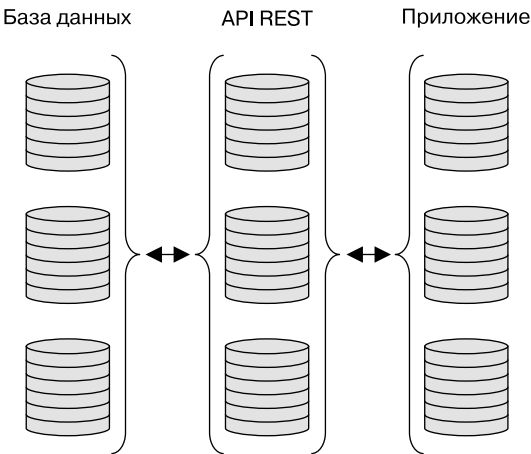


Рис. 2.25. Можно масштабировать приложения MEAN посредством использования кластеров серверов для каждой из частей всего вашего приложения

2.7. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Обычную архитектуру стека MEAN со SPA на AngularJS, использующим API REST, на основе Node.js, Express и Mongo.
- ❑ Соображения, которые необходимо учитывать при решении о том, создавать SPA или нет.
- ❑ Проектирование гибкой архитектуры на основе стека MEAN.
- ❑ Рекомендуемое решение — создание API для предоставления доступа к слою данных.
- ❑ Шаги, которые мы предпримем для создания учебного приложения Loc8r.
- ❑ Аппаратные архитектуры для разработки и промышленной эксплуатации.

В главе 3 мы начнем свой путь с создания проекта Express, который будет связывать все воедино.

ЧАСТЬ II

Создание веб-приложения на платформе Node

Node.js представляет собой основу любого приложения на стеке MEAN, так что именно с него и начнем. На протяжении части II мы создадим ориентированное на работу с данными веб-приложение, использующее Node.js, Express и MongoDB. По ходу дела мы изучим каждую из технологий, доведя приложение до состояния полностью функционирующего веб-приложения на Node.

В главе 3 мы начнем всерьез работать над созданием и настройкой основанного на MEAN проекта, знакомясь по пути с Express еще до того, как значительно глубже разберемся с ним в главе 4, создав статическую версию приложения. Хорошо поняв приложение, в главе 5 мы подключим MongoDB и Mongoose для проектирования и создания необходимой нам модели данных.

Хорошая архитектура приложения должна включать API данных, вместо того чтобы сцеплять с логикой приложения взаимодействия с БД. В главе 6 мы создадим API REST с помощью Express, MongoDB и Mongoose, прежде чем объединить его с приложением путем использования API REST из нашего статического приложения. В конце части II у нас уже будет ориентированный на работу с данными сайт, использующий Node.js, MongoDB и Express, а также полнофункциональный API REST.

Глава 3

Создание и настройка проекта на стеке MEAN

В этой главе:

- ❑ управление зависимостями с помощью файла `package.json`;
- ❑ создание и настройка проектов Express;
- ❑ настройка среды MVC;
- ❑ добавление Twitter Bootstrap для работы с макетами;
- ❑ публикация приложения по URL в Интернете, использование Git и Heroku.

Теперь мы действительно готовы начать разработку и в этой главе всерьез приступим к созданию приложения. Как вы можете помнить из глав 1 и 2, на протяжении книги мы собираемся создать приложение под названием `Loc8r`. Это будет учитывающее местоположение пользователя веб-приложение, выводящее списки расположенных поблизости мест с Wi-Fi и приглашающее пользователей войти и оставить отзывы.

В стеке MEAN Express — это фреймворк веб-приложений Node. Вместе Node и Express составляют основу стека, так что с них и начнем. Рисунок 3.1 демонстрирует, на чем мы — в смысле построения архитектуры приложения — сосредоточим свое внимание в этой главе. Мы выполним две вещи:

- ❑ создадим проект и инкапсулируем приложение Express, которое обеспечит оболочку для всего остального, кроме БД;
- ❑ настроим основное приложение Express.

ПОЛУЧЕНИЕ ИСХОДНОГО КОДА

Исходный код для нашего приложения находится на GitHub по адресу <http://www.github.com/simonholmes/getting-MEAN>. У каждой главы, в которой будут значительные изменения, имеется своя ветвь. Я советую вам по ходу книги самостоятельно создавать исходный код с нуля, но если хотите, то можете получить код, который мы напишем в данной главе, с GitHub, из ветви `chapter-03`. Выполните в терминале в новом каталоге следующую команду для его клонирования, если у вас уже установлен Git:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN.git
```

Это даст вам копию кода, хранящуюся на GitHub. Для запуска приложения понадобится установить определенные зависимости с помощью следующих команд:

```
$ cd getting-MEAN
```

```
$ npm install
```

Не беспокойтесь, если что-то из описанного вам пока непонятно или если часть команд не работает. На протяжении данной главы по ходу работы мы будем осваивать эти технологии.

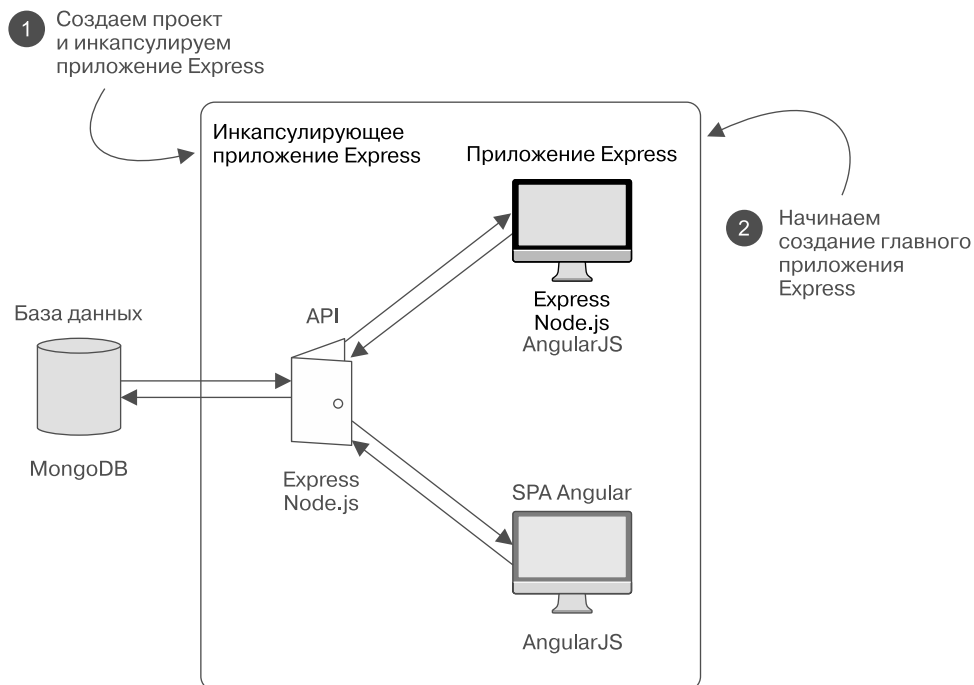


Рис. 3.1. Создание инкапсулирующего приложения Express и начало настройки главного приложения Express

Мы начнем с закладки фундамента, изучая Express на предмет выяснения возможностей управления зависимостями и модулями с помощью `npm` и файла `package.json`. Эта базовая информация понадобится нам для начала серьезной работы над проектом Express и его настройки.

Прежде чем начать что-то делать по-настоящему, убедимся, что на вашей машине установлено все необходимое. После этого рассмотрим создание проектов Express из командной строки и различные параметры, которые можно при этом задать.

Express великолепен, но его можно сделать еще лучше — и лучше с ним познакомиться, — слегка с ним повозившись и кое-что поменяв. Для этого нам понадобится рассмотреть архитектуру MVC (Model — View — Controller, модель — представление — контроллер). Здесь мы слегка «заглянем под капот» Express и посмотрим, что он делает, внося в него изменения для обеспечения прозрачности настроек MVC.

После настройки Express нужным образом подключим проект Twitter Bootstrap и сделаем сайт адаптивным, поменяв шаблоны Jade. А в конце этой главы мы разместим обновленное адаптивное приложение MVC Express по URL в Интернете с помощью Heroku и Git.

3.1. Краткий обзор Express, Node и npm

Как уже упоминалось, Express — фреймворк веб-приложений для Node. Проще говоря, приложение Express — всего лишь приложение Node, которому случилось использовать Express в качестве фреймворка. А `npm`, как вы можете помнить из главы 1, — система управления пакетами, устанавливаемая вместе с Node. Она дает вам возможность скачивать модули или пакеты Node для расширения функциональности вашего приложения.

Но как работать с ними вместе и как вы на самом деле будете их использовать? Ключ к решению этой загадки — файл `package.json`.

3.1.1. Описание пакетов с помощью `package.json`

В корневом каталоге каждого приложения Node должен быть файл под названием `package.json`. В нем содержатся различные метаданные о проекте, включая пакеты, от которых зависит его выполнение. Листинг 3.1 демонстрирует пример файла `package.json`, который можно найти в корне нового проекта Express.

Листинг 3.1. Пример package.json из нового проекта Express

```

{
  "name": "application-name",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.9.0",
    "body-parser": "~1.8.1",
    "cookie-parser": "~1.3.3",
    "morgan": "~1.3.0",
    "serve-favicon": "~2.1.3",
    "debug": "~2.0.0",
    "jade": "~1.6.0"
  }
}

```

Различные метаданные,
описывающие приложение

Зависимости пакетов,
необходимые для запуска
приложения

Это файл целиком, так что он не особенно сложен. В начале файла находятся различные метаданные, за которыми следует раздел зависимостей. В инсталляции проекта Express по умолчанию зависимостей немного. Express сам по себе имеет модульную структуру, так что вы можете добавлять компоненты или обновлять их по отдельности.

Работа с версиями зависимостей в package.json

Рядом с названием каждой зависимости имеется номер версии, которую будет использовать приложение. Обратите внимание на то, что перед ними стоит префикс ~.

Взглянем на описание зависимости для Express 4.9.0. Он задает конкретную версию на трех уровнях:

- ❑ старшая версия (4);
- ❑ младшая версия (9);
- ❑ патч-версия (0).

Задание префикса ~ для всего номера версии равноценно замене патч-версии джокерным символом, означающим, что приложение будет использовать самую свежую из доступных патч-версий. Это считается рекомендуемым решением, так как патчи должны содержать только исправления, которые никак не повлияют на приложение. Но различные старшие и младшие версии могут включать изменения, вызывающие проблемы с приложением, так что желательно избегать использования их самых свежих версий.

3.1.2. Установка зависимостей Node с помощью npm

У любого приложения или модуля Node могут быть зависимости, перечисленные в файле `package.json`. Их установка очень проста и выполняется единообразно независимо от приложения или модуля.

При использовании командной строки терминала в том же каталоге, где находится файл `package.json`, необходимо всего лишь выполнить следующую команду:

```
$ npm install
```

Она указывает npm установить все перечисленные в файле `package.json` зависимости. При ее выполнении npm загрузит все перечисленные в качестве зависимостей пакеты и установит их в специальный каталог `node_modules` приложения. Рисунок 3.2 иллюстрирует три ключевые части этого процесса.

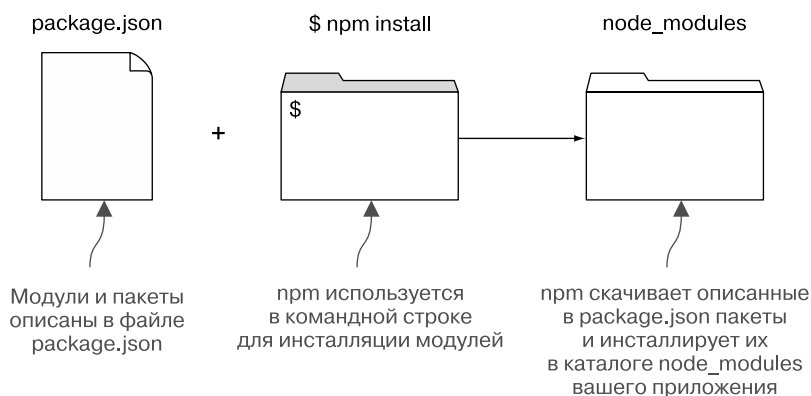


Рис. 3.2. Модули npm, описанные в `package.json`, скачиваются и устанавливаются в каталог `node_modules` приложения при запуске в терминале команды `npm install`

npm установит каждый пакет в его собственный подкаталог, поскольку каждый из них фактически сам является пакетом Node. Поэтому у каждого из этих пакетов также имеется его собственный файл `package.json`, задающий метаданные, включая зависимости. Зачастую у пакетов имеется и собственный каталог `node_modules`. Вам не нужно заботиться о ручной установке всех этих вложенных зависимостей, поскольку это сделает исходная команда `npm install`.

Добавление пакетов в существующий проект

Вряд ли у вас с самого начала будет полный список зависимостей для проекта. Гораздо вероятнее, что вы начнете с нескольких ключевых, о которых известно, что они вам понадобятся, и, возможно, каких-то всегда используемых вами в работе.

С помощью `npm` можно очень легко добавить дополнительные пакеты в приложение, когда вам это понадобится. Нужно просто найти название пакета, который вы хотите установить, и открыть командную строку в том же каталоге, где находится файл `package.json`. А затем выполнить вот такую простую команду:

```
$ npm install --save package-name
```

По этой команде `npm` скачает и установит новый пакет в каталог `node_modules`. Флаг `--save` указывает `npm` добавить этот пакет в список зависимостей в файле `package.json`.

Обновление пакетов до более поздних версий

`npm` скачивает и заново устанавливает существующие пакеты только при обновлении до новой версии. При вводе команды `npm install` `npm` обходит все зависимости и проверяет:

- ❑ указанную в файле `package.json` версию;
- ❑ последнюю патч-версию на `npm` (если вы указали префикс `~`);
- ❑ версию, установленную в каталоге `node_modules` (если таковая имеется).

Если установленная версия отличается от описания в файле `package.json`, то `npm` скачает и установит версию, указанную в `package.json`. Аналогично, если вы указали джокерный символ для патча и доступна более поздняя патч-версия, то `npm` скачает и установит ее вместо предыдущей версии.

Вооруженные этими знаниями, мы можем начать создание первого проекта Express.

3.2. Создание проекта Express

У любого путешествия должен быть отправной пункт, в случае разработки приложения на основе стека MEAN это создание нового проекта Express. Необходимо, чтобы на вашей машине для разработки были установлены следующие пять главных объектов:

- ❑ Node и `npm`;
- ❑ установленный глобально генератор приложений Express;
- ❑ Git;
- ❑ Heroku;
- ❑ интерфейс командной строки (command-line interface (CLI)) или терминал.

3.2.1. Установка отдельных частей

Если у вас еще не установлены Node, npm или генератор приложений Express, обратитесь к приложению А для получения инструкций и ссылок на ресурсы в Интернете. Их можно установить в Windows, Mac OS X и большинстве распространенных дистрибутивов Linux.

К концу данной главы мы также успеем воспользоваться Git для управления исходными текстами нашего приложения Loc8r и разместим его по реальному URL с помощью Heroku. Пожалуйста, просмотрите приложение Б, в котором описываются настройки Git и Heroku.

В зависимости от вашей операционной системы вам может понадобиться установить новый CLI или терминал. Из приложения Б можете узнать о том, касается это вас или нет.

ПРИМЕЧАНИЕ

На протяжении данной книги я часто называю CLI терминалом. Так что, когда я говорю: «Выполните эту команду в терминале», просто выполните ее в том CLI, который вы используете. Команды терминала, которые в данной книге приводятся в качестве фрагментов кода, начинаются со знака \$. Вводить его в терминале не следует, он лишь указывает, что речь идет об операторе командной строки. Например, при использовании команды `$ echo 'Добро пожаловать в стек MEAN'`, вам нужно ввести только `echo 'Добро пожаловать в стек MEAN'`.

3.2.2. Проверка установленных пакетов

Для создания нового проекта Express требуется, чтобы были установлены Node и npm, а также глобально установлен генератор приложений Express. Проверить их наличие можно путем проверки номера версии в терминале с помощью следующих команд:

```
$ node --version
$ npm --version
$ express --version
```

Каждая из этих команд должна вывести в терминал номер версии. Если какая-либо из них завершается неудачей, обратитесь к приложению А за инструкциями по установке.

3.2.3. Создание каталога проекта

Допустим, что все хорошо, и начнем с создания на вашей машине нового каталога с названием `loc8r`. Он может располагаться на Рабочем столе, в вашей папке Документы, в папке Dropbox — это неважно, если у вас есть к этому каталогу полный доступ для чтения и записи.

Лично я выполняю немало своих разработок на основе стека MEAN в папках Dropbox, благодаря чему они сразу же подвергаются резервному копированию и доступны на любой из моих машин. Если вы работаете в корпоративной среде, это может оказаться для вас неподходящим, так что создавайте каталог там, где вам удобнее.

3.2.4. Конфигурирование установки Express

Проект Express устанавливается из командной строки, а параметры конфигурации передаются с помощью параметров используемой вами команды. Если работать в командной строке вам непривычно, не беспокойтесь: то, что мы будем делать в этой книге, несложно и легко запоминается. Как только вы начнете использовать командную строку, вероятно, вы ее полюбите за обеспечиваемую ею быстроту некоторых операций.

Например (пока что не выполняйте этого), вы можете установить Express в каталог с помощью простой команды:

```
$ express
```

Эта команда выполнит установку фреймворка с настройками по умолчанию в текущий каталог. Возможно, это неплохо для начала, но давайте в первую очередь взглянем на некоторые параметры конфигурации.

Параметры конфигурации при создании проекта Express

Что можно настроить при создании проекта Express? При создании проекта Express подобным образом можно задать следующие параметры:

- какие шаблонизаторы HTML использовать;
- какой препроцессор CSS использовать;
- включать ли поддержку сеансов.

Установка по умолчанию использует шаблонизатор Jade, но во фреймворке не будет поддержки ни предварительной обработки CSS, ни сеансов. Вы можете задать несколько различных параметров, как показано в табл. 3.1.

Таблица 3.1. Конфигурационные параметры командной строки при создании нового проекта Express

Команда конфигурации	Производимое действие
--css less stylus	Добавляет в проект препроцессор CSS, Less или Stylus в зависимости от того, что вы укажете в команде
--ejs	Меняет шаблонизатор HTML с Jade на EJS
--jshtml	Меняет шаблонизатор HTML с Jade на JsHtml
--hogan	Меняет шаблонизатор HTML с Jade на Hogan

Например (и это отнюдь не то, что мы будем тут делать), если вы хотите создать проект, использующий препроцессор CSS Less и шаблонизатор Hogan, то вам нужно будет выполнить в терминале следующую команду:

```
$ express --css less --hogan
```

Для упрощения проекта мы не будем использовать в нем предварительную обработку CSS, так что можем обойтись предоставляемым по умолчанию обычным CSS. Но нам все же понадобится шаблонизатор, так что давайте рассмотрим имеющиеся варианты.

Различные шаблонизаторы

При использовании Express подобным образом имеется четыре возможных варианта шаблонизаторов: Jade, EJS, JsHtml и Hogan. Основной технологический процесс шаблонизатора таков: вы создаете шаблон HTML, включая заполнители для данных, а затем передаете ему какие-то данные. Далее шаблонизатор собирает все это вместе для создания получаемой браузером итоговой разметки HTML.

У всех перечисленных шаблонизаторов есть свои достоинства и особенности, так что хорошо, если у вас уже есть любимчик. В этой книге мы будем использовать Jade. Он обладает большими возможностями и предоставляет всю функциональность, которая только может нам понадобиться. Поскольку Jade — шаблонизатор по умолчанию в Express, вы обнаружите, что он использован в большинстве примеров и проектов в Интернете, так что познакомиться с ним отнюдь не помешает. Минималистичный стиль Jade делает его идеальным для примеров кода в книге!

Быстрый взгляд на Jade

Jade по сравнению с другими шаблонизаторами необычен тем, что он не содержит в шаблонах тегов HTML. Вместо этого Jade применяет более минималистичный подход, используя имена тегов, отступы и вдохновленный CSS метод ссылок для

описания структуры HTML. Исключением из этого является тег `<div>`. В силу его распространенности, если имя тега будет опущено, Jade станет считать, что вы имели в виду `<div>`.

ПОДСКАЗКА

Шаблоны Jade необходимо структурировать с помощью пробелов, а не символов табуляции.

Следующий фрагмент кода демонстрирует простой пример шаблона Jade и скомпилированный результат:

<pre>#banner.page-header h1 Моя страница p.lead Добро пожаловать на мою страницу</pre>	<div style="border-left: 1px solid black; padding-left: 10px;"> Шаблон Jade, не содержащий тегов HTML </div>
<pre><div id="banner" class="page-header"> <h1>Моя страница</h1> <p class="lead">Добро пожаловать на мою страницу</p> </div></pre>	<div style="border-left: 1px solid black; padding-left: 10px;"> Скомпилированный результат представляет собой легко узнаваемый HTML </div>

С первых строк ввода и вывода можно увидеть, что:

- если название тега не указывается, то создается `<div>`;
- `#banner` в Jade превращается в `id="banner"` в HTML;
- `.page-header` в Jade превращается в `class="page-header"` в HTML.

Имея за плечами эти начальные знания, мы можем создать проект.

3.2.5. Создание проекта Express и его проверка

Итак, мы узнали основную команду для создания проекта Express и решили использовать параметры настройки по умолчанию, так что давайте сделаем это — создадим новый проект. В подразделе «Проверка установленных пакетов» выше вы должны были создать каталог `loc8r`. Перейдите в этот каталог в терминале и выполните следующую команду:

```
$ express
```

Она создаст группу каталогов и файлов внутри каталога `loc8r`, формирующих основу приложения `Loc8r`. Но еще не все готово. Следующее, что вам нужно сделать, — установить зависимости. Как вы помните, это делается просто выпол-

нением следующей команды из командной строки терминала в том же каталоге, где находится файл `package.json`:

```
$ npm install
```

Как только вы выполните ее, то увидите, что окно терминала наполнится всеми скачиваемыми ею файлами. После окончания выполнения приложение будет готово к пробному запуску.

Пробный запуск проекта

Запуск приложения — это легко. Скоро мы рассмотрим лучший способ сделать это, но если вы так же нетерпеливы, как и я, то захотите прямо сейчас увидеть, что сделанное вами до сих пор работает.

Выполните в терминале, в каталоге `loc8r`, следующую команду:

```
$ npm start
```

Вы должны получить примерно такое подтверждение:

```
loc8r@0.0.0 start /path/to/your/application/folder
```

Это значит, что приложение Express запущено! Понаблюдать его в работе можно, открыв браузер и перейдя по адресу `localhost:3000`. Надеюсь, что вы увидите что-то напоминающее скриншот, приведенный на рис. 3.3.

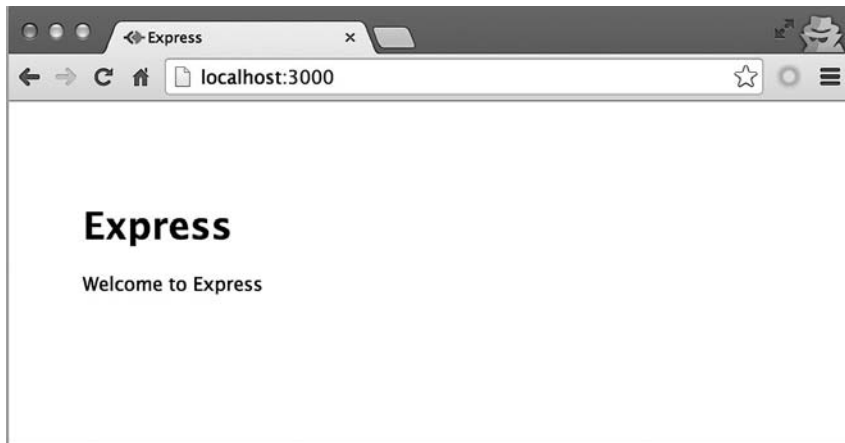


Рис. 3.3. Целевая страница пустого проекта Express

Правда, надо сказать, что это пока не слишком впечатляющее зрелище, но привести приложение Express в рабочее состояние, вплоть до состояния функционирования через браузер, было совсем не сложно, не так ли?

Если вы перейдете обратно в терминал, то должны увидеть несколько журнальных записей, подтверждающих, что были запрошены страница и таблица стилей. Чтобы разобраться с Express получше, взглянем на эти записи поближе.

Как Express обрабатывает запросы

Целевая страница Express по умолчанию довольно проста. В ней есть немного HTML-кода, в котором определенный текстовый контент передается в виде данных с помощью маршрутизации Express. Имеется также CSS-файл. Журнальные сообщения в терминале должны подтверждать тот факт, что именно это было запрошено у Express и возвращено браузеру.

О ПРОМЕЖУТОЧНОМ ПРОГРАММНОМ ОБЕСПЕЧЕНИИ EXPRESS

Посередине файла `app.js` имеется набор строк, начинающийся с `app.use`. Они известны как промежуточное программное обеспечение (ПО) (`middleware`). Когда приложению поступает запрос, он по очереди проходит через каждый элемент промежуточного ПО. Каждый элемент промежуточного ПО может делать или не делать что-либо с запросом, но последний всегда передается следующему элементу до тех пор, пока не достигнет логики самого приложения, которая возвратит ответ.

Рассмотрим, например, команду `app.use(express.cookieParser())`. Промежуточное ПО получает входящий запрос, выделяет путем синтаксического разбора всю информацию о `cookie`-файлах, а затем прикрепляет данные к запросу таким образом, чтобы удобно было ссылаться на них в коде контроллера.

Пока что вам не нужно знать, что именно делает каждый элемент промежуточного ПО, но вполне возможно, что при создании приложений вы сами будете добавлять что-либо в соответствующий список.

Все запросы к серверу Express проходят через промежуточное ПО, описанное в файле `app.js` (см. врезку «О промежуточном программном обеспечении Express»). Помимо прочего, существует элемент промежуточного ПО по умолчанию, выполняющий поиск путей к статистическим файлам. Когда промежуточное ПО находит соответствие пути и файла, Express возвращает результат асинхронно, гарантируя тем самым, что процесс Node не занят этой операцией, блокируя таким образом другие операции. После того как запрос пройдет все промежуточное ПО, Express пытается найти соответствие пути запроса и заданного пути. Мы рассмотрим это подробнее дальше в данной главе.

Рисунок 3.4 иллюстрирует этот поток выполнения на примере домашней страницы Express по умолчанию, приведенной на рис. 3.3.

Поток выполнения на рис. 3.4 показывает выполнение отдельных запросов и различия в их обработке Express. Оба запроса сначала проходят через промежуточное ПО, но результаты этого процесса сильно различаются.

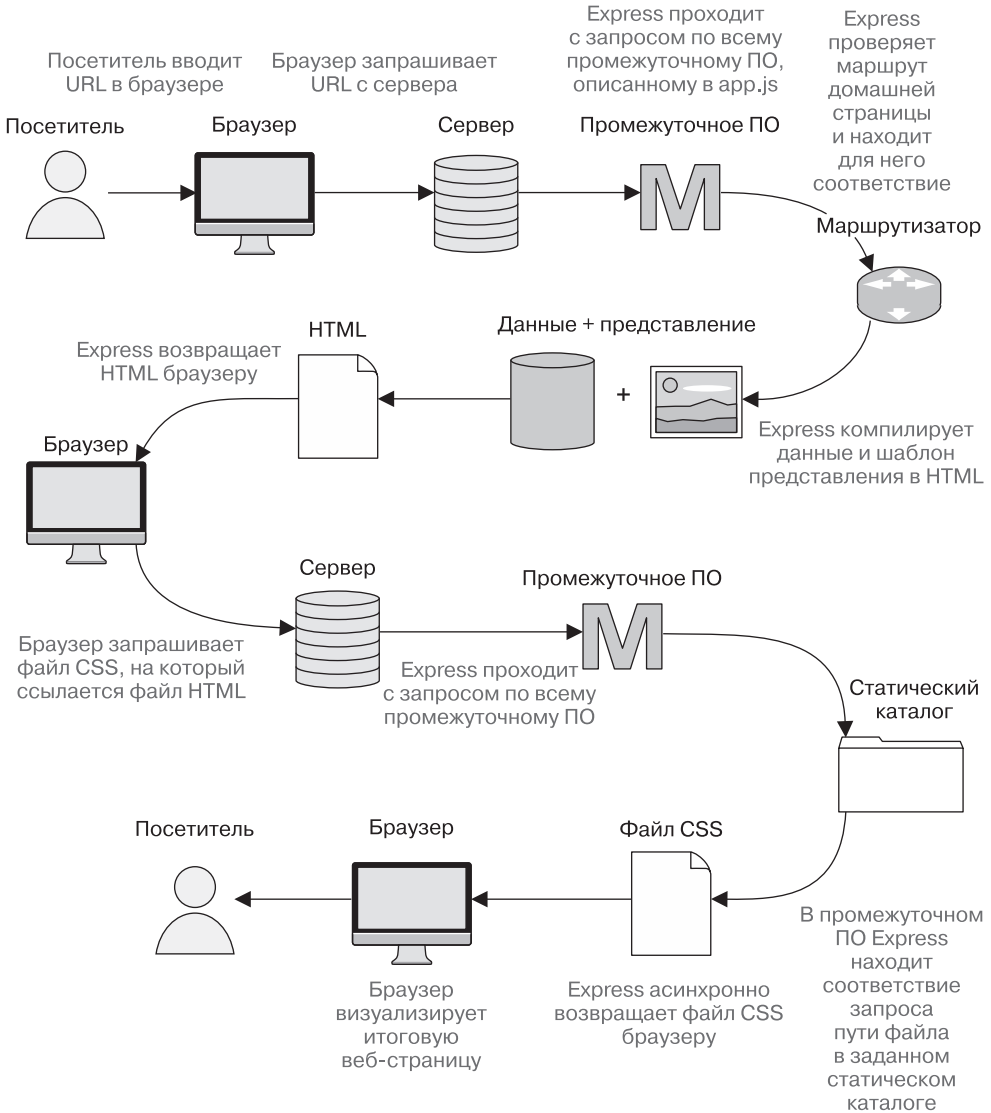


Рис. 3.4. Ключевые взаимодействия и процессы, через которые проходит Express при ответе на запрос к целевой странице по умолчанию. Node обрабатывает страницу HTML для компиляции данных и шаблона представления, а файл CSS выдается асинхронно из статического каталога

3.2.6. Перезапуск приложения

Приложение Node компилируется до запуска, так что если вы внесете изменения в код приложения во время его выполнения, они не будут подхвачены до остановки и перезапуска процесса Node. Обратите внимание на то, что это справедливо только для кода приложения: шаблоны Jade, файлы CSS и клиентский JavaScript могут обновляться на лету.

Перезапуск процесса Node — двухшаговая процедура. Сначала нужно остановить запущенный процесс. Это можно сделать в терминале, нажав **Ctrl+C**. Затем нужно в терминале запустить процесс снова с помощью той же команды, что и ранее, — `npm start`.

Это не выглядит сложным, но когда вы разрабатываете и тестируете реальное приложение, необходимость выполнять эти два шага каждый раз, когда нужно проверить обновление, начинает довольно сильно раздражать. К счастью, существует способ получше.

3.2.7. Автоматический перезапуск приложения с помощью NODEMON

Существуют сервисы, разработанные специально для мониторинга кода приложения, перезапускающие процесс при обнаружении изменений. Один из таких сервисов, который мы и будем использовать в данной книге, — это *nodemon*. Он просто служит адаптером для приложения Node и, кроме мониторинга изменений, никак в работу не вмешивается.

Для использования *nodemon* вам сначала нужно установить его глобально аналогично тому, как вы поступили с Express. Это выполняется в терминале с помощью `npm`:

```
$ npm install -g nodemon
```

После завершения установки вы сможете использовать *nodemon* где только пожелаете. Пользоваться им очень просто. Вместо того чтобы набирать `node` для запуска приложения, вы набираете `nodemon`. Так что, убедившись, что вы находитесь в каталоге `loc8r` в терминале и что вы остановили процесс Node, если он все еще работал, введите следующую команду:

```
$ nodemon
```

После этого вы должны увидеть, что в терминал было выведено еще несколько строк, подтверждающих, что *nodemon* запущен и что он выполнил `node ./bin/www`. Если вы вернетесь в браузер и обновите страницу, то увидите, что приложение по-прежнему работает.

ПРИМЕЧАНИЕ

Nodemon предназначен только для облегчения рабочего процесса в вашей среде разработки, его не следует использовать в реальной среде в ходе промышленной эксплуатации.

3.3. Модифицируем Express для MVC

Что представляет собой архитектура MVC? MVC расшифровывается как Model — View — Controller (модель — представление — контроллер), и эта архитектура нацелена на отделение друг от друга данных (модели), отображения (представления) и логики приложения (контроллера). Цель такого разделения — уничтожить тесные связи между компонентами, что теоретически должно сделать код более удобным для сопровождения и повторного использования. Дополнительно мы получаем бонус, заключающийся в том, что эти компоненты отлично подходят для используемого нами подхода ускоренной разработки прототипа и предоставляют возможность концентрировать усилия на одном аспекте по мере обсуждения каждой из частей стека MEAN.

Существует немало книг, посвященных нюансам MVC, но здесь мы не будем углубляться в детали. Остановимся на высокоуровневом обсуждении MVC и рассмотрим его использование вместе с Express для создания приложения Loc8r.

3.3.1. Обзор MVC с высоты птичьего полета

Большинство создаваемых вами приложений и сайтов будут спроектированы так, чтобы получать входящий запрос, проделывать с ним что-либо и возвращать ответ. Упрощенно этот цикл в архитектуре MVC работает примерно так.

1. В приложение поступает запрос.
2. Запрос маршрутизируется к контроллеру.
3. Контроллер, если необходимо, выполняет запрос к модели.
4. Модель возвращает ответ контроллеру.
5. Контроллер отправляет ответ представлению.
6. Представление отправляет ответ инициатору первоначального запроса.

В действительности в зависимости от настроек контроллер может фактически компилировать представление перед отправкой ответа посетителю. Результат,

однако, остается тем же самым, так что держите перед глазами этот простой поток выполнения в качестве визуальной картины происходящего в приложении `loc8r` (рис. 3.5).

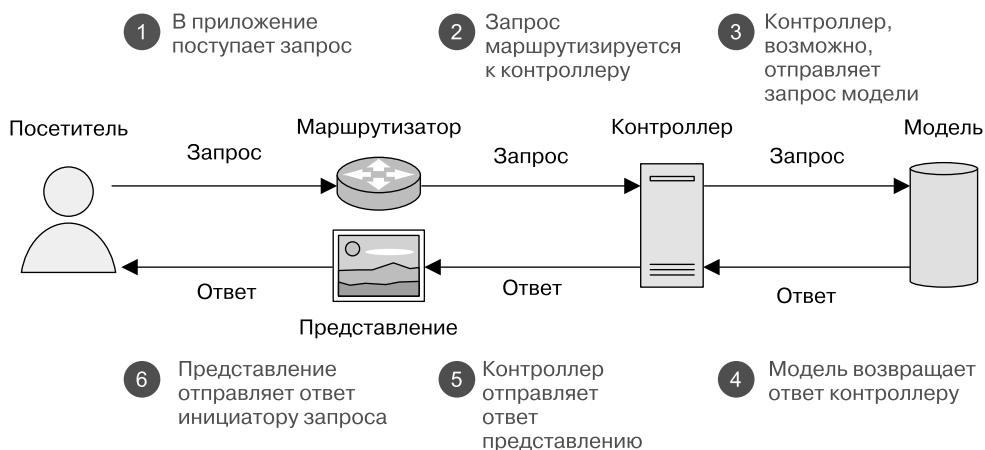


Рис. 3.5. Поток выполнения «запрос — ответ» простейшей архитектуры MVC

Рисунок 3.5 демонстрирует отдельные части архитектуры MVC и их взаимосвязи. Он также иллюстрирует то, что, помимо модели, представления и контроллера, необходим механизм маршрутизации. Теперь, когда вы увидели, как должен работать основной поток выполнения приложения `loc8r`, наступает время изменить настройки Express, чтобы воплотить это в жизнь.

3.3.2. Изменение структуры каталогов

Если вы заглянете в только что созданный проект Express в каталоге `loc8r`, то увидите структуру файлов, включающую каталог представлений (`views`) и даже каталог маршрутов (`routes`), но не найдете никаких упоминаний о моделях и контроллерах. Вместо того чтобы захламлять корневой уровень приложения какими-то новыми каталогами, сохраним аккуратный вид с одним каталогом для всей архитектуры MVC. Выполните следующие три шага.

1. Создайте новый каталог `app_server`.
2. В каталоге `app_server` создайте два новых каталога: `models` и `controllers`.
3. Переместите каталоги `views` и `routes` из корневого каталога приложения в каталог `app_server`.

Рисунок 3.6 иллюстрирует эти изменения и демонстрирует структуру каталогов до и после модификации.

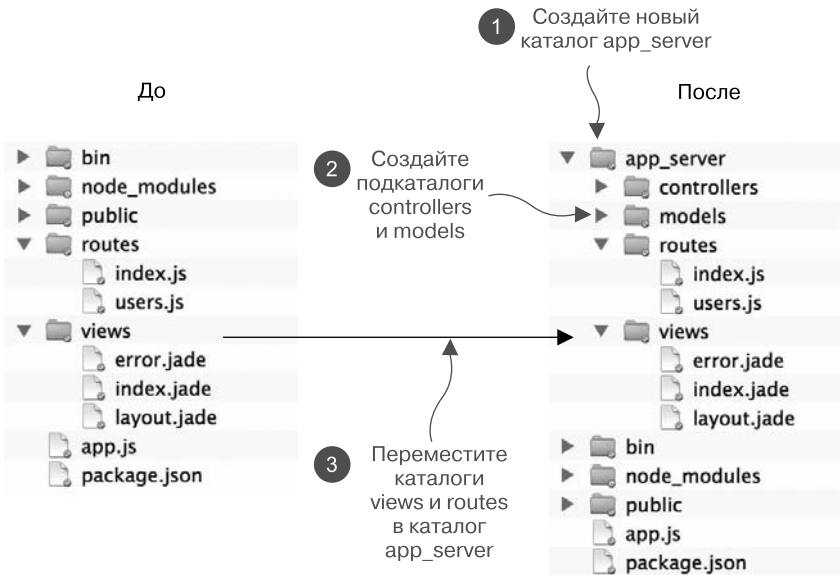


Рис. 3.6. Изменение структуры каталогов проекта Express в архитектуре MVC

Теперь у нас в приложении сделаны действительно понятные настройки MVC, что в дальнейшем облегчает разделение ответственности. Но если вы сейчас попытаетесь запустить приложение, то обнаружите, что оно не работает, ведь мы только что его «поломали». Так что теперь исправим поломанное. Express не знает, что мы добавили новые каталоги, и не имеет ни малейшего представления о том, для чего мы хотим их использовать. Поэтому нужно ему об этом сообщить.

3.3.3. Новые каталоги `views` и `routes`

Первое, что нам нужно сделать, — сообщить Express о перемещении каталогов `views` и `routes`, поскольку он будет искать уже не существующие каталоги и файлы.

Новое расположение каталога `views`

Express будет искать файлы в каталоге `/views`, но мы только что переместили его в каталог `/app_server/views`. Изменить это очень просто. Найдите в файле `app.js` следующую строку:

```
app.set('views', path.join(__dirname, 'views'));
```

и поменяйте ее на следующую (изменения выделены полужирным шрифтом):

```
app.set('views', path.join(__dirname, 'app_server', 'views'));
```

Наше приложение все еще не работает, поскольку мы переместили и каталог `routes`, так что сообщим Express и об этом.

Новое расположение каталога `routes`

Express будет искать файлы в `/routes`, но мы только что переместили его в `/app_server/routes`. Изменить это тоже очень просто. Найдите в файле `app.js` следующие строки:

```
var routes = require('./routes/index');  
var users = require('./routes/users');
```

и поменяйте их на такие (изменения выделены полужирным шрифтом>):

```
var routes = require('./app_server/routes/index');  
var users = require('./app_server/routes/users');
```

Если вы сохраните эти изменения и запустите приложение снова, то обнаружите, что мы все починили и приложение опять работает!

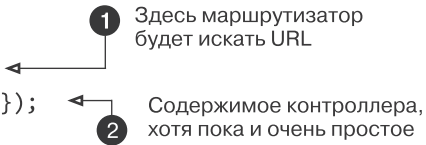
3.3.4. Отделение контроллеров от маршрутов

При настройках Express по умолчанию контроллеры практически являются частью маршрутов, но нам хотелось бы отделить их. Контроллеры должны управлять логикой приложения, а маршруты — устанавливать соответствие URL-запросов контроллерам.

Понятие об описании маршрутов

Чтобы понять, как работают маршруты, взглянем на маршрут, уже заданный для выдачи домашней страницы Express по умолчанию. В файле `index.js` в `app_server/routes` можно увидеть следующий фрагмент кода:

```
/* Получаем (GET) домашнюю страницу */  
router.get('/', function(req, res) {  
  res.render('index', { title: 'Express' });  
});
```



На позиции **1** в коде вы можете видеть `router.get('/',` означающее, что маршрутизатор ищет запрос GET по пути URL домашней страницы, в данном случае просто `'/'`. Анонимная функция, выполняющая код **2**, на самом деле контроллер. Этот пример очень прост, без всякого кода приложения. Итак, **1** и **2** — те части, которые мы хотим здесь разделить.

Вместо того чтобы сразу помещать код контроллера в каталог `controllers`, мы сначала попробуем этот подход на том же файле. Для этого возьмем аноним-

ную функцию из описания маршрута и опишем ее как именованную функцию. Затем передадим имя этой функции в виде обратного вызова в описании маршрута. Оба этих шага приведены в листинге 3.2, который можно поместить в файл `app_server/routes/index.js`.

Листинг 3.2. Извлечение кода контроллера из маршрута: шаг 1

```
var homepageController = function (req, res) {
  res.render('index', { title: 'Express' });
};

/* Получаем (GET) домашнюю страницу */
router.get('/', homepageController);
```

Берем анонимную функцию и описываем ее как именованную

Передаем имя этой функции в виде обратного вызова в описании маршрута

Если вы теперь обновите домашнюю страницу, то все будет работать как и раньше. Мы ничего не меняли в работе сайта, просто продвинулись на шаг вперед к разделению обязанностей.

ПОНЯТИЕ О RES.RENDER

Подробнее мы рассмотрим это понятие в главе 4, но `render` — это функция Express для компиляции шаблона представления с целью отправки его в виде получаемого браузером HTML-ответа. Метод `render` принимает в качестве параметров имя шаблона представления и объект данных JavaScript в виде следующей конструкции:

```
res.render('index', {title:'express'});
```

Объект JavaScript, содержащий данные для использования в шаблоне

Имя файла шаблона — в данном случае ссылается на `index.jade`

Обратите внимание на то, что расширение файла шаблона не обязательно, так что на `index.jade` можно сослаться просто как на `index`. Также не требуется задавать путь к каталогу `view`, поскольку это уже было сделано в основном файле настроек Express.

Теперь, когда мы разобрались, как работает описание маршрутов, настало время поместить код контроллера в соответствующее ему место.

Перемещение контроллера из файла маршрутов

Для того чтобы сослаться на код, находящийся во внешнем файле, в Node необходимо создать модуль в новом файле и затем воспользоваться для него командой `require` в исходном файле. (См. следующую врезку, в которой описываются глобальные принципы этого процесса.)

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ МОДУЛЕЙ NODE

Перемещение части кода из файла Node для создания внешнего модуля, к счастью, не представляет трудности. По сути, вы создаете новый файл для своего кода, выбирая, какие его части хотели бы сделать видимыми для исходного файла, а затем запрашиваете (require) новый файл из исходного.

В новом файле модуля желаемые части кода сделаются видимыми с помощью метода module.exports вот таким образом:

```
module.exports = function () {  
    console.log("Это делается видимым для запрашивающей стороны");  
};
```

Затем можно выполнить команду require в вашем главном файле вот так:

```
require('./yourModule');
```

Если вам нужно, чтобы у вашего модуля были доступными отдельные именованные методы, можете задать это в новом файле следующим способом:

```
module.exports.logThis = function(message){  
    console.log(message);  
};
```

Чтобы сослаться на это в исходном файле, необходимо присвоить ваш модуль именованной переменной и затем вызвать метод. Например, в вашем главном файле:

```
var yourModule = require('./yourModule');  
yourModule.logThis("Ура, работает!");
```

Эта команда присваивает ваш модуль переменной yourModule. Экспортированная функция logThis становится доступной в виде метода yourModule.

Обратите внимание на то, что при использовании функции require вам не требуется задавать расширение файла. Функция require будет искать несколько вещей: файл JavaScript с соответствующим именем или файл index.js в каталоге с заданным именем.

Итак, первое, что нам нужно сделать, — создать файл, в котором будет находиться код контроллера. Создаем новый файл main.js в каталоге app_server/controllers. В этом файле создадим метод для экспорта index и используем его для хранения кода res.render, как показано в листинге 3.3.

Листинг 3.3. Настройка контроллера домашней страницы в app_server/controllers/main.js

```
/* Получаем (GET) домашнюю страницу */  
module.exports.index = function(req, res){  
    res.render('index', { title: 'Express' });  
};
```

Создаем метод для экспорта index
Включаем код контроллера для домашней страницы

Это все, что нужно для экспорта контроллера. Следующий шаг — запросить этот модуль контроллера из файла маршрутов так, чтобы можно было использовать данный метод в описаниях маршрутов. Листинг 3.4 демонстрирует, как теперь должен выглядеть основной файл маршрутов index.js.

Листинг 3.4. Обновление файла маршрутов для использования внешних контроллеров

```
var express = require('express');
var router = express.Router();
var ctrlMain = require('../controllers/main'); ← 1 Запрашиваем
                                                главный файл контроллеров

/* Получаем (GET) домашнюю страницу */
router.get('/', ctrlMain.index); ← 2 Ссылаемся на метод
module.exports = router;           для экспорта index контроллеров
                                   в описаниях маршрутов
```

В приведенном листинге маршрут связывается с новым контроллером путем запрашивания файла контроллера **1** и ссылки на функцию контроллера во втором параметре функции `router.get` **2**.

Теперь у нас имеется архитектура маршрутизации и контроллеров (рис. 3.7), где `app.js` запрашивает (`require`) файл `routes/index.js`, который, в свою очередь, запрашивает (`require`) `controllers/main.js`.

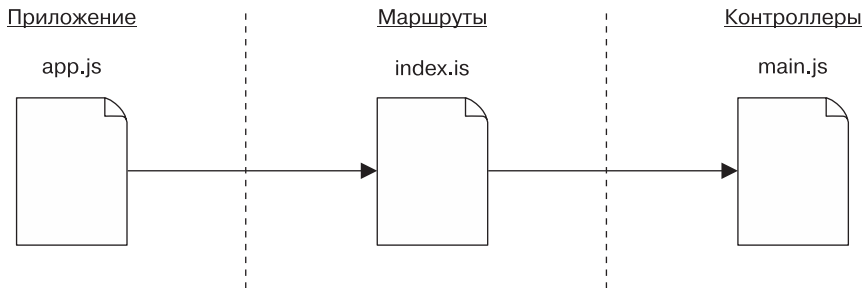


Рис. 3.7. Отделение логики контроллера от описаний маршрутов

Если вы попробуете это сейчас в своем браузере, то увидите, что домашняя страница Express по умолчанию опять отображается правильно.

Теперь все касающееся Express настроено, так что наступило время создавать приложение. Но прежде нужно сделать еще пару вещей, первая из которых — добавление в приложение Twitter Bootstrap.

3.4. Импорт Bootstrap для быстроты и адаптивности макетов

Как обсуждалось в главе 1, приложение `Loc8r` будет использовать фреймворк Twitter Bootstrap для ускорения разработки адаптивного программного проекта. Мы также сделаем приложение выделяющимся, добавив тему. Цель этого — обеспечить непрерывное быстрое создание приложения, а не отвлекаться на семантику разработки адаптивного интерфейса.

3.4.1. Скачивание Bootstrap и добавление его в приложение

Инструкции по скачиванию Bootstrap, получению пользовательской темы и добавлению файлов в каталог проекта находятся в приложении Б. Главное здесь то, что файлы Bootstrap все статические, отправляемые непосредственно на браузер, они не требуют какой-либо обработки движком Node. В вашем приложении Express уже предусмотрен каталог для этой цели — каталог `public`. Когда все будет готово, каталог `public` будет выглядеть примерно так, как на рис. 3.8.

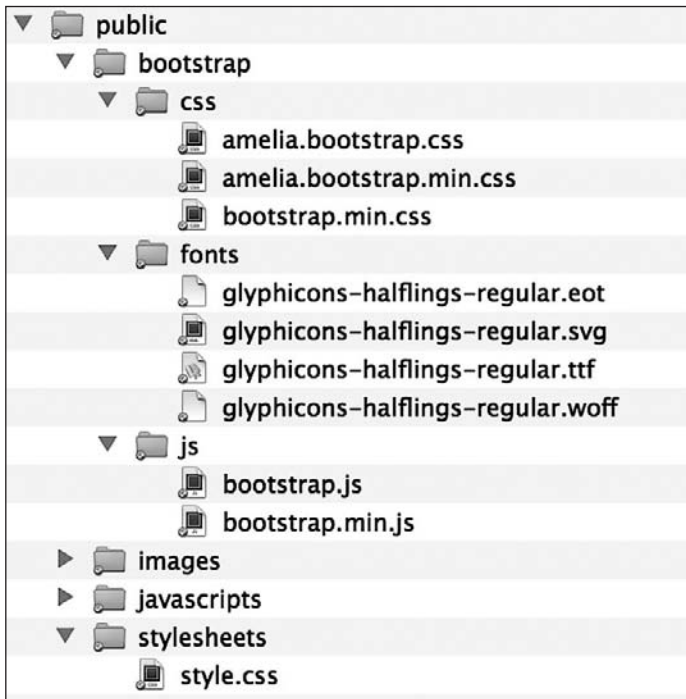


Рис. 3.8. Структура каталога `public` в приложении Express после добавления Bootstrap

Bootstrap также требует библиотеки jQuery для загрузки некоторых интерактивных компонентов. Сослаться на нее можно напрямую из CDN, но мы скачаем ее по адресу <http://jquery.com/download/>, чтобы она была в нашем приложении. Будем использовать самую последнюю из версий 1.x, которой на момент написания этой книги является 1.11.1. Итак, скачайте jQuery и сохраните ее в каталоге `public/javascripts` приложения.

3.4.2. Использование Bootstrap в приложении

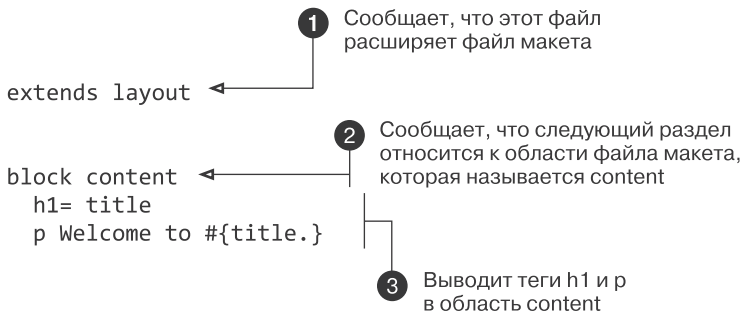
Теперь, когда все части Bootstrap включены в приложение, настало время привязать его к клиентской части. Это означает, что нам пора взглянуть на шаблоны Jade.

Работа с шаблонами Jade

Шаблоны Jade часто настраиваются для работы с помощью главного файла макета, задающего области, которые будут расширяться другими файлами Jade. Это вполне обоснованное решение при создании веб-приложения, поскольку в нем может быть много экранов или страниц с одинаковой базовой структурой и различным контентом.

Именно поэтому Jade включен в установку Express по умолчанию. Если вы взглянете на каталог `views` в приложении, то увидите два файла: `layout.jade` и `index.jade`. Файл `index.jade` управляет контентом индексной страницы приложения. Откройте его — внутри находится не так уж много, все содержимое показано в листинге 3.5.

Листинг 3.5. Весь файл `index.jade`



Здесь происходит гораздо больше, чем кажется на первый взгляд. Вверху файла находится оператор, который объявляет данный файл расширением другого файла **1**, в данном случае файла макета. За ним следует оператор, определяющий блок **2** кода, относящийся к некоторой области файла макета, в данном случае области под названием `content`. Наконец, тут имеется минимальный контент, отображаемый на индексной странице Express: один тег `<h1>` и один тег `<p>` **3**.

Здесь нет ни ссылок на теги `<head>` или `<body>`, ни каких-либо ссылок на таблицы стилей. Все они обрабатываются в файле макета, ведь именно там, скорее всего,

вы будете добавлять в приложение глобальные сценарии и таблицы стилей. Откройте файл `layout.jade`, и вы увидите что-то напоминающее листинг 3.6.

Листинг 3.6. Файл `index.jade` по умолчанию

```
doctype html
html
  head
    title= title
    link(rel='stylesheet',
        href='/stylesheets/style.css')
  body
    block content
```

← Пустой именованный блок, который могут использовать другие шаблоны

Листинг 3.6 демонстрирует файл макета, используемый для базовой индексной страницы в инсталляции Express по умолчанию. Как видите, в нем есть разделы `head` и `body`, а внутри раздела `body` имеется строка `block content`, внутри которой нет ничего. На этот именованный блок могут ссылаться другие шаблоны Jade, такие как файл `index.jade` из листинга 3.5. `block content` из индексного файла помещается в область `block content` файла макета при компиляции представлений.

Добавление Bootstrap к приложению в целом

Если вам нужно добавить какие-либо внешние вспомогательные файлы к приложению в целом, то при текущих настройках использование файла макета вполне оправданно. Так, в файле `layout.jade` нужно проделать четыре действия:

- добавить ссылку на файл CSS Bootstrap;
- добавить ссылку на файл JavaScript Bootstrap;
- добавить ссылку на библиотеку jQuery, необходимую для Bootstrap;
- добавить метаданные об окне просмотра, чтобы страница хорошо масштабировалась на мобильных устройствах.

Файл CSS и метаданные об окне просмотра должны находиться вверху документа, два файла сценариев — в конце раздела `body`. Листинг 3.7 демонстрирует размещение всего перечисленного в файле `layout.jade`, новые строки выделены полужирным шрифтом.

Листинг 3.7. Обновленный файл `layout.jade`, включающий ссылки Bootstrap

```
doctype html
html
  head
    meta(name='viewport', content='width=device-width,
    ➔ initial-scale=1.0')
    title= title
```

Задание метаданных об окне просмотра для лучшего отображения на мобильных устройствах ←

```

link(rel='stylesheet', href='/bootstrap/css/amelia.bootstrap.css') ←
link(rel='stylesheet', href='/stylesheets/style.css')
body
  block content
  script(src='/javascripts/jquery-1.11.1.min.js') ←
  script(src='/bootstrap/js/bootstrap.min.js') ←

```

Включает Bootstrap CSS из темы

Внедряет jQuery, необходимую для Bootstrap

Внедряет файл JavaScript Bootstrap

После выполнения этого кода любой созданный вами шаблон автоматически будет включать Bootstrap и станет масштабироваться на мобильных устройствах, если, конечно, новые шаблоны будут расширять шаблон макета.

Наконец, прежде, чем проверить работу всего этого, удалите содержимое файла `style.css` в `/public/stylesheets/`. Это предотвратит перекрытие файлов Bootstrap стилями Express по умолчанию. Немного позднее мы захотим добавить в приложение Local собственные стили, так что удалять файл не нужно.

Проверяем, что все работает

Если приложение еще не запущено с помощью `nodemon`, запустите его и посмотрите на него в браузере. Контент не изменился, но внешний вид должен был измениться. То, что вы видите, должно быть похоже на рис. 3.9.

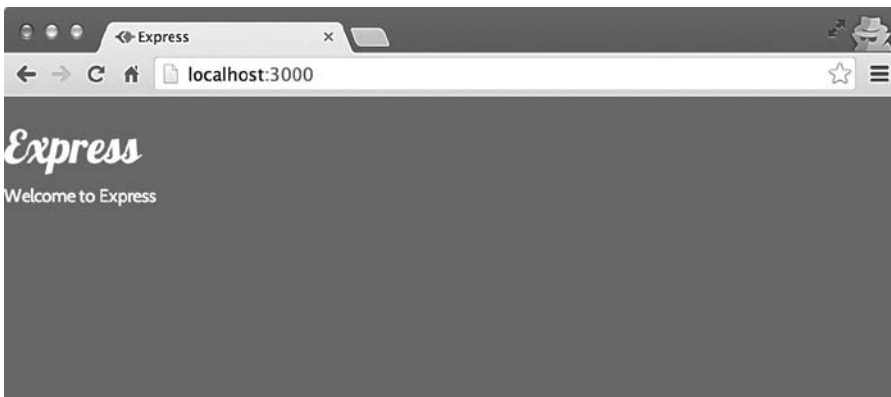


Рис. 3.9. Влияние темы Bootstrap на индексную страницу Express по умолчанию

Не забывайте, что исходный код приложения вы можете получить с GitHub, из ветви `chapter-03`. Для его клонирования выполните в новом каталоге в терминале следующую команду:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN.git
```

Итак, у нас уже что-то заработало локально, посмотрим теперь, как можно заставить приложение работать на промышленном сервере в Интернете.

3.5. Выводим приложение в Интернет с помощью Heroku

При работе с приложениями Node проблемой часто становится их развертывание на промышленном сервере в Интернете. Мы хотим пораньше избавиться от этой головной боли и уже сейчас выложить приложение Loc8r в Интернет. По мере того как оно будет строиться, мы можем выкладывать обновления. Для построения прототипа это очень удачно, поскольку сильно облегчает демонстрацию другим людям хода нашей работы.

Как упоминалось в главе 1, есть несколько провайдеров платформы как услуги (PaaS): Google Cloud Platform, Nodejitsu, OpenShift и Heroku. Мы будем использовать Heroku, но ничто не мешает вам попробовать и остальные варианты.

3.5.1. Установка и настройка Heroku

Прежде чем использовать Heroku, вам нужно зарегистрировать бесплатную учетную запись и установить Heroku Toolbelt на машине, которую вы используете для разработки. В приложении Б этот процесс описывается более подробно. Вам также понадобится совместимый с **bash терминал: используемый по умолчанию на Mac терминал подходит, но CLI по умолчанию для пользователей Windows — нет.** Если вы работаете в Windows, вам понадобится скачать, например, терминал GitHub, поставляемый в качестве части приложения GitHub для настольных компьютеров.

Как только вы все установите и настроите, мы сможем продолжить и подготавливать приложение к выкладыванию в Интернет.

Обновление PACKAGE.JSON

Платформа Heroku может выполнять приложения, основанные на самых различных типах баз исходных текстов, так что нужно сообщить ей, какое именно приложение мы хотим выполнить. Помимо информации о том, что мы запускаем приложение Node, использующее npm в качестве системы управления пакетами, нужно сообщить ей, какую версию мы запускаем, чтобы гарантировать соответствие настроек промышленной версии и версии, предназначенной для разработки.

Если вы не знаете точно, какие версии Node и npm используете, выяснить это можно с помощью всего лишь пары команд терминала:

```
$ node --version
$ npm --version
```

На момент написания данной книги эти команды возвращали **v4.2.1** и **2.2.0** соответственно. Вам нужно добавить эти версии в новый раздел **engines** в файле

`package.json`, используя уже встречавшийся вам префикс `~`, добавляющий джокерный символ для патч-версии. Полностью обновленный файл `package.json` показан в листинге 3.8, где добавленный раздел выделен полужирным шрифтом.

Листинг 3.8. Добавление раздела `engines` в `package.json`

```
{
  "name": "Loc8r",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },

  "engines": {
    "node": "~4.2.1",
    "npm": "~2.2.0"
  },

  "dependencies": {
    "express": "~4.9.0",
    "body-parser": "~1.8.1",
    "cookie-parser": "~1.3.3",
    "morgan": "~1.3.0",
    "serve-favicon": "~2.1.3",
    "debug": "~2.0.0",
    "jade": "~1.6.0"
  }
}
```

Добавляем раздел `engines` в `package.json`, чтобы сообщить Heroku, на какой платформе основано ваше приложение и какую версию использовать

При отправке на Heroku эти строки сообщат ему, что приложение использует последнюю патч-версию Node 4.2 и последнюю патч-версию `npm` 2.2.

Создание Procfile

Файл `package.json` сообщает Heroku, что речь идет о приложении Node, но не сообщает, как его запустить. Для этой цели нам понадобится `Procfile`. `Procfile` используется для объявления типов процессов, используемых приложением, и применяемых для их запуска команд.

Для `Loc8r` нужен веб-процесс, который будет запускать приложение Node. Поэтому создадим в корневом каталоге приложения файл под названием `Procfile` — с учетом регистра клавиатуры, расширение файла отсутствует. Внесите в `Procfile` следующую строку:

```
web: npm start
```

При отправке на Heroku этот файл сообщит ему, что приложению требуется веб-процесс и что необходимо выполнить команду `npm start`.

Локальное тестирование с помощью Foreman

Heroku Toolbelt поставляется с утилитой под названием Foreman. Ее можно использовать для проверки наших настроек и локального запуска приложения до размещения на Heroku. Если приложение уже запущено, остановите его, нажав `Ctrl+C` в окне терминала, в котором запущен процесс. Затем введите в окне терминала следующую команду:

```
$ foreman start
```

Если с настройками все нормально, она опять запустит приложение на локальной машине, но на этот раз на другом порте — 5000. Полученное в окне терминала подтверждение будет выглядеть примерно так:

```
16:09:01 web.1 | started with pid 91976
16:09:02 web.1 | > loc8r@0.0.1 start /path/to/your/application/folder
16:09:02 web.1 | > node ./bin/www
```

Если вы запустите браузер и перейдете по адресу `localhost:5000` (обратите внимание на то, что вместо порта 3000 используется порт 5000), то должны увидеть, что приложение опять запущено и работает.

Теперь, когда мы знаем, что наши настройки действуют, наступает время отправить приложение на Heroku.

3.5.2. Запускаем сайт в Интернет с помощью Git

В качестве метода развертывания Heroku использует Git. Если вы уже применяете Git, такой подход вам очень понравится, если же нет, то, возможно, вы будете немного опасаться, ведь мир Git может оказаться очень сложным. Но это не обязательно будет так, и когда вы начнете всерьез с ним работать, вы полюбите этот подход!

Хранение приложения в Git

Первое, что нужно сделать, — сохранить приложение в Git на локальной машине. Этот процесс состоит из трех шагов.

1. Задать каталог приложения в качестве репозитория Git.
2. Сообщить Git, какие файлы нужно добавить в репозиторий.
3. Зафиксировать эти изменения в репозитории.

Это может показаться сложным, но не пугайтесь. Для каждого шага вам понадобится всего одна простая команда терминала. Если приложение запущено ло-

кально, остановите его в терминале, нажав **Ctrl+C**. Затем, убедившись, что вы все еще находитесь в корневом каталоге приложения, выполните в терминале следующие команды:

```
$ git init
$ git add .
$ git commit -m "Первый коммит"
```

Задает каталог приложения
в качестве репозитория Git

Добавляет все находящееся
в каталоге в репозиторий

Фиксирует изменения
в репозитории,
выводя сообщение

Эти три действия вместе создадут локальный репозиторий Git, содержащий полную базу исходных текстов для приложения. Когда мы позднее будем модернизировать приложение и захотим выложить изменения в Интернет, то будем использовать две последние команды с другим сообщением для обновления репозитория.

Локальный репозиторий готов. Настало время создать приложение Heroku.

Создание приложения Heroku

Следующим шагом мы создадим приложение на Heroku в качестве удаленного репозитория для нашего локального репозитория. Все это выполняется с помощью команды терминала:

```
$ heroku create
```

После ее выполнения вы увидите в терминале подтверждение URL, по которому будет располагаться приложение, адрес репозитория Git и наименование удаленного репозитория, например:

```
http://shrouded-tor-1673.herokuapp.com/ |
➡ git@heroku.com:shrouded-tor-1673.git
Git remote heroku added
```

Если вы войдете в свою учетную запись Heroku в браузере, то увидите также, что там появилось приложение. Итак, у вас теперь есть контейнер Heroku для приложения, и следующим шагом будет отправка туда кода приложения.

Развертывание приложения на Heroku

На текущий момент у вас имеется приложение, хранящееся в локальном репозитории Git, и вы создали новый удаленный репозиторий на Heroku. Удаленный репозиторий пока что пуст, так что вам нужно отправить содержимое вашего локального репозитория в удаленный репозиторий Heroku.

Если вы не знакомы с Git, то сообщу: в нем имеется отдельная команда для выполнения этого действия со следующей логической структурой:



Эта команда отправит содержимое локального репозитория Git в удаленный репозиторий Heroku. Пока что в вашем репозитории имеется только одна ветвь, а именно ветвь `master`, которую вы и отправляете на Heroku. (См. более подробную информацию в следующей врезке.)

Когда вы выполняете приведенную команду, терминал по ходу процесса отображает массу журнальных сообщений, завершая все подтверждением того факта, что приложение было развернуто на Heroku. Это будет выглядеть примерно следующим образом, за исключением того, что URL окажется другим:

```
http://shrouded-tor-1673.herokuapp.com deployed to Heroku
```

ЧТО ТАКОЕ ВЕТВИ HEROKU?

Если вы просто работаете с одной и той же версией исходного кода и периодически отправляете ее в удаленный репозиторий, такой как Heroku или Git, то вы работаете в ветви `master`. Это совершенно нормально для линейной разработки с одним разработчиком. Если же у вас несколько разработчиков или ваше приложение уже опубликовано, то маловероятно, чтобы вы захотели вести разработку в ветви `master`. Вместо этого на основе кода из ветви `master` вы можете создать новую ветвь, в которой будете продолжать разработку, добавлять исправления или создавать новые возможности. После завершения работы в ветви ее можно слить с ветвью `master`.

Запуск динамического веб-контейнера на Heroku

Heroku использует понятие динамических контейнеров для запуска и масштабирования приложения. Чем больше у вас динамических контейнеров, тем больше системных ресурсов и процессов доступно вашему приложению. Добавлять новые динамические контейнеры при увеличении вашего приложения и росте его популярности очень просто.

В Heroku есть замечательный бесплатный пакет, идеально подходящий для создания прототипа приложения и его пробной версии. Вы получаете один динамический веб-контейнер бесплатно, чего более чем достаточно для наших целей.

Прежде чем вы сможете увидеть приложение онлайн, необходимо добавить один динамический веб-контейнер. Это легко можно сделать с помощью команды терминала:

```
$ heroku ps:scale web=1
```

При ее выполнении терминал выведет подтверждение:

```
Scaling web dynos... done, now running 1
```

Теперь проверим URL в Интернете.

Просмотр приложения по URL в Интернете

Все уже на своих местах и приложение выложено в Интернет! Вы можете убедиться в этом, введя полученный в подтверждении URL через вашу учетную запись в Негоки или с помощью следующей команды терминала:

```
$ heroku open
```

Она запустит приложение в вашем браузере по умолчанию, и вы увидите что-то напоминающее рис. 3.10.

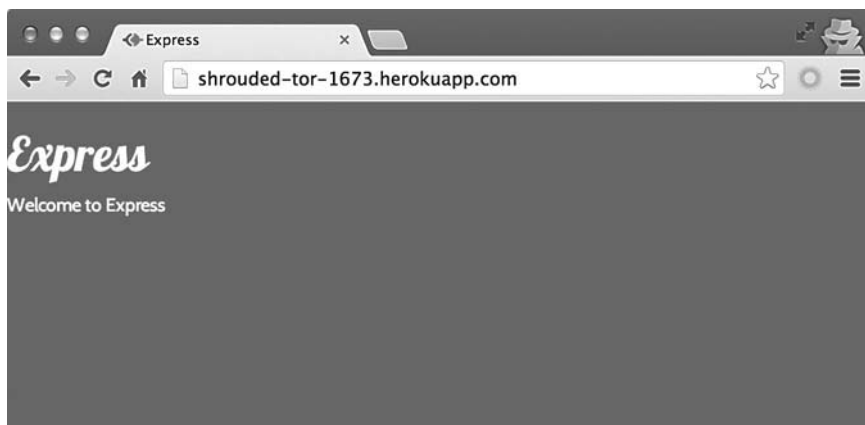


Рис. 3.10. Приложение Express MVC запущено по URL в Интернете

Конечно, ваш URL будет другим, и в Негоки вы можете изменить его, чтобы использовать вместо полученного от Негоки адреса свое доменное имя. В настройках приложения на сайте Негоки вы также можете поменять его на более осмысленный поддомен сайта herokuapp.com.

Наличие прототипа на доступном через Интернет адресе очень удобно как для тестирования в различных браузерах и устройствах, так и для отправки сотрудникам и партнерам.

Простота процесса обновления

Теперь, когда приложение Heroku настроено, обновлять его очень удобно. Всякий раз, когда вам нужно отправить какие-либо изменения, вы просто вводите следующие три команды:

```
$ git add .
$ git commit -m "Commit message here"
$ git push heroku master
```

Добавляет все изменения в локальный репозиторий Git

Фиксирует изменения в локальном репозитории с каким-то полезным сообщением

Отправляет изменения в репозиторий Heroku

Вот и все об этом, по крайней мере на текущий момент. Все может усложниться в том случае, когда приходится иметь дело с несколькими разработчиками и несколькими ветвями кода, но сам процесс отправки кода на Heroku остается неизменным.

3.6. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Создание нового приложения Express.
- ❑ Управление зависимостями приложения с помощью npm и файла `package.json`.
- ❑ Модификацию Express для соответствия подходу MVC к архитектуре.
- ❑ Маршруты и контроллеры.
- ❑ Создание новых модулей Node.
- ❑ Публикацию приложения Express в Интернете на платформе Heroku с помощью Git.

В следующей главе, когда мы будем создавать прототип приложения Loc8t, вы познакомитесь с Express еще ближе.

Глава 4

Создание статического сайта с помощью Node и Express

В этой главе:

- ❑ создание прототипа приложения посредством построения его статической версии;
- ❑ описание маршрутов для различных URL приложения;
- ❑ создание представлений в Express с помощью Jade и Bootstrap;
- ❑ использование контроллеров в Express для привязки маршрутов к представлениям;
- ❑ передача данных от контроллеров представлениям.

В главе 3 мы получили работающее приложение в стиле MVC, использующее Bootstrap для разработки макетов страниц. Наш следующий шаг — создать на этой основе статический сайт, по которому можно будет перемещаться. Это важнейший шаг в создании любого сайта или приложения. Даже если у вас есть лишь эскиз или какие-то наброски, с которых можно начать, ничем нельзя заменить быстрое создание реалистичного прототипа сайта, который можно будет использовать в браузере. В этом статическом сайте мы возьмем данные из представлений и поместим их в контроллеры. К концу главы у нас будут «умные» представления, способные отображать передаваемые им данные, и контроллеры, передающие жестко зашитые в них данные представлениям.

ПОЛУЧЕНИЕ ИСХОДНОГО КОДА

Если вы еще не создали приложение из главы 3, то можете получить исходный код с GitHub, из ветви `chapter-03`, по адресу <http://www.github.com/simonholmes/getting-MEAN>. Выполните в терминале в новом каталоге следующие команды для его клонирования и установки зависимостей модулей npm:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

В смысле построения архитектуры приложения эта глава будет сосредоточена на приложении Express (рис. 4.1).

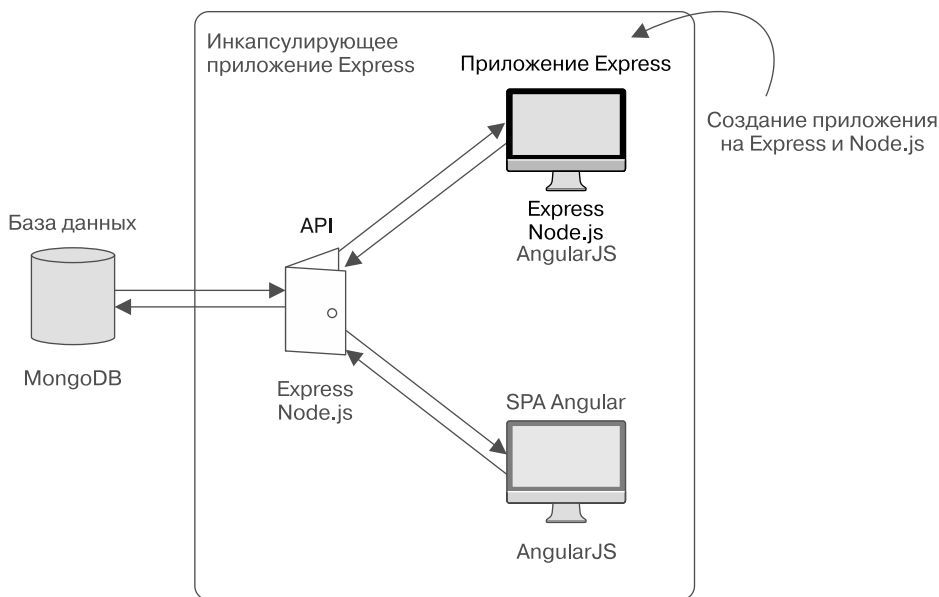


Рис. 4.1. Использование Express и Node для создания статического сайта для тестирования представлений

В соответствии с двумя основными шагами, которые мы выполним в этой главе, доступны две версии исходного кода. Первая версия содержит все данные в представлениях и показывает приложение по состоянию на конец раздела 4.4. Она доступна на GitHub, в ветви `chapter-04-views`.

Во второй версии данные находятся в контроллерах, а состояние приложения соответствует концу данной главы. Она доступна на GitHub, в ветви `chapter-04`.

Для получения одной из них воспользуйтесь следующими командами в новом каталоге в терминале, не забыв указать желаемую ветвь:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

Что ж, вернемся к Express.

4.1. Описание маршрутов в Express

В главе 2 мы распланировали создание приложения и выбрали четыре страницы, которые будем создавать. На рис. 4.2 показан набор страниц *Locations* (Местоположения) и страница из набора *Others* (Другие).

Местоположения



Другие

About (О нас)



Рис. 4.2. Наборы экранов, которые мы будем создавать в приложении *Loc8r*

Набор экранов — это замечательно, но они должны соответствовать входящим URL. Прежде чем мы приступим к написанию кода, хорошо было бы установить это соответствие и найти хороший образец.

Взгляните на табл. 4.1. Она демонстрирует простое соответствие экранов и URL и станет основой маршрутизации нашего приложения.

Таблица 4.1. Описание путей URL или маршрутов для каждого из экранов прототипа

Набор	Экран	Путь URL
Locations (Местоположения)	Список мест (это будет домашняя страница)	/
Locations (Местоположения)	Подробности о конкретном месте	/location
Locations (Местоположения)	Форма отзыва о месте	/location/review/new
Others (Другие)	О Loc8r	/about

Например, когда кто-то заходит на домашнюю страницу, мы хотим показать ему список мест, но когда кто-то заходит по пути URL `/about`, мы хотели бы показать ему информацию о Loc8r.

4.1.1. Различные файлы контроллеров для разных логических наборов

В главе 3, как вы помните, мы переместили всю логику контроллеров из описаний маршрутов во внешний файл. Заглядывая в будущее, мы знаем, что наше приложение будет расти, и не хотели бы, чтобы все контроллеры находились в одном файле. Разумной отправной точкой для их разбиения будет разделение по логическим наборам.

Глядя на принятые нами логические наборы, разобьем контроллеры на `Locations` (Местоположения) и `Others` (Другие). Чтобы понять, как это будет работать с точки зрения файловой архитектуры, сделаем набросок — что-то вроде изображенного на рис. 4.3.

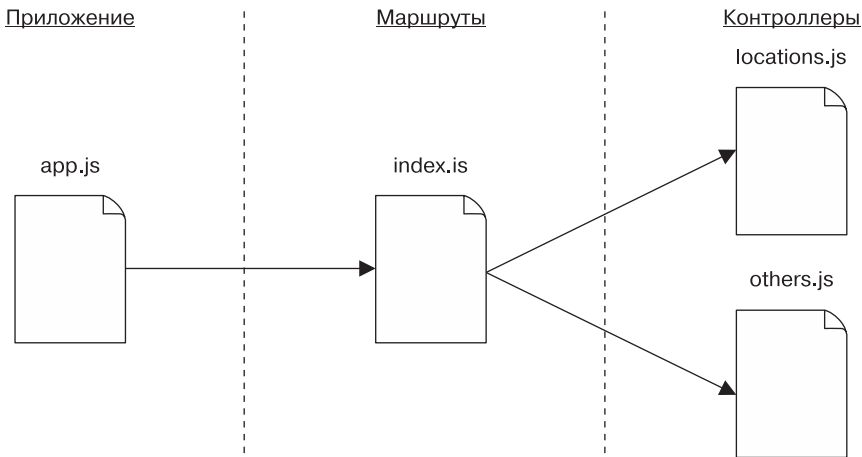


Рис. 4.3. Предлагаемая файловая архитектура для маршрутов и контроллеров в приложении

Приложение содержит файл маршрутов, в свою очередь, включающий несколько файлов контроллеров, каждый из которых назван по соответствующим логическим наборам.

Здесь мы видим один файл маршрута и один файл контроллера для каждого логического набора экранов. Так спроектировано, чтобы помочь нам сформировать код в соответствии с организацией приложения. Вскоре мы рассмотрим контроллеры подробнее, а пока что поработаем с маршрутами.

Время планирования завершилось, настало время действовать! Возвращайтесь к среде разработки и открывайте приложение. Начнем работу с файла маршрутов `index.js`.

4.1.2. Запрос файлов контроллера

Как показано на рис. 4.3, в данном файле маршрутов нам необходимо сослаться на два файла контроллеров. Файлы контроллеров еще не созданы, мы сделаем это в ближайшее время.

Эти файлы будут носить названия `locations.js` и `others.js` и храниться в `app_server/controllers`. В `index.js` мы выполняем `require` обоих этих файлов и присваиваем каждый из них соответствующей переменной, как показано в листинге 4.1.

Листинг 4.1. Запрашиваем файлы контроллеров в `routes/index.js`

```
var express = require('express');
var router = express.Router();
var ctrlLocations = require('../controllers/locations');
var ctrlOthers = require('../controllers/others');
```



Заменяем существующую ссылку на `ctrlMain` двумя новыми переменными `require`

Теперь у нас есть две переменные, на которые можно ссылаться в описаниях маршрутов.

4.1.3. Настройка маршрутов

В `index.js` нам понадобятся маршруты для трех экранов в логическом наборе `Locations` (Местоположения), а также страница `About` (О нас) в логическом наборе `Others` (Другие). Для каждого из этих маршрутов необходима ссылка на контроллеры. Не забывайте, что маршруты служат просто средством установления соответствия, получая URL входящего запроса и устанавливая его соответствие конкретному элементу функциональности приложения.

Из табл. 4.1 нам уже известно, соответствия каких путей нужно задать, так что понятно, что нужно свести все это воедино в файле `routes/index.js`. Все, что должно быть в этом файле, показано в листинге 4.2.

Листинг 4.2. Описание маршрутов и задание их соответствий контроллерам

```
var express = require('express');
var router = express.Router();
var ctrlLocations = require('../controllers/locations');
var ctrlOthers = require('../controllers/others');

/* Страницы местоположений */
router.get('/', ctrlLocations.homelist);
router.get('/location', ctrlLocations.locationInfo);
router.get('/location/review/new',
           ctrlLocations.addReview);

/* Другие страницы */
router.get('/about', ctrlOthers.about);

module.exports = router;
```

Запрашиваем файлы контроллеров

Задаем маршруты для местоположений и устанавливаем их соответствие функциям контроллеров

Задаем остальные маршруты

Этот файл маршрутизации устанавливает соответствие заданных URL конкретным контроллерам, хотя мы их еще и не создали. Так что займемся этим и создадим контроллеры.

4.2. Создание основных контроллеров

На данной стадии мы собираемся создать действительно простейшие контроллеры, просто чтобы наше приложение работало и мы смогли проверить различные URL и маршрутизацию.

4.2.1. Настройка контроллеров

В каталоге контроллеров (в каталоге `app_server`) у нас пока что один файл — `main.js`, содержащий одну-единственную функцию, управляющую домашней страницей. Это показано в следующем фрагменте кода:

```
/* Получить (GET) домашнюю страницу */
module.exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

Нам фактически больше не нужен основной файл контроллеров, но мы можем использовать его в качестве шаблона. Начнем с переименования его в `others.js`.

Добавление контроллеров для Others (Другие)

Как вы помните из листинга 4.2, в `others.js` нам нужен один контроллер под названием `about`. Так что переименуем существующий контроллер `index` в `about`, оставив пока тот же самый шаблон представления и поменяв свойство для заго-

ловка на что-нибудь подходящее. Это поможет вам при проверке того, что маршрут работает как полагается. В листинге 4.3 показано полное содержимое файла контроллера `others.js` с учетом этих небольших изменений.

Листинг 4.3. Файл контроллеров `Others` (Другие)

```
/* Получить (GET) страницу About (О нас) */
module.exports.about = function(req, res){
  res.render('index', { title: 'About' });
};
```

Описываем маршрут, используя тот же шаблон представления, но поменяв название на `About` (О нас)

Итак, первый шаг выполнен, но приложение все еще не работает, так как пока отсутствуют контроллеры для маршрутов `Locations` (Местоположения).

Добавление контроллеров для `Locations` (Местоположения)

Добавление контроллеров для `Locations` (Местоположения), по существу, ничем особо не будет отличаться от описанного в предыдущем разделе. В файле маршрутов мы указываем имя искомого контроллера и имена трех функций контроллера.

В каталоге `controllers` создаем файл под названием `locations.js` и три простые функции контроллера: `homelist`, `locationInfo` и `addReview`. Следующий листинг демонстрирует, как это должно выглядеть.

Листинг 4.4. Файл контроллера для `Locations` (Местоположения)

```
/* Получить (GET) домашнюю страницу */
module.exports.homelist = function(req, res){
  res.render('index', { title: 'Home' });
};
/* Получить (GET) страницу с информацией о местоположениях */
module.exports.locationInfo = function(req, res){
  res.render('index', { title: 'Location info' });
};
/* Получить (GET) страницу добавления отзыва */
module.exports.addReview = function(req, res){
  res.render('index', { title: 'Add review' });
};
```

Похоже, все готово к использованию, так что приступим к проверке.

4.2.2. Тестирование контроллеров и маршрутов

Теперь, когда контроллеры и маршруты готовы к использованию, можете приступить к запуску приложения. Если вы еще не запустили его с помощью `nodemon`, перейдите в корневой каталог приложения в терминале и сделайте это:

```
$ nodemon
```

ДИАГНОСТИКА И УСТРАНЕНИЕ НЕПОЛАДОК

Если на этой стадии при перезапуске приложения возникли проблемы, то основное, что нужно проверить, — правильность наименований всех файлов, функций и ссылок. Посмотрите на сообщения об ошибках в окне терминала и подумайте, не дают ли они вам каких-то ключей к нахождению источника проблемы. Иногда они даже полезнее всего остального! Рассмотрим следующую потенциальную ошибку, выбрав наиболее интересные для нас фрагменты:

```
module.js:340
  throw err;
    ^
Error: Cannot find module '../controllers/other'

    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:364:17)
    at require (module.js:380:17)
    at module.exports (/Users/sholmes/Dropbox/Manning/Getting-MEAN/Code/Loc8r/BookCode/routes/index.js:2:3)
    at Object.<anonymous> (/Users/sholmes/Dropbox/Manning/Getting-MEAN/Code/Loc8r/BookCode/app.js:26:20)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
```

1 Первый ключ к нахождению: модуль не найден

2 Второй ключ к нахождению: файл, генерирующий ошибку

Во-первых, можно увидеть, что не найден модуль под названием `other` 1. Дальше в трассе вызовов в стеке можно найти файл, являющийся источником ошибки 2. Так что если затем вы откроете файл `routes/index.js`, то обнаружите, что написали `require('../controllers/other')`, в то время как хотели запросить файл `others.js`. Для устранения проблемы нужно всего лишь исправить ссылку, поменяв ее на `require('../controllers/others')`.

Если все получилось, этот процесс избавит вас от ошибок, то есть все маршруты будут указывать на контроллеры. Значит, вы сможете перейти в браузер и проверить каждый из созданных четырех маршрутов, таких как `localhost:3000` для домашней страницы и `localhost:3000/location` для страницы информации о конкретном месте. Поскольку мы поменяли отправляемые каждым из контроллеров шаблону представления данные, то легко сможем убедиться в корректной работе каждого из них, так как название и заголовок будут различными на всех страницах. Набор скриншотов для свежесозданных маршрутов и контроллеров демонстрирует рис. 4.4.

Отсюда видно, что каждый маршрут получает свой контент, так что мы знаем, что настройки маршрутизации и контроллеров работают правильно.

Следующий этап в процессе создания прототипа — помещение на каждый экран каких-нибудь HTML, макета и контента. Мы сделаем это с помощью представлений.

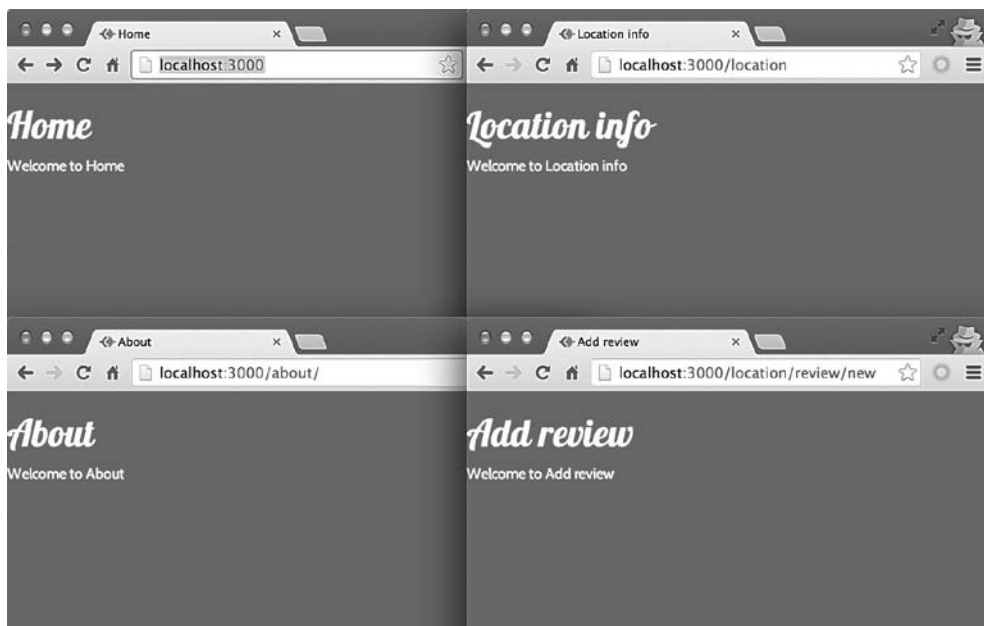


Рис. 4.4. Скриншоты для четырех созданных до сих пор маршрутов с различным текстом заголовков, поступающим через связанный с каждым из маршрутов контроллер

4.3. Создание представлений

После решения вопроса с пустыми страницами, путями и маршрутами наступает время добавить в наше приложение макет и какой-нибудь контент. Это по-настоящему вдохнет жизнь в приложение, и мы увидим, как идея реализуется на практике. На этом шаге будем использовать такие технологии, как Jade и Bootstrap. Jade — шаблонизатор по умолчанию в Express (хотя при желании вы можете использовать другие), а Bootstrap — фреймворк макетов для клиентской части, весьма облегчающий создание адаптивного сайта, по-разному отображающегося на настольном компьютере и мобильных устройствах.

4.3.1. Краткий обзор Bootstrap

Прежде чем начать, кратко рассмотрим Bootstrap. Мы не станем углубляться в детали относительно Bootstrap и всего, что он делает, но будет полезно рассмотреть некоторые из ключевых его понятий, прежде чем мы попробуем добавить его в файл шаблона.

Адаптивная система сеток Bootstrap

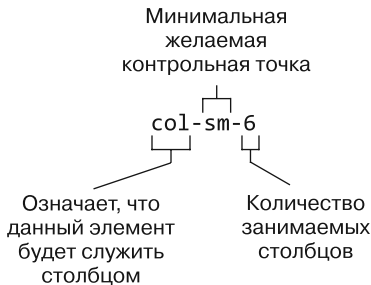
Bootstrap использует сетку с 12 столбцами. Вне зависимости от размера используемого вами экрана столбцов всегда будет 12. На телефоне столбцы будут узкими, а на большом внешнем мониторе — широкими. основополагающая идея Bootstrap состоит в том, что вы можете задать количество используемых элементом столбцов и это количество может быть разным для экранов разного размера.

Bootstrap имеет четко определенные контрольные точки для различных видов устройств, указываемые с помощью медиазапросов CSS. Они различаются шириной экрана в пикселах. Эти контрольные точки перечислены в табл. 4.2 вместе с примерами устройств, соответствующих каждому размеру.

Таблица 4.2. Контрольные точки, на которые ориентируется Bootstrap для различных типов устройств

Название контрольной точки	Ключ CSS	Пример устройства	Ширина, пикселей
Очень маленькие устройства	xs	Телефон	Менее 768
Маленькие устройства	sm	Планшет	768 и более
Средние устройства	md	Ноутбук	992 и более
Большие устройства	lg	Внешний монитор	1200 и более

Для задания ширины элемента необходимо объединить ключ CSS из табл. 4.2 с количеством столбцов, которые вы хотели бы охватить. Класс, обозначающий столбец, конструируется следующим образом:



Класс `col-sm-6` сделает так, что элемент, к которому он применяется, будет занимать шесть столбцов на экранах размера `sm` и более. То есть на планшетах, ноутбуках и мониторах этот столбец будет занимать половину доступной ширины экрана.

Чтобы добиться адаптивности, можно применить к конкретному элементу несколько классов. Так, если вам нужно, чтобы содержимое `div` занимало всю ширину экрана на телефоне, но только половину ширины на планшетах и бóльших устройствах, можете использовать следующий фрагмент кода:

```
<div class="col-xs-12 col-sm-6"></div>
```

Класс `col-xs-12` указывает макету использовать 12 столбцов на очень маленьких устройствах, а класс `col-sm-6` указывает макету использовать шесть столбцов на маленьких и бóльших устройствах. Рисунок 4.5 иллюстрирует эффект, производимый этим кодом на различных устройствах, если таких фрагментов два, один за другим на странице, вот так:

```
<div class="col-xs-12 col-sm-6">DIV ONE</div>
<div class="col-xs-12 col-sm-6">DIV TWO</div>
```

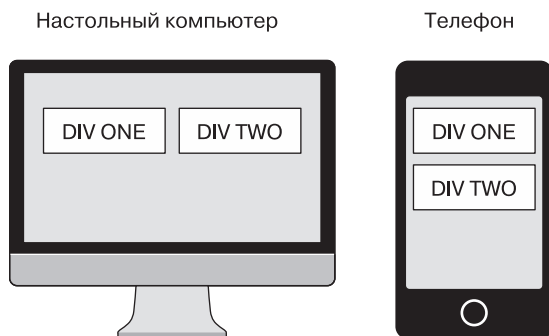


Рис. 4.5. Адаптивная система столбцов Bootstrap на настольном компьютере и мобильном устройстве. Классы CSS используются для определения количества столбцов (из 12), которые должен занимать каждый элемент при различных разрешениях экрана

Такой подход открывает путь для очень содержательного способа объединения адаптивных шаблонов, и мы будем активно его использовать для страниц `Loc8r`. Кстати, пора ими заняться.

4.3.2. Настройка каркаса HTML с помощью шаблонов Jade и Bootstrap

Существуют определенные общие требования ко всем страницам нашего приложения. Вверху страницы должны быть навигационная панель и логотип, внизу, в нижнем колонтитуле, — уведомление об авторских правах, а посередине — область контента. Мы стремимся создать что-то вроде рис. 4.6.



Рис. 4.6. Основная структура многократно используемого макета, включающая стандартную навигационную панель и нижний колонтитул с расширяемой и изменяемой областью контента между ними

Такой каркас для макета довольно прост, но вполне удовлетворяет наши потребности. Он обеспечивает единообразный вид, в то же время позволяя всем типам контента находиться посередине.

Как вы видели в главе 3, шаблоны Jade используют принцип расширяемых макетов, позволяя вам описывать данный тип повторяющейся структуры лишь один раз в файле макета. В файле макета можно задать, какие части могут расширяться: как только выполнена настройка этого файла макета, можно расширять его столько раз, сколько хочется. Создание каркаса в файле макета означает, что вам нужно сделать это лишь однажды, причем сопровождать его нужно будет лишь в одном месте.

Макет

При построении общего каркаса мы в основном собираемся работать с файлом `layout.jade` в каталоге `app_server/views`. На текущий момент он минимален и выглядит примерно как в следующем фрагменте кода:

```
doctype html
html
  head
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    title= title
    link(rel='stylesheet', href='/bootstrap/css/amelia.bootstrap.css')
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
    script(src='/javascripts/jquery-1.11.1.min.js')
    script(src='/bootstrap/js/bootstrap.min.js')
```

В области `body` пока что нет никакого HTML-контента, только расширяемый блок `content` и пара ссылок на сценарии. Мы собираемся сохранить все это, но хотели бы добавить раздел для навигации выше блока `content` и нижний колонтитул после него.

Создание области навигации

Bootstrap предоставляет набор элементов и классов, пригодных для создания закрепленной в верхней части экрана навигационной панели, и сворачивает опции в раскрывающееся меню на мобильных устройствах. Мы не будем исследовать здесь нюансы классов Bootstrap CSS, так как достаточно всего лишь взять код примера с сайта Bootstrap, немного его поменять и обновить в нем ссылки, поменяв на правильные.

В области навигации нам нужны:

- ❑ логотип Loc8r со ссылкой на домашнюю страницу;
- ❑ ссылка About (О нас) слева, указывающая на URL страницы /about.

Код для выполнения всего этого приведен в следующем фрагменте, его можно поместить в файле `layout.jade` над строкой `block content`:

```
.navbar.navbar-default.navbar-fixed-top
  .container
    .navbar-header
      a.navbar-brand(href='/') Loc8r
      button.navbar-toggle(type='button', data-toggle='collapse',
        data-target='#navbar-main')
        span.icon-bar
        span.icon-bar
        span.icon-bar
    #navbar-main.navbar-collapse.collapse
      ul.nav.navbar-nav
        li
          a(href='/about/') About
```

Настройка закрепленной вверху экрана навигационной панели Bootstrap

Добавление украшенной логотипом ссылки на домашнюю страницу

Настройка сворачивающейся навигационной панели для меньших разрешений экрана

Добавление ссылки About (О нас) в левой части панели

Если вы введете и запустите это, то можете увидеть, что навигационная панель теперь перекрывает заголовок страницы. Мы исправим это при создании макетов для области контента в подразделе «Создание шаблона» далее и в разделе 4.4, так что беспокоиться не о чем.

ПОДСКАЗКА

Помните, что Jade не включает каких-либо тегов HTML и что правильное структурированное расположение текста (отступы) критически важно для обеспечения ожидаемого результата.

Вот и все, что касается навигационной панели, на первых порах нам этого достаточно. Если вы новичок в Jade и Bootstrap, вам потребуется определенное время на то, чтобы привыкнуть к их подходу и синтаксису, но, как видите, благодаря им можно сделать много с помощью небольшого количества кода.

Оборачивание контента

Если двигаться по странице сверху вниз, то следующей областью будет блок `content`. Ничего особенного с ним делать не надо, так как его контент будет определяться другими файлами Jade. В нынешнем положении блок `content` прикреплен к левому краю страницы и ничем не ограничен, то есть будет растягиваться на полную ширину любого устройства.

Благодаря Bootstrap решить эту задачу несложно. Просто нужно обернуть блок `content` в контейнер `div`, вот так:

```
.container
  block content
```

`div` с классом `container` будет центрирован по ширине окна и на больших мониторах ограничен разумной максимальной шириной. Содержимое контейнера `div` останется при этом, как обычно, выровненным по левому краю.

Добавление нижнего колонтитула

Внизу страницы добавим стандартный нижний колонтитул. В него можно включить набор ссылок, условий или политику конфиденциальности. Пока что для простоты внесем в него только уведомление об авторских правах. Так как оно находится в файле макета, будет несложно обновить его на всех страницах, если позднее нам это понадобится.

Следующий фрагмент демонстрирует весь необходимый для нижнего колонтитула код:

```
footer
  .row
    .col-xs-12
      small &copy; Simon Holmes 2014
```

Лучше всего поместить это внутрь контейнера `div`, содержащего блок `content`, так что при добавлении убедитесь, что строка `footer` находится на том же уровне отступа, что и строка `block content`.

А теперь все вместе

Теперь, когда мы разобрались с навигационной панелью, областью контента и нижним колонтитулом, файл макета готов. Полностью код `layout.jade` показан в листинге 4.5 (изменения выделены полужирным шрифтом).

Листинг 4.5. Окончательный код каркаса макета в файле `app_server/views/layout.jade`

```
doctype 5
html
  head
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    title= title
    link(rel='stylesheet', href='/bootstrap/css/amelia.bootstrap.css')
```

```

link(rel='stylesheet', href='/stylesheets/style.css')
body
  .navbar.navbar-default.navbar-fixed-top ← Начинаем макет
    .container                               с закрепленной
      .navbar-header                         навигационной панели
        a.navbar-brand(href='/') Loc8r
        button.navbar-toggle(type='button', data-toggle='collapse',
        data-target='#navbar-main')
        span.icon-bar
        span.icon-bar
        span.icon-bar
        #navbar-main.navbar-collapse.collapse
        ul.nav.navbar-nav
          li
            a(href='/about/') About
  .container
    block content ← Расширяемый блок
  footer                                       content теперь обернут
    .row                                       в контейнер div
    .col-xs-12
      small &copy; Simon Holmes 2014
script(src='/javascripts/jquery-1.11.1.min.js')
script(src='/bootstrap/js/bootstrap.min.js')

```

Вот и все, что требуется для создания каркаса адаптивного макета с помощью Bootstrap, Jade и Express. Если у вас все это есть, то, запустив приложение, вы увидите что-то вроде скриншотов, приведенных на рис. 4.7, в зависимости от вашего устройства.

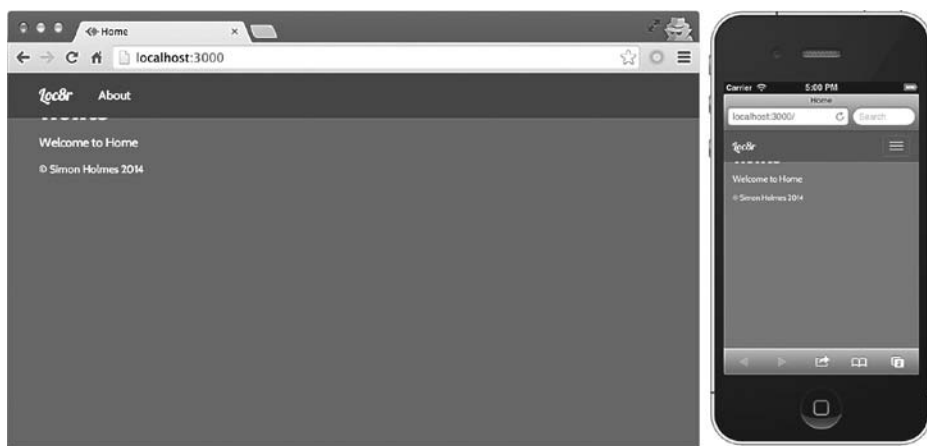


Рис. 4.7. Домашняя страница после настройки шаблона макета. Bootstrap автоматически свернул навигационную панель в раскрывающееся меню на телефонном экране маленького размера. Навигационная панель перекрывает контент, но это будет исправлено при создании макетов контента

Вы можете заметить, что навигационная панель по-прежнему перекрывает контент, но мы разберемся с этим совсем скоро, когда начнем заниматься макетами контента. То, что навигация работает так, как мы хотим, — хороший признак, хотя мы хотели бы, чтобы навигационная панель всегда была на виду, прикрепленная к верхней части окна. Также обратите внимание на то, как Bootstrap свернул навигационную панель в раскрывающееся меню на меньшем экране телефона. Получилось неплохо, не правда ли, при совсем небольших усилиях с нашей стороны?

ПОДСКАЗКА

Если вам не удастся зайти на ваш сайт для разработки с телефона, всегда можно попробовать поменять размер окна браузера. Также Google Chrome предоставляет возможность эмуляции различных мобильных устройств посредством консоли JavaScript.

Теперь, когда общий шаблон макета завершен, настало время начать создавать непосредственно страницы приложения.

4.3.3. Создание шаблона

При создании шаблонов начинайте с любого, который кажется вам наиболее подходящим. Это может быть самый сложный, или самый простой, или просто первый на основном пути пользователя. Лучше всего начать с домашней страницы `Loc8r` — это пример, который мы будем рассматривать наиболее подробно.

Описание макета

Главная цель домашней страницы — отображение списка мест. У каждого места должны быть название, адрес, расстояние до него, список предоставляемых услуг и пользовательские оценки. Нам также хотелось бы добавить на страницу заголовки и для большей наглядности — какой-то текст, чтобы пользователи при первом же посещении поняли, с чем имеют дело.

Может оказаться полезно нарисовать, как я это делаю, эскиз одного или двух макетов на листе бумаги или доске. Мне представляется очень удобным в качестве отправной точки создания макета убедиться, что у вас на странице есть все необходимые составные части, не углубляясь в технические подробности какого-либо кода. Мой эскиз домашней страницы `Loc8r` демонстрирует рис. 4.8. Вы можете увидеть на нем два макета: для настольного компьютера и телефона. На этой стадии не помешает определиться с параметрами адаптивности, помня о том, что Bootstrap может делать и как он работает.

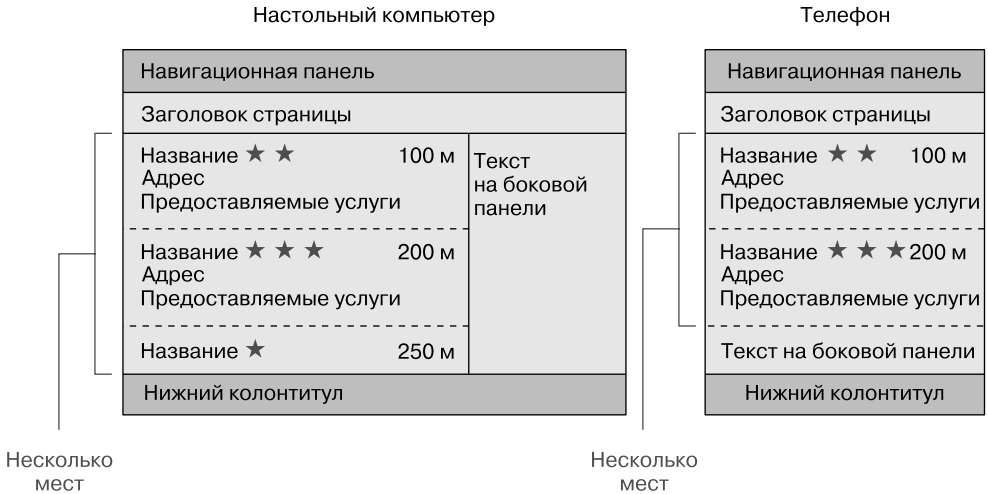


Рис. 4.8. Эскизы макетов домашней страницы для настольного компьютера и телефона. Рисование эскизов может дать вам предварительное представление о том, что вы собираетесь создавать, не отвлекая на тонкости Photoshop или технические детали кода

На данном этапе макеты отнюдь не окончательны, и по мере создания кода мы, вполне возможно, будем их менять и исправлять. Но любой путь становится легче при наличии точки назначения и карты, и это именно то, что дают нам эскизы. Это помогает при написании кода двигаться в верном направлении. Несколько затраченных на это минут с лихвой окупятся позднее — перемещать части приложения или даже выбрасывать их, чтобы начать все заново, гораздо проще, когда есть эскиз, а не просто масса исходного кода.

Теперь, когда мы получили представление о макете и необходимых составных частях контента, настало время объединить все это в новом шаблоне.

Настройка представления и контроллера

Первый шаг заключается в создании нового файла представления и привязке его к контроллеру. Создаем в каталоге `app_server/views` копию представления `index.jade` и сохраняем ее в том же каталоге под названием `locations-list.jade`. Лучше не называть ее `homepage` (домашняя страница) или подобным образом, так как позднее мы можем поменять свое мнение по поводу того, что должно отображаться на домашней странице. А так название представления понятно и его можно использовать где угодно без всякой путаницы.

Второй шаг — сообщить контроллеру домашней страницы, что мы хотим использовать это новое представление. Контроллер домашней страницы — это

файл `locations.js` в `app_server/controllers`. Измените в нем представление, вызываемое контроллером `homelist`. Это должно выглядеть так, как показано в следующем фрагменте кода (изменения выделены полужирным шрифтом):

```
module.exports.homelist = function(req, res){
  res.render('locations-list', { title: 'Home' });
};
```

Теперь создадим шаблон представления.

Написание кода шаблона: макет страницы

При фактическом написании кода макетов я предпочитаю начинать с больших кусков, а затем переходить к мелким деталям.

На данном этапе нам нужно решить, сколько из 12 столбцов Bootstrap должен занимать каждый элемент на каких устройствах. Следующий фрагмент кода демонстрирует макет с тремя различными областями страницы List (Список) приложения Loc8r:

```

#banner.page-header
  .row
    .col-lg-6
      h1 Loc8r
      small &nbsp; Find places to work with wifi near you!
  .row
    .col-xs-12.col-sm-8
      p List area.
    .col-xs-12.col-sm-4
      p.lead Loc8r helps you find places to work when out and about.
```

Заголовок страницы, занимающий всю ширину, содержит столбец, ограничивающий ширину текста до шести столбцов на больших экранах для удобства чтения

Контейнер для дополнительной или расположенной на боковой панели информации, занимающий все 12 столбцов на очень маленьких устройствах и четыре столбца на маленьких и больших устройствах

Контейнер для списка мест, занимающий все 12 столбцов на очень маленьких устройствах и восемь столбцов на маленьких и больших устройствах

Я основательно протестировал столбцы при различных разрешениях, пока не был полностью удовлетворен получившимся. Эмуляторы устройств могут облегчить этот процесс, но самый простой способ — просто менять ширину браузера, чтобы сработали различные контрольные точки Bootstrap. Когда вы добьетесь устраивающего вас результата, то сможете отправить его на Heroku, чтобы проверить по-настоящему на своем телефоне или планшете.

Написание кода шаблона: список мест

Теперь, когда контейнеры для домашней страницы описаны, пришло время основной области. У нас есть представление о том, что мы хотим там видеть, отраженное в эскизах, нарисованных для макета страницы. Для каждого места нужно указать название, адрес, оценку, расстояние до него и основные из предоставляемых услуг.

Поскольку мы создаем работоспособный прототип, все данные должны быть пока что жестко зашиты в шаблон. Это самый быстрый способ собрать шаблон и убедиться, что вся нужная информация отображается так, как нам хочется. Если вы работаете с существующим источником данных или на доступные для использования данные наложены какие-то ограничения, это нужно учитывать при создании макетов.

Создание полностью удовлетворяющего вас макета требует некоторой толики тестирования, но Jade и Bootstrap, работая вместе, существенно облегчают этот процесс. Следующий фрагмент кода показывает, что я сделал для одного отдельного места:

```

    .row.list-group
      .col-xs-12.list-group-item
        h4
          a(href="/location") Starcups ← Настроиваем группу-список Bootstrap
            и создаем отдельный элемент,
            охватывающий все 12 столбцов
            small &nbsp;
              span.glyphicon.glyphicon-star ← Наименование перечня
              span.glyphicon.glyphicon-star ← и ссылка на местоположение
              span.glyphicon.glyphicon-star
              span.glyphicon.glyphicon-star-empty
              span.glyphicon.glyphicon-star-empty
            span.badge.pull-right.badge-default 100m ← Используем значки
            p.address 125 High Street, Reading, RG6 1PS ← из набора Glyphicons
            span.label.label-warning Hot drinks ← фреймворка Bootstrap
            span.label.label-warning Food ← для вывода оценок
            span.label.label-warning Premium WiFi ← в виде звезд
            &nbsp;
            span.label.label-warning Premium WiFi
            &nbsp;
  
```

Используем вспомогательный класс badge фреймворка Bootstrap для нахождения расстояния до места

Вывод предоставляемых в данном месте услуг с помощью классов меток фреймворка Bootstrap

И снова вы можете видеть, как много можно сделать, приложив относительно небольшие усилия и написав небольшое количество кода, благодаря сочетанию Jade и Bootstrap. Чтобы вы имели представление о том, что делает предыдущий фрагмент кода, скажу: он будет визуализировать что-то вроде рис. 4.9.

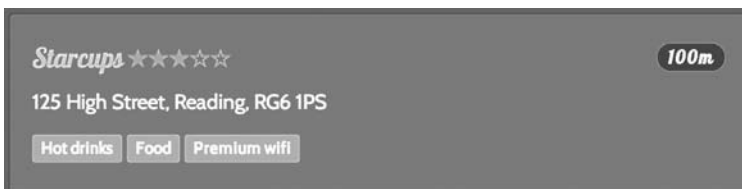


Рис. 4.9. Визуализация на экране отдельного места на странице List (Список)

Этот раздел настроен так, чтобы занимать всю ширину доступной области, то есть 12 столбцов на всех устройствах. Тем не менее не забывайте, что этот раздел вложен внутрь адаптивного столбца, так что полная ширина — это полная ширина охватывающего столбца, и вовсе не обязательно, что она соответствует окну просмотра браузера.

Наверное, будет разумнее собрать все воедино и посмотреть на получившееся в действии.

Написание кода шаблона: собираем все воедино

Итак, у нас есть макет элементов страницы, структура области списка и определенные зашитые данные. Посмотрим, на что это похоже все вместе. Чтобы лучше почувствовать, как макет будет вести себя в браузере, неплохо будет скопировать и изменить страницу List (Список) для отображения количества мест. Этот код, включающий для краткости только одно место, показан в листинге 4.6.

Листинг 4.6. Полный шаблон для `app_server/views/locations-list.jade`

```

extends layout
block content
  #banner.page-header
    .row
      .col-lg-6
        h1 Loc8r
        small &nbsp;Find places to work with wifi near you!
    .row
      .col-xs-12.col-sm-8
        .row.list-group
          .col-xs-12.list-group-item
            h4
              a(href="/location") Starcups
            small &nbsp;
              span.glyphicon.glyphicon-star
              span.glyphicon.glyphicon-star
              span.glyphicon.glyphicon-star
              span.glyphicon.glyphicon-star-empty
              span.glyphicon.glyphicon-star-empty
            span.badge.pull-right.badge-default 100m
            p.address 125 High Street, Reading, RG6 1PS
            p
              span.label.label-warning Hot drinks
              | &nbsp;
              span.label.label-warning Food
              | &nbsp;
              span.label.label-warning Premium wifi
              | &nbsp;
            .col-xs-12.col-sm-4
              p.lead Looking for wifi and a seat? Loc8r helps you find places
                to work when out and about. Perhaps with coffee, cake or a pint?
                Let Loc8r help you find the place you're looking for.

```

Начало области заголовка

Начало адаптивного раздела столбца основного перечня

Отдельный перечень; повторяем этот раздел, чтобы создать список из нескольких элементов

Настраиваем область боковой панели и заполняем ее каким-нибудь содержимым

Когда все это будет сделано, шаблон перечня на домашней странице будет готов. Если вы затем запустите приложение и перейдете по адресу localhost:3000, то увидите что-то вроде изображенного на рис. 4.10.

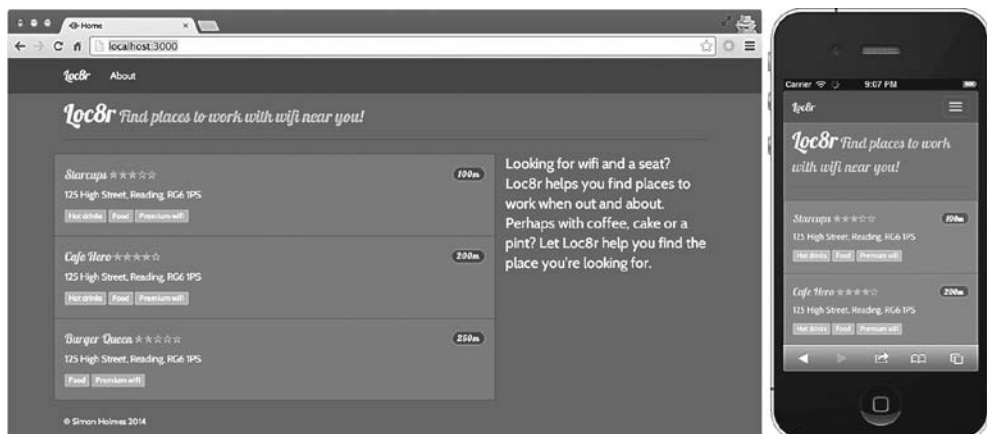


Рис. 4.10. Адаптивный шаблон для домашней страницы в действии на различных устройствах

Видите, как меняется макет в представлении на настольном компьютере по сравнению с представлением на мобильном устройстве? Это все благодаря адаптивному фреймворку Bootstrap и выбору классов CSS. При прокрутке вниз в представлении на мобильном устройстве вы можете увидеть текстовый контент боковой панели между основным списком и нижним колонтитулом. На меньшем экране важнее отображать в доступном пространстве список, а не текст.

Замечательно, мы получили адаптивный макет для домашней страницы с помощью Jade и Bootstrap в Express и Node. Давайте, не откладывая, добавим остальные представления.

4.4. Добавление остальных представлений

Набор Locations (Местоположения) страницы List (Список) сделан, так что перейдем к созданию остальных страниц, чтобы дать пользователям сайт, по которому они смогут перемещаться. В этом разделе мы обсудим добавление следующих страниц:

- ❑ Details (Подробности);
- ❑ Add Review (Добавление отзыва);
- ❑ About (О нас).

Впрочем, мы не станем рассматривать этот процесс столь же подробно для всех них, я просто дам пояснение, исходный код и результат. Исходный код всегда можно скачать с GitHub, если вы предпочитаете этот вариант.

4.4.1. Страница Details (Подробности)

Следующий по логике шаг и, вполне возможно, вторая по важности страница — это страница Details (Подробности) для отдельного места.

На этой странице должна отображаться вся информация о месте, включая:

- название;
- адрес;
- часы работы;
- предоставляемые услуги;
- схему проезда;
- отзывы, каждый с оценкой;
- имя оставившего отзыв;
- дату написания отзыва;
- текст отзыва.

Изрядный объем информации! Это самый сложный отдельный шаблон в нашем приложении.

Подготовка

Первое, что нужно сделать, — изменить контроллер для данной страницы, чтобы использовать другое представление. Найдите в файле `locations.js` в каталоге `app_server/controllers` контроллер `locationInfo`. Поменяйте имя представления на `location-info` в соответствии со следующим фрагментом кода:

```
module.exports.locationInfo = function(req, res){
  res.render('location-info', { title: 'Location info' });
};
```

Помните: если вы запустите приложение на данном этапе, оно не будет работать, поскольку Express не сможет найти шаблон представления. Ничего удивительного, ведь мы его еще не создали. Этому посвящен следующий раздел.

Представление

Создайте новый файл в `app_server/views` и сохраните его под названием `location-info.jade`. Его контент показан в листинге 4.7. Это самый большой листинг в нашей книге. Напоминаю, что для целей данного этапа разработки прототипа мы генерируем интерактивные страницы с жестко зашитыми непосредственно в них данными.

Листинг 4.7. Представление для страницы Details (Подробности), `app_server/views/location-info.js`

```

extends layout
block content
  .row.page-header
    .col-lg-12
      h1 Starcups
  .row
    .col-xs-12.col-md-9
      .row
        .col-xs-12.col-sm-6
          p.rating
            span.glyphicon.glyphicon-star
            span.glyphicon.glyphicon-star
            span.glyphicon.glyphicon-star
            span.glyphicon.glyphicon-star-empty
            span.glyphicon.glyphicon-star-empty
          p 125 High Street, Reading, RG6 1PS
        .panel.panel-primary
          .panel-heading
            h2.panel-title Opening hours
          .panel-body
            p Monday - Friday : 7:00am - 7:00pm
            p Saturday : 8:00am - 5:00pm
            p Sunday : closed
        .panel.panel-primary
          .panel-heading
            h2.panel-title Facilities
          .panel-body
            span.label.label-warning
              span.glyphicon.glyphicon-ok
              | &nbsp;&nbsp;Hot drinks
              | &nbsp;&nbsp;
            span.label.label-warning
              span.glyphicon.glyphicon-ok
              | &nbsp;&nbsp;Food
              | &nbsp;&nbsp;
            span.label.label-warning
              span.glyphicon.glyphicon-ok
              | &nbsp;&nbsp;Premium wifi
              | &nbsp;&nbsp;
        .col-xs-12.col-sm-6.location-map
          .panel.panel-primary
            .panel-heading
              h2.panel-title Location map
            .panel-body
              img.img-responsive.img-rounded(src=
                'http://maps.googleapis.com/maps/api/staticmap?center=
                51.455041,-0.9690884&zoom=17&size=400x350&sensor=
                false&markers=51.455041,-0.9690884&scale=2')

```

Начинаем со страницы заголовка

Настраиваем необходимые для шаблона вложенные адаптивные столбцы

Один из нескольких компонентов панели Bootstrap, используемых для задания информационных областей, в данном случае часов работы

Используем статическое изображение из Google Maps, включая в строку запроса координаты 51.455041, -0.9690884

```

    .row
      .col-xs-12
        .panel.panel-primary.review-panel
          .panel-heading
            a.btn.btn-default.pull-right(href='/location/review/new') Add
            review ←
          h2.panel-title Customer reviews
          .panel-body.review-container
            .row
              .review
                .well.well-sm.review-header
                  span.rating
                    span.glyphicon.glyphicon-star
                    span.glyphicon.glyphicon-star
                    span.glyphicon.glyphicon-star
                    span.glyphicon.glyphicon-star
                    span.glyphicon.glyphicon-star
                  span.reviewAuthor Simon Holmes
                  small.reviewTimestamp 16 July 2013
                .col-xs-12
                  p What a great place. I can't say enough good things
                    about it.
              .row
                .review
                  .well.well-sm.review-header
                    span.rating
                      span.glyphicon.glyphicon-star
                      span.glyphicon.glyphicon-star
                      span.glyphicon.glyphicon-star
                      span.glyphicon.glyphicon-star-empty
                      span.glyphicon.glyphicon-star-empty
                    span.reviewAuthor Charlie Chaplin
                    small.reviewTimestamp 16 June 2013
                  .col-xs-12
                    p It was okay. Coffee wasn't great, but the wifi was fast.
                .col-xs-12.col-md-3
                  p.lead Simon's cafe is on Loc8r because it has accessible wifi
                    and space to sit down with your laptop and get some work done.
                  p If you've been and you like it - or if you don't - please
                    leave a review to help other people just like you.

```

Создаем ссылку на страницу Add Review (Добавление отзыва) с помощью вспомогательного класса кнопок фреймворка Bootstrap

Итоговый адаптивный столбец для контекстно зависимой информации на боковой панели

Это оказался очень длинный шаблон, и вскоре мы рассмотрим, как можно его сократить. Но и страница сама по себе довольно сложная и содержит массу информации, а также несколько вложенных адаптивных столбцов. Представьте себе, однако, насколько длиннее она была бы, если бы была написана на чистом HTML!

Добавляем немного стиля

Этот шаблон будет выглядеть нормально и в нынешнем состоянии, но в нем есть определенные стилистические проблемы, которые можно легко решить с помощью всего нескольких строк CSS. Когда мы настраивали проект, именно для этого со-

хранили таблицу стилей Express по умолчанию на ее нынешнем месте, хотя и убрали из нее все содержимое. Этот файл носит название `style.css` и находится в каталоге `public/stylesheets`. Скопируйте следующий фрагмент кода в этот файл и сохраните его:

```
.review {padding-bottom: 5px;}
.panel-body.review-container {padding-top: 0;}
.review-header {margin-bottom: 10px;}
.reviewAuthor {margin: 0 5px;}
.reviewTimestamp {color: #ccc;}
```

После сохранения макет страницы `Details` (Подробности) будет закончен и можно перейти по адресу `localhost:3000/location` для его проверки. Рисунок 4.11 демонстрирует, как будет выглядеть этот макет в браузере и на мобильном устройстве.

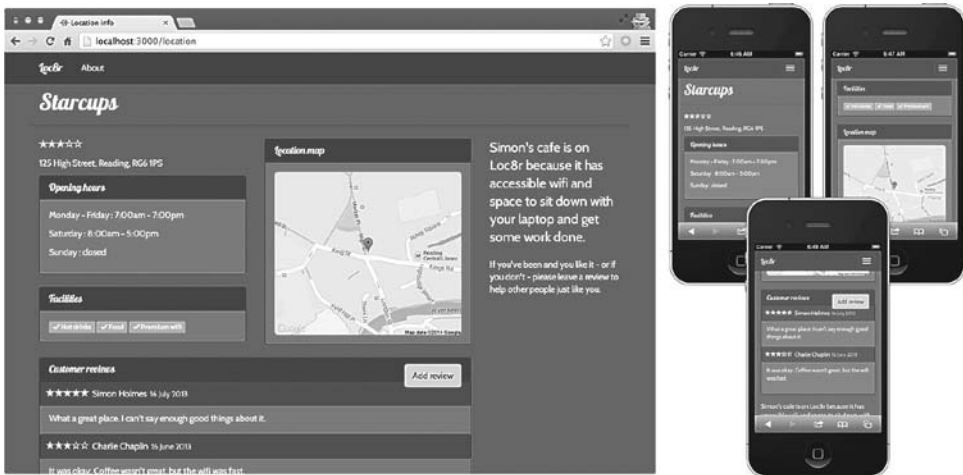


Рис. 4.11. Макет страницы `Details` (Подробности) на настольных компьютерах и мобильных устройствах

Следующий шаг на пути пользователя — страница `Add Review` (Добавление отзыва), требования к которой намного проще.

4.4.2. Создаем страницу `Add Review` (Добавление отзыва)

Это будет довольно простая страница. Она должна содержать только форму с именем пользователя и два поля ввода для оценки и отзыва.

Сначала нужно изменить контроллер, чтобы он ссылался на новое представление. В файле `app_server/controllers/locations.js` поменяйте контроллер

`addReview` для использования нового представления `location-review-form` в соответствии со следующим фрагментом кода:

```
module.exports.addReview = function(req, res){
  res.render('location-review-form', { title: 'Add review' });
};
```

Далее нужно создать само представление. В каталоге представлений `app_server/views` создайте новый файл с названием `location-review-form.jade`. Поскольку мы планируем создавать работоспособный прототип, то не собираемся никуда вносить данные формы, ведь цель заключается просто в том, чтобы действие выполняло переадресацию на страницу `Details` (Подробности), отображающую данные отзывов. Затем в форме мы зададим действие `/location` и метод `get`. Это даст нам необходимую на текущий момент функциональность. Код для страницы формы отзыва целиком показан в листинге 4.8.

Листинг 4.8. Представление для страницы Add Review (Добавление отзыва), `app_server/views/location-review.form.js`

```
extends layout
block content
  .row.page-header
    .col-lg-12
      h1 Review Starcups
  .row
    .col-xs-12.col-md-6
      form.form-horizontal(action="/location", method="get", role="form")
        .form-group
          label.col-xs-10.col-sm-2.control-label(for="name") Name
          .col-xs-12.col-sm-10
            input#name.form-control(name="name")
        .form-group
          label.col-xs-10.col-sm-2.control-label(for="rating") Rating
          .col-xs-12.col-sm-2
            select#rating.form-control.input-sm(name="rating")
              option 5
              option 4
              option 3
              option 2
              option 1
        .form-group
          label.col-sm-2.control-label(for="review") Review
          .col-sm-10
            textarea#review.form-control(name="review", rows="5")
          button.btn.btn-default.pull-right Add my review
        .col-xs-12.col-md-4
```

Задаем действие формы — `/location` и метод — `get`

Поле ввода имени человека, оставляющего отзыв

Раскрывающийся список выбора оценок от 1 до 5

Текстовое поле для отзыва

Кнопка отправки для формы

В Bootstrap есть множество вспомогательных классов для работы с формами, как можно видеть из листинга 4.8. Но это очень простая страница, и когда вы ее запустите, она будет выглядеть как на рис. 4.12.

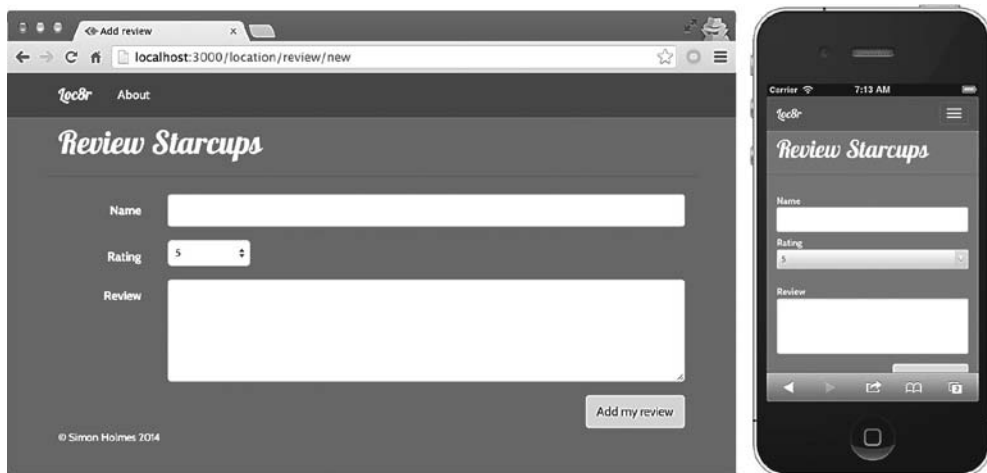


Рис. 4.12. Полная страница Add Review (Добавление отзыва) в настольном и мобильном представлениях

Страница Add Review (Добавление отзыва) отмечает окончание пути пользователя по набору экранов Locations (Местоположения). Осталось создать только страницу About (О нас).

4.4.3. Страница About (О нас)

Последняя страница статического прототипа — страница About (О нас). Это будет просто страница с заголовком и каким-то контентом, так что ничего сложного. Но макет может оказаться полезен для последующих страниц, таких, где излагаются политика конфиденциальности или условия, так что мы лучше сделаем универсальное, пригодное для повторного использования представление.

Контроллер для страницы About (О нас) находится в файле `others.js` в `app_server/controllers`. Найдите там контроллер под названием `about` и поменяйте название представления на `generic-text`, как в следующем фрагменте кода:

```
module.exports.about = function(req, res){
  res.render('generic-text', { title: 'About' });
};
```

Далее создайте представление `generic-text.jade` в `app_server/views`. Это относительно небольшой шаблон, который должен выглядеть так, как в листинге 4.9.

Листинг 4.9. Представление для чисто текстовых страниц, `app_server/views/generic-text.jade`

```
extends layout
block content
  #banner.page-header
    .row
      .col-md-6.col-sm-12
        h1= title
    .row
      .col-md-6.col-sm-12
        p
          | Loc8r was created to help people find places to sit down
          and get a bit of work done.
          | <br /><br />
          | Lorem ipsum dolor sit amet, consectetur adipiscing elit.
          Nunc sed lorem ac nisi dignissim accumsan.
```

Используем | для создания строк неформатированного текста внутри тега <p>

Листинг 4.9 — это очень простой макет. Не беспокойтесь на данном этапе насчет помещения страницезависимого контента в общее представление, мы скоро займемся этим и сделаем страницу пригодной для повторного использования. На текущий момент для завершения работоспособного статического прототипа этого достаточно.

Вероятно, вам захочется добавить сюда дополнительные строки, чтобы страница выглядела так, как будто в ней имеется настоящий контент. Обратите внимание на то, что строки, начинающиеся с вертикальной черты (|), могут, если нужно, содержать теги HTML. Рисунок 4.13 демонстрирует, как это могло бы выглядеть в браузере в случае несколько большего количества контента.

Это была последняя из четырех страниц, необходимых нам для статического сайта. Теперь можно отправить получившееся на `Heroku` и дать людям возможность набрать соответствующий URL и походить по сайту. Если вы забыли, как это делать, следующий фрагмент кода демонстрирует нужные вам команды терминала при условии, что вы уже настроили `Heroku` (в терминале вам необходимо находиться в корневом каталоге приложения):

```
$ git add .
$ git commit -m "Добавление шаблонов представлений"
$ git push heroku master
```

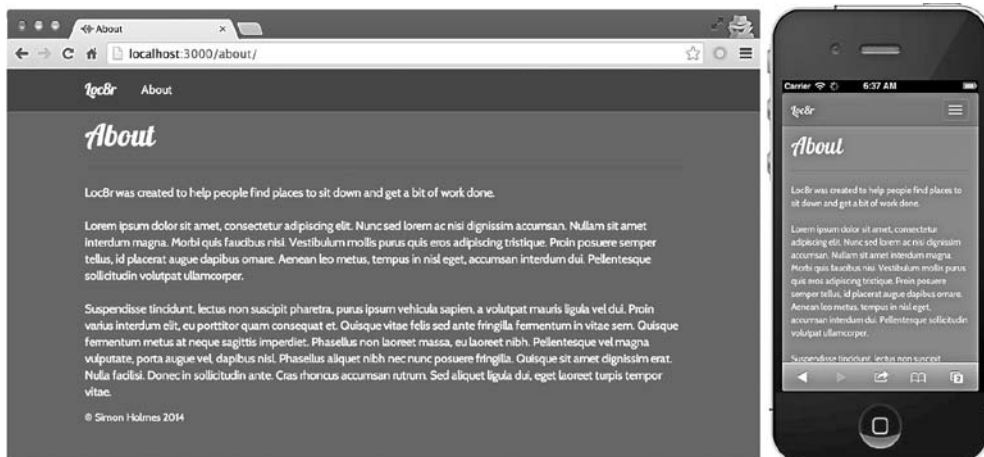



Рис. 4.13. Общий текстовый шаблон, визуализирующий страницу About (О нас)

ПОЛУЧЕНИЕ ИСХОДНОГО КОДА

Исходный код для приложения по состоянию на текущий момент книги доступен на GitHub, в ветви `chapter-04-views`, по адресу <http://www.github.com/simonholmes/getting-MEAN>. Выполните в терминале в новом каталоге следующие команды для его клонирования и установки зависимостей модулей npm:

```
$ git clone -b chapter-04-views https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

Что же дальше? **Маршруты, представления и контроллеры настроены для статического сайта**, по которому можно перемещаться. И вы только что отправили его на Heroku, так что и другие люди смогут его опробовать. В некотором смысле это конечная цель данного этапа, и вы можете остановиться на этом, пока не наиграетесь с путями пользователей и не получите какие-то отклики. Это, безусловно, самое удобное время в процессе создания приложения, чтобы выполнить большие и радикальные изменения.

Если вы точно собираетесь создавать SPA Angular и если вы довольны сделанным до настоящего момента, то вряд ли будете продолжать создавать статический прототип. Вместо этого вы наверняка начнете создавать приложения на AngularJS.

Но следующий шаг, который мы собираемся сейчас предпринять, продолжает процесс создания приложения Express. Так, не отступая от статического сайта, мы удалим данные из представлений и поместим их в контроллеры.

4.5. Извлечение данных из представлений и их интеллектуализация

На текущий момент весь контент и данные содержатся в представлениях. Это идеально подходит для тестирования и перемещения частей приложения, но нам хотелось бы продвигаться вперед. Конечная цель архитектуры MVC — представления без контента или данных. Представления должны просто получать данные, которые они демонстрируют конечным пользователям, не завися от этих данных. Представлениям необходимо знать структуру данных, но их содержание для самого представления не должно иметь значения.

Представьте себе архитектуру MVC: модель хранит данные, затем контроллер их обрабатывает и, наконец, представление визуализирует обработанные данные. Мы еще не имели дела с моделью — ее очередь наступит скоро, уже в главе 5. А пока работаем с представлениями и контроллерами. Чтобы сделать представления более интеллектуальными и заставить их выполнять свою функцию, необходимо удалить данные из представлений и поместить их в контроллеры. Поток данных в архитектуре MVC, а также изменения, которые мы хотим сделать, чтобы приблизиться на шаг к цели, иллюстрирует рис. 4.14.

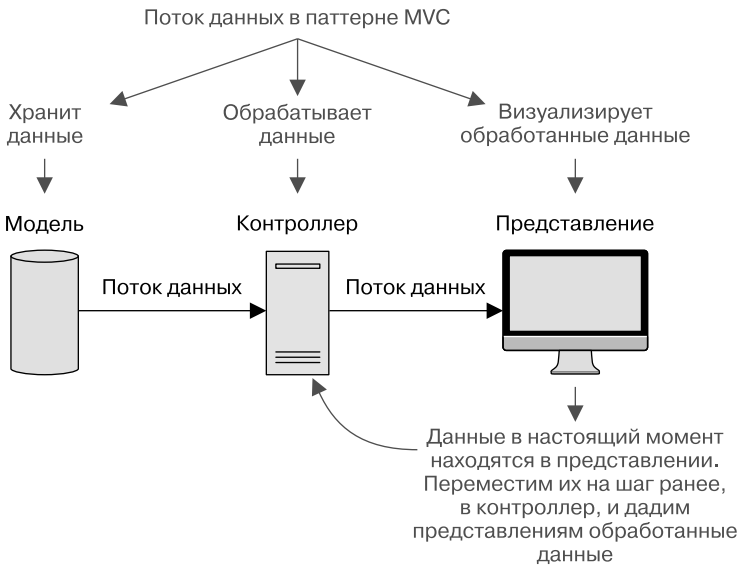


Рис. 4.14. Движение данных в паттерне MVC: из модели через контроллер в представление. Сейчас в прототипе данные находятся в представлении, но мы хотели бы переместить их на шаг ранее, в контроллер

Выполнение этих изменений сейчас даст нам возможность придать представлениям окончательный вид и быть готовыми к следующему шагу. Вдобавок мы начнем представлять, как обработанные данные должны будут выглядеть в контроллерах. Так, вместо того, чтобы начинать со структуры данных, мы начнем с идеальной клиентской части и не торопясь выполним обратное проектирование данных по шагам MVC, по мере того как начнем лучше понимать требования.

Как же мы собираемся это сделать? Начав с домашней страницы, мы извлечем каждый элемент контента из представлений Jade. Мы модернизируем файлы Jade для использования переменных вместо контента и поместим контент в виде переменных в контроллер. Далее контроллер сможет передать эти переменные в представление. Роли различных частей, а также движение и использование данных показаны на рис. 4.15.

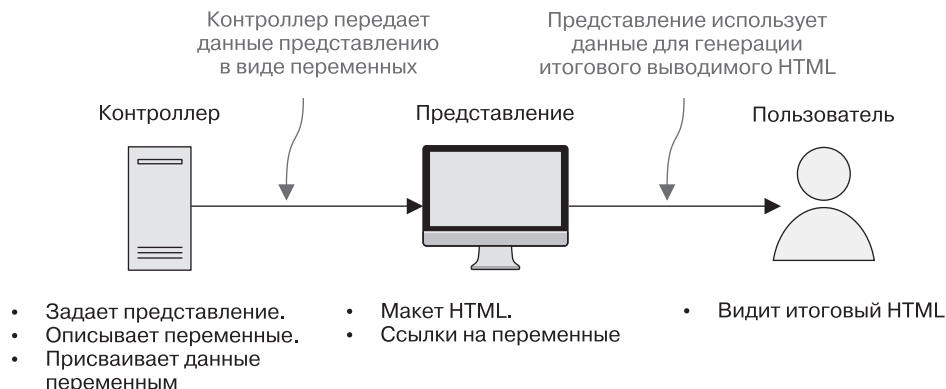


Рис. 4.15. Когда контроллер точно определяет данные, он передает их представлению в виде переменных, представление затем использует эти данные для генерации итогового HTML, выдаваемого пользователю

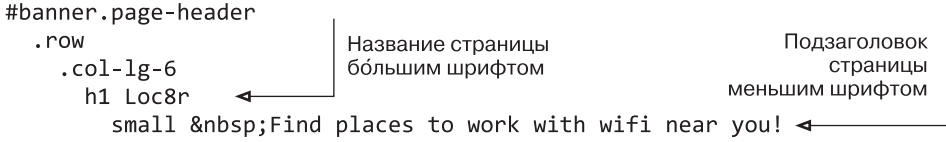
К концу этого этапа данные по-прежнему останутся жестко зашитыми, но в контроллерах вместо представлений. Представления теперь будут более «умными» и способными принимать и отображать любые присылаемые — в правильном формате, конечно, — им данные.

4.5.1. Как переместить данные из представления в контроллер

Мы собираемся начать с домашней страницы и переместить данные из представления `locations-list.jade` в функцию `homelist` в файле контроллеров `locations.js`. Начнем сверху, с чего-то довольно простого, а именно с заголовка страницы.

Следующий фрагмент кода демонстрирует раздел заголовка страницы представления `list.jade`, содержащий два элемента контента:

```
#banner.page-header
  .row
    .col-lg-6
      h1 Loc8r
      small &nbsp;Find places to work with wifi near you!
```



Эти два элемента контента — первое, что мы переместим в контроллер. Контроллер домашней страницы сейчас выглядит следующим образом:

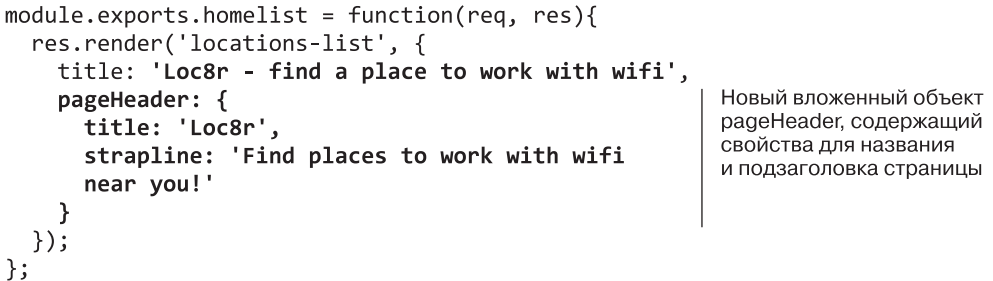
```
module.exports.homelist = function(req, res){
  res.render('locations-list', { title: 'Home' });
};
```

Он уже отправляет представлению один элемент данных. Вспомните, что второй параметр в функции `render` — объект JavaScript, содержащий данные, которые должны отправиться представлению. В данном случае контроллер `homelist` отправляет представлению объект данных `{ title: 'Home' }`. Это используется файлом макета для помещения строки `Home` в HTML `<title>` — отнюдь не лучший выбор текста.

Модернизация контроллера

Итак, поменяем название на что-то более подходящее для страницы, а также добавим два элемента данных к заголовку страницы. Прделаем эти изменения сначала в контроллере, как показано далее (изменения выделены полужирным шрифтом):

```
module.exports.homelist = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    }
  });
};
```



Для обеспечения аккуратности и удобства дальнейшего сопровождения название и подзаголовок сгруппированы в объекте `pageHeader`. Это хорошая привычка, которая облегчит модернизацию контроллеров и их сопровождение в будущем.

Обновление представления

Теперь, когда контроллер передает эти элементы данных представлению, можно обновить представление так, чтобы оно ссылалось на них вместо жестко зашитого контента. Ссылки на подобные вложенные элементы данных выполняются с помощью синтаксиса с использованием точки. Так, чтобы сослаться на подзаголовок

заголовка страницы в представлении `list.jade`, мы будем использовать синтаксис `pageHeader.strapline`. Следующий фрагмент кода демонстрирует раздел заголовка страницы представления (изменения выделены полужирным шрифтом):

```
#banner.page-header
  .row
    .col-lg-6
      h1= pageHeader.title
      small &nbsp;#{pageHeader.strapline}
```

= означает, что следующий контент представляет собой буферизованный код, в данном случае объект JavaScript

Разделители `#{}` используются для вставки данных в конкретное место, такое как часть текстового элемента

Код выполняет вывод `pageHeader.title` и `pageHeader.strapline` в соответствующие места представления (см. подробности о различных методах ссылки на данные в шаблонах Jade в следующей врезке).

ССЫЛКИ НА ДАННЫЕ В ШАБЛОНАХ JADE

Существует два основных синтаксиса для ссылки на данные в шаблонах Jade. Первый называется интерполяцией и обычно используется для вставки данных в середину какого-то другого контента. Интерполированные данные описываются с помощью открывающего разделителя `#{` и закрывающего `}`. Обычно их используют вот так:

```
h1 Добро пожаловать в #{pageHeader.title}
```

Если данные содержат HTML-код, он будет экранирован из соображений безопасности. Это значит, что конечные пользователи увидят любые теги HTML в виде текста и браузер не будет интерпретировать их как HTML. Если же вы хотите, чтобы браузер визуализировал какой-либо содержащийся в данных HTML, то можете использовать следующий синтаксис:

```
h1 Добро пожаловать в !{pageHeader.title}
```

Это несет потенциальную угрозу безопасности, поступать так можно, только если вы доверяете источникам данных. Следует запрещать отображение вводимых пользователями данных подобным образом, например, с помощью дополнительных проверок безопасности.

Второй способ вывода данных — с помощью буферизованного кода. Вместо того чтобы вставлять в строку данные, ее конструируют с помощью JavaScript. Это делается с помощью знака `=` сразу за объявлением тега, вот так:

```
h1= "Добро пожаловать в " + pageHeader.title
```

Опять-таки любой HTML при этом будет экранирован из соображений безопасности. Если вам необходим неэкранированный HTML в выводимых данных, можно использовать немного другой синтаксис:

```
h1!= "Добро пожаловать в " + pageHeader.title
```

Опять же будьте осторожны при использовании этого приема. Везде, где только можно, старайтесь применять один из методов с экранированием, просто для надежности.

Если вы теперь запустите приложение и перейдете на домашнюю страницу, вы увидите единственное изменение — обновленный `<title>`. Все остальное выглядит точно так же, как прежде, просто некоторые данные теперь поступают из контроллера.

Этот раздел был простым примером того, что мы делаем на данном этапе и как мы это делаем. Более сложная часть домашней страницы — раздел с перечнем, так что давайте посмотрим, как можно создать его.

4.5.2. Работаем со сложными повторяющимися данными

Первое, что нужно иметь в виду относительно раздела перечня, — то, что в нем несколько записей, которые следуют одному паттерну данных и одному паттерну макета. Аналогично тому, как мы только что поступили с заголовком страницы, начнем работу с данными с переноса их из представления в контроллер.

Говоря языком данных JavaScript, повторяющийся паттерн отлично согласуется с идеей массива объектов. У нас будет один массив с несколькими объектами, причем каждый отдельный объект содержит всю информацию, относящуюся к соответствующему перечню.

Анализируем данные в представлении

Давайте взглянем на перечень и посмотрим, какую информацию должен отправлять контроллер. Рисунок 4.16 напоминает, как выглядит перечень в представлении домашней страницы.

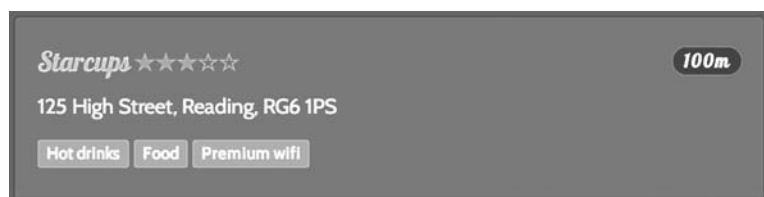


Рис. 4.16. Отдельный перечень, демонстрирующий требуемые данные

Из этого скриншота видим, что отдельному перечню на домашней странице необходимы следующие данные:

- ❑ название;
- ❑ оценка;
- ❑ удаленность;
- ❑ адрес;
- ❑ список предоставляемых услуг.

Создав объект JavaScript из данных, взятых со скриншота, приведенного на рис. 4.16, можно сделать что-то довольно простое наподобие приведенного в следующем фрагменте кода:

```
{
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '100m'
}
```

Список предоставляемых
услуг отправляется
в виде массива
строковых значений

Это все необходимые для отдельного места данные. Для нескольких мест понадобится массив подобных данных.

Добавление массива повторяющихся данных в контроллер

Итак, нам просто нужно создать массив объектов отдельных мест, взяв по желанию имеющиеся в настоящий момент в представлении данные, и добавить его в объект данных, передаваемый функции `render` в контроллере. Следующий фрагмент кода демонстрирует измененный контроллер `homelist`, включающий массив мест:

```
module.exports.homelist = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    locations: [{
      name: 'Starcups',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 3,
      facilities: ['Hot drinks', 'Food', 'Premium wifi'],
      distance: '100m'
    }, {
      name: 'Cafe Hero',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 4,
      facilities: ['Hot drinks', 'Food', 'Premium wifi'],
      distance: '200m'
    }, {
      name: 'Burger Queen',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 2,
      facilities: ['Food', 'Premium wifi'],
      distance: '250m'
    }
  ]
});
};
```

Массив мест
передается
представлению
в виде `locations`
для визуализации

Здесь в массиве отправляются подробности для трех мест. Конечно, можно добавить еще много других, но для начала хватит и этого. Теперь нам нужно заставить представление визуализировать эту информацию вместо жестко зашитых сейчас в него данных.

Организация цикла по массивам в представлении Jade

Контроллер отправляет представлению Jade массив в виде переменной `locations`. Jade предоставляет очень простой синтаксис для организации циклов по массивам. В одной строке мы указываем, какой массив использовать и какое имя переменной применять в качестве ключа. Ключ — это просто поименованная ссылка на текущий элемент массива, так что его содержимое меняется по мере продвижения цикла по массиву. Конструкция цикла Jade выглядит следующим образом:

Дайте имя ключу, который вы хотели бы использовать для доступа к данным

```
each location in locations
```

Имя массива, по которому организуется цикл

Все, что вложено внутри этой строки в Jade, будет повторено для каждого элемента массива. Посмотрим на пример применения этого синтаксиса, используя данные о местоположениях и часть интересующего нас представления. В файле представления каждое местоположение начинается с кода из следующего фрагмента, только всякий раз с другим именем:

```
.col-xs-12.list-group-item
  h4
    a(href="/location") Starcups
```

Мы можем воспользоваться синтаксисом `each/in` для организации цикла по всем местоположениям в массиве `locations` и вывода названия каждого из них. В следующем фрагменте кода показано, как это работает:

```
each location in locations
  .col-xs-12.list-group-item
    h4
      a(href="/location")= location.name
```

Настраиваем цикл, задавая переменную `location` в качестве ключа

Цикл проходит по всем вложенным элементам

Выводим название каждого местоположения, обращаясь к его свойству `name`

С учетом имеющихся данных контроллера с тремя местоположениями в них применение предыдущего кода приведет к следующему HTML-коду:

```
<div class="col-xs-12 list-group-item">
  <h4>
    <a href="/location">Starcups</a>
  </h4>
</div>
<div class="col-xs-12 list-group-item">
  <h4>
    <a href="/location">Cafe Hero</a>
  </h4>
</div>
<div class="col-xs-12 list-group-item">
  <h4>
    <a href="/location">Burger Queen</a>
  </h4>
</div>
```

Как видите, конструкции HTML — `div`, `h4` и теги — повторяются трижды. Но название местоположения различное во всех трех конструкциях и соответствует данным в контроллере.

Таким образом, организация цикла по массивам не представляет особых сложностей и с помощью такого простого теста мы уже получили первые несколько строк необходимого текста обновленного представления. Теперь нам просто нужно проделать это с остальными данными, используемыми в перечнях. С оценками в виде звезд мы так сделать не сможем, так что пока пропустим их и займемся ими чуть позже.

Для остальных данных мы можем написать следующий фрагмент кода, который выведет все данные для каждого перечня. Поскольку предоставляемые услуги передаются в виде массива, нам нужно будет организовать цикл по этому массиву для каждого перечня:

```
each location in locations
  .col-xs-12.list-group-item
    h4
      a(href="/location")= location.name
      small &nbsp;
        span.glyphicon.glyphicon-star
        span.glyphicon.glyphicon-star
        span.glyphicon.glyphicon-star
        span.glyphicon.glyphicon-star-empty
        span.glyphicon.glyphicon-star-empty
      span.badge.pull-right.badge-default= location.distance
    p.address= location.address
    p
      each facility in location.facilities
        span.label.label-warning= facility
        &nbsp;
```

Проходим в цикле по вложенному массиву для вывода услуг по каждому месту

Организация цикла по массиву предоставляемых услуг не является проблемой, и Jade выполняет это с легкостью. Извлечение остальных данных, таких как состояние и адрес, тоже выполняется очень просто с помощью уже рассмотренных методов.

Единственное, чем мы еще не занимались, — это звезды для оценок. Для них понадобится немного встроенного кода JavaScript.

4.5.3. Манипуляции данными и представлениями с помощью кода

Для оценок с присвоением звезд представление выводит `spans` с различными классами с помощью системы Glyphicon фреймворка Bootstrap. Всего имеется пять звезд, каждая из которых может быть сплошной или пустотелой в зависимости от оценки. Например, оценка 5 покажет пять сплошных звезд, оценка 3 представит три сплошные и две пустотелые звезды (рис. 4.17), а оценка 0 покажет пять пустотелых звезд.



Рис. 4.17. Система Glyphicon оценки с присвоением звезд в действии — демонстрирует оценку 3

Чтобы сгенерировать подобный результат, мы используем кое-какой код внутри шаблона Jade. По сути, этот код представляет собой JavaScript с добавлением в него некоторых условных обозначений, отражающих специфику Jade. Чтобы добавить строку встроенного кода в шаблон Jade, поставим в начале строки тире или дефис. Это сообщит Jade, что нужно запустить код JavaScript, а не передать его браузеру.

Для генерации вывода звезд нам понадобится несколько циклов `for`. Первый цикл выведет нужное количество сплошных звезд, а второй — оставшиеся пустотелые звезды. Следующий фрагмент кода демонстрирует, как это выглядит и работает в Jade:

```
- for (var i = 1; i <= location.rating; i++)
  span.glyphicon.glyphicon-star
- for (i = location.rating; i < 5; i++)
  span.glyphicon.glyphicon-star-empty
```

Обратите внимание на то, что синтаксис очень похож на JavaScript, но в нем нет задающих выполняемый блок кода фигурных скобок. Вместо этого блок кода за-

дается отступами, как и все остальное в Jade. Также обратите внимание на смесь кода и Jade. Строки кода говорят: «Каждый раз, когда значение выражения равно true, визуализирую контент Jade в соответствии с отступами». Это действительно удобный подход, ведь при нем не нужно пытаться сконструировать свой HTML-код с помощью JavaScript.

Итак, весь контент и макет для домашней страницы созданы и мы можем двигаться дальше. Разве что есть еще одна вещь, которую можно выполнить, чтобы улучшить то, что у нас существует, и сделать часть кода пригодной для повторного использования.

4.5.4. Инструкции `include` и `mixin` для создания пригодных для повторного использования компонентов макета

Только что написанный нами код оценок с присвоением звезд будет весьма полезен для других макетов. Он пригодится, например, на странице `Details` (Подробности) и, возможно, еще во многих местах в будущем. Не хотелось бы добавлять его на каждую страницу вручную. Вдруг мы решим, что `Glyphicons` нам больше не подходит, и захотим поменять разметку? Безусловно, нежелательно менять ее отдельно на каждой странице, выводящей оценки, по крайней мере если можно этого избежать.

К счастью, Jade предоставляет возможность создавать пригодные для повторного использования компоненты с помощью инструкций `include` и *примесей* (`mixin`).

Описание примесей Jade

По сути, *примесь* в Jade — это функция. Можно описать примесь вверху вашего файла и использовать ее затем во многих местах. Как это сделать, совершенно очевидно: просто задаете имя примеси, а затем вставляете ее содержимое с помощью отступов. Следующий фрагмент кода демонстрирует описание простейшей примеси:

```
mixin welcome
  p Welcome
```

При вызове оно выведет текст `Welcome` в теге `<p>`.

Как и функции JavaScript, примеси могут принимать параметры. Это окажется очень удобным для создания примеси, которая нам необходима для отображения оценок, так как вывод HTML-содержимого будет различным в зависимости от фактической оценки. Следующий фрагмент кода демонстрирует, как

это работает, задавая примесь, которую мы будем использовать на домашней странице для вывода оценок в виде звезд:

```

mixin outputRating(rating) ←
  - for (var i = 1; i <= rating; i++)
    span.glyphicon.glyphicon-star
  - for (i = rating; i < 5; i++)
    span.glyphicon.glyphicon-star-empty

```

Задаем примесь `outputRating`, принимающую один параметр — `rating`

Используем параметр `rating` в циклах для вывода правильного HTML

В известном смысле это работает аналогично функции JavaScript. Когда вы описываете примесь, то можете указать принимаемые ею параметры. Затем в примеси можно эти параметры использовать. Можете взять предыдущий фрагмент кода и вставить его вверху файла `locations-list.jade`, между строками `extends layout` и `block content`.

Вызов примесей Jade

Описав примесь, вы, конечно, хотели бы ее использовать. Синтаксис вызова примеси таков: просто поставьте `+` перед ее именем. Если у нее нет параметров, как у только что рассмотренной примеси `welcome`, вызов будет выглядеть следующим образом:

```
+welcome
```

Будет вызвана примесь `welcome` и выведен текст `Welcome` в теге `<p>`.

Вызов примеси с параметрами столь же прост. Просто передайте значения параметров в скобках, как вы поступили бы с функцией JavaScript. В файле `locations-list.jade`, в месте, где мы выводим оценки, значение оценки хранится в переменной `location.rating`, как показано в следующем фрагменте кода:

```

small &nbsp;
- for (var i = 1; i <= location.rating; i++)
  span.glyphicon.glyphicon-star
- for (i = location.rating; i < 5; i++)
  span.glyphicon.glyphicon-star-empty

```

Этот код можно заменить вызовом новой примеси `outputRating`, передав `location.rating` как параметр. Это будет выглядеть так, как в следующем фрагменте кода:

```

small &nbsp;
+outputRating(location.rating)

```

Теперь выводиться будет тот же HTML-код, что и раньше, но мы вынесли часть кода из содержимого макета. Пока что повторно применять этот прием можно только в том же файле, но дальше мы воспользуемся директивой `include`, чтобы обеспечить доступ из других файлов.

Использование инструкций include в Jade

Чтобы к нашей свежееиспеченной примеси можно было обращаться из других шаблонов Jade, необходимо сделать ее включаемым файлом. Это очень просто!

В каталоге `app_server/views` создайте подкаталог `_includes`. В этом каталоге создайте новый файл `sharedHTMLfunctions.jade` и вставьте в него определение примеси `outputRating`, вот так:

```
mixin outputRating(rating)
  - for (var i = 1; i <= rating; i++)
    span.glyphicon.glyphicon-star
  - for (i = rating; i < 5; i++)
    span.glyphicon.glyphicon-star-empty
```

Сохраните файл — и включаемый файл готов. Существует очень простой синтаксис для использования включаемого файла в макете Jade. Просто используйте ключевое слово `include` со следующим за ним относительным путем к включаемому файлу. Следующий фрагмент кода показывает строку кода, которую можно использовать вместо кода примеси вверху `location-listings.jade`:

```
include _includes/sharedHTMLfunctions
```

Теперь, вместо того чтобы использовать код примеси, встроенный в шаблон, мы можем вызывать ее из включаемого файла. Обратите внимание, что можно опустить расширение файла `.jade` при вызове директивы `include`. Так что теперь при создании нового шаблона, для которого требуются оценки в виде звезд, можно будет легко сослаться на данный включаемый файл и вызвать примесь `outputRating`.

Вот теперь мы действительно закончили работу над домашней страницей!

4.5.5. Законченная домашняя страница

В этом разделе мы произвели немало изменений в домашней странице. Посмотрим, к чему мы пришли в итоге. Сначала обратим внимание на обновленный контроллер. Листинг 4.10 демонстрирует итоговый контроллер `homelist`, включающий жестко зашитые данные.

Листинг 4.10. Контроллер `homelist`, передающий жестко зашитые данные представлению

```
module.exports.homelist = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
  },
```

Обновляем текст для тега HTML `<title>`

Добавляем текст для заголовка страницы в виде двух элементов внутри объекта

```

sidebar: "Looking for wifi and a seat? Loc8r helps you find places
to work when out and about. Perhaps with coffee, cake or a pint?
Let Loc8r help you find the place you're looking for.",
locations: [{
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '100m'
},{
  name: 'Cafe Hero',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 4,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '200m'
},{
  name: 'Burger Queen',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 2,
  facilities: ['Food', 'Premium wifi'],
  distance: '250m'
}]
});
};

```

Добавляем текст для боковой панели

Создаем массив, содержащий по одному объекту для каждого места в списке

Видя все это вместе, вы, вероятно, начнете понимать, куда мы движемся при таком подходе. Мы добились ясной картины всех необходимых для домашней страницы Loc8r данных. Это пригодится нам в следующей главе. Контроллер содержит текст для боковой панели. Мы не обсуждали этот шаг, но перенос этого текста из представления в контроллер состоит всего лишь в создании новой переменной для него в контроллере и ссылки на нее в представлении.

С помощью этого процесса мы добились важной вещи — исключения данных из представления. Создание представления с данными — отличный первый шаг, ведь это позволило нам сконцентрироваться на взаимодействии конечных пользователей с приложением, не отвлекаясь на технические детали. Теперь, когда мы переместили данные из представления в контроллер, представление стало динамическим и более «умным». Представление знает, какие элементы данных ему требуются, но ему все равно, что в них содержится. В листинге 4.11 показано итоговое представление для домашней страницы.

Листинг 4.11. Итоговое представление для домашней страницы, `app_server/views/locations-list.jade`

```

extends layout

include _includes/sharedHTMLfunctions ←
block content

```

Указываем внешний включаемый файл, содержащий примесь `outputRating`

```

#banner.page-header
  .row
    .col-lg-6
      h1= pageHeader.title
      small &nbsp;#{pageHeader.strapline}
    .row
      .col-xs-12.col-sm-8
        .row.list-group
          each location in locations
            .col-xs-12.list-group-item
              h4
                a(href="/location")= location.name
                small &nbsp;#{
                  +outputRating(location.rating)
                  span.badge.pull-right.badge-default= location.distance
                }
                p.address= location.address
              p
                each facility in location.facilities
                  span.label.label-warning= facility
                  | &nbsp;
            .col-xs-12.col-sm-4
              p.lead= sidebar

```

Выводим текст заголовка страницы с помощью различных методов

Цикл по массиву мест

Вызываем примесь `outputRating` для каждого места, передавая значение оценки текущего места

Ссылаемся на контент боковой панели из контроллера

Совсем небольшой шаблон, правда? Особенно учитывая все, что он делает. Это свидетельство мощи совместно работающих Jade и Bootstrap. Не говоря уже о побочном результате — удалении всего контента (обратите внимание на то, что контент боковой панели тоже извлекается из контроллера).

Мы на один шаг приблизились к цели MVC и разработки в целом — разделению обязанностей. По крайней мере для домашней страницы.

4.5.6. Модификация остальных представлений и контроллеров

Мы рассмотрели процесс для домашней страницы довольно подробно, но тратить столько времени на каждую из остальных страниц не станем. Однако прежде, чем перейти к следующему этапу разработки — созданию модели данных, нам необходимо выполнить этот процесс для всех страниц. Конечная цель — чтобы ни в одном из представлений не было данных, вместо этого представления станут интеллектуальными, а данные будут жестко защищены в соответствующие контроллеры.

Процесс для каждой страницы будет состоять из следующих шагов.

1. Внимательное изучение данных в представлении.
2. Создание структуры для этих данных в контроллере.
3. Замена данных в представлении ссылками на данные в контроллере.
4. Поиск возможностей для повторного использования кода.

В приложении В данный процесс рассматривается для каждой из трех оставшихся страниц. Там показано, как должны выглядеть код контроллера и представления для каждой из них. По окончании ни в одном из представлений не должно содержаться никаких жестко зашитых данных, необходимые для каждой страницы данные должен передавать контроллер. Набор скриншотов итоговых страниц, которые должны получиться к концу этого этапа, показан на рис. 4.18.



Рис. 4.18. Скриншоты всех четырех страниц статического прототипа с использованием интеллектуальных представлений и жестко зашитых в контроллеры данных

Мы добрались до конца первого этапа ускоренной разработки прототипа и готовы приступить к следующему этапу.

ПОЛУЧЕНИЕ ИСХОДНОГО КОДА

Исходный код для приложения по состоянию на текущий момент доступен на GitHub, в ветви `chapter-04` репозитория `getting-MEAN`. Выполните в терминале в новом каталоге следующие команды для его клонирования и установки зависимостей:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

4.6. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Описание и организацию маршрутов в Express.
- ❑ Создание модулей Node, содержащих контроллеры.
- ❑ Использование нескольких наборов контроллеров с маршрутами.
- ❑ Создание представлений с помощью Jade и Bootstrap.
- ❑ Создание пригодных для повторного использования компонентов Jade, примесей.
- ❑ Отображение динамических данных в шаблонах Jade.
- ❑ Передачу данных от контроллеров представлениям.

В главе 5 мы продолжим перемещать данные назад по архитектуре MVC с помощью MongoDB и Mongoose для создания модели данных. Все правильно, настало время базы данных!

Глава 5

Создание модели данных с помощью MongoDB и Mongoose

В этой главе:

- ❑ как MongoDB помогает сопрягать приложение Express/Node с базой данных MongoDB;
- ❑ описание схем для модели данных с помощью Mongoose;
- ❑ подключение приложения к БД;
- ❑ управление базами данных из командной оболочки MongoDB;
- ❑ перевод БД в промышленную эксплуатацию;
- ❑ использование правильной БД в зависимости от среды, выбор между локальной и промышленной версиями приложения.

В главе 4 мы закончили на перемещении данных из представления назад (вниз) по пути MVC в контроллер. В конце концов, контроллеры будут передавать данные представлениям, но они не должны эти данные хранить. Рисунок 5.1 напоминает, как выглядит поток данных в паттерне MVC.

Для хранения данных нам понадобится БД, а именно MongoDB. Итак, следующий шаг в процессе — создание БД и модели данных.

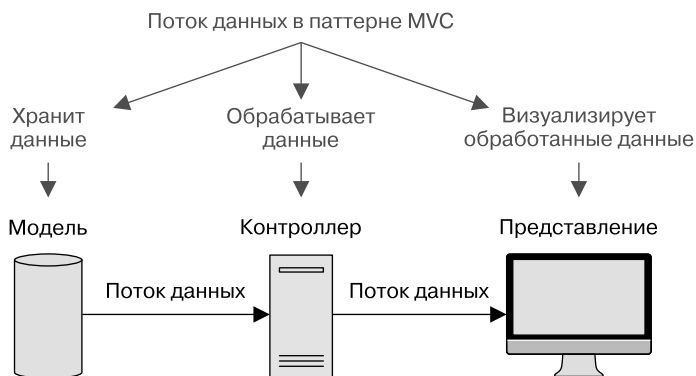


Рис. 5.1. В паттерне MVC данные содержатся в модели, обрабатываются контроллером и затем визуализируются представлением

ПРИМЕЧАНИЕ

Если вы еще не создали приложение из главы 4, то можете получить исходный код с GitHub, из ветви `chapter-04`, по адресу <http://www.github.com/simonholmes/getting-MEAN>. Выполните в терминале в новом каталоге следующие команды для его клонирования:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN.git
```

Подключим приложение к БД, прежде чем воспользоваться Mongoose для описания схем и моделей. Когда структура будет нас удовлетворять, можно будет добавить какие-то тестовые данные прямо в базу данных MongoDB. Последним шагом будет проверка работы всего этого после отправки на Heroku. Последовательность действий, выполняемых на этих четырех шагах, демонстрирует рис. 5.2.

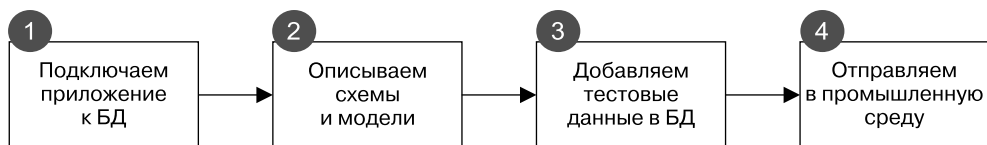


Рис. 5.2. Четыре основных шага данной главы, начиная от подключения приложения к БД и заканчивая отправкой всего получившегося в среду промышленной эксплуатации

Для тех, кто волнуется, что пропустил раздел или два: не волнуйтесь, мы пока еще не создали БД. И не должны будем создавать. В других стеках технологий это может быть проблемой и источником ошибок. Но при работе с MongoDB не нужно создавать БД до подключения к ней. MongoDB создаст базу данных тогда, когда мы впервые попытаемся ее использовать.

На рис. 5.3 показано, на чем мы сосредоточим внимание в данной главе, говоря языком общей архитектуры.

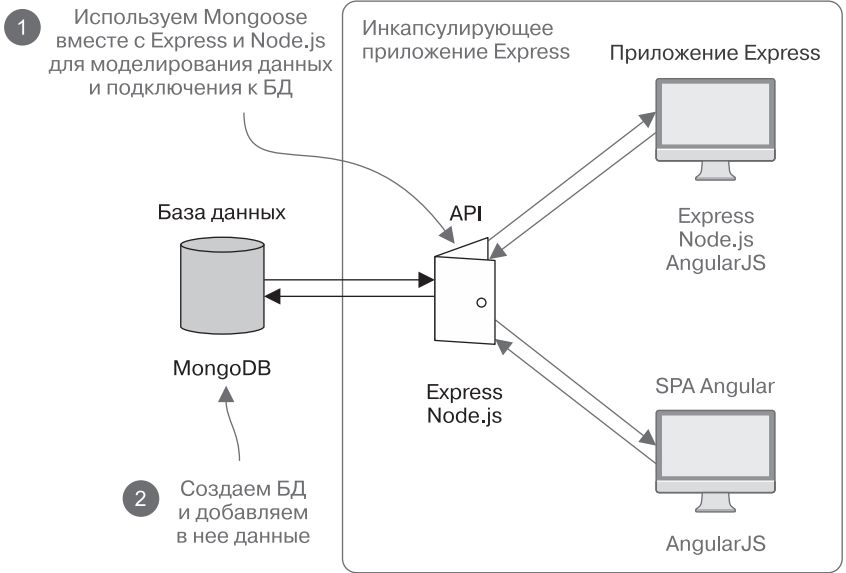


Рис. 5.3. База данных MongoDB и использование Mongoose в Express для моделирования данных и управления соединениями с БД

Мы, конечно, будем работать с базой данных MongoDB, но большая часть работы будет выполнена в Express и Node. В главе 2 мы обсуждали преимущества расщепления данных путем создания API перед тесной интеграцией их в основное приложение Express. Так что, хотя мы будем работать в Express и Node, причем по-прежнему внутри того же инкапсулирующего приложения, фактически мы начнем закладывать основы слоя API.

ПРИМЕЧАНИЕ

Чтобы следить за ходом изложения данной главы, вам понадобится установленная MongoDB. Если вы еще не установили ее, загляните в приложение А, чтобы получить инструкции.

Исходный код приложения по состоянию на конец данной главы доступен на GitHub, в ветви chapter-05. Выполните в терминале в новом каталоге следующие команды для его клонирования и установки зависимостей модулей npm:

```
$ git clone -b chapter-05 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

5.1. Подключение приложения Express к MongoDB с помощью Mongoose

Мы можем напрямую подключить наше приложение к MongoDB и заставить их взаимодействовать друг с другом посредством нативного драйвера. Хотя нативный драйвер MongoDB обладает широкими возможностями, работать с ним нелегко. К тому же у него нет встроенного способа описания и сопровождения структур данных. Mongoose предоставляет большую часть функциональности нативного драйвера, но более удобным способом, спроектированным в расчете на включение в технологические процессы разработки приложений.

В чем Mongoose действительно силен, так это в предоставляемых возможностях описания структур данных и моделей, их сопровождения и использования для взаимодействия с БД. И все это осуществляется непосредственно из кода приложения. Как часть подобного подхода, Mongoose включает способность добавлять проверки к описаниям данных, а значит, нам не нужно будет писать код валидации в каждом месте нашего приложения, где данные отправляются в БД.

Таким образом, Mongoose находит свое место в стеке внутри приложения Express в качестве транспортного соединения между приложением и БД (рис. 5.4).

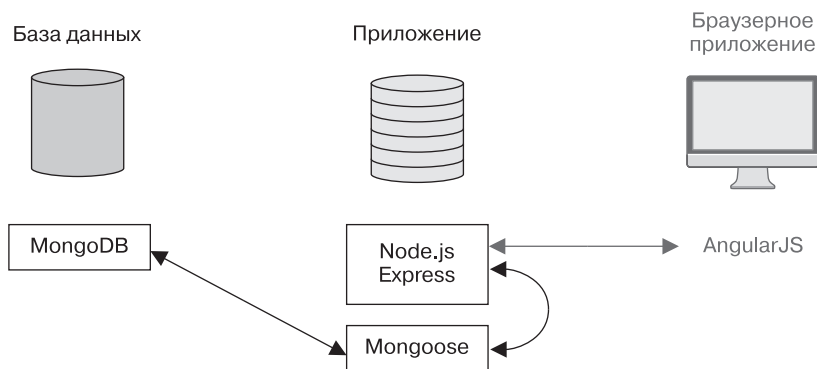


Рис. 5.4. Взаимодействие данных в стеке MEAN и место Mongoose в нем. Приложение Node/Express взаимодействует с MongoDB через Mongoose, а Node и Express могут затем также взаимодействовать с Angular

MongoDB обменивается данными только с Mongoose, а Mongoose, в свою очередь, — с Node и Express. AngularJS не будет непосредственно обмениваться данными с MongoDB или Mongoose, а только с приложением Express.

У вас на машине уже должна быть установлена MongoDB (рассматривается в приложении А), но не Mongoose. Mongoose не устанавливается глобально, а добавляется непосредственно в приложение. Сейчас мы это и сделаем.

5.1.1. Добавление Mongoose в приложение

MongoDB доступен в виде модуля npm. Как вы могли видеть в главе 3, самый быстрый и удобный способ установки модуля npm — через командную строку. Установить Mongoose и добавить его в список зависимостей в `package.json` можно с помощью одной команды.

Так что перейдите в терминал и убедитесь, что приглашение командной строки находится в корневом каталоге приложения, там же, где файл `package.json`, после чего выполните следующую команду:

```
$ npm install --save mongoose
```

Флаг `--save` сообщает npm о необходимости добавить Mongoose в список зависимостей в `package.json`. После выполнения этой команды вы увидите новый подкаталог `mongoose` в каталоге `node_modules` приложения, а раздел зависимостей файла `package.json` будет выглядеть следующим образом:

```
"dependencies": {  
  "express": "~4.9.0",  
  "body-parser": "~1.8.1",  
  "cookie-parser": "~1.3.3",  
  "morgan": "~1.3.0",  
  "serve-favicon": "~2.1.3",  
  "debug": "~2.0.0",  
  "jade": "~1.6.0",  
  "mongoose": "~3.8.20"  
}
```

Конечно, номера версий у вас могут слегка отличаться, но на момент написания данной книги последней стабильной версией Mongoose была 3.8.20. Теперь, когда мы установили Mongoose, подключим его к приложению.

5.1.2. Добавление в приложение соединения с Mongoose

На данном этапе подключим наше приложение к БД. Базу данных мы пока что не создали, но это не имеет значения, поскольку MongoDB создаст БД, когда мы впервые попытаемся ее использовать. Это может показаться немного странным, но представляет собой немалое преимущество при сборке приложения воедино: нам не приходится бросать код приложения и углубляться в другую среду.

Соединение MongoDB и Mongoose

При подключении к базе данных MongoDB Mongoose открывает пул из пяти пригодных для многократного использования соединений. Этот пул соединений совместно используется всеми запросами. Пять — просто количество по умолчанию, которое можно при необходимости увеличить или уменьшить в настройках соединений.

РЕКОМЕНДУЕМОЕ РЕШЕНИЕ

Открытие и закрытие соединений с БД может занять некоторое время, особенно если БД расположена на отдельном сервере или сервисе. Так что эти операции лучше выполнять только в случае необходимости. Рекомендуемое решение: открывать соединение с БД при запуске приложения и оставлять его открытым вплоть до перезапуска или остановки приложения. Именно этот подход мы и собираемся использовать.

Настройка файла соединений

Когда мы впервые занимались файловой структурой приложения, то создали в каталоге `app_server` три подкаталога: `models`, `views` и `controllers`. Работать с данными и моделями мы будем, основываясь в большей степени на каталоге `app_server/models`.

Настройка файла соединений представляет собой двухшаговый процесс: создание файла и запрос его в приложении, что позволит его использовать.

Первый шаг: создаем файл под названием `db.js` в `app_server/models` и сохраняем его. Теперь мы просто запрашиваем (`require`) Mongoose в этом файле с помощью следующей однострочной команды:

```
var mongoose = require( 'mongoose' );
```

Второй шаг: внедряем этот файл в приложение путем его запроса в `app.js`. Поскольку создание соединения между приложением и БД может занять довольно продолжительное время, лучше сделать это пораньше в настройках. Отредактируйте верхнюю часть `app.js` таким образом, чтобы она выглядела так, как показано в следующем фрагменте кода (изменения выделены полужирным шрифтом):

```
var express = require('express');  
var path = require('path');  
var favicon = require('serve-favicon');  
var logger = require('morgan');  
var cookieParser = require('cookie-parser');  
var bodyParser = require('body-parser');  
require('./app_server/models/db');
```

Мы не собираемся экспортировать какие-либо функции из `db.js`, так что не требуется присваивать его переменной при выполнении его `require`. Он понадобится нам в приложении, но вряд ли нужно будет обращаться к каким-либо его методам из `app.js`.

Если вы перезапустите приложение, оно должно работать как и раньше, но теперь с `Mongoose`. Если вы получите ошибку, убедитесь, что путь в операторе `require` соответствует пути к новому файлу, что `package.json` включает зависимость `Mongoose` и что вы выполнили команду `npm install` из терминала, находясь в корневом каталоге приложения.

Создание соединения Mongoose

Создание соединения `Mongoose` представляет собой просто объявление URI для вашей БД и передачу его методу `connect` `Mongoose`. URI базы данных представляет собой строку следующей конструкции:



Имя пользователя, пароль и порт не являются обязательными. Так что на локальной машине URI базы данных окажется довольно простым. Пока что, если предположить, что `MongoDB` установлена на вашей локальной машине, все, что нужно для создания соединения, — добавить следующий фрагмент кода в файл `db.js`:

```
var dbURI = 'mongodb://localhost/Loc8r';  
mongoose.connect(dbURI);
```

Если запустить приложение с этим кодом, добавленным в `db.js`, оно запустится и будет работать точно так же, как и раньше. Как же вам удостовериться, что соединение работает правильно? Ответ: с помощью событий соединений.

Мониторинг событий соединения с Mongoose

`MongoDB` публикует события на основе статуса соединения, и к ним можно очень легко подсоединиться, чтобы увидеть, что происходит. Мы будем использовать события, чтобы видеть, когда установлено соединение, когда возникают ошибки

и когда происходит разрыв соединения. Следующий фрагмент демонстрирует необходимый для этого код:

<pre>mongoose.connection.on('connected', function () { console.log('Mongoose connected to ' + dbURI); }); mongoose.connection.on('error',function (err) { console.log('Mongoose connection error: ' + err); }); mongoose.connection.on('disconnected', function () { console.log('Mongoose disconnected'); });</pre>	<table border="0"> <tr> <td style="border-left: 1px solid black; padding-left: 5px;">Мониторинг успешного соединения через Mongoose</td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;">Проверка на ошибку соединения</td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;">Проверка на событие разрыва соединения</td> </tr> </table>	Мониторинг успешного соединения через Mongoose	Проверка на ошибку соединения	Проверка на событие разрыва соединения
Мониторинг успешного соединения через Mongoose				
Проверка на ошибку соединения				
Проверка на событие разрыва соединения				

Если все это добавлено в `db.js`, при перезапуске приложения вы увидите, что в окне терминала будут выведены следующие подтверждения:

```
Express server listening on port 3000
Mongoose connected to mongodb://localhost/Loc8r
```

Если вы еще раз перезапустите приложение, то можете заметить отсутствие каких-либо сообщений о разрыве соединения. Это происходит потому, что соединения Windows не закрываются автоматически при остановке или перезапуске приложений. Для этого нам понадобится прослушивать изменения в процессах Node.

Закрытие соединения Mongoose

Закрытие соединения Mongoose при остановке приложения — почти столь же рекомендуемая практика, как и открытие соединения при его запуске. В силу двусторонности соединения (один его конец находится в вашем приложении, а второй — в MongoDB) MongoDB необходимо знать, когда вы хотите закрыть соединение, чтобы не держать открытыми лишние соединения.

Для контроля остановки приложения нам придется прослушивать процесс Node.js на предмет наступления события SIGINT.

ПРОСЛУШИВАНИЕ НА ПРЕДМЕТ СОБЫТИЯ SIGINT

SIGINT — сигнал уровня операционной системы, возбуждаемый в Unix-подобных операционных системах, таких как Linux и Mac OS X. Он возбуждается также в части последних версий Windows. Если вы работаете в Windows и события разрыва соединения не возбуждаются, можно их эмулировать. Чтобы эмулировать такое поведение в Windows, необходимо сначала добавить в приложение новый пакет npm — `readline`. Измените в своем файле `package.json` раздел зависимостей следующим образом:

```
"dependencies": {
  "express": "3.4.x",
```

```
"jade": "*",  
"mongoose": "3.8.x",  
"readline": "0.0.x"  
}
```

После этого установите его в приложение посредством выполнения `npm install` из командной строки в том же каталоге, где находится файл `package.json`.

В файле `db.js` перед кодом прослушвателя событий добавьте следующее:

```
var readline = require ("readline");  
if (process.platform === "win32"){  
  var rl = readline.createInterface ({  
    input: process.stdin,  
    output: process.stdout  
  });  
  rl.on ("SIGINT", function (){  
    process.emit ("SIGINT");  
  });  
}
```

Это сгенерирует сигнал `SIGINT` на машинах под управлением Windows, позволяя вам перехватить его и мягко закончить все, что нужно, до завершения процесса.

Если вы используете `nodemon` для автоматического перезапуска приложения, то вам придется прослушивать процесс Node на предмет еще одного события, а именно `SIGUSR2`. Heroku использует также событие `SIGTERM`, так что нужно будет слушать на предмет и его появления.

Перехват событий завершения процесса

При перехвате любого из этих событий мы мешаем нормальному ходу выполнения, так что необходимо гарантировать возобновление требуемого поведения вручную. После закрытия соединения `Mongoose`, конечно.

Для всего этого нам будут нужны три прослушвателя событий и одна функция для закрытия соединения с БД. Закрытие БД — асинхронная операция, так что понадобится передавать любую необходимую для перезапуска или завершения процесса Node функцию в качестве обратного вызова. Во время этого можно вывести в консоль сообщение, подтверждающее закрытие соединения и объясняющее его причину. Можно обернуть все это в функцию `gracefulShutdown` (мягкий останов) в `db.js`, как показано в следующем фрагменте кода:

```

var gracefulShutdown = function (msg, callback) {
  mongoose.connection.close(function () {
    console.log('Mongoose disconnected through ' + msg);
    callback();
  });
};

```

Определяем функцию для приема сообщений и функции обратного вызова

Выводим сообщение и выполняем обратный вызов после закрытия соединения Mongoose

Закрываем соединение Mongoose, передавая анонимную функцию, которая будет выполнена при закрытии

Теперь необходимо вызывать эту функцию при завершении приложения или когда его перезапускает nodemon. Чтобы это происходило, нужно добавить в `db.js` два прослушателя событий, как показано в следующем фрагменте кода:

```

process.once('SIGUSR2', function () {
  gracefulShutdown('nodemon restart', function () {
    process.kill(process.pid, 'SIGUSR2');
  });
});
process.on('SIGINT', function () {
  gracefulShutdown('app termination', function () {
    process.exit(0);
  });
});
process.on('SIGTERM', function() {
  gracefulShutdown('Heroku app shutdown', function () {
    process.exit(0);
  });
});

```

Прослушиваем на предмет SIGINT, генерируемого при завершении приложения

Прослушиваем на предмет используемого nodemon SIGUSR2

Отправляем сообщение функции gracefulShutdown и функцию обратного вызова для уничтожения процесса, снова генерируя SIGUSR2

Отправляем сообщение функции gracefulShutdown и функцию обратного вызова для выхода из процесса Node

Прослушиваем на предмет SIGTERM, генерируемого, когда Heroku останавливает процесс

Теперь при завершении приложения оно мягко закрывает соединение Mongoose до его фактического завершения. Аналогично, когда nodemon выполняет перезапуск приложения по причине изменений в исходных файлах, приложение

сначала закрывает текущее соединение Mongoose. Прослушиватель nodemon использует `process.once`, а не `process.on`, так как нам требуется выполнить прослушивание на предмет SIGUSR2 однократно. Nodemon также слушает на предмет того же события, и нам не нужно перехватывать его каждый раз, мешая nodemon работать.

СОВЕТ

Очень важно в каждом создаваемом вами приложении правильно управлять открытием и закрытием соединений с БД. Если в используемой вами среде сигналы завершения процессов другие, обязательно убедитесь, что прослушиваете на предмет их всех.

Законченный файл соединений

Мы многое добавили в файл `db.js`, так что воспользуемся моментом, чтобы подвести итоги. Ранее мы:

- ❑ описали строку соединения с БД;
- ❑ открыли соединение Mongoose при запуске приложения;
- ❑ выполнили мониторинг событий соединения Mongoose;
- ❑ выполнили мониторинг некоторых событий процессов Node, чтобы можно было закрыть соединение Mongoose при завершении приложения.

Целиком файл `db.js` будет выглядеть так, как показано в листинге 5.1. Обратите внимание на то, что он не включает дополнительного кода, необходимого Windows для генерации события SIGINT.

Листинг 5.1. Весь файл соединений с БД `db.js` в `app_server/models`

```
var mongoose = require( 'mongoose' );
var gracefulShutdown;
var dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI);

mongoose.connection.on('connected', function () {
  console.log('Mongoose connected to ' + dbURI);
});
mongoose.connection.on('error',function (err) {
  console.log('Mongoose connection error: ' + err);
});
mongoose.connection.on('disconnected', function () {
  console.log('Mongoose disconnected');
```

Задаем строку соединения с БД и используем ее для открытия соединения Mongoose

Прослушиваем на предмет событий соединения Mongoose и выводим состояния в консоль

```
});
```

```

gracefulShutdown = function (msg, callback) {
  mongoose.connection.close(function () {
    console.log('Mongoose disconnected through ' + msg);
    callback();
  });
};

```

Пригодная для повторного использования функция закрытия соединения Mongoose

```

// Для перезапуска nodemon
process.once('SIGUSR2', function () {
  gracefulShutdown('nodemon restart', function () {
    process.kill(process.pid, 'SIGUSR2');
  });
});
// Для завершения приложения
process.on('SIGINT', function() {
  gracefulShutdown('app termination', function () {
    process.exit(0);
  });
});
// Для завершения приложения Heroku
process.on('SIGTERM', function() {
  gracefulShutdown('Heroku app shutdown', function () {
    process.exit(0);
  });
});

```

Прослушиваем процессы Node на предмет сигналов завершения или перезапуска и вызываем в подходящий момент функцию gracefulShutdown, передавая функцию обратного вызова-продолжения

Как только вы сделали подобный файл, можете его копировать из приложения в приложение, ведь события, на предмет которых вы слушаете, всегда одни и те же. Все, что вам нужно будет сделать каждый раз, — поменять строку соединения с БД. Не забывайте, что мы также *запросили* этот файл в `app.js`, где он находится недалеко от верха, так что соединение открывается в самом начале существования приложения.

ИСПОЛЬЗОВАНИЕ НЕСКОЛЬКИХ БАЗ ДАННЫХ

То, что вы видели до сих пор, носит название соединения по умолчанию и хорошо подходит для одиночного соединения, открытого на протяжении всего времени работы приложения. Но если необходимо подключиться ко второй БД, например, для журналирования или управления сеансами пользователей, то можно использовать именованное соединение. При этом вместо метода `mongoose.connect` придется использовать другой метод, `mongoose.createConnection`, присвоив его переменной. Вы можете увидеть это в следующем фрагменте кода:

```

var dbURIlog = 'mongodb://localhost/Loc8rLog';
var logDB = mongoose.createConnection(dbURIlog);

```

Этот код создает объект для нового соединения Mongoose с именем logDB. Взаимодействовать с ним можно теми же способами, что и с mongoose.connection для соединения по умолчанию. Вот несколько примеров:

```
logDB.on('connected', function () {  
  console.log('Mongoose connected to ' + dbURIlog);  
});  
logDB.close(function () {  
  console.log('Mongoose log disconnected');  
});
```

Мониторинг события соединения для именованного соединения

Закрытие именованного соединения

5.2. Зачем нужно моделировать данные

В главе 1 мы говорили, что MongoDB скорее хранилище документов, а не традиционная табличная БД со строками и столбцами. Это дает MongoDB колоссальную свободу и гибкость, но иногда нам желательна, то есть необходима, структура для данных.

Возьмем, например, домашнюю страницу Loc8r. Раздел перечня, показанный на рис. 5.5, содержит определенный набор данных, общий для всех местоположений.

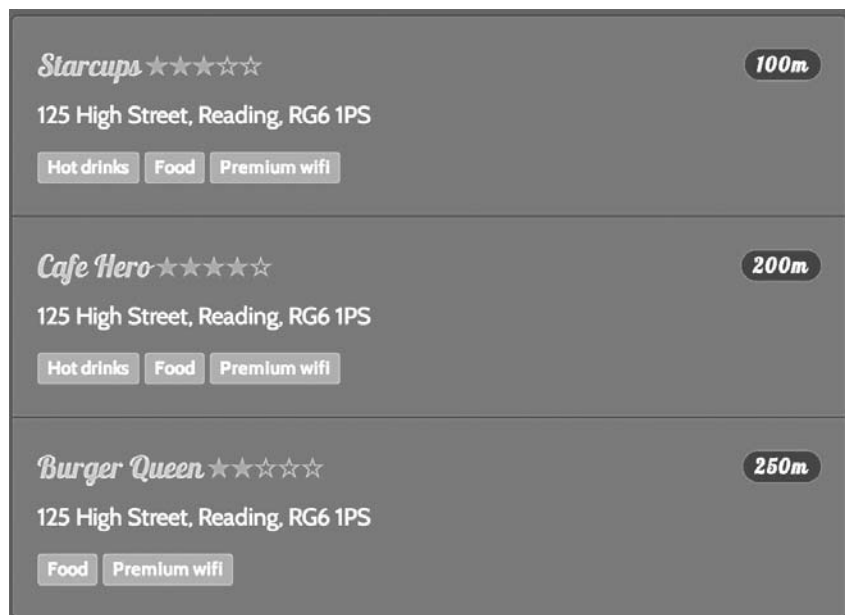


Рис. 5.5. У раздела перечня домашней страницы имеются четко определенные требования к данным и их структуре

Эти элементы данных нужны странице для всех мест, и у записей данных для каждого места должна быть согласованная структура именования. Без этого приложение не сможет найти данные и использовать их. На текущем этапе разработки данные хранятся в контроллере и передаются представлению. В терминах архитектуры MVC мы начинаем работу с данными в *представлении*, а затем передаем их на шаг назад, в *контроллер*. Теперь нам осталось только переместить их еще на один шаг назад, туда, где они должны находиться, — в *модель*. Текущее положение, подчеркивая конечную цель, иллюстрирует рис. 5.6.

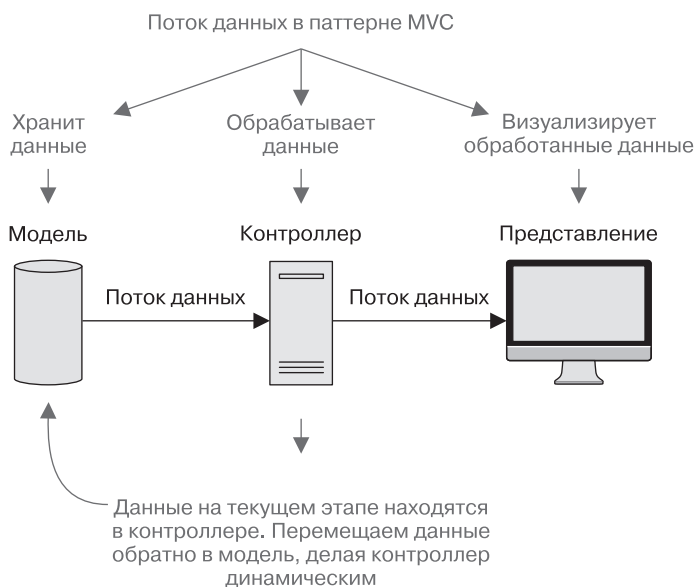


Рис. 5.6. Поток данных в паттерне MVC: из модели через контроллер в представление. На текущем этапе данные в прототипе находятся в контроллере, так что мы хотим переместить их на шаг назад, в модель

Одно из следствий перемещения данных назад по потоку выполнения MVC, шаг за шагом, как мы делали до сих пор, заключается в том, что благодаря этому выкристаллизовываются требования к структуре данных. Это гарантирует точное отражение структурой данных нужд нашего приложения. Если вы попытаетесь сначала описать модель, то закончите гаданием о том, как будет выглядеть приложение и как оно будет работать.

Так что, говоря о моделировании данных, мы на самом деле описываем свои пожелания относительно их структуры. В приложении мы можем или создать описания и управлять ими вручную и делать всю тяжелую работу самостоятельно, или использовать Mongoose, позволив ему делать тяжелую работу за нас.

5.2.1. Что такое Mongoose и как он работает

Mongoose был специально создан в качестве объектно-документного средства моделирования (Object-Document Modeler (ODM)) MongoDB для приложений Node. Один из ключевых принципов — возможность управления моделью данных изнутри вашего приложения. Не нужно возиться непосредственно с базами данных, внешними фреймворками или реляционными подпрограммами отображения — можно просто описать модель данных, находясь в комфортных условиях своего приложения.

Прежде всего договоримся относительно названий.

- В MongoDB каждая запись в БД называется *документом*.
- В MongoDB набор документов называется *коллекцией* (можете думать о ней как о таблице, если вам привычнее реляционные БД).
- В Mongoose описание документа называется *схемой*.
- Каждая отдельная сущность данных в схеме называется *путем*.

Эти соглашения о названиях и связь их друг с другом на примере стопки визитных карточек иллюстрирует рис. 5.7.

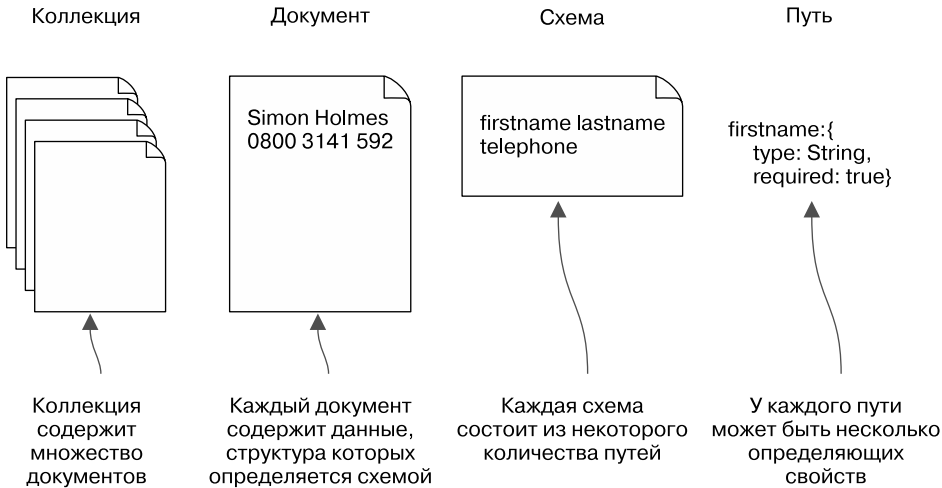


Рис. 5.7. Отношения между коллекциями, документами, схемами и путями в MongoDB и Mongoose на примере визитных карточек

Еще одно, последнее определение, для моделей. *Модель* — скомпилированная версия схемы. Все взаимодействия данных с помощью Mongoose проходят через модель. Мы поработаем с моделями в главе 6, а пока что сосредоточимся на их создании.

5.2.2. Как Mongoose моделирует данные

Если мы будем описывать наши данные в приложении, то каким образом? Конечно, на JavaScript! В объектах JavaScript, если быть точным. Мы уже взглянули на это краешком глаза на рис. 5.7, но давайте на примере простого документа MongoDB посмотрим, как может выглядеть схема Mongoose. В следующем фрагменте кода показан документ MongoDB, за которым следует схема Mongoose:

<pre>{ "firstname" : "Simon", "surname" : "Holmes", "_id" : ObjectId("52279effc62ca8b0c1000007") }</pre>		Пример документа MongoDB
<pre>{ firstname : String, surname : String }</pre>		Соответствующая схема Mongoose

Как вы видите, схема очень напоминает сами данные. Она описывает имя для каждого пути данных и содержащийся в нем тип данных. В данном примере мы просто объявляем пути `firstname` и `surname` как строковые значения.

О ПУТИ `_ID`

Вы могли обратить внимание на то, что мы не объявляли путь `ID` в схеме. `ID` — это уникальный идентификатор, первичный ключ, если такое название вам больше нравится, для каждого документа. MongoDB автоматически генерирует этот путь при создании каждого документа и присваивает ему уникальное значение `ObjectId`. Значение спроектировано с расчетом на то, чтобы оставаться уникальным при любых условиях посредством сочетания времени, прошедшего с момента начала отсчета времени Unix, с идентификаторами машины и процесса, а также счетчиком.

Можете использовать собственный уникальный ключ, если вам так удобнее, например, если у вас уже имеется БД. В нашей книге и в приложении `Loc8r` будем придерживаться варианта по умолчанию, `ObjectId`.

5.2.3. Анализ пути схемы

Базовая структура описания отдельного пути представляет собой название пути со следующим за ним объектом свойств. В коде выше вы видели фактически сокращенное написание для случая, когда нужно всего лишь описать тип данных для конкретного пути. Итак, путь схемы составляется из двух частей — названия пути и объекта свойств — следующим образом:

<code>firstname:</code>	<code>{type:String}</code>
Название пути	Объект свойств

РАЗРЕШЕННЫЕ ТИПЫ СХЕМ

Тип схемы — свойство, задающее тип данных для данного пути. Оно необходимо для всех путей. Если тип — единственное свойство пути, то можно использовать сокращенное описание. Существует восемь типов схем, которые вы можете использовать:

- `String` — любая строка в кодировке UTF-8;
- `Number` — Mongoose не поддерживает длинные числа или числа с двойной точностью, но допускает расширение для их поддержки с помощью плагинов Mongoose. Поддерживаемого по умолчанию типа достаточно в большинстве случаев;
- `Date` — обычно возвращается из MongoDB в виде объекта `ISODate`;
- `Boolean` — `true` или `false`;
- `Buffer` — для двоичной информации, например изображений;
- `Mixed` — любой тип данных;
- `Array` — может быть или массивом данных соответствующего типа, или массивом вложенных поддокументов;
- `ObjectId` — для уникального ID в пути, отличном от `_id`. Обычно используется для ссылок на пути `_id` в других документах.

При необходимости использовать другой тип схемы можете описать собственные типы схем или применять существующий плагин Mongoose из <http://plugins.mongoosejs.com/>.

Имя пути соответствует соглашениям и требованиям к описанию объектов JavaScript. Так что в нем не место пробелам или спецсимволам, и лучше стараться избегать зарезервированных слов. Лично я применяю для имен путей так называемый верблюжий регистр¹. Если вы используете существующую БД, то берите те имена путей, которые уже есть в документах. Если же создаете новую, учтите, что имена путей будут использоваться в документах, так что тщательно их продумывайте.

Объект свойств — объект JavaScript совершенно другого плана. Он задает характеристики хранящихся в пути данных. Как минимум он содержит тип данных, но может включать также характеристики для проверки, граничные условия, значения по умолчанию и т. д. Мы изучим и используем некоторые из этих возможностей в следующих нескольких главах в процессе превращения `Lo8g` в приложение, ориентированное на работу с данными.

А сейчас приступим к работе и начнем описывать схемы, которые понадобятся в приложении.

¹ См. <https://ru.wikipedia.org/wiki/CamelCase>. — *Примеч. пер.*

5.3. Описание простых схем Mongoose

Мы только что упоминали, что схема Mongoose, по существу, объект JavaScript, описываемый нами из приложения. Начнем с настройки и включения файлов, чтобы разделаться с этим и иметь возможность сосредоточиться на схеме.

Как вы, наверное, и подозревали, мы будем описывать схему в каталоге модели, рядом с `db.js`. По сути, мы собираемся запрашивать ее в `db.js`, чтобы сделать доступной для приложения. Итак, в каталоге моделей в `app_server` создайте новый пустой файл `locations.js`. Вам, естественно, понадобится Mongoose для описания схемы Mongoose, так что введите в `locations.js` следующую строку:

```
var mongoose = require( 'mongoose' );
```

Мы хотим внедрить этот файл в приложение путем запроса его в `db.js`, так что добавьте в самом конце `db.js` следующую строку:

```
require('./locations');
```

После этого настройка завершена и все готово к работе.

5.3.1. Основы настройки схемы

Mongoose предоставляет функцию-конструктор для описания новых схем, которую обычно присваивают переменной, чтобы можно было позднее к ней обратиться. Это выглядит так:

```
var locationSchema = new mongoose.Schema({ });
```

Фактически это именно та конструкция, которую мы собираемся использовать, так что добавьте эту строку в модель `locations.js`, конечно, ниже запрашивающей Mongoose строки. Пустой объект внутри скобок `mongoose.Schema({ })` — то место, где мы будем описывать схему.

Описание схемы на основе данных контроллера

Одно из следствий перемещения данных обратно из представления в контроллер — то, что контроллер в конечном счете дает хорошее представление о необходимой структуре данных. Начнем с простого и взглянем на контроллер `homelist` в `app_server/controllers/locations.js`. Контроллер `homelist` передает в представление данные для показа на домашней странице. То, как одно из мест будет выглядеть на домашней странице, демонстрирует рис. 5.8.

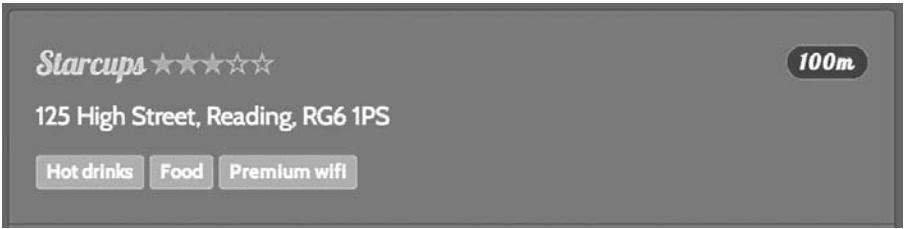


Рис. 5.8. Отображение отдельного места в списке на домашней странице

Следующий фрагмент кода показывает данные для этого места в том виде, в котором они находятся в контроллере:

```
locations: [{
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '100m'
}]
```

Annotations for the code above:
- name — строка
- address — еще одна строка
- rating — числовое значение
- facilities — массив строк

Мы вернемся к расстоянию чуть позже, когда понадобится его вычислять. В остальных четырех элементах данных ничего сложного нет: две строки, одно число и один массив строк. С уже имеющимися у вас знаниями можно использовать эту информацию для описания базовой схемы, как показано далее:

```
var locationSchema = new mongoose.Schema({
  name: String,
  address: String,
  rating: Number,
  facilities: [String]
});
```

Annotation for the code above:
1. Объявляем массив одного из типов схемы путем объявления этого типа внутри квадратных скобок

Обратите внимание на простой способ объявления предоставляемых услуг как массива 1. Если массив содержит лишь один из типов схемы, такой как `String`, его можно описать просто путем заключения соответствующего типа схемы в квадратные скобки.

Присвоение значений по умолчанию

В некоторых случаях удобно задавать значение по умолчанию при создании на основе схемы нового документа MongoDB. Хорошая кандидатура для этого — путь `rating` в схеме `locationSchema`. При добавлении в БД нового местоположения у него не будет никаких отзывов, а значит, и оценки. Но наше представление предполагает оценку от нуля до пяти звезд, которую должен будет передавать контроллер.

Поэтому хотелось бы задать для каждого нового документа значение оценки по умолчанию, равное 0. Mongoose позволяет делать это изнутри схемы. Помните, что запись `rating: Number` — сокращение для `rating: {type: Number}`? Так вот, можно задать и другие необязательные элементы в объекте определения, включая и значение по умолчанию. Это значит, что можно изменить путь для оценки в схеме следующим образом:

```
rating: {type: Number, "default": 0}
```

Слово `default` не обязательно брать в кавычки, но это зарезервированное слово в JavaScript, так что стоит сделать это.

Добавление возможности базовой проверки данных: обязательные поля

С помощью Mongoose можно быстро добавить возможность определенной базовой проверки правильности данных на уровне схемы. Это помогает поддерживать целостность данных и защищать БД от проблем с отсутствующими и деформированными данными. Вспомогательные методы Mongoose сильно облегчают эту задачу, а значит, вам не нужно будет всякий раз писать или импортировать код.

Первый пример подобных проверок гарантирует, что указанные поля не пусты, перед сохранением документа в БД. Вместо того чтобы писать в коде команды для проверки каждого поля, можно просто добавить флаг `required: true` к тем объектам определений каждого из путей, которые должны, по вашему мнению, быть обязательными. В схеме `locationSchema` необходимо, чтобы у каждого пути было название, так что можно поменять название пути следующим образом:

```
name: {type: String, required: true}
```

Если вы теперь попытаетесь сохранить местоположение без названия, Mongoose вернет ошибку проверки, которую можно будет сразу же перехватить в коде, не обращаясь к БД.

Добавление возможности базовой проверки данных: граничные условия для числовых значений

Можно использовать схожий метод для задания максимального и минимального значений числового пути. Эти валидаторы называются `max` и `min`. Каждому местоположению присваивается оценка, для которой мы только что задали значение по умолчанию, равное 0. Значение не должно быть меньше 0 или больше 5, так что можно поменять путь `rating` следующим образом:

```
rating: {type: Number, "default": 0, min: 0, max: 5}
```

С этим изменением Mongoose не позволит вам сохранить значение оценки меньше 0 или больше 5. Он вернет ошибку проверки, которую вы сможете обработать в своем коде. Замечательное свойство этого подхода — то, что приложению не нужно будет обращаться к БД для проверки граничных условий. Еще один плюс: не нужно писать код проверки во всех местах приложения, где может понадобиться добавить, обновить или вычислить значение оценки.

5.3.2. Использование географических данных в MongoDB и Mongoose

Когда мы только приступили к отображению данных приложения из контроллера на схему Mongoose, мы оставили вопрос о расстоянии на потом. Теперь пришло время обсудить, как мы будем обрабатывать географическую информацию.

MongoDB может хранить географические данные в виде координат долготы и широты и даже создавать на их основе *индексы* и управлять ими. Эта способность дает пользователям возможность выполнять быстрый поиск мест, расположенных близко друг от друга или поблизости от конкретных долготы и широты. Безусловно, это очень полезно для создания приложения, основанного на учете местоположения!

ОБ ИНДЕКСАХ В MONGODB

Индексы в любой системе баз данных позволяют выполнять более быстрые и эффективные запросы, и MongoDB не исключение. Если путь проиндексирован, MongoDB может использовать индекс, чтобы быстро отобрать подмножества данных, не просматривая все документы в коллекции.

Рассмотрим систему хранения документов, которая могла бы использоваться у вас дома. Представьте, что вам нужно найти конкретную выписку по счету банковской карты. Допустим, вы храните все бумаги в одном ящике шкафа. Если они помещены туда случайным образом, вам придется просмотреть все виды не относящихся к делу документов, прежде чем найдете искомое. Если же вы «проиндексировали» свои бумаги — разложили по папкам, то сможете быстро найти папку, относящуюся к банковской карте. Как только вытащите ее, вам нужно будет просмотреть один-единственный набор документов, что значительно повысит эффективность поиска.

Это очень похоже на то, как работает индексация в БД. Однако в БД у вас может быть несколько индексов для каждого документа, что делает возможным эффективный поиск по различным запросам.

Индексы, однако, требуют расхода ресурсов БД на сопровождение точно так же, как корректное распределение ваших бумаг по папкам требует времени. Так что для улучшения общей производительности старайтесь ограничивать индексы БД теми путями, которые действительно требуют индексации и используются в большинстве запросов.

Данные для отдельных географических местоположений хранятся в соответствии со спецификациями формата GeoJSON, который мы вскоре увидим в действии. Поддержка Mongoose этого типа данных позволяет вам описывать в схеме геопространственные пути. Mongoose, будучи слоем абстракции поверх MongoDB, старается упростить эти действия. Для добавления пути GeoJSON в вашу схему нужно всего лишь:

- описать путь в виде массива данных типа `Number`;
- описать путь как такой, у которого имеется индекс `2dsphere`.

Чтобы осуществить это, можете добавить путь `coords` в свою схему для местоположений. Если вы выполнили два названных шага, ваша схема должна выглядеть так, как показано в следующем фрагменте кода:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'}
});
```

Главное здесь — `2dsphere`, так как именно это значение дает MongoDB возможность проводить правильные вычисления при выполнении запросов и возвращении результатов. Мы ближе познакомимся с этим процессом в главе 6, когда будем создавать API и начнем взаимодействие с данными.

СОВЕТ

Для соответствия спецификациям GeoJSON пара координат должна вводиться в массив в правильном порядке: сначала долгота, потом широта.

Теперь мы рассмотрели все основы, и в нашей схеме для `Loc8r` уже есть все необходимое для удовлетворения выдвигаемых домашней страницей требований. Пришло время взглянуть на страницу `Details` (Подробности). Требования к данным у этой страницы более сложные, так что посмотрим, как удовлетворить их с помощью схем Mongoose.

5.3.3. Создание более сложных схем с поддокументами

Данные, которые мы применяли до сих пор, были более или менее простыми, их можно было хранить в довольно плоской схеме. Мы использовали пару массивов для предоставляемых услуг и координат местоположений, но опять же эти массивы были простыми и содержали всего по одному типу данных каждый.

Теперь мы посмотрим, что происходит в процессе работы с несколько более сложным набором данных.

Начнем с повторного знакомства со страницей **Details** (Подробности) и выводимыми на ней данными. Рисунок 5.9 демонстрирует скриншот этой страницы и показывает все различные области информации.

Наименование, оценка и адрес находятся в самом верху страницы, чуть ниже описываются предоставляемые услуги. Справа находится карта, основанная на географических координатах. Все это мы уже охватили в базовой схеме. Две области, для которых у нас все еще ничего нет: *часы работы* и *отзывы клиентов*.

Данные, которыми снабжается это представление, в настоящий момент хранятся в контроллере `locationInfo` в `app_server/controllers/locations.js`. Листинг 5.2 показывает соответствующую часть данных в этом контроллере.

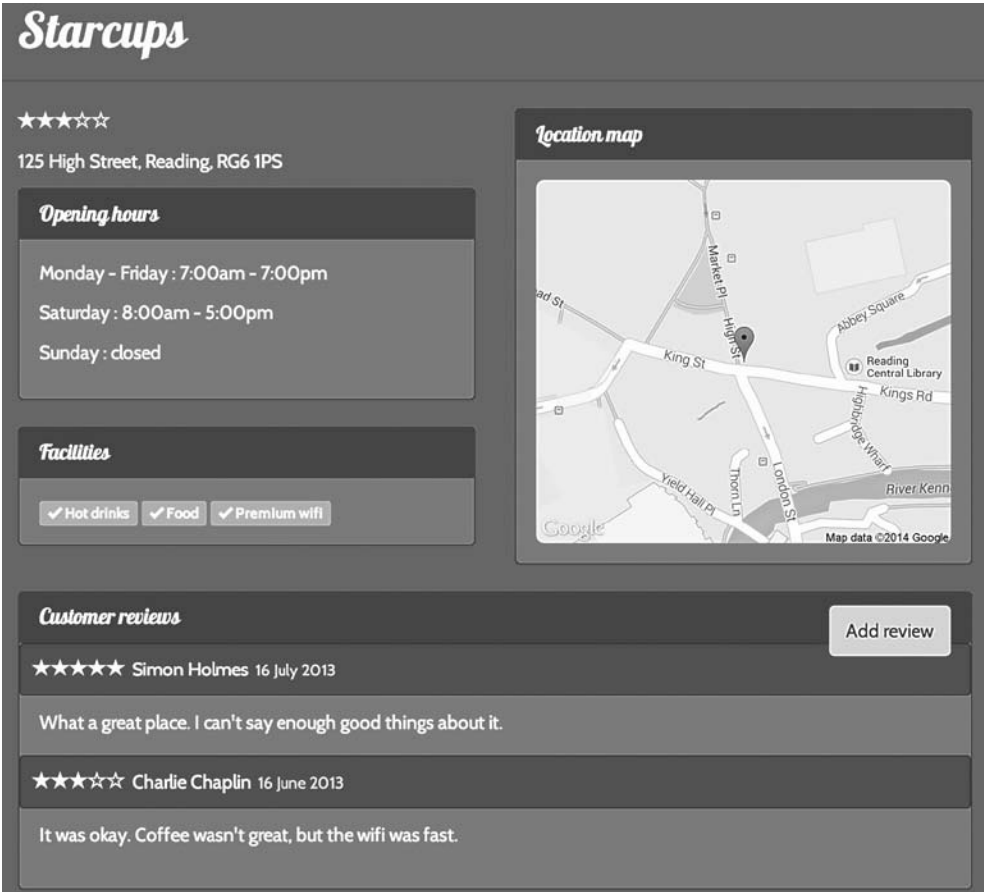


Рис. 5.9. Информация, отображаемая для отдельного места на странице **Details** (Подробности)

Листинг 5.2. Данные в контроллере, предназначенные для страницы Details (Подробности)

```
location: {
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: {lat: 51.455041, lng: -0.9690884},
  openingTimes: [{
    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  },{
    days: 'Saturday',
    opening: '8:00am',
    closing: '5:00pm',
    closed: false
  },{
    days: 'Sunday',
    closed: true
  }],
  reviews: [{
    author: 'Simon Holmes',
    rating: 5,
    timestamp: '16 July 2013',
    reviewText: 'What a great place. I can\'t say
enough good things about it.'
  },{
    author: 'Charlie Chaplin',
    rating: 3,
    timestamp: '16 June 2013',
    reviewText: 'It was okay. Coffee wasn\'t great,
but the wifi was fast.'
  }]
}
```

Уже охвачено
существующей схемой

Данные о часах работы
хранятся в виде
массива объектов

Отзывы также
передаются
представлению в виде
массива объектов

Таким образом, здесь имеются массивы объектов для часов работы и отзывов. В реляционной БД их можно было бы создать в виде отдельных таблиц, а затем выполнить их соединение (`join`) в запросе, когда понадобится хранящаяся в них информация. Но документоориентированные БД, включая MongoDB, работают иначе. В документоориентированной БД все относящееся к родительскому документу должно содержаться *внутри* этого документа. Концептуальные различия между этими двумя подходами демонстрирует рис. 5.10.

MongoDB вводит понятие *поддокументов*, предназначенных для хранения этих повторяющихся вложенных данных. Поддокументы очень похожи на документы: у них есть собственная схема и при создании MongoDB присваивает каждому из них уникальный `_id`. Но поддокументы должны быть вложены в документ, и доступ к ним возможен только по пути через этот родительский документ.

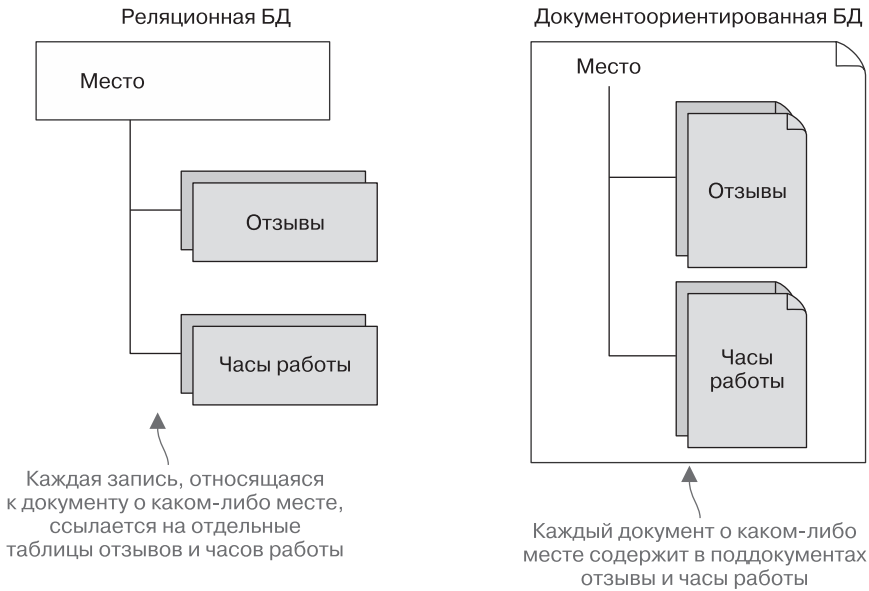


Рис. 5.10. Различия в том, как хранят относящуюся к родительскому элементу повторяющуюся информацию реляционные и документоориентированные БД

Вложенные схемы в Mongoose для описания поддокументов

Поддокументы описываются в Mongoose с помощью вложенных схем. Это значит, что одна схема вложена внутрь другой. Давайте создадим такую схему, чтобы увидеть, как это работает в коде. Первый шаг — описание новой схемы для поддокумента. Начнем с часов работы и создадим следующую схему. Обратите внимание на то, что это должно находиться в том же файле, что и описание схемы `locationSchema`, причем, что существенно, *перед* этим описанием:

```
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
```

ВАРИАНТЫ ХРАНЕНИЯ ИНФОРМАЦИИ О ВРЕМЕНИ

В схеме для часов работы мы сталкиваемся с интересной ситуацией: нам необходимо хранить информацию о времени, например, 7:30 a.m., но без связанной с ней даты.

Мы используем тут тип `String`, как не требующий никакой обработки перед помещением в БД или после извлечения из нее. При этом каждая запись легко понятна. Недостаток его — в сложности выполнения с ним каких-либо вычислений.

Один из вариантов — создать объект даты с произвольными данными и вручную задать часы и минуты, вот так:

```
var d = new Date();
d.setHours(15);
d.setMinutes(30);
```

Значение d теперь равно
Wed Apr 09 2014 15:30:40 GMT+0100 (BST)

С помощью такого метода мы легко можем отделить время от даты. Недостаток его — хранение ненужных данных и формальная некорректность.

Второй вариант: хранение количества минут после полуночи. Так, 7:30 a.m. равно $(7 \cdot 60) + 30 = 450$. Эти вычисления достаточно просты для того, чтобы можно было выполнять их при помещении данных в БД и при их извлечении из нее. Но на первый взгляд эти данные выглядят бессмысленно.

Чтобы сделать даты интеллектуальными, лично я предпочел бы второй вариант. Если вы хотите попробовать что-то новое, это будет неплохим упражнением. Но в целях удобства и чтобы не отвлекаться, мы продолжим использовать в данной книге метод, связанный со String.

Описание схемы по-прежнему довольно простое и устанавливает соответствие для данных из контроллера. У нас есть два обязательных поля, булев флаг `closed` и `days`, на которые ссылается каждый поддокумент.

Вложение этой схемы в схему местоположений — еще одна несложная задача. Необходимо только добавить новый путь в родительскую схему и описать его как массив схем поддокументов. Следующий фрагмент кода демонстрирует, как можно вложить `openingTimeSchema` в `locationSchema`:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema]
});
```

Добавляем вложенную схему, ссылаясь на объект другой схемы как массив

Сделав это, мы можем теперь добавлять к данному месту нужное количество поддокументов о часах работы, которые будут храниться в соответствующем этому месту документе. В следующем фрагменте кода приведен пример документа из MongoDB, основанного на этой схеме (поддокументы для часов работы выделены полужирным шрифтом):

```
{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f5"),
  "name": "Starcups",
  "address": "125 High Street, Reading, RG6 1PS",
  "rating": 3,
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
```

```

"coords": [-0.9690884, 51.455041],
"openingTimes": [{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f6"),
  "days": "Monday - Friday",
  "opening": "7:00am",
  "closing": "7:00pm",
  "closed": false
}, {
  "_id": ObjectId("52ef3a9f79c44a86710fe7f7"),
  "days": "Saturday",
  "opening": "8:00am",
  "closing": "5:00pm",
  "closed": false
}, {
  "_id": ObjectId("52ef3a9f79c44a86710fe7f8"),
  "days": "Sunday",
  "closing": "5:00pm",
  "closed": true
}]
}

```

В MongoDB вложенные поддокументы для часов работы располагаются внутри документа для конкретного места

Разделавшись со схемой для часов работы, мы можем двигаться дальше и взглянуть на схему поддокументов для отзывов.

Добавление второго набора поддокументов

Ни MongoDB, ни Mongoose не ограничивают количество путей поддокументов в документе. А значит, мы можем использовать сделанное для часов работы, продублировав процесс для отзывов.

Шаг первый: изучаем данные, использованные в отзыве, показанном в следующем фрагменте кода:

```

{
  author: 'Simon Holmes',
  rating: 5,
  timestamp: '16 July 2013',
  reviewText: 'What a great place. I can\'t say enough good things about it.'
}

```

Шаг второй: описываем соответствующую новую схему `reviewSchema` в `app_server/models/location.js`:

```

var reviewSchema = new mongoose.Schema({
  author: String,
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdAt: {type: Date, "default": Date.now}
});

```

Шаг третий: добавляем эту схему `reviewSchema` как новый путь в схему `locationSchema`:

```

var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},

```

```

address: String,
rating: {type: Number, "default": 0, min: 0, max: 5},
facilities: [String],
coords: {type: [Number], index: '2dsphere'},
openingTimes: [openingTimeSchema],
reviews: [reviewSchema]
});

```

После описания схемы для отзывов и добавления ее к основной схеме для местоположений мы получили все необходимое для хранения данных обо всех местах в структурированном виде.

5.3.4. Окончательная схема

На протяжении этого раздела мы многое сделали внутри файла, так что посмотрим на все это вместе и разберемся, что к чему. В листинге 5.3 показано содержимое файла `locations.js` из каталога `app_server/models`, описывающего схему для данных о местах.

Листинг 5.3. Окончательное описание схемы для мест, включая вложенные схемы

```

var mongoose = require( 'mongoose' );
var reviewSchema = new mongoose.Schema({
  author: String
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdAt: {type: Date, default: Date.now}
});
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema],
  reviews: [reviewSchema]
});

```

Запрашиваем Mongoose, чтобы можно было использовать его методы

Описываем схему для отзывов

Описываем схему для часов работы

Начало описания основной схемы для места

Используем 2dsphere, чтобы добавить поддержку пар координат GeoJSON долготы/широта

Ссылаемся на схемы для часов работы и отзывов для добавления вложенных поддокументов

У всех документов и поддокументов имеются схемы, описывающие их структуры. Кроме того, мы добавили некоторые значения по умолчанию и базовые проверки правильности данных. Чтобы все это стало для вас понятнее, рассмотрите листинг 5.4, который демонстрирует пример основанного на этой схеме документа MongoDB.

Листинг 5.4. Пример документа MongoDB, основанного на схеме для мест

```
{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f5"),
  "name": "Starcups",
  "address": "125 High Street, Reading, RG6 1PS",
  "rating": 3,
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "coords": [-0.9690884, 51.455041],
  "openingTimes": [{
    "_id": ObjectId("52ef3a9f79c44a86710fe7f6"),
    "days": "Monday - Friday",
    "opening": "7:00am",
    "closing": "7:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f7"),
    "days": "Saturday",
    "opening": "8:00am",
    "closing": "5:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f8"),
    "days": "Sunday",
    "closed": true
  }
  ],
  "reviews": [{
    "_id": ObjectId("52ef3a9f79c44a86710fe7f9"),
    "author": "Simon Holmes",
    "rating": 5,
    "createdOn": ISODate("2013-07-15T23:00:00Z"),
    "reviewText": "What a great place. I can't say
enough good things about it."
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7fa"),
    "author": "Charlie Chaplin",
    "rating": 3,
    "createdOn": ISODate("2013-06-15T23:00:00Z"),
    "reviewText": "It was okay. Coffee wasn't great,
but the wifi was fast."
  }
  ]
}
```

← Координаты хранятся в виде пар GeoJSON [долгота, широта]

Часы работы хранятся в виде вложенного массива объектов-поддокументов

Отзывы тоже массив поддокументов

Это все должно дать вам представление о том, как выглядит основанный на известной схеме документ MongoDB, включая поддокументы. В подобном удобочитаемом виде это объект JSON, хотя формально MongoDB хранит его как BSON, то есть двоичный JSON (Binary JSON).

5.3.5. Компиляция схем MongoDB в модели

Во время работы с данными приложение не взаимодействует непосредственно со схемой — взаимодействие происходит через модели.

Модель в Mongoose — это скомпилированная версия схемы. Однократно скомпилированный, отдельный экземпляр модели отображается непосредственно на отдельный документ в БД. Именно благодаря этому взаимно однозначному соответствию модель может создавать, читать, сохранять и удалять данные. Рисунок 5.11 иллюстрирует эту схему.

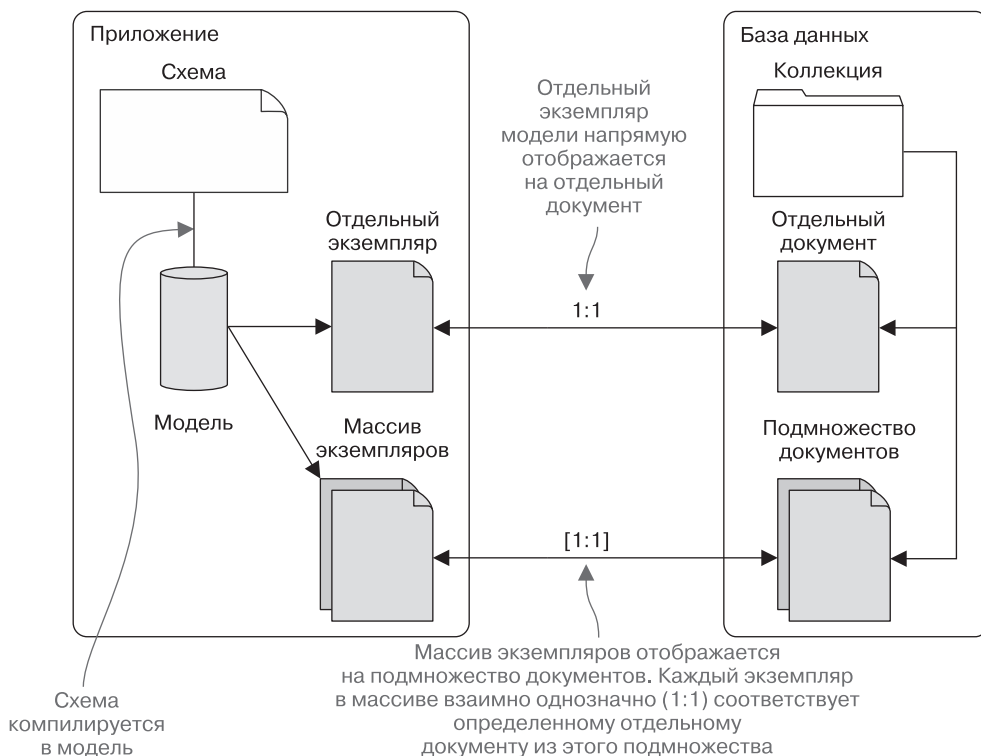


Рис. 5.11. Приложение и БД взаимодействуют друг с другом посредством моделей. Отдельный экземпляр модели взаимно однозначно (1:1) соответствует отдельному документу в БД. Благодаря этому соответствию происходит управление созданием, чтением, обновлением и удалением данных

Компиляция модели из схемы

Все, что содержит в названии слово «компиляция», кажется несколько запутанным. На самом деле компиляция модели Mongoose из схемы — чрезвычайно простая задача, на одну строчку. Нужно только убедиться, что схема закончена, прежде чем вызывать команду `model`. Структура команды `model` выглядит следующим образом:

```
mongoose.model('Location', locationSchema, 'Locations');
```

Название соединения Название модели Используемая схема Название коллекции MongoDB (необязательный параметр)

СОВЕТ

Название коллекции MongoDB — необязательный элемент. Если вы его не укажете, MongoDB использует название модели в нижнем регистре в форме множественного числа. Например, если модель называется `Location` и вы не зададите иное имя, коллекция будет называться `locations`.

Поскольку мы создаем БД и не обращаемся к существующему источнику данных, то можем использовать название коллекции по умолчанию, так что нам нет необходимости включать этот параметр в команду `model`. Таким образом, для создания модели схемы мест мы можем добавить к коду следующую строку, поставив ее сразу после описания схемы `locationSchema`:

```
mongoose.model('Location', locationSchema);
```

Готово! Мы рассмотрели схему данных для мест и скомпилировали эту схему в модель, которую сможем использовать в приложении. Теперь нам нужно добавить какие-нибудь данные.

5.4. Использование командной оболочки MongoDB для создания базы данных MongoDB и добавления данных

Для построения приложения `Loc8r` мы создадим новую БД и вручную добавим какие-нибудь тестовые данные. Это значит, что у вас есть возможность создать собственную версию `Loc8r` для тестирования и в то же время непосредственно поэкспериментировать с MongoDB.

5.4.1. Основы командной оболочки MongoDB

Командная оболочка MongoDB — утилита командной строки, устанавливаемая вместе с MongoDB и предоставляющая возможность взаимодействия с любыми базами данных MongoDB в вашей системе. Она обладает большими возможностями и может выполнять многие вещи — мы углубимся в основы лишь до той степени, которая необходима, чтобы начать работу.

Запуск командной оболочки MongoDB

Войдите в командную оболочку, выполнив в терминале следующую строку:

```
$ mongo
```

При этом в терминале должна появиться примерно следующая пара строк, сообщающая версию командной оболочки и подтверждающая подключение к тестовой БД:

```
MongoDB shell version: 2.4.6  
connecting to: test
```

СОВЕТ

Новые строки в командной оболочке начинаются с `>`, чтобы можно было отличить их от стандартной точки входа командной строки. Приведенные в данном разделе команды командной оболочки будут начинаться с `>` вместо `$`, чтобы сделать очевидным использование нами командной оболочки, но вводить символ `>`, как и `$`, не нужно.

Список локальных баз данных

Следующая простая команда предназначена для вывода списка всех локальных баз данных MongoDB. Введите в командной оболочке строку:

```
> show dbs
```

Эта команда вернет список названий и размеров локальных баз данных MongoDB. Если вы пока что не создали никаких БД, то все равно увидите две БД по умолчанию, что-то вроде такого:

```
local    0.078125GB  
test     (empty)
```

Использование конкретной базы данных

При запуске командная оболочка MongoDB автоматически подключается к пустой тестовой БД. Если вы хотите переключиться на другую базу данных, например на БД по умолчанию под названием `local`, можете воспользоваться следующей командой:

```
> use local
```

В ответ на эту команду командная оболочка выведет сообщение:

```
switched to db local
```

Это сообщение подтверждает название БД, к которой подключилась командная оболочка.

Получение списка коллекций в базе данных

Как только вы подключитесь к конкретной БД, можете без труда вывести список коллекций с помощью следующей команды:

```
> show collections
```

Если вы используете локальную БД, то, вероятно, увидите в ответ одно выведенное в терминал название коллекции: `startup_log`.

Просмотр содержимого коллекции

Командная оболочка MongoDB позволяет выполнять запросы к коллекциям в БД. Структура запроса (операции поиска) следующая:

```
db.collectionName.find(queryObject)
```

└───┬───┘ └───┬───┘
 | |
 | |
 └───┬───┘ └───┬───┘
 | |
 | |
 | |

Задаёт название Необязательный объект
коллекции, к которой с параметрами запроса
выполняется запрос

Объект `query` используется для конкретизации того, что вы хотите найти в коллекции, и позднее, в главе 6, мы рассмотрим примеры объектов `query` (Mongoose тоже использует объекты `query`). Простейший запрос — пустой запрос, возвращающий все документы коллекции. Не волнуйтесь, что коллекция может быть большой, MongoDB вернет подмножество документов, которое вы сможете просматривать постранично. Используя коллекцию `startup_log` в качестве примера, вы можете выполнить следующую команду:

```
> db.startup_log.find()
```

Она вернет некоторое количество документов из журнала загрузки MongoDB, содержимое которых недостаточно интересно, чтобы приводить его тут. Эта команда полезна при запуске БД и при необходимости убедиться в том, что все сохранено так, как ожидается.

5.4.2. Создание базы данных MongoDB

На самом деле вам не нужно *создавать* базу данных MongoDB, достаточно просто начать ее использовать. Для приложения `Loc8r` имеет смысл назвать базу данных `Loc8r`. Так что начните использовать ее в командной оболочке следующей командой:

```
> use Loc8r
```

Если вы выполните команду `show collections`, она пока ничего не вернет, но если выполнить `show dbs`, можно увидеть, что база данных была добавлена в список БД и в настоящий момент пуста:

```
Loc8r    (empty)
local    0.078125GB
test     (empty)
```

Данное сообщение показывает, что она была добавлена в список баз данных.

Создание коллекций и документов

Аналогично вам не требуется явным образом создавать коллекции, так как MongoDB создаст их для вас при первом сохранении в них данных.

БОЛЕЕ БЛИЗКИЕ ВАМ ДАННЫЕ О МЕСТАХ

Loc8r в первую очередь связана с данными, основанными на учете местоположения, все примеры — вымышленные места, расположенные неподалеку от тех мест, где я жил в Великобритании. Вы можете сделать свою версию более близкой себе, изменив названия, адреса и координаты.

Чтобы узнать свои координаты, можете посетить сайт <http://whatsmylatlng.com/>. На странице имеется кнопка для определения местоположения с помощью JavaScript, которая даст вам намного более точное местоположение, чем первоначальное приближение. Обратите внимание на то, что координаты выдаются вам в порядке «широта/долгота», так что вам необходимо будет поменять их местами для БД, так как там долгота должна указываться первой.

Чтобы получить координаты любого адреса, можно использовать <http://mygeoposition.com/>. Этот сайт дает возможность ввести адрес или перетащить указатель для получения географических координат. И не забывайте, что пары в MongoDB должны выглядеть следующим образом: долгота, затем широта.

Для модели `Location` вам понадобится коллекция `locations`; не забывайте, что название коллекции по умолчанию — это название модели в нижнем регистре в форме множественного числа. Создать и сохранить новый документ можно путем передачи объекта данных команде `save` коллекции, как показано в следующем фрагменте кода:

```
> db.locations.save({
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: [-0.9690884, 51.455041],
  openingTimes: [{
```

← Обратите внимание,
что название коллекции
задано в виде части команды `save`

```

    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  }, {
    days: 'Saturday',
    opening: '8:00am',
    closing: '5:00pm',
    closed: false
  }, {
    days: 'Sunday',
    closed: true
  }
]
})

```

Этот код за один шаг создаст новую коллекцию `locations`, а также первый документ в этой коллекции. Если вы теперь выполните `show collections` в командной оболочке MongoDB, то увидите, что возвращается новая коллекция `locations` вместе с автоматически созданной коллекцией `system.indexes`, например:

```

> show collections
locations
system.indexes

```

Вы можете выполнить запрос к коллекции для поиска всех документов — пока что там только один, так что будет возвращено не так уж много информации. Можно применить к коллекции также команду `find`:

```

> db.locations.find()
{
  "_id": ObjectId("530efe98d382e7fa4345f173"),
  "address": "125 High Street, Reading, RG6 1PS",
  "coords": [-0.9690884, 51.455041],
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "name": "Starcups",
  "openingTimes": [{
    "days": "Monday - Friday",
    "opening": "7:00am",
    "closing": "7:00pm",
    "closed": false
  }, {
    "days": "Saturday",
    "opening": "8:00am",
    "closing": "5:00pm",
    "closed": false
  }, {
    "days": "Sunday",
    "closing": true
  }
],
  "rating": 3,
}

```

Не забудьте, что нужно использовать команду `find` применительно к самой коллекции

MongoDB автоматически добавила уникальный идентификатор для этого документа

Этот фрагмент кода был отформатирован для удобства чтения, возвращенный MongoDB в командную оболочку документ не имел бы разрывов строк и отступов. Но командная оболочка MongoDB может приукрасить его для вас, если добавить `.pretty()` в конец команды, вот так:

```
> db.locations.find().pretty()
```

Обратите внимание на то, что порядок данных в возвращенном документе не совпадает с порядком данных в предоставленном вами объекте. Так как эта структура данных не основана на столбцах, не имеет значения, каким образом MongoDB хранит отдельные пути в документе. Данные все равно всегда располагаются в правильных путях, а хранимые в массивах данные всегда сохраняют один и тот же порядок.

Добавление поддокументов

Вероятно, вы обратили внимание на то, что в нашем первом документе набор данных был неполон — в нем отсутствовали поддокументы отзывов. Можно добавить их в исходную команду `save`, как мы поступили с часами работы, или обновить существующий документ и внести их туда.

В MongoDB имеется команда `update`, принимающая два входных параметра, первый из которых — запрос, так что она знает, какой документ обновлять, а второй содержит инструкции о том, что делать после нахождения документа. На данном этапе мы можем использовать совсем простой запрос и искать место по названию (Starcups), так как уверены в отсутствии дублирующихся записей. Для объекта с инструкциями можно использовать команду `$push`, чтобы добавить новый объект в путь отзывов, — не имеет значения, что пути отзывов может еще не существовать, MongoDB добавит его в качестве части операции `push`.

Все вместе это будет выглядеть как в следующем фрагменте кода:

```
> db.locations.update({ name: 'Starcups' }, { $push: { reviews: { author: 'Simon Holmes', id: ObjectId(), rating: 5, timestamp: new Date("Jul 16, 2013"), reviewText: "What a great place. I can't say enough good things about it." } } })
```

Начинаем с объекта запроса для поиска нужного документа

После нахождения документа вносим поддокумент в путь отзывов

Поддокумент содержит эти данные

Если вы выполните эту команду в командной оболочке MongoDB во время использования базы данных `Loc8r`, она добавит отзыв в документ. Это можно повторять столько раз, сколько нужно, меняя данные, чтобы добавить несколько отзывов.

Обратите внимание на команду `new Date`, служащую для задания времени создания отзыва. Ее использование гарантирует, что MongoDB будет хранить дату в виде объекта даты в формате ISO, а не строки — именно этого требует наша схема и это позволяет сделать работу с данными дат более гибкой.

Повторяем процесс

Благодаря этим нескольким командам мы получили одно место, на котором можно протестировать приложение, но в идеале нам хотелось бы иметь больше. Так что вперед, добавьте еще несколько мест в БД.

Когда закончите эту работу и данные будут введены, можно начинать использовать эти данные из приложения — в этом случае мы собираемся создать API. Но прежде, чем перейти к этому в главе 6, выполним еще одну вспомогательную операцию. Нам хотелось бы отправлять на Heroku регулярные обновления, и теперь, когда мы добавили соединение с БД и модели данных, необходимо убедиться, что они поддерживаются в Heroku.

5.5. Введение базы данных в промышленную эксплуатацию

Если вы вывели свое приложение в реальный мир, не стоит оставлять базу данных на локальной машине. База данных тоже должна быть доступна извне. В этом разделе мы собираемся поместить базу данных в среду промышленной эксплуатации и модернизировать приложение `Loc8r` так, чтобы оно использовало опубликованную БД для опубликованного сайта и локальную БД — для сайта, предназначенного для разработки. Мы начнем с использования бесплатного пакета сервиса под названием `MongoLab`, который можно использовать в качестве дополнения к Heroku. Если вы предпочитаете другого провайдера или собственный сервер баз данных — не проблема. Первая часть этого раздела посвящена настройке `MongoLab`, но в следующих частях описаны миграция данных и настройка строк соединения в приложении Node, которые являются платформонезависимыми.

5.5.1. Настройка `MongoLab` и получение URI базы данных

Первая задача состоит в получении доступного извне URI базы данных, чтобы можно было поместить туда данные и добавить его в приложение. Мы будем здесь использовать сервис `MongoLab` из-за его неплохого бесплатного пакета, отличной документации и очень отзывчивой группы поддержки.

Существует несколько способов настройки БД на MongoLab. Самый быстрый и удобный из них — использовать дополнение посредством Heroku. Именно этим мы здесь и займемся, однако при этом понадобится зарегистрировать на Heroku реальную банковскую карту. Heroku заставляет вас делать это при использовании его дополнений в своей экосистеме, чтобы защититься от злоупотреблений. Использование бесплатного пакета «песочницы» MongoLab не влечет никаких расходов. Если это вас не устраивает, загляните в следующую врезку, где рассказывается о настройке MongoLab вручную.

РУЧНАЯ НАСТРОЙКА MONGOLAB

Если не хотите, вам не обязательно использовать систему дополнений Heroku. Лучше всего настроить в облаке базу данных MongoDB и получить строку соединения для нее.

Можете воспользоваться документацией MongoLab, которая направит вас на верный путь: <http://docs.mongolab.com/>.

Вкратце шаги выполнения таковы.

1. Зарегистрироваться для получения бесплатной учетной записи.
2. Создать новую БД (для бесплатного пакета необходимо выбрать Single Node, Sandbox).
3. Добавить пользователя.
4. Получить URI базы данных (строку соединения). Строка соединения будет выглядеть так:

```
mongodb://dbuser:dbpassword@ds059957.mongolab.com:59957/loc8r-dev
```

Конечно, все части у вас будут иными и вам придется заменить имя пользователя и пароль теми, которые вы задали на шаге 3.

Как только у вас будет полная строка соединения, ее желательно сохранить в качестве части конфигурации Heroku. Это можно сделать следующей командой (командная строка терминала должна находиться в корневом каталоге вашего приложения):

```
$ heroku config:set MONGOLAB_URI=your_db_uri
```

Замените `your_db_uri` полной строкой соединения, включая протокол `mongodb://`. Автоматически выполнить настройку `MONGOLAB_URI` в вашей конфигурации Heroku можно быстро и легко. Эти выполняемые вручную шаги приведут вас к тому же результату, что и упомянутый быстрый способ, и теперь вы можете вернуться к основному тексту.

Добавление MongoLab в приложение Heroku

Самый быстрый способ добавить MongoLab в качестве дополнения Heroku — через терминал. Убедитесь, что вы находитесь в корневом каталоге своего приложения, и выполните следующую команду:

```
$ heroku addons:add mongolab
```

Невероятно, но факт! База данных MongoDB готова и ждет вас в облаке. Проверить это и открыть веб-интерфейс к новой БД можно с помощью следующей команды:

```
$ heroku addons:open mongolab
```

Для использования БД необходимо знать ее URI.

Получение URI базы данных

Получить полный URI базы данных можно с помощью командной строки. Благодаря этому способу вы получите полную строку соединения, которую можно будет использовать в приложении, а также увидите различные компоненты, которые пригодятся при внесении данных в БД.

Команда для получения URI базы данных:

```
$ heroku config:get MONGOLAB_URI
```

Эта команда выведет полную строку соединения, которая будет выглядеть примерно так:

```
mailto:mongodb://heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669.mongolab.com:33669/heroku_app20110907
```

Держите свой вариант этой строки под рукой, так как скоро вы будете использовать его в приложении. Прежде всего необходимо разложить его на составные части.

Разложение URI на составные части

Хотя этот URI выглядит как случайная мешанина символов, его можно разделить на составные части, чтобы понять смысл. Из раздела 5.1.2 нам известно, что URI базы данных имеет следующую структуру:

```
mongodb://username:password@localhost:27027/database
```

Протокол MongoDB	Учетные данные для входа в БД	Адрес сервера	Порт	Имя БД

Так что можете разложить полученный от MongoLab URI на следующие составные части:

- ❑ имя пользователя — heroku_app20110907;
- ❑ пароль — 4rqhlidfdqq6vgdi06c15jrlpf;
- ❑ адрес сервера — ds033669.mongolab.com;
- ❑ порт — 33669;
- ❑ название базы данных — heroku_app20110907.

Все это взято из приведенного здесь примера URI, так что у вас части будут другими. Запишите их, они вам пригодятся.

5.5.2. Помещение данных в базу данных

Теперь, когда доступная извне БД настроена и все необходимые для соединения с ней подробности известны, можно поместить в нее данные. Пошаговый план выполнения этой задачи таков.

1. Создать временный каталог для хранения дампа данных.
2. Выполнить дамп данных из БД версии Loc8r, предназначенной для разработки.
3. Восстановить данные в промышленную БД.
4. Протестировать промышленную БД.

Все эти шаги можно быстро выполнить через терминал, что мы и сделаем. Это позволит избежать переключений между разными средами.

Создание временного каталога

Простейший первый шаг, который, если вам так удобнее, можете выполнить в интерфейсе операционной системы, состоит в создании временного каталога для размещения дампа ваших данных. Следующая команда выполняет это в операционных системах Mac и Linux:

```
$ mkdir -p ~/tmp/mongodump
```

Теперь у вас есть куда поместить дамп данных.

Выполнение дампа данных из тестовой базы данных

Выполнение дампа выглядит так, как будто вы удаляете все из локальной, предназначенной для разработки БД, но это впечатление неверное. Процесс представляет собой скорее экспорт, а не очистку.

Для этого используется команда `mongodump`, принимающая три входных параметра:

- `-h` — сервер, на котором располагается приложение (и порт);
- `-d` — название базы данных;
- `-o` — целевой каталог для вывода.

Собрав все это воедино и используя порт MongoDB по умолчанию — 27017, вы должны получить следующую команду:

```
$ mongodump -h localhost:27017 -d Loc8r -o ~/tmp/mongodump
```

Выполнив ее, вы получите промежуточный дамп данных.

Восстановление данных в промышленную базу данных

Процесс помещения данных в реальную БД аналогичен выполнению дампа, только вместо `mongodump` используется команда `mongorestore`. На входе она ожидает следующих параметров:

- ❑ `-h` — промышленный сервер и порт;
- ❑ `-d` — название промышленной БД;
- ❑ `-u` — имя пользователя для промышленной БД;
- ❑ `-p` — пароль для промышленной БД.

Собрав все это воедино и используя имеющуюся информацию об URI базы данных, вы должны получить следующую команду:

```
$ mongorestore -h ds033669.mongolab.com:33669 -d heroku_app20110907 -u heroku_app20110907 -p 4rqhlidfdqq6vgdi06c15jr1pf ~/tmp/mongodump/Loc8r
```

Конечно, ваш вариант будет несколько иным, поскольку у вас другие сервер, название промышленной БД, имя пользователя и пароль. Когда выполните команду `mongorestore`, она поместит данные из дампа в вашу БД.

Тестирование промышленной базы данных

Возможности командной оболочки MongoDB не ограничиваются обращением к базам данных на вашей локальной машине. Командную оболочку можно использовать и для соединения с внешними базами данных, если, конечно, у вас имеются нужные учетные данные.

Для соединения командной оболочки MongoDB с внешней БД используется та же команда `mongo`, только добавляется информация о БД, к которой вы хотите подключиться. Необходимо включить в параметры имя сервера, порт и название БД. Также при необходимости можно указать имя пользователя и пароль. Целиком структура этой команды выглядит следующим образом:

```
$ mongo hostname:port/database_name -u username -p password
```

Например, при использовании рассматриваемых нами в этой главе настроек мы получим такую команду:

```
$ mongo ds033669.mongolab.com:33669/heroku_app20110907 -u heroku_app20110907 -p 4rqhlidfdqq6vgdi06c15jr1pf
```

Она соединит вас с БД через командную оболочку MongoDB. После установления соединения можете использовать уже знакомые вам команды для ее опроса:

```
> show collections
> db.locations.find()
```

Теперь у вас имеются две базы данных и две строки соединения, и очень важно всегда использовать правильную.

5.5.3. Заставляем приложение использовать правильную базу данных

Итак, у вас есть исходная, предназначенная для разработки БД на локальной машине и новая промышленная БД в MongoLab (или где-то еще). Хотелось бы продолжать использовать тестовую БД при разработке приложения, но необходимо, чтобы промышленная версия приложения применяла промышленную БД. Однако обе они используют один и тот же исходный код. Данную проблему иллюстрирует рис. 5.12.

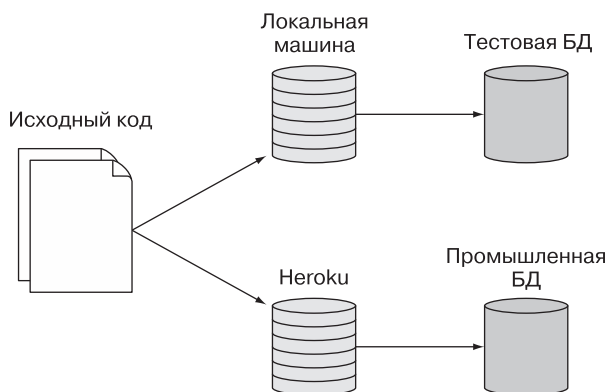


Рис. 5.12. Исходный код выполняется в двух местах, каждое из которых должно подключаться к своей БД

Итак, у нас имеется один набор исходного кода, выполняемый в двух средах, каждая из которых должна использовать свою БД. Справиться с проблемой можно с помощью переменной среды Node `NODE_ENV`.

Переменная среды `NODE_ENV`

Переменные среды влияют на то, как выполняется основной процесс. Та из них, которую мы хотим рассмотреть и использовать, — `NODE_ENV`. Приложение уже использует `NODE_ENV`, вы просто пока что нигде ее не видели. По умолчанию Heroku устанавливает для переменной `NODE_ENV` значение `production`, чтобы приложение на их серверах работало в промышленном режиме.

ПРОВЕРЯЕМ, ИСПОЛЬЗУЕТ ЛИ HEROKU ПРОМЫШЛЕННЫЙ РЕЖИМ _____

В определенных случаях в зависимости от настроек приложения Heroku может не работать в промышленном режиме. Убедиться, что переменные среды настроены правильно, можно с помощью следующей команды терминала:

```
$ heroku config:set NODE_ENV=production
```

Прочитать `NODE_ENV` из любого места приложения можно с помощью следующего оператора:

```
process.env.NODE_ENV
```

Если эта переменная не задана в вашей среде, то он вернет `undefined`. Задать различные переменные среды можно при запуске приложения Node, вставив в начале команды запуска присваивание, например:

```
$ NODE_ENV=production nodemon
```

Эта команда запустит приложение в промышленном режиме, а значение `process.env.NODE_ENV` будет установлено равным `production`.

СОВЕТ

Не устанавливайте переменную `NODE_ENV` из приложения, только читайте ее.

Задание URI базы данных в зависимости от среды

Соединение БД для нашего приложения находится в файле `db.js` в каталоге `app_server/models`. Относящаяся к соединению часть этого файла в настоящий момент выглядит следующим образом:

```
var dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI);
```

Для изменения значения `dbURI` в зависимости от текущей среды достаточно всего лишь воспользоваться оператором `if` для проверки значения `NODE_ENV`. В следующем фрагменте кода показано, как выполнить это для передачи соединения с промышленной MongoDB. Обратите внимание на то, что вам нужно использовать свою собственную строку соединения, а не приведенную в примере:

```
var dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI = 'mongodb://
heroku_app20110907:4rqhlidfdqq6vgdi06c15jr1pf@ds033669.mongolab.com:3366
9/heroku_app20110907';
}
mongoose.connect(dbURI);
```

Если исходный код будет находиться в общедоступном репозитории, вы, вероятно, не захотите показывать всем свои учетные данные для БД. Обойти это можно с помощью переменной среды. Благодаря MongoLab на Heroku у вас уже есть одна настроенная переменная — именно так мы ранее получили доступ к строке соединения (если вы настроили учетную запись MongoLab вручную, это настроенная вами переменная конфигурации Heroku). Если же вы используете другого провайдера, не добавляющего ничего к конфигурации Heroku, то можете указать

свой URI с помощью команды `heroku config:set`, которую мы использовали, чтобы убедиться, что Heroku выполняется в промышленном режиме.

Следующий фрагмент кода демонстрирует использование строки соединения, задаваемой через переменные среды:

```
var dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI = process.env.MONGOLAB_URI;
}
mongoose.connect(dbURI);
```

Это значит, что теперь вы можете использовать код совместно, но доступ к вашим учетным данным для БД будет только у вас.

Тестирование перед пуском

Путем задания переменной среды при запуске приложения из терминала можно протестировать эти модернизации кода до отправки его на Heroku. События соединения Mongoose, которые мы настроили ранее, выводят журнал в консоль при выполнении соединения с БД для контроля используемого URI.

Обычный запуск приложения из терминала выглядит вот так:

```
$ nodemon
Express server listening on port 3000
Mongoose connected to mongodb://localhost/Loc8r
```

Для сравнения: запуск приложения в промышленном режиме выглядит следующим образом:

```
$ NODE_ENV=production nodemon
Express server listening on port 3000
Mongoose connected to mongodb://
heroku_app20110907:4rqhldfdqq6vgdi06c15jrlpf@ds033669.mongolab.com:33669/
heroku_app20110907
```

Вероятно, при выполнении этих команд вы обратили внимание на то, что подтверждение соединения Mongoose появляется в промышленной среде через значительный промежуток времени. Это происходит из-за задержки использования отдельного сервера для БД. Именно поэтому стоит открывать соединение с БД при запуске приложения и оставлять его открытым.

Отмечу, что предыдущая проверка промышленной среды может завершиться неудачей в некоторых версиях Windows и в редких случаях в Linux. Это происходит, если ваша система не в состоянии извлечь переменные среды Heroku. При этом вы все равно можете выполнить тестирование промышленной БД с помощью вставки `MONGOLAB_URI` в начало команды запуска приложения (`start`), как в следующем фрагменте кода (обратите внимание: его необходимо ввести в виде одной строки):

```
$ NODE_ENV=production MONGOLAB_URI=mongodb://
<username>:<password>@<hostname>:<port>/<database> nodemon start
```

Теперь вне зависимости от используемой операционной системы вы сможете запустить приложение локально, подключив его к промышленной БД.

Тестирование на Heroku

Если локальные проверки прошли успешно и вы можете подключиться к удаленной БД посредством временного запуска приложения в промышленном режиме, значит, вы готовы отправить его на Heroku. Воспользуйтесь теми же командами, что и обычно, для отправки туда последней версии кода:

```
$ git add .
$ git commit -m "Сообщение, описывающее детали коммита"
$ git push heroku master
```

Heroku позволяет вам с легкостью взглянуть на последние 100 строк журналов с помощью выполнения команды терминала. Можно заглянуть в эти журналы, чтобы увидеть вывод журнальных сообщений вашей консоли, одним из которых будет сообщение `Mongoose connected to...` Для отображения журналов выполните в терминале следующую команду:

```
$ heroku logs
```

Она выведет последние 100 строк в окно терминала, самые последние строки будут внизу. Прокрутите окно до сообщения `Mongoose connected to...`, которое будет выглядеть примерно так:

```
2014-03-08T08:19:42.269603+00:00 app[web.1]: Mongoose connected to mongodb://
heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669.mongo-
lab.com:33669/heroku_app20110907
```

Если вы это видите, значит, промышленное приложение на Heroku подключилось к вашей промышленной БД.

Итак, данные описаны и промоделированы, а наше приложение Loc8r подключилось к БД. Но мы пока что совсем не взаимодействовали с БД — это нам предстоит позже!

ПОЛУЧЕНИЕ ИСХОДНОГО КОДА

Исходный код для приложения по состоянию на текущий момент доступен на GitHub, в ветви `chapter-05` репозитория `getting-MEAN`. Для его клонирования и установки зависимостей модулей npm выполните в терминале в новом каталоге следующие команды:

```
$ git clone -b chapter-05 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

5.6. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Использование Mongoose для подключения приложения Express с базой данных MongoDB.
- ❑ Рекомендуемые решения для управления соединениями Mongoose.
- ❑ Моделирование данных с помощью схем Mongoose.
- ❑ Компиляцию схем в модели.
- ❑ Использование командной оболочки MongoDB для непосредственной работы с БД.
- ❑ Размещение БД по промышленному URI.
- ❑ Соединение с разными базами данных из различных сред.

В главе 6 мы собираемся использовать Express для создания API REST, чтобы затем можно было обращаться к БД через веб-сервисы.

Глава 6

Создание API REST: делаем базу данных MongoDB доступной приложению

В этой главе:

- ❑ правила создания API REST;
- ❑ паттерны API;
- ❑ типичные функции CRUD (создание, чтение, обновление, удаление);
- ❑ использование Express и Mongoose для взаимодействия с MongoDB;
- ❑ тестирование конечных точек API.

У нас уже есть база данных MongoDB, но взаимодействовать с ней мы можем только через командную оболочку MongoDB. В этой главе создадим API REST, что даст возможность работать с БД через HTTP-вызовы и выполнять типичные функции CRUD: создание, чтение, обновление и удаление.

В основном мы будем иметь дело с Node и Express, применяя Mongoose в качестве вспомогательного средства при взаимодействии. Место этой главы в общей архитектуре приложения демонстрирует рис. 6.1.

Мы начнем с изучения правил API REST, обсудим, как важно правильно формировать структуру URL, какие методы запросов (GET, POST, PUT и DELETE) следует использовать для каких действий и то, что API должен возвращать в ответе данные и соответствующий код состояния HTTP. Когда мы разберемся с этим, перейдем к построению API для Local и рассмотрим все типичные операции CRUD. По ходу дела обсудим многие относящиеся к Mongoose моменты, займемся программированием Node, а также дополнительной маршрутизацией Express.

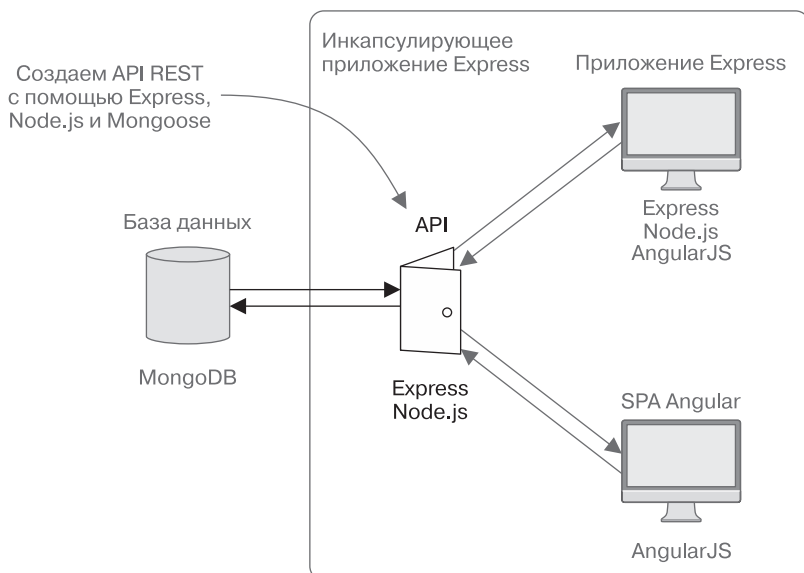


Рис. 6.1. Эта глава будет посвящена созданию API, взаимодействующего с БД и предоставляющего приложениям интерфейс для связи.

ПРИМЕЧАНИЕ

Если вы еще не создали приложение из главы 5, то можете получить исходный код с GitHub, из ветви `chapter-05`, по адресу <http://www.github.com/simonholmes/getting-MEAN>. Для его клонирования и установки зависимостей npm выполните в терминале в новом каталоге следующие команды:

```
$ git clone -b chapter-05 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

6.1. Правила API REST

Начну с напоминания о том, что такое API REST. Как вы помните из главы 2:

- ❑ REST расшифровывается как «передача состояния представления» (REpresentational State Transfer) и представляет собой скорее архитектурный стиль, чем жесткий протокол. В REST отсутствует сохранение состояния, он ничего не знает о состоянии или истории действий текущего пользователя;
- ❑ API — аббревиатура для Application Program Interface — интерфейс программирования приложений, который дает приложениям возможность общаться друг с другом.

Итак, REST — интерфейс к вашему приложению, в котором отсутствует сохранение состояния. В случае использования стека MEAN API REST используется для создания интерфейса (без сохранения состояния), обеспечивающего другим приложениям возможность работать с соответствующими данными.

У API REST имеется свой набор стандартов. Хотя это и не обязательно, но лучше их придерживаться, чтобы все создаваемые вами API были единообразными. Это также значит, что вы привыкнете делать все правильно на случай, если захотите сделать свой API общедоступным.

Проще говоря, API REST принимает входящий HTTP-запрос, как-то его обрабатывает и всегда возвращает HTTP-ответ (рис. 6.2).

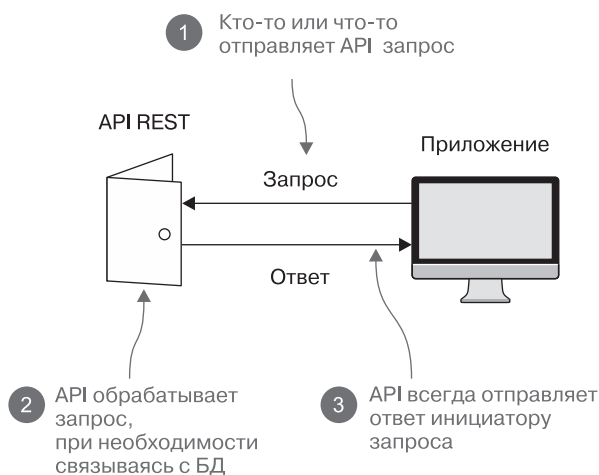


Рис. 6.2. API REST принимает входящие HTTP-запросы, выполняет какую-то их обработку и возвращает HTTP-ответы

Стандарты, которым мы собираемся следовать в Лоc8г, связаны с запросами и ответами.

6.1.1. URL запросов

Стандарт, касающийся URL запросов для API REST, очень прост. Следование этому стандарту обеспечит удобство взаимосвязи с вашим API, его использования и поддержки.

Чтобы разобраться с этим, нужно сначала подумать о коллекциях в вашей БД, ведь обычно у вас есть набор различных URL API для каждой коллекции. Также у вас может быть набор URL для каждого набора поддокументов. У каждого URL в наборе будет тот же базовый путь, а у некоторых могут быть еще дополнительные параметры.

Набором URL необходимо охватить несколько действий, обычно связанных со стандартными операциями CRUD. Общие действия, которые, вероятно, вам понадобятся:

- ❑ создание нового элемента;
- ❑ чтение списка из нескольких элементов;
- ❑ чтение конкретного элемента;
- ❑ обновление конкретного элемента;
- ❑ удаление конкретного элемента.

Если говорить о Loc8r, то в БД имеется коллекция Locations (Местоположения), с которой нам необходимо взаимодействовать. В табл. 6.1 показано, как могли бы выглядеть URL и параметры для этой коллекции.

Таблица 6.1. Путь URL и параметры для API к коллекции Locations (Местоположения): у всех один и тот же базовый путь, а у некоторых и одинаковый параметр locationid

Действие	Путь URL	Параметр	Пример
Создание нового местоположения	/locations		http://loc8r.com/api/locations
Чтение списка местоположений	/locations		http://loc8r.com/api/locations
Чтение конкретного местоположения	/locations	locationid	http://loc8r.com/api/locations/123
Обновление конкретного местоположения	/locations	locationid	http://loc8r.com/api/locations/123
Удаление конкретного местоположения	/locations	locationid	http://loc8r.com/api/locations/123

Как вы можете видеть из табл. 6.1, у всех действий одинаковые пути URL, а три из них ожидают на входе один и тот же параметр, задающий место. Это вызывает очевидный вопрос: как же можно использовать один и тот же URL для запуска различных действий? Ключ к ответу — в методах запроса.

6.1.2. Методы запроса

HTTP-запросы могут использовать различные методы для указания серверу, какой тип действия применять. Наиболее распространенный тип запроса — запрос GET — метод, который используется, когда вы вводите URL в адресную строку своего браузера. Еще один распространенный метод, POST, часто используется при отправке данных форм.

Методы, которые мы будем применять в API, их типичные сценарии использования, а также то, что они могут возвращать при ответе, демонстрирует табл. 6.2.

Таблица 6.2. Четыре метода запроса, используемые в API REST

Метод запроса	Использование	Ответ
POST	Создание (Create) новых данных в БД	Новый объект данных, такой же, как и в БД
GET	Чтение (Read) данных из БД	Объект данных, соответствующий запросу
PUT	Обновление (Update) документа в БД	Обновленный объект данных, такой же, как находится в БД
DELETE	Удаление (Delete) объекта из БД	Null

Четыре метода запроса, которые мы будем использовать, — POST, GET, PUT и DELETE. Если вы посмотрите на первые слова в столбце «Использование» таблицы, то увидите, что для каждой из четырех операций CRUD имеется свой метод.

СОВЕТ

Каждый из четырех методов CRUD использует различные методы запроса.

Методы важны, поскольку в хорошо спроектированном API REST одни и те же URL часто будут использоваться для различных действий. В подобных случаях именно метод говорит серверу, какой тип операции выполнять. Несколько позднее в данной главе мы рассмотрим создание и организацию в Express маршрутов для этой цели.

Итак, если поставить путям и параметрам в соответствие подходящий метод запроса, мы получим план API (табл. 6.3).

Таблица 6.3. Метод запроса используется, чтобы связать URL с необходимым действием, предоставляя API возможность применять один и тот же URL для различных действий

Действие	Метод	Путь URL	Параметры	Пример
Создание нового местоположения	POST	/locations		http://loc8r.com/api/locations
Чтение списка местоположений	GET	/locations		http://loc8r.com/api/locations
Чтение конкретного местоположения	GET	/locations	locationid	http://loc8r.com/api/locations/123

Действие	Метод	Путь URL	Параметры	Пример
Обновление конкретного местоположения	PUT	/locations	locationid	http://loc8r.com/api/locations/123
Удаление конкретного местоположения	DELETE	/locations	locationid	http://loc8r.com/api/locations/123

Таблица 6.3 демонстрирует пути и методы, используемые запросами для взаимодействия с данными о местах. Так как тут пять действий, но только два различных паттерна URL, то можно использовать методы запроса для получения желаемых результатов.

В Loc8r пока что имеется только одна коллекция, следовательно, это наша отправная точка. Но у документов в коллекции Locations (Местоположения) имеются отзывы в виде поддокументов, так что давайте не откладывая установим для них соответствия.

URL API для поддокументов

С поддокументами можно поступить аналогичным образом, но для них необходим дополнительный параметр. В каждом запросе должен быть задан идентификатор местоположения, а в некоторых — еще и идентификатор отзыва. Список действий и соответствующих им методов, путей URL и параметров демонстрирует табл. 6.4.

Таблица 6.4. Спецификации URL API для взаимодействия с поддокументами, каждый базовый путь должен содержать идентификатор родительского документа

Действие	Метод	Путь URL	Параметр	Пример
Создание нового отзыва	POST	/locations/ locationid/ reviews	locationid	http://loc8r.com/api/ locations/123/reviews
Чтение конкретного отзыва	GET	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/ locations/123/reviews/abc
Обновление конкретного отзыва	PUT	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/ locations/123/reviews/abc
Удаление конкретного отзыва	DELETE	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/ locations/123/reviews/abc

Как вы могли заметить, для поддокументов у нас нет действия «чтение списка отзывов». Причина в том, что список отзывов мы извлекаем как часть основного документа. Предшествующие таблицы могли дать вам представление о том, как создавать основные спецификации запросов к API. URL, параметры и действия будут разными в различных приложениях, но подход останется одним и тем же.

Мы рассмотрели запросы. Вторая половина потока выполнения, которую рассмотрим, прежде чем заняться кодом, — ответы.

6.1.3. Ответы и коды состояния

Хороший API как хороший друг. Если вы «даете пять» другу, он никогда не оставит вашу руку в подвешенном состоянии. То же самое и с хорошим API. Если вы выполняете запрос, хороший API всегда ответит и не оставит запрос висеть. Каждый отдельный запрос к API должен возвращать ответ. Различие между хорошим и плохим API показано на рис. 6.3.

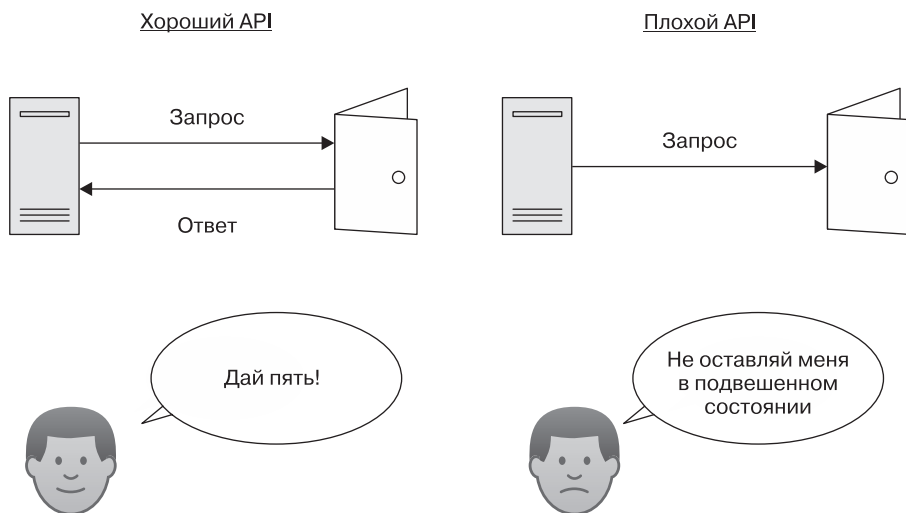


Рис. 6.3. Хороший API всегда возвращает ответ и не оставляет вас в подвешенном состоянии

Для успешного функционирования API стандартизация ответов так же важна, как и стандартизация формата запросов. Ответ содержит два ключевых компонента:

- ❑ возвращаемые данные;
- ❑ код состояния HTTP.

Правильное сочетание возвращаемых данных с соответствующим кодом состояния должно обеспечить инициатору запроса всю необходимую для продолжения работы информацию.

Возвращение данных из API

Ваш API должен возвращать данные в непротиворечивом формате. Обычные для API форматы — XML и JSON. Для нашего API будем использовать JSON, поскольку он естественным образом подходит для стека MEAN и занимает меньше места, чем XML, а значит, поможет ускорить время ответа API.

Наш API будет возвращать для каждого запроса одно из трех:

- ❑ объект JSON, содержащий данные, отвечающие на исходный запрос;
- ❑ объект JSON, содержащий информацию об ошибке;
- ❑ пустой (null) ответ.

В данной главе мы обсудим, как выполнить все это при создании API Loc8r. Также, помимо возвращения данных, любой API REST должен возвращать правильный код состояния.

Коды состояния HTTP

Хороший API REST должен возвращать правильный код состояния. Большинству людей знаком код состояния 404 — то, что возвращает веб-сервер, когда пользователь запрашивает отсутствующую страницу. Вероятно, это самая распространенная ошибка в Интернете, но существуют десятки других кодов, соответствующих ошибкам пользователя, ошибкам сервера, перенаправлениям и успешно завершившимся запросам. В табл. 6.5 показаны десять наиболее распространенных кодов состояния HTTP и то, где они могут пригодиться при создании API.

Таблица 6.5. Наиболее распространенные коды состояния HTTP и то, как их можно использовать при отправке ответов на запросы API

Код состояния	Название	Сценарий использования
200	ОК	Успешный запрос GET или PUT
201	Создано	Успешный запрос POST
204	Нет содержимого	Успешный запрос DELETE
400	«Плохой» запрос	Неудачный (по причине недопустимого содержимого ¹) запрос GET, POST или PUT

Продолжение ↗

¹ Проще говоря, из-за синтаксической ошибки. — *Примеч. пер.*

Таблица 6.5 (продолжение)

Код состояния	Название	Сценарий использования
401	Не авторизован	Запрос URL с ограниченным доступом с неправильными учетными данными
403	Запрещено	Попытка выполнения недопустимого запроса
404	Не найдено	Запрос завершился неудачей из-за неправильного параметра в URL ¹
405	Метод не поддерживается	Метод запроса не разрешен к использованию для данного URL
409	Конфликт	Неудачный запрос POST в случае, когда уже существует другой объект с теми же данными
500	Внутренняя ошибка сервера	Проблема с сервером или сервером БД

По мере продвижения по этой главе и создания API Loc8t мы воспользуемся несколькими из этих кодов состояния, возвращая при этом соответствующие данные.

6.2. Настройка API в Express

Мы уже получили представление о действиях, которые должен выполнять API, и необходимых для этого путях URL. Как мы знаем из главы 4, чтобы Express выполнял что-либо на основе входящего запроса URL, необходимо настроить контроллеры и маршруты. Контроллеры будут выполнять действие, а маршруты — устанавливать соответствие входящих запросов требуемым контроллерам.

Файлы для маршрутов и контроллеров в нашем приложении уже настроены, так что мы можем ими воспользоваться. Лучшим вариантом, однако, будет хранить код API отдельно, чтобы избежать путаницы и проблем в приложении. По существу, это одна из основных причин для создания API. Кроме того, отделение кода API облегчает его дальнейшее вынесение в другое приложение, если это понадобится. Безусловно, желательно иметь тут возможность удобного расцепления.

¹ Иначе говоря, сервер не нашел требуемого документа. — *Примеч. пер.*

Так что первое, что нам нужно будет сделать, — создать отдельную область в приложении для файлов API. На верхнем уровне каталогов приложения *создайте новый каталог* под названием `app_api`. Если вы следили за ходом событий и создавали приложение по мере чтения книги, то он должен располагаться рядом с каталогом `app_server`.

В этом каталоге будет находиться все, что относится непосредственно к API: маршруты, контроллеры и модели. Когда мы все это настроим, то рассмотрим некоторые способы проверки этих заглушек API.

6.2.1. Создание маршрутов

Аналогично тому, как мы поступали с маршрутами для основного приложения Express, у нас будет файл `index.js` в каталоге `app_api/routes`, в котором будут находиться все используемые в API маршруты. Начнем с того, что сделаем ссылку на этот файл в основном файле приложения `app.js`.

Включение маршрутов в приложение

Сначала нужно сообщить приложению о добавлении дополнительных маршрутов, которые нужно будет искать, и о том, когда их следует использовать. В `app.js` уже имеется строка для запроса (`require`) серверных маршрутов приложения, ее можно просто продублировать и задать путь к маршрутам API следующим образом:

```
var routes = require('./app_server/routes/index');  
var routesApi = require('./app_api/routes/index');
```

Далее необходимо сообщить приложению, когда использовать маршруты. В `app.js` уже есть строка, которая указывает приложению проверять серверные маршруты приложения для всех входящих запросов:

```
app.use('/', routes);
```

Обратите внимание на `'/'` в качестве первого параметра. Благодаря этому мы можем задать подмножество URL, к которым будут относиться маршруты. Например, опишем все наши маршруты для API как начинающиеся с `/api/`. Добавляя показанную в следующем фрагменте кода строку, можем сообщить приложению, что нужно использовать маршруты API только тогда, когда маршрут начинается с `/api`:

```
app.use('/', routes);  
app.use('/api', routesApi);
```

Что ж, давайте настроим эти URL.

Задание методов запроса в маршрутах

До сих пор мы использовали в маршрутах только метод GET, как в следующем фрагменте кода из основных маршрутов приложения:

```
router.get('/location', ctrlLocations.locationInfo);
```

Для использования остальных методов (POST, PUT и DELETE) достаточно заменить `get` соответствующим ключевым словом: `post`, `put` или `delete`. Следующий фрагмент кода демонстрирует пример использования метода POST для создания нового местоположения:

```
router.post('/locations', ctrlLocations.locationsCreate);
```

Обратите внимание на то, что мы не указывали `/api` в начале пути. В `app.js` уже указано, что эти маршруты должны использоваться только тогда, когда путь начинается с `/api`, поэтому предполагается, что в начале всех заданных в этом файле путей будет поставлено `/api`.

Задание необходимых параметров URL

URL API часто содержат параметры для определения конкретных документов или поддокументов — мест и отзывов в случае `Loc8r`. Задать эти параметры в маршрутах очень просто: достаточно при описании маршрута поставить перед названием параметра двоеточие.

Допустим, вы хотите получить доступ к отзыву с идентификатором `abc`, относящемуся к местоположению с идентификатором `123`. Путь будет следующим:

```
/api/locations/123/reviews/abc
```

Заменив идентификаторы именами параметров, которым предшествует двоеточие, вы получите следующий путь:

```
/api/locations/:locationid/reviews/:reviewid
```

В данном случае Express найдет только пути, соответствующие этому паттерну. Так, идентификатор места должен быть задан и должен находиться в URL между `locations/` и `/reviews`, причем идентификатор отзыва также должен быть задан в конце URL. При задании контроллеру такого пути параметры будут доступны для использования в коде по заданным в пути именам, в данном случае `locationid` и `reviewid`.

Очень скоро мы рассмотрим, как к ним можно обратиться, но сначала нужно настроить маршруты для API нашего `Loc8r`.

Описание маршрутов API Loc8r

Теперь мы знаем, как сделать так, чтобы маршруты принимали параметры, и знаем, какие действия, методы и пути нам понадобятся в API. Так что можем использовать все это для создания описаний маршрутов для API Loc8r.

Если вы еще не сделали этого, то создайте файл `index.js` в каталоге `app_api/routes`. Чтобы отдельные файлы не получались огромными, разнесем контроллеры для мест и отзывов по различным файлам. Листинг 6.1 демонстрирует, как должны выглядеть описанные маршруты.

Листинг 6.1. Описанные в файле `app_api/routes/locations.js` маршруты

```
var express = require('express');
var router = express.Router();
var ctrlLocations = require('../controllers/locations');
var ctrlReviews = require('../controllers/reviews');

// местоположения
router.get('/locations', ctrlLocations.locationsListByDistance);
router.post('/locations', ctrlLocations.locationsCreate);
router.get('/locations/:locationid', ctrlLocations.locationsReadOne);
router.put('/locations/:locationid', ctrlLocations.locationsUpdateOne);
router.delete('/locations/:locationid',
              ctrlLocations.locationsDeleteOne);

// отзывы
router.post('/locations/:locationid/reviews',
            ctrlReviews.reviewsCreate);
router.get('/locations/:locationid/reviews/:reviewid',
            ctrlReviews.reviewsReadOne);
router.put('/locations/:locationid/reviews/:reviewid',
            ctrlReviews.reviewsUpdateOne);
router.delete('/locations/:locationid/reviews/:reviewid',
              ctrlReviews.reviewsDeleteOne);

module.exports = router;
```

Включаем файл контроллера (мы создадим его позже)

Описываем маршруты для местоположений

Описываем маршруты для отзывов

Экспортируем маршруты

В этом файле маршрутизации нужно выполнить `require` соответствующих файлов контроллеров. Мы еще не создали файлы контроллеров и сделаем это буквально через минуту. Подобный подход удобен, поскольку посредством описания здесь всех маршрутов и объявления соответствующих функций контроллеров мы сформировали общее представление о том, какие контроллеры нам необходимы.

Теперь в приложении есть два набора маршрутов: основные маршруты приложения Express и новые маршруты API. Приложение, правда, сейчас не запустится, поскольку никаких контроллеров, на которые ссылаются маршруты API, не существует.

6.2.2. Создание заглушек для контроллеров

Чтобы приложение могло запуститься, создадим функции-заглушки для контроллеров. В действительности эти функции ничего делать не будут, но они предотвратят сбой приложения, пока мы будем создавать функциональность API.

Первым этапом, конечно, станет создание файлов контроллеров. Нам известно, где они должны находиться и как называться, поскольку мы уже объявили их в каталоге `app_api/routes`. Нам нужны два файла — `locations.js` и `reviews.js` — в каталоге `app_api/controllers`.

Можно создать для каждой из функций контроллера заглушку в виде пустой функции для экспорта, как показано в следующем фрагменте кода. Не забудьте, что каждый контроллер нужно поместить в соответствующий файл в зависимости от того, предназначен он для местоположений или отзывов:

```
module.exports.locationsCreate = function (req, res) { };
```

Однако для тестирования маршрутизации и работы функций нам нужно будет возвращать какой-то ответ.

6.2.3. Возврат JSON из запроса Express

При создании приложения Express мы визуализировали шаблон представления для отправки HTML браузеру, но при создании API вместо этого нужно отправлять код состояния и какие-либо данные JSON. Express сильно упрощает эту задачу благодаря следующим командам:

```
res.status(status); ← Отправляем код состояния  
res.json(content); ← Отправляем данные ответа,  
                    например {"status": "success"}
```

Эти две функции можно использовать в заглушках для проверки работоспособности, как показано в следующем фрагменте кода:

```
module.exports.locationsCreate = function (req, res) {  
  res.status(200);  
  res.json({"status" : "success"});  
};
```

Возврат JSON и кода состояния — очень распространенная задача для API, так что стоит вынести эти два оператора в отдельную функцию. К тому же это облегчит тестирование кода. Так что создадим функцию `sendJsonResponse` в обоих файлах контроллеров и вызовем ее из каждой из заглушек контроллеров, вот так:

```
var sendJsonResponse = function(res, status, content) {
  res.status(status);
  res.json(content);
};

module.exports.locationsCreate = function (req, res) {
  sendJsonResponse(res, 200, {"status" : "success"});
};
```

Новая вспомогательная функция, принимающая объект ответа, код состояния и объект данных

Вызываем новую функцию из каждой функции контроллера

Теперь можем отправить JSON-ответ и соответствующий код состояния с помощью одной строки кода. Мы не раз воспользуемся этим в нашем API!

6.2.4. Включение модели

Жизненно необходимо, чтобы API мог взаимодействовать с БД, без этого он будет практически бесполезным! Чтобы достичь этого с помощью Mongoose, сначала понадобится запросить (`require`) Mongoose в файлах контроллеров, а затем внедрить модель `Location` (Местоположение). Прямо вверху файлов контроллеров, над всеми функциями-заглушками, добавьте следующие две строки:

```
var mongoose = require('mongoose');
var Loc = mongoose.model('Location');
```

Первая строка обеспечивает контроллеру доступ к соединению с БД, а вторая внедряет модель `Location` (Местоположение), чтобы можно было взаимодействовать с коллекцией `Locations` (Местоположения).

Если посмотреть на структуру файлов нашего приложения, то можно увидеть каталог `app_api/models`, содержащий соединения с БД, и настройки Mongoose в каталоге `app_server`. Но с базой данных работает API, а не основное приложение Express. Если эти два приложения будут разделены, то модель должна оказаться частью API, так что именно там она должна находиться.

Просто переместите каталог `app_api/models` из каталога `app_server` в `app_api`, чтобы структура каталогов выглядела так, как показано на рис. 6.4.

Конечно, мы должны сообщить приложению о перемещении каталога `app_api/models`, так что отредактируем запрашивающую модель строку в `app.js`, чтобы она указывала на правильное место:

```
require('./app_api/models/db');
```

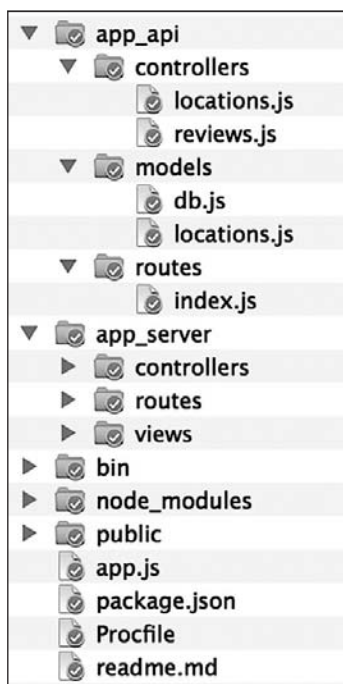


Рис. 6.4. Структура каталогов приложения на текущий момент: в `app_api` находятся модели, контроллеры и маршруты (каталоги `models`, `controllers` и `routes`), а в `app_server` — представления, контроллеры и маршруты (каталоги `views`, `controllers` и `routes`)

После этого приложение опять должно запускаться и подключаться к БД. Следующий вопрос: как проверить работу API?

6.2.5. Тестирование API

Можно быстро проверить маршруты в браузере, перейдя по соответствующему URL, например `http://localhost:3000/api/locations/1234`. При этом вы должны увидеть выдаваемый в браузере ответ об успехе операции (рис. 6.5).

Для тестирования запросов GET это подходит, но мало что даст для методов POST, PUT и DELETE. Протестировать подобные вызовы API вам помогут несколько инструментов, из которых мой любимый — расширение для браузера Chrome под названием «клиент REST Postman».

Postman предоставляет возможность проверить необходимые URL API для различных методов запросов, позволяя задавать дополнительные параметры строки запроса или данные форм. После нажатия кнопки **Send** (Отправить) он выполняет запрос к заданному вами URL и отображает данные ответа и код состояния.

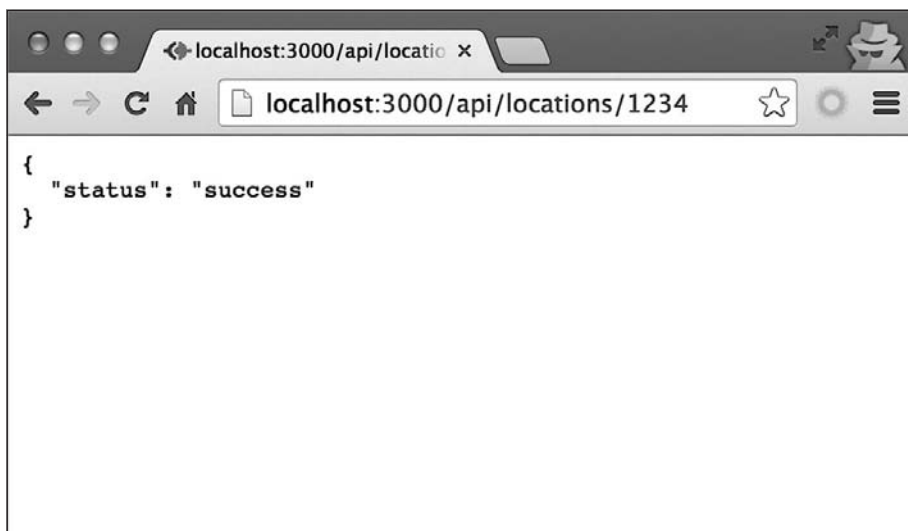


Рис. 6.5. Проверка запроса GET нашего API в браузере

Скриншот Postman, выполняющего запрос PUT по тому же URL, что и раньше, демонстрирует рис. 6.6.

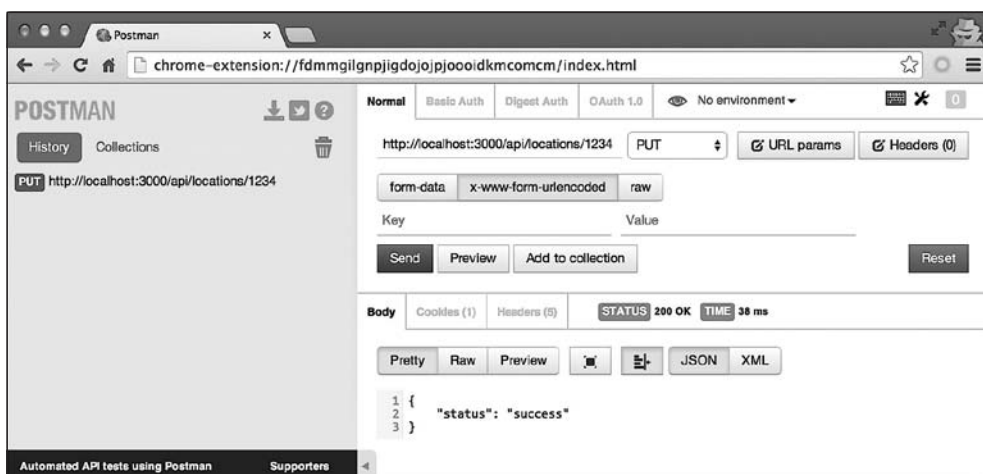


Рис. 6.6. Использование клиента REST Postman в Chrome для тестирования запроса PUT к API

Неплохой мыслью будет установить и запустить Postman или какой-то другой клиент REST прямо сейчас. Вам не раз придется применять его при чтении данной главы, когда мы будем создавать API REST. А пока что начнем с использования запросов GET для чтения данных из MongoDB.

6.3. Методы GET: чтение данных из MongoDB

Методы GET всецело связаны с выполнением запросов к БД и возвращением каких-либо данных. Как перечислено в табл. 6.6, в наших маршрутах для Loc8r имеется три запроса GET, выполняющих различные действия.

Таблица 6.6. Три запроса GET API приложения Loc8r

Действие	Метод	Путь URL	Параметр	Пример
Чтение списка местоположений	GET	/locations		http://loc8r.com/api/locations
Чтение конкретного местоположения	GET	/locations/	locationid	http://loc8r.com/api/locations/123
Чтение конкретного отзыва	GET	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/locations/123/reviews/abc

Рассмотрим сначала, как выполнить поиск отдельного местоположения, поскольку это позволяет познакомиться с методологией работы Mongoose. Далее мы найдем отдельный документ по идентификатору, а затем приступим к поиску нескольких документов.

6.3.1. Поиск отдельного документа в MongoDB с помощью Mongoose

Mongoose взаимодействует с БД посредством моделей, именно поэтому мы импортировали модель Locations (Местоположения) как Loc вверху файлов контроллеров. У модели Mongoose имеется несколько методов для управления взаимодействиями, как описано в следующей врезке.

МЕТОДЫ ЗАПРОСОВ MONGOOSE

Моделям Mongoose доступны несколько методов для запросов к БД. Вот ключевые из них:

- `find` — общий поиск на основе указанного объекта запроса;
- `findById` — поиск по конкретному ID;
- `findOne` — получение первого документа, соответствующего указанному запросу;

- `geoNear` — поиск местоположений, близких географически к заданным широте и долготе;
- `geoSearch` — добавление функциональности запроса к операции `geoNear`.

Мы будем использовать часть из них в данной книге.

Для поиска в MongoDB отдельного документа с известным идентификатором у `Mongoose` имеется метод `findById`.

Метод `findById` для модели

Метод `findById` довольно прост, имеет один входной параметр — идентификатор, по которому выполняется поиск. Поскольку это метод модели, он используется следующим образом:

```
Loc.findById(locationid)
```

Этот оператор не запускает выполнение запроса к БД, он дает модели информацию о запросе. Для запуска выполнения запроса к БД у моделей `Mongoose` имеется метод `exec`.

Запуск выполнения запроса с помощью метода `exec`

Метод `exec` выполняет запрос и передает функцию обратного вызова, которая будет выполнена после завершения операции. Функция обратного вызова принимает два входных параметра: объект ошибки и экземпляр найденного документа. Поскольку это функция обратного вызова, то имена параметров остаются на ваше усмотрение.

Эти методы можно организовать цепочкой следующим образом:

```
Loc
  .findById(locationid)
  .exec(function(err, location) {
    console.log("findById complete");
  });
```

Использование метода `findById` для модели `Location` (Местоположение) с помощью `Loc`

Выполнение запроса

По завершении — вывод в журнал сообщения

Такой подход гарантирует асинхронность взаимодействия с БД, а следовательно, не блокирует основной процесс `Node`.

СОВЕТ

Если вы чувствуете себя неуверенно, работая с обратными вызовами, областями действия и переменными, загляните в выложенное в Интернете приложение, раздел «Обратные вызовы JavaScript».

Использование метода findById в контроллере

Для поиска отдельного места по идентификатору мы используем контроллер `locationsReadOne`, расположенный в файле `locations.js` в каталоге `app_api/controllers`.

Как нам уже известно, операция состоит из применения методов `findById` и `exec` к модели `Location` (Местоположение). Чтобы это заработало в контексте контроллера, необходимо сделать следующее:

- ❑ получить из URL параметр `locationid` и передать его методу `findById`;
- ❑ обеспечить функцию вывода для метода `exec`.

Благодаря Express можно очень легко извлечь описанные в маршрутах параметры URL. Параметры хранятся в объекте `params`, присоединенном к объекту запроса. При вот таком описании маршрута:

```
app.get('/api/locations/:locationid', ctrlLocations.locationsReadOne);
```

обратиться к параметру `locationid` из контроллера можно следующим образом:

```
req.params.locationid
```

В качестве функции вывода мы можем использовать созданную ранее функцию `sendJsonResponse`. Собрав все вместе, получаем следующее:

```
module.exports.locationsReadOne = function(req, res) {
  Loc
    .findById(req.params.locationid)
    .exec(function(err, location) {
      sendJsonResponse(res, 200, location);
    });
};
```

Получаем из параметров URL `locationid` и передаем его методу `findById`

Описываем функцию обратного вызова для приема возможных параметров

Отправляем найденный документ в виде ответа JSON

Теперь у нас есть простейший контроллер API. Можно попробовать его в действии, получив идентификатор одного из местоположений в MongoDB и перейдя по URL в браузере или вызвав его в Postman. Для получения одного из значений ID можно выполнить команду `db.locations.find()` в командной оболочке MongoDB — она выведет список всех имеющихся мест, каждое из которых будет включать значение `_id`. Когда вы соберете все части URL вместе, вывод должен оказаться полным списком местоположений в том виде, в каком он хранится в MongoDB: вы должны увидеть что-то напоминающее рис. 6.7.

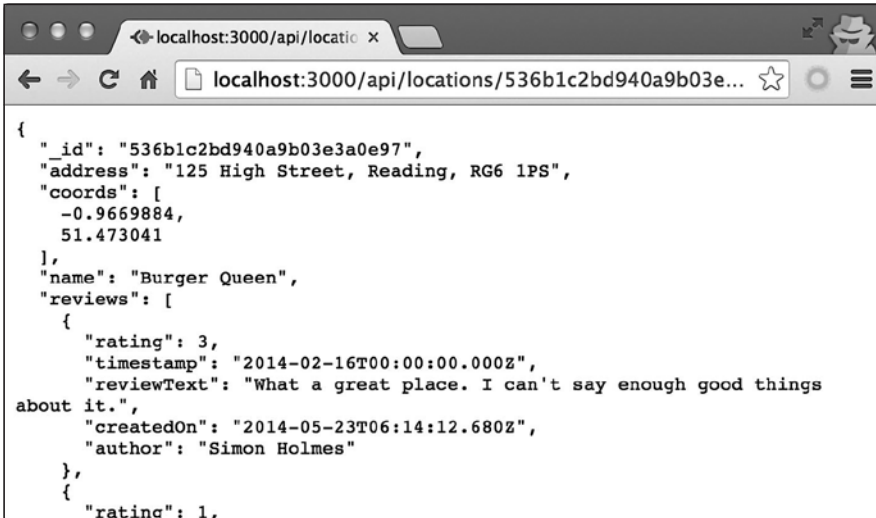


Рис. 6.7. Простейший контроллер для поиска отдельного местоположения по идентификатору, возвращающий браузеру объект JSON в случае обнаружения соответствующего идентификатора

Попробовали ли вы этот простейший контроллер? Попытались ли указать в URL неправильный идентификатор места? Если да, то должны были увидеть, что вам ничего не возвращается — ни предупреждения, ни сообщения, просто код состояния 200, извещающий, что все в порядке, но при этом никаких данных.

Перехват ошибок

Проблема с этим простейшим контроллером в том, что он выводит только ответ об успехе независимо от того, успешным ли было выполнение. Подобное поведение для API — это не очень здорово. Хороший API должен возвращать код ошибки, если что-то пошло не так.

Для возвращения кода ошибки контроллер должен быть настроен таким образом, чтобы перехватывать потенциальные ошибки и отправлять соответствующий ответ. При таком раскладе перехват ошибок обычно использует операторы `if`. Каждому оператору `if` должен соответствовать оператор `else` или должен быть оператор `return`.

СОВЕТ

Код API ни при каких обстоятельствах не должен оставлять запрос без ответа.

В нашем простейшем контроллере мы должны перехватывать три типа ошибок:

- ❑ параметры запроса не содержат `locationid`;
- ❑ метод `findById` не возвращает местоположения;
- ❑ метод `findById` возвращает ошибку.

Код состояния неудачно завершившегося запроса GET — 404. С учетом этого итоговый код контроллера для поиска и возврата отдельного местоположения выглядит так, как показано в листинге 6.2.

Листинг 6.2. Контроллер `locationsReadOne`

```

module.exports.locationsReadOne = function(req, res) {
  if (req.params && req.params.locationid) {
    loc
      .findById(req.params.locationid)
      .exec(function(err, location) {

        if (!location) {
          sendJsonResponse(res, 404, {
            "message": "locationid not found"
          });
          return;
        } else if (err) {
          sendJsonResponse(res, 404, err);
          return;
        }

        sendJsonResponse(res, 200, location);
      });
  } else {
    sendJsonResponse(res, 404, {
      "message": "No locationid in request"
    });
  }
};

```

1 Прерывание из-за обнаружения ошибки 1: проверяем, имеется ли в параметрах запроса `locationid`

2 Прерывание из-за обнаружения ошибки 2: если `Mongoose` не возвращает местоположения, отправляем сообщение 404 и выходим из области видимости функции с помощью оператора `return`

3 Прерывание из-за обнаружения ошибки 3: если `Mongoose` вернул ошибку, отправляем ее в качестве ответа 404 и выходим из контроллера с помощью оператора `return`

4 Если `Mongoose` не вернул ошибку, продолжаем работать как прежде и отправляем объект местоположения в ответе [с кодом состояния] 200

5 Если параметры запроса не содержат `locationid`, отправляем соответствующий ответ [с кодом состояния] 404

Листинг 6.2 использует оба метода перехвата с помощью операторов `if`. Прерывание вследствие обнаружения ошибки 1 **1** использует `if` для проверки существования в объекте запроса объекта `params`, а также того, что объект `params` содержит значение `locationid`. Этот условный оператор завершается оператором `else` **5** для случая, когда не найдены или объект `params`, или значение `locationid`. И прерывание вследствие обнаружения ошибки 2 **2**, и прерывание вследствие обнаружения ошибки 3 **3** используют `if` для проверки того, не возвращает ли ошибку `Mongoose`. Каждый из этих `if` содержит оператор `return`, предотвращающий дальнейшее выполнение кода в области видимости функции обратного вызова. Если ошибки найдено не было, оператор `return` пропускается и код переходит к отправке успешного ответа **4**.

Каждое из этих прерываний вследствие обнаружения ошибки обеспечивает ответы на случай как успешного, так и неудачного завершения, полностью исключая вероятность того, что запрашивающая сторона будет оставлена в подвешенном состоянии. Если хотите, можете также сгенерировать несколько сообщений для `console.log`, чтобы легче было потом отслеживать в терминале происходящее. В исходном коде в GitHub приведено несколько примеров этого.

На рис. 6.8 показано различие между успешным и неудачным запросами с использованием расширения Postman браузера Chrome.

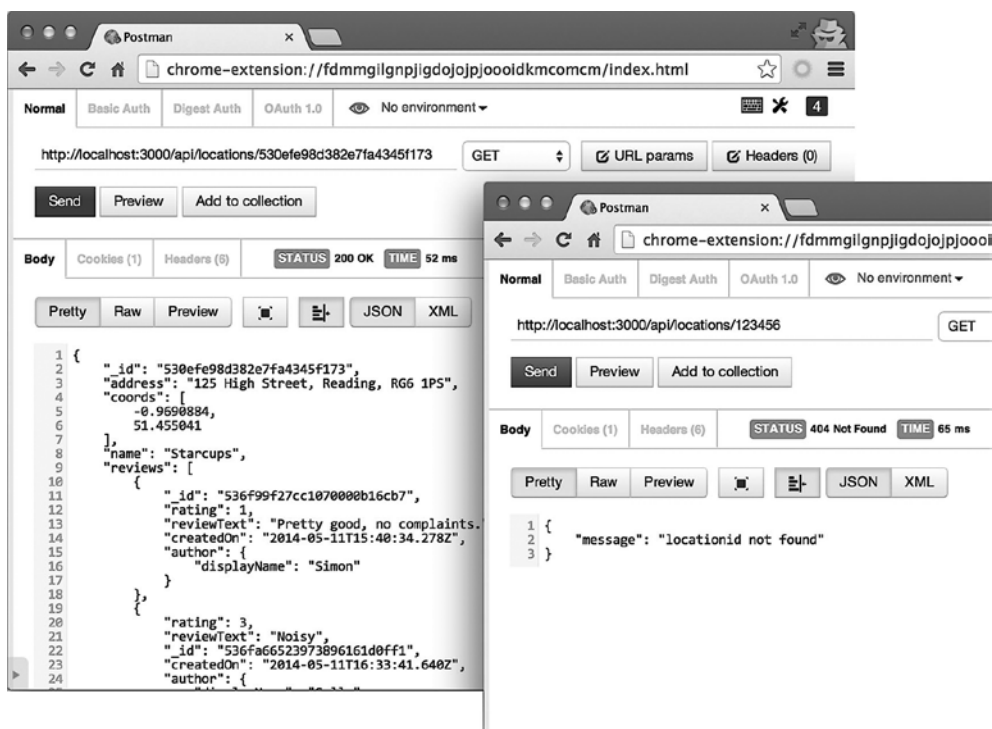


Рис. 6.8. Тестируем успешный (слева) и неудачный (справа) ответы API с помощью Postman

Мы закончили один маршрут API. Настало время взглянуть на второй запрос GET, предназначенный для возврата отдельного отзыва.

6.3.2. Поиск отдельного поддокумента на основе идентификаторов

Чтобы найти поддокумент, необходимо сначала найти родительский документ, аналогично тому, как мы находили отдельное местоположение по идентификатору. Когда документ будет найден, можно искать конкретный поддокумент.

Это значит, что можно использовать контроллер `locationsReadOne` в качестве отправной точки и внести в него несколько модификаций для создания контроллера `reviewsReadOne`. Эти модификации:

- ❑ принятие и использование дополнительного параметра URL `reviewid`;
- ❑ извлечение из документа только названия и отзывов, вместо того чтобы заставлять MongoDB возвращать весь документ;
- ❑ поиск отзыва с соответствующим ID;
- ❑ возвращение соответствующего JSON-ответа.

Для выполнения всего этого мы можем использовать несколько новых методов Mongoose.

Ограничение возвращаемых из MongoDB путей

При извлечении документа из MongoDB не всегда необходим документ целиком, иногда требуются только какие-то конкретные данные. Ограничение объема передаваемых данных благотворно сказывается на скорости и потреблении трафика.

Mongoose использует для этого метод `select`, объединяемый в цепочку с запросом модели. Например, следующий фрагмент кода сообщит MongoDB, что мы хотим получить только название какого-то места и отзывы о нем:

```
Loc
  .findById(req.params.locationid)
  .select('name reviews')
  .exec();
```

Метод `select` принимает разделенную пробелами строку из путей, которые мы хотим извлечь.

Использование Mongoose для поиска конкретного поддокумента

Mongoose предоставляет также вспомогательный метод для поиска поддокумента по ID. В Mongoose имеется метод, который обрабатывает массив поддокументов, принимая на входе искомый идентификатор. Метод `id` возвращает единственный подходящий поддокумент и используется следующим образом:

```
Loc
  .findById(req.params.locationid)
  .select('name reviews')
  .exec(
    function(err, location) {
      var review;
      review = location.reviews.id(req.params.reviewid);
    }
  );
```

| Передаем reviewid
из параметров
в метод id

В этом фрагменте кода в переменную `review` в обратном вызове будет возвращен один отзыв.

Добавление перехвата ошибок и объединение всего воедино

Теперь у нас имеются все ингредиенты, необходимые для создания контроллера `reviewsReadOne`. Начав с копии контроллера `locationsReadOne`, мы можем выполнить все изменения, необходимые для возврата только одного отзыва.

Листинг 6.3 демонстрирует контроллер `reviewsReadOne` из файла `review.js` (изменения выделены полужирным шрифтом).

Листинг 6.3. Контроллер для поиска отдельного отзыва

```

module.exports.reviewsReadOne = function(req, res) {
  if (req.params && req.params.locationid && req.params.reviewid) {
    Loc
      .findById(req.params.locationid)
      .select('name reviews')
      .exec(
        function(err, location) {
          var response, review;
          if (!location) {
            sendJsonResponse(res, 404, {
              "message": "locationid not found"
            });
            return;
          } else if (err) {
            sendJsonResponse(res, 400, err);
            return;
          }

          if (location.reviews && location.reviews.length > 0) {
            review = location.reviews.id(req.params.reviewid);
            if (!review) {
              sendJsonResponse(res, 404, {
                "message": "reviewid not found"
              });
            } else {
              response = {
                location : {
                  name : location.name,
                  id : req.params.locationid
                },
                review : review
              };
              sendJsonResponse(res, 200, response);
            }
          }
        }
      );
  }
}

```

Проверяем, что среди параметров имеется `reviewid`

Добавляем метод Mongoose `select` к запросу модели, сообщая, что мы хотим получить название местоположения и отзывы о нем

Проверяем, что у возвращенного местоположения имеются отзывы

Используем метод `id` поддокумента Mongoose в качестве вспомогательного метода для поиска соответствующего идентификатора

Если отзыв найден, создаем объект ответа, возвращая отзыв, название места и ID

Если отзыв не найден, возвращаем ответ

```

    }
    } else {
      sendJsonResponse(res, 404, {
        "message": "No reviews found"
      });
    }
  }
);
} else {
  sendJsonResponse(res, 404, {
    "message": "Not found, locationid and reviewid are both required"
  });
}
};

```

Если никакие отзывы не найдены, возвращаем соответствующее сообщение об ошибке

После сохранения все это можно протестировать опять-таки с помощью Postman. Вам понадобятся правильные значения ID, которые можно получить непосредственно из MongoDB через командную оболочку Mongo. Команда `db.locations.find()` возвращает все местоположения и отзывы о них. Можете проверить, что получится, если указать для места или отзыва ошибочный идентификатор или использовать идентификатор отзыва о другом месте.

6.3.3. Поиск нескольких документов с помощью геопространственных запросов

Домашняя страница Loc8r должна отображать список мест, основываясь на текущем географическом расположении пользователя. В MongoDB и Mongoose имеются специальные методы геопространственных запросов для поиска мест, расположенных вблизи от заданного.

Мы будем использовать здесь метод `geoNear` Mongoose для поиска списка мест, расположенных вблизи от заданной точки, вплоть до заданного максимального расстояния. `geoNear` — метод модели, принимающий три параметра:

- ❑ географическую точку `geoJSON`;
- ❑ объект с параметрами;
- ❑ функцию обратного вызова.

Следующий фрагмент кода демонстрирует его базовую структуру:

```
Loc.geoNear(point, options, callback);
```


В отличие от метода `findById`, у `geoNear` нет метода `exec`. Вместо этого `geoNear` выполняется сразу же, а код, который необходимо выполнить по его завершении, пересылается в обратном вызове.

Структура точки geoJSON

Первый параметр метода `geoNear` — точка `geoJSON`. Точка `geoJSON` — простой объект JSON, содержащий в массиве широту и долготу. Структура точки `geoJSON` показана в следующем фрагменте кода:

```
var point = {
  type: "Point",
  coordinates: [lng, lat]
};
```

Объявляем объект

Описываем его как имеющий тип Point (Точка)

Задаем координаты долготы и широты в массиве, сначала указывая долготу

В настроенном тут для получения списка местоположений маршруте нет координат в параметрах URL, а значит, их необходимо задать иначе. Строка запроса идеально подходит для этого типа данных, следовательно, URL запроса будет выглядеть примерно следующим образом:

```
api/locations?lng=-0.7992599&lat=51.378091
```

Express, конечно, позволяет вам обращаться к значениям в строке запроса, помещая их в объект `query`, присоединяемый к объекту `request`, например `req.query.lng`. Значения долготы и широты после извлечения будут иметь строковый тип, но их необходимо добавить в объект `point` в виде чисел. Для этого можно воспользоваться функцией JavaScript `parseFloat`. В следующем фрагменте кода все собрано вместе: показано, как получить координаты из строки запроса и создать необходимую для функции `geoNear` точку `geoJSON`:

```
module.exports.locationsListByDistance = function(req, res) {
  var lng = parseFloat(req.query.lng);
  var lat = parseFloat(req.query.lat);
  var point = {
    type: "Point",
    coordinates: [lng, lat]
  };
  Loc.geoNear(point, options, callback);
};
```

Получаем координаты из строки запроса и преобразуем их из строк в числа

Создаем точку geoJSON

Отправляем точку в качестве первого параметра метода geoNear

Конечно, этот контроллер пока что не будет работать, ведь как `options`, так и `callback` еще не определены. Мы займемся этим сейчас, начав с параметров (`options`).

Добавление в geoNear обязательных параметров

У метода `geoNear` есть только один обязательный параметр — `spherical`. Он определяет, будет ли поиск выполняться на сферическом объекте или на плоскости. В наши дни принято считать, что Земля круглая, так что установим параметр `spherical` в значение `true`.

Создавая объект для хранения параметров, получаем следующий фрагмент кода:

```
var geoOptions = {
  spherical: true
};
```

Теперь поиск будет выполняться на основе координат на сфере.

Ограничение результатов geoNear по количеству

Зачастую хочется позаботиться о сервере API — и о видимой конечным пользователям скорости его реакции, — ограничив количество результатов при возврате списка. Этого можно добиться добавлением в метод `geoNear` параметра `num`. Нужно просто указать максимальное количество результатов, которые необходимо вернуть.

Следующий фрагмент кода демонстрирует предыдущий объект `geoOptions` с добавлением этого параметра, ограничивающего возвращаемый набор данных десятью объектами:

```
var geoOptions = {
  spherical: true,
  num: 10
};
```

Теперь поиск будет возвращать не более десяти ближайших результатов.

Ограничение результатов geoNear по расстоянию

Еще один способ контроля работы API при возврате основанных на учете местоположения данных — ограничение списка результатов расстоянием от центральной точки. Теоретически это лишь вопрос добавления еще одного параметра под названием `maxDistance`. Проблема в том, что MongoDB выполняет вычисления в радианах, а не в метрах или милях и ожидает на входе `maxDistance` в радианах. Это дает возможность MongoDB легко выполнять вычисления на сфере любого размера, а не только на Земле, но не облегчает нашу задачу.

Преобразовать физические расстояния в радианы несложно (рис. 6.9).

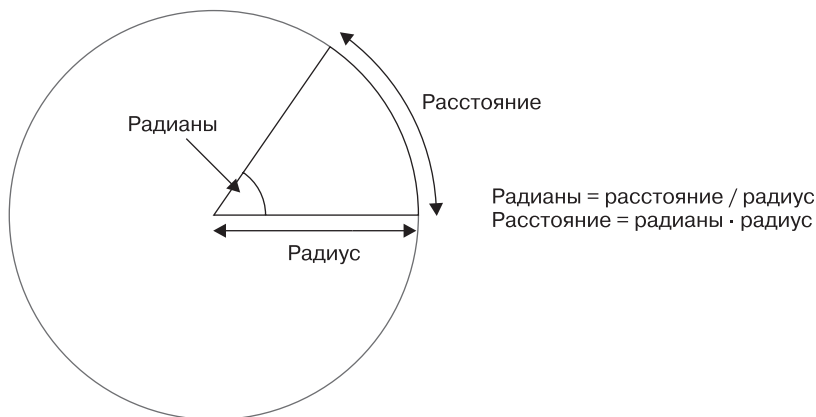


Рис. 6.9. Зависимость между расстоянием и радианами и преобразование одного в другое при известном радиусе

Радиус Земли равен 6371 км, или 3959 миль, но в данной книге мы будем использовать километры, поскольку, честно говоря, с ними работать проще! Обладая этой информацией, мы можем создать функцию `theEarth`, предоставляющую два метода для выполнения этих вычислений. Следующий фрагмент кода должен находиться вверху файла `locations.js` контроллера API, сразу после запроса Mongoose и настройки модели:

```

var theEarth = (function(){
  var earthRadius = 6371; // километров, или 3959 миль ←
  var getDistanceFromRads = function(rads) {
    return parseFloat(rads * earthRadius);
  };
  var getRadsFromDistance = function(distance) {
    return parseFloat(distance / earthRadius);
  };
  return {
    getDistanceFromRads : getDistanceFromRads,
    getRadsFromDistance : getRadsFromDistance
  };
})();

```

Определяем константу для радиуса Земли

Создаем функцию для преобразования радианов в расстояние

Создаем функцию для преобразования расстояния в радианы

Делаем эти две функции видимыми

Мы получили пригодные для повторного использования функции вычисления расстояния.

СОВЕТ

Если такой паттерн программирования вам непривычен, загляните в раздел «Написание модульного JavaScript» приложения к книге, расположенного в Интернете.

Теперь мы можем добавить в параметры значение `maxDistance`, а затем добавить эти параметры в контроллер следующим образом:

```
module.exports.locationsListByDistance = function(req, res) {
  var lng = parseFloat(req.query.lng);
  var lat = parseFloat(req.query.lat);
  var point = {
    type: "Point",
    coordinates: [lng, lat]
  };
  var geoOptions = {
    spherical: true,
    maxDistance: theEarth.getRadsFromDistance(20),
    num: 10
  };
  Loc.geoNear(point, geoOptions, callback);
};
```

Создаем объект параметров, включая указание максимального расстояния 20 км

Изменяем функцию `geoNear` для использования объекта `geoOptions`

УПРАЖНЕНИЕ

Попробуйте извлечь максимальное расстояние из строки запроса, вместо того чтобы зашивать его в функцию. На GitHub в коде для данной главы дано решение этого упражнения.

Это был последний из параметров, необходимых для поиска в БД с помощью `geoNear`, так что настало время начать работу над выводимой информацией.

Обзор вывода `geoNear`

У завершающего обратного вызова метода `geoNear` имеется три параметра, стоящих в следующем порядке.

1. Объект ошибок.
2. Объект результатов.
3. Объект статистики.

Если запрос завершится удачно, объект ошибок будет неопределенным, объект результатов будет содержать массив результатов, а объект статистики — информацию о запросе, такую как затраченное время, количество просмотренных докумен-

тов, среднее расстояние и максимальное расстояние для возвращенных документов. Мы начнем с работы над удачно завершающимся запросом, а потом добавим перехват ошибок.

После успешно завершившегося запроса `geoNear` MongoDB возвращает массив объектов. Каждый из этих объектов содержит значение расстояния и возвращенный документ из БД. Другими словами, MongoDB не добавляет расстояние в данные. Следующий фрагмент кода демонстрирует пример возвращаемых данных, сокращенный для экономии места:

```
[{
  dis: 0.002532674663406363,
  obj: {
    name: 'Starcups',
    address: '125 High Street, Reading, RG6 1PS'
  }
}]
```

В этом массиве содержится только один объект, но при успешном завершении запроса обычно возвращаются сразу несколько. Метод `geoNear` фактически возвращает весь документ в объекте `obj`. Из-за этого возникают две задачи.

1. API не должно возвращать больше данных, чем необходимо.
2. Желательно возвращать расстояние в привычном виде (не в радианах) как составную часть возвращаемого набора данных.

Так что до того, как отправлять данные обратно в качестве ответа, необходимо определенным образом их обработать.

Обработка вывода `geoNear`

Прежде чем API сможет отправить ответ, следует убедиться в том, что оно отправляет то и только то, что нужно. Нам известно, какие данные необходимы для перечня на домашней странице, так как мы уже сделали контроллер домашней страницы в файле `app_server/controllers/location.js`. Функция `homelist` отправляет некоторое количество объектов местоположений вот таким образом:

```
{
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '100m'
}
```

Для создания объекта в соответствии с этими строками из результатов необходимо просто организовать цикл по результатам и поместить соответствующие данные в новый массив. Затем можно вернуть эти обработанные данные

с ответом, содержащим код состояния 200. Как это могло бы выглядеть, показано в следующем фрагменте кода:

```
Loc.geoNear(point, options,
            function (err, results, stats) {
  var locations = [];
  results.forEach(function(doc) {
    locations.push({
      distance: theEarth.getDistanceFromRads(doc.dis),
      name: doc.obj.name,
      address: doc.obj.address,
      rating: doc.obj.rating,
      facilities: doc.obj.facilities,
      _id: doc.obj._id
    });
  });
  sendJsonResponse(res, 200, locations);
});
```

Создаем новый массив для хранения обработанных данных результатов

Цикл по результатам запроса geoNear

Получение расстояния и преобразование из радианов в километры с помощью созданной ранее вспомогательной функции

Помещение остатка запрошенных данных в возвращаемый объект

Отправка обработанных данных обратно в виде JSON-ответа

Если вы протестируете этот маршрут API с помощью Postman (не забудьте добавить координаты долготы и широты в строку запроса), то увидите что-то напоминающее рис. 6.10.

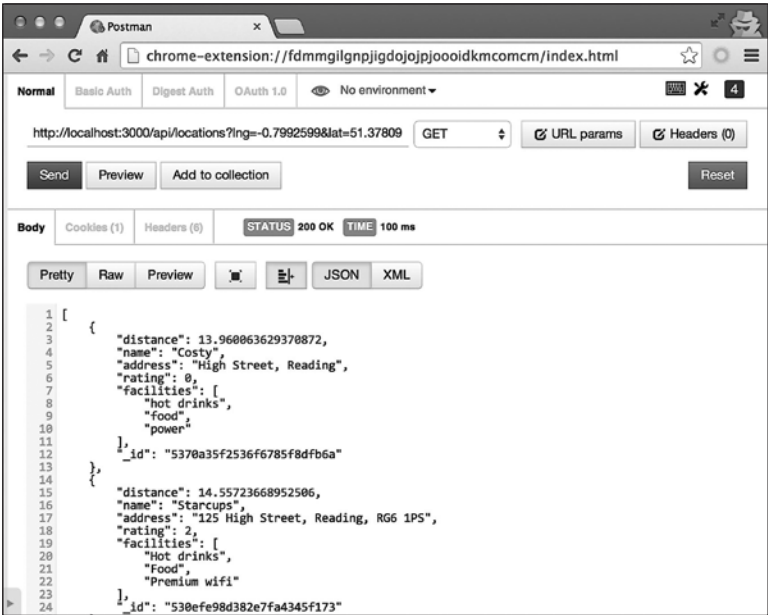


Рис. 6.10. Тестирование маршрута для списка местоположений в Postman должно вернуть код состояния 200 и список результатов, зависящих от географических координат, переданных в строке запроса

УПРАЖНЕНИЕ

Попробуйте передать результаты внешней поименованной функции для построения списка местоположений. Эта функция должна возвращать обработанный список, который затем можно будет передать в JSON-ответе.

Если вы при проверке работы этого маршрута передадите слишком удаленные от тестовых данных координаты, то все равно должны получить код состояния 200, но возвращаемый массив будет пустым.

Добавление перехвата ошибок

И снова мы начали с создания функциональности для случая успеха, а теперь необходимо добавить прерывания на случай обнаружения ошибок, чтобы гарантировать, что API всегда будет отправлять адекватный ответ.

Необходимые нам прерывания должны проверять, что:

- ❑ все параметры были отправлены правильно;
- ❑ функция `geoNear` не возвращает ошибку.

Листинг 6.4 демонстрирует итоговый полностью готовый контроллер, включая прерывания вследствие обнаружения ошибок.

Листинг 6.4. Контроллер для списка местоположений `locationsListByDistance`

```
module.exports.locationsListByDistance = function(req, res) {
  var lng = parseFloat(req.query.lng);
  var lat = parseFloat(req.query.lat);
  var point = {
    type: "Point",
    coordinates: [lng, lat]
  };
  var geoOptions = {
    spherical: true,
    maxDistance: theEarth.getRadsFromDistance(20),
    num: 10
  };
  if (!lng || !lat) {
    sendJsonResponse(res, 404, {
      "message": "lng and lat query
        parameters are required"
    });
    return;
  }
  Loc.geoNear(point, geoOptions, function(err, results, stats) {
    var locations = [];
    if (err) {
      sendJsonResponse(res, 404, err);
    }
  });
}
```

Проверяем, что параметры запроса — долгота и широта — заданы, причем в правильном формате, если нет — возвращаем ошибку 404 и сообщение

Если запрос `geoNear` возвращает ошибку, отправляем ее в качестве ответа с кодом состояния 404

```

} else {
  results.forEach(function(doc) {
    locations.push({
      distance: theEarth.getDistanceFromRads(doc.dis),
      name: doc.obj.name,
      address: doc.obj.address,
      rating: doc.obj.rating,
      facilities: doc.obj.facilities,
      _id: doc.obj._id
    });
  });
  sendJsonResponse(res, 200, locations);
}
});
};

```

Рассмотрение запросов GET, которые должен обслуживать наш API, завершено, пришло время заняться запросами POST.

6.4. Методы POST: добавление данных в MongoDB

Методы POST связаны с созданием документов или поддокументов в БД с последующим возвращением сохраненных данных в качестве подтверждения. В маршрутах для Loc8r имеется два запроса POST, выполняющих разные действия, перечисленные в табл. 6.7.

Таблица 6.7. Два запроса POST из API Loc8r

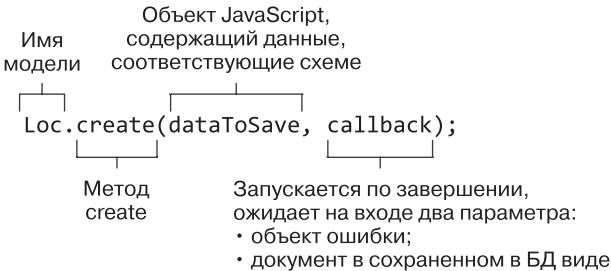
Действие	Метод	Путь URL	Параметр	Пример
Создание нового местоположения	POST	/locations		http://loc8r.com/api/locations
Создание нового отзыва	POST	/locations/ locationid/ reviews	locationid	http://api.loc8r.com/ locations/123/reviews

Методы POST работают путем добавления отправленных им данных форм в БД. Аналогично тому, как параметры URL доступны через `req.params`, а строки запросов — через `req.query`, контроллеры Express обращаются к отправленным данным форм через `req.body`.

Начнем с рассмотрения способов создания документов.

6.4.1. Создание новых документов в MongoDB

В базе данных для `Loc8r` каждое местоположение представляет собой документ, и именно их мы будем создавать в данном разделе. `Mongoose` вряд ли мог бы сделать процесс создания документов MongoDB еще более удобным. Берется модель, для нее вызывается метод `create`, передаются какие-либо данные и функция обратного вызова. Такова минимальная структура, связанная с нашей моделью `Loc`:



Итак, это довольно просто. В процессе создания можно выделить два основных этапа.

1. Получение отправленных данных формы и использование их для создания объекта JavaScript, соответствующего схеме.
2. Отправка нужного ответа в обратном вызове в зависимости от успешного или неудачного выполнения операции `create`.

Глядя на этап 1, мы понимаем, что можем получить отправленные нам в форме данные с помощью модели `req.body`, да и этап 2 вам уже должен быть знаком. Так что перейдем прямо к коду. Следующий листинг демонстрирует целиком контроллер `locationsCreate`, предназначенный для создания нового документа.

Листинг 6.5. Полный код контроллера, предназначенного для создания нового документа

```
module.exports.locationsCreate = function(req, res) {
  Loc.create({
    name: req.body.name,
    address: req.body.address,
    facilities: req.body.facilities.split(","),
    coords: [parseFloat(req.body.lng),
              parseFloat(req.body.lat)],
    openingTimes: [{
      days: req.body.days1,
      opening: req.body.opening1,
      closing: req.body.closing1,
      closed: req.body.closed1,
    }, {
      days: req.body.days2,
```

Создаем для модели метод create

Создаем массив предоставляемых услуг, разделяя список запятыми

Выполняем синтаксический разбор координат для преобразования из строк в числа

```

    opening: req.body.opening2,
    closing: req.body.closing2,
    closed: req.body.closed2,
  }]
}, function(err, location) {
  if (err) {
    sendJsonResponse(res, 400, err);
  } else {
    sendJsonResponse(res, 201, location);
  }
});
};

```

Предоставляем функцию обратного вызова, содержащую подходящие ответы для случаев успеха и неудачи

Приведенный код демонстрирует простоту создания нового документа и сохранения каких-либо данных в MongoDB. Ради краткости мы ограничили массив `openingTimes` двумя элементами, но его легко можно расширить, а еще лучше — использовать цикл с проверкой на существование значений.

Вы могли также заметить, что здесь нигде не задавались оценки (`rating`). Как вы помните, в схеме мы задали значение по умолчанию, равное `0`, как показано в следующем фрагменте кода:

```
rating: {type: Number, "default": 0, min: 0, max: 5},
```

Это выполняется при создании документа, в результате устанавливается начальное значение, равное `0`. В этом коде вам в глаза должно было броситься еще кое-что. Здесь нет проверок!

6.4.2. Проверка данных с помощью Mongoose

В этом контроллере нет никакого кода проверок, так что же мешает кому-то добавить множество пустых или неполных документов? Ранее в некоторых схемах Mongoose мы присваивали флагу `required` значение `true`. Когда этот флаг задан, Mongoose не будет отправлять данные в MongoDB.

В следующей базовой схеме для местоположений, например, можно увидеть, что как `name`, так и `coords` являются обязательными полями:

```

var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere', required: true},
  openingTimes: [openingTimeSchema],
  reviews: [reviewSchema]
});

```

Если любое из этих полей отсутствует, метод `create` сгенерирует ошибку и не станет пытаться сохранить документ в БД.

Проверка этого маршрута API в Postman будет выглядеть так, как показано на рис. 6.11. Обратите внимание на то, что указан метод `post` и что выбранный тип данных (над списком названий и значений) — `x-www-form-urlencoded`.

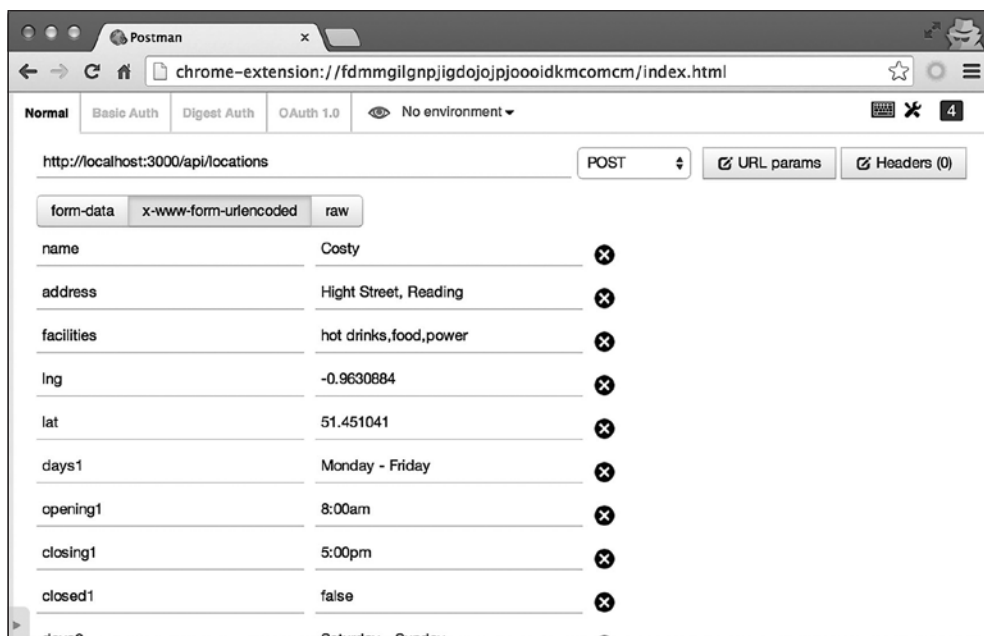


Рис. 6.11. Проверяем метод `POST` в Postman, убеждаясь, что параметры метода и данных форм заданы правильно

6.4.3. Создание новых поддокументов в MongoDB

В контексте местоположений `Loc8r` отзывы являются поддокументами. Поддокументы создаются и сохраняются через их родительские документы. Другими словами, для создания и сохранения нового поддокумента необходимо следующее:

1. Найти соответствующий родительский документ.
2. Добавить новый поддокумент.
3. Сохранить родительский документ.

Найти соответствующий родительский документ — не проблема, ведь мы уже это делали и можем использовать результат как основу для следующего контроллера, `reviewsCreate`. После нахождения родительского документа мы можем вызвать внешнюю функцию для выполнения следующего этапа, как показано в листинге 6.6.

Листинг 6.6. Контроллер для создания отзыва

```

module.exports.reviewsCreate = function(req, res) {
  var locationid = req.params.locationid;
  if (locationid) {
    Loc
      .findById(locationid)
      .select('reviews')
      .exec(
        function(err, location) {
          if (err) {
            sendJsonResponse(res, 400, err);
          } else {
            doAddReview(req, res, location); ←
          }
        }
      );
  } else {
    sendJsonResponse(res, 404, {
      "message": "Not found, locationid required"
    });
  }
};

```

В случае успеха операции поиска будет вызвана новая функция для добавления отзыва, которой будут переданы объекты запроса, ответа и местоположения

Здесь не происходит ничего для нас нового, мы все это уже видели раньше. Вставив вызов новой функции, мы сделали код аккуратнее за счет снижения уровня вложенности и количества отступов и тем самым упростили тестирование.

Добавление и сохранение поддокумента

После нахождения родительского документа и извлечения из него списка существующих поддокументов необходимо добавить новый поддокумент. Поддокументы — по сути, массивы объектов, и простейший способ добавления нового объекта — создание объекта данных и использование метода `push` JavaScript. Следующий фрагмент кода демонстрирует это:

```

location.reviews.push({
  author: req.body.author,
  rating: req.body.rating,
  reviewText: req.body.reviewText
});

```

Мы получаем отправленные данные формы, поэтому используем `req.body`.

Как только поддокумент добавлен, необходимо сохранить родительский документ, поскольку поддокументы не могут быть сохранены сами по себе. Для сохранения документа у MongoDB имеется метод модели `save`, ожидающий на входе функцию обратного вызова с параметром ошибки и параметром возвращенного объекта. Следующий фрагмент кода показывает это в действии:

```
location.save(function(err, location) {
  var thisReview;
  if (err) {
    sendJsonResponse(res, 400, err);
  } else {
    thisReview = location.reviews[location.reviews.length - 1];
    sendJsonResponse(res, 201, thisReview);
  }
});
```

Ищем последний отзыв в возвращенном массиве, поскольку MongoDB вернет весь родительский документ, а не только новый поддокумент

Возвращенный методом `save` поддокумент — это весь родительский документ, а не только новый поддокумент. Чтобы вернуть в ответе API нужные данные, а именно поддокумент, необходимо извлечь последний поддокумент из массива ❶.

При добавлении документов и поддокументов необходимо учитывать возможное влияние этого действия на другие данные. В `Loc8r` добавление нового отзыва влечет добавление новой оценки. Эта оценка повлияет на оценку документа в целом. Поэтому при успешном сохранении отзыва мы вызовем еще одну функцию для обновления средней оценки.

Объединяя все это и добавляя перехват ошибок в функцию `doAddReview`, получаем следующий код (листинг 6.7).

Листинг 6.7. Добавление и сохранение поддокумента

```
var doAddReview = function(req, res, location) {
  if (!location) {
    sendJsonResponse(res, 404, {
      "message": "locationid not found"
    });
  } else {
    location.reviews.push({
      author: req.body.author,
      rating: req.body.rating,
      reviewText: req.body.reviewText
    });
    location.save(function(err, location) {
      var thisReview;
      if (err) {
        sendJsonResponse(res, 400, err);
      } else {
        updateAverageRating(location._id);
        thisReview = location.reviews[location.reviews.length - 1];
        sendJsonResponse(res, 201, thisReview);
      }
    });
  }
};
```

Если указан родительский документ...

...вставляем новые данные в массив поддокументов...

...перед сохранением

В случае успешного сохранения вызываем функцию для обновления средней оценки

Извлекаем последний добавленный отзыв и возвращаем его в качестве подтверждения ответа JSON

Обновление средней оценки

Вычисление средней оценки — не слишком сложная задача, так что не будем говорить о ней слишком подробно. Задача состоит из следующих этапов.

1. Найти по заданному идентификатору нужный документ.
2. Организовать цикл по поддокументам отзывов, складывая оценки.
3. Вычислить среднее значение оценки.
4. Обновить значение оценки родительского документа.
5. Сохранить документ.

Преобразование списка этапов в код дает нам что-то вроде следующего листинга, который необходимо поместить в файл контроллера `reviews.js` среди относящихся к отзывам контроллеров (листинг 6.8).

Листинг 6.8. Вычисление и обновление средней оценки

```
var updateAverageRating = function(locationid) {
  Loc
  .findById(locationid)
  .select('rating reviews')
  .exec(
    function(err, location) {
      if (!err) {
        doSetAverageRating(location);
      }
    }
  );
};

var doSetAverageRating = function(location) {
  var i, reviewCount, ratingAverage, ratingTotal;
  if (location.reviews && location.reviews.length > 0) {
    reviewCount = location.reviews.length;
    ratingTotal = 0;
    for (i = 0; i < reviewCount; i++) {
      ratingTotal = ratingTotal + location.reviews[i].rating;
    }
    ratingAverage = parseInt(ratingTotal / reviewCount, 10);
    location.rating = ratingAverage;
    location.save(function(err) {
      if (err) {
        console.log(err);
      } else {
        console.log("Average rating updated to",
          ratingAverage);
      }
    });
  }
};
```

Находим по заданному идентификатору нужный документ

Проходим в цикле по поддокументам отзывов, складывая оценки

Обновляем значение оценки родительского документа

Сохраняем родительский документ

Вычисляем среднее значение оценки

Вы могли заметить, что мы не отправляем тут никакого JSON-ответа. Дело в том, что мы его уже отправили. Вся операция асинхронна и не должна влиять на отправку ответа API, подтверждающего сохранение отзыва.

Добавление отзыва — не единственный случай, когда нам требуется обновлять среднюю оценку. Поэтому еще важнее сделать эти функции доступными из внешних контроллеров и как можно менее сцепленными с действиями по созданию отзыва.

Сделанное только что дало возможность взглянуть на использование Mongoose для обновления данных в MongoDB, а теперь перейдем к методам PUT нашего API.

6.5. Методы PUT: обновление данных в MongoDB

Методы PUT связаны с обновлением существующих документов или поддокументов в БД и последующим возвращением сохраненных данных в качестве подтверждения. В маршрутах для Loc8r есть два запроса PUT, выполняющих различные задачи (табл. 6.8).

Таблица 6.8. Два запроса PUT из API Loc8r для обновления местоположений и отзывов

Действие	Метод	Путь URL	Параметр	Пример
Обновление конкретного местоположения	PUT	/locations	locationid	http://loc8r.com/api/locations/123
Обновление конкретного отзыва	PUT	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/locations/123/reviews/abc

Методы PUT аналогичны методам POST, поскольку основаны на использовании отправленных им данных форм. Но методы PUT используют данные для обновления существующих документов, а не создания новых.

6.5.1. Использование Mongoose для обновления документа в MongoDB

В Loc8r нам может понадобиться обновить место для добавления новых услуг, изменения часов работы или исправления каких-либо других данных. Подход к обновлению данных в документе, вероятно, покажется вам знакомым. Он состоит из следующих этапов.

1. Найти соответствующий документ.
2. Выполнить какие-либо изменения экземпляра.
3. Сохранить документ.
4. Отправить JSON-ответ.

Реализовать его можно благодаря тому, что экземпляр модели Mongoose однозначно соответствует документу в MongoDB. Когда запрос находит документ, вы получаете экземпляр модели. Если вы внесете в этот экземпляр изменения, а затем сохраните его, Mongoose обновит исходный документ в БД в соответствии с изменениями.

6.5.2. Метод Mongoose save

На самом деле мы уже видели его в действии при обновлении значения средней оценки. Метод `save` применяется к экземпляру модели, возвращаемому функцией `find`. Он ожидает на входе функцию обратного вызова с обычными параметрами в виде объекта ошибки и возвращенного объекта данных.

Упрощенный каркас подобного подхода показан в следующем фрагменте кода:

```

Loc
  .findById(req.params.locationid) ← Поиск документа
  .exec(                               для обновления
    function(err, location) {
      location.name = req.body.name; ← Вносим изменения
                                     в экземпляр модели, меняя
                                     значение одного из путей
      location.save(function(err, location) { ← Сохраняем документ
        if (err) {                               с помощью
          sendJsonResponse(res, 404, err);       метода save Mongoose
        } else {
          sendJsonResponse(res, 200, location); ← Возвращаем ответ
        }                                         об успехе или неудаче
      });
    });
  );
};

```

Здесь ясно видны отдельные этапы поиска, обновления, сохранения и отправки ответа. Воплощение этого каркаса в контроллере `locationsUpdateOne` с добавлением перехвата ошибок и данными, которые мы хотели бы сохранить, дает следующий листинг (листинг 6.9).

Листинг 6.9. Внесение изменений в существующий документ в MongoDB

```

module.exports.locationsUpdateOne = function(req, res) {
  if (!req.params.locationid) {
    sendJsonResponse(res, 404, {
      "message": "Not found, locationid is required"
    });
    return;
  }
  Loc
  .findById(req.params.locationid) ← Поиск документа
  .select('-reviews -rating')      ← местоположения
  .exec(                             по заданному идентификатору
    function(err, location) {
      if (!location) {
        sendJsonResponse(res, 404, {
          "message": "locationid not found"
        });
        return;
      } else if (err) {
        sendJsonResponse(res, 400, err);
        return;
      }
      location.name = req.body.name;
      location.address = req.body.address;
      location.facilities = req.body.facilities.split(",");
      location.coords = [parseFloat(req.body.lng),
        parseFloat(req.body.lat)];
      location.openingTimes = [{
        days: req.body.days1,
        opening: req.body.opening1,
        closing: req.body.closing1,
        closed: req.body.closed1,
      }, {
        days: req.body.days2,
        opening: req.body.opening2,
        closing: req.body.closing2,
        closed: req.body.closed2,
      }];
      location.save(function(err, location) { ← Сохранение
        if (err) {                               экземпляра
          sendJsonResponse(res, 404, err);
        } else {
          sendJsonResponse(res, 200, location); ← Отправка
        }                                       соответствующего
      }                                       ответа в зависимости
    }                                       от результата
  }                                       операции сохранения
});
};

```

Обновление путей значениями из отправленной формы

Безусловно, теперь, когда все реализовано, здесь стало намного больше кода, но все равно можно легко выделить ключевые этапы процесса обновления.

Наиболее зоркие из вас могли обратить внимание на нечто странное в операторе `select`:

```
.select('-reviews -rating')
```

Ранее мы использовали метод `select` для указания столбцов, которые *хотим* выбрать. Ставя перед названием пути тире, мы сообщаем, что *не хотим* извлекать его из БД. То есть данный оператор дает указание извлечь все, кроме `reviews` и `rating`.

6.5.3. Обновление существующего поддокумента в MongoDB

Обновление поддокумента выполняется практически так же, как и обновление документа, за одним исключением. После нахождения документа нужно найти поддокумент, в котором вы будете вносить свои изменения. После этого метод `save` применяется к документу, а не к поддокументу. Так что этапы обновления поддокумента будут следующими.

1. Найти соответствующий документ.
2. Найти соответствующий поддокумент.
3. Внести изменения в поддокумент.
4. Сохранить документ.
5. Отправить JSON-ответ

Для `Loc8r` обновляемые поддокументы — это отзывы, так что важно не забыть пересчитать среднюю оценку. Это единственное, что нам нужно сделать дополнительно, помимо перечисленных пяти этапов. Листинг 6.10 демонстрирует все это в готовом виде в контроллере `reviewsUpdateOne`.

Листинг 6.10. Обновление поддокумента в MongoDB

```
module.exports.reviewsUpdateOne = function(req, res) {
  if (!req.params.locationid || !req.params.reviewid) {
    sendJsonResponse(res, 404, {
      "message": "Not found, locationid and reviewid are both required"
    });
    return;
  }
  Loc
  .findById(req.params.locationid) ← Находим
    .select('reviews')                родительский документ
}
```

```

.exec(
  function(err, location) {
    var thisReview;
    if (!location) {
      sendJsonResponse(res, 404, {
        "message": "locationid not found"
      });
      return;
    } else if (err) {
      sendJsonResponse(res, 400, err);
      return;
    }
    if (location.reviews && location.reviews.length > 0) {
      thisReview = location.reviews.id(req.params.reviewid);
      if (!thisReview) {
        sendJsonResponse(res, 404, {
          "message": "reviewid not found"
        });
      } else {
        thisReview.author = req.body.author;
        thisReview.rating = req.body.rating;
        thisReview.reviewText = req.body.reviewText;
        location.save(function(err, location) {
          if (err) {
            sendJsonResponse(res, 404, err);
          } else {
            updateAverageRating(location._id);
            sendJsonResponse(res, 200, thisReview);
          }
        });
      }
    } else {
      sendJsonResponse(res, 404, {
        "message": "No review to update"
      });
    }
  }
);
};

```

Находим поддокумент

Выполняем изменения в поддокументе на основе предоставленных данных формы

Сохраняем родительский документ

Возвращаем JSON-ответ, отправляя объект поддокумента в зависимости от успешности сохранения

В этом листинге хорошо видны пять этапов обновления: найти документ, найти поддокумент, выполнить изменения, сохранить и отправить ответ. Снова значительная часть кода связана с перехватом ошибок, поскольку это жизненно важно для создания надежного и адаптивного API. Конечно же, вы не захотите сохранять неправильные данные, отправлять ошибочные ответы или удалять данные, которые удалять не нужно. Кстати, говоря об удалении данных, давайте рассмотрим последний из четырех используемых нами методов API — DELETE.

6.6. Метод DELETE: удаление данных из MongoDB

Метод DELETE, что неудивительно, предназначен для удаления документов или поддокументов из БД. В маршрутах для Loc8r имеется запрос DELETE для удаления местоположения и еще один — для удаления отзыва. Подробности приведены в табл. 6.9.

Таблица 6.9. Два запроса DELETE из API Loc8r для удаления местоположений и отзывов

Действие	Метод	Путь URL	Параметр	Пример
Удаление конкретного местоположения	DELETE	/locations	locationid	http://loc8r.com/api/locations/123
Удаление конкретного отзыва	DELETE	/locations/ locationid/ reviews	locationid reviewid	http://loc8r.com/api/ locations/123/reviews/ abc

Начнем с рассмотрения процесса удаления документов.

6.6.1. Удаление документов из MongoDB

Mongoose чрезвычайно упрощает удаление документов из MongoDB, предоставляя нам метод `findByIdAndRemove`. Этот метод принимает на входе всего один параметр — ID документа, который нужно удалить.

В случае ошибки API должен вернуть 404, а в случае удачного выполнения — 204. Листинг 6.11 демонстрирует все это в готовом виде в контроллере `locationsDeleteOne`.

Листинг 6.11. Удаление документа с заданным идентификатором из MongoDB

```

module.exports.locationsDeleteOne = function(req, res) {
  var locationid = req.params.locationid;
  if (locationid) {
    Loc
      .findByIdAndRemove(locationid) ← Вызываем
      .exec(                               метод findByIdAndRemove,
        function(err, location) {           передавая ему locationid
          if (err) {                       Выполняем
            sendJsonResponse(res, 404, err); ← метод
            return;
          }
          sendJsonResponse(res, 204, null); ← Возвращаем ответ
        }                                   в зависимости от успешного
      )                                    или неудачного выполнения
  }
};
} else {
  sendJsonResponse(res, 404, {

```

```
    "message": "No locationid"
  });
}
};
```

Это быстрый и удобный способ удаления документа, но, если хотите, можете разбить его на два этапа: сначала найти документ, а затем его удалить. Это дает возможность при необходимости выполнить с документом какие-то действия, прежде чем удалить. Это показано в следующем фрагменте кода:

```
Loc
  .findById(locationid)
  .exec(
    function (err, location) {
      // Выполнить с документом какие-то действия
      Loc.remove(function(err, location){
        // Подтвердить успешность выполнения или его неудачу
      });
    }
  );
```

При этом появляется не только дополнительный уровень вложенности, но и дополнительный уровень гибкости, который может оказаться вам полезен.

6.6.2. Удаление поддокумента из MongoDB

Процесс удаления поддокумента не отличается от других проделанных нами с поддокументами действий — все выполняется через родительский документ. Этапы удаления поддокумента следующие.

1. Найти родительский документ.
2. Найти соответствующий поддокумент.
3. Удалить поддокумент.
4. Сохранить родительский документ.
5. Подтвердить успешность выполнения или его неудачу.

Собственно удаление поддокумента тоже не представляет трудностей благодаря еще одному вспомогательному методу Mongoose. Вы уже видели, что поддокумент можно найти по его идентификатору с помощью метода `id` следующим образом:

```
location.reviews.id(reviewid)
```

Mongoose позволяет связать с ним в цепочку метод `remove`, добавив его в конец предыдущего оператора, вот так:

```
location.reviews.id(reviewid).remove()
```

Этот оператор удалит поддокумент из массива. Не забывайте, конечно, что родительский документ необходимо будет после этого сохранить, чтобы сохранить изменения в БД. Собрав в контроллере `reviewsDeleteOne` все этапы воедино и добавив перехват ошибок, мы получим следующий листинг (листинг 6.12).

Листинг 6.12. Поиск и удаление поддокумента из MongoDB

```

module.exports.reviewsDeleteOne = function(req, res) {
  if (!req.params.locationid || !req.params.reviewid) {
    sendJsonResponse(res, 404, {
      "message": "Not found, locationid and reviewid are both required"
    });
    return;
  }
  Loc
  .findById(req.params.locationid) ← Ищем соответствующий
  .select('reviews')                родительский документ
  .exec(
    function(err, location) {
      if (!location) {
        sendJsonResponse(res, 404, {
          "message": "locationid not found"
        });
        return;
      } else if (err) {
        sendJsonResponse(res, 400, err);
        return;
      }
      if (location.reviews && location.reviews.length > 0) {
        if (!location.reviews.id(req.params.reviewid)) {
          sendJsonResponse(res, 404, {
            "message": "reviewid not found"
          });
          } else {
            location.reviews.id(req.params.reviewid).remove(); ← Находим и удаляем
                                                                    соответствующий
                                                                    поддокумент за один шаг
          location.save(function(err) {
            if (err) {
              sendJsonResponse(res, 404, err); ← Возвращаем ответ
            } else {
              updateAverageRating(location._id); ← в зависимости
              sendJsonResponse(res, 204, null); ← от успешного
                                                    или неудачного
                                                    выполнения
            }
          });
        } else {
          sendJsonResponse(res, 404, {
            "message": "No review to delete"
          });
        }
      }
    }
  );
};

```

Сохраняем
родительский
документ

Опять же бóльшая часть приведенного кода — перехват ошибок: имеется семь возможных ответов API и только один из них — об успешном выполнении. Фактически удаление поддокумента — очень простая задача, нужно только быть совершенно уверенными, что удаляешь нужный поддокумент.

Так как мы тут удаляем отзыв, с которым связана оценка, нужно не забыть вызвать функцию `updateAverageRating` для пересчета средней оценки для данного места. Конечно, вызывать ее нужно только в случае успеха операции удаления.

Вот и все. Мы создали API REST в Express и Node, который умеет принимать HTTP-запросы GET, POST, PUT и DELETE для выполнения операций CRUD в базе данных MongoDB.

6.7. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Рекомендуемые решения для создания API REST, включая различные URL, методы запросов и коды ответов.
- ❑ Соответствие HTTP-запросов GET, POST, PUT и DELETE типичным операциям CRUD.
- ❑ Вспомогательные методы Mongoose.
- ❑ Взаимодействие с данными посредством моделей Mongoose, а также взаимно однозначное соответствие экземпляра модели документу в БД.
- ❑ Работа с поддокументами через их родительские документы, поскольку обращаться к поддокументам самим по себе (или сохранять их) нельзя.
- ❑ Обеспечение надежности API путем проверок на всевозможные ошибки, которые только можно себе представить, что гарантирует, что запрос никогда не останется без ответа.

В главе 7 мы собираемся рассмотреть использование API из приложения Express, наконец-то делая сайт `Loc8r` ориентированным на работу с БД!

Глава 7

Потребление API REST: использование API из Express

В этой главе:

- ❑ обращение к API из приложения Express;
- ❑ обработка и использование возвращаемых API данных;
- ❑ работа с кодами ответов API;
- ❑ отправка данных из браузера обратно API;
- ❑ проверки и прерывания вследствие обнаружения ошибок.

Это захватывающая глава! В ней мы впервые свяжем клиентскую часть с серверной. Мы уберем зашитые данные из контроллеров и дойдем до отображения вместо этого в браузере данных из БД. Сверх того при создании новых поддокументов мы организуем вставку данных из браузера в БД посредством API.

Основные технологии, на которых будет сосредоточена эта глава, — Node и Express. Наши грандиозные планы и место этой главы в общей архитектуре иллюстрирует рис. 7.1.

В данной главе мы обсудим, как обращаться к API из Express и как обрабатывать ответы. Мы будем выполнять обращения к API для чтения из БД и записи в нее. По ходу дела мы рассмотрим перехват ошибок, обработку данных и созда-

ние пригодного для повторного использования кода посредством разделения обязанностей. Ближе к концу главы мы рассмотрим различные слои архитектуры, в которые можно добавить проверку правильности данных, и их практическую пользу.

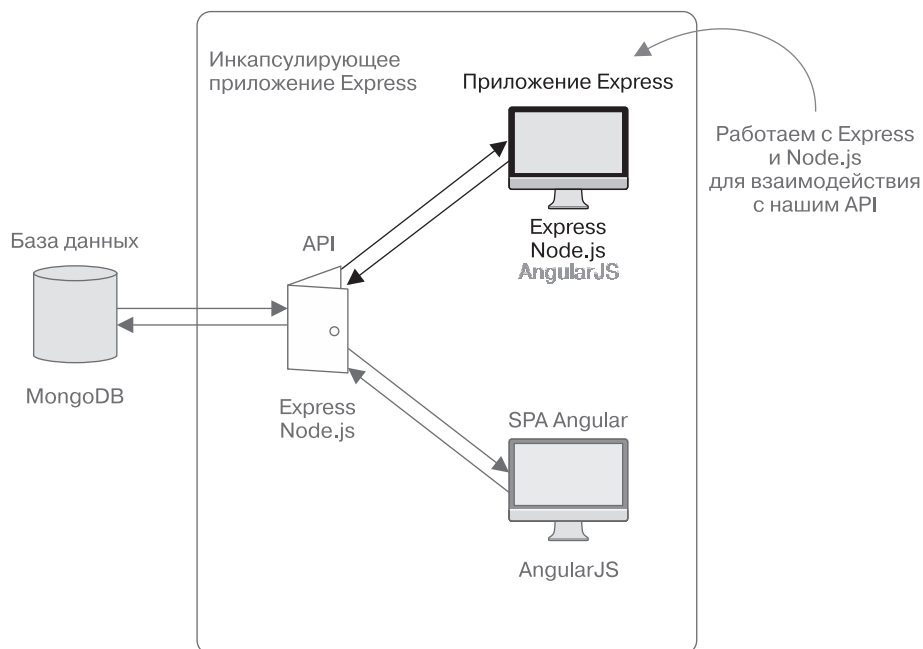


Рис. 7.1. Эта глава будет посвящена модификации приложения Express из главы 4 для целей взаимодействия с API REST, созданным в главе 6

Начнем с рассмотрения обращений к API из приложения Express.

7.1. Обращение к API из Express

Первое, что нам необходимо рассмотреть, — то, как обратиться к API из Express. Это касается не только нашего API, такой подход можно использовать для вызова любого API.

Наше приложение Express должно уметь выполнять вызовы URL API, настроенных в главе 6, конечно, с использованием нужного метода запроса, а затем интерпретировать ответ. Для этого воспользуемся модулем `request`.

7.1.1. Добавление в наш проект модуля request

Модуль `request` ничем не отличается от пакетов, которые мы использовали раньше, его можно добавить в наш проект с помощью `npm`. Для установки самой свежей версии и добавления ее в файл `package.json` перейдите в терминал и введите следующую команду:

```
$ npm install --save request
```

После завершения работы `npm` можно включить `request` в файлы, которые будут его использовать. Только один файл из `Loc8r` должен будет выполнять обращения к API — это файл, содержащий все контроллеры для основного серверного приложения. Итак, добавьте вверху файла `locations.js`, размещенного в каталоге `app_server/controllers`, следующую строку, чтобы выполнить `require` пакета `request`:

```
var request = require('request');
```

Теперь все готово и мы можем приступить к работе!

7.1.2. Настройка параметров по умолчанию

У каждого вызова API с помощью `request` должен быть полный URL. Это значит, что URL не должен быть относительной ссылкой, а должен содержать полный адрес. Но этот URL может различаться для сред разработки и эксплуатации.

Чтобы избежать необходимости выполнения проверки в каждом выполняющем вызов API контроллере, можно однократно настроить параметры конфигурации по умолчанию вверху файла контроллеров. Чтобы использовался соответствующий URL в зависимости от среды, можно воспользоваться нашей доброй знакомой — переменной среды `NODE_ENV`.

Реализуя это, получаем такую верхнюю часть файла контроллеров, как показано в листинге 7.1.

Листинг 7.1. Добавление запроса и параметров по умолчанию в файл контроллеров

```
var request = require('request');
var apiOptions = {
  server : "http://localhost:3000"
};
if (process.env.NODE_ENV === 'production') {
  apiOptions.server = "https://getting-mean-loc8r.herokuapp.com";
}
```

Настройка URL сервера по умолчанию для локальной разработки

Если приложение работает в промышленном режиме, задаем другой базовый URL, меняя на реальный адрес приложения в Интернете

После этого каждый выполняемый нами вызов API может ссылаться на `apiOptions.server` и будет использовать правильный базовый URL.

7.1.3. Модуль request

Простейшая конструкция выполнения запроса — это всего одна команда, принимающая на входе параметры и функцию обратного вызова:

```

      Объект JavaScript
      с описанием запроса
    ┌───┴───┐
request(options, callback)
    └───┬───┘
          Запускаемая
          после получения ответа функция
  
```

Параметры задают все необходимое для запроса, включая URL, метод запроса, тело запроса и параметры строки запроса. Это те параметры, которые мы будем использовать в данной главе, подробнее они описаны в табл. 7.1.

Таблица 7.1. Четыре часто используемых параметра запроса для задания вызова API

Параметр	Описание	Обязательность
url	Полный URL предназначенного к выполнению запроса, включая протокол, домен, путь и параметры URL	Да
method	Метод запроса, например GET, POST, PUT или DELETE	Нет — если не указано, то по умолчанию используется GET
json	Тело запроса в виде объекта JavaScript; если данных тела не требуется, должен быть отправлен пустой объект	Да — гарантирует, что при синтаксическом разборе тело ответа также будет интерпретироваться как JSON
qs	Объект JavaScript, представляющий параметры строки запроса	Нет

Следующий фрагмент кода демонстрирует пример использования всего этого для запроса GET. У запроса GET не обязательно должно быть тело для отправки, но могут быть параметры строки запроса.

```

var requestOptions = {
  url : "http://yourapi.com/api/path",
  method : "GET",
  json : {},
  qs : {
    offset : 20
  }
};
  
```

Задаем URL для будущего вызова API
 Задаем метод запроса
 Описываем тело запроса, даже если это пустой объект JSON
 При необходимости добавляем какие-либо параметры строки запроса для использования в API

Имеется еще очень много параметров, которые можно задать, но наиболее распространенные — эти четыре, и именно их мы будем использовать в данной главе. Чтобы получить больше информации о других параметрах, загляните в справочник в репозитории на GitHub <https://github.com/mikeal/request>.

Функция обратного вызова запускается после того, как API вернет ответ. У нее имеется три параметра: объект ошибки, полный ответ и подвергнутое синтаксическому разбору тело ответа. Если не было перехвачено ошибки, объект ошибки будет хранить `null`. Тремя наиболее полезными элементами данных в нашем коде будут код состояния ответа, тело ответа и сгенерированные ошибки. Следующий фрагмент кода демонстрирует пример конструкции обратного вызова для функции `request`:

```
function(err, response, body) {
  if (err) {
    console.log(err);
  } else if (response.statusCode === 200) {
    console.log(body);
  } else {
    console.log(response.statusCode);
  }
}
```

Если была передана ошибка, выполняем с ней какие-то действия

Если код состояния ответа равен 200 (запрос был успешен), выводим JSON-тело ответа

Если код состояния ответа не равен 200, делаем что-то другое

Полный объект ответа содержит огромное количество информации, так что не будем разбирать его тут. Вы всегда сможете посмотреть все самостоятельно в сообщении `console.log`, когда мы начнем добавлять в приложение обращения к API.

Собранный воедино каркас выполнения вызовов API будет выглядеть следующим образом:

```
var requestOptions = {
  url : "http://yourapi.com/api/path",
  method : "GET",
  json : {},
  qs : {
    offset : 20
  }
};

request(requestOptions, function(err, response, body) {
  if (err) {
    console.log(err);
  } else if (response.statusCode === 200) {
    console.log(body);
  } else {
    console.log(response.statusCode);
  }
});
```

Задание параметров запроса

Выполнение запроса, отправка параметров и предоставление функции обратного вызова для использования ответов нужным образом

Отправимся дальше — воплотим эту теорию в жизнь и начнем модификацию контроллеров `Loc8r` для использования уже созданного API.

7.2. Использование списков данных из API: домашняя страница `Loc8r`

На текущий момент в файле контроллера, который будет совершать работу, уже должен быть выполнен запрос модуля `request`, а также заданы значения по умолчанию. Так что теперь начинается самое интересное — модификация контроллеров для выполнения вызовов API и извлечение из БД данных для страниц.

У нас имеется две основные страницы, которым требуются данные: домашняя страница с перечнем местоположений и страница `Details` (Подробности) с более подробной информацией о конкретном месте. Начнем с самого начала и получим из БД данные для домашней страницы.

Текущий контроллер домашней страницы содержит только оператор `res.render`, отправляющий представлению жестко зашитые данные. Но мы хотели бы, чтобы все происходило иначе: домашняя страница визуализировалась после того, как API вернет какие-то данные. Контроллер домашней страницы все равно будет выполнять немало работы, так что давайте переместим эту визуализацию в отдельную функцию.

7.2.1. Разделение обязанностей: перенос визуализации в поименованную функцию

Есть несколько причин для перемещения визуализации в отдельную поименованную функцию. Во-первых, тем самым мы расцепляем визуализацию и логику приложения. Для процесса визуализации не имеет значения, откуда или как он получает данные: если данные находятся в нужном формате, он будет их использовать. Использование отдельной функции приближает нас к идеальной ситуации, когда каждая функция выполняет ровно одну задачу. Дополнительное преимущество этого состоит в пригодности функции для многократного использования, так что мы сможем вызывать ее из многих мест.

Вторая причина создания новой функции для визуализации домашней страницы — то, что процесс визуализации происходит внутри обратного вызова запроса API. А это делает код неудобным не только для тестирования, но и для чтения. Необходимый уровень вложенности сделает функцию контроллера большой и сложнораспределенной. Рекомендуемым решением будет попытаться избежать

этого, ведь такая функция окажется сложной для чтения и понимания в случае, если когда-нибудь придется снова к ней обратиться.

Вначале нужно создать в файле `locations.js` из каталога `app_server/controllers` новую функцию `renderHomepage` и переместить в нее содержимое контроллера `homelist`. Не забудьте убедиться в том, что она принимает также параметры `req` и `res`. В листинге 7.2 показана сильно урезанная версия того, что мы тут делаем. Теперь можно вызвать ее из контроллера `homelist`, как показано в листинге 7.2, и все будет работать как и прежде.

Листинг 7.2. Перемещение содержимого контроллера `homelist` во внешнюю функцию

```
var renderHomepage = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    ...
  });
};
module.exports.homelist = function(req, res){
  renderHomepage(req, res);
};
```

| Включаем весь код
| из вызова `res.render`
| (вырезано для краткости)

| Вызываем новую
| функцию `renderHomepage`
| из контроллера `homelist`

Это уже что-то, но мы еще далеки от результата — нам нужны данные!

7.2.2. Создание запроса API

Мы добудем необходимые данные, обратившись за ними к API, а для этого нужно произвести запрос. Для создания запроса нужно знать, какие URL, метод, тело JSON и строку запроса отправлять. Вспоминая главу 6 или обращаясь к самому коду API, можно увидеть, что необходимо предоставить указанную в табл. 7.2 информацию.

Таблица 7.2. Информация, необходимая для выполнения запроса к API о списке местоположений

Параметр	Значение
URL	SERVER:PORT/api/locations
Метод	GET
Тело JSON	null
Строка запроса	lng, lat, maxDistance

Преобразование этой информации в запрос не представляет трудностей. Как мы уже видели в данной главе, параметры для запроса представляют собой про-

сто объект JavaScript. Пока что мы жестко зашьем значения долготы и широты в параметры, так как это быстрее и удобнее для тестирования. Далее в книге сделаем приложение учитывающим местоположение. Пока что выберем координаты, близкие к месту хранения тестовых данных. Максимальное расстояние задаем равным 20 км.

При выполнении запроса передадим простую функцию обратного вызова для вызова функции `renderHomePage`, чтобы не оставлять браузер в подвешенном состоянии.

Реализация этого в коде в контроллере `homelist` выглядит следующим образом (листинг 7.3).

Листинг 7.3. Модификация контроллера `homelist` для вызова API перед визуализацией страницы

```
module.exports.homelist = function(req, res){
  var requestOptions, path;
  path = '/api/locations';
  requestOptions = {
    url : apiOptions.server + path,
    method : "GET",
    json : {},
    qs : {
      lng : -0.7992599,
      lat : 51.378091,
      maxDistance : 20
    }
  };
  request(
    requestOptions,
    function(err, response, body) {
      renderHomePage(req, res);
    }
  );
};
```

Задаем путь для запроса API (сервер уже указан вверху файла)

Задаем параметры запроса, включая URL, метод, пустое тело JSON и жестко зашитые параметры строки запроса

Выполняем запрос, отправляя параметры запроса

Предоставляем функцию обратного вызова для визуализации домашней страницы

Если вы это сохраните и перезапустите приложение, то домашняя страница должна отображаться точно так же, как и раньше. Мы уже выполняем запрос к API, правда, пока еще игнорируем ответ.

7.2.3. Использование данных ответа API

Учитывая, какие усилия мы приложили для обращения к API, самое меньшее, что можно сделать потом, — это использовать отправляемые им в ответ данные. Позднее мы обеспечим устойчивость к ошибкам, но пока что начнем с простого обеспечения работоспособности. При этом предположим, что тело ответа возвращается

в обратном вызове, так что мы можем передать его прямо в функцию `renderHomepage`, как показано в листинге 7.4.

Листинг 7.4. Модифицируем содержимое контроллера `homelist` для использования ответа API

```
request(
  requestOptions,
  function(err, response, body) {
    renderHomepage(req, res, body);
  }
);
```

← Передаем возвращенное
запросом тело
в функцию `renderHomepage`

В соответствии с тем, как мы программировали API, возвращаемый API ответ должен быть массивом местоположений. Функции `renderHomepage` требуется массив местоположений для отправки представлению, так что попробуем просто передать его. Изменения в листинге 7.5 выделены полужирным шрифтом.

Листинг 7.5. Модифицируем функцию `renderHomepage` для использования данных из API

```
var renderHomepage = function(req, res, responseBody){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places
      to work when out and about. Perhaps with coffee, cake or a pint? Let
      Loc8r help you find the place you're looking for.",
    locations: responseBody
  });
};
```

← Добавляем
дополнительный
параметр `responseBody`
в объявление функции

← Удаляем жестко зашитый
массив местоположений
и передаем вместо
него `responseBody`

Неужели это действительно так просто? Попробуйте в браузере — и увидите, что произойдет. Надеемся, вы увидите что-то вроде рис. 7.2.

Выглядит неплохо, да? Нужно что-нибудь сделать с отображением расстояния, но за исключением этого все данные приходят так, как мы и хотели. Подключение данных оказалось быстрым и удобным благодаря всей той работе, которую мы проделали загодя, проектируя представления, создавая основанные на представлениях контроллеры и разрабатывая основанную на контроллерах модель.

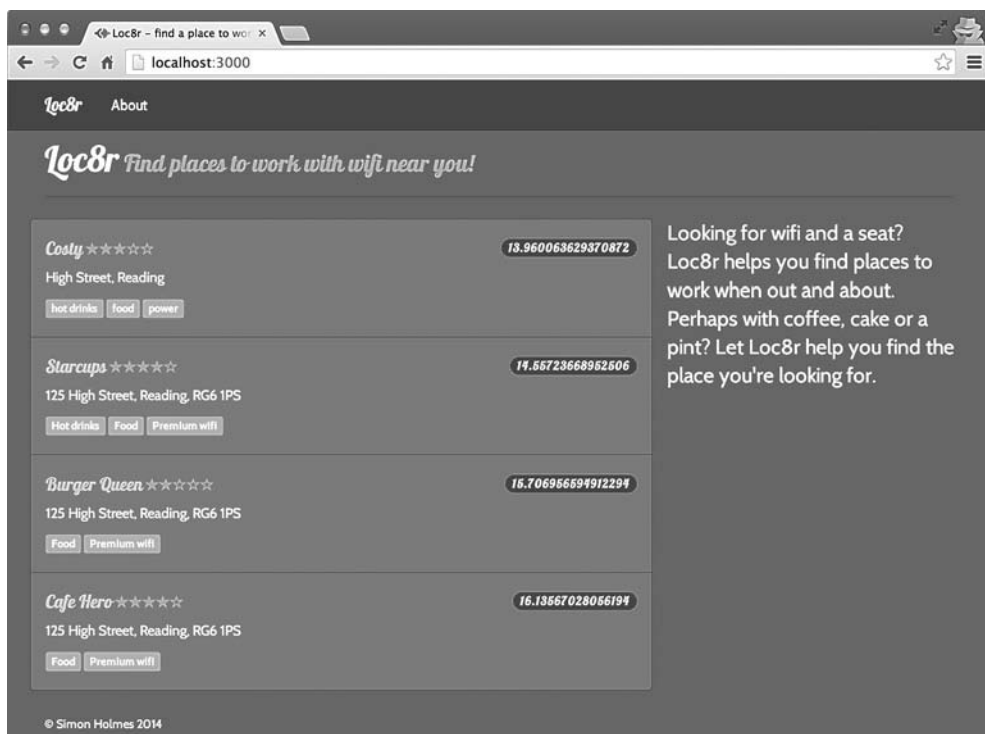


Рис. 7.2. Первый взгляд на использование данных из БД в браузере — мы близки к цели!

Мы добились того, чтобы все работало. Теперь необходимо кое-что улучшить. У нас пока нет перехвата ошибок, и нужно что-то сделать с расстояниями.

7.2.4. Изменение данных перед их отображением: приводим в порядок расстояния

Сейчас расстояния в списке отображаются с 15 цифрами после запятой и без единиц измерения, так что они чрезвычайно точны и совершенно бесполезны! Желательно, чтобы было понятно, в метрах или километрах указано каждое расстояние, а также округлить значения до целого числа метров или до десятых долей километра. Это нужно сделать до отправки данных функции `renderHomepage`, так как она должна служить для работы с самой визуализацией, а не для правки данных.

Для этого необходимо организовать цикл по массиву возвращенных местоположений, форматируя значение для расстояния до каждого из них. Вместо того чтобы делать это встроенным образом, мы создадим (в том же файле) внешнюю функцию `_formatDistance`, принимающую на входе расстояние и возвращающую его в красиво отформатированном виде.

Все вместе это будет выглядеть так, как показано в листинге 7.6. Обратите внимание на то, что каркас контроллера `homelist` не был включен в этот код ради краткости, а оператор `request` все еще находится в контроллере.

Листинг 7.6. Добавление и использование функции для форматирования возвращаемых API расстояний

```
request(
  requestOptions,
  function(err, response, body) {
    var i, data;
    data = body;
    for (i=0; i<data.length; i++) {
      data[i].distance =
        _formatDistance(data[i].distance);
    }
    renderHomepage(req, res, data);
  }
);
```

Присваиваем возвращенные данные тела новой переменной

Выполняем цикл по массиву, форматируя значения расстояний для местоположений

Отправляем на визуализацию измененные данные вместо исходного тела

```
var _formatDistance = function (distance) {
  var numDistance, unit;
  if (distance > 1) {
    numDistance = parseFloat(distance).toFixed(1);
    unit = 'km';
  } else {
    numDistance = parseInt(distance * 1000,10);
    unit = 'm';
  }
  return numDistance + unit;
};
```

Если расстояние превышает 1 км, округляем до одной цифры после запятой и добавляем единицу измерения [км]

В противном случае преобразуем в метры и округляем до ближайшего целого числа метров перед добавлением единицы измерения [м]

Если вы выполните эти изменения и обновите страницу, то должны будете увидеть, что расстояния теперь приобрели надлежащий вид и стали действительно пригодными для использования (рис. 7.3).

Уже лучше, домашняя страница выглядит похожей на ту, какой мы хотели ее видеть. В качестве упражнения можете добавить в функцию `_formatDistance` перехват ошибок, чтобы убедиться, что параметр `distance` действительно был передан и представляет собой число.

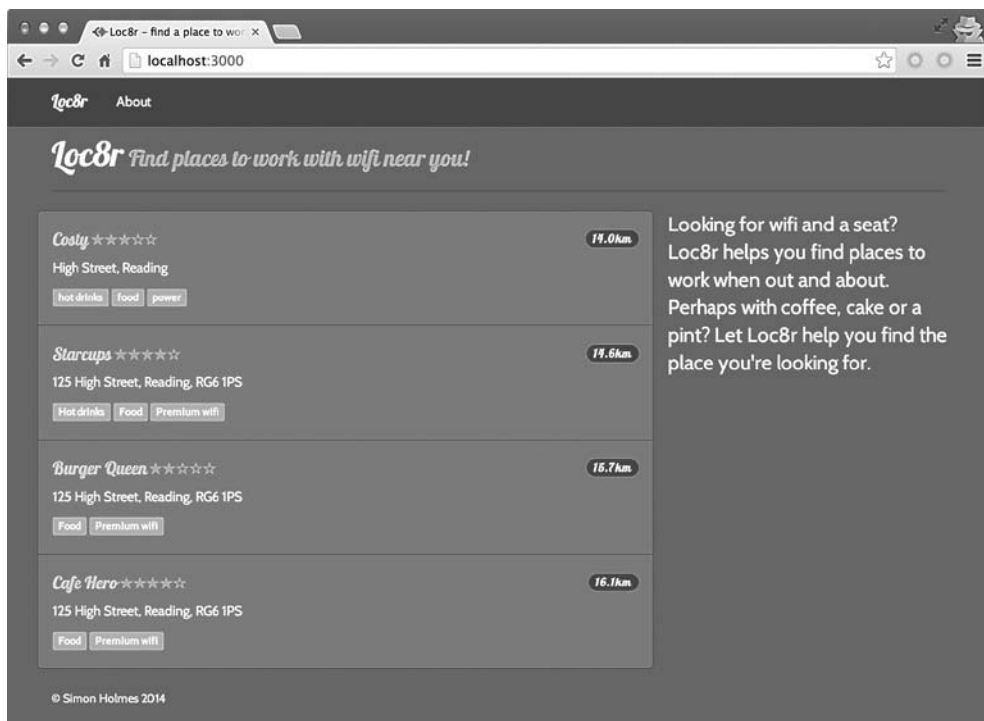


Рис. 7.3. Домашняя страница выглядит лучше после форматирования возвращенных API расстояний

7.2.5. Перехват возвращаемых API ошибок

До сих пор мы неявно предполагали, что с кодом состояния успешного завершения **200** API всегда будет возвращать массив данных. Но не обязательно всегда будет так. Мы запрограммировали API на возврат кода состояния **200** даже в том случае, когда рядом не найдено никаких местоположений. В настоящее время в подобном случае домашняя страница будет отображаться без какого-либо контента в центральной области. Пользователю было бы гораздо удобнее, если бы при отсутствии поблизости каких-то местоположений выводилось сообщение.

Нам известно также, что API может возвращать ошибки **404**, так что желательно убедиться, что мы обрабатываем их как полагается. Нам не нужно на самом деле отображать пользователю **404** в подобном случае, так как при этом не будет самой домашней страницы. Лучшим вариантом будет отправлять при этом сообщение браузеру в контексте домашней страницы.

Обработка этих сценариев использования не должна оказаться слишком сложной. Давайте посмотрим, как это сделать, начиная с контроллера.

Обеспечиваем устойчивость обратного вызова запроса к ошибкам

Одна из главных причин для перехвата ошибок — необходимость гарантировать, что они не вызовут сбой всего кода. Первое слабое место, похоже, имеется в обратном вызове `request`, где мы меняем ответ перед отправкой данных на визуализацию. Это допустимо, если данные всегда единообразны, но мы не можем позволить себе роскошь быть в этом уверенными.

Обратный вызов `request` в настоящий момент выполняет цикл `for` для форматирования расстояний вне зависимости от того, какие данные возвращает API. Вообще говоря, выполнять это необходимо только тогда, когда API возвращает код `200` и какие-то результаты.

Листинг 7.7 демонстрирует, как можно легко добиться этого посредством добавления простого оператора `if` для проверки кода состояния и длины возвращенных данных.

Листинг 7.7. Проверяем, вернул ли API какие-то данные, прежде чем пытаться их использовать

```
request(
  requestOptions,
  function(err, response, body) {
    var i, data;
    data = body;
    if (response.statusCode === 200 && data.length) {
      for (i=0; i<data.length; i++) {
        data[i].distance = _formatDistance(data[i].distance);
      }
    }
    renderHomepage(req, res, data);
  }
);
```

Выполняем цикл для форматирования расстояний только в том случае, если API вернул код состояния 200 и какие-то данные

Модификация этого фрагмента кода должна гарантировать, что обратный вызов не выдаст сбой и не сгенерирует ошибку в случае, если API вернул код состояния, отличный от `200`. Следующее звено в цепочке — функция `renderHomepage`.

Формирование выводимых сообщений в зависимости от данных ответа

Как и при обратном вызове `request`, при работе с функцией `renderHomepage` наша исходная цель — заставить ее работать в случае передачи для отображения массива местоположений. Теперь, когда могут быть переданы различные типы данных, необходимо сделать так, чтобы все возможные варианты обрабатывались соответствующим образом.

Тело ответа может представлять собой:

- массив местоположений;
- пустой массив, если не найдено никаких местоположений;
- строку, содержащую сообщение, в случае, когда API вернул ошибку.

У нас уже есть код для работы с массивом местоположений, так что осталось решить вопрос с двумя остальными вариантами. При перехвате этих ошибок хотелось бы также сформировать сообщение для возможной отправки представлению.

Для этого нам понадобится изменить функцию `renderHomepage` таким образом, чтобы она выполняла еще и следующее:

- создавала переменную-контейнер для сообщения;
- проверяла, является ли тело ответа массивом, в противном случае формировала соответствующее сообщение;
- если ответ — массив, формировала сообщение в случае, если он пуст (то есть если не были возвращены никакие местоположения);
- отправляла сообщение представлению.

Листинг 7.8 показывает, как это выглядит в исходном коде.

Листинг 7.8. Выводим сообщения, если API не возвращает данных о местоположениях

```
var renderHomepage = function(req, res,
                             responseBody){
  var message;
  if (!(responseBody instanceof Array)) {
    message = "API lookup error";
    responseBody = [];
  } else {
    if (!responseBody.length) {
      message = "No places found nearby";
    }
  }
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places to
      work when out and about. Perhaps with coffee, cake or a pint? Let
      Loc8r help you find the place you're looking for.",
    locations: responseBody,
    message: message
  });
};
```

Объявляем переменную для хранения сообщения

Если ответ не является массивом, задаем сообщение и указываем значением `responseBody` пустой массив

Если ответ — массив нулевой длины, задаем сообщение

Добавляем `message` в число отправляемых представлению переменных

Единственное, что может тут удивить, — место, в котором мы задаем в качестве значения `responseBody` пустой массив, если он изначально передавался в виде строки. Это было сделано, чтобы избежать генерации представлением ошибки. Представление ожидает, что в переменной `locations` ему будет передан массив, оно фактически не обращает внимания, если ему был отправлен пустой массив, но сгенерирует ошибку, если отправлена строка.

Последнее звено в этой цепочке — переделка представления, чтобы оно отображало сообщение, если таковое было отправлено.

Переделываем представление для отображения сообщений об ошибках

Итак, мы перехватываем ошибки из API, а также обрабатываем их для передачи чего-либо обратно пользователю. Итоговый шаг — дать пользователю возможность увидеть сообщение путем добавления заглушки в шаблон представления.

Нам не потребуется тут ничего слишком затейливого — достаточно простого раздела `div` с классом ошибки, который будет содержать сообщения. Листинг 7.9 демонстрирует раздел `block content` представления домашней страницы `locations-list.jade` из каталога `app_server/views`.

Листинг 7.9. Изменение представления для отображения при необходимости сообщения об ошибке

```
block content
  #banner.page-header
    .row
      .col-lg-6
        h1= pageHeader.title
        small &nbsp;#{pageHeader.strapline}
    .row
      .col-xs-12.col-sm-8
        .error= message
        .row.list-group
          each location in locations
            .col-xs-12.list-group-item
              h4
                a(href="/location")= location.name
                small &nbsp;
                  +outputRating(location.rating)
                span.badge.pull-right.badge-default= location.distance
                p.address= location.address
                p
                  each facility in location.facilities
                    span.label.label-warning= facility
                    &nbsp;
      .col-xs-12.col-sm-4
        p.lead= sidebar
```

← Добавляем `div` в область основного контента, который станет отображать сообщение, если таковое будет послано

Это довольно просто — даже не просто, а примитивно. На текущий момент этого вполне достаточно. Все, что осталось сделать, — протестировать.

Тестирование перехвата ошибок API

Как и при создании любого нового кода, теперь необходимо убедиться, что все работает. Простейший способ проверить это — изменить отправляемые в `requestOptions` значения строки запроса.

Для проверки прерывания на случай ошибки «не найдено никаких местоположений поблизости» (no places found nearby), можно или установить значение `maxDistance` равным очень маленькому числу (помните, что оно задается в километрах), или установить значения `lng` и `lat` равными координатам точки, возле которой нет местоположений, например:

```
requestOptions = {
  url : apiOptions.server + path,
  method : "GET",
  json : {},
  qs : {
    lng : 1,
    lat : 1,
    maxDistance : 0.002
  }
};
```

Меняем отправленные в запросе значения строки запроса, чтобы не было возвращено никаких результатов

ИСПРАВЛЯЕМ ИНТЕРЕСНУЮ ОШИБКУ

Пробовали ли вы проверить перехват ошибок API путем задания значений `lng` и `lat` равными 0? Вы могли бы ожидать увидеть при этом сообщение no places found nearby («не найдено никаких местоположений поблизости»), но вместо этого видите API lookup error («ошибка поиска API»). Это происходит из-за ошибки в коде перехвата ошибок API.

В контроллере `locationsListByDistance` отсутствие строковых параметров `lng` и `lat` обнаруживается с помощью типичной проверки JavaScript на ложность. В нашем коде просто используется следующее: `if (!lng || !lat)`.

В подобных тестах на ложность JavaScript ищет любые значения, которые он считает ложными, например пустые строки, неопределенные значения, `null` и, что для нас существенно, 0. Из-за 0 в коде появляется неожиданная ошибка. Если кто-то находится на экваторе или нулевом (то есть гринвичском) меридиане, он получит ошибку API.

Для исправления этой ситуации нужно сделать так, чтобы тест на ложность гласил: «если ложно, но не ноль». Код будет выглядеть вот так: `if (!(!lng && lng!==0) || (!lat && lat!==0))`.

Модификация контроллера описанным образом исключит эту ошибку.

Аналогичный подход можно использовать для тестирования ошибки 404. API ожидает, что будут отправлены все строковые параметры запроса, и вернет 404, если какие-то из них отсутствуют. Так что для быстрой проверки кода можно просто закомментировать один из них, как показано далее:

```
requestOptions = {
  url : apiOptions.server + path,
  method : "GET",
  json : {},
  qs : {
    // lng : -0.7992599, ←
    lat : 51.378091,
    maxDistance : 20
  }
};
```

Комментируем в запросе один из строковых параметров строки запроса, чтобы проверить, что произойдет при возврате API ошибки 404

Сделайте обе эти вещи по очереди и обновите домашнюю страницу, чтобы увидеть появление различных сообщений (рис. 7.4).

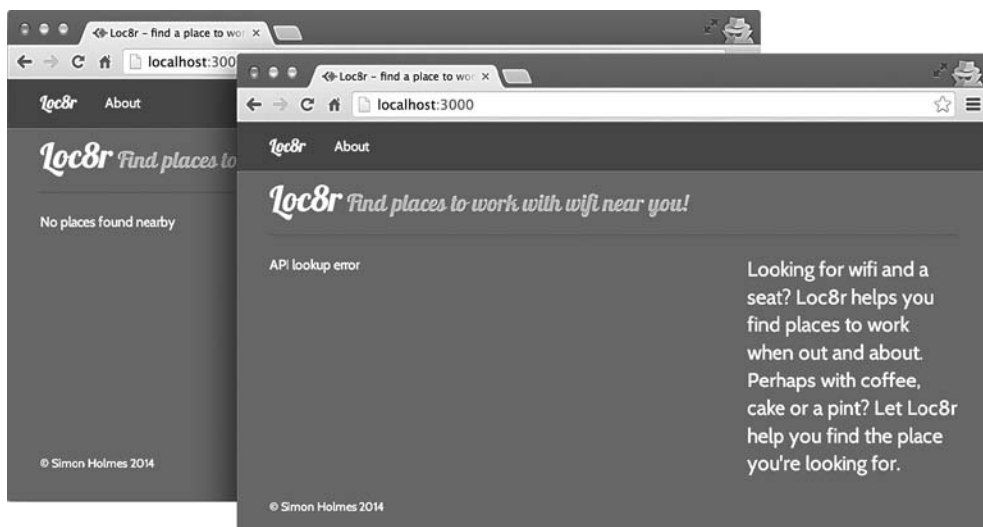


Рис. 7.4. Демонстрируем в представлениях сообщение после перехвата возвращенных API ошибок

Как видим, домашняя страница настроена хорошо. Приложение Express отправляет запрос созданному нами API, который извлекает данные из базы данных MongoDB и отправляет их обратно приложению. Когда приложение получает ответ от API, оно определяет, что с этим ответом делать, после чего отображает в браузере либо данные, либо сообщение об ошибке.

Теперь сделаем то же самое для страницы Details (Подробности), на этот раз работая с отдельными экземплярами данных.

7.3. Получение от API отдельных документов: страница Details приложения Loc8r

Страница Details (Подробности) должна отображать всю имеющуюся у нас информацию о конкретном местоположении: название, адрес, оценку, отзывы, предоставляемые услуги и карту, где оно обозначено. На данный момент в ней используются жестко зашитые в контроллер данные, и выглядит она так, как показано на рис. 7.5.

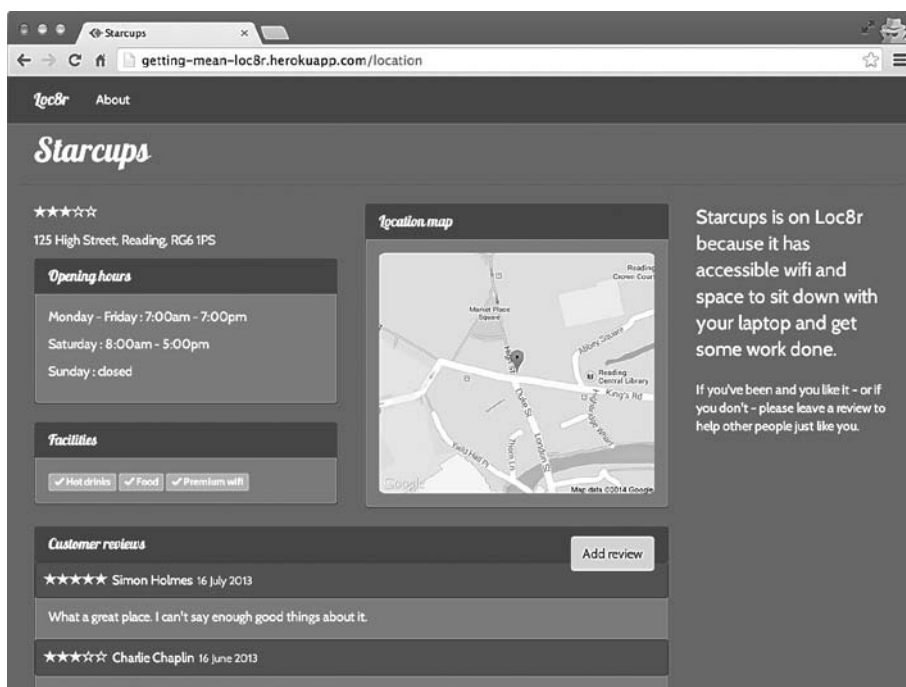


Рис. 7.5. Страница Details (Подробности) в текущем виде, использующая жестко зашитые в контроллер данные

В данном разделе мы модифицируем приложение так, чтобы иметь возможность указывать, для какого местоположения необходимы подробности, получать их из API и выводить в браузере. Конечно, не забудем и о перехвате ошибок.

7.3.1. Настройка URL и маршрутов для обращения к конкретным документам MongoDB

Текущий путь к странице Details (Подробности) — просто `/location`. Он не дает нам возможности указывать, на какое местоположение мы хотим посмотреть. Для решения этой задачи позаимствуем метод маршрутов API, где мы указывали ID документа местоположения в виде параметра URL.

Маршрут API для отдельного местоположения — `/api/locations/:locationid`. Можно сделать то же самое для основного приложения Express, добавив в маршрут параметр `locationid`. Маршруты основного приложения для местоположений находятся в файле `locations.js` в каталоге `/routes`. Следующий фрагмент кода демонстрирует несложные изменения, которые необходимо сделать для того, чтобы маршрут для подробностей о местоположении принимал параметр URL `locationid`:

```
router.get('/', ctrlLocations.homelist);
router.get('/location/:locationid', ctrlLocations.locationInfo); ←
router.get('/location/review/new', ctrlLocations.addReview);
```

Добавляем параметр `locationid`
в маршрут для отдельного
местоположения

Что ж, хорошо... Но откуда мы возьмем идентификаторы местоположений? При рассмотрении всего приложения, наилучшим местом для начала представляется домашняя страница, так как именно оттуда ведут ссылки на страницу Details (Подробности).

Когда API возвращает для домашней страницы массив местоположений, каждый объект местоположения содержит свой уникальный идентификатор. Весь объект уже передан представлению, так что не должно составить труда поменять представление домашней страницы, добавив этот идентификатор в качестве параметра URL.

На деле это совсем не сложно! Листинг 7.10 демонстрирует те небольшие изменения, которые необходимо выполнить в файле `locations-list.jade`, чтобы присоединить уникальный идентификатор каждого местоположения к ссылке, ведущей на страницу Details (Подробности).

Листинг 7.10. Модифицируем представление списка для добавления идентификатора местоположения в соответствующие ссылки

```
block content
  #banner.page-header
    .row
      .col-lg-6
        h1= pageHeader.title
        small &nbsp;&nbsp;&nbsp;#{pageHeader.strapline}
```

```

.row
.col-xs-12.col-sm-8
.error= message
.row.list-group
  each location in locations
    .col-xs-12.list-group-item
      h4
        a(href="/location/#{location._id}")= location.name ←
        small &nbsp;
          +outputRating(location.rating)
        span.badge.pull-right.badge-default= location.distance
        p.address= location.address
        p
          each facility in location.facilities
            span.label.label-warning= facility
            &nbsp;
.col-xs-12.col-sm-4
.p.lead= sidebar

```

При проходе цикла по каждому местоположению в массиве извлекаем уникальный идентификатор из объекта и добавляем его в конец href ссылки на страницу Details (Подробности)

Если бы все в жизни было так же просто. Теперь домашняя страница содержит уникальные ссылки для каждого из местоположений, и все они ведут на страницу Details (Подробности). Нам осталось только сделать так, чтобы они отображали правильные данные.

7.3.2. Разделение обязанностей: перемещаем визуализацию в поименованную функцию

Подобно тому как мы сделали для домашней страницы, переместим визуализацию страницы Details (Подробности) в отдельную поименованную функцию. Цель этого действия — отделить функциональность визуализации от вызовов API и обработки данных.

В листинге 7.11 показаны сокращенная версия новой функции `renderDetailPage` и обращение к ней из контроллера `locationInfo`.

Листинг 7.11. Перемещение содержимого контроллера `locationInfo` во внешнюю функцию

```

var renderDetailPage = function (req, res) {
  res.render('location-info', {
    title: 'Starcups',
    ...
  });
};
module.exports.locationInfo = function(req, res){
  renderDetailPage(req, res); ←
};

```

Создаем новую функцию `renderDetailPage` и перемещаем в нее все содержимое контроллера `locationInfo`

Вызываем новую функцию из контроллера, не забывая передать ей параметры `req` и `res`

Мы получили аккуратный, «чистый» контроллер, готовый к выполнению запросов к API.

7.3.3. Выполнение запросов к API с использованием уникального идентификатора из параметра URL

URL для вызова API должен содержать идентификатор местоположения. Этот идентификатор теперь имеется у страницы Details (Подробности) в виде параметра URL `locationid`, так что мы можем получить его значение с помощью модели `req.params` и добавить его к `path` в параметрах запроса. Запрос представляет собой запрос GET, так что значение `json` будет пустым объектом.

Зная все это, можем использовать созданный нами в контроллере домашней страницы паттерн для формирования и выполнения запроса к API. После ответа API воспользуемся функцией `renderDetailPage`. Все вместе это показано в листинге 7.12.

Листинг 7.12. Модифицируем контроллер `locationInfo` для обращения к API

```
module.exports.locationInfo = function(req, res){
  var requestOptions, path;
  path = "/api/locations/" + req.params.locationid; ←
  requestOptions = {
    url : apiOptions.server + path,
    method : "GET",
    json : {}
  };
  request(
    requestOptions,
    function(err, response, body) {
      renderDetailPage(req, res); ←
    }
  );
};
```

Получаем параметр `locationid` из URL и добавляем его к пути API

Задаем все необходимые для обращения к API параметры запроса

После ответа API вызываем функцию `renderDetailPage`

Когда вы выполните это, то увидите те же статические данные, что и раньше, так как мы пока что не передаем возвращенные из API данные в представление. Можете добавить какие-либо операторы вывода журнальных сообщений на консоль в функцию обратного вызова `request`, если хотите быстро взглянуть на возвращаемые данные.

Если все работает как надо и вас устраивает, приступим к передаче данных в представление.

7.3.4. Передаем данные из API в представление

Предположим пока, что API возвращает правильные данные, — мы займемся обработкой ошибок позднее. Этим данным необходима лишь небольшая предварительная обработка: координаты возвращаются из API в виде массива, но представлению они требуются в виде поименованных пар «ключ — значение» в объекте.

В листинге 7.13 показано, как можно сделать это в контексте оператора `request`, преобразуя данные из API до отправки их функции `renderDetailPage`.

Листинг 7.13. Предварительная обработка данных в контроллере

```
request(
  requestOptions,
  function(err, response, body) {
    var data = body;
    data.coords = {
      lng : body.coords[0],
      lat : body.coords[1]
    };
    renderDetailPage(req, res, data);
  }
);
```

Создаем копию возвращаемых данных в новой переменной

Делаем из свойства `coords` объект, присваивая `lng` и `lat`, извлеченные из ответа API значения

Отправляем преобразованные данные на визуализацию

Следующий логичный шаг — изменить функцию `renderDetailPage`, чтобы использовать эти данные вместо жестко зашитых. Чтобы все заработало, необходимо убедиться, что функция принимает эти данные в качестве параметра, а затем обновляет надлежащим образом переданные представлению значения. В листинге 7.14 необходимые изменения выделены полужирным шрифтом.

Листинг 7.14. Обновление `renderDetailPage` для получения доступа к данным и их использования с помощью API

```
var renderDetailPage = function (req, res, locDetail) {
  res.render('location-info', {
    title: locDetail.name,
    pageHeader: {title: locDetail.name},
    sidebar: {
      context: 'is on Loc8r because it has accessible wifi and space
to sit down with your laptop and get some work done.',
      callToAction: 'If you\'ve been and you like it - or if you
don\'t - please leave a review to help other people just like you.'
    },
    location: locDetail
  });
};
```

Добавляем новый параметр для данных в описание функции

Ссылаемся на конкретные элементы данных по мере необходимости

Передаем представлению весь объект данных `locDetail` со всеми подробностями

Нам удалось реализовать подход с отправкой полного объекта подобным образом благодаря тому, что в свое время наша модель была основана на том, что требовалось представлению и контроллеру. Если вы сейчас запустите приложение, то увидите страницу, заполненную извлеченными из БД данными. Соответствующий скриншот показан на рис. 7.6.

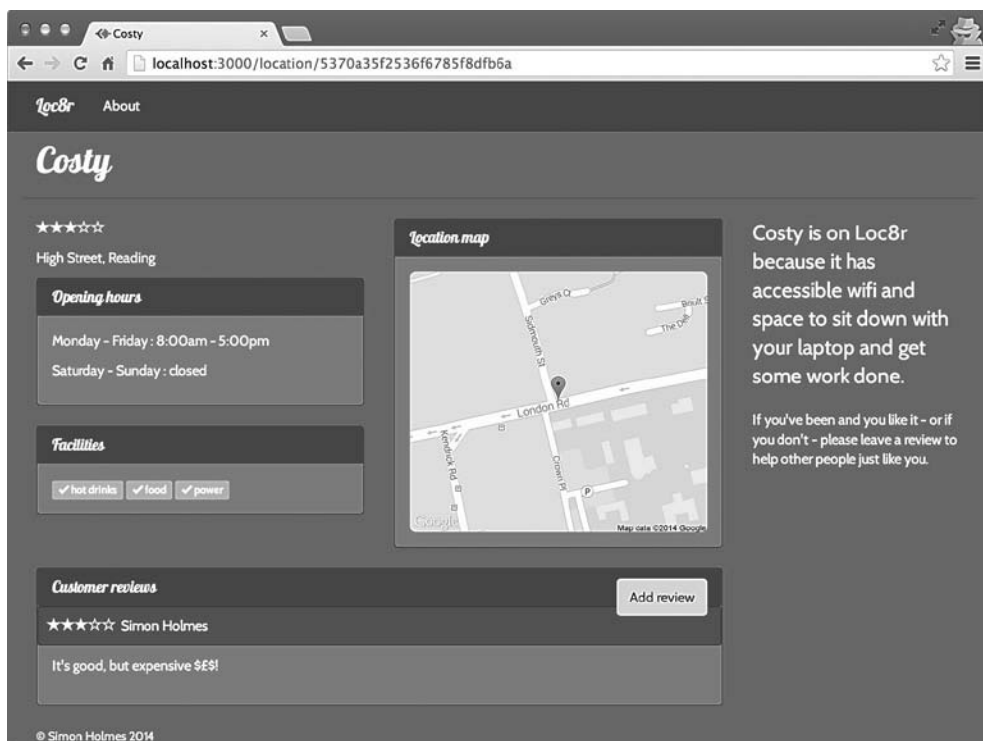


Рис. 7.6. Страница Details (Подробности) с данными, извлеченными из MongoDB через API

Наблюдательный читатель, конечно, заметит проблему со скриншотом, приведенным на рис. 7.6. У отзыва отсутствует соответствующая дата.

7.3.5. Отладка и исправление ошибок с представлением

Итак, с представлением имеется проблема: оно не выводит дату надлежащим образом. Может, нам не стоило быть слишком уж самоуверенными, говоря о том, что наша модель данных основывается на представлении и контроллере? Давайте посмотрим, что же происходит.

Заглянув сначала в файл `Jade info.jade` в каталоге `app_server/views`, мы можем найти строку, выводящую данный раздел:

```
small.reviewTimestamp #{review.timestamp}
```

Теперь необходимо посмотреть на схему, чтобы узнать, не поменяли ли мы что-то при описании модели. Схема для отзывов находится в `locations.js` в `app_api/models` и выглядит так, как показано в следующем фрагменте кода:

```
var reviewSchema = new mongoose.Schema({
  author: String,
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdAt: {type: Date, "default": Date.now}
});
```

О да, мы видим, что поменяли название для времени создания на `createdAt` — более подходящее название для этого пути.

Исправленный в соответствии с этим значением файл `Jade` выглядит следующим образом:

```
small.reviewTimestamp #{review.createdAt}
```

Выполнение этих изменений и обновление страницы дает нам рис. 7.7.



Рис. 7.7. Извлечение названия и даты непосредственно из возвращаемых данных; формат даты не очень удобен для пользователя

Получилось! В некотором роде. Дата теперь отображается, но не в удобном для чтения формате, как нам бы хотелось. Мы сумеем исправить это с помощью `Jade`.

Форматирование дат с помощью примесей `Jade`

Ранее, при настройке представлений, мы использовали примеси `Jade` для вывода оценочных звезд на основе имеющихся числовых оценок. Примеси в `Jade` напоминают функции — им можно передавать параметры при вызове, с их помощью выполнять при необходимости какой-либо код JavaScript и генерировать нужный вывод.

Форматирование дат — вещь, которая может пригодиться во многих местах, так что сделаем для этой задачи примесь. Наша примесь `outputRating` находится в файле `sharedHTMLfunctions.jade` в каталоге `app_server/views/_includes`. Добавим в этот файл новую примесь под названием `formatDate`.

В основном в этой примеси будем использовать JavaScript для преобразования даты из расширенного формата ISO в более удобочитаемый формат «*День Месяц Год*», например *24 июня 2014*. Объект даты ISO фактически приходит в виде строки, так что первое, что нам нужно будет сделать, — преобразовать его в объект даты JavaScript. После этого можно будет использовать различные методы работы с датами JavaScript для обращения к разным частям даты.

В листинге 7.15 продемонстрирована реализация этого в примеси. Напоминаю, что строки JavaScript в файле Jade должно предварять тире.

Листинг 7.15. Создание примеси Jade для форматирования дат

```

mixin formatDate(dateString)
  -var date = new Date(dateString);
  -var monthNames = [ "January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October", "November", "December" ];
  -var d = date.getDate();
  -var m = monthNames[date.getMonth()];
  -var y = date.getFullYear();
  -var output = d + ' ' + m + ' ' + y;
  =output

```

Преобразуем предоставленную дату из строки в объект даты

Используем методы JavaScript для работы с датами для извлечения и преобразования необходимых частей даты

Собираем части даты обратно в желаемом формате и выполняем визуализацию вывода

Создаем массив значений для названий месяцев

Такая примесь будет принимать на входе дату и преобразовывать ее для вывода в необходимом нам формате. Так как примесь будет выполнять визуализацию вывода, нам просто нужно вызвать ее из соответствующего места кода. Следующий код демонстрирует это вновь на основе тех же двух отдельных строк из всего шаблона:

```

span.reviewAuthor #{review.author.displayName}
small.reviewTimestamp
  +formatDate(review.createdOn)

```

Вызываем примесь в отдельной строке, передавая дату создания отзыва; не забываем убедиться в том, что для этой строки сделаны правильные отступы

Вызов примеси должен располагаться в отдельной строке, так что не забудьте позаботиться об отступах: дата должна быть вложена в тег `<small>`.

Теперь страница `Details` (Подробности) готова и выглядит так, как должна (рис. 7.8).

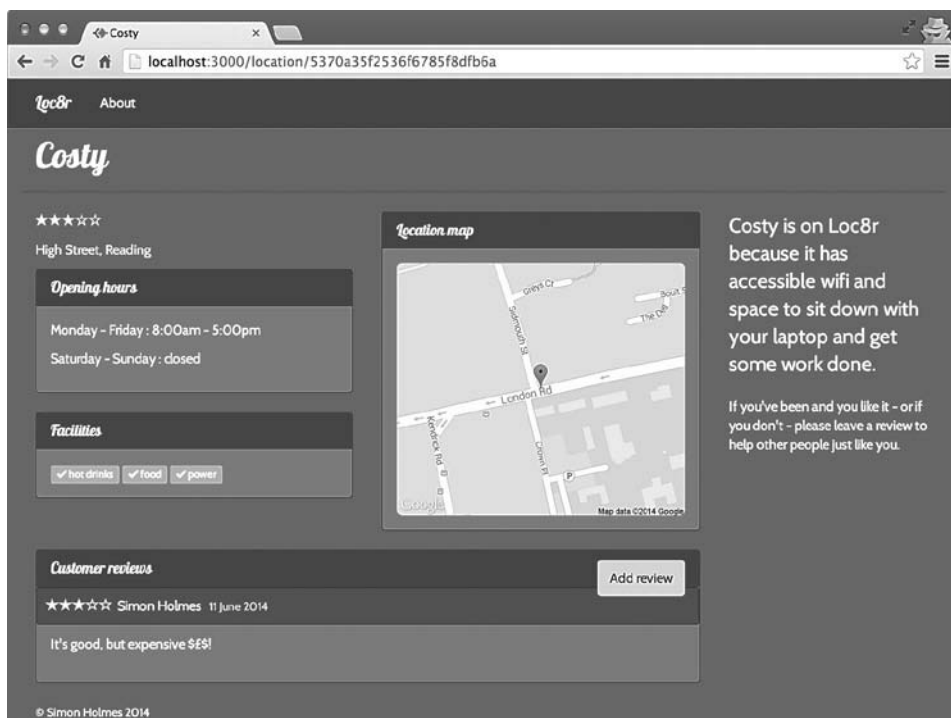


Рис. 7.8. Завершенный вариант страницы Details (Подробности). Идентификатор местоположения передается из URL в API, API извлекает данные и передает их обратно странице для форматирования и правильной визуализации

Отлично! Это именно то, чего мы и хотели. Если URL содержит имеющийся в БД идентификатор, страница отображается правильно. Но что произойдет, если идентификатор неправильный или отсутствует в БД?

7.3.6. Создание страниц ошибок в зависимости от статуса

Если идентификатор из URL отсутствует в БД, API вернет ошибку 404. Источник этой ошибки — URL в браузере, так что браузер тоже должен возвращать 404 — данные для идентификатора не найдены и, по существу, страница не может быть найдена.

С помощью уже рассмотренных в этой главе методик можно без труда перехватить возврат API кода состояния 404, используя проверку `response.statusCode` в обратном вызове `request`. Обращать его внутри обратного вызова нежелательно, так что просто передадим поток выполнения в новую функцию, которую сможем вызывать, `_showError`.

Перехват всех кодов ошибок

Мы можем поступить даже лучше, чем просто перехватывать ответ **404**, а именно инвертировать ситуацию и искать все ответы API, не являющиеся ответом об успешном выполнении **200**. Можно передавать код состояния функции `_showError`, позволяя ей определять, что делать дальше. Чтобы функция `_showError` могла все контролировать, передадим ей также объекты `req` и `res`.

Листинг 7.16 демонстрирует, как необходимо модифицировать обратный вызов для визуализации страницы **Details** (Подробности) для успешных обращений к API и маршрутизации всех остальных ошибок в универсальную функцию-перехватчик `_showError`.

Листинг 7.16. Перехватываем все ошибки, вызванные тем, что API вернул не код состояния **200**

```
request(
  requestOptions,
  function(err, response, body) {
    var data = body;
    if (response.statusCode === 200) {
      data.coords = {
        lng : body.coords[0],
        lat : body.coords[1]
      };
      renderDetailPage(req, res, data);
    } else {
      _showError(req, res, response.statusCode);
    }
  }
);
```

Проверяем на ответ об успехе от API

Продолжаем визуализацию страницы, если проверка успешна

Если проверка завершилась неудачей, передаем ошибку в функцию `_showError`

Отлично, теперь мы будем пытаться визуализировать страницу **Details** (Подробности), только если получили от API что-либо для отображения. А что мы должны делать с ошибками? Пока что просто будем отправлять пользователям сообщение о наличии проблемы.

Отображаем сообщения об ошибках

Мы не собираемся делать ничего из ряда вон выходящего, просто хотим дать пользователю знать, что что-то происходит, и указать, что именно. У нас уже есть подходящий для этого обобщенный шаблон Jade, он называется `generic-text.jade` и нуждается лишь в заголовке и каком-то содержимом.

Если хотите, можете создать индивидуальную страницу для каждого типа ошибки, но пока достаточно просто перехватить их и дать пользователю знать об этом.

Кроме того, необходимо сообщить об этом браузеру путем возврата соответствующего кода состояния при отображении страницы.

Листинг 7.17 демонстрирует вид функции `_showError`, принимающей параметр состояния, который, помимо того что, передается в качестве кода состояния ответа, используется для задания заголовка и контента страницы. Для страницы 404 имеется отдельное сообщение, а для любых других передаваемых ошибок — обобщенное сообщение.

Листинг 7.17. Создаем обрабатывающую ошибки функцию для отличных от 200 кодов состояния API

```
var _showError = function (req, res, status) {
  var title, content;
  if (status === 404) {
    title = "404, page not found";
    content = "Oh dear. Looks like we can't find this page. Sorry.";
  } else {
    title = status + ", something's gone wrong";
    content = "Something, somewhere, has gone just a little bit wrong.";
  }
  res.status(status);
  res.render('generic-text', {
    title : title,
    content : content
  });
};
```

Если передаваемый код состояния равен 404, задаем заголовок и контент страницы

В противном случае задаем обобщенное сообщение на случай перехвата любых других ошибок

Отправляем данные представлению для компиляции и отправки браузеру

Используем параметр состояния для задания состояния ответа

Эту функцию может при необходимости использовать любой контроллер. Помимо этого, функция сделана таким образом, что в нее, если понадобится, легко можно добавить новые отдельные сообщения об ошибках для конкретных кодов.

Для тестирования страницы ошибки 404 достаточно лишь немного изменить ID местоположения в URL. При этом вы увидите нечто напоминающее рис. 7.9.

Мы закончили работу со страницей `Details` (Подробности). Теперь можем успешно отображать любую информацию из БД для заданного местоположения, а также выдавать посетителю сообщение об ошибке 404, если местоположение найти не удалось.

Двигаясь дальше по пути пользователя, видим, что наша следующая и последняя задача — обеспечить возможность добавления отзывов.

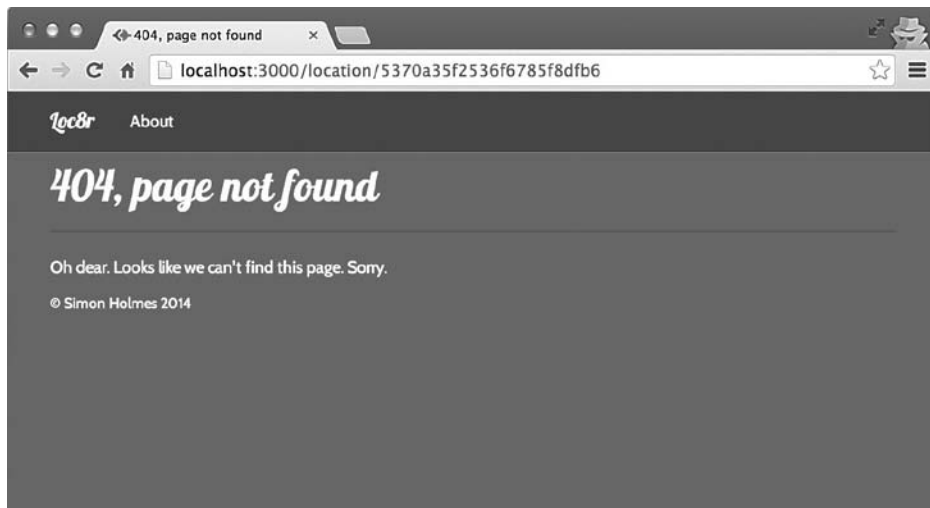


Рис. 7.9. Если идентификатор местоположения не найден API в базе данных, отображается страница ошибки 404

7.4. Добавление данных в БД через API: добавляем отзывы Loc8r

В этом разделе мы рассмотрим, как получать и обрабатывать переданные пользователем данные и отправлять их API. Отзывы добавляются в Loc8r путем нажатия кнопки Add Review (Добавить отзыв) на странице Details (Подробности) местоположения, заполнения формы и подтверждения ее отправки. По крайней мере, таков наш план. У нас уже имеются экраны для этого, но пока еще нет необходимой для них функциональности. Мы исправим это положение прямо сейчас.

Вот краткий список того, что нам понадобится сделать.

1. Обеспечить учет в форме отзыва местоположения, для которого предназначен отзыв.
2. Создать маршрут для формы для (выполнения метода) POST.
3. Отправить API новые данные отзыва.
4. Отобразить новый отзыв в соответствующем месте на странице Details (Подробности).

Обратите внимание на то, что на данном этапе разработки у нас еще нет метода аутентификации, а значит, отсутствует понятие «учетная запись пользователя».

7.4.1. Настройка маршрутизации и представлений

Первый пункт в нашем списке относится к обеспечению доступности идентификатора местоположения для страницы Add Review (Добавить отзыв) таким образом, чтобы можно было его использовать после отправки формы. В конце концов, это же уникальный идентификатор, который понадобится API для добавления отзыва.

Лучший подход для этого — включить идентификатор в URL аналогично тому, как мы поступили с самой страницей Details (Подробности).

Описываем два маршрута для отзывов

Включение идентификатора местоположения в URL будет означать изменение маршрута для страницы Add Review (Добавить отзыв), а именно добавление в него параметра `locationid`. Тем временем можно заняться вторым пунктом в списке и создать маршрут для формы для (выполнения метода) POST. В идеале это должен быть тот же самый путь, что и у формы отзывов, только связанный с другим методом запроса и другим контроллером.

Следующий фрагмент кода показывает, как мы могли бы модифицировать маршруты в файле `index.js` из каталога `/routes`:

```
router.get('/', ctrlLocations.homelist);
router.get('/location/:locationid', ctrlLocations.locationInfo);
router.get('/location/:locationid/reviews/new', ctrlLocations.addReview);
router.post('/location/:locationid/reviews/new', ctrlLocations.doAddReview);
```

Вставляем параметр `locationid`
в существующий маршрут
для формы отзывов

Создаем новый маршрут для того же URL,
но с применением метода POST
и ссылкой на другой контроллер

Все эти маршруты необходимы нам в данном разделе, но перезапуск приложения закончится сбоем, поскольку маршрут POST ссылается на не существующий пока контроллер. Исправить ситуацию можно добавлением функции-заглушки в файл контроллера. Добавьте следующий фрагмент кода в файл `locations.js` в каталоге `app_server/controllers` — и приложение опять будет запускаться без проблем:

```
module.exports.doAddReview = function(req, res){
};
```

Теперь приложение опять запускается, но если вы перейдете по ссылке на страницу Add Review (Добавить отзыв), то получите сообщение об ошибке. Конечно, нам же нужно обновить ссылку на страницу Add Review (Добавить отзыв) со страницы Details (Подробности).

Исправление представления для подробностей местоположения

Нам нужно добавить идентификатор местоположения, определенный для кнопки Add Review (Добавить отзыв) href на странице Details (Подробности). Контроллер для этой страницы передает полный объект данных, возвращаемый из API, который наряду с остальными данными содержит поле `_id`. При передаче представлению этот объект данных называется `location`.

Следующий фрагмент кода показывает одну строку из шаблона `location-info.jade` в каталоге `app_server/views`, демонстрируя, как добавить идентификатор местоположения к ссылке для кнопки Add Review (Добавить отзыв):

```
a.btn.btn-default.pull-right(href="/location/#{location._id}/reviews/new")
| Add review
```

После изменения и сохранения можно перейти по ссылке на форму отзыва для каждого конкретного местоположения. Тут осталось лишь несколько проблем: форма все еще ничего никуда не отправляет и название местоположения в настоящий момент жестко зашито в контроллере.

Модификация представления для формы отзыва

Далее нам необходимо убедиться, что форма отправляет данные на правильный URL. Сейчас при подтверждении отправки формы она просто выполняет запрос GET на URL `/location`, как показано в следующем фрагменте кода:

```
form.form-horizontal(action="/location", method="get", role="form")
```

Эта строка взята из файла `location-review-form.jade`, находящегося в каталоге `app_server/views`. Путь `/location` более не является допустимым в нашем приложении, кроме того, нам хотелось бы использовать запрос POST вместо запроса GET. На самом деле URL, по которому мы хотим отправлять форму, такой же, как и URL для Add Review (Добавить отзыв): `/location/:locationid/reviews/new`.

Простейший путь добиться этого — задать пустую строку в качестве действия формы, а в качестве метода — `post`, как показано в следующем фрагменте кода:

```
form.form-horizontal(action="", method="post", role="form")
```

Теперь при подтверждении отправки формы будет выполнен запрос POST по URL текущей страницы.

Создание поименованной функции для визуализации страницы Add Review

Как и для других страниц, переместим визуализацию этой страницы в отдельную поименованную функцию. Это позволит выполнить требуемое разделение обязанностей при кодировании и подготовит к последующим шагам.

Листинг 7.18 демонстрирует, как должен выглядеть код.

Листинг 7.18. Создание внешней функции для содержимого контроллера addReview

```
var renderReviewForm = function (req, res) {
  res.render('location-review-form', {
    title: 'Review Starcups on Loc8r',
    pageHeader: { title: 'Review Starcups' }
  });
};
/* Выполняем запрос GET для страницы 'Add review' */
module.exports.addReview = function(req, res){
  renderReviewForm(req, res);
};
```

Создаем новую функцию renderReviewForm и перенесим в нее содержимое контроллера addReview

Вызываем новую функцию из контроллера addReview, передавая те же параметры

Может показаться немного странным: создать поименованную функцию, чтобы единственным кодом в контроллере был ее вызов, но скоро это окажется очень полезным.

Получение подробностей о местоположении

Мы хотим отображать на странице Add Review (Добавить отзыв) название местоположения, чтобы пользователь не забывал контекст. Это значит, что нам понадобится снова обратиться к API, предоставив ему идентификатор местоположения, и отправить информацию обратно в контроллер и представление. Мы только что сделали это для страницы Details (Подробности), хотя и с другим контроллером. Если делать это умно, то много нового кода писать не придется.

Вместо того чтобы дублировать код с последующей поддержкой двух кусков, мы применим подход DRY¹ (Don't repeat yourself — «Не повторяйся»). Обе страницы, Details (Подробности) и Add Review (Добавить отзыв), должны обращаться к API для получения информации о местоположении, а затем выполнять с ней какие-то действия. Так почему бы не создать для этого новую функцию? Большая часть кода уже имеется в контроллере locationInfo, так что необходимо просто поменять способ вызова итоговой функции. Вместо того чтобы явно вызывать функцию renderDetailPage, мы выполним обратный вызов.

¹ См. https://ru.wikipedia.org/wiki/Don't_repeat_yourself. — *Примеч. пер.*

Таким образом, у нас будет новая функция `getLocationInfo`, выполняющая запрос API. В случае удачного запроса затем должна быть вызвана переданная функция обратного вызова. Контроллер `locationInfo` будет теперь вызывать эту функцию, передавая функцию обратного вызова, просто обращаясь к функции `renderDetailPage`. Контроллер `addReview` также сможет вызывать новую функцию, передавая ей в обратном вызове функцию `renderReviewForm`.

Мы получаем одну функцию для выполнения обращений к API, выдающую различные результаты в зависимости от переданной ей функции обратного вызова. Листинг 7.19 демонстрирует это.

Листинг 7.19. Создаем новую, пригодную для многократного использования функцию для получения информации о местоположении

```
var getLocationInfo = function (req, res,
                                callback) {
    var requestOptions, path;
    path = "/api/locations/" + req.params.locationid;
    requestOptions = {
        url : apiOptions.server + path,
        method : "GET",
        json : {}
    };
    request(
        requestOptions,
        function(err, response, body) {
            var data = body;
            if (response.statusCode === 200) {
                data.coords = {
                    lng : body.coords[0],
                    lat : body.coords[1]
                };
                callback(req, res, data);
            } else {
                _showError(req, res, response.statusCode);
            }
        }
    );
};

module.exports.locationInfo = function(req, res){
    getLocationInfo(req, res, function(req, res, responseData) {
        renderDetailPage(req, res, responseData);
    });
};

module.exports.addReview = function(req, res){
```

Новая функция `getLocationInfo` принимает обратный вызов в качестве третьего параметра и содержит весь код, ранее находившийся в контроллере `locationInfo`

В случае сообщения API об успехе вызываем обратный вызов вместо поименованной функции

Вызываем в контроллере `locationInfo` функцию `getLocationInfo`, передавая функцию обратного вызова, которая по завершении вызовет функцию `renderDetailPage`


```
getLocationInfo(req, res, function(req, res, responseData) {
  renderReviewForm(req, res, responseData);
});
```

Вызываем также getLocationInfo из контроллера addReview, но на этот раз передаем в обратном вызове renderReviewForm

СОВЕТ

Если этот метод создания собственных обработчиков обратных вызовов для вас нов или непонятен, загляните в выложенное в Интернет приложение, в частности в раздел «Обратные вызовы JavaScript».

Мы использовали подход **DRY** для решения задачи. Проще всего было бы скопировать код API и вставить его из одного контроллера в другой, что, будем откровенны, совершенно нормально, если вы понимаете свой код и знаете, что нужно сделать, чтобы он заработал. Но, когда вы видите два фрагмента кода, делающие практически одно и то же, всегда стремитесь применить подход **DRY** — это сделает ваш код аккуратнее и удобнее в сопровождении.

Отображение подробностей о местоположении

Мы кое-что забыли. Функция для визуализации формы все еще содержит жестко зашитые данные, вместо того чтобы использовать данные из API. Небольшие корректировки в функции изменят эту ситуацию, как проиллюстрировано в следующем листинге.

Листинг 7.20. Удаляем жестко зашитые данные из функции renderReviewForm

```
var renderReviewForm = function (req, res, locDetail) {
  res.render('location-review-form', {
    title: 'Review ' + locDetail.name + ' on Loc8r',
    pageHeader: { title: 'Review ' + locDetail.name }
  });
};
```

Модифицируем функцию renderReviewForm так, чтобы она принимала новый параметр с данными

Заменяем жестко зашитые данные ссылками на данные

Теперь страница Add Review (Добавить отзыв) опять выглядит нормально, отображая правильное название на основе найденного в URL идентификатора (рис. 7.10).

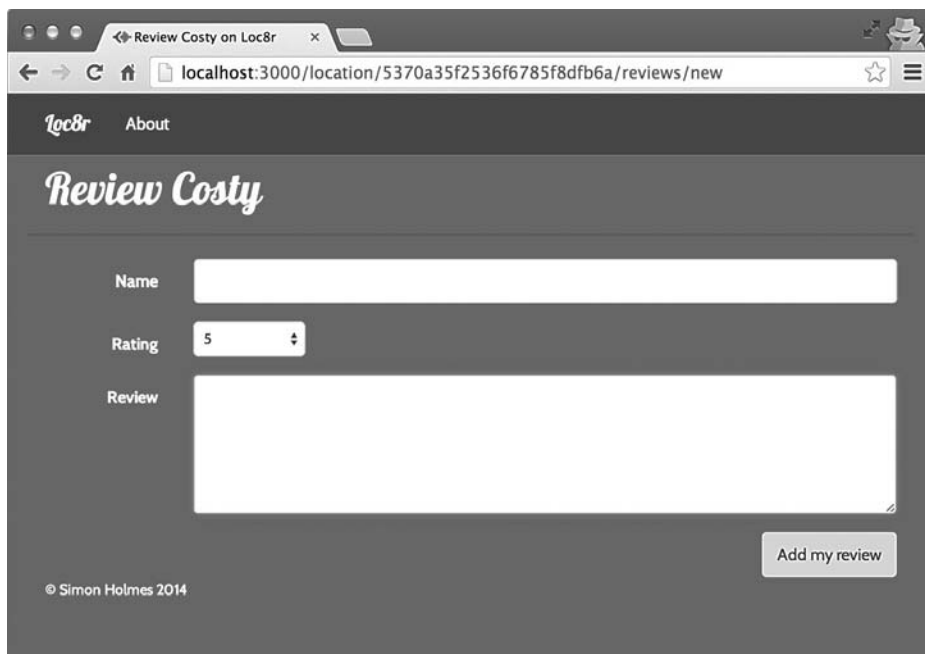


Рис. 7.10. Страница Add Review (Добавить отзыв) извлекает название местоположения посредством API, исходя из содержащегося в URL идентификатора

7.4.2. Отправка данных отзывов API методом POST

Страница Add Review (Добавить отзыв) уже настроена и готова к работе. У нас даже готовы маршрут и контроллер для действия POST. Однако контроллер `doAddReview` пока что представляет собой всего лишь пустую заглушку.

План действий для этого контроллера следующий.

1. Получить идентификатор местоположения из URL, чтобы сформировать URL запроса API.
2. Получить отправленные в форме данные и скомпоновать их для API.
3. Выполнить обращение к API.
4. Отобразить новый отзыв в соответствующем месте, если необходимо.
5. Отобразить страницу ошибки в случае неудачи.

Единственная часть этого плана, о которой мы еще не говорили, — передача данных API: до сих пор мы просто передавали пустой объект JSON, чтобы гарантировать JSON-форматирование ответа. Теперь мы хотим взять данные формы и передать их API в ожидаемом им формате. У нас имеются три поля в форме и три

ссылки, ожидаемые API. Все, что нужно сделать, — связать одно с другим. Поля формы и пути модели показаны в табл. 7.3.

Таблица 7.3. Соответствие названий полей формы и ожидаемых API путей модели

Поле формы	Ссылка API
name	author
rating	rating
review	reviewText

Превратить это соответствие в объект JavaScript не составляет труда. Нужно просто создать новый объект, содержащий ожидаемые API имена переменных, и воспользоваться моделью `req.body` для получения значений из отправленной формы. Следующий фрагмент кода показывает этот процесс отдельно, мы поместим его в контроллер буквально через минуту:

```
var postdata = {
  author: req.body.name,
  rating: parseInt(req.body.rating, 10),
  reviewText: req.body.review
};
```

Теперь, увидев, как это работает, мы можем добавить код в используемый нами для этих контроллеров API стандартный паттерн и создать контроллер `doAddReview`. Обратите внимание на то, что возвращаемый API в случае успешной операции POST код состояния равен 201, а не 200, который мы использовали ранее для запросов GET. В листинге 7.21 показан использующий все описанное ранее контроллер `doAddReview`.

Листинг 7.21. Используемый для отправки API данных отзывает контроллер `doAddReview`

```
module.exports.doAddReview = function(req, res){
  var requestOptions, path, locationid, postdata;
  locationid = req.params.locationid;
  path = "/api/locations/" +
    locationid + '/reviews';
  postdata = {
    author: req.body.name,
    rating: parseInt(req.body.rating, 10),
    reviewText: req.body.review
  };
  requestOptions = {
    url : apiOptions.server + path,
    method : "POST",
    json : postdata
  };
  request(
    ←————— Выполняем запрос
```

Получаем идентификатор местоположения из URL, чтобы сформировать URL API

Создаем объект данных для отправки API с помощью отправленных пользователем данных формы

```

requestOptions,
function(err, response, body) {
  if (response.statusCode === 201) {
    res.redirect('/location/' + locationid);
  } else {
    _showError(req, res, response.statusCode);
  }
}
);
};

```

Перенаправляем на страницу Details (Подробности) при успешном добавлении отзыва или демонстрируем страницу ошибки, если API вернул ошибку

Теперь мы можем создать отзыв и отправить его, а затем увидеть на странице Details (Подробности) (рис. 7.11).

Теперь, когда все работает, рассмотрим вкратце добавление проверок формы.

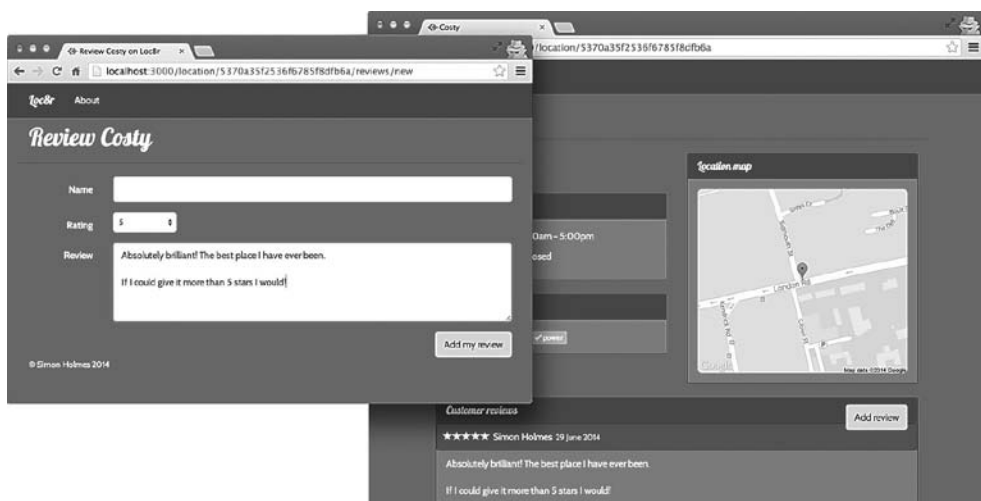


Рис. 7.11. После заполнения формы отзыва и отправки ее содержимого этот отзыв демонстрируется на своем месте на странице Details (Подробности)

7.5. Защита целостности данных с помощью их проверок

Всегда, когда приложение получает данные извне и вносит их в БД, необходимо убедиться, что данные полны и точны — настолько, насколько это возможно или насколько имеет смысл. Например, если кто-то добавляет адрес электронной почты, вы должны проверить соответствие его формата формату адреса электронной почты, но проверить программными средствами, что это *существующий* адрес электронной почты, невозможно.

В данном разделе мы рассмотрим способы добавления в приложение проверок, которые должны предотвратить отправку пользователями пустых отзывов. Существует три места, куда можно добавить проверки:

- ❑ на уровне схемы (с помощью Mongoose) — до сохранения данных;
- ❑ на уровне приложения — до отправки данных API;
- ❑ на стороне клиента — до отправки формы.

Рассмотрим каждое из них по очереди, добавляя на каждом этапе какие-либо проверки.

7.5.1. Проверка на уровне схемы с помощью Mongoose

Проверка данных до их сохранения — вероятно, самый важный этап. Это итоговый шаг, последняя возможность удостовериться в том, что все так, как должно быть. Этот этап особенно важен, если данные доступны через API: если мы не можем контролировать все использующие API приложения, то не можем и гарантировать качество получаемых данных. Поэтому важно убедиться в допустимости данных до их сохранения.

Модификация схемы

При первой настройке схемы в главе 5 мы рассматривали выполнение некоторых проверок в Mongoose. Мы указали обязательность пути `rating`, но хотелось бы, чтобы пути `authordisplayName` и `reviewText` тоже были обязательными. В отсутствие любого из этих полей отзыв не имеет смысла. Добавить их в схему несложно, все будет выглядеть так, как показано в листинге 7.22 (схема находится в файле `locations.js` в каталоге `app_api/model`).

Листинг 7.22. Добавление возможности проверок отзывов на уровне схемы

```
var reviewSchema = new mongoose.Schema({
  author: {type: String, required: true}
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: {type: String, required: true},
  createdAt: {type: Date, "default": Date.now}
});
```

←
 createdAt не должно быть обязательным,
 поскольку Mongoose автоматически
 заполняет его при создании нового отзыва

Делаем все эти пути
 обязательными полями,
 так как в отсутствие
 любого из них отзыв
 не имеет смысла

Теперь уже не получится сохранить отзыв без хотя бы какого-нибудь текста отзыва. Можно попробовать это сделать, но мы увидим ошибку, показанную на рис. 7.12.

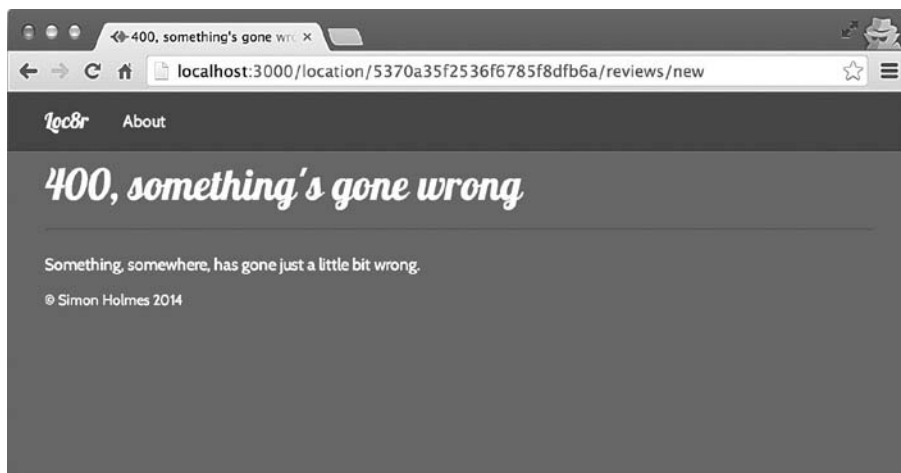


Рис. 7.12. Сообщение об ошибке, демонстрируемое при попытке сохранить отзыв без текста отзыва, когда схема декларирует его обязательность

С одной стороны, хорошо, что мы таким образом защищаем БД, но с другой — для пользователя это не лучший опыт взаимодействия. Лучше попытаться перехватить эту ошибку и позволить посетителю попробовать еще раз.

Перехват ошибок проверки Mongoose

Если вы попытаетесь сохранить документ, в котором отсутствуют одно или несколько обязательных полей, Mongoose вернет ошибку. Для этого не требуется обращения к БД, так как Mongoose сам хранит схему и знает, какие поля обязательны, а какие — нет. Следующий фрагмент кода демонстрирует пример подобного сообщения об ошибке:

```
{
  message: 'Validation failed',
  name: 'ValidationError',
  errors: {
    'reviews.1.reviewText': {
      message: 'Path `reviewText` is required.',
      name: 'ValidatorError',
      path: 'reviewText',
      type: 'required',
      value: ''
    }
  }
}
```

В потоке выполнения приложения это происходит в обратном вызове из функции `save`. Если мы посмотрим на команду `save` в функции `doAddReview` (в файле `locations.js` в каталоге `app_api/controllers`), то увидим, где возникает ошибка и где

мы задаем код состояния **400**. Следующий фрагмент кода демонстрирует это, включая временное сообщение журнала консоли для отображения ошибки в терминале:

```
location.save(function(err, location) {
  var thisReview;
  if (err) {
    console.log(err);
    sendJSONresponse(res, 400, err);
  } else {
    updateAverageRating(location._id);
    thisReview = location.reviews[location.reviews.length - 1];
    sendJSONresponse(res, 201, thisReview);
  }
});
```

Ошибки проверки Mongoose возвращаются через объект ошибки вслед за попыткой сохранения

Для возврата этого сообщения в виде тела ответа вместе с кодом состояния **400** API использует функцию `sendJSONresponse`. Таким образом, мы можем найти эту информацию в своем приложении, обратившись к телу ответа при возврате API кода состояния **400**.

Сделать это можно в `app_server`, точнее, в функции `doAddReview` в файле `controllers/locations.js`. При перехвате ошибки проверки нам хотелось бы дать пользователю возможность попробовать еще раз путем переадресации на страницу `Add Review` (Добавить отзыв). А чтобы страница знала, что попытка была предпринята, можно передать соответствующий флаг в строке запроса.

В листинге 7.23 показан код для этого в соответствующем месте, внутри обратного вызова оператора запроса функции `doAddReview`.

Листинг 7.23. Перехват ошибок проверки, возвращаемых API

```
request(
  requestOptions,
  function(err, response, body) {
    if (response.statusCode === 201) {
      res.redirect('/location/' + locationid);
    } else if (response.statusCode === 400 && body.name && body.name ===
      "ValidationError" ) {
      res.redirect('/location/' + locationid + '/reviews/new?err=val');
    } else {
      console.log(body);
      _showError(req, res, response.statusCode);
    }
  }
);
```

Добавляем проверки на то, что код состояния — 400, что у тела имеется имя и что это имя — `ValidationError`

Если да, то перенаправляем на форму отзыва, передавая флаг ошибки в строке запроса

Итак, теперь, когда API возвращает ошибку проверки, мы можем ее перехватить и отправить пользователя обратно на форму, чтобы он мог предпринять еще одну

попытку. Передача значения в строке запроса означает, что мы можем выполнить его поиск в контроллере, отображающем форму отзыва, и отправить представлению сообщение, чтобы известить пользователя о проблеме.

Отображение сообщений об ошибках в браузере

Для отображения сообщения об ошибке в представлении необходимо при появлении параметра `err` в строке запроса отправить представлению переменную. За передачу переменных в представление отвечает функция `renderReviewForm`. При вызове она также передает объект `req`, содержащий объект `query`, что упрощает передачу параметра `err` при его наличии. В листинге 7.24 выделены необходимые для этого несложные изменения.

Листинг 7.24. Модификация контроллера для передачи строки ошибки представлению

```
var renderReviewForm = function (req, res, locDetail) {
  res.render('location-review-form', {
    title: 'Review ' + locDetail.name + ' on Loc8r',
    pageHeader: { title: 'Review ' + locDetail.name },
    error: req.query.err ← Отправляем представлению новую переменную ошибки,
  });                               передавая в ней параметр запроса, если таковой имеется
};
```

Объект `query` всегда является частью объекта `req` независимо от наличия в нем какого-либо содержимого. Поэтому нет необходимости перехватывать это и проверять его наличие — если параметр `err` не найден, будет просто возвращено `undefined`.

Все, что осталось, — сделать что-то с информацией в представлении, известив пользователя о сути проблемы. Если возникла ошибка проверки, мы будем показывать пользователю сообщение поверх формы. Чтобы оформить его и разместить на странице, воспользуемся компонентом оповещения Bootstrap — он представляет собой просто `div` с некоторыми связанными классами и атрибутами. Следующий фрагмент кода демонстрирует две строки, которые необходимо добавить в соответствующем месте представления `location-review-form`:

```
form.form-horizontal(action="", method="post", role="form")
- if (error == "val")
  .alert.alert-danger(role="alert") All fields required, please try again
```

Так что теперь при возврате API ошибки проверки мы будем ее перехватывать и отображать пользователю сообщение (рис. 7.13).

Такая разновидность проверки на уровне API важна и обычно становится хорошей отправной точкой, поскольку всегда защищает БД от противоречивых или неполных данных вне зависимости от их происхождения. Но для конечных пользователей опыт взаимодействия с приложением не всегда оказывается наилучшим — им приходится отправлять форму, и приложение обращается к API, прежде чем страница перезагружается с ошибкой. Безусловно, тут есть что улучшить,

и первым шагом будет выполнение некоторых проверок на уровне приложения до отправки данных API.

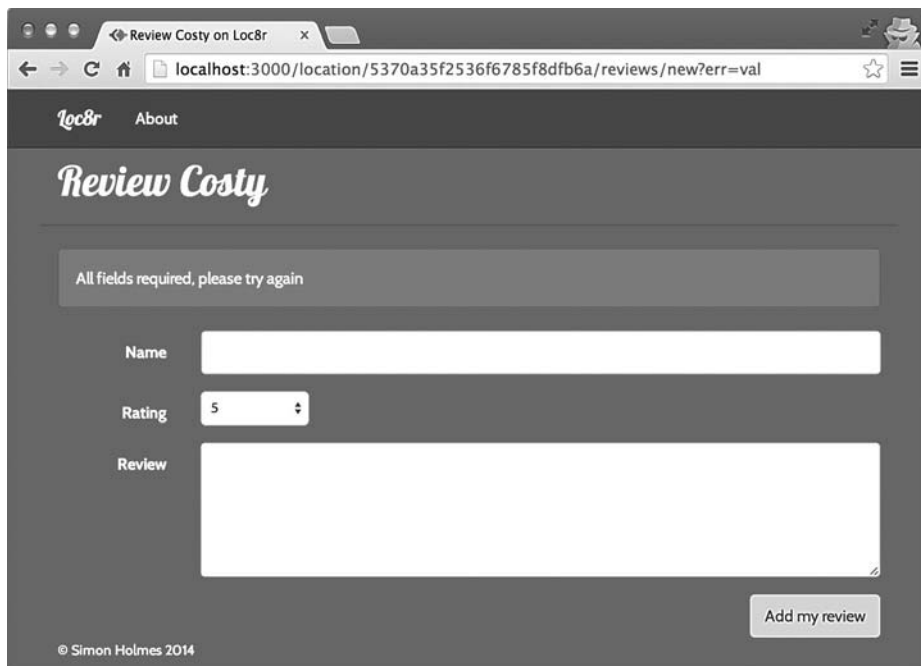


Рис. 7.13. В браузере отображается сообщение об ошибке проверки — конечный результат процесса, запущенного перехватом Mongoose ошибки и ее возвратом

7.5.2. Проверка на уровне приложения с помощью Node и Express

Проверка на уровне схемы — последний оборонительный рубеж перед БД. Приложение не должно всецело полагаться на нее, лучше попытаться избежать лишних обращений к API, снизив таким образом накладные расходы и ускорив работу с точки зрения пользователя. Один из способов сделать это — добавить проверку на уровне приложения, проверяя отправленные данные до пересылки их API.

В нашем приложении необходимые для отзывов проверки довольно просты. Можно добавить несколько простых проверок, чтобы убедиться в наличии значения у каждого из полей. Если такая проверка не пройдет, мы перенаправим пользователя обратно к форме, добавив в строку запроса тот же флаг ошибки, что и раньше. Если же проверки окажутся успешными, то можно дать возможность контроллеру продолжить выполнение, перейдя к методу запроса. Листинг 7.25 демонстрирует дополнения, которые необходимо сделать в контроллере `doAddReview` в файле `locations.js` из каталога `app_server/controllers`.

Листинг 7.25. Добавляем простые проверки в контроллер Express

```

module.exports.doAddReview = function(req, res){
  var requestOptions, path, locationid, postdata;
  locationid = req.params.locationid;
  path = "/api/locations/" + locationid + '/reviews';
  postdata = {
    author: req.body.name,
    rating: parseInt(req.body.rating, 10),
    reviewText: req.body.review
  };
  requestOptions = {
    url : apiOptions.server + path,
    method : "POST",
    json : postdata
  };
  if (!postdata.author || !postdata.rating || !postdata.reviewText) {
    res.redirect('/location/' + locationid + '/reviews/new?err=val');
  } else {
    request(
      requestOptions,
      function(err, response, body) {
        if (response.statusCode === 201) {
          res.redirect('/location/' + locationid);
        } else if (response.statusCode === 400 && body.name && body.name ===
          "ValidationError" ) {
          res.redirect('/location/' + locationid + '/reviews/new?err=val');
        } else {
          console.log(body);
          _showError(req, res, response.statusCode);
        }
      }
    );
  }
};

```

Если любое из трех обязательных полей ложно, перенаправляем на страницу Add Review (Добавить отзыв), добавляя используемую для отображения сообщения об ошибке строку запроса

В противном случае продолжаем как и прежде

Результат этого будет таким же, как и раньше: если текст отзыва отсутствует, пользователю будет показано сообщение об ошибке на странице Add Review (Добавить отзыв). Пользователю неизвестно, что мы больше не отправляем данные API, но это на один вызов меньше, а значит, приложение работает быстрее. Но мы можем еще ускорить его благодаря третьему слою проверок — браузерной проверке.

7.5.3. Проверка в браузере с помощью jQuery

Аналогично тому, как проверка на уровне приложения ускоряет работу за счет того, что не требуется выполнять вызов API, проверка на стороне клиента в браузере может ускорить работу за счет перехвата ошибки до отправки формы приложению, что исключает еще один вызов. При перехвате ошибки на этом этапе пользователь останется на той же странице.

Чтобы выполнить код JavaScript в браузере, необходимо поместить его в каталог `public` в приложении. Express интерпретирует содержимое этого каталога как статические файлы, загружаемые в браузер вместо выполнения на сервере. Если у вас еще нет каталога `javascripts` в каталоге `public`, создайте его сейчас. В этом новом каталоге создайте новый файл `validation.js`.

Написание функции проверок jQuery

В этом новом файле `validation.js` поместим функцию jQuery, которая будет делать следующее:

- прослушивать на предмет события подтверждения отправки формы отзыва;
- проверять, у всех ли обязательных полей заданы значения;
- если одно из полей пусто, отображать сообщение об ошибке аналогично тому, как это происходило в остальных типах проверок, и блокировать отправку формы.

Не будем углубляться тут в семантику jQuery, предполагая, что вы хотя бы немного знакомы с ней или какой-нибудь аналогичной библиотекой. В листинге 7.26 показан код для выполнения этих действий.

Листинг 7.26. Создаем функцию проверки формы jQuery

```
$('#addReview').submit(function (e) {
  $('#.alert.alert-danger').hide();
  if (!$('input#name').val() || !$('select#rating').val() ||
    !$('textarea#review').val()) {
    if ($('#.alert.alert-danger').length) {
      $('#.alert.alert-danger').show();
    } else {
      $(this).prepend('<div role="alert" class="alert
        alert-danger">All fields required, please try again</div>');
    }
    return false;
  }
});
```

Прослушиваем на предмет события, подтверждающего отправку формы отзыва

Проверяем, отсутствуют ли какие-то значения

Блокируем отправку формы при отсутствии значения

Отображаем сообщение об ошибке или вставляем его на странице при отсутствии какого-либо из значений

Чтобы это работало, необходимо убедиться в настройке для формы идентификатора `addReview` таким образом, чтобы jQuery мог прослушивать на предмет наступления нужного события. Нам также необходимо добавить этот сценарий на страницу, чтобы браузер мог его выполнить.

Добавление jQuery на страницу

Включим данный файл jQuery в конце тела, рядом с другими клиентскими JavaScript-файлами. Это задается в представлении `layout.jade` в каталоге `app_server/views`, в самом низу. Добавьте ниже всех строк новую строку, указывающую на новый файл, как показано в следующем фрагменте кода:

```
script(src='/bootstrap/js/bootstrap.min.js')
script(src='/javascripts/validation.js')
```

Вот и все. Теперь форма будет проверяться в браузере без отправки куда-либо данных и без перезагрузки страницы и соответствующих обращений к серверу.

СОВЕТ

Вам может показаться, что проверки на стороне клиента более чем достаточно, но остальные виды проверок жизненно важны для надежной работы приложения. В браузере может быть отключен JavaScript, из-за чего эта проверка не сможет быть выполнена, или проверка может быть обойдена, а данные — отправлены напрямую, или по URL действия формы, или конечной точке API.

7.6. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Использование модуля `request` для выполнения обращений к API из Express.
- ❑ Выполнение POST- и GET-запросов к конечным точкам API.
- ❑ Разделение обязанностей путем отделения функций визуализации от логики запросов API.
- ❑ Использование простого паттерна для логики API в каждом контроллере.
- ❑ Использование кодов состояния ответов API для проверки успешности выполнения запроса.
- ❑ Использование проверок данных в трех местах архитектуры, а также когда и почему нужно использовать каждую из них.

В главе 8 мы введем в наш стек Angular и начнем экспериментировать с некоторыми интерактивными компонентами клиентской части поверх приложения Express.

ЧАСТЬ III

Добавление динамической клиентской части с помощью Angular

AngularJS — одна из самых захватывающих и быстро развивающихся технологий нашего времени, а также ключевая часть стека MEAN. Мы уже немало сделали с Express — серверным фреймворком. AngularJS — фреймворк клиентской части, который дает нам возможность создавать целые работающие в браузере приложения.

В главе 8 мы познакомимся с AngularJS, узнаем, из-за чего вся эта шумиха вокруг него, и разберемся в семантике его синтаксиса и связанном с ним жаргоне. Кривая изучения Angular может (но не обязательно) оказаться довольно крутой. Когда мы займемся Angular в главе 8, то рассмотрим, как использовать его для создания компонентов для существующей веб-страницы, включая обращения к API REST для получения данных.

Главы 9 и 10 посвящены использованию Angular для создания одностраничного приложения. Основываясь на полученных в главе 8 знаниях, мы переделаем Loc8r в SPA. По ходу дела обратим внимание на рекомендуемые решения, научимся создавать модульные приложения, удобные в сопровождении благодаря допускающим повторное использование компонентам. К концу части III мы получим полностью работающее одностраничное приложение, взаимодействующее с API REST для создания и чтения данных.

Глава 8

Добавление компонентов Angular в приложение Express

В этой главе:

- ❑ знакомимся с Angular;
- ❑ добавляем Angular на существующую страницу;
- ❑ фильтруем списки данных;
- ❑ используем Angular для чтения данных;
- ❑ изучаем жаргон Angular — контроллеры, область видимости, фильтры, директивы, сервисы.

Наконец-то! Пришло время посмотреть на последнюю часть стека MEAN — Angular! Мы рассмотрим, как включить Angular в приложение Express и разработать несколько компонентов для улучшения уже существующего приложения `Loc8r`. По ходу дела обсудим семантику и ключевые специальные термины мира Angular. В отличие от других частей стека MEAN, Angular упрям в том смысле, что при работе с ним все должно выполняться строго определенным образом.

Рисунок 8.1 иллюстрирует наше местонахождение в общем плане, добавляя Angular в клиентскую часть существующего приложения Express.

Применяемый в этой главе подход — то, как бы вы поступили, если бы захотели улучшить страницу, проект или приложение путем добавления Angular. Создание приложения, целиком основанного на Angular, будет рассмотрено в главах 9 и 10. Мы воспользуемся этой главой и для того, чтобы немного познакомиться с Angular.

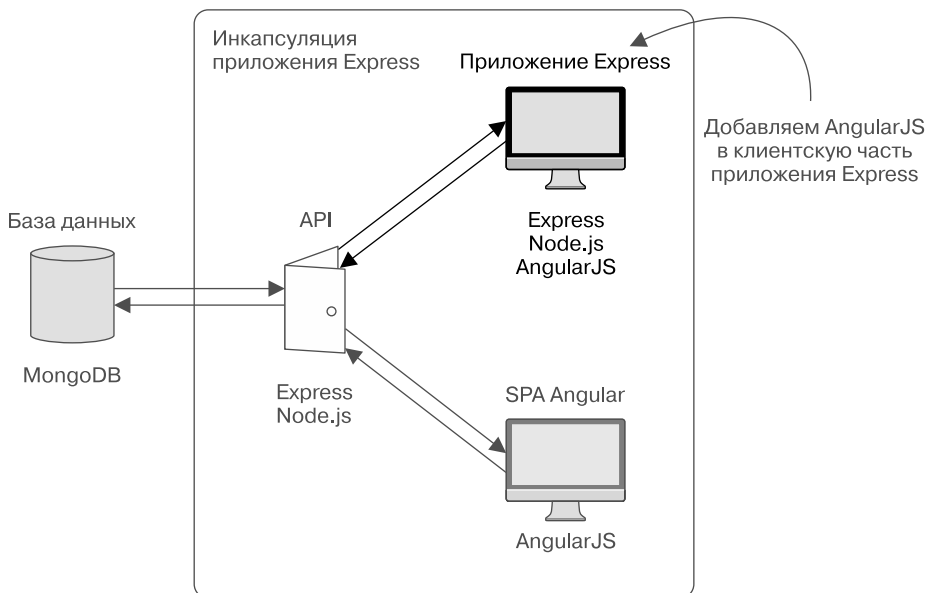


Рис. 8.1. Глава 8 посвящена добавлению Angular в клиентскую часть существующего приложения Express

В этой главе сосредоточимся на улучшении пользовательского опыта взаимодействия с домашней страницей. Мы воспользуемся Angular для загрузки данных в список местоположений и добавим для пользователей возможность поиска и фильтрации списка. Для начала будем использовать статические данные, чтобы научиться работать с Angular, прежде чем начнем получать данные из API и воспользуемся API местоположений HTML5 для получения географического расположения пользователя.

8.1. Настройка и запуск Angular

Angular — второй фреймворк JavaScript в стеке MEAN, первый — Express. Express, как мы видели, располагается на сервере, в то время как Angular — на стороне клиента, в браузере. Подобно Express, Angular позволяет разделять обязанности, работая с представлениями, данными и логикой в различных областях. Такой подход вполне в стиле MVC, но Angular считается фреймворком MVW, где **W** означает «все, что вам подходит» (whatever works for you). Иногда это могут быть контроллеры, или представление-модель, или сервисы. Зависит от того, что вы делаете в конкретный момент.

Теперь, когда вы кое-что узнали об Angular, перейдем к самому интересному. Начнем с экспериментов с некоторыми основами Angular, познакомимся с понятием двусторонней привязки данных (two-way data binding), включая представления, модели, представления-модели и контроллеры.

Angular — клиентский фреймворк, так что он не требует особой установки. Процесс описан в приложении А, но по сути он представляет собой просто скачивание последней стабильной версии с сайта <http://angularjs.org/>.

8.1.1. Открываем для себя двустороннюю привязку данных

Так что же означает двусторонняя привязка данных? Ранее, в главе 1, мы обсуждали вкратце, как взаимосвязаны в Angular модель данных и представление и то, что оба они — динамические. Это значит, что внесение изменений в представление вызывает изменение модели и, наоборот, внесение изменений в модель вызывает изменение представления. Не забывайте, что мы тут не говорим о какой-либо БД — все происходит в браузере. Подобную двустороннюю привязку иллюстрирует рис. 8.2.

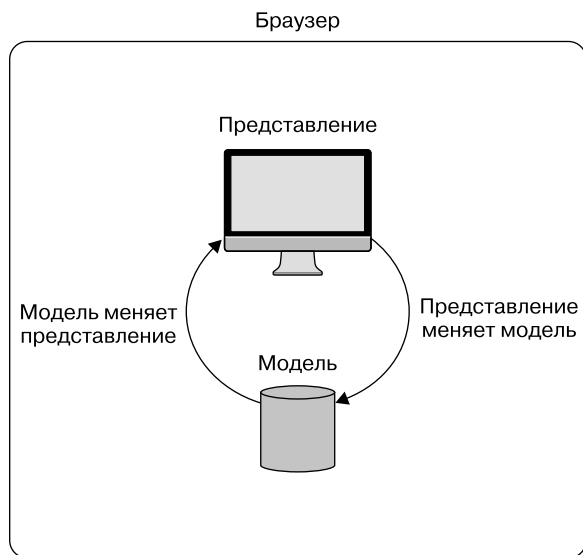


Рис. 8.2. В Angular представление и модель связаны воедино с помощью двусторонней привязки данных, причем все происходит в браузере

Рисунки могут быть очень красочными, но примеры все равно лучше. Так что посмотрим на наш первый кусочек кода Angular.

Начинаем с HTML-страницы

Пусть у нас имеется очень простая HTML-страница с полем для ввода и местом для отображения вводимых данных. В следующем фрагменте кода есть поле `input` и тег `<h1>`, и мы хотели бы сразу же отображать вводимый в поле ввода текст после слова `Hello` в `<h1>`:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular binding test</title>
</head>
<body>
  <input />
  <h1>Hello </h1>
</body>
</html>
```

Если у вас есть опыт работы с JavaScript или jQuery, то вы, думая о возможной реализации этого, вероятно, представляете себе написание кода для привязки к событиям нажатий клавиш в поле `input` с последующей вставкой полученного в тег `<h1>`. Это вполне выполнимо и не так уже сложно. Но с помощью Angular можно сделать нечто подобное вообще без написания какого-либо кода JavaScript!

Делаем страницу приложением Angular

Чтобы сделать страницу приложением Angular, необходимо включить Angular в нее: Angular, конечно, исключительно догадлив, но он не умеет читать мысли! Добавить его несложно — это просто один внешний файл JavaScript, можно или скачать и сослаться на него непосредственно, или сослаться на CDN-версию.

Допустим, мы скачали его и ссылаемся на него локально. Тогда добавить его на страницу можно следующим образом:

```
<script src="angular.min.js"></script>
```

Просто сослаться на файл недостаточно, необходимо сообщить Angular, что данная страница является приложением Angular. Для этого можно добавить простой атрибут `ng-app` к открывающему тегу `<html>`, вот так:

```
<html ng-app>
```

Это даст Angular знать, что все находящееся в пределах тега `<html>` можно считать частью приложения.

СОВЕТ

ng-app можно назначить любому элементу страницы при необходимости ограничить область, к которой имеет доступ Angular. Его часто помещают в тег <html>, чтобы Angular мог работать на всей странице.

Связываем ввод и вывод

Как уже упоминалось, мы будем брать ввод из формы и отображать его в HTML без написания какого-либо JavaScript. Звучит невероятно, но мы действительно собираемся это сделать. Нам нужно всего лишь привязать модель Angular к вводу и выводу, а Angular сделает все остальное. Обе привязки должны ссылаться на одно и то же имя, чтобы Angular знал, что они используют одну и ту же модель.

Сначала мы привяжем модель к вводу, назначив ей имя, например, myInput следующим образом:

```
<input ng-model="myInput" />
```

Далее выведем значение модели в HTML там, где хотим. Для описания таких привязок Angular использует двойные фигурные скобки {{ }}. Чтобы привязать переменную модели в представлении, необходимо просто поместить ее имя между двойными фигурными скобками, вот так:

```
<h1>Hello {{ myInput }}</h1>
```

Собрав все вместе, получаем следующий листинг (листинг 8.1).

Листинг 8.1. Простая привязка Angular, привязывающая модель к вводу и выводу

```
<!DOCTYPE html>
<html ng-app>
<head>
<script src="angular.min.js"></script>
  <meta charset="utf-8">
  <title>Angular binding test</title>
</head>
<body>
  <input ng-model="myInput" />
  <h1>Hello {{ myInput }}</h1>
</body>
</html>
```

Выглядит не очень впечатляюще, не правда ли? Но запустите это в браузере и посмотрите, что получится, ведь на скриншотах видно не слишком хорошо. Если вы запустите это в браузере и начнете вводить текст в поле для ввода, то увидите показанное на рис. 8.3 поведение.

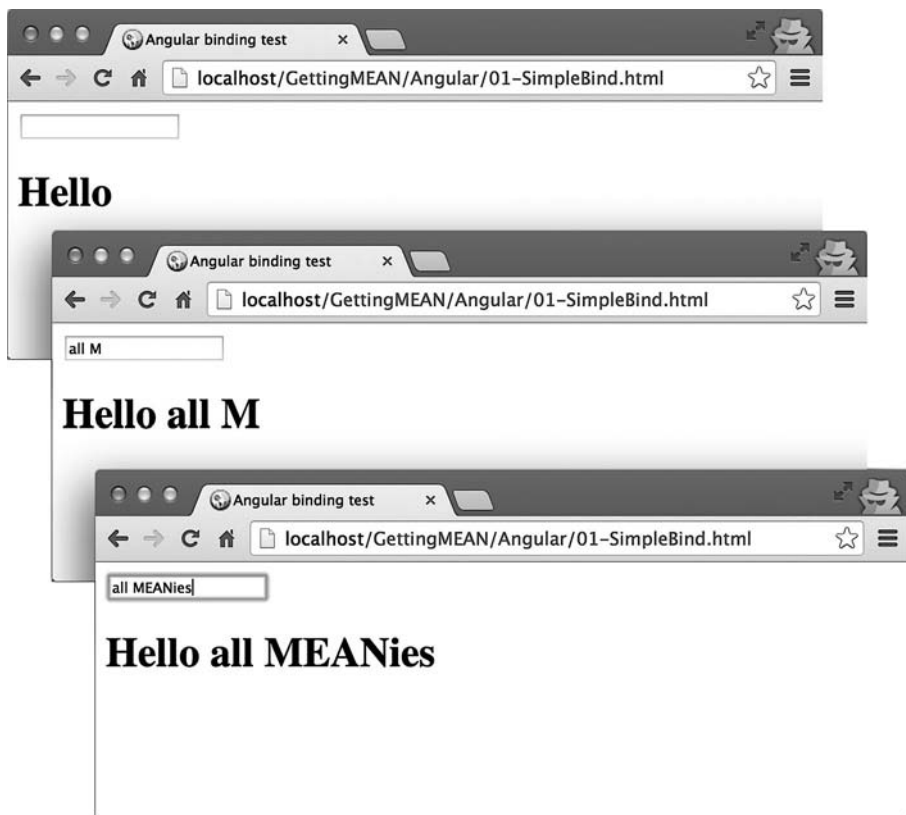


Рис. 8.3. С помощью простой привязки данных в Angular вводимый в поле для ввода текст сразу же выводится в теге `<h1>`. И ни одной строки кода писать не придется

Это работает, поскольку модель Angular привязана как к вводу, так и к выводу. На странице могут быть два элемента, но модель, содержащая данные, только одна.

Это все очень хорошо, но, вполне вероятно, вам захочется добиться с помощью Angular чего-то большего. А для этого придется написать немного кода.

8.1.2. Готовим все для настоящих достижений (и кода JavaScript)

Чтобы познакомиться с терминологией, которая пригодится при написании кода Angular, поставим перед собой простую цель — присвоить значение по умолчанию модели `myInput`. Для этого нам понадобится описать приложение Angular в качестве модуля и создать контроллер для управления областью видимости. Не переживайте о том, что это кажется бессмысленным, — оно обретет смысл к концу текущего раздела.

Создание приложения Angular в виде модуля

Для создания и написания кода приложения Angular необходимо описать модуль. Модуль описывается на JavaScript под выбранным нами именем, на которое затем будет ссылаться атрибут `ng-app` в HTML.

Начнем с этого, чтобы избавиться от данного куска работы. Просто добавляем имя приложения Angular или модуля в качестве значения атрибута `ng-app` в теге `<html>`. В следующем фрагменте кода указываем, что используем приложение Angular с именем `myApp`:

```
<html ng-app="myApp">
```

Во внешнем файле JavaScript создадим описание этого приложения с помощью простого оператора-сеттера. Конечно, мы должны задать то же самое имя:

```
angular.module('myApp', []);
```

Использование сеттер-синтаксиса — рекомендуемое решение при описании модуля, но чтобы разобраться в распространенном подходе, который вы можете увидеть в примерах кода в Интернете, прочитайте следующую врезку.

РАСПРОСТРАНЕННЫЙ, НО НЕ ТАКОЙ ХОРОШИЙ ПОДХОД

Рекомендуемое решение для описания модуля — использовать сеттер-синтаксис, подобный следующему:

```
angular.module('myApp', []);
```

```
// Хорошо
```

Но среди примеров кода в Интернете часто можно встретить присвоение переменной:

```
var myApp = angular.module('myApp', []);
```

```
// Не слишком хорошо
```

Этот подход преобладает главным образом благодаря оригинальной документации Angular, где он демонстрируется, но по мере обновления документации его постепенно заменяют. Использовать сеттер-синтаксис лучше по нескольким причинам. Он дает вам возможность использовать геттер-синтаксис для контроллеров и не только, как мы вскоре увидим, и делает код более удобным для чтения, а также снизит риск повторного использования имени существующей переменной.

Описание контроллера

Когда приложение создано и описано, можно подключить к нему контроллеры. Контроллеры описываются на JavaScript и прикрепляются к конкретному элементу HTML. Контроллер затем будет работать внутри соответствующего элемента.

Для нашего примера прикрепим контроллер к `body`. Это делается с помощью атрибута Angular `ng-controller`, которому присваивается имя контроллера,

который мы хотели бы использовать. В следующем фрагменте кода используется имя контроллера `myController`:

```
<body ng-controller="myController">
```

Теперь приходит очередь кода. Описав модуль приложения с помощью сеттер-синтаксиса, для описания контроллера мы можем воспользоваться геттер-синтаксисом. Следующий фрагмент кода демонстрирует прикрепление контроллера `myController` к модулю Angular `myApp`:

```
angular
  .module('myApp')
  .controller('myController', function() {
    // здесь находится код контроллера
  });
```

Получаем модуль `myApp`

Задаем контроллер `myController`, включая код контроллера в анонимную функцию

Это отлично работает, но для удобства чтения, повторного использования и тестирования лучше использовать поименованную функцию вместо анонимной. В следующем фрагменте кода мы как раз это и сделаем, создав новую поименованную функцию `myController` для кода контроллера:

```
var myController = function() {
  // здесь находится код контроллера
};
angular
  .module('myApp')
  .controller('myController', myController);
```

Размещаем контроллеры в поименованных функциях ради повышения качества кода

После обновления страницы мини-приложение должно работать по-прежнему, принимая вводимые данные и отображая их по мере ввода. Далее мы рассмотрим понятие области видимости, чтобы узнать, как задать значение по умолчанию для нашей модели.

Знакомимся с областями видимости

У приложений Angular, как и в коде JavaScript, существуют области видимости. Аналогично понятию глобальной области видимости в JavaScript, в Angular есть *rootScope*. *rootScope* соответствует всему приложению, она создается неявным образом с помощью директивы `ng-app` в HTML. Модель для первой версии нашего приложения «Hello world!» находилась внутри этой области видимости.

Внутри *rootScope* может быть вложено несколько дочерних областей видимости. Дочерняя область видимости также создается неявным образом при добавлении в HTML директивы `ng-controller`. Фактически при добавлении контроллера в приложение мы перемещаем модель из области видимости *rootScope* в область видимости контроллера.

В Angular области видимости связывают воедино представление, модель и контроллер, так как все они используют одну и ту же область видимости. Мы уже были

свидетелями функционирования области видимости при работе с моделью и представлением, хотя и не осознавали этого. Можно сделать область видимости более заметной, работая с ней в контроллере.

Функция контроллера может принимать параметр `$scope`, поскольку область видимости уже была создана Angular. Он должен носить название `$scope`, поскольку зависит от провайдера Angular `$scopeProvider`. Параметр `$scope` предоставляет нам непосредственный доступ к модели. Поэтому для задания значения по умолчанию необходимо всего лишь использовать стандартную точечную нотацию JavaScript для обращения к свойствам объекта. Следующий фрагмент кода показывает изменения в функции контроллера, необходимые для приема параметра `$scope` и использования его для задания значения по умолчанию:

```
var myController = function($scope) {
  $scope.myInput = "world!";
};
```

Принимаем параметр `$scope` для получения доступа к области видимости

`myInput` — свойство области видимости, так что можно легко присвоить ему значение

То, что мы задали значение по умолчанию, не означает его неизменности. Обновление поля `input` все равно будет обновлять модель и отображать ее. Рисунок 8.4 демонстрирует это в действии.

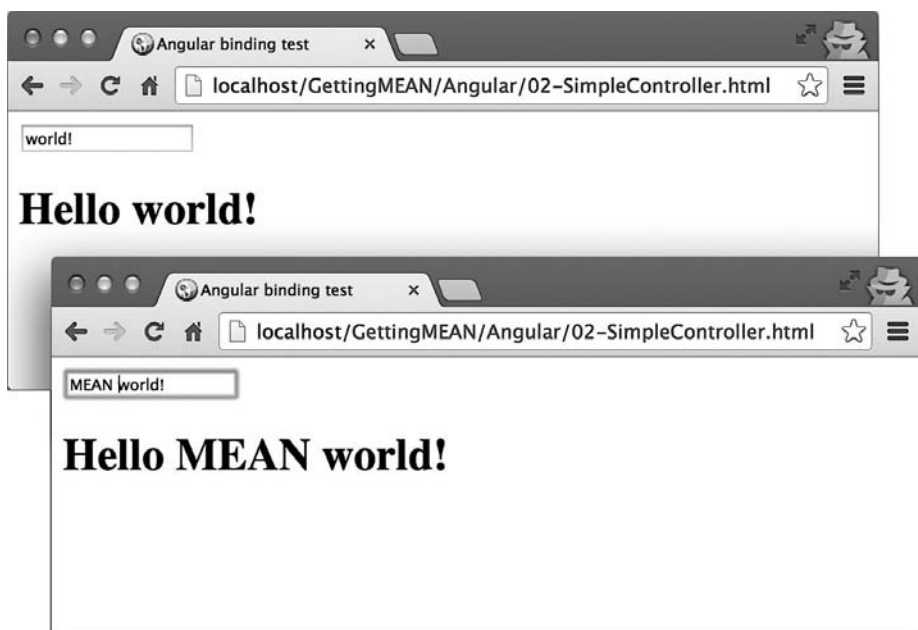


Рис. 8.4. Контроллер присваивает значение по умолчанию посредством области видимости, но это значение все равно можно менять с помощью `input`

В начале данного раздела говорилось, что мы опишем приложение Angular в виде модуля и создадим контроллер для управления областью видимости. Тогда это могло не иметь для вас особого смысла, но, надеюсь, звучит вполне осмысленно сейчас. Это основополагающие компоненты знаний об Angular.

Итак, теперь, когда мы понимаем, что делаем и откуда берется Angular, приступим к добавлению новых компонентов в приложение Loc8r.

8.2. Отображение и фильтрация списка для домашней страницы

В этом разделе мы увидим, как добавить в приложение компонент Angular. В данном случае заменим уже существующую функциональность. Мы хотим поменять способ, с помощью которого запрограммирована домашняя страница, и использовать Angular для отображения списка местоположений на домашней странице вместо выдачи HTML-кода с помощью Express. На протяжении данного раздела мы охватим немало функциональности Angular, которая может оказаться полезной в большинстве других проектов, включая фильтрацию списков, фильтры форматов данных, сервисы, внедрение зависимостей и использование директив для добавления допускающего повторное использование HTML-кода.

8.2.1. Добавление Angular в приложение Express

Из раздела 8.1 мы узнали, что необходимо сделать для добавления Angular в приложение Loc8r, а именно:

- ❑ загрузить файл библиотеки Angular;
- ❑ создать новый файл JavaScript для нашего кода;
- ❑ включить эти файлы в HTML-код;
- ❑ описать в HTML-коде приложение Angular.

Это не займет много времени, так что начнем.

Добавляем файлы JavaScript в проект

Поскольку Angular — клиентский фреймворк, необходимо сделать так, чтобы Express отправлял файлы JavaScript браузеру, а не пытался их выполнить. Каталог `public` уже настроен на выдачу статистических файлов, так что это идеальное место для добавления нескольких новых файлов JavaScript.

В каталоге `public` создайте новый каталог Angular. Переместите туда скачанный заархивированный файл библиотеки Angular. В том же каталоге создайте новый файл JavaScript `loc8rApp.js` — в нем будет содержаться наш код для этой главы.

Пока мы находимся тут, создадим сеттер модуля Angular для нашего приложения и назовем его `loc8rApp`. В файле `loc8rApp.js` введите следующий фрагмент кода и сохраните его:

```
angular.module('loc8rApp', []);
```

Пока что этого достаточно. Далее нам необходимо будет модифицировать представления для включения JavaScript и описания приложения Angular.

Настройка HTML

Модификация представления — несложное дело. На данном этапе добавим все необходимое в файл `layout.jade`, чтобы все страницы приложения могли использовать Angular.

Сначала добавим ссылки на два файла JavaScript рядом с другими внешними файлами внизу файла Jade, как показано в листинге 8.2.

Листинг 8.2. Добавление библиотеки Angular и кода приложения в HTML

```
script(src='/angular/angular.min.js')
script(src='/angular/loc8rApp.js')
script(src='/javascripts/jquery-1.11.1.min.js')
script(src='/bootstrap/js/bootstrap.min.js')
script(src='/javascript/validation.js')
```

Далее нам нужно будет просто описать приложение Angular в HTML. Это мы опять сделаем в теге `<html>`. Следующий фрагмент кода демонстрирует, как это делается в синтаксисе Jade при использовании имени `loc8rApp`:

```
html(ng-app='loc8rApp')
```

Теперь все страницы `Loc8r` готовы к выполнению каких-либо действий Angular.

8.2.2. Перемещение выдачи данных из Express в Angular

Если мы собираемся использовать Angular для отображения списка местоположений, то Angular понадобятся данные для этого списка. Начнем с проверки работоспособности подхода путем использования жестко зашитых в Angular данных аналогично тому, как мы делали при создании приложения Express, прежде чем в конце концов извлекать их из БД.

Для этого нам понадобится сделать следующее:

- ❑ убрать вызов API из контроллера Express для домашней страницы;
- ❑ добавить какие-либо жестко зашитые данные в область видимости приложения Angular;
- ❑ модифицировать шаблон представления для привязки данных Angular.

Начнем с модификации Express.

Удаление вызова API для домашней страницы из Express

В данных условиях практически все в контроллере Express для домашней страницы (`homelist` в `app_server/controllers/locations.js`) связано с обращением к API за данными. Мы удалим это обращение и будем просто вызывать функцию `renderHomepage`, как показано в следующем фрагменте кода:

```
module.exports.homelist = function(req, res){
  renderHomepage(req, res);
};
```

Функция `renderHomepage` также предназначена для работы с входящими данными, которые больше не нужны. В листинге 8.3 мы удалим все ссылки на `responseBody` и `message`.

Листинг 8.3. Убираем динамический контент из функции Express `renderHomepage`

```
var renderHomepage = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places to
work when out and about. Perhaps with coffee, cake or a pint? Let
Loc8r help you find the place you're looking for."
  });
};
```

Вот и все, что нам нужно было сделать с функцией контроллера. Теперь обратимся ненадолго к коду Angular для добавления некоторых данных в область видимости.

Добавление жестко зашитых данных в область видимости Angular

Начнем с того, что жестко зашьем некоторые данные в область видимости, чтобы можно было разобраться с представлением. Первый шаг — создание функции для кода нашего контроллера в файле `loc8rApp.js` в каталоге `/public/angular`. Назовем ее `locationListCtrl`, не забыв передать параметр `$scope` и присвоив свойству `data` массив объектов местоположений.

Листинг 8.4 демонстрирует создание контроллера с двумя местоположениями (вероятно, вы захотите добавить еще несколько, чтобы протестировать фильтрацию).

Листинг 8.4. Создаем контроллер Angular и некоторые тестовые данные в `loc8rApp.js`

```
var locationListCtrl = function ($scope) {
  $scope.data = {
    locations: [{
      name: 'Burger Queen',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 3,
      facilities: ['Hot drinks',
                  'Food', 'Premium wifi'],
      distance: '0.296456',
      _id: '5370a35f2536f6785f8dfb6a'
    }, {
      name: 'Costy',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 5,
      facilities: ['Hot drinks', 'Food', 'Alcoholic drinks'],
      distance: '0.7865456',
      _id: '5370a35f2536f6785f8dfb6a'
    }
  ]
};
```

Создаем новую функцию для кода контроллера Angular, принимающую параметр `$scope`

Создаем присоединенное к `$scope` свойство `data` и присваиваем ему несколько наборов данных для местоположений

Это все, что нам необходимо сделать в контроллере для помещения данных в область видимости, чтобы представление могло их использовать. Вскоре мы модифицируем этот код для получения из БД настоящих данных. Последнее, что желательно не забыть, — привязать контроллер к приложению Angular. Для этого воспользуемся геттер-синтаксисом, находящимся в самом низу файла `loc8rApp.js`, после кода контроллера, как показано в следующем фрагменте кода:

```
angular
  .module('loc8rApp')
  .controller('locationListCtrl', locationListCtrl);
```

Теперь контроллер привязан к приложению и помещает данные в область видимости. Но представление нигде не описывает контроллер Angular, так что он пока не будет использоваться и все еще применяет привязки данных Express. Поэтому вернемся к приложению Express и модифицируем представление для домашней страницы.

Модификация представления Jade для привязки к контроллеру Angular

Представление Jade все еще пытается использовать привязки Express и будет выдавать сбой, поскольку мы удалили весь нужный код из Express. Так что необходимо добавить в соответствующий элемент директиву `ng-controller`, заменить цикл Jade на цикл Angular и поменять привязки данных Jade на привязки данных Angular.

Хитрость тут в использовании цикла Angular. Вместо цикла `for` в Angular имеется директива `ng-repeat`, дающая возможность организовывать цикл по элементам массива. Директива `ng-repeat` будет выполнять цикл по массиву, выводя HTML и привязки данных для каждого элемента массива. Для получения более подробной информации обратитесь к следующей врезке.

ИСПОЛЬЗОВАНИЕ NG-REPEAT ДЛЯ ОРГАНИЗАЦИИ ЦИКЛА ПО ОБЪЕКТАМ ДАННЫХ

Директива Angular `ng-repeat` будет выполнять цикл по заданному массиву, визуализируя HTML-код для каждого из элементов данных. Например, начнем с простого контроллера, добавляющего массив данных в область видимости:

```
var myController = function($scope) {
    $scope.items = ["one", "two", "three"];
};
```

Вывести элементы массива с помощью директивы `ng-repeat` можно следующим образом:

```
<body ng-controller="myController">
  <ul>
    <li ng-repeat="item in items">{{ item }}</li>
  </ul>
</body>
```

В итоге будет выведено, не считая добавляемых Angular в элементы Express атрибутов, следующее:

```
<ul>
  <li>one</li>
  <li>two</li>
  <li>three</li>
</ul>
```

Обратите внимание на то, что Angular начнет повторение с того элемента, в который вы поместили директиву `ng-repeat`, так что будьте внимательнее при ее размещении. В приведенном примере очень легко можно было ошибиться и случайно поместить ее в тег `` вместо тега ``, что привело бы к выводу следующей разметки:

```
<ul>
  <li>one</li>
</ul>
<ul>
  <li>two</li>
</ul>
<ul>
  <li>three</li>
</ul>
```

А это, вероятно, совсем не то, чего вам хотелось. `ng-repeat` очень удобна и обладает большими возможностями, просто будьте внимательнее при ее размещении.

Все, что мы хотим сделать, содержится в основном столбце контента страницы, так что он будет подходящим местом для описания контроллера. Внутри этого контроллера все привязки данных и циклы Jade необходимо заменить их эквивалентами Angular. Листинг 8.5 иллюстрирует изменения, которые необходимы для этого.

Листинг 8.5. Меняем привязки Jade на привязки Angular

```
.col-xs-12.col-sm-8(ng-controller="locationListCtrl")
  .error {{ message }}
  .row.list-group
    >.col-xs-12.list-group-item(ng-repeat="location in data.locations")
      h4
        a(href="/location/{{ location._id }}")
          {{ location.name }}
        small {{ location.rating }}
        span.badge.pull-right.badge-default
          {{ location.distance }}
        p.address {{ location.address }}
      p
        span.label.label-warning.label-facility(ng-repeat="facility
          in location.facilities"
            | {{ facility }}
        )
```

Описываем контроллер Angular в div центрального столбца

Выводим подробности для каждого местоположения

Добавляем вложенный цикл `ng-repeat` для вывода предоставляемых услуг по каждому местоположению

Выполняем `ng-repeat` для каждого элемента массива местоположений в `@scope.data`

Это последний элемент пазла в первой попытке заставить Angular работать на домашней странице. Мы добавили сценарии, описали приложение в HTML и JavaScript, создали контроллер и привязали данные в HTML. Так что теперь при перезагрузке домашней страницы мы будем видеть что-то напоминающее рис. 8.5.

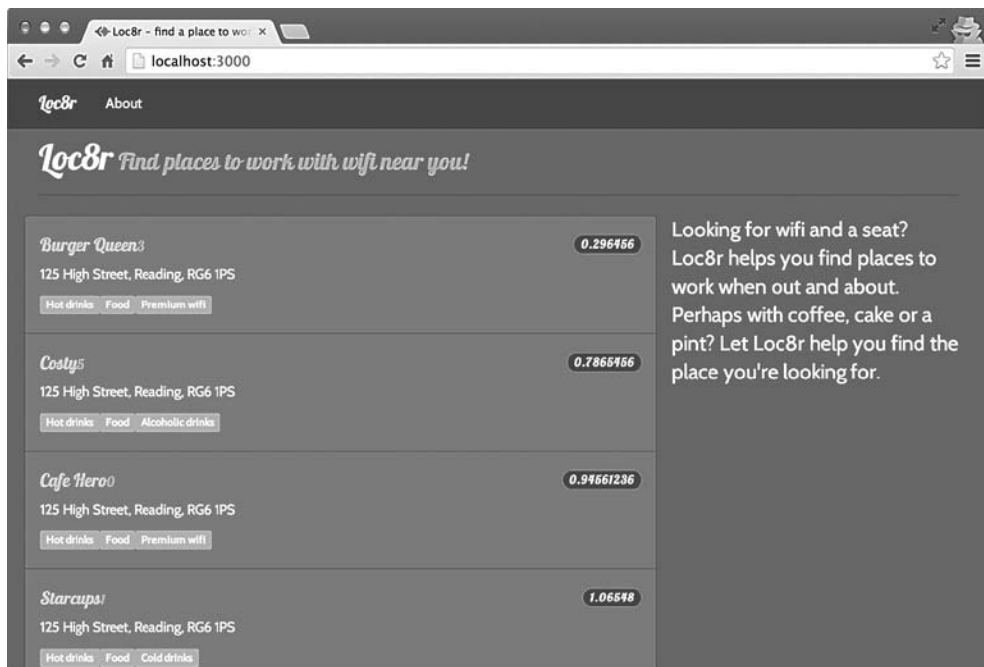


Рис. 8.5. Вместо Express данные для домашней страницы предоставляет и визуализирует Angular. Несколько проблем в макете все еще требуют исправления

Из рис. 8.5 видно, что кое-что требует коррекции. Мы применили Express для форматирования расстояния перед отправкой его представлению, а также использовали примесь Jade для вывода звезд в качестве оценок вместо чисел. Angular предоставляет нам несколько инструментов, чтобы сделать все это немного по-другому: фильтры и директивы. Сначала рассмотрим фильтры.

8.2.3. Фильтры Angular для форматирования данных

Фильтры позволяют задавать выбранный вами формат для конкретного элемента данных. В Angular имеются несколько встроенных фильтров, таких как форматирование даты, валюты и текста. Их можно использовать непосредственно в при-

вязке данных в HTML. Для применения фильтра к значению вставьте после него вертикальную черту | и название фильтра.

Следующий фрагмент кода демонстрирует пример использования фильтра `currency`:

```
<div>{{ 123.2345 | currency }}</div>
<!-- Output: $123.23 -->
```

Некоторые фильтры позволяют вам задавать параметры. Фильтру `currency` можно указать выводить другой символ валюты, как показано в следующем фрагменте кода:

```
<div>{{ 123.2345 + 321.321 | currency:"£" }}</div>
<!-- Output: £444.56 -->
```

Конечно, фильтры могут работать не только с числовыми значениями — можно работать также со строками и датами. Вот несколько коротких примеров:

```
<div>{{ "Let's shout" | uppercase }}</div>
<!-- Вывод: LET'S SHOUT -->
{{ timestamp | date:"d MMM уууу" }}
<!-- Output: 21 Aug 2014 -->
```

Необходимо, чтобы timestamp был описан в области видимости как объект даты

Поставляемые вместе с Angular встроенные фильтры довольно неплохи. И хотя полезно будет иметь их в своем распоряжении, иногда вам может понадобиться сделать что-то иное, как в случае с `Loc8r`, где необходимо форматировать расстояния. Хорошая новость заключается в том, что можно создавать для своего приложения собственные пользовательские фильтры.

Создание пользовательского фильтра

API `Loc8r` возвращает расстояния в виде длинных чисел, например `0,296 456`. Это расстояние на самом деле указано в километрах, но не показывает явным образом какой-либо единицы измерения. Хотелось бы, чтобы для конечного пользователя оно выглядело понятнее.

Мы уже решили эту задачу в Express, но теперь необходимо решить ее в Angular. Поскольку мы используем JavaScript как в серверной, так и в клиентской части, то можем взять нужный код из Express и вставить его в Angular.

В Express, в файле `locations.js` из каталога `app_server/controllers`, у нас было две функции для форматирования данных: `_isNumeric` и `_formatDistance`. Мы можем сохранить логику этих функций. Единственное, что необходимо сделать при перемещении их в Angular, — изменить функцию `_formatDistance` таким образом, чтобы она возвращала функцию для выполнения обработки, вместо того чтобы выполнять обработку самостоятельно. В листинге 8.6 приведен код, который необходимо поместить в файл `Loc8rApp.js`.

Листинг 8.6. Создание пользовательского фильтра для форматирования расстояний

Вспомогательная
функция `_isNumeric`
скопирована
непосредственно
из кода Express

```
var _isNumeric = function (n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
};
var formatDistance = function () {
    return function (distance) {
        var numDistance, unit;
        if (distance && _isNumeric(distance)) {
            if (distance > 1) {
                numDistance = parseFloat(distance).toFixed(1);
                unit = 'km';
            } else {
                numDistance = parseInt(distance * 1000,10);
                unit = 'm';
            }
            return numDistance + unit;
        } else {
            return "?";
        }
    };
};
```

Для использования
в качестве фильтра Angular
функция `formatDistance`
должна возвращать функцию,
принимающую на входе
параметр расстояния,
а не принимать его сама

Содержимое функции
остается тем же
и может быть
скопировано
непосредственно
из приложения
Express

Этот код демонстрирует создание пользовательского фильтра, но теперь нам нужно добавить его в приложение и использовать в HTML.

Добавление и использование пользовательских фильтров

После создания пользовательского фильтра нам хотелось бы добавить его в приложение, чтобы иметь возможность использовать. Мы можем присоединить его в цепочку в геттер-синтаксисе модуля `Angular` внизу файла `loc8rApp.js`, там, где мы зарегистрировали контроллер для приложения. Принцип остается тем же, только вместо описания контроллера мы описываем фильтр.

Следующий фрагмент кода демонстрирует, как должен выглядеть этот раздел кода:

```
angular
    .module('loc8rApp')
    .controller('locationListCtrl', locationListCtrl)
    .filter('formatDistance', formatDistance);
```


Теперь фильтр доступен для использования в приложении, так что мы можем сослаться на него в коде точно так же, как на любой другой фильтр. В следующем фрагменте кода добавляем фильтр в соответствующее место в шаблоне Jade:

```
.col-xs-12.list-group-item(ng-repeat="location in data.locations")
  h4
    a(href="/location/{{ location._id }}") {{ location.name }}
    small {{ location.rating }}
    span.badge.pull-right.badge-default {{ location.distance | formatDistance
}}
  p.address {{ location.address }}
```

Теперь расстояние будет аккуратно форматироваться точно так же, как и раньше, и мы получим что-то вроде рис. 8.6.

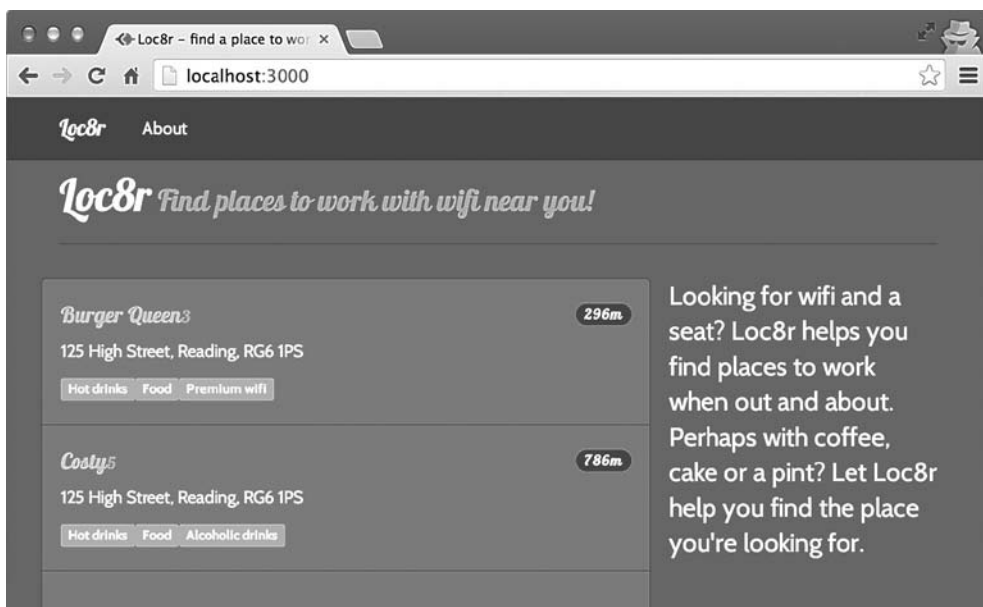


Рис. 8.6. Пользовательский фильтр Angular форматирует и отображает расстояния так, как нам нужно

На этом, рассмотрев некоторые из встроенных фильтров Angular и научившись создавать собственные, завершим краткий обзор фильтров. Следующими в нашем списке того, что необходимо сделать, значатся директивы для HTML-сниппетов, чтобы снова отображать звезды для оценок.

8.2.4. Директивы Angular для создания HTML-сниппетов

В Angular директивы дают возможность создавать HTML-сниппеты. Отдельный сниппет можно использовать в любом количестве различных контроллеров. Это действительно полезная возможность для того, чтобы сделать свое приложение более единообразным и удобным в сопровождении. А поскольку эти сниппеты выполняются в контексте приложения Angular, они обладают всеми достоинствами привязки данных, с которыми мы познакомились ранее.

В качестве бонуса браузеры могут кэшировать директивы, сохраненные в виде HTML-файлов, что позволяет ускорять работу приложения при переключении пользователя между представлениями. Но вернемся немного назад и рассмотрим, как добавить директиву в приложение.

Добавление директивы в приложение Angular

В приложении Angular начнем с создания кода для простейшей директивы и будем постепенно наращивать возможности. Директиву можно добавить в приложение Angular с помощью уже знакомого вам геттер-синтаксиса и поименованной функции. В качестве отправной точки для создания директив мы хотели бы возвращать простой шаблон, выводящий оценку в виде числа, так что в браузере она будет выглядеть так же, как сейчас.

Следующий листинг демонстрирует, что именно необходимо добавить или поменять в файле `loc8rApp.js`.

Листинг 8.7. Создаем директиву и добавляем ее в модуль Angular

```
var ratingStars = function () {
  return {
    template : "{{ location.rating }}"
  };
};

angular
  .module('loc8rApp')
  .controller('locationListCtrl', locationListCtrl)
  .filter('formatDistance', formatDistance)
  .directive('ratingStars', ratingStars);
```

Создаем новую функцию `ratingStars` и возвращаем простейший шаблон, привязанный к оценке местоположения

Регистрируем директиву в приложении

Этот простой шаблон — то же самое, что у нас сейчас имеется в шаблоне Jade. То, что имя функции написано в верблюжьем регистре, — существенно, позднее вы поймете почему, когда мы будем модифицировать представление Jade для использования этой новой директивы.

Прикрепление директивы к шаблону HTML

Директива в приложении Angular — это, конечно, очень хорошо, но нам необходимо сообщить HTML-коду, в каком месте мы хотели бы ее использовать. Стандартный путь для выполнения этого — добавление атрибута в тег, в котором будет находиться директива.

Имя атрибута имеет значение и должно соответствовать имени директивы, но в другом формате. В отличие от JavaScript, HTML-код нечувствителен к регистру, так что не имеет смысла использовать в нем *верблюжий регистр*. Вместо этого символы в верхнем регистре преобразуются в символы в нижнем регистре и перед ними ставится дефис. Так что имя директивы `ratingStars` будет выглядеть в HTML как `rating-stars`.

В HTML-коде в применении к тегу `<div>` это будет выглядеть следующим образом:

```
<div rating-stars></div>
```

В домашней странице мы хотели бы добавить его к тегу `small`. Шаблоны Jade позволяют добавлять атрибуты без значения — это именно то, что здесь требуется. Это мы и делаем в следующем фрагменте кода:

```
h4
  a(href="/location/{{ location._id }}") {{ location.name }}
  small(rating-stars)
```

←

Используем пустое имя атрибута для тега `small`, в котором будет содержаться оценка

Это простое изменение позволяет Angular привязать директиву `ratingStars` к тегу `small`, вставляя в него контент. Если вы теперь перезагрузите страницу, то не увидите каких-либо отличий, ведь мы изменили только способ вывода Angular числа.

Это неплохо для начала, но основная идея директив в том, что их можно использовать многократно. Мы уже получили некий код для многократного использования, но сейчас сделаем его таковым в еще большей степени, убрав зависимость от наличия присвоенного переменной `location.rating` значения в текущей области видимости. Мы сделаем это с помощью так называемой *изолированной области видимости* (*isolate scope*).

Передача переменных директиве с помощью изолированной области видимости

Текущий шаблон для отображения оценок будет функционировать только в том случае, когда директива включена в область видимости, содержащую значение для переменной `location.rating`. Пока что это может выглядеть вполне достаточным,

но что, если мы захотим отображать оценочные звезды для каждого отзыва в списке отзывов? Вряд ли они станут ссылаться на объект `location.rating`, а если и да, то вряд ли это на самом деле будет тот фрагмент информации, который мы хотели бы использовать.

Чтобы справиться с этой задачей, можно создать изолированную область видимости для директивы, добавив в описание директивы параметр области видимости. Листинг 8.8 демонстрирует это, создавая для директивы новую переменную области видимости `thisRating`. Значение `'=rating'` говорит Angular, что необходимо использовать атрибут `rating` того же элемента HTML, в котором описана директива.

Листинг 8.8. Модификация директивы для использования изолированной области видимости для оценки

```
var ratingStars = function () {
  return {
    scope: {
      thisRating : '=rating'
    },
    template : "{{ thisRating }}"
  };
};
```

Добавляем параметр области видимости в описание директивы для создания изолированной области видимости

Создаем новую переменную `thisRating` и сообщаем Angular о необходимости получения значения из атрибута `rating`

Модифицируем шаблон для использования новой переменной

Теперь эта директива ожидает, что элемент HTML, к которому она привязана, будет содержать атрибут `rating`, который, в свою очередь, будет содержать значение оценки. Следующий фрагмент кода демонстрирует, какие изменения в шаблоне Jade необходимо выполнить для этого:

```
h4
  a(href="/location/{{ location._id }}") {{ location.name }}
  small(rating-stars, rating="location.rating")
```

Создаем новый атрибут `rating` и присваиваем ему значение оценки

Это изменение означает, что, где бы мы ни захотели использовать звезды для оценок, мы сможем добавить в элемент HTML два атрибута: один для привязки директивы, второй — для передачи значения оценки. Таким образом, директива больше не зависит от наличия конкретных значений в родительской области видимости.

Но даже после всего этого звезды пока не отображаются. Сейчас мы с этим разберемся.

Использование внешнего файла HTML для шаблона

Эмпирическое правило гласит: директивы, за исключением простейших, должны находиться в своих собственных файлах HTML. Наша директива для отображения звезд в качестве оценок несколько сложнее простого отображения передаваемого ей числа, так что поместим ее во внешний HTML-файл. Помимо разделения обязанностей — отделения разметки от логики приложения, такой подход дает возможность браузерам кешировать HTML-файлы.

Первое, что необходимо сделать для перемещения шаблона во внешний HTML-файл, — заменить `template` из описания директивы на `templateUrl`. Переменная `templateUrl` должна содержать путь к HTML-файлу. Листинг 8.9 показывает, как изменить директиву `ratingStars` для использования файла `/angular/rating-stars.html`.

Листинг 8.9. Изменение директивы для использования внешнего файла для шаблона

```
var ratingStars = function () {
  return {
    scope: {
      thisRating : '=rating'
    },
    templateUrl: '/angular/rating-stars.html'
  };
};
```

Меняем `template` на `templateUrl` и указываем путь к тому HTML-файлу, который хотим использовать

HTML-файл пока еще не существует, поэтому создадим его прямо сейчас. Создайте в каталоге `/public/angular` файл `rating-stars.html`. В этом файле мы начнем с HTML, необходимого для вывода оценок: следующий фрагмент кода показывает разметку, необходимую для отображения оценки в три звезды:

```
<span class="glyphicon glyphicon-star"></span>
<span class="glyphicon glyphicon-star"></span>
<span class="glyphicon glyphicon-star"></span>
<span class="glyphicon glyphicon-star-empty"></span>
<span class="glyphicon glyphicon-star-empty"></span>
```

Класс выводит сплошные звезды в количестве трех единиц

Класс выводит пустотелые звезды в количестве двух единиц

Чтобы сделать шаблон интеллектуальным, можно воспользоваться привязкой Angular для вставки в соответствующих местах суффикса класса `-empty`. Например, если оценка меньше 2, только первая звезда должна быть сплошной, а остальные четыре — пустотелыми. Для этого можно воспользоваться тернарным оператором JavaScript (сокращенная запись простого условного оператора `if-else`), как показано в листинге 8.10.

Листинг 8.10. Создание шаблона Angular для звезд оценок

```
<span class="glyphicon glyphicon-star{{ thisRating<1 ? '-empty' : '' }}"></span>
<span class="glyphicon glyphicon-star{{ thisRating<2 ? '-empty' : '' }}"></span>
<span class="glyphicon glyphicon-star{{ thisRating<3 ? '-empty' : '' }}"></span>
<span class="glyphicon glyphicon-star{{ thisRating<4 ? '-empty' : '' }}"></span>
<span class="glyphicon glyphicon-star{{ thisRating<5 ? '-empty' : '' }}"></span>
```

Для каждой звезды, если `thisRating` меньше количества звезд, Angular будет добавлять к классу суффикс `-empty`. При перезагрузке страницы мы опять увидим звезды (рис. 8.7).

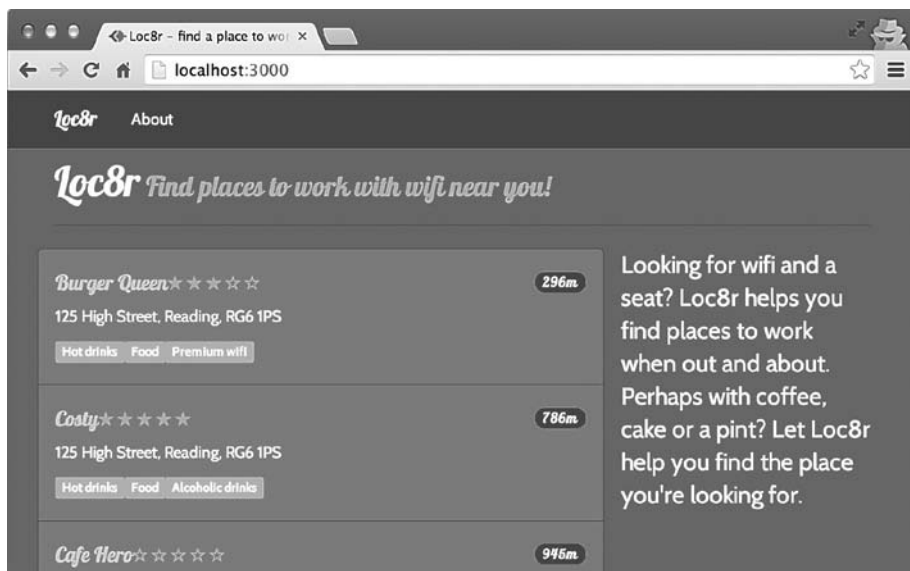


Рис. 8.7. Директива Angular для оценок доделана, звезды для оценок снова отображаются на своих местах

Этот шаг завершает работу по правильному отображению списка с помощью Angular. Но мы собирались добавить текстовый ввод, чтобы можно было фильтровать результаты. Давайте сделаем это.

Фильтрация списка результатов с помощью Angular

Ранее в данной главе мы увидели, как поменять вывод конкретного элемента данных с помощью фильтров. Поразительно то, что можно поступить аналогично со списком результатов, добавив фильтр в директиву `ng-repeat`. Фактически мы собираемся отфильтровать список местоположений на домашней странице без всякого JavaScript.

Наша цель — отфильтровать список в соответствии с тем, что вводит пользователь, так что нам понадобится создать поле для ввода. Привяжем поле `input` к модели, а затем применим его в качестве фильтра к директиве `ng-repeat`. Это действительно настолько просто!

Листинг 8.11 демонстрирует необходимые для этого настройки в шаблоне Jade для домашней страницы.

Листинг 8.11. Создание и использование текстового фильтра для списка результатов

```

.col-xs-12.col-sm-8(ng-controller="locationListCtrl")
  label(for="filter") Filter results
  input#filter(type="text", name="filter", ng-model="textFilter") ←
  .error {{ message }}
  .row.list-group
    .col-xs-12.list-group-item(ng-repeat="location in data.locations
      | filter: textFilter") ←

```

Создаем новое поле ввода и привязываем его к модели textFilter

Применяем фильтр к директиве ng-repeat, ссылаясь на textFilter

Вот и все. Представьте себе, сколько усилий пришлось бы приложить для программирования всей этой функциональности вручную, с нуля! Прежде чем мы увидим скриншот, добавим небольшое улучшение в CSS в файле `public/stylesheets/style.css`, чтобы он не выглядел настолько сжато:

```
#filter {margin-left: 4px;}
```

Теперь мы можем взглянуть на то, как это выглядит и как работает (рис. 8.8).

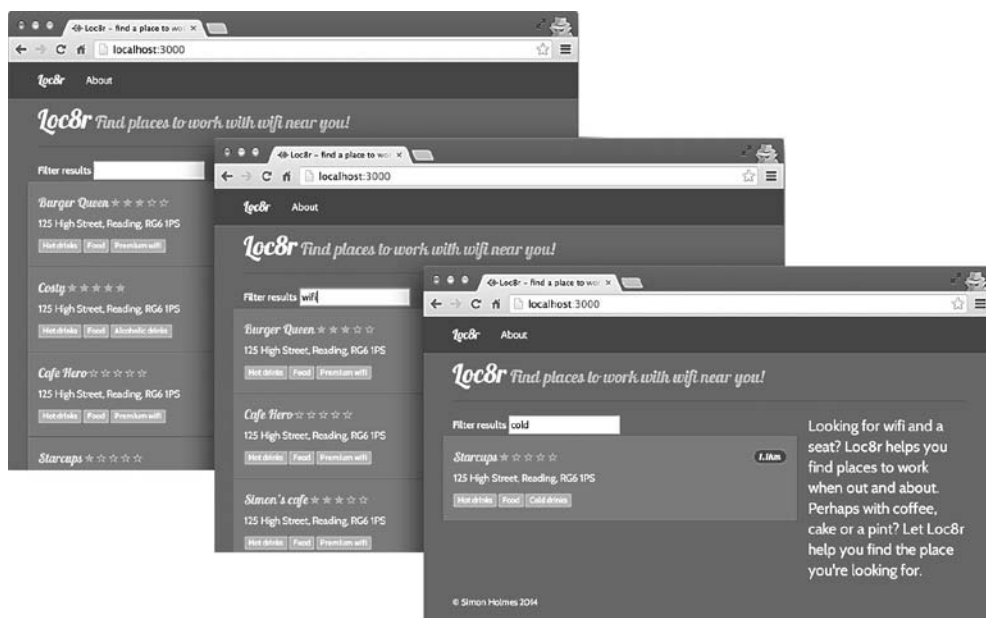


Рис. 8.8. Текстовый фильтр работает на домашней странице, фильтруя список результатов по мере того, как пользователь набирает его на клавиатуре

Полагаю, вы согласны: выглядит весьма аккуратно. Но не забываем, что данные для этого все еще жестко зашиты в контроллер. В следующем разделе рассмотрим, как с помощью нашего API извлечь данные из БД.

8.3. Получение данных из API

В этом разделе мы планируем предпринять несколько вещей. Во-первых, можем похвалить себя за то, что дальновидно сделали для нашей базы данных API REST. Когда ликование чуть-чуть утихнет, воспользуемся API из Angular для получения данных для домашней страницы, а затем сделаем так, чтобы страница учитывала местоположение. Это значит, что приложение теперь начнет показывать места, расположенные возле вас, а не только вблизи жестко заданных вами координат.

Что ж, приступим. Начнем с перемещения данных из контроллера в сервис.

8.3.1. Сервисы для данных

Сервисы — автономные единицы функциональности, которые можно объединять для обеспечения полной функциональности приложения. Постепенно вы начнете использовать сервисы в Angular очень часто, поскольку выполнение большей части логики приложения желательно поручить им, чтобы обеспечить возможность его многократного применения несколькими контроллерами. В данном случае мы создадим информационный сервис, который сможем использовать где угодно.

При доступе к данным контроллеру должен быть безразличен их источник, важно только то, кто их запрашивает. Именно это мы и собираемся сейчас сделать: переместить данные из контроллера в сервис с последующим вызовом контроллером сервиса.

Создание простого информационного сервиса

Способ описания сервиса в Angular уже должен быть вам знаком. Мы создадим поименованную функцию, а затем зарегистрируем сервис в приложении.

Итак, объявим функцию `loc8rData`, которая будет возвращать данные, в настоящий момент находящиеся в контроллере. Затем регистрируем ее в приложении. Листинг 8.12 демонстрирует оба этих шага.

Листинг 8.12. Создание простого информационного сервиса и добавление его в модуль

```
var loc8rData = function () {
  return [{
    name: 'Burger Queen',
    address: '125 High Street, Reading, RG6 1PS',
    rating: 3,
    facilities: ['Hot drinks', 'Food', 'Premium wifi'],
    distance: '0.296456',
    _id: '5370a35f2536f6785f8dfb6a'
  }];
};
angular
  .module('loc8rApp')
  .controller('locationListCtrl', locationListCtrl)
  .filter('formatDistance', formatDistance)
  .directive('ratingStars', ratingStars)
  .service('loc8rData', loc8rData);
```

Создаем новую поименованную функцию для возвращения данных

Регистрируем сервис в приложении

Вполне интуитивно понятно, не правда ли? И полностью соответствует паттернам, рассмотренным нами ранее в данной главе.

Использование сервиса из контроллера

Для использования сервиса из контроллера необходимо сначала передать сервис контроллеру. Передача сервиса не представляет сложностей — просто добавляем его имя в качестве параметра в описание функции для контроллера.

Как вы и могли ожидать, после передачи сервиса контроллеру его можно использовать. Следующий фрагмент кода демонстрирует модификацию функции контроллера для приема сервиса и его использования:

```
var locationListCtrl = function ($scope, loc8rData) {
  $scope.data = { locations: loc8rData };
};
```

Передаем имя сервиса в функцию контроллера в качестве параметра

Вызываем сервис, который затем возвращает данные

При перезапуске приложения мы не увидим каких-либо изменений на домашней странице. Выполненные тут изменения — неприметная подготовка к следующему этапу, которым является получение данных из API.

8.3.2. Выполнение HTTP-запросов к API из Angular

Выполнение HTTP-запросов из JavaScript не новинка. В jQuery уже давно имеется неплохая поддержка Ajax, а в главе 7 мы использовали модуль `request` для выполнения HTTP-запросов из Node. В Angular для подобного типа запросов имеется модуль `$http`.

Добавление и использование сервиса `$http`

Добавить сервис `$http` в приложение — несложная задача. По сути, просто нужно воспользоваться им в нашем информационном сервисе. Не забывайте, что контроллеру неважно, откуда берутся данные, он просто знает, кто их запрашивает.

Чтобы начать работу с сервисом `$http`, необходимо знать о нем две ключевые вещи.

- Сервис `$http` предоставляется другому сервису путем передачи его функции в виде параметра, аналогично тому, как мы делали с `$scope` в контроллере.
- У сервиса `$http` имеется метод `get`, для вызова которого достаточно одного-единственного параметра на входе — URL.

Эти два момента вместе приводят к тому, что модифицированная функция информационного сервиса `loc8rData` будет выглядеть так, как показано в следующем фрагменте кода:

```
var loc8rData = function ($http) {
  return $http.get('/api/locations?lng=-0.79&lat=51.3&maxDistance=20');
}
```

Передаем сервис `$http` в существующую функцию сервиса

Убираем жестко зашитые данные и возвращаем вызов `$http.get`, убедившись, что он обращается по правильному URL

Итак, теперь вместо возврата каких-то жестко зашитых данных информационный сервис возвращает вызов метода `$http.get`. Но если вы перезагрузите страницу, надеясь увидеть данные на домашней странице, то будете разочарованы, хотя и сможете увидеть в консоли JavaScript вашего браузера (в зависимости от браузера), что запрос ХНР к API был загружен и вернул какие-то данные.

Так почему же страница не отображает данные? Это происходит потому, что `$http.get` — асинхронный метод, а контроллер пытается использовать его как синхронный. Давайте посмотрим, как использовать его надлежащим образом.

Учитываем асинхронную природу \$http

Поскольку метод `$http` недоступен при работе с веб-сервисами, неизвестно, сколько времени займет его выполнение. А никому не хотелось бы, чтобы вся программа JavaScript прекращала работу на время ожидания ответа, так что имеет смысл выполнять `$http` асинхронно.

В настоящий момент код нашего контроллера выглядит вот так:

```
var locationListCtrl = function ($scope, loc8rData) {
  $scope.data = { locations: loc8rData };
};
```

Он выполняется синхронно и, как мы видели, не работает с методом `$http`. Вместо этого `$http` возвращает промис с двумя методами: `success` и `error`. Это значит, что вызов `$http.get` вместо возврата данных вызовет или метод `success`, или метод `error`.

Поясню это с помощью небольшого примера кода. Сервис `loc8rData` сейчас возвращает метод `$http.get`, так что начнем с его вызова. Затем присоединим к нему методы `success` и `error`, применив синтаксис с использованием точки. Получив ответ, `loc8rData` вызовет или `success`, или `error` в зависимости от ответа.

Основной каркас примера показан в листинге 8.13, демонстрирующем также получение возвращаемых при успешном запросе данных и передачу их в область видимости.

Листинг 8.13. Модификация контроллера для синхронной работы с промисами `$http`

```
var locationListCtrl = function ($scope, loc8rData) {
  loc8rData
  .success(function(data) {
    ▶ $scope.data = { locations: data };
  })
  .error(function (e) {
    console.log(e);
  });
};
```

Обращаемся к сервису `loc8rData`, который возвращает вызов `$http.get`

Если в ответе говорится, что все выполнено успешно, передаем возвращенные данные в функцию обратного вызова

Если веб-сервис вернул ошибку, передаем ее функции обратного вызова

Передаем эти данные в область видимости

После выполнения мы снова сможем увидеть данные из БД на нашей домашней странице. Это минимальная функциональность, но, поскольку код асинхронный, было бы неплохо в то время, пока отображается пустая страница и выполняется запрос `$http`, сообщать пользователю, что происходит.

Сообщаем пользователю, что происходит

Проблема с асинхронными обращениями к данным на стороне клиента состоит в том, что сначала пользователь видит страницу, визуализированную без всякого контента. Это продолжается до тех пор, пока асинхронный вызов не завершится и не вернет какие-то данные. Это не очень-то хорошо! Если на вашей странице в течение даже очень короткого промежутка времени нет данных, пользователи могут подумать, что страница пуста, и нажать на кнопку возврата к предыдущей странице.

Поэтому при выполнении асинхронных обращений за данными всегда оповещайте пользователя, что в фоновом режиме что-то выполняется. В `Loc8r` мы будем это делать, выводя простое сообщение, главным образом, потому, что у нас есть нужный для этого HTML-код.

У нас уже есть в шаблоне Jade раздел `div`, в котором будет находиться сообщение, как видно из выделенного полужирным шрифтом текста в следующем фрагменте кода:

```
.col-xs-12.col-sm-8(ng-controller="locationListCtrl")
  label(for="filter") Filter results
  input#filter(type="text", name="filter", ng-model="textFilter")
  .error {{ message }}
```

Этот раздел `div` и сообщение были частью исходного шаблона Jade, когда мы использовали Express для генерации итогового HTML-кода. При настройке контроллера Angular мы модифицировали `message`, привязав его к Angular. Так что можем использовать привязку `message` в области видимости в контроллере. Листинг 8.14 демонстрирует модификацию области видимости для отображения различных сообщений на разных этапах процесса.

Листинг 8.14. Задаем выводимое на разных этапах процесса сообщение

```
var locationListCtrl = function ($scope, loc8rData) {
  $scope.message = "Searching for nearby places";
  loc8rData
    .success(function(data) {
      $scope.message = data.length > 0 ? "" :
        "No locations found";
      $scope.data = { locations: data };
    })
    .error(function (e) {
      $scope.message = "Sorry, something's gone wrong ";
    });
};
```

Задаем сообщение по умолчанию, извещающее пользователя, что мы выполняем какие-то действия в фоновом режиме

Если запрос завершается успешно и возвращает какие-то данные, очищаем сообщение, в ином случае извещаем пользователя, что ничего найти не удалось

Если асинхронный вызов вернул ошибку, извещаем пользователя, что что-то пошло не так

Вот такая простая вещь может радикально отразиться на взаимодействии с пользователем. Если вы хотите добиться еще большего и включить Ajax Spinner — обычно анимированный GIF, изображающий вращающееся колесико, — вперед! Но мы оставим все как есть, поскольку нам предстоит кое-что поинтереснее.

Сейчас мы собираемся добиться того, чтобы приложение отображало места, находящиеся рядом именно с вами, а не просто рядом с зашитыми вами координатами.

8.3.3. Добавляем HTML-геолокацию для поиска местоположений, находящихся рядом с вами

Главная идея Lос8г заключается в учете местоположения пользователя, а следовательно, в способности находить расположенные рядом с ним места. До сих пор мы имитировали это путем жесткого указания географических координат в запросах API. Теперь все поменяем, добавив HTML5-геолокацию.

Чтобы все заработало, необходимо сделать следующее:

- добавить обращение к API HTML5-геолокации в приложение Angular;
- выполнять поиск мест только после получения местоположения;
- передавать координаты информационному сервису Angular, убрав жестко зашитые координаты;
- по ходу работы выводить сообщения, чтобы пользователь понимал, что происходит.

Начнем с первого пункта — добавим JavaScript-функции геолокации путем создания нового сервиса.

Создание геолокационного сервиса Angular

Способность находить местоположение пользователей выглядит кандидатом на повторное использование как в этом, так и в других проектах. Поэтому создадим еще один сервис, в котором будет находиться этот автономный элемент функциональности. Как правило, любой код, взаимодействующий с различными API, выполняющий логику приложения или какие-либо операции, должен быть реализован в виде сервиса. *Пусть контроллеры управляют сервисами, а не выполняют функции.*

Мы не станем прямо сейчас углубляться в детали того, как работает геолокационный API HTML5/JavaScript. Отметим только, что у современных браузеров имеется метод объекта `navigator`, который можно вызвать для получения координат пользователя. Однако пользователь должен дать разрешение на это. Метод принимает два параметра: обратный вызов на случай успешного выполнения и обратный вызов для ошибки — и выглядит следующим образом:

```
navigator.geolocation.getCurrentPosition(cbSuccess, cbError);
```

Нам понадобится обернуть стандартный геолокационный сценарий в функцию, чтобы можно было использовать его в качестве сервиса, а также добавить перехват ошибок на случай, если браузер пользователя не поддерживает такой возможности. Листинг 8.15 демонстрирует код, необходимый для создания сервиса `geolocation`, и предоставляет метод `getPosition`, который сможет вызывать контроллер.

Листинг 8.15. Создаем сервис `geolocation`, возвращающий метод для получения текущего местоположения

```

var geolocation = function () {
  var getPosition = function (cbSuccess, cbError, cbNoGeo) {
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(cbSuccess,
                                              cbError);
    }
    else {
      cbNoGeo();
    }
  };
  return {
    getPosition : getPosition
  };
};

```

Создаем сервис `geolocation`

Описываем функцию `getPosition`, принимающую на входе три функции обратного вызова: для успешного завершения, ошибки и случая, когда возможность не поддерживается

Если геолокация не поддерживается, выполняем обратный вызов для случая, когда возможность не поддерживается

Если геолокация поддерживается, вызываем нативный метод, передавая функции обратного вызова для успешного завершения и ошибки

Возвращаем функцию `getPosition`, чтобы можно было вызвать ее из контроллера

Этот код дает нам сервис `geolocation` с методом `getPosition`, которому можно передать три функции обратного вызова. Сервис проверит, поддерживает ли браузер геолокацию, после чего попытается получить координаты. Затем сервис выполнит один из трех различных обратных вызовов в зависимости от того, поддерживается ли геолокация и удалось ли получить координаты.

Следующий этап — добавление этого сервиса в приложение.

Добавляем геолокационный сервис в приложение

Чтобы использовать новый геолокационный сервис, необходимо зарегистрировать его в приложении и внедрить в контроллер.

В следующем фрагменте кода полужирным шрифтом выделено дополнение, которое необходимо сделать в описании модуля, чтобы зарегистрировать сервис в приложении:

```
angular
  .module('loc8rApp')
  .controller('locationListCtrl', locationListCtrl)
  .filter('formatDistance', formatDistance)
  .directive('ratingStars', ratingStars)
  .service('loc8rData', loc8rData)
  .service('geolocation', geolocation);
```

Когда геолокационный сервис зарегистрирован в приложении, необходимо его внедрить в контроллер `locationListCtrl`, чтобы последний мог его использовать. Для этого нужно просто добавить имя сервиса в принимаемые функцией контроллера параметры. Следующий фрагмент кода демонстрирует это (содержимое контроллера не включено во фрагмент кода ради краткости):

```
var locationListCtrl = function ($scope, loc8rData, geolocation) { ←
  // Код контроллера
};
```

Добавляем имя геолокационного сервиса в принимаемые функцией контроллера параметры

Такой подход вам, наверное, уже кажется знакомым. Следующий этап — заставить контроллер взаимодействовать с сервисом.

Использование геолокационного сервиса из контроллера

Теперь у контроллера имеется доступ к геолокационному сервису, так давайте воспользуемся им! Обратиться к нему очень просто: нужно всего лишь вызвать метод `geolocation.getPosition` и передать ему три функции обратного вызова, которые мы хотим использовать.

В контроллере создадим три функции, по одной для каждого из возможных исходов:

- ❑ успешная попытка геолокации;
- ❑ неудачная попытка геолокации;
- ❑ геолокация не поддерживается.

Мы также поменяем отображаемые для пользователей сообщения, чтобы пользователи понимали, что система что-то делает. Это особенно важно сейчас, поскольку геолокация занимает 1–2 секунды.

Листинг 8.16 демонстрирует вид контроллера после выполнения всех этих изменений, создавая новые функции и вызывая геолокационный сервис.

Листинг 8.16. Модификация контроллера для использования нового геолокационного сервиса

```

1  Задаем сообщение
   по умолчанию

var locationListCtrl = function ($scope, loc8rData, geolocation) {
  → $scope.message = "Checking your location";

  $scope.getData = function (position) {
    $scope.message = "Searching for nearby places";
    loc8rData
      .success(function(data) {
        $scope.message = data.length > 0 ? "" :
          "No locations found";
        $scope.data = { locations: data };
      })
      .error(function (e) {
        $scope.message = "Sorry, something's gone wrong";
      });
  });

  $scope.showError = function (error) {
    $scope.$apply(function() {
      $scope.message = error.message;
    });
  });

  $scope.noGeo = function () {
    $scope.$apply(function() {
      $scope.message = "Geolocation not supported
        by this browser.";
    });
  });

  geolocation.getPosition($scope.getData,$scope.showError,$scope.noGeo); ←
};

```

2
Функция для запуска при успешной геолокации

3
Функция для запуска в случае, когда геолокация поддерживается, но завершилась неудачей

4
Функция для запуска в случае, когда геолокация не поддерживается браузером

5
Передаем функцию геолокационному сервису

Первое, что мы тут делаем, — сообщаем пользователям, задавая сообщение по умолчанию **1**, что мы выполняем поиск их местоположения. Затем создаем функцию **2** для запуска в случае, когда геолокация завершилась успешно. Нативный

геолокационный API передает этой функции обратного вызова объект `position`. Эта функция затем обратится к сервису `loc8rData` для получения списка местоположений.

Далее идет функция ③, запускаемая в случае, когда геолокация поддерживается, но завершилась неудачно. Нативный геолокационный API передает обратному вызову объект `error`, содержащий свойство `message`, которое можно вывести для пользователя. Обратите внимание на адаптер `$scope.$apply()` и загляните во врезку, чтобы понять, что он собой представляет и зачем здесь нужен.

Функция `поGeo` ④ запускается, если геолокация не поддерживается браузером, и задает сообщение для отображения пользователю. И вновь обратите внимание на использование `$scope.$apply()`.

И наконец, мы вызываем геолокационный сервис ⑤, передавая три функции обратного вызова в качестве параметров.

ОТНОСИТЕЛЬНО `$SCOPE.$APPLY()`

Во фрагменте кода, работающем с функциональностью геолокации, мы использовали `$scope.$apply()` в двух из трех функций обратного вызова после изменения значения в области видимости. Если бы мы этого не сделали, то сообщения не были бы отображены в браузере. Но почему так и почему только две функции обратного вызова из трех?

`$scope.$apply()` используется, когда Angular может быть неизвестно о каких-то изменениях в области видимости. Обычно так бывает после асинхронного события, такого как обратный вызов или действие пользователя. Цель этой функции — передавать изменения в области видимости представлению.

На самом деле `$scope.$apply()` часто используется в Angular, но обычно это незаметно. Мы не используем его в случае обратного вызова при успешном завершении, поскольку в этом случае изменения в области видимости выполняются внутри возвращаемых промисов сервиса `$http`. Неявным образом Angular обертывает эти промисы в функцию `$scope.$apply()`, так что вам этого делать не надо.

Если вы перезагрузите приложение в браузере, то получите предложение поделиться информацией о своем местоположении, которое вы можете принять или отклонить. Если вы запретите своему браузеру получать доступ к информации о вашем местоположении, будет вызвана функция обратного вызова `$scope.showError`, а на экране отображено сообщение об ошибке. Как это выглядит в браузере Chrome, демонстрирует рис. 8.9.

Если вы ответите «Нет», чтобы протестировать это сообщение об ошибке, очень вероятно, что браузер запомнит эту настройку. Если это произойдет, вы сможете поменять пользовательские настройки своего браузера, чтобы проверить, что произойдет, если вы ответите «Да».

Если вы все же ответите «Да», приложение вызовет наш информационный сервис для получения данных из API. Звучит неплохо, но мы упустили один

шаг: пока что не передали координаты информационному сервису, так что он по-прежнему будет использовать жестко зашитые значения `lat` и `lng`. Давайте исправим этот недочет.

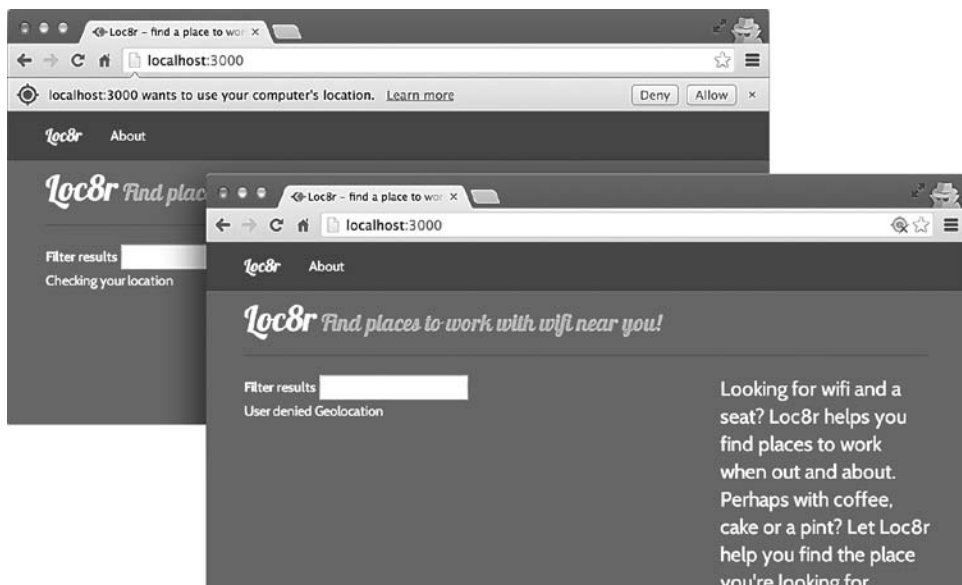


Рис. 8.9. Когда добавлена геолокация, Loc8r будет просить разрешения узнать ваше местоположение. Если вы ответите «Нет», будет отображено простое сообщение

Передаем геолокационные координаты информационному сервису

Мы почти добились желаемого. Геолокационный сервис возвращает координаты приложению Angular, так что нужно только передать их нашему информационно-му сервису, чтобы можно было добавить их в обращение к API.

Первое, что необходимо сделать, — модифицировать сервис `loc8rData`, обернув обращение к API в функцию, чтобы можно было передать данные. Мы не можем передать параметры в саму функцию `loc8rData`, поскольку это конструктор сервиса. В листинге 8.17 сделаем так, чтобы новая вложенная функция принимала два параметра, `lat` и `lng`, и модифицируем вызов API для их использования.

Листинг 8.17. Меняем сервис `loc8rData` для возврата функции вместо данных

```
var loc8rData = function ($http) {
  var locationByCoords = function (lat, lng) {
```

Создаем новую функцию внутри функции сервиса, принимающую два параметра: `lat` и `lng`

```

return $http.get('/api/locations?lng=' + lng + '&lat=' +
    lat + '&maxDistance=20');
};
return {
    locationByCoords : locationByCoords
};
};

```

Исключаем жестко зашитые значения из вызова API и заменяем их переменными lat и lng

Возвращаем функцию locationByCoords, делая ее доступной в качестве метода сервиса

Теперь у сервиса `loc8rData` имеется метод `locationByCoords`, к которому может обратиться контроллер, передав ему координаты. Теперь исправим контроллер.

Как мы видели, API геолокации возвращает функции обратного вызова параметр `position` для случая успешного завершения, в данном случае `$scope.getData.position` — объект JavaScript, содержащий различные элементы данных, в том числе путь `coords`, в котором находятся значения широты и долготы.

Листинг 8.18 демонстрирует, как нужно поменять функцию `$scope.getData` в контроллере, модифицировав обращение к информационному сервису для использования нового метода и передачи параметров.

Листинг 8.18. Модификация контроллера для передачи сервисам координат

```

$scope.getData = function (position) {
    var lat = position.coords.latitude,
        lng = position.coords.longitude;

```

Описываем переменные для значений широты и долготы из объекта position

```

    $scope.message = "Searching for nearby places";
    loc8rData.locationByCoords(lat, lng)
        .success(function(data) {
            $scope.message = data.length > 0 ? "" : "No locations found";
            $scope.data = { locations: data };
        })
        .error(function (e) {
            $scope.message = "Sorry, something's gone wrong";
        });
};

```

Вместо того чтобы просто обращаться к сервису loc8rData по названию, меняем код, вызывая новый метод locationByCoords и передавая ему переменные lat и lng

Это был последний фрагмент пазла. Теперь `Loc8r` определяет ваше текущее местоположение и выводит список находящихся рядом с вами мест, что и было нашей первоначальной задумкой. Имеется также возможность фильтрации результатов на домашней странице с помощью текстового ввода.

Вид домашней страницы в настоящее время демонстрирует рис. 8.10. Она не слишком изменилась по сравнению с тем, какой была в начале данной главы, но основная область ее контента теперь представляет собой модуль Angular.

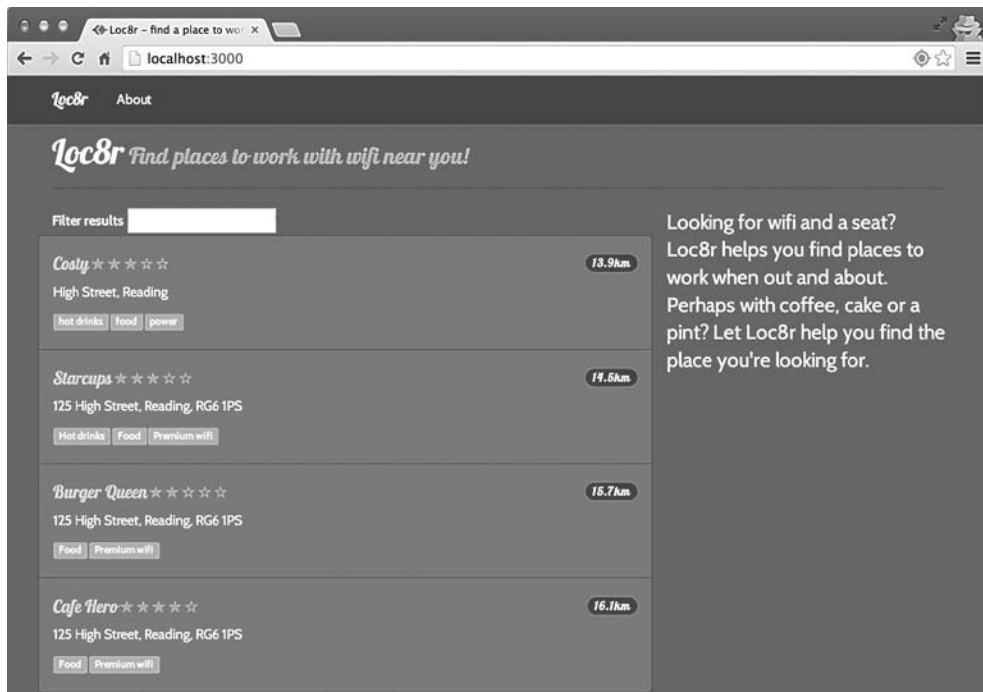


Рис. 8.10. Единственное внешнее отличие домашней страницы от ее вида в начале главы — появление текстового поля для фильтрации. Но за кулисами основную область контента теперь генерирует Angular, который определяет также ваше фактическое местоположение и отображает находящиеся рядом объекты

Вот и все. Почти. Радостно добавляя Angular на домашнюю страницу, мы между делом внесли ошибку в страницу Add Review (Добавить отзыв). Сейчас увидим, что это за ошибка, и разберемся, как ее исправить.

8.4. Обеспечиваем надлежащую работу форм

Добавив Angular в шаблон сайта в файле `layout.jade`, мы нечаянно создали ошибку в форме добавления отзыва. При создании формы отзыва мы оставили `action` незаполненным, чтобы форма отправлялась на текущий URL. Это именно то, чего мы хотели, и до сих пор оно работало отлично. Так в чем же проблема?

Когда Angular наталкивается на форму без действия, он блокирует выполнение ее отправки. Во многом это имеет смысл. Если вы создаете приложение Angular и включаете в него форму, вероятно, хотели бы, чтобы Angular обрабатывал ее

и управлял отправкой и адресом назначения. Вряд ли вам бы хотелось каждый раз вручную предотвращать отправку, так что Angular делает это за вас.

Это очень удобно всегда, за исключением той ситуации, в которой мы очутились сейчас. Исправить ее можно добавлением в форму какого-либо действия, что предотвратит вмешательство Angular в ее работу.

Первый шаг — передать Jade текущий URL из контроллера. Простейший способ получить точное представление URL — с помощью свойства `originalUrl` объекта `req`. Листинг 8.19 демонстрирует изменения, которые необходимо сделать в функции `renderReviewForm` в файле `locations.js` из каталога `app_server/controllers`.

Листинг 8.19. Передаем URL из контроллера формы отзыва

```
var renderReviewForm = function (req, res, locDetail) {
  res.render('location-review-form', {
    title: 'Review ' + locDetail.name + ' on Loc8r',
    pageHeader: { title: 'Review ' + locDetail.name },
    error: req.query.err,
    url: req.originalUrl
  });
};
```

Второй шаг — просто вывести параметр `url` в атрибут `action` описания формы в файле `location-review-form.jade` в каталоге `app_server/views`. Это показано в следующем фрагменте кода:

```
form.form-horizontal(action="#{url}", method="post", role="form")
```

После этих двух маленьких изменений форма будет работать как и раньше, а Angular станет ее игнорировать.

8.5. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Простую и двустороннюю привязку данных.
- ❑ Рекомендуемые решения по использованию геттер- и сеттер-синтаксиса для описания модулей и их компонентов.
- ❑ Описание контроллеров в коде и HTML для управления приложением.
- ❑ Функционирование областей видимости в Angular, в частности `rootScope` и области видимости уровня контроллеров.
- ❑ Создание сниппетов HTML с помощью директив.

- ❑ Использование фильтров для модификации выводимого текста, а также повтор целых элементов HTML.
- ❑ Использование сервисов для добавления допускающих многократное использование элементов функциональности и для работы с данными.
- ❑ Сервис `$http` для выполнения асинхронных обращений к API.
- ❑ Когда и почему следует использовать `$scope.$apply()`.
- ❑ Работу с API геолокации HTML5.

В главе 9 мы планируем начать преобразовывать приложение `Loc8r` в целом работающее на Angular одностраничное приложение. Так что вдохните поглубже и готовьтесь — это будет что-то!

Глава 9

Создание одностраничного приложения с помощью Angular: фундамент

В этой главе:

- ❑ настройка Express для выдачи одностраничного приложения;
- ❑ рекомендуемые решения по организации кода в большом приложении Angular;
- ❑ использование Angular вместо Express для выполнения маршрутизации URL;
- ❑ сокращение файлов Angular и объединение нескольких файлов в один.

В главе 8 мы рассмотрели, как использовать Angular для добавления компонента на существующую страницу. В следующих двух главах собираемся вывести Angular на новый уровень и использовать его для создания одностраничного приложения. Это значит, что вместо выполнения логики приложения на сервере с помощью Express мы будем делать все в браузере с помощью Angular. К концу данной главы у нас будет готов каркас для SPA и первая его часть будет работать, используя Angular для выполнения маршрутизации к домашней странице и отображения контента.

Наше место в общем плане книги — пересоздание основного приложения в виде SPA Angular — показывает рис. 9.1.

При обычном процессе разработки вы вряд ли создавали бы целое приложение на сервере, чтобы затем пересоздать его в качестве SPA. В идеале еще на ранних стадиях планирования вы должны определиться, нужно ли вам SPA, и начать работу с применением соответствующей технологии. Но для обучения, которым мы

заняты, это неплохой подход: мы уже знакомы с функциональностью сайта, макеты созданы. Это позволит нам сосредоточиться на более захватывающей перспективе — увидеть создание полного приложения Angular.

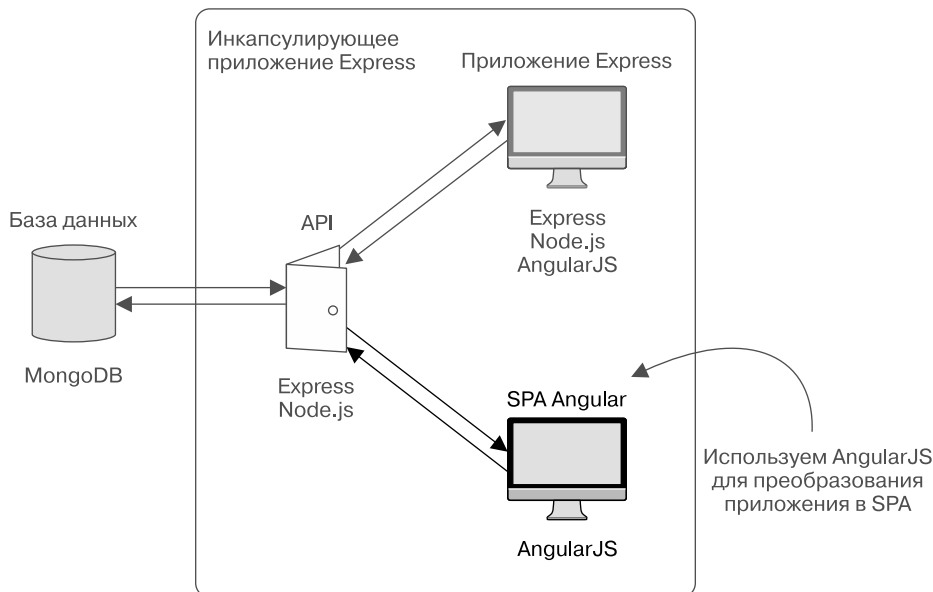


Рис. 9.1. В этой главе мы пересоздадим приложение `Loc8r` в виде SPA Angular, переместив логику приложения из серверной части в клиентскую

Начнем с модификации инкапсулирующего приложения Express с целью включения клиентского приложения до того, как создать домашнюю страницу. По мере того как мы будем добавлять контроллеры, сервисы, фильтры и директивы, мы изучим рекомендуемые решения, такие как защита глобальной области видимости, создание допускающих повторное использование компонентов и сокращение кода для улучшения производительности браузера.

9.1. Подготовительные работы для Angular SPA

В этом разделе настроим некоторые вещи так, чтобы можно было остаток главы посвятить созданию функциональности SPA. В главе 8 мы собрали весь код Angular для компонентов домашней страницы в один файл. Это подходит для относительно небольших единичных компонентов, но вы, вероятно, заметили, что у нас уже около 140 строк и стало уже сложновато работать с файлом и находить в нем нужный фрагмент кода.

Исходя из соображений управляемости, качества кода и возможности повторного использования, рекомендуемым решением для создания SPA Angular будет один файл для одного элемента функциональности. Так что у каждого контроллера, сервиса, директивы и фильтра должен быть собственный файл. Именно такой подход мы будем использовать, что означает необходимость хорошей организации файлов.

Подготовка базовых файлов. Необходимо организовать и разместить базовые файлы нашего приложения в правильных местах, чтобы можно было легко его компоновать. Начнем с создания для кода SPA нового каталога в корневом каталоге приложения.

9.1.1. Создание каталога `app_client` для приложения на стороне клиента

Самое первое, что необходимо сделать, — создать место внутри основного приложения Express, где будет находиться SPA Angular. У нас уже есть соглашение об именах для разделения основных частей приложения с каталогами `app_server` и `app_api`. Дополнительно создадим каталог `app_client` в корневом каталоге приложения.

Именно тут мы будем хранить весь код приложения для SPA Angular. Хотелось бы, чтобы помещенный в этот каталог код JavaScript передавался для выполнения браузеру, а не запускался на сервере. Для этого понадобится сообщить Express, что данный каталог содержит статическое содержимое, которое при запросе необходимо выдавать браузеру в неизменном виде.

Как вы можете помнить из ранее изложенного в данной книге, эта возможность настраивается в основном файле Express `app.js`. Там уже указано, что каталог `public` — статический. В следующем фрагменте кода продублируем эту строку в файле `app.js`, указав в качестве имени каталога `app_client`:

```
app.use(express.static(path.join(__dirname, 'public')));  
app.use(express.static(path.join(__dirname, 'app_client')));
```

Теперь при запросе браузером ресурса из нашего нового каталога он будет выдаваться в неизменном виде.

9.1.2. Создание основного файла приложения SPA

В каталоге `app_client` у нас будет находиться основной файл приложения. В главе 8 мы научились пользоваться сеттер- и геттер-синтаксисом, и именно при подобном подходе с применением распределенных файлов такой синтаксис по праву занимает свое место. В основном файле должен находиться единственный сеттер модуля для нашего приложения.

Создайте в каталоге `app_client` новый файл `app.js` и добавьте в него приведенный ниже сеттер модуля Angular:


```
angular.module('loc8rApp', []);
```

Теперь, когда сеттер модуля готов, мы сможем добавлять контроллеры, фильтры и многое другое в приложение посредством использования геттер-синтаксиса для модулей.

9.1.3. Добавление основного файла приложения SPA в макет Jade

В браузере будет использоваться приложение Angular, так что нам больше не нужен компонент домашней страницы, созданный в главе 8. Поэтому можно изменить ссылку в файле `layout.jade` (в каталоге `app_server/views`) следующим образом:

```
script(src='/angular/angular.min.js')
script(src='/app.js')
```



Меняем существующую ссылку на `angular/loc8rApp.js` на `/app.js`

Вот и все, что касается первой части нашего фундамента. У нас уже есть куда поместить код приложения, и мы создали сеттер модуля и добавили его в макет Jade, чтобы он выдавался браузеру.

Но если вы сейчас попытаетесь запустить получившееся, то увидите в браузере ошибку JavaScript, так как он все еще пытается использовать шаблон Jade для домашней страницы, но без подключенного к нему компонента Angular из главы 8.

Нам понадобится создать новый код приложения для работы домашней страницы, но сначала нужно, чтобы Angular взял на себя управление маршрутизацией.

9.2. Переключение с Express-маршрутизации на Angular-маршрутизацию

Один из принципов SPA заключается в том, что страница не перезагружается полностью при переходе посетителя с одного экрана на другой. Браузер не должен выполнять запрос к серверу всякий раз, когда пользователь делает щелчок кнопкой мыши, чтобы увидеть другую страницу. Но мы также хотели бы, чтобы пользователи могли перейти на любую страницу напрямую и сразу увидеть нужный контент.

Пока что при посещении пользователем одного из URL нашего сайта браузер отправляет запрос серверу, который обрабатывает запрос перед отправкой браузеру HTML-кода для страницы (см. иллюстрацию этого процесса на схеме 1 рис. 9.2). Это происходит при каждом новом запросе страницы во время нахождения пользователя на сайте, причем сервер всякий раз повторно отправляет всю страницу.

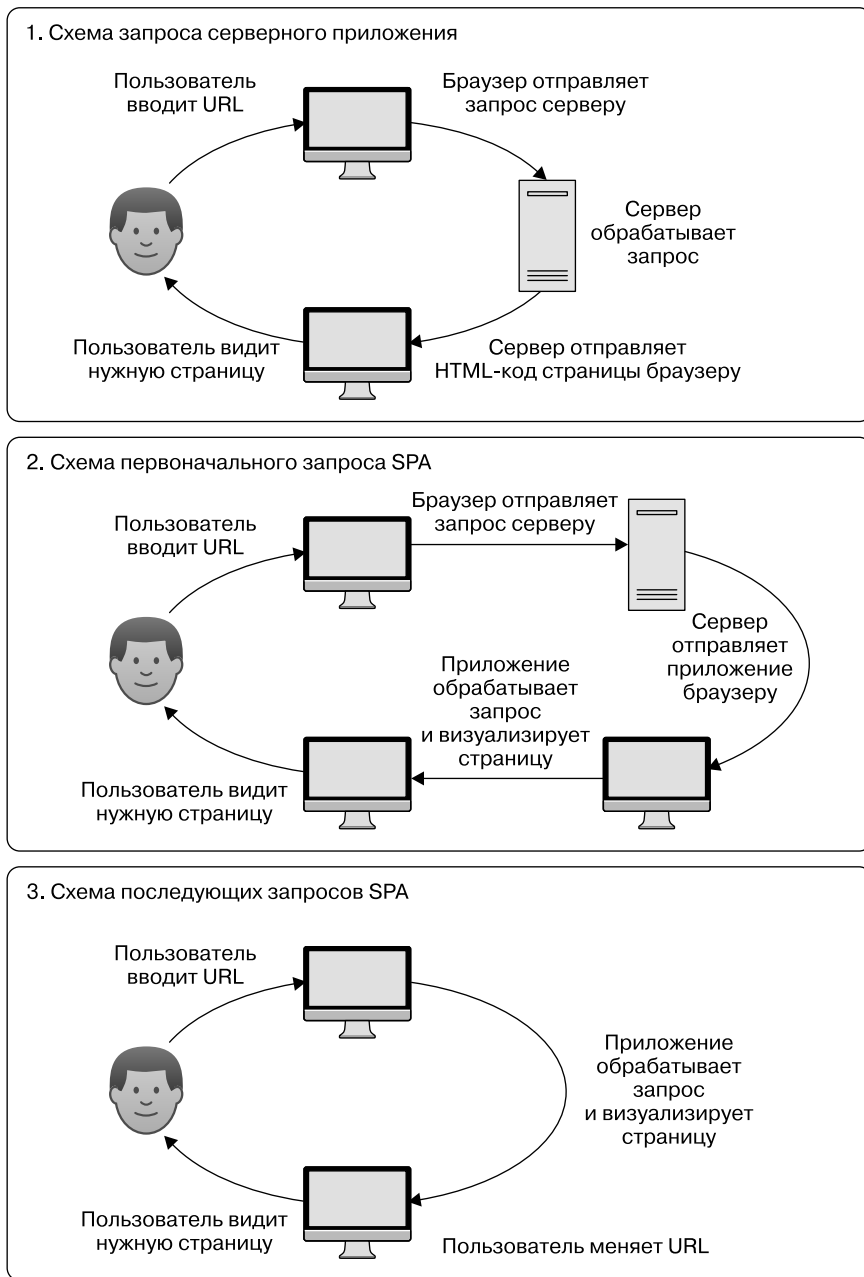


Рис. 9.2. Три схемы для различных подходов. Серверные приложения обращаются к серверу при каждом новом запросе страницы, но SPA обычно обращаются к серверу только для первоначальной загрузки страницы. Последующие запросы обрабатываются самим SPA

В SPA, когда посетитель впервые переходит по одному из URL на сайте, браузер отправляет запрос серверу. Как показывает схема 2 на рис. 9.2, сервер затем возвращает все приложение браузеру и именно это клиентское приложение обрабатывает запрос перед отображением пользователю нужной страницы. Когда пользователь щелкает для перехода на другую страницу сайта, приложение в браузере обрабатывает запрос и меняет видимый пользователем контент, а не выполняет еще один запрос к серверу. Этот более короткий процесс показан на схеме 3 рис. 9.2.

Самое интересное в этом для нас в данный момент: откуда браузер знает, что именно отображать для различных URL. В Express мы настраивали набор маршрутов, указывающих на различные контроллеры. В Angular можем сделать нечто подобное, но сначала необходимо предотвратить перехват Express управления.

9.2.1. Отключение маршрутизации Express

Как показано на рис. 9.2, каждая запрошенная у приложения страница маршрутизируется через Express. Поступает запрос URL, и Express определяет, какой контроллер должен использоваться для его обработки. Мы собираемся применить другой подход, при котором Express будет применяться для выдачи приложения Angular и включающей его страницы, а Angular — выполнять остальную маршрутизацию.

Чтобы забрать у Express управление маршрутизацией, сделаем две вещи:

- создадим новый контроллер для выдачи базового шаблона браузеру;
- изменим маршрут Express домашней страницы для использования нового контроллера.

Создаем новый контроллер Express для выдачи шаблона страницы

Прежде чем менять основной маршрут Express, создадим для него новый контроллер. Вопрос в следующем: что мы хотели бы отправлять браузеру? Мы хотели бы, чтобы в нашем SPA навигационная панель и нижний колонтитул были одинаковыми на всех страницах, а контент в центре менялся в зависимости от действий пользователя. В нашем шаблоне `layout.jade` содержатся заголовок, нижний колонтитул и заглушка для контента в центре. Похоже, это отличный кандидат на многократное использование.

В следующем фрагменте кода показан новый контроллер, предназначенный для добавления в конец файла `others.js` в каталоге `app_server/controllers`:

```
module.exports.angularApp = function(req, res){
  res.render('layout', { title: 'Loc8r' });
};
```

При обращении к этому контроллеру он будет просто отправлять макет браузеру. Так что теперь нам нужно организовать его вызов.

Меняем маршрутизацию домашней страницы

Маршрут для домашней страницы находится в файле `index` каталога `app_server/routes`, и теперь необходимо его поменять, чтобы использовался новый контроллер. Следующий фрагмент кода демонстрирует необходимые изменения в маршрутизации:

```
router.get('/', ctrlOthers.angularApp); ←  
router.get('/location/:locationid', ctrlLocations.locationInfo);  
router.get('/location/:locationid/review/new', ctrlLocations.addReview);  
router.post('/location/:locationid/review/new', ctrlLocations.doAddReview);
```

Задаем такой маршрут домашней страницы, чтобы использовался новый контроллер

КАК НАСЧЕТ ОСТАЛЬНЫХ МАРШРУТОВ EXPRESS? _____

Остальные маршруты Express фактически стали лишними и не будут использоваться. Можете удалить или закомментировать их, если хотите. Если их оставить, это тоже не вызовет конфликтов с маршрутизацией Angular, поскольку все пути в Angular будут немного различаться.

Чтобы избежать перезагрузок страниц, Angular по умолчанию меняет путь URL после #, который обычно является анкером страницы. Первоначальное назначение # в URL заключалось в перемещении посетителя в определенное место на длинной странице, Angular использует его для перемещения посетителя в определенное место приложения.

В версии Express путь URL для страницы About (О нас) выглядит следующим образом:
`/about`

После настройки маршрутизации Angular путь URL для той же страницы будет выглядеть вот так:

```
}/#{about
```

Мы обсудим это позднее, в главе 10, когда будем рассматривать удаление # из различных URL Angular для улучшения внешнего связывания.

Теперь при посещении домашней страницы вы увидите приятную пустую страницу, как показано на рис. 9.3, на которой будут только навигационная панель и нижний колонтитул (из-за отсутствия контента нижний колонтитул сейчас спрятан за навигационной панелью).

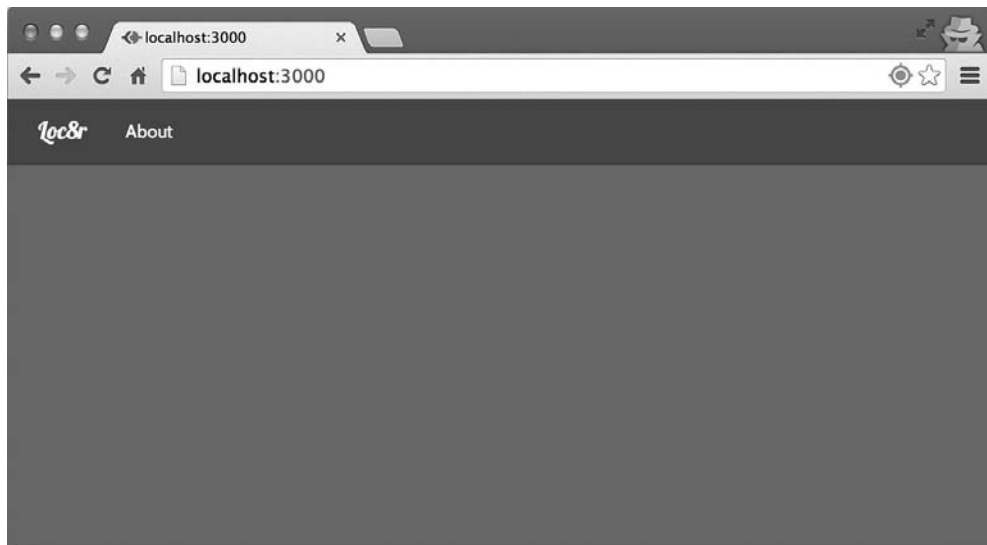


Рис. 9.3. При визуализации с помощью Express в браузере только шаблона `layout.jade` домашняя страница выглядит довольно пустой и готовой для кода Angular

Перейдем к следующему этапу и добавим маршрутизацию Angular.

9.2.2. Добавляем в приложение `ngRoute` (`angular-route`)

В более старых версиях Angular маршрутизация была встроена в основную библиотеку, но потом ее сделали внешней зависимостью, чтобы можно было сопровождать ее отдельно. Так что необходимо скачать ее и добавить в наше приложение.

Так как это часть базовой функциональности, соответствующий файл можно скачать с сайта code.angularjs.org. На этом сайте имеется каталог для каждого выпуска Angular, так что вы наверняка сможете скачать нужную версию. Мы используем Angular 1.2.19, так что перейдите по адресу <http://code.angularjs.org/1.2.19/>, чтобы увидеть список файлов для скачивания.

Нужный нам модуль называется `angular-route`, поэтому скачайте два минифицированных файла:

- ❑ `angular-route.min.js`;
- ❑ `angular-route.min.js.map`.

Теперь пришло время добавить файлы в приложение.

Добавляем файлы angular-route в приложение

Переместим эти файлы библиотек в новый каталог `app_client`, чтобы браузеры могли использовать их как статические файлы. Чтобы все было аккуратно, создайте новый каталог `lib` внутри `app_client` и поместите туда два файла `angular-route`. Это даст возможность отделить файлы библиотек от кода приложения.

Конечно, браузеру необходимо знать, как обратиться к файлам JavaScript, так что нам опять нужно добавить ссылку на файл в `layout.jade`, как показано в следующем фрагменте кода:

```
script(src='/angular/angular.min.js')
script(src='/lib/angular-route.min.js')
script(src='/app.js')
```

Теперь браузер знает о файле JavaScript и загрузит его. Этот файл, в свою очередь, ссылается на файл `.map` для отображения при необходимости более осмысленных сообщений об ошибках. Все это хорошо, но наше приложение Angular не знает, что мы хотим его использовать, так что давайте разберемся с этим сейчас.

Добавление angular-route в приложение Angular

Чтобы добавить `angular-route` в наше приложение Angular, необходимо добавить его в качестве зависимости в описание модуля. Название файла должно быть `angular-route`, но имя модуля — `ngRoute`, так что именно эту зависимость нужно добавить. Этим модулем генерируется провайдер `$routeProvider`, который можно передать функции `config` фреймворка Angular. Именно в функции `config` будем описывать различные маршруты для нашего приложения.

Все вместе это показано в листинге 9.1, демонстрирующем полностью модифицированное содержимое файла `app_client/app.js`.

Листинг 9.1. Добавляем `ngRoute` и `config` в приложение Angular

```
angular.module('loc8rApp', ['ngRoute']);
function config($routeProvider) {
  $routeProvider
    .when('/', {
    })
    .otherwise({redirectTo: '/'});
}
angular
  .module('loc8rApp')
  .config(['$routeProvider', config]);
```

1 Добавляем `ngRoute` в виде зависимости модуля

2 Функция модуля `config`, содержащая описание маршрутов

3 Добавляем `config` в модуль, передавая `$routeProvider` в качестве зависимости

Из-за синтаксиса Angular, а также использования внедрения зависимостей, модулей и провайдеров этот код может выглядеть более сложным, чем есть на самом деле. Нашему модулю необходимо воспользоваться зависимостью `ngRoute`, чтобы сделать возможной маршрутизацию ❶, а функция `config` ❷ — то место, где мы описываем маршруты. Модулю ❸ необходимо сообщить о функции `config` с помощью геттер-синтаксиса модуля.

В нынешнем состоянии маршрутизация делает немного, но синтаксис вполне понятен. При обращении по пути URL `'/'` — это домашняя страница — она не будет делать ничего, при обращении по другому URL она перенаправит обращение на домашнюю страницу. По мере продвижения по данной главе мы добавим в эту конфигурацию дополнительные разделы `.when`, чтобы отображать различные страницы. Но сначала добьемся, чтобы домашняя страница что-то делала.

9.3. Добавление первых представлений, контроллеров и сервисов

В этом разделе мы собираемся восстановить функциональность домашней страницы. Для этого нам понадобится создать представление и контроллер Angular, а также воспользоваться созданными в главе 8 сервисами для геолокации и обращения к API данных.

В разделе 9.1 мы обсуждали разделение функциональности по отдельным файлам. Именно это мы сейчас и сделаем. Каждое представление, контроллер и сервис должны находиться в отдельном файле, и эти файлы должны быть сгруппированы по каталогам разумным образом. Это означает их группировку по каталогам на основе функциональности, а не типа файлов. Например, контроллер и представление для домашней страницы тесно связаны друг с другом, так что мы поместим их в один каталог.

РЕКОМЕНДУЕМОЕ РЕШЕНИЕ

Все представления, контроллеры, сервисы и фильтры обязательно должны находиться в отдельных файлах. У этих файлов должна быть единообразная схема именования, и они должны быть сгруппированы по каталогам в соответствии со своей функциональностью.

С учетом этого начнем с создания подкаталога `home` в каталоге `app_client`. Он послужит местом хранения файлов нашей домашней страницы, начиная с представления.

9.3.1. Создаем представление Angular

Мы собираемся создать HTML-шаблон для контента домашней страницы, связать его с маршрутизацией Angular и отобразить на странице. Перейдем прямо к делу и создадим представление.

Создание шаблона представления

Мы уже знаем, как должна выглядеть и вести себя домашняя страница, и у нас уже имеется для нее шаблон, включающий определенные привязки Angular. Этот шаблон создан на Jade, но теперь нужно преобразовать его в HTML, не забывая заменить все привязки Jade их аналогами Angular. Создайте в подкаталоге `home` каталога `app_client` новый файл `home.view.html` для представления.

Листинг 9.2 демонстрирует содержимое файла `home.view.html` после его преобразования из Jade. Обратите внимание на то, что мы пока убираем фильтр `formatDistance` из элемента для расстояний до объектов, чтобы избежать ошибок. Мы вернем его обратно позднее.

Листинг 9.2. Шаблон представления Angular для домашней страницы `home.view.html`

```
<div id="banner" class="page-header">
  <div class="row">
    <div class="col-lg-6"></div>
    <h1>
      {{ pageHeader.title }}
      <small>{{ pageHeader.strapline }}</small>
    </h1>
  </div>
</div>
<div class="row">
  <div class="col-xs-12 col-sm-8">
    <label for="filter">Filter results</label>
    <input id="filter" type="text", name="filter", ng-model="textFilter">
    <div class="error">{{ message }}</div>
    <div class="row list-group">
      <div class="col-xs-12 list-group-item" ng-repeat="location in
        data.locations | filter : textFilter">
        <h4>
```

Меняем привязки для `pageHeader` на привязки Angular

```

    <a href="/location/{{ location._id }}">{{ location.name }}</a>
    <small class="rating-stars" rating-stars
      rating="location.rating"></small>
    <span class="badge pull-right badge-default">{{ location.distance
  }}</span>
</h4>
<p class="address">{{ location.address }}</p>
<p>
  <span class="label label-warning label-facility"
    ng-repeat="facility in location.facilities">
    {{ facility }}
  </span>
</p>
</div>
</div>
</div>
</div>
<div class="col-xs-12 col-sm-4">
  <p class="lead">{{ sidebar.content }}</p>
</div>
</div>

```

Временно убираем
фильтр formatDistance

Делаем содержимое
боковой панели
привязкой Angular

Этот HTML-код не делает ничего, что мы бы еще не видели, и при подключении данных вернемся к нашей обычной домашней странице. Следующий этап — сообщить модулю Angular о необходимости использовать это представление для домашней страницы.

Назначение представления маршруту

Angular необходимо знать о существовании нового шаблона представления, а также о том, когда его использовать. Для этого мы вернемся к функции `config` нашего маршрута в `app_client/app.js`. Зададим в операторе `when` для пути домашней страницы параметр `templateUrl`, указывающий на новый HTML-файл, вот так:

```

function config ($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'home/home.view.html'
    })
    .otherwise({redirectTo: '/'});
}

```

Добавляем templateUrl
в конфигурацию
маршрута для назначения
используемого шаблона
представления

Теперь Angular будет использовать шаблон `home.view.html` для пути URL `'/'`. Осталась только одна проблема: мы не сообщили Angular, где отображать этот шаблон в браузере. Ничего страшного, это всего лишь маленькое дополнение к базовому шаблону HTML.

Задаем место отображения представления Angular

В главе 8, когда мы делали некоторые элементы HTML внешними, мы включили их обратно в приложение в виде директив. Тут мы собираемся сделать то же самое, используя прилагаемую к `ngRoute` директиву. Эта директива называется `ng-view` и используется Angular в качестве контейнера, в котором можно менять представления.

Использовать его очень просто. У нас уже есть `.container` в файле `layout.jade`, в котором сейчас находится оператор `block content`, предназначенный для внедрения контента Jade для отдельных страниц. Следующий фрагмент кода демонстрирует добавление нового раздела `div` с `ng-view` в качестве атрибута внутри `.container` для использования Angular:

```
.container
  div(ng-view) ← | Добавляем пустой div в качестве контейнера ng-view,
  block content   | в который Angular будет вставлять представления
```

Теперь можно перейти в браузер и посмотреть, что у нас получилось. Немного, но больше, чем раньше! В представлении пока что нет данных, но по крайней мере можно увидеть являющееся частью представления поле для ввода `Filter Results` (Отфильтровать результаты).

Теперь обеспечим наличие данных, добавив контроллер.

9.3.2. Добавляем контроллер к маршруту

Процесс добавления контроллера к маршруту также не особо сложен. Он состоит из нескольких шагов.

1. Создаем контроллер в отдельном файле.
2. Привязываем контроллер к приложению Angular.
3. Сообщаем функции маршрута `config`, когда использовать контроллер.
4. Сообщаем браузеру о файле.

Начнем с создания нового файла и контроллера.

Создание контроллера

В том же каталоге, где находится файл `home.view.html`, создаем новый файл `home.controller.js` — соглашения о наименованиях очень просты! В этом файле воспользуемся геттер-синтаксисом модуля для добавления в приложение нового контроллера и опишем простой контроллер для добавления в область видимости данных таким образом, чтобы видеть заголовок страницы и контент боковой панели в браузере.

Этот шаг не является для нас чем-то новым, так что посмотрим на содержимое нового файла контроллера в листинге 9.3.

Листинг 9.3. Создание нового контроллера домашней страницы

```
angular
  .module('loc8rApp')
  .controller('homeCtrl', homeCtrl);

function homeCtrl ($scope) {
  $scope.pageHeader = {
    title: 'Loc8r',
    strapline: 'Find places to work
               with wifi near you!'
  };
  $scope.sidebar = {
    content: "Looking for wifi and a seat etc etc"
  };
}
```

Используем геттер для модуля для добавления в приложение нового контроллера

Описываем новый контроллер homeCtrl и привязываем данные для заголовка страницы и боковой панели

Новый контроллер homeCtrl просто привязывает несколько элементов данных, которые мы привыкли отправлять Jade из контроллеров Express, но технически после главы 8 ничего нового для нас тут нет. При создании приложения с использованием отдельных файлов становится очевидным колоссальное преимущество использования геттер/сеттер-синтаксиса для модулей. Каждый контроллер, сервис или фильтр можно добавить в приложение из собственного отдельного файла. Немалое преимущество этого заключается в том, что вам не приходится пытаться управлять всем списком зависимостей из основного файла app.js.

Добавление контроллера в функцию config маршрута

Функции config маршрута необходимо знать, какой контроллер использовать для какого пути. Это еще один несложный шаг, представляющий собой добавление в функцию config для пути домашней страницы параметра контроллера. В параметре контроллера название контроллера указывается в виде строки следующим образом:

```
function config ($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'home/home.view.html',
      controller: 'homeCtrl'
    })
    .otherwise({redirectTo: '/'});
}
```

Добавляем параметр контроллера в функцию config маршрута, передавая название контроллера в виде строки

Мы практически у цели, осталось только сообщить браузеру о файле, чтобы браузер мог его загрузить.

Сообщаем браузеру о файле контроллера

Никаких сюрпризов нас тут не ожидает — необходимо добавить файл сценария к `layout.jade`, чтобы браузер мог его загрузить, а Angular — использовать его содержимое. Следующий фрагмент кода демонстрирует изменения, которые нужно выполнить внизу файла `layout.jade` для добавления нового файла контроллера:

```
script(src='/angular/angular.min.js')
script(src='/lib/angular-route.min.js')
script(src='/app.js')
script(src='/home/home.controller.js')
```

Если вы теперь переключитесь на браузер и снова взглянете на домашнюю страницу, то увидите, что как заголовок, так и боковая панель отображают контент. Вы увидите что-то напоминающее скриншот, приведенный на рис. 9.4.

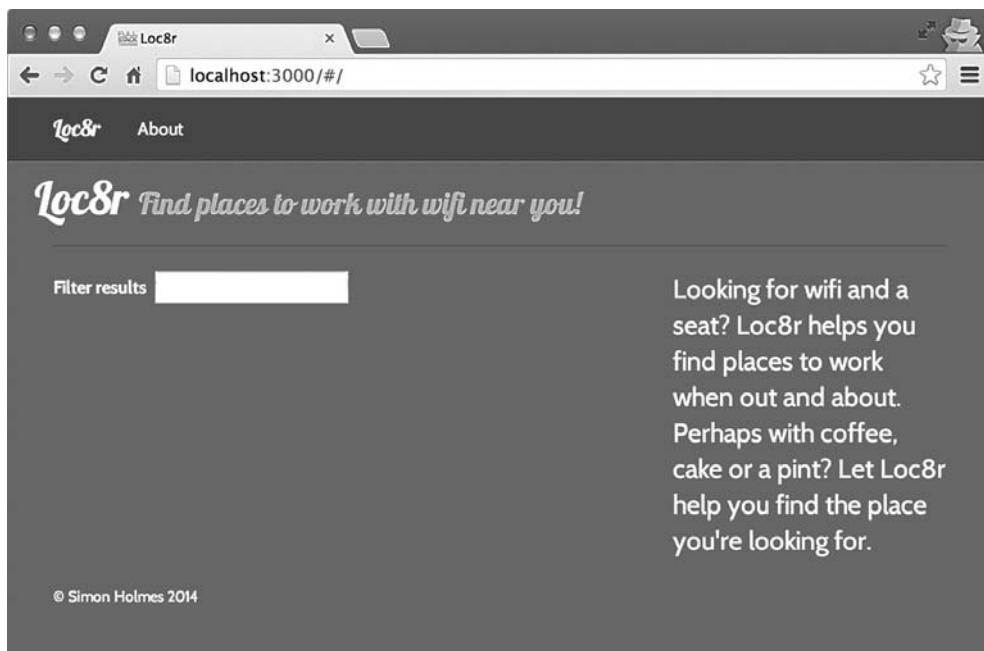


Рис. 9.4. Первый взгляд на наше SPA с заданием данных для заголовка домашней страницы и боковой панели в Angular-контроллере `homeCtrl`

Прежде чем продолжить, хотелось бы в порядке воплощения рекомендуемого решения несколько модифицировать контроллер.

9.3.3. Рекомендуемое решение для контроллера: использование синтаксиса `controllerAs`

Angular предоставляет способ для создания *модели представления*, к которой можно привязать свои данные, вместо того чтобы привязывать все непосредственно к объекту `$scope`. В следовании этому подходу есть несколько преимуществ. Во-первых, он заставляет вас обращаться с данными правильно, избегая вероятности присвоения значения непосредственно `$scope`. Во-вторых, он сохраняет чистоту объекта `$scope`. В-третьих, его можно использовать при необходимости публикации или подписки на события `$scope`. Желательно оставить `$scope` для случаев, когда он действительно необходим, например для работы со `$scope.$apply`.

Так как же это работает и как выглядит? Начнем с описания маршрута.

Объявление `controllerAs` в описании маршрута

Первое, что нужно сделать, — сообщить Angular о желании использовать данный контроллер с помощью синтаксиса `controllerAs`. Речь идет всего лишь о добавлении параметра в описание маршрута. Вам нужно указать параметр `controllerAs` и передать ему в виде строки название переменной `viewModel` (модель представления), которую вы хотели бы использовать.

Другое рекомендуемое решение — выбор стандартного имени. Обычно используют имя `vm`, расшифровывающееся как View Model (модель представления). Следующий фрагмент кода демонстрирует это в описании маршрута:

```
function config ($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'home/home.view.html',
      controller: 'homeCtrl',
      controllerAs: 'vm'
    })
    .otherwise({redirectTo: '/'});
}
```

Добавляем параметр `controllerAs` в описание маршрута, передавая имя переменной в виде строки

Мы будем использовать переменную `vm` как в контроллере, так и в представлении и разместим в ней все находящиеся сейчас в `$scope` объекты данных.

Описываем ViewModel в контроллере

При использовании контроллера в приложении он «за кулисами» генерируется с помощью JavaScript-метода `new`, создающего отдельный экземпляр. При использовании синтаксиса `controllerAs` Angular использует внутри функции `this`, привязывая его к `$scope`.

Использовать `this` в JavaScript может оказаться затруднительно из-за его сильной зависимости от контекста. Каждая вложенная функция будет иметь собственный `this`, а не наследовать его из родительской области видимости. Чтобы справиться с этим, просто объявите переменную сверху контроллера и привяжите ей `this`.

Ради единообразия и соблюдения рекомендуемого решения объявим переменную с названием `vm`, но она может называться как угодно. Название объявленной тут переменной не обязано совпадать с названием переменной, объявленной в описании маршрута.

При задании в объявлении переменной `vm` ее значения, равного `this`, можно поменять все привязки данных, заменив `$scope` на `vm`. После удаления всех ссылок на `$scope` можно удалить `$scope` и из описания функции. Модифицированный контроллер `home` выглядит следующим образом:

Листинг 9.4. Меняем контроллер `home`, чтобы использовать `vm` и синтаксис `controllerAs`

```
function homeCtrl () {
  ← Убираем $scope
  ← из описания функции
  var vm = this;
  vm.pageHeader = {
    title: 'Loc8r',
    strapline: 'Find places to work
                with wifi near you!'
  };
  vm.sidebar = {
    content: "Looking for wifi
              and a seat? Etc etc..."
  };
}
← Присваиваем this
  ← переменной vm
← Меняем привязки данных
  ← для использования vm
  ← вместо $scope
```

Используем ViewModel в представлении

Последний шаг в применении синтаксиса `controllerAs` заключается в использовании переменной `viewModel` в представлении. Данные из каждой привязки данных сейчас находятся в объекте `vm`, так что все привязки данных необходимо поменять. Все это в готовом виде вы можете увидеть в листинге 9.5.

Листинг 9.5. Меняем представление home для использования привязок данных vm

```

<div id="banner" class="page-header">
  <div class="row">
    <div class="col-lg-6"></div>
    <h1>
      {{ vm.pageHeader.title }}
      <small>{{ vm.pageHeader.strapline }}</small>
    </h1>
  </div>
</div>
<div class="row">
  <div class="col-xs-12 col-sm-8">
    <label for="filter">Filter results</label>
    <input id="filter" type="text", name="filter",
      ng-model="textFilter">
    <div class="error">{{ vm.message }}</div>#
    <div class="row list-group">
      <div class="col-xs-12 list-group-item"
        ng-repeat="location in vm.data.locations
          | filter : textFilter">
        <h4>
          <a href="/location/{{ location._id }}">{{
            location.name }}</a>
          <small class="rating-stars" rating-stars
            rating="location.rating"></small>
          <span class="badge pull-right badge-default">{{
            location.distance }}</span>
        </h4>
        <p class="address">{{ location.address }}</p>
        <p>
          <span class="label label-warning label-facility" ng-
            repeat="facility in location.facilities">
            {{ facility }}
          </span>
        </p>
      </div>
    </div>
  </div>
  <div class="col-xs-12 col-sm-4">
    <p class="lead">{{ vm.sidebar.content }}</p>
  </div>
</div>

```

Меняем привязки данных, добавляя перед каждым элементом vm

После этого домашняя страница снова работает, теперь уже с использованием синтаксиса `controllerAs`. В дальнейшем будем применять такой подход для всех контроллеров.

Вернемся теперь к задаче обеспечения нормальной работы домашней страницы, получения местоположения посетителя и отображения списка близлежащих объектов. В главе 8 мы создали некоторые сервисы для работы с геолокацией и данными и планируем опять ввести их в SPA.

9.3.4. Сервисы

В главе 8 мы создали два сервиса: `geolocation` и `loc8rData`. Мы хотим извлечь их оттуда и добавить в качестве сервисов в SPA. Для добавления каждого из них необходимо:

- ❑ создать новый файл;
- ❑ вставить код сервиса;
- ❑ зарегистрировать сервис в приложении;
- ❑ добавить файл в `layout.jade`;
- ❑ вызвать сервис из контроллера `home`.

Создание файлов сервисов

Придерживаясь подхода с использованием для всех сервисов отдельных файлов, создадим по новому файлу для каждого из них. Эти сервисы не обязательно должны быть подкомпонентами домашней страницы, их можно будет повторно использовать по всему сайту. Поэтому вместо размещения их в каталоге `home` создайте новый подкаталог `common` в каталоге `app_client`, а в нем — подкаталог `services`, в результате чего получится путь `app_client/common/services`.

Создайте в этом каталоге новый файл `loc8rData.service.js` и вставьте в него код сервиса `loc8rData` из главы 8, как показано в листинге 9.6, не забыв также зарегистрировать его в приложении Angular.

Листинг 9.6. Создаем сервис `loc8rData`

```
angular
  .module('loc8rApp')
  .service('loc8rData', loc8rData);
function loc8rData ($http) {
  var locationByCoords = function (lat, lng) {
    return $http.get('/api/locations?lng=' + lng + '&lat=' + lat +
      '&maxDistance=20');
  };
  return {
```

```
    locationByCoords : locationByCoords
  };
}
```

Мы сообщим браузеру об этом файле буквально через минуту, но сначала сделаем то же самое для сервиса `geolocation`, как показано в листинге 9.7, на этот раз в новом файле `geolocation.service.js`.

Листинг 9.7. Создаем сервис `geolocation`

```
angular
  .module('loc8rApp')
  .service('geolocation', geolocation);
function geolocation () {
  var getPosition = function (cbSuccess, cbError, cbNoGeo) {
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(cbSuccess, cbError);
    }
    else {
      cbNoGeo();
    }
  };
  return {
    getPosition : getPosition
  };
}
```

Ничего нового ни в одном из этих файлов нет. Мы просто взяли созданный ранее код, поместили каждый сервис в отдельный файл и зарегистрировали их в приложении Angular. Теперь нам нужно сообщить браузеру о необходимости их загрузки.

Обеспечиваем доступ браузера к файлам

Никаких особых сюрпризов: мы добавим эти два файла в список сценариев, включенный в `layout.jade`, вот так:

```
script(src='/angular/angular.min.js')
script(src='/lib/angular-route.min.js')
script(src='/app.js')
script(src='/home/home.controller.js')
script(src='/common/services/loc8rData.service.js')
script(src='/common/services/geolocation.service.js')
```

Довольно несложно. Теперь необходимо внедрить код для использования сервисов.

Сервисы из контроллера

Мы уже сделали всю тяжелую работу, так как нужный код имеется в созданном в главе 8 контроллере. Извлечем код оттуда и поместим его в `homeCtrl` в нашем SPA.

Листинг 9.8 демонстрирует вид контроллера домашней страницы после выполнения этих действий, включая замену всех экземпляров `$scope` на `vm`. Не забываем передавать имена сервисов в контроллер, чтобы он мог их использовать.

Листинг 9.8. Модифицируем контроллер домашней страницы для использования двух сервисов

```
function homeCtrl (loc8rData, geolocation) {
  var vm = this;
  vm.pageHeader = {
    title: 'Loc8r',
    strapline: 'Find places to work with wifi near you!'
  };
  vm.sidebar = {
    content: "Looking for wifi and a seat? Etc etc"
  };
  vm.message = "Checking your location";
  vm.getData = function (position) {
    var lat = position.coords.latitude,
        lng = position.coords.longitude;
    vm.message = "Searching for nearby places";
    loc8rData.locationByCoords(lat, lng)
      .success(function(data) {
        vm.message = data.length > 0 ? "" :
          "No locations found nearby";
        vm.data = { locations: data };
      })
      .error(function (e) {
        vm.message = "Sorry, something's gone wrong";
      });
  };
  vm.showError = function (error) {
    vm.$apply(function() {
      vm.message = error.message;
    });
  };
  vm.noGeo = function () {
    vm.$apply(function() {
      vm.message = "Geolocation is
                    not supported by this browser.";
    });
  };
  geolocation.getPosition(vm.getData, vm.showError, vm.noGeo);
}

```

Вставляем функциональность из контроллера из главы 8, заменяя все экземпляры `$scope` на `vm`

Передаем имена сервисов в контроллер

Выглядит неплохо, не правда ли? Тут есть только одна проблема. `$apply` — метод `$scope`, и объект `vm` его не наследует и не имеет к нему доступа, поскольку `vm` является дочерним от `$scope`. Так что придется вновь добавить метод `$scope`.

Использование `$scope` при необходимости

Применяемый нами синтаксис `controllerAs` идеален для того, чтобы избавиться от метода `$scope` и избежать его излишнего использования. При этом подходе вы задействуете `$scope` только в случае крайней необходимости, что помогает обеспечить дополнительную ясность и понятность вашего кода и происходящих за его кулисами процессов.

Для использования `$scope.$apply` в контроллере `home` необходимо просто передать метод `$scope` точно так же, как и любую другую зависимость, а затем поменять нынешний вызов `vm.$apply` обратно на `$scope.$apply`. В листинге 9.9 показаны требуемые изменения (остальная часть кода опущена для краткости).

Листинг 9.9. Изменения, необходимые для использования `$scope.$apply` в контроллере `home`

```
function homeCtrl ($scope, loc&rData, geolocation) {
    vm.showError = function (error) {
        $scope.$apply(function() {
            vm.message = error.message;
        });
    };
    vm.noGeo = function () {
        $scope.$apply(function() {
            vm.message = "Geolocation is not supported by this browser.";
        });
    };
}

```

Передаем \$scope в контроллер в виде зависимости

Меняем имеющиеся vm.\$apply на \$scope.\$apply

Отлично, *теперь* все готово. Теперь область видимости снова будет обновляться в соответствии с асинхронными действиями сервиса геолокации.

РЕКОМЕНДУЕМОЕ РЕШЕНИЕ

Используйте метод `$scope` только тогда, когда это действительно необходимо. Применяйте подход с `ViewModel controllerAs` везде, где только можно.

Сейчас проверка сайта должна отобразить список местоположений, как показано на рис. 9.5.

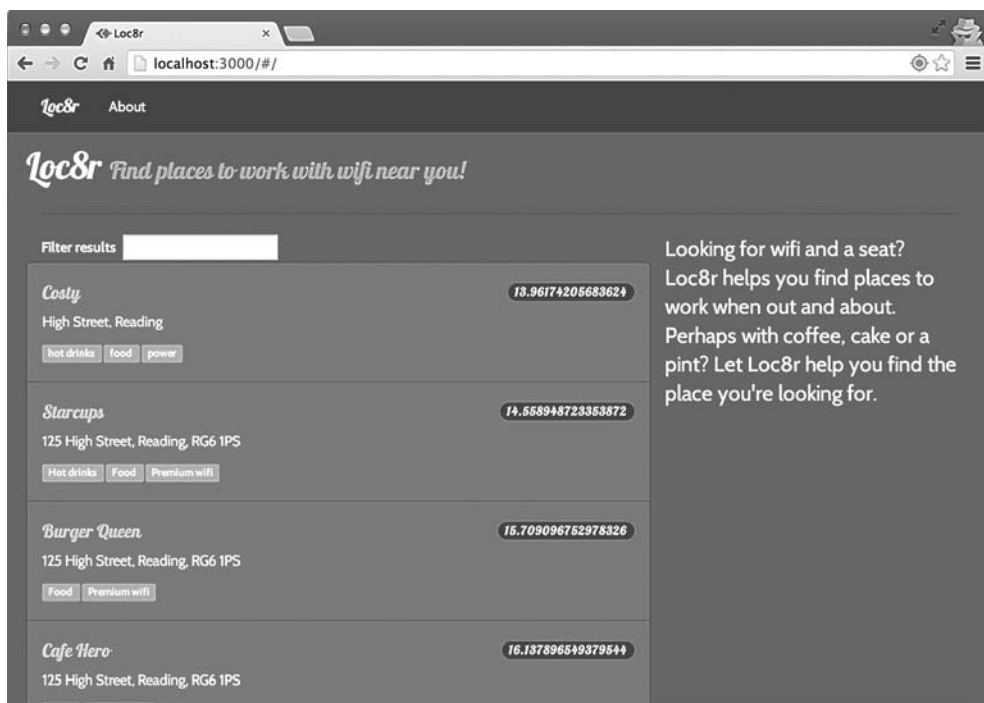


Рис. 9.5. Домашняя страница возвращает список местоположений с помощью сервисов geolocation и loc8rData

Все готово, осталось сделать несколько небольших улучшений, вновь введя в работу фильтры и директивы.

9.3.5. Использование фильтров и директив

Нам необходимо отображать на домашней странице звезды для оценок и правильно форматировать расстояния. В главе 8 мы создали фильтр `formatDistance` и директиву `ratingStar`, подключим их. Надеюсь, это продемонстрирует возможности и преимущества создания допускающих повторное использование модулей и компонентов, которые можно перемещать и подключать к различным проектам по мере необходимости.

Придерживаясь подхода «один компонент — один файл», будем создавать все фильтры и директивы в виде отдельных файлов и информировать о них браузер.

Создание файла фильтра `formatDistance`

Фильтр `formatDistance` спроектирован в расчете на многократное использование. Пока что мы применяем его только на домашней странице, но его можно использовать где угодно. Поэтому аналогично сервисам добавим его в каталог `common`. В каталоге `app_client/common` создаем новый подкаталог `filters`. В каталоге `filters` создаем новый файл `formatDistance.filter.js`.

В этот файл необходимо поместить функцию `formatDistance`, созданную в главе 8, а также используемую ею вспомогательную функцию `_isNumeric`. Добавок следует зарегистрировать эту функцию в качестве фильтра в приложении Angular с помощью геттер-синтаксиса. Все это показано в листинге 9.10.

Листинг 9.10. Создаем файл `formatDistance.filter.js`

```
angular
  .module('loc8rApp')
  .filter('formatDistance', formatDistance);
var _isNumeric = function (n) {
  return !isNaN(parseFloat(n)) && isFinite(n);
};
function formatDistance () {
  return function (distance) {
    var numDistance, unit;
    if (distance && _isNumeric(distance)) {
      if (distance > 1) {
        numDistance = parseFloat(distance).toFixed(1);
        unit = 'km';
      } else {
        numDistance = parseInt(distance * 1000,10);
        unit = 'm';
      }
      return numDistance + unit;
    } else {
      return "?";
    }
  };
}
```

Создание файлов директивы `ratingStar`

Директиве необходимы два файла: описывающий директиву файл JavaScript и шаблон HTML, который она будет использовать. Директива `ratingStar` применяется во многих местах, так что мы тоже разместим ее в каталоге `app_client/common`.

В каталоге `common` создаем каталог `directive`, в котором создаем подкаталог `ratingStars`, так что получается путь `app_client/common/directive/ratingStars`. Скопируйте в этот каталог созданный в главе 8 файл `/public/rating-stars.html` и ради соблюдения соглашения об именах модулей SPA переименуйте его в `ratingStars.template.html`.

ПРИМЕЧАНИЕ

Используемое нами в Angular соглашение об именах базируется на подходе «одна функция — один файл». Там, где в файле имеется одна ключевая функция, мы будем называть файл и содержащий его каталог в соответствии с именем этой функции.

Теперь создайте файл `ratingStars.directive.js` в том же каталоге. В нем будет находиться JavaScript-код для директивы, и, конечно, нам понадобится зарегистрировать его в приложении.

Именно это мы и делаем в листинге 9.11, не забыв поменять путь URL шаблона.

Листинг 9.11. Создаем файл директивы `ratingStars`

```
angular
  .module('loc8rApp')
  .directive('ratingStars', ratingStars);
function ratingStars () {
  return {
    restrict: 'EA',
    scope: {
      thisRating : '=rating'
    },
    templateUrl: '/common/directives/ratingStars/ratingStars.template.html'
  };
}
```

Здесь мы добавили кое-что новое, а именно — атрибут `restrict`. Он указывает Angular использовать директиву `ratingStars` только при обнаружении строки `rating-stars` в определенных местах. В данном случае `E` и `A` означают элемент (`element`) и атрибут (`attribute`), так что `rating-stars` может быть или своим собственным элементом, или атрибутом другого элемента. Существуют еще опции `C` — класс (`class`) и `M` — комментарий (`comment`), но рекомендуемое решение — использовать `EA`.

Прежде чем продолжить работу, необходимо добавить ссылки на файл фильтра `formatDistance` и файл директивы `ratingStars` в `layout.jade`, чтобы браузер мог их загрузить.

Настройка представления домашней страницы

Далее необходимо обеспечить использование директивы и фильтра представлением домашней страницы. Мы не внесли никаких относящихся к звездам для оценок изменений в HTML-код, но вынесли фильтр из элемента `distance`, чтобы предотвратить генерацию ошибок Angular.

Так, следующий фрагмент кода демонстрирует единственное изменение, которое нам необходимо сделать в файле `home.view.html`, а именно — вернуть туда фильтр `formatDistance`:

```
<h4>
  <a href="/location/{{ location._id }}">{{ location.name }}</a>
  <small class="rating-stars" rating-stars rating="location.rating"></small>
  <span class="badge pull-right badge-default">{{ location.distance |
    formatDistance }}</span>
</h4>
```

Мы закончили работу над домашней страницей! Давайте посмотрим на нее в браузере. Рисунок 9.6 демонстрирует домашнюю страницу с возвращенными на свои места звездами для оценок и исправленным отображением расстояний.

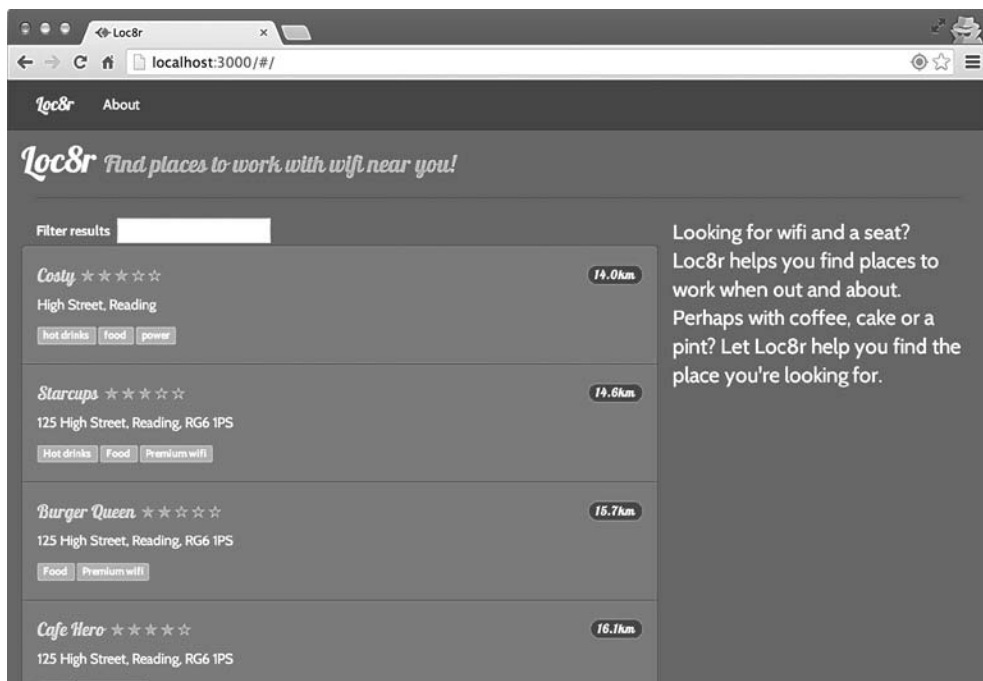


Рис. 9.6. После добавления фильтра `formatDistance` и директивы `ratingStars` домашняя страница опять выглядит должным образом

Что ж, отлично. Мы подготовили все для запуска SPA на Angular и добились запуска и функционирования нашего первого представления. Но мы уже добавили в шаблон шесть файлов для загрузки браузером, а создали только одну домашнюю страницу. Получилось немалое количество запросов, которые должен выполнять браузер. Что еще хуже, все файлы используют глобальные переменные. Так что давайте потратим немного времени и рассмотрим некоторые методы и рекомендуемые решения, которые можно использовать для исправления этих проблем, прежде чем продолжим создание приложения.

9.4. Улучшаем производительность браузера

Используемый нами модульный подход к программированию великолепен с точки зрения удобства сопровождения базы кода, но не очень подходит для браузеров, которым приходится загружать все мелкие файлы отдельно. У нас их уже немало, а ведь мы закончили только домашнюю страницу.

В этом разделе мы планируем уменьшить:

- количество глобальных переменных;
- количество скачиваемых браузером файлов;
- совокупный размер JavaScript-файлов.

Начнем с глобальных переменных. Решим эту задачу, обернув каждый файл в немедленно выполняемое функциональное выражение (immediately invoked function expression (IIFE)).

9.4.1. Обертываем каждый файл в IIFE

IIFE — способ инкапсуляции какого-либо кода JavaScript в отдельной области видимости со скрытием ее содержимого из глобальной области видимости. Прочитать об этом подробнее можно в приложении к книге, размещенном в Интернете.

Если вкратце, то IIFE выглядит так, как показано в следующем фрагменте кода, в котором мы обернули оператор `console.log`:

```
(function() {  
  console.log("Output immediately");  
})();
```

Этот код помещает функцию `console.log` в область видимости функции и немедленно вызывает ее. Именно это мы хотели бы сделать с нашим кодом.

В настоящий момент все файлы выполняются в глобальной области видимости. Это плохо, поскольку загромождает глобальную область видимости, повышает риск конфликта имен переменных и делает код приложения потенциально доступным для злоупотреблений.

ПРИМЕЧАНИЕ

Нашему приложению Angular не требуется связующая глобальная область видимости, ведь все связано посредством геттеров/сеттеров модулей `angular.module('loc8rApp')`.

В листинге 9.12 можно увидеть, как все это выглядит в `app_client/app.js`.

Листинг 9.12. Обертывание файлов приложения Angular в IIFE, например `app.js`

```
(function () {      ←—————  Открываем IIFE

  angular.module('loc8rApp', ['ngRoute']);
  function config($routeProvider) {
    $routeProvider
      .when('/', {
        templateUrl: 'home/home.view.html',
        controller: 'homeCtrl',
        controllerAs: 'vm'
      })
      .otherwise({redirectTo: '/'});
  }
  angular
    .module('loc8rApp')
    .config(['$routeProvider', config]);

})();      ←—————  Закрываем IIFE и выполняем вызов
```

Пока что мы тут фактически не поменяли никакого кода, просто обернули все содержимое файла в IIFE. Теперь необходимо просто пройтись по всему коду и обернуть подобным образом каждый из созданных в этой главе JavaScript-файлов:

- `/app_client/common/directives/ratingStars/ratingStars.directive.js;`
- `/app_client/common/filters/formatDistance.filter.js;`
- `/app_client/common/services/geolocation.service.js;`
- `/app_client/common/services/loc8rData.service.js;`
- `/app_client/home/home.controller.js.`

Вот и все, что относится к первой цели — уменьшению количества глобальных переменных. В дальнейшем будем предполагать, что все JavaScript-файлы приложения Angular обернуты в IIFE.

Теперь хотелось бы заняться уменьшением количества файлов и совокупного их размера. Это влечет за собой сокращение сценариев. Если мы попробуем сделать это прямо сейчас, работа приложения будет нарушена, так что давайте разберемся, почему так происходит и что можно с этим сделать.

9.4.2. Внедряем зависимости вручную для защиты от сокращения

Сокращение существующего кода приведет к нарушению работы приложения. Это произойдет потому, что мы внедряем имена зависимостей в контроллер и функции сервисов как параметры. При сокращении кода эти имена превратятся в отдельные буквы.

Например, описание функции `homeCtrl` выглядит следующим образом:

```
function homeCtrl ($scope, loc8rData, geolocation)
```

А после сокращения оно будет выглядеть примерно вот так:

```
function homeCtrl(a,b,c){
```

Уточним для ясности: сокращение не нарушает код JavaScript. Каждая из созданных нами функций в отдельности будет работать. Проблема связана с Angular, поскольку параметры там — это не обычные параметры, используемые только в контексте функции. Параметры являются также ссылками на имена других частей приложения, например сервисов. Если говорить о предыдущих фрагментах кода, то приложение знает, что такое сервис `loc8rData`, но понятия не имеет о сервисе под названием `b`.

Предохранить от этого можно, вручную внедрив зависимости в виде строк, которые не будут меняться в процессе сокращения. Для этого Angular предоставляет метод `$inject` для конструкторов контроллеров и сервисов. Метод `$inject` принимает на входе массив строк, являющихся зависимостями для конкретного контроллера или сервиса и соответствующих передаваемым в виде параметров.

Это один из тех случаев, когда пример действительно может пролить свет на то, о чем мы тут говорим. В следующем фрагменте кода мы добавляем внедрение зависимости для контроллера домашней страницы непосредственно перед описанием функции:

```
homeCtrl.$inject = ['$scope', 'loc8rData', 'geolocation'];  
function homeCtrl ($scope, loc8rData, geolocation) {
```

Метод `$inject` применяется к имени функции, принимая на входе массив зависимостей. Массив должен содержать строки, поскольку они не меняются в процессе сокращения. Содержимое массива должно находиться в том же порядке, что и параметры функции.

Нам осталось сделать только одно — внедрить метод `$http` в сервис `loc8rData` следующим образом:

```
loc8rData.$inject = ['$http'];  
function loc8rData ($http) {
```

Организованное подобным образом внедрение зависимостей представляется довольно простым. Просто не забудьте проделать это для каждого контроллера или сервиса, которому это необходимо. Теперь мы можем приступить к сокращению сценариев.

9.4.3. Используем UglifyJS для сокращения и конкатенации сценариев

Для сокращения и конкатенации сценариев приложения Angular будем использовать инструмент от стороннего разработчика под названием UglifyJS¹. При запуске приложения Node в Express UglifyJS берет файлы с исходным кодом приложения Angular, помещает их все в один файл и сжимает его. Мы модифицируем приложение так, чтобы оно использовало этот файл вместо нескольких файлов, применяемых в настоящий момент.

Итак, начнем с установки UglifyJS.

Установка UglifyJS

Добавить новый модуль `npm` и поменять `package.json` — задача несложная, мы уже делали это неоднократно.

Открываем командную строку в корневом каталоге приложения, там, где находится файл `package.json`. В командной строке выполняем следующее:

```
$ npm install uglify-js --save
```

Эта команда установит модуль UglifyJS и добавит его в `package.json`.

Добавление UglifyJS в приложение

Теперь, когда установка выполнена, пришло время внедрить его в приложение. Сделаем это прямо в корневом каталоге приложения, в `/app.js`, с которого все начинается. Необходимо выполнить запрос UglifyJS, и нам также понадобится ссылка на модуль Node по умолчанию, который называется `fs`. Это название расшифровывается как «файловая система» (`filesystem`) — для сохранения сокращенного файла нам понадобится доступ к файловой системе.

¹ Uglify (англ.) — «обезобразивать». — *Примеч. пер.*

В листинге 9.13 показаны изменения, которые необходимо сделать в `/app.js` для внедрения этих двух модулей.

Листинг 9.13. Добавляем UglifyJS в приложение Node

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
require('./app_api/models/db');
var uglifyJs = require("uglify-js");
var fs = require('fs');
```

Отлично, все готово. Давайте использовать!

«Обезображиваем» файлы JavaScript

Вспользуемся UglifyJS для объединения всех файлов приложения Angular в один с последующим сокращением. Этот процесс выполняется в оперативной памяти, так что сразу после генерации объединенного файла мы используем файловую систему для его сохранения.

Хотелось бы сделать это вскоре после запуска приложения, так что вставим соответствующий код довольно высоко в основном файле `app.js` в корневом каталоге проекта. Место сразу после объявления механизма представления подойдет. Мы хотим добавить три отдельные части кода:

- перечисление всех файлов, которые хотим объединить в массив;
- вызов UglifyJS для объединения и сокращения файла в памяти;
- сохранение «обезображенного» кода в каталоге `public`.

В листинге 9.14 представлен код, который необходимо добавить, начиная с описания массива файлов, предназначенных для «обезображивания».

Листинг 9.14. «Обезображиваем» и сохраняем новый файл

```
routeapp.set('views', path.join(__dirname, 'app_server', 'views'));
app.set('view engine', 'jade');
var appClientFiles = [
  'app_client/app.js',
  'app_client/home/home.controller.js',
  'app_client/common/services/geolocation.service.js',
  'app_client/common/services/loc8rData.service.js',
  'app_client/common/filters/formatDistance.filter.js',
  'app_client/common/directives/ratingStars/
    ratingStars.directive.js'
];
```

Этап 1: описываем массив файлов для «обезображивания»

```

var uglified = uglifyJs.minify(appClientFiles, { compress : false });

fs.writeFile('public/angular/loc8r.min.js', uglified.code, function (err){
  if(err) {
    console.log(err);
  } else {
    console.log('Script generated and saved:
                loc8r.min.js');
  }
});

```

Сохраняем сгенерированный файл

Запускаем процесс uglifyJs.minify для массива файлов

Теперь, если вы перезапустите приложение Node, то обнаружите, что в каталог `public/angular` был добавлен новый файл `loc8r.min.js`. На данном этапе размер сокращенного файла составляет 2160 байт, тогда как общий размер отдельных файлов — примерно 3575 байт. Так что, помимо уменьшения количества запросов, которые должен выполнить браузер, с шести до одного, мы уменьшили размер файлов почти на 40 %. Экономия 1,5 Кбайт не выглядит очень большой, но не забывайте, что наше приложение пока что очень маленькое и имеет дело только с домашней страницей.

Но, конечно, браузер не извлечет из этого никакой выгоды, если мы не сообщим ему о новом файле.

Использование сокращенного файла в HTML

Замена отдельных файлов одним в HTML-коде — просто вопрос привязки. Необходимо просто изменить `layout.jade`, закомментировав отдельные файлы и добавив сокращенный файл, как показано в следующем фрагменте кода:

```

script(src='/angular/loc8r.min.js') ←
  //- script(src='/app.js')
  //- script(src='/common/services/loc8rData.service.js')
  //- script(src='/common/services/geolocation.service.js')
  //- script(src='/common/directives/ratingStars/ratingStars.directive.js')
  //- script(src='/common/filters/formatDistance.filter.js')
  //- script(src='/home/home.controller.js')

```

Добавляем ссылку на новый, объединенный и сокращенный, файл

Комментируем отдельные файлы

«Почему комментируем, а не удаляем?» — спросите вы. Один сокращенный файл гораздо менее удобен для отладки, чем исходные отдельные файлы. Все ошибки окажутся в строке 1, имена функций и переменных поменяются и т. д. Так что, если у нас возникнут проблемы, мы сможем вернуться к использованию отдельных файлов для отладки. При этом опять будут отображаться осмысленные имена файлов и номера строк.

Часть этого можно автоматизировать с помощью системы сборки или инструмента для запуска задач, таких как Gulp или Grunt. Их можно настроить таким образом, чтобы генерировать сокращенный файл только при развертывании для промышленной эксплуатации. Или настроить для отслеживания изменений в определенных файлах и создания сокращенной версии на лету. Они могут даже генерировать карты кода, которые связывают сокращенные версии файлов с исходными для упрощения отладки.

Мы не станем подробно рассматривать здесь Gulp или Grunt, но они, безусловно, заслуживают изучения.

Предотвращение рекурсивного цикла Nodemon

Все работает как и должно, но если вы запустите приложение с помощью nodemon, то обнаружите, что в самом начале оно несколько раз перезапускается. Это происходит из-за того, что nodemon перезапускает приложение при изменении файлов, а мы перекомпилируем сокращенный файл заново каждый раз при запуске приложения. Так что этот цикл, наверное, не такая уж неожиданность.

Исправить это можно, сообщив nodemon о необходимости игнорировать изменения в любых файлах из каталога public. Этот каталог содержит статические ресурсы, выдаваемые браузеру, так что повторная компоновка приложения для отражения каких-либо изменений не требуется.

Передать параметры конфигурации nodemon можно, просто создав файл nodemon.json в корневом каталоге приложения. Следующий фрагмент кода подключит его в режим вывода подробностей (так что вы получите в консоли массу информации) и заставит игнорировать все находящееся в каталоге public:

```
{
  "verbose": true,
  "ignore": ["public/*"]
}
```

Теперь при перезапуске приложения с помощью nodemon вы увидите в терминале несколько другие сообщения, но главное, оно не будет перезапускаться несколько раз.

9.5. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Настройку Express для выдачи SPA.
- ❑ Использование маршрутизации Angular вместо Express для выдачи страниц.
- ❑ Связывание представлений и контроллеров с маршрутами.
- ❑ Использование синтаксиса `controllerAs`.
- ❑ Рекомендуемое решение — использовать для всего отдельные файлы.
- ❑ Защиту глобальной области видимости с помощью ПФЕ.
- ❑ Внедрение зависимостей.
- ❑ Сокращение и конкатенацию отдельных файлов в один маленький файл приложения.

В главе 10 мы планируем продолжить создание приложения на основе фундамента SPA и рассмотреть новые для нас вещи, такие как добавление дополнительных страниц, удаление # из URL, использование параметров маршрутов, добавление предварительно собранных компонентов Angular и вставка данных обратно в БД. Если коротко, масса интересного!

Глава 10

Создание одностраничного приложения с помощью Angular: следующий уровень

В этой главе:

- ❑ формирование изящных URL;
- ❑ добавление в SPA нескольких представлений;
- ❑ переход с одной страницы на другую без перезагрузки приложения;
- ❑ использование AngularUI для получения компонентов Twitter Bootstrap в виде предварительно сконфигурированных директив Angular.

В этой главе мы продолжаем начатое в главе 9 построение одностраничного приложения. К концу главы `Loc8r` станет единым приложением Angular и будет использовать наше API для получения данных.

Рисунок 10.1 демонстрирует наше местоположение в общем плане: мы все еще переделываем основное приложение в SPA Angular.

Начнем с расщепления приложения Angular с серверным приложением — оно все еще встроено в шаблон Jade. В качестве части этой задачи мы узнаем, как сделать элегантные URL, убрав `#`. После этого создадим недостающие страницы и функциональность и рассмотрим внедрение HTML в привязки, а также использование в маршрутах параметров URL и предварительно сконфигурированных директив, основанных на компонентах Twitter Bootstrap. И конечно, между делом уделим должное внимание рекомендуемым решениям.

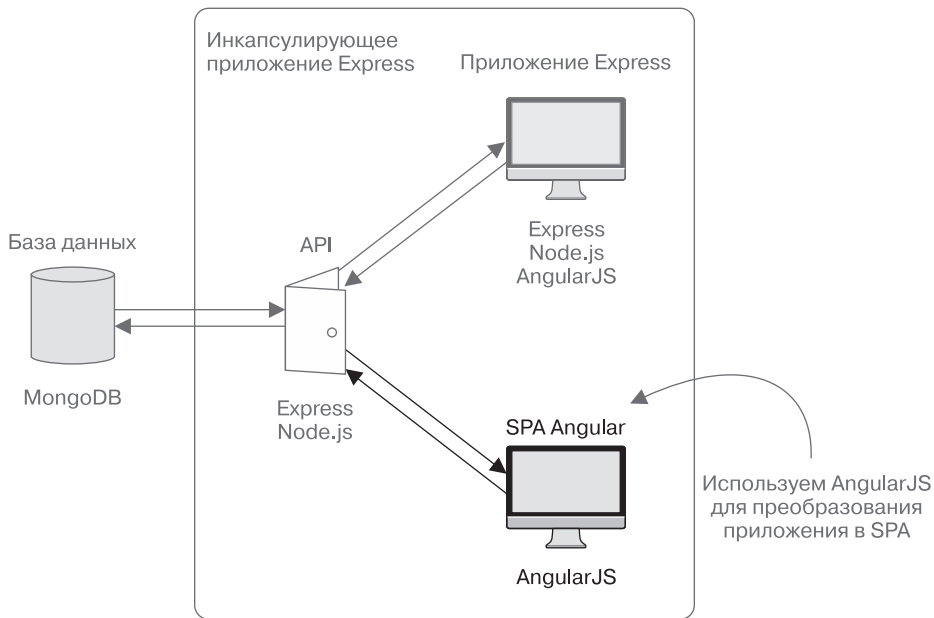


Рис. 10.1. В этой главе продолжается начатая в главе 9 работа по пересозданию приложения `Loc8r` в виде SPA Angular с перемещением логики приложения из серверной части в клиентскую

10.1. Полнофункциональное SPA: перестаем основываться на серверном приложении

В нынешнем виде в нашем приложении навигация, каркас страницы, заголовок и нижний колонтитул — все находится в шаблоне Jade. Для использования этого шаблона имеется контроллер в каталоге `app_server`. Такой вариант хорошо работает и отлично подходит для некоторых сценариев. Но для настоящего SPA нужно, чтобы все имеющее отношение к клиентскому приложению находилось в `app_client`. Идея тут в том, чтобы можно было легко перемещать все SPA в целом и размещать его где угодно, например в CDN, если вам понадобилось извлечь его из инкапсулирующего приложения Express.

Чтобы достичь этого, начнем с создания HTML-страницы хоста `app_client` и соответствующего изменения маршрутизации Express. Далее извлечем разделы HTML-страницы и превратим их в допускающие повторное использование компоненты, а именно в директивы. Наконец, рассмотрим способ «облагораживания» URL посредством удаления `#`.

10.1.1. Создание изолированной HTML-страницы хоста

Что ж, первый шаг — создание HTML-страницы хоста таким образом, который бы не основывался на серверных маршрутах и контроллерах.

Создаем новый index.html

HTML-код, с которого мы собираемся начать, — такой же, как и уже сгенерированный файлом `layout.jade`. Если мы возьмем его и преобразуем в HTML-код, он будет выглядеть так, как показано в листинге 10.1. Сохраните этот файл под именем `app_client/index.html`.

Листинг 10.1. Преобразованная в HTML-код страница хоста

```
<!DOCTYPE html>
<html ng-app="loc8rApp">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Loc8r</title>
    <link rel="stylesheet" href="/bootstrap/css/amelia.bootstrap.css">
    <link rel="stylesheet" href="/stylesheets/style.css">
  </head>
  <body>
    <div class="navbar navbar-default navbar-fixed-top">
      <div class="container">
        <div class="navbar-header"><a href="/" class="navbar-brand">Loc8r</a>
          <button type="button" data-toggle="collapse" data-target="#navbar-main" class="navbar-toggle"><span class="icon-bar"></span><span class="icon-bar"></span><span class="icon-bar"></span></button>
        </div>
        <div id="navbar-main" class="navbar-collapse collapse">
          <ul class="nav navbar-nav">
            <li><a href="/#about">About</a></li>
          </ul>
        </div>
      </div>
    </div>
    <div class="container">
      <div ng-view>
      </div>
      <footer>
        <div class="row">
          <div class="col-xs-12"><small>&copy; Simon Holmes 2014</small></div>
        </div>
      </footer>
    </div>
    <script src="/angular/angular.min.js"></script>
```

```

<script src="/lib/angular-route.min.js"></script>
<script src="/lib/angular-sanitize.min.js"></script>
<script src="/angular/loc8r.min.js"></script>
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/
  jquery.min.js"></script>
<script src="/bootstrap/js/bootstrap.min.js"></script>
<script src="/javascript/validation.js"></script>
</body>
</html>

```

Ничего сверхъестественного тут нет, так что пойдём дальше и поменяем приложение, чтобы оно действительно его использовало.

Маршрутизация к статическому HTML-файлу из Express

Задумаемся на минуту о том, когда мы хотели бы отправлять этот HTML-файл браузеру. Определенно тогда, когда кто-то посещает домашнюю страницу. Но если мы сможем избавиться от # в URL, что мы и собираемся сделать через несколько страниц, то нам хотелось бы выводить его для URL любого вида.

Поскольку мы используем Angular для маршрутизации, нам не хотелось бы дублировать ее и заниматься маршрутами еще и в Express. В то же время не хочется на все запросы возвращать этот HTML-файл, поскольку мы также обслуживаем API-запросы от приложения и выдаем статические ресурсы (например, CSS, JavaScript и изображения). Как же следует поступить?

Как вы можете помнить, маршрутизация Express прекращает работу при нахождении первого соответствия после выполнения всего промежуточного ПО. Если никаких совпадений маршрутов не найдено, она продолжает работу. Мы можем воспользоваться здесь этой особенностью. Если отключить все маршруты для `app_server`, ни для одного из запросов к этим URL не будет найдено соответствия, а значит, они все продолжат работу до конца.

В конце мы можем перехватить все запросы URL, для которых не нашлось соответствия, и выдать новый HTML-файл. Следующий фрагмент кода демонстрирует необходимые для этого изменения в файле `app.js`, включая комментирование первоначальных маршрутов:

```

// require('./routes')(app);
require('./app_api/routes')(app);

app.use(function(req, res) {
  res.sendFile(path.join(__dirname, 'app_client', 'index.html'));
});

```

← Комментируем или удаляем строку, выполняющую запрос маршрутов серверного приложения

Добавляем универсальную функцию-перехватчик `app.use`, возвращающую ответ на любые запросы путем отправки HTML-файла

Если вы перезапустите приложение и перейдете на домашнюю страницу, то увидите, что она работает так же, как и раньше, однако теперь нет необходимости в маршрутах и контроллерах серверного приложения для выдачи базовой HTML-страницы.

СОВЕТ

При использовании этого подхода все URL, для которых не нашлось соответствия, будут возвращать ответ в виде отправки загружающего приложение Angular HTML-файла. Так что маршрутизация Angular вполне удовлетворительно станет обрабатывать незнакомые запросы.

Теперь давайте сделаем весь этот HTML-код частью приложения Angular.

10.1.2. Создаем допускающие многократное использование директивы для каркаса страницы

Итак, теперь мы отправляем простейшую HTML-страницу для выдачи приложения, но на этой странице имеется масса разметки. Лучше, если эта разметка будет находиться внутри приложения, чтобы удобнее было работать с ней в Angular. Не забывайте, что вы хотели бы с помощью Angular создать DOM, а не управлять им позднее, как было бы с jQuery.

Возьмем из HTML-страницы нижний колонтитул и навигацию и превратим их в директивы, чтобы можно было включить их в любую страницу. То же самое сделаем с заголовком страницы, находящимся сейчас в представлении домашней страницы.

Как говорилось в главе 8, директива состоит из двух основных частей: JavaScript-файла для ее описания и шаблона представления для ее отображения. В свою очередь, JavaScript-файл должен быть добавлен в `app.js`, чтобы приложение могло его использовать, а каждая директива — размещена при необходимости в качестве элемента (или атрибута элемента) в представлениях хоста.

Создаем директиву Footer

Нижний колонтитул — простейший компонент, поскольку в нем должно быть лишь немного HTML-кода. Тут не обойдется без небольшой уловки. Если вы хотите использовать директиву в качестве элемента, то не можете назвать ее так же, как существующий тег. Поэтому нельзя назвать директиву нижнего колонтитула `footer` и попытаться включить ее в сайт как `<footer>`, поскольку спецификация HTML уже содержит тег `footer`.

Назовем нижний колонтитул `footerGeneric` и создадим каталог `footerGeneric` в каталоге `app_client/common/directives`. Здесь мы разместим необходимые для директивы HTML- и JavaScript-файлы.

Начинаем с HTML и создадим файл `footerGeneric.template.html`, вставив HTML-код для нижнего колонтитула, как показано в следующем фрагменте кода:

```
<footer>
  <div class="row">
    <div class="col-xs-12"><small>&copy; Simon Holmes 2014</small></div>
  </div>
</footer>
```

Далее необходимо создать соответствующий JavaScript-файл под названием `footerGeneric.directive.js`. В листинге 10.2 мы воспользуемся этим файлом для описания новой директивы, регистрации ее в основном приложении и назначения в качестве шаблона представления только что созданного HTML-файла.

Листинг 10.2. Описываем обобщенный нижний колонтитул в виде директивы: `footerGeneric.directive.js`

```
(function () {

  angular
    .module('loc8rApp')
    .directive('footerGeneric', footerGeneric);

  function footerGeneric () {
    return {
      restrict: 'EA',
      templateUrl: '/common/directives/footerGeneric/
        footerGeneric.template.html'
    };
  }

})();
```

После создания и сохранения этого файла не забудьте добавить его в массив `appClientFiles` в `app.js`. Теперь, если мы захотим включить нижний колонтитул в одну из страниц Angular, то сможем использовать новый элемент `<footer-generic></footer-generic>`.

Переносим навигацию в директиву

Подход, применяемый для директивы навигации, очень похож на таковой для нижнего колонтитула. Она содержит больше HTML-кода, но не должна выполнять никаких хитрых действий с данными. Так что просто создаем новый каталог `navigation` там же, где и каталог директивы нижнего колонтитула — `app_client/common/directives`.

В этом каталоге также будут находиться HTML- и JavaScript-файлы. Листинг 10.3 демонстрирует необходимый для директивы навигации HTML-код.

Листинг 10.3. HTML навигации: navigation.template.html

```
<div class="navbar navbar-default navbar-fixed-top">
  <div class="container">
    <div class="navbar-header"><a href="/" class="navbar-brand">Loc8r</a>
      <button type="button" data-toggle="collapse" data-target="#navbar-main"
        class="navbar-toggle"><span class="icon-bar"></span><span class="icon-
        bar"></span><span class="icon-bar"></span></button>
    </div>
    <div id="navbar-main" class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/about">About</a></li>
      </ul>
    </div>
  </div>
</div>
```

Листинг 10.4 показывает описание JavaScript для директивы навигации.

Листинг 10.4. Описываем директиву навигации: navigation.directive.js

```
(function () {

  angular
    .module('loc8rApp')
    .directive('navigation', navigation);

  function navigation () {
    return {
      restrict: 'EA',
      templateUrl: '/common/directives/navigation/navigation.template.html'
    };
  }
})();
```

Не забудьте добавить JavaScript-файл в массив в файле `app.js`! Основная причина создания этих отдельных файлов и каталогов — удобство сопровождения и повторного использования. Если каждый файл и каталог выполняет одну задачу, легче понять, куда нужно заглянуть, чтобы исправить или поменять что-то. Проще и перенести компонент из одного проекта в другой.

Создаем директиву для заголовка страницы

Заголовок страницы должен отображать на разных страницах различные данные. Аналогично тому, как мы поступили с директивой `rating-stars`, создадим изолированную область видимости в Angular и передадим туда данные.

Как и раньше, создаем каталог с именем `pageHeader`, а также пустые файлы HTML и JavaScript. Начнем с HTML-разметки из следующего фрагмента кода. Можно взять его непосредственно из шаблона представления домашней страницы, но нам необходимо поменять привязку данных. Поскольку мы используем изолированную область видимости, то у нас нет прямого доступа к данным в `vm`. Вместо этого укажем, что хотели бы, чтобы данные для названия и подзаголовка находились в объекте `content`, например:

```
<div id="banner" class="page-header">
<div class="row">
<div class="col-lg-6"></div>
<h1>
  {{ content.title }}
<small>{{ content.strapline }}</small>
</h1>
</div>
```

Директива будет ожидать передачи названия и подзаголовка в виде свойств объекта `content`

Далее описываем директиву аналогично тому, как сделали с остальными, но на этот раз добавляя параметр `scope`, как показано в листинге 10.5. Этот параметр мы будем использовать для передачи объекта `content`, необходимого HTML-коду.

Листинг 10.5. Описываем директиву заголовка страницы: `pageHeader.directive.js`

```
(function () {
  angular
    .module('loc8rApp')
    .directive('pageHeader', pageHeader);

  function pageHeader () {
    return {
      restrict: 'EA',
      scope: {
        content : '=content'
      },
      templateUrl: '/common/directives/pageHeader/pageHeader.template.html'
    };
  }
})();
```

Описываем изолированную область видимости, передавая объект `content`

При использовании этой директивы в представлении контроллера нам придется передавать объект `content` в виде атрибута элемента. Например, можем использовать его следующим образом:

```
<page-headercontent="vm.pageHeader"></page-header>
```

Подобный стиль использования изолированной области видимости защищает директиву от любых изменений названий в области видимости. Главное, что

вы передаете ей ожидаемый ею контент, а как вы называли его прежде, неважно. Опять же это делает код по-настоящему позволяющим многократное использование.

Как и прежде, не забудьте добавить этот файл в массив `appClientFiles` в файле `app.js` Express.

Итоговый шаблон домашней страницы

Теперь, после создания всех директив, можно добавить их в шаблон представления домашней страницы, как показано в листинге 10.6. Новые директивы выделены полужирным шрифтом, но ради сохранения структуры DOM нам придется добавить еще немного разметки контейнера из файла `index.html`.

Листинг 10.6. Шаблон представления домашней страницы полностью

```
<navigation></navigation>

<div class="container">
  <page-header content="vm.pageHeader"></page-header>

  <div class="row">
    <div class="col-xs-12 col-sm-8">
      <label for="filter">Filter results</label>
      <input id="filter" type="text", name="filter",
        ng-model="textFilter">
      <div class="error">{{ vm.message }}</div>
      <div class="row list-group">
        <div class="col-xs-12 list-group-item" ng-repeat="location in
          vm.data.locations | filter : textFilter">
          <h4>
            <a href="#/location/{{ location._id }}">{{ location.name }}</a>
            <small class="rating-stars" rating-stars
              rating="location.rating"></small>
            <span class="badge pull-right badge-default">{{
              location.distance | formatDistance }}</span>
          </h4>
          <p class="address">{{ location.address }}</p>
          <p>
            <span class="label label-warning label-facility"
              ng-repeat="facility in location.facilities">
              {{ facility }}
            </span>
          </p>
        </div>
      </div>
    </div>
  </div>
```

```

    </div>
    <div class="col-xs-12 col-sm-4">
      <p class="lead">{{ vm.sidebar.content }}</p>
    </div>
  </div>

  <footer-generic></footer-generic>
</div>

```

Подобное использование директив — если им даны разумные имена — по настоящему помогает понять структуру вашего представления с первого взгляда, не увязнув в обилии разметки.

Мы изъяли всю разметку из файла `index.html`, так как же он выглядит сейчас?

Итоговый файл `index.html`

После извлечения всей HTML-разметки и перенесения ее в директивы и представления в файле `index.html` осталось не так уж много. Но именно этого мы и добивались. Поскольку всем контентом страницы теперь управляют контроллеры и представления, необходимо переместить директиву `ng-view` в тег `body` (листинг 10.7).

Листинг 10.7. Итоговый файл `index.html`

```

<!DOCTYPE html>
<html ng-app="loc8rApp">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Loc8r</title>
    <link rel="stylesheet" href="/bootstrap/css/amelia.bootstrap.css">
    <link rel="stylesheet" href="/stylesheets/style.css">
  </head>
  <body ng-view>
    <script src="/angular/angular.min.js"></script>
    <script src="/lib/angular-route.min.js"></script>
    <script src="/angular/loc8r.min.js"></script>
    <script src="//ajax.googleapis.com/
      ajax/libs/jquery/1.10.2/
      jquery.min.js"></script>
    <script src="/bootstrap/js/bootstrap.min.js"></script>
    <script src="/javascript/validation.js"></script>
  </body>
</html>

```

← Директива `ng-view` теперь располагается в теге `body`, так что мы можем контролировать в Angular всю страницу

Теперь у нас настоящее SPA. Есть минимальный HTML-файл, а всем остальным управляет Angular. Вид браузера и исходный текст HTML-кода демонстрирует рис. 10.2.

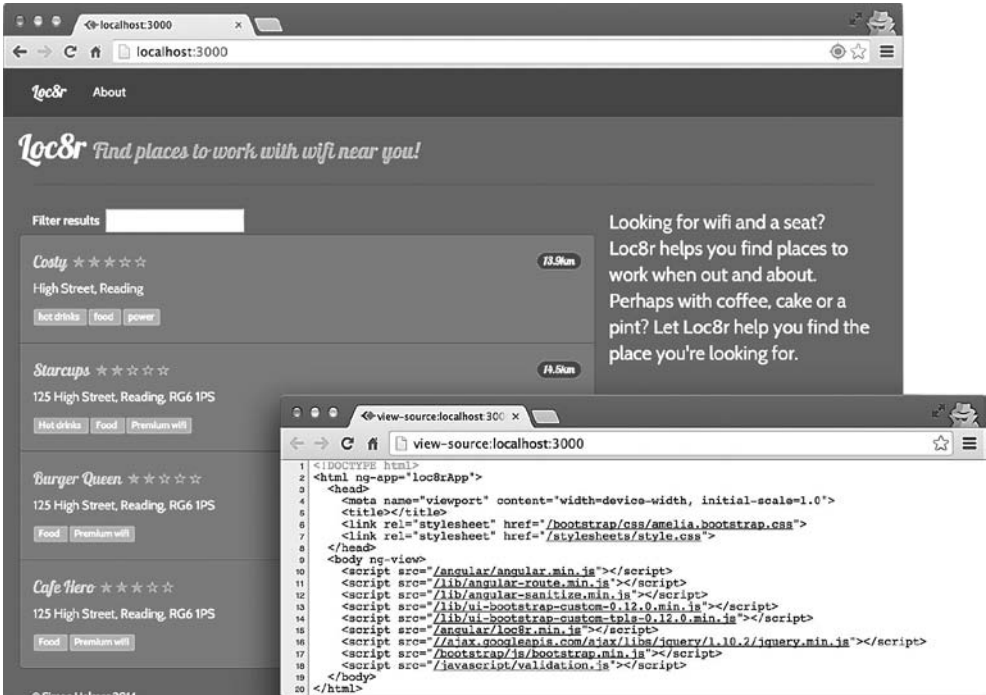


Рис. 10.2. Теперь это настоящее SPA: исходный код HTML-страницы минимален, но приложение все же полнофункционально

Далее мы планируем рассмотреть возможность удаления # из URL.

10.1.3. Удаление # из URL

Частое требование к SPA — «красивые» URL. Наши URL неплохи, но все они содержат #. Angular обеспечивает способ удаления # из адресной строки, но, пожалуйста, обратите внимание на то, что он плохо работает для браузера Internet Explorer версий 9 и старше. Если вам необходимо поддерживать ранние версии Internet Explorer, не используйте изложенное в данном разделе.

Спецификации HTML5 позволяют браузерам вставлять состояния в историю просмотров. Основная причина этого — желание предоставить браузерам способ использования в SPA кнопки возврата таким образом, чтобы она не уведила с просматриваемого сайта.

Маршрутизация Angular может неплохо это использовать. Нам просто нужно включить эту возможность!

Используем \$locationProvider и режим HTML5

Чтобы воспользоваться режимом HTML5, необходимо добавить новый провайдер в конфигурацию приложения Angular. Мы уже применяем провайдер \$routeProvider, а теперь нужно передать туда \$locationProvider. Он является родным для Angular, так что нам не нужно скачивать какие-то дополнительные библиотеки.

Активизировать режим HTML5 можно с помощью простой команды из одной строки, вот так:

```
$locationProvider.html5Mode(true);
```

В листинге 10.8 мы внесем некоторые изменения в файл app_client/app.js, чтобы поменять все URL шаблона, передать в провайдер config\$locationProvider и установить переменную html5Mode в значение true.

Листинг 10.8. Активизируем API HTML5

```
(function () {
    angular.module('loc8rApp', ['ngRoute']);

    function config($routeProvider, $locationProvider) {
        $routeProvider
            .when('/', {
                templateUrl: 'home/home.view.html',
                controller: 'homeCtrl',
                controllerAs: 'vm'
            })
            .otherwise({redirectTo: '/'});

        $locationProvider.html5Mode(true);
    }

    angular
        .module('loc8rApp')
        .config(['$routeProvider', '$locationProvider', config]);
})();
```

Принимает \$locationProvider в качестве параметра config

Присваивает html5Mode значение true

Добавляет \$locationProvider в качестве зависимости config

Теперь при перезагрузке приложения в современном браузере URL для домашней страницы больше не будет заканчиваться #/. Он будет аккуратно заканчиваться вашим доменом. Если вы загрузите страницу в IE9, она по-прежнему будет работать, но в адресе будет присутствовать #.

Работаем с Internet Explorer

Рассмотренная разновидность маршрутизации не будет работать в IE8 или 9, поскольку у них нет доступа к API HTML5. Тем не менее приложение будет в них работать, только Angular вернется к использованию #. Так что навигация по приложению будет работать нормально, хотя и с элементом /#/ в URL.

Проблема возникает, когда кто-то пытается скопировать и вставить внешнюю гиперссылку без знака # и попытаться использовать ее в IE9. При этом Internet Explorer будет просто визуализировать страницу, ведь именно так настроена маршрутизация по умолчанию.

Но — учтите, я вам этого не говорил! — есть маленькая грязная уловка, которой вы можете воспользоваться: вставить соответствующий код вверху контроллера домашней страницы. При ее выполнении контроллер домашней страницы будет проверять, каково имя пути URL. Если оно не соответствует домашней странице, то есть не равно просто /, контроллер домашней страницы возьмет имя пути, предварит его # и перенаправит страницу, как показано в следующем фрагменте кода:

```
if (window.location.pathname !== '/') {  
  window.location.href = '/' + window.location.pathname;  
}
```

Я упомянул, что эта уловка грязная. Если кто-нибудь вставит URL без # в IE9, то получит мерцание изображения, когда домашняя страница начнет загружаться до того, как Angular выполнит перенаправление на правильный маршрут. Это не лучший вариант, так что хорошо подумайте, прежде чем ступить на этот путь при необходимости поддержки старых версий Internet Explorer.

Ладно, хватит гнусного хакерства! Вернемся к нашим баранам и добавим в приложение еще одну страницу.

10.2. Добавление дополнительных страниц и динамическое внедрение HTML

Идея SPA заключается в том, что приложение на стороне сервера выдает браузеру одну страницу, а приложение на стороне клиента выполняет все остальное. В этом разделе мы рассмотрим включение дополнительных страниц на примере добавления страницы About (О нас). А также разберемся с проблемами, возникающими при попытке внедрения HTML в привязку Angular.

10.2.1. Добавление в SPA нового маршрута и страницы

Вероятно, вы уже представляете себе, как это будет работать: для добавления маршрута, указывающего на новый шаблон и контроллер, используется функция `config` из файла `app.js`. Если вы именно так это себе и представляли, то вы угадали!

Модификация ссылки на навигационной панели

Первое, что нам необходимо сделать, — изменить запись About (О нас) на основной навигационной панели. В настоящий момент она указывает на /about, но это не подходит для маршрутизации Angular. Даже хотя мы используем режим HTML5 и вы не видите # в URL, все пути должны следовать за #. Поэтому все, что нам нужно сделать, — поменять файл navigation.template.html, вставив # в ссылку About (О нас), как показано в следующем фрагменте кода:

```
<ul class="nav navbar-nav">
<li><a href="/#about">About</a></li>
</ul>
```

Добавляем # в ссылку для страницы About

Что ж, это был несложный первый шаг. Далее мы обратимся к Angular и добавим описание маршрута.

Добавляем описание маршрута

Теперь необходимо добавить маршрут в конфигурацию Angular `$routeProvider` в файле `app_client/app.js`. Для этого можно продублировать запись для домашней страницы и поменять путь, URL шаблона и имя контроллера.

Аналогично тому, как мы делали при создании страницы About (О нас) на стороне сервера, опишем обобщенное представление для простой текстовой страницы, допускающее многократное использование. В листинге 10.9 можно увидеть новый маршрут, добавленный в `config`, со всеми необходимыми изменениями.

Листинг 10.9. Добавляем новое описание маршрута Angular для страницы About (О нас)

```
function config ($routeProvider, locationProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'home/home.view.html',
      controller: 'homeCtrl',
      controllerAs: 'vm'
    })
    .when('/about', {
      templateUrl: '/common/views/genericText.view.html',
      controller: 'aboutCtrl',
      controllerAs: 'vm'
    })
    .otherwise({redirectTo: '/'});
  $locationProvider.html5Mode(true);
}
```

Задаем новый путь — /about

Задаем путь для обобщенного шаблона представления

Сообщаем маршруту о необходимости использовать контроллер 'aboutCtrl'

Оставляем значение controllerAs равным 'vm'

Теперь мы отлично понимаем, что нам необходимо сделать дальше. Этот маршрут включает пока еще не существующие контроллер и шаблон представления. Разберемся сначала с контроллером.

Создаем контроллер

Это будет новый контроллер, так что для него необходим новый файл. Поэтому создаем в `app_client` каталог `about`, а в нем — новый файл `about.controller.js`. Он будет содержать контроллер для страницы `About (О нас)`.

В этом файле создадим **PIFE для защиты областей видимости**, подключим контроллер к приложению `loc8rApp` и, конечно, опишем контроллер. Контроллер в данном случае довольно прост: нам нужно только задать название страницы и ее контент. Контент в данном случае будет тот, который мы использовали в `Express`, а именно выполняющий функцию экспорта `about` в `app_server/controllers/main.js`. Разрывы строк в тексте должны иметь вид `\n`, они понадобятся при дальнейшем чтении этого раздела.

В листинге 10.10 показан полный код файла `about.controller.js`. Я немного урезал текст в основной области контента ради экономии чернил и спасения деревьев.

Листинг 10.10. Создание контроллера `Angular` для страницы `About (О нас)`

```
(function () {

  angular
    .module('loc8rApp')
    .controller('aboutCtrl', aboutCtrl);

  function aboutCtrl() {
    var vm = this;

    vm.pageHeader = {
      title: 'About Loc8r',
    };
    vm.main = {
      content: 'Loc8r was created to help people find places to sit down and
        get a bit of work done.\n\nLorem ipsum dolor sit amet, consectetur
        adipiscing elit.'
    };
  }

})();
```

Как контроллер он довольно прост. Ничего сверхъестественного тут не происходит, мы просто используем переменную `vm` для хранения данных модели, так же, как

поступали с контроллером домашней страницы. Нам, конечно, необходимо сделать так, чтобы приложение знало об этом файле. Следующий фрагмент кода демонстрирует добавление его в массив `appClientFiles` в основном файле `app.js` в Express:

```
var appClientFiles = [
  'app_client/app.js',
  'app_client/home/home.controller.js',
  'app_client/about/about.controller.js',
  'app_client/common/services/geolocation.service.js',
  'app_client/common/services/loc8rData.service.js',
  'app_client/common/filters/formatDistance.filter.js',
  'app_client/common/directives/ratingStars/ratingStars.directive.js'
];
```

При перезапуске приложения Node наш новый файл будет добавлен в используемый в настоящее время единый сокращенный файл. Но если мы сейчас посмотрим на страницу, она окажется пустой, ведь шаблон представления еще не создан.

Создание нового обобщенного шаблона представления

Мы уже описали в функции `config` маршрута, где будет находиться файл нового обобщенного текстового шаблона: `/app_client/common/views/genericText.view.html`. Так что вперед, создайте этот файл. Преобразуем Jade в шаблон Angular, вставляем директивы для разметки и получаем следующее:

```
<navigation></navigation>

<div class="container">
  <page-header content="vm.pageHeader"></page-header>

  <div class="row">
    <div class="col-md-6 col-sm-12">
      <p>{{ vm.main.content }}</p>
    </div>
  </div>

<footer-generic></footer-generic>
</div>
```

И снова ничего необычного. Просто какой-то HTML-код и обычные привязки Angular. Если мы посмотрим на эту страницу в браузере, то увидим, что контент поступает, но разрывы строк не отображаются (рис. 10.3).

Это не очень хорошо. Нам хотелось бы, чтобы текст был удобочитаемым и отображался так, как предполагалось изначально. Если мы с помощью фильтра сумели поменять способ отображения расстояний на домашней странице, то почему бы не сделать то же самое для исправления разрывов строк? Давайте попробуем и создадим новый фильтр.

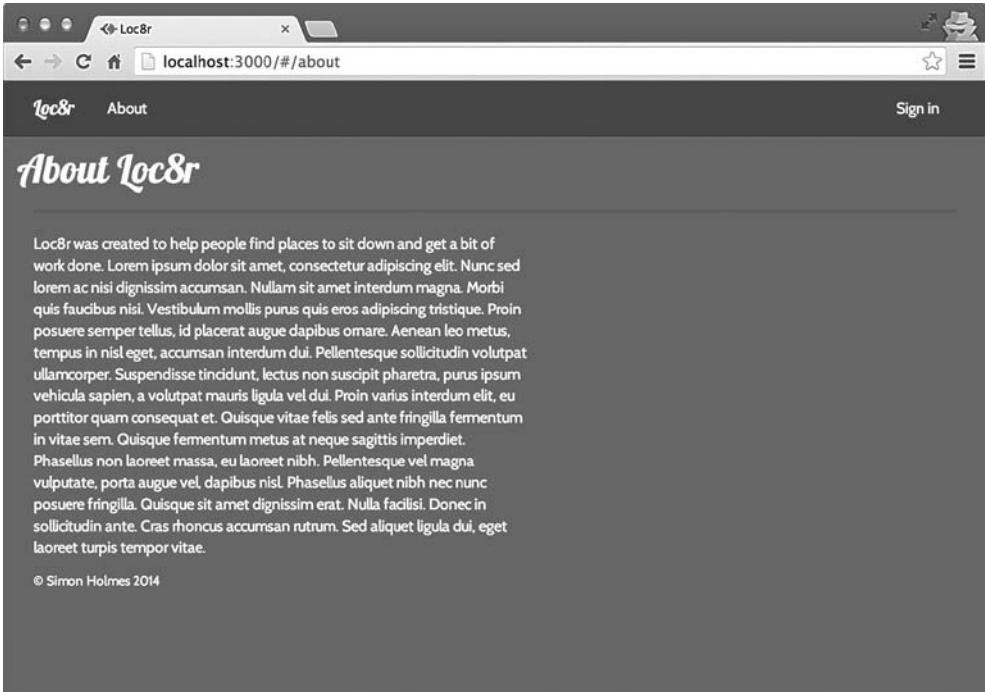


Рис. 10.3. Контент для страницы About (О нас) поступает из контроллера, но разрывы строк игнорируются

10.2.2. Создание фильтра для преобразования разрывов строк

Итак, мы хотим создать фильтр, который примет на входе какой-то текст и заменит каждую копию `\n` тегом `
`. Фактически мы уже решили эту задачу в Jade, воспользовавшись JavaScript-командой `replace`, как показано в следующем фрагменте кода:

```
p !{{(content).replace(/\n/g, '<br/>')}}
```

На Angular нам не удастся сделать это встроенным образом, вместо этого придется создать фильтр и применить его к привязке.

Создаем фильтр `addHtmlLineBreaks`

Вероятно, это будет часто используемый фильтр, так что поместим его в каталог `common\filters` рядом с файлом `formatDistance.filter.js`. Содержимое файла необходимо обернуть в `IFE`, а фильтр — зарегистрировать в приложении.

Сам фильтр довольно прост, он возвращает функцию, принимающую на входе поступающий текст и заменяющую каждый `\n` на `
`. Создайте новый файл `addHtmlLineBreaks.filter.js` и введите туда показанное в следующем фрагменте кода содержимое:

```
(function () {  
  
  angular  
    .module('loc8rApp')  
    .filter('addHtmlLineBreaks', addHtmlLineBreaks);  
  
  function addHtmlLineBreaks () {  
    return function (text) {  
      var output = text.replace(/\n/g, '<br/>');  
      return output;  
    };  
  }  
  
})();
```

Прежде чем делать что-либо с новым фильтром, не забудьте добавить его в массив `appClientFiles` в файле `app.js` в Express. А затем опробуем его.

Применение фильтра к привязке

Применение фильтра к привязке не представляет собой ничего сложного, мы уже несколько раз это делали. Просто добавляем в HTML-код символ вертикальной черты (`|`) после привязанного объекта данных и указываем вслед за ним название фильтра, вот так:

```
<p>{{ vm.main.content | addHtmlLineBreaks }}</p>
```

Несложно, правда? Но если мы опробуем это в браузере, то окажется, что все не так, как мы ожидали. Разрывы строк заменены на `
`, но они отображаются в виде текста, вместо того чтобы визуализироваться как HTML (рис. 10.4).

Хм-м-м-м. Не совсем то, что мы хотели, но по крайней мере фильтр, похоже, работает! Причина вывода подобного результата весьма серьезная: безопасность. Angular защищает вас и ваше приложение от атак злоумышленников, предотвращая внедрение HTML в привязки данных. Рассмотрим, например, обзоры местоположений, писать которые мы даем возможность посетителям. Если туда можно вставить любой HTML, какой только захочется, кто-нибудь легко сможет вставить тег `<script>` и выполнить JavaScript для взлома страницы.

Но существует способ разрешить использование некоторого подмножества HTML в привязке, который мы сейчас и рассмотрим.

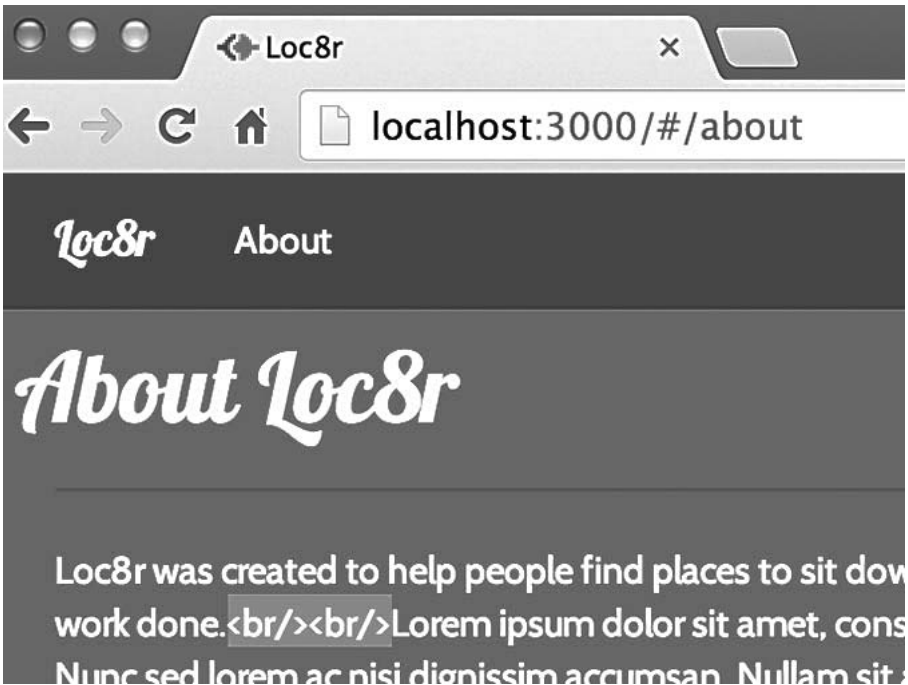


Рис. 10.4. Теги `
`, вставляемые нашим фильтром, визуализируются как текст, а не как теги HTML

10.2.3. Передача HTML в привязку Angular

Мы не первые, у кого есть вполне законная причина для передачи какого-либо HTML-кода в привязку, так что Angular предоставляет такую возможность. Можно использовать сервис `angular-sanitize`, позволяющий включать в привязку данных определенное подмножество HTML-тегов.

Скачиваем `angular-sanitize`

В главе 9 мы скачивали сервис `angular-route` с сайта `code.angularjs.org`, а теперь должны сделать то же самое с `angular-sanitize`: найти нужную ветвь, соответствующую используемой вами редакции Angular (в нашем случае 1.2.19), и скачать два сокращенных файла `angular-sanitize`, а именно `angular-sanitize.min.js` и `angular-sanitize.min.js.map`. Поместите эти файлы рядом с файлами `angular-route` в каталоге `/app_client/lib`.

После этого сделайте их доступными браузеру путем добавления ссылки на JavaScript-файл в `index.html`, как показано в следующем фрагменте кода:

```
<script src="/angular/angular.min.js"></script>
<script src="/lib/angular-route.min.js"></script>
<script src="/lib/angular-sanitize.min.js"></script>
<script src="/angular/loc8r.min.js"></script>
```

Отлично, дальше нужно будет известить приложение, что мы хотим использовать этот сервис.

Добавление ngSanitize в качестве зависимости приложения

Чтобы сообщить приложению, что мы хотим использовать `angular-sanitize`, задействуем тот же подход, который применяли для `ngRoute`, и добавим этот сервис в качестве зависимости в сеттер модуля. В данном случае доступное для приложения название сервиса — `ngSanitize`, так что добавьте его в массив зависимостей в файле `app_clients/app.js`, как показано в следующем фрагменте кода:

```
angular.module('loc8rApp', ['ngRoute', 'ngSanitize']);
```

Теперь, когда сервис доступен приложению, контроллер или фильтр можно не модифицировать. Но нам совсем не обязательно захочется пропускать каждую отдельную привязку данных через этот сервис, скорее мы предпочли бы вручную выбирать, каким из них будет разрешено выполнять синтаксический разбор HTML. Angular опять предусмотрел все для нашего удобства, поскольку для использования `ngSanitize` нужно привязать данные к директиве, а не реализовывать встроенную привязку данных.

Выполняем привязку к элементу HTML как к директиве

Итак, сервис `ngSanitize` не просто работает со всеми привязками данных во всех шаблонах, он предоставляет директиву, к которой вы можете выполнить привязку. Эта директива называется `ng-bind-html`. Как и другие директивы, она добавляется в виде атрибута элемента HTML, причем привязка и фильтр передаются в качестве его значения.

В следующем фрагменте кода показано, как это можно использовать. Здесь передаются привязка данных для контента и фильтр `addHtmlLineBreaks`. Он находится в файле `genericText.view.html`:

```
<divclass="row">
  <div class="col-md-6 col-sm-12">
    <p ng-bind-html="vm.main.content | addHtmlLineBreaks"></p>
  </div>
</div>
```

Сейчас, если вы перезагрузите страницу в браузере, то должны увидеть разрывы страниц, выглядящие так, как показано на рис. 10.5.

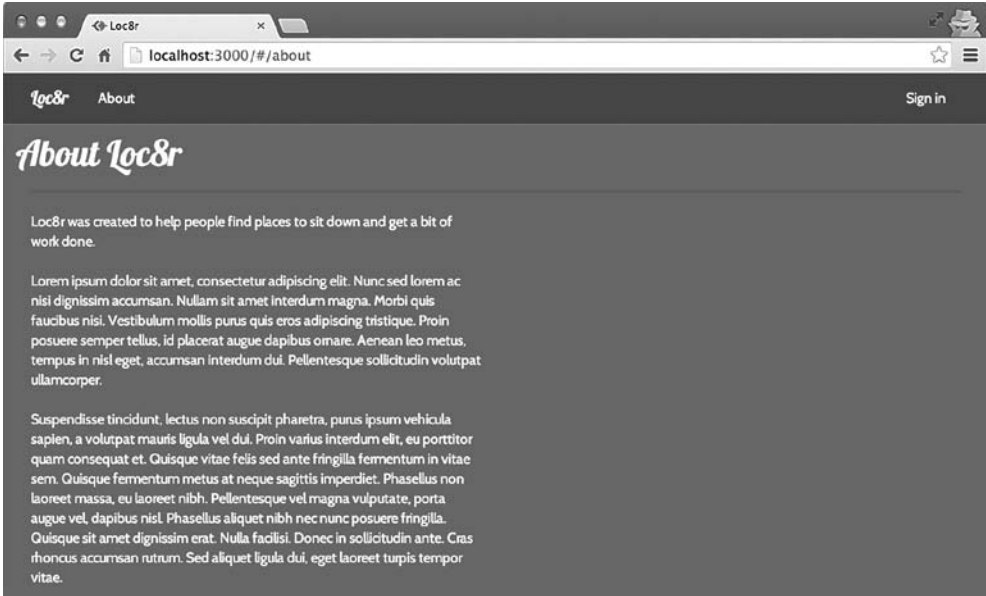


Рис. 10.5. При использовании фильтра `addHtmlLineBreaks` в сочетании с `ngSanitize` мы видим, что разрывы строк наконец-то визуализируются так, как задумывалось

Хорошие новости: всегда приятно видеть, что все получается и несколько частей пазла становятся на свои места. В этом маленьком разделе мы увидели, как добавлять в SPA новую страницу, а также как внедрять HTML-код в привязку Angular. Теперь планируем взглянуть на что-то более интересное, реализовав страницу `Details` (Подробности) в нашем SPA.

10.3. Более сложные представления и параметры маршрутизации

В этом разделе мы планируем добавить в SPA Angular страницу `Details` (Подробности). Одним из критических аспектов тут будет извлечение ID местоположения из параметра URL, чтобы гарантировать, что полученные нами данные правильны. Подобное использование параметров URL — распространенная практика, и это очень удобная методика для любого фреймворка. Нам также понадобится изменить сервис данных для обращения к API с запросом о подробностях конкретного местоположения. При преобразовании представления Jade в шаблон Angular мы обнаружим и еще несколько вещей, которые Angular делает, чтобы нам помочь.

Прежде чем углубиться в эти действия, нам нужно будет создать простейший маршрут, контроллер и представление.

10.3.1. Развертываем фреймворк страницы

Мы уже делали это несколько раз, так что немного ускорим процесс. Нам нужно добавить маршрут из функции `config` и создать задаваемые им файлы контроллера и представления. Контроллер должен будет также добавить его в файлы приложения, чтобы можно было его использовать.

Описываем маршрут страницы

Необходимо добавить новый маршрут в `app_client/app.js`. Так как мы хотели бы принимать на входе параметр URL, то опишем маршрут так же, как мы делали в Express, вставляя переменную `locationid` с предшествующим двоеточием в конце пути (листинг 10.11).

Листинг 10.11. Добавляем путь страницы Details (Подробности) в конфигурацию приложения Angular

```
function config($routeProvider, locationProvider) {
    $routeProvider
        .when('/', {
            templateUrl: 'home/home.view.html',
            controller: 'homeCtrl',
            controllerAs: 'vm'
        })
        .when('/about', {
            templateUrl: '/common/views/genericText.view.html',
            controller: 'aboutCtrl',
            controllerAs: 'vm'
        })
        .when('/location/:locationid', {
            templateUrl: '/locationDetail/locationDetail.view.html',
            controller: 'locationDetailCtrl',
            controllerAs: 'vm'
        })
        .otherwise({redirectTo: '/'});
    $locationProvider.html5Mode(true);
}
```

Создаем файл контроллера

Шаблон и контроллер страницы Details (Подробности) будут тесно сцеплены, они вряд ли будут работать с какими-то другими шаблонами или контроллерами. Учитывая это, поместим их вместе в один и тот же каталог. Создадим новый каталог `locationDetail` в папке `app_client` и поместим фреймворк контроллера, показанный в листинге 10.12, в новый файл `locationDetail.controller.js`.

Листинг 10.12. Фреймворк контроллера для страницы Details (Подробности)

```
(function () {

  angular
    .module('loc8rApp')
    .controller('locationDetailCtrl', locationDetailCtrl);

  function locationDetailCtrl () {
    var vm = this;
    vm.pageHeader = {
      title: 'Location detail page'
    };
  }

})();
```

Важный шаг, о котором легко забыть: *добавить этот файл в массив appClientFiles в файле Express app.js.*

Создаем шаблон представления

В том же каталоге, где находится файл контроллера, создайте файл шаблона представления с именем `locationDetail.view.html`. Пока что мы просто разместим в нем стандартный каркас для страницы, как можно видеть из следующего фрагмента кода:

```
<navigation></navigation>

<div class="container">
  <page-header content="vm.pageHeader"></page-header>

  <footer-generic></footer-generic>
</div>
```

Меняем ссылки в списке на домашней странице

Файлы уже сделаны, но нужно также обеспечить возможность переходить на страницу Details (Подробности) из списка на домашней странице. Аналогично тому, как мы делали со ссылкой About (О нас), нам нужно добавить # перед ссылками в списке, чтобы у Angular был к ним доступ.

Найдите в файле `home.view.html` строку, визуализирующую название местоположения, — она располагается в теге `<h4>`. В соответствии с показанным в следующем фрагменте кода добавьте `/#` перед `href`:

```
<h4>
<a href="/#/location/{{ location._id }}">{{ location.name }}</a>
```

Вот и все, фундамент готов. Теперь рассмотрим извлечение параметра URL и его использование для получения нужных данных.

10.3.2. Параметры URL в контроллерах и сервисах

Довольно часто бывает нужно получить и использовать параметр URL, поэтому неудивительно, что для решения этой задачи у Angular имеется встроенный сервис. Он носит название `$routeParams` и чрезвычайно прост в использовании.

Использование `$routeParams` для получения параметров URL

Чтобы использовать `$routeParams` в контроллере, необходимо внедрить его в виде зависимости и передать в функцию. После этого `$routeParams` служит объектом, хранящим все соответствующие параметры URL. Использовать его очень легко: просто обратитесь к нему в коде, поменяв функцию `locationDetailCtrl` так, как показано в следующем фрагменте кода:

```
locationDetailCtrl.$inject = ['$routeParams'];
function locationDetailCtrl ($routeParams) {
  var vm = this;
  vm.locationid = $routeParams.locationid;
  vm.pageHeader = {
    title: vm.locationid
  };
}
```

Внедряем сервис `$routeParams` в контроллер, защитив его от сокращения

Передаем `$routeParams` в контроллер, чтобы можно было его использовать

Получаем из `$routeParams.locationid` и сохраняем его в модели представления

Используем `locationid` в названии страницы

Видите, насколько просто? На скриншоте на рис. 10.6 видно, что идентификатор местоположения извлекается из URL и выводится в заголовке страницы. Это еще не все, но приятно видеть, что приложение работает.



Рис. 10.6. С помощью `$routeParams` можно извлечь из URL идентификатор местоположения и использовать его в контроллере, что продемонстрировано тут путем его вывода в заголовке страницы

Воспользуемся этим идентификатором местоположения для получения каких-нибудь данных из API, чтобы страница опять стала пригодной для использования. Для этого нам понадобится создать сервис данных, чтобы обратиться к нужному API.

Создание сервиса данных для обращения к API

В построенном в главе 6 API мы создали конечную точку, которая принимает идентификатор местоположения и возвращает связанные с ним данные. Соответствующий путь URL — `/api/location/:locationid`. Чтобы опрашивать этот URL, добавим в сервис `loc8rData` новый метод.

Листинг 10.13 демонстрирует простоту выполнения этого, добавляя и делая доступным метод `locationById`, принимающий параметр `locationid`. В дальнейшем этот метод использует `locationid` в обращении `$http` к конечной точке API.

Листинг 10.13. Добавление метода в сервис данных для обращения к API

```
function loc8rData ($http) {
  var locationByCoords = function (lat, lng) {
    return $http.get('/api/locations?lng=' + lng + '&lat=' + lat +
      '&maxDistance=20');
  };

  var locationById = function (locationid) {
    return $http.get('/api/locations/' + locationid);
  };

  return {
    locationByCoords : locationByCoords,
    locationById : locationById
  };
}
```

Создаем новый метод `locationById`, принимающий на входе параметр `locationid`

...и использующий `locationid` в обращении к API

Делаем метод `locationById` доступным для вызова из контроллера

Сервис для получения данных

Для использования данного сервиса нам понадобится внедрить сервис `loc8rData` в контроллер. Сразу после этого можно будет следовать паттерну, который мы использовали для домашней страницы, где обращались к API для получения списка местоположений. Напомню, что сервис данных применяет метод `$http` асинхронно: по завершении будет вызван один из двух промисов — `success` или `error`.

В листинге 10.14 внедрим сервис `loc8rData` в функцию `locationDetailCtrl` и вызовем метод `locationById`, передав ему в качестве параметра идентификатор местоположения. В случае успешного выполнения запроса сохраним возвращенные данные в модели представления в `vm.data.location` и отобразим наименование местоположения в заголовке страницы.

Листинг 10.14. Использование сервиса из контроллера для получения данных о местоположении

```

locationDetailCtrl.$inject = ['$routeParams', 'loc8rData'];
function locationDetailCtrl ($routeParams, loc8rData) {
    var vm = this;
    vm.locationid = $routeParams.locationid;

    ▶loc8rData.locationById(vm.locationid)
      .success(function(data) {
        vm.data = { location: data };
        vm.pageHeader = {
          title: vm.data.location.name
        };
      })
      .error(function (e) {
        console.log(e);
      });
}

```

Внедряем сервис loc8rData в качестве зависимости и передаем в контроллер

Если запрос оказался успешен, сохраняем возвращенные данные в модели представления

Выводим название местоположения в заголовке страницы

Если запрос завершился неудачей, выводим в консоль браузера сообщение об ошибке

Вызываем метод locationById, передавая ему идентификатор местоположения в качестве параметра

Весьма действенно и совсем не сложно. Работа этого кода, выводящего название местоположения в заголовке страницы, показана на рис. 10.7.

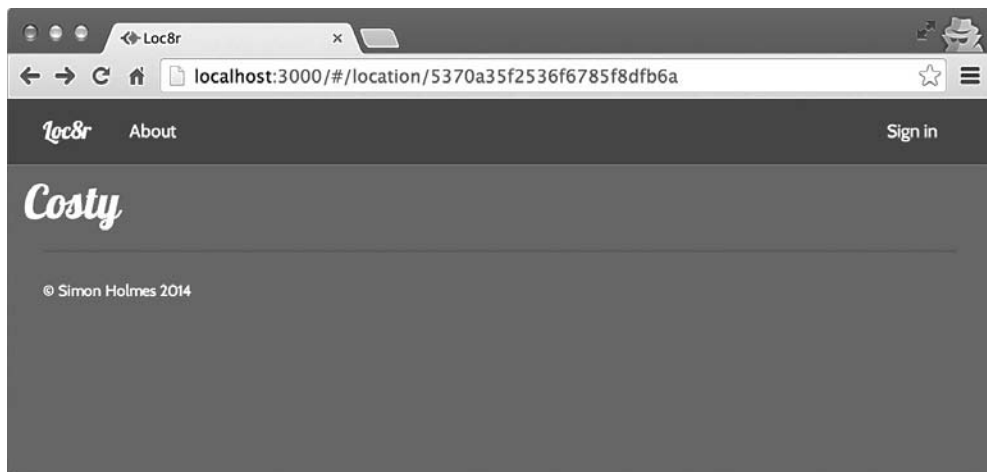


Рис. 10.7. Подтверждаем получение данных из API, выводя название местоположения в заголовке страницы

Мы завершили еще один этап. Теперь необходимо собрать представление во-едино.

10.3.3. Создаем представление для страницы Details (Подробности)

Следующий этап — перестройка представления. У нас есть шаблон Jade с привязками данных Jade, и нужно преобразовать его в HTML-код с привязками данных Angular. Нужно создать довольно много привязок данных, а также несколько циклов на основе `ng-repeat`. Мы снова воспользуемся директивой `rating-stars` для отображения общей оценки и оценки для каждого отзыва. Также нам нужно будет обеспечить разрывы строк в тексте отзыва с помощью фильтра `addHtmlLineBreaks`.

Создаем основной шаблон

Листинг 10.15 демонстрирует готовый код (привязки выделены полужирным шрифтом). Этот код необходимо вставить в файл `locationDetail.view.html` между заголовком страницы и нижним колонтитулом. Кое-что, например часы работы, мы пока пропустили — заполним эти пробелы, когда все будет готово и протестировано.

Листинг 10.15. Представление Angular для страницы Details (Подробности)

```

<div class="row">
  <div class="col-xs-12 col-md-9">
    <div class="row">
      <div class="col-xs-12 col-sm-6">
        <p class="rating" rating-stars
          rating="vm.data.location.rating"></p>
        <p>{{ vm.data.location.address }}</p>
        <div class="panel panel-primary">
          <div class="panel-heading">
            <h2 class="panel-title">Opening hours</h2>
          </div>
          <div class="panel-body">
            <!-- Здесь будут находиться часы работы -->
          </div>
        </div>
        <div class="panel panel-primary">
          <div class="panel-heading">
            <h2 class="panel-title">Facilities</h2>
          </div>
          <div class="panel-body">
            <span class="label label-warning label-facility"
              ng-repeat="facility in vm.data.location.facilities">
              <span class="glyphicon glyphicon-ok"></span>
              {{ facility }}
            </span>
          </div>
        </div>
      </div>
      <div class="col-xs-12 col-sm-6 location-map">
        <div class="panel panel-primary">

```

Используем директиву `rating-stars` для отображения средней оценки по местоположению

Цикл по предоставляемым услугам

```
<div class="panel-heading">
  <h2 class="panel-title">Location map</h2>
</div>
<div class="panel-body">
  
  </div>
</div>
</div>
</div>
<div class="row">
  <div class="col-xs-12">
    <div class="panel panel-primary review-panel">
      <div class="panel-heading"><a href=""
        class="btn btn-default
        pull-right">Add review</a>
      <h2 class="panel-title">Customer reviews</h2>
      </div>
      <div class="panel-body review-container">
        <div class="review" ng-repeat="review in
          vm.data.location.reviews">
          <div class="row">
            <div class="well well-sm review-header">
              <span class="rating" rating-stars
                rating="review.rating"></span>
              <span class="reviewAuthor">{{ review.author }}</span>
              <small class="reviewTimestamp">{{ review.createdOn }}</
                small>
            </div>
            <div class="col-xs-12">
              <p ng-bind-html="review.reviewText
                | addHtmlLineBreaks"></p>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
  <div class="col-xs-12 col-md-3">
    <p class="lead">{{ vm.data.location.name }} is on Loc8r because
      it has accessible wifi and space to sit down with your laptop
      and get some work done.</p>
    <p>If you've been and you like it - or if you don't - please
      leave a review to help other people just like you.</p>
  </div>
</div>
</div>
```

Используем директиву rating-stars, чтобы отобразить оценку для каждого отзыва

Цикл по отзывам

Применяем фильтр addHtmlLineBreaks к текстам отзывов и привязываем в виде HTML

Теперь листинг стал весьма длинным! Но этого и следовало ожидать, ведь на странице **Details** (Подробности) происходит многое. Если вы посмотрите на эту страницу в браузере, то увидите, что неплохо было бы исправить еще кое-что: мы до сих пор не отображаем часы работы, отзывы показываются, начиная с самого старого, а данные отзывов необходимо отформатировать.

Добавляем логику в стиле **if-else** с использованием **ng-switch** для отображения часов работы

Нет ничего необычного в наличии в шаблоне какого-либо варианта логики **if-else** («если... то... иначе...») для отображения различных частей HTML-кода в зависимости от какого-то параметра. Для каждого варианта часов работы мы хотели бы отображать диапазон дней, а также или сообщение о том, что закрыто, или время начала и завершения работы. В шаблоне **Jade** у нас было немного такой логики — простой оператор **if**, проверяющий флаг **closed** на равенство **true**, как показано в следующем фрагменте кода:

```
if time.closed
| closed
else
| #{time.opening} - #{time.closing}
```

Нам хотелось бы чего-то подобного в представлении **Angular**. Вместо использования **if** **Angular** работает аналогично методу **switch** **JavaScript**, в котором вверху задается условное выражение, которое вы хотели бы проверить, а затем указываются различные варианты действий в зависимости от значения условного выражения.

Ключевые директивы тут: **ng-switch-on**, служащая для описания исходного условного выражения, **ng-switch-when** для указания конкретных значений и **ng-switch-default** как запасной вариант на случай, если ни одно из конкретных значений не соответствует условному выражению. Все это показано в действии в листинге 10.16, где мы добавляем часы работы в HTML-представление.

Листинг 10.16. Используем **ng-switch** для отображения часов работы

```
Выбираем действие
на основе
значения time.closed

<div class="panel-heading">
  <h2 class="panel-title">Opening hours</h2>
</div>
<div class="panel-body">
  <p ng-repeat="time in vm.data.location.openingTimes"
  >ng-switch on="time.closed">
    {{ time.days }} :
    <span class="opening-time" ng-switch-when="true">closed</span>
    <span class="opening-time" ng-switch-default>{{ time.opening +
    Если time.closed
    равно true, просто
    выводим "closed"
```

```

    " - " + time.closing }}</span>
  </p>
</div>

```

В противном случае выполняем действие по умолчанию — отображаем время начала и завершения работы

Теперь в нашем представлении имеется фрагмент логики. Обратите внимание на то, что, так как все команды `ng-switch` представляют собой директивы, их необходимо добавить в теги HTML. Отлично, давайте сделаем так, чтобы сначала отображались наиболее свежие отзывы.

Меняем порядок отображения списка с помощью фильтра `orderBy`

Для упрощения сортировки элементов в списке `ng-repeat` Angular предоставляет фильтр `orderBy`. Отмечу, что `orderBy` можно использовать только для массивов, так что для сортировки объекта его нужно сначала преобразовать в массив.

Фильтр `orderBy` может принимать несколько параметров. Во-первых, необходимо указать, какой список нужно отсортировать. Это может быть функция, но чаще указывают имя свойства сортируемого списка. Именно так мы и поступим, использовав свойство `createdOn` каждого отзыва.

Второй параметр — необязательный, он определяет, нужно ли изменить порядок сортировки на обратный. Это булево значение, по умолчанию, если не задано, оно равно `false`. Мы установим его в `true`, так как хотим выводить сначала более поздние отзывы.

Листинг 10.17 демонстрирует изменения в шаблоне представления, выполненные для добавления в директиву `ng-repeat` фильтра.

Листинг 10.17. Отображаем отзывы в соответствии с датами с помощью фильтра `orderBy`

Добавляем в `ng-repeat` фильтр `orderBy`, задавая свойство для сортировки и параметр обратной сортировки

```

<div class="review" ng-repeat="review in vm.data.location.reviews |
➤ orderBy:'createdOn':true">
  <div class="well well-sm review-header">
    <span class="rating" rating-stars rating="review.rating"></span>
    <span class="reviewAuthor">{{ review.author }}</span>
    <small class="reviewTimestamp">{{ review.createdOn }}</small>
  </div>
  <div class="col-xs-12">
    <p ng-bind-html="review.reviewText | addHtmlLineBreaks"></p>
  </div>
</div>

```

Теперь, если вы перезагрузите страницу, вы увидите, что отзывы отображаются в нужном порядке — сначала самые новые. Правда, понять, что это так, нелегко, ведь формат даты не очень удобен для пользователя. Давайте это исправим.

Исправляем формат даты с помощью фильтра date

Еще один поставляемый с Angular фильтр — `date`, форматирующий заданную дату в нужном стиле. Он принимает только один параметр — формат для даты.

Чтобы использовать свое форматирование, вам необходимо передать строку, описывающую желаемый формат вывода. В ней может быть слишком много различных опций, чтобы рассматривать их тут, но уловить основную идею формата несложно. Чтобы получить формат «1 December 2014», необходимо задать строку формата `'dMMMMуууу'`, как показано в листинге 10.18.

Листинг 10.18. Применяем фильтр `date` для форматирования дат отзывов

```
<div class="review" ng-repeat="review in vm.data.location.reviews |
  orderBy:'createdOn':true">
  <div class="well well-sm review-header">
    <span class="rating" rating-stars rating="review.rating"></span>
    <span class="reviewAuthor">{{ review.author }}</span>
    <small class="reviewTimestamp">{{ review.createdOn | date : 'd MMMM уууу'
    }}</small>
  </div>
  <div class="col-xs-12">
    <p ng-bind-html="review.reviewText | addHtmlLineBreaks"></p>
  </div>
</div>
```

На этом мы завершаем работу с макетом и форматированием страницы `Details` (Подробности). Следующий, последний этап заключается в обеспечении возможности добавления отзывов, но мы отбросим идею использования для этого отдельной страницы. Взамен ради лучшего пользовательского взаимодействия с приложением реализуем эту возможность в модальном всплывающем окне на странице `Details` (Подробности).

10.4. Использование компонентов AngularUI для создания модального всплывающего окна

В этом разделе рассмотрим добавление сторонних компонентов в приложения Angular, а также отправку данных форм. Мы предоставим пользователям возможность добавить в `Loc8r` отзывы непосредственно со страницы `Details` (Подробности), создав модальное всплывающее окно, отображаемое при нажатии на кнопку `Add Review` (Добавить отзыв). Модальное окно будет выводить форму отзыва, позволяя пользователю ввести свое имя, оценку и отзыв. При подтверждении отправки отзывов мы будем отправлять их в API, чтобы сохранить в базе данных, и добавлять

в список отзывов на странице. При этом пользователи не уйдут со страницы Details (Подробности).

Первый этап — создание модального всплывающего окна, отображаемого при нажатии на кнопку Add Review (Добавить отзыв).

10.4.1. Подготовка AngularUI

Вместо создания модального окна с нуля и разработки всего управляющего им кода мы можем переложить эту непростую работу на плечи команды AngularUI. Они создали немало компонентов Bootstrap, написанных на чистом Angular. Эти компоненты основаны только на Angular, без всякого jQuery или собственного JavaScript Bootstrap.

Загружаем AngularUI

Получить AngularUI можно по адресу <http://angular-ui.github.io/bootstrap/>. Вы можете скачать всю библиотеку, включающую около 20 компонентов. Правда, если вы хотите использовать в своем приложении один-единственный компонент, получится стрельба из пушки по воробьям. Вместо этого вы можете создать пользовательскую сборку, нажав на кнопку Build и выбрав нужные компоненты — в нашем случае Modal (рис. 10.8), — и скачать модули.

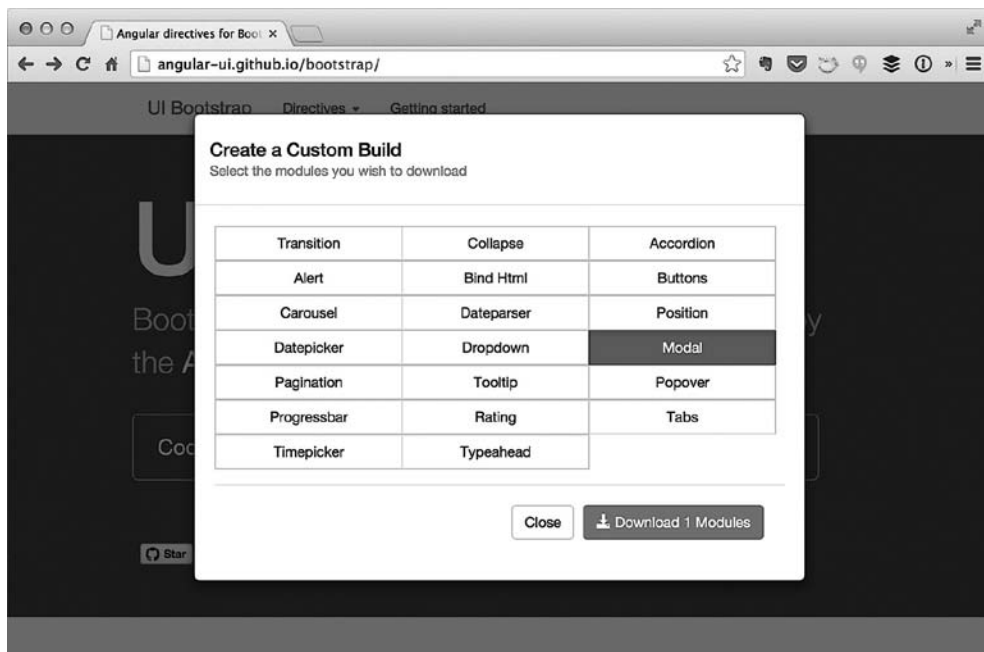


Рис. 10.8. Создаем пользовательскую сборку AngularUI только с нужными компонентами

Использование пользовательской сборки значительно уменьшит размер файла — в данном случае с 65 до 13 Кбайт. Откройте скачанный вами файл ZIP-архива и скопируйте два сокращенных JavaScript-файла в каталог `lib` в `app_client`.

Теперь можно сослаться на них в файле `index.html` рядом с другими файлами библиотек, перед основным файлом приложения, как показано в следующем фрагменте кода:

```
<script src="/angular/angular.min.js"></script>
<script src="/lib/angular-route.min.js"></script>
<script src="/lib/angular-sanitize.min.js"></script>
<script src="/lib/ui-bootstrap-custom-0.12.0.min.js"></script>
<script src="/lib/ui-bootstrap-custom-tpls-0.12.0.min.js"></script>
<script src="/angular/loc8r.min.js"></script>
```

После добавления этих файлов можно использовать их в приложении.

Используем AngularUI в приложении

Для использования компонентов AngularUI в приложении необходимо описать их как зависимости на уровне приложения. После этого нужно описать модальный компонент как зависимость контроллера для страницы, на которой мы собираемся его использовать.

Чтобы добавить AngularUI в качестве зависимости приложения, нужно просто добавить `'ui.bootstrap'` в массив зависимостей в файле `app_client/app.js`, как показано в следующем фрагменте кода:

```
angular.module('loc8rApp', ['ngRoute', 'ngSanitize', 'ui.bootstrap']);
```

После этого нужно сообщить контроллеру о желании использовать модальный компонент путем внедрения зависимости `$modal`, как показано в следующем фрагменте кода (не забудьте, что необходимо передать его как параметр в функцию контроллера и добавить в массив `$inject`):

```
locationDetailCtrl.$inject = ['$routeParams', '$modal', 'loc8rData'];
function locationDetailCtrl ($routeParams, $modal, loc8rData) {
```

Выполнив два этих действия, мы можем двигаться дальше и создать модальное окно.

10.4.2. Добавление и использование обработчика нажатий

Наша цель — напомним себе, чего именно мы хотим добиться, — заключается в отображении всплывающего диалогового окна при нажатии пользователем кнопки `Add Review` (Добавить отзыв). Так что необходимо добавить обработчик нажатий для этой кнопки, создать соответствующую функцию в контроллере, а затем разобратся с созданием модального окна.

Добавляем обработчик ng-click

Для создания прослушивателя на предмет нажатий, который бы вызывал метод нашего приложения Angular, лучше воспользоваться обработчиком нажатий `ng-click`, а не `href` или `onclick`. Он ведет себя подобно `onclick`, но позволяет обращаться к методам модели представления.

В следующем фрагменте кода добавим в файле `locationDetail.view.html` к кнопке Add Review (Добавить отзыв) обработчик `ng-click`, который будет вызывать функцию `popupReviewForm` из модели представления:

```
<a ng-click="vm.popupReviewForm()" class="btn btn-default pull-right">Add review</a>
```

Отлично, следующий этап — создание метода `popupReviewForm` в контроллере.

Добавляем вызываемый обработчиком нажатий метод

Создание метода в контроллере сводится к простому объявлению `vm.popupReviewForm` в виде функции. В листинге 10.19 добавим новую функцию и заставим ее генерировать предупреждающее сообщение, чтобы мы смогли проверить, что метод и `ng-click` работают друг с другом должным образом.

Листинг 10.19. Добавление метода в контроллер

```
function locationDetailCtrl ($routeParams, $modal, loc8rData) {
  var vm = this;
  vm.locationid = $routeParams.locationid;

  loc8rData.locationById(vm.locationid)
    .success(function(data) {
      vm.data = { location: data };
      vm.pageHeader = {
        title: vm.data.location.name
      };
    })
    .error(function(e) {
      console.log(e);
    });

  vm.popupReviewForm = function () {
    alert("Let's add a review!");
  };
}
```

Итак, если мы выполним эти изменения и перейдем на страницу подробностей о местоположении, то при нажатии кнопки Add Review (Добавить отзыв) должны увидеть предупреждающее сообщение. Рисунок 10.9 демонстрирует это все в действии, подтверждая, что мы должным образом связали кнопку и метод.

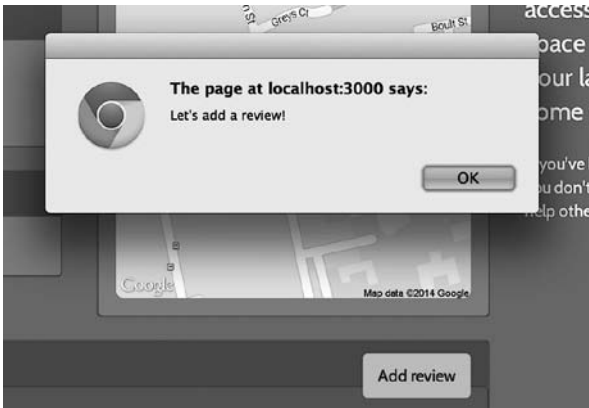


Рис. 10.9. Проверяем функционирование обработчика нажатий Add Review (Добавить отзыв) с помощью простого окна предупреждения

10.4.3. Создаем модальное окно Bootstrap с помощью AngularUI

Теперь настало время создать модальное диалоговое окно. Хотя здесь мы такого не делаем, вполне возможно, что у вас на странице будет несколько действий, которые могут вызвать появление модального окна. Чтобы избежать перекрестного смешивания кода, каждое модальное окно создается в виде отдельного экземпляра с собственным шаблоном и контроллером.

Давайте посмотрим на описание экземпляра, а затем добавим представление и контроллер.

Описываем экземпляр модального окна AngularUI

Для описания нового экземпляра модального окна используем обработчик `popupReviewForm` из предыдущего раздела. Мы присвоим ему URL шаблона и контроллер аналогично тому, как делали с описаниями директив и маршрутов.

Для этой цели применяется следующий синтаксис:

```
vm.popupReviewForm = function () {
  var modalInstance = $modal.open({
    templateUrl: '/reviewModal/reviewModal.view.html',
    controller: 'reviewModalCtrlasvm',
  });
};
```

Обратите внимание на то, что здесь мы применяем иной подход к синтаксису `controllerAs`. Компонент модального окна может использовать этот подход, но пока что не поддерживает применение параметра `controllerAs` для его задания. Вместо него мы встраиваем задание имени модели представления точно так же, как описывали бы его внутри элемента HTML.

Добавляем представление модального окна

Чтобы создать HTML для модального окна отзыва, скомбинируем уже имеющуюся форму отзыва и разметку шаблона для модального окна Bootstrap. Затем добавим некоторые привязки данных к полям формы и опишем функцию, которую можно будет использовать для отмены модального окна.

Если мы соберем все это вместе, то получим листинг 10.20. Сохраним его в заданном в листинге 10.19 в качестве URL шаблона месте — `reviewModal.view.html` в каталоге `app_client/reviewModal/`.

Листинг 10.20. HTML-представление для всплывающего модального окна

```

<div class="modal-content">
  <form id="addReview" name="addReview" role="form"
    class="form-horizontal">
    <div class="modal-header">
      <button type="button" ng-click="vm.modal.cancel()"
        class="close"><span aria-hidden="true">×</span><span
          class="sr-only">Close</span></button>
      <h4 id="myModalLabel" class="modal-title">Add your
        review for {{vm.locationName }}</h4>
    </div>
    <div class="modal-body">
      <div class="form-group">
        <label for="name" class="col-xs-2 col-sm-2
          control-label">Name</label>
        <div class="col-xs-10 col-sm-10">
          <input id="name" name="name" required="required"
            ng-model="vm.formData.name" class="form-control"/>
        </div>
      </div>
      <div class="form-group">
        <label for="rating" class="col-xs-10 col-sm-2
          control-label">Rating</label>
        <div class="col-xs-12 col-sm-2">
          <select id="rating" name="rating" ng-model="vm.formData.rating"
            class="form-control input-sm">
            <option>5</option>
            <option>4</option>
            <option>3</option>
            <option>2</option>
            <option>1</option>
          </select>
        </div>
      </div>
      <div class="form-group">
        <label for="review" class="col-sm-2 control-label">Review</label>
        <div class="col-sm-10">
          <textarea id="review" name="review" rows="5"
            required="required" ng-model="vm.formData.reviewText"
            class="form-control"></textarea>
        </div>
      </div>
    </div>
  </form>
</div>

```

Добавляем кнопку Close (Закреть) и задаем срабатывающий при ее нажатии метод модели представления

Добавляем привязки данных для полей формы

```

        </div>
      </div>
    </div>
    <div class="modal-footer">
      <button ng-click="vm.modal.cancel()" type="button" class="btn btn-
        default">Cancel</button>
      <button type="submit" class="btn
        btn-primary">Submit review</button>
    </div>
  </form>
</div>

```

В кнопке Cancel (Отмена) срабатывает тот же метод, что и в кнопке Close (Закреть)

Ничего особенно сложного, просто большое количество разметки. Конечно, для использования этого представления нужно будет создать контроллер.

Создаем контроллер модального окна

При описании модального окна в контроллере представления местоположения мы указали имя контроллера, которое будем использовать, — `reviewModalCtrl`. Теперь пришло время создать его в файле `reviewModal.controller.js`, возле только что созданного представления.

Начнем с простейшей конструкции контроллера. У контроллера модального окна имеется созданная компонентом AngularUI зависимость `$modalInstance`, которую мы внедрим. У `$modalInstance` есть метод `dismiss`, который можно вызвать нажатием кнопок `Cancel` (Отмена) или `Close` (Закреть). Для этого создадим метод `vm.modal.cancel`, на который сошлемся в представлении и который используем для закрытия модального окна. Все это объединено в листинге 10.21.

Листинг 10.21. Первоначальный вариант контроллера модального окна для отзыва
(function () {

```

angular
  .module('loc8rApp')
  .controller('reviewModalCtrl', reviewModalCtrl);

reviewModalCtrl.$inject = ['$modalInstance'];
function reviewModalCtrl ($modalInstance) {
  var vm = this;

  vm.modal = {
    cancel : function () {
      $modalInstance.dismiss('cancel');
    }
  };
}

})();

```

Внедряем `$modalInstance` в контроллер

Создаем метод `vm.modal.cancel()` и используем его для вызова метода `$modalInstance.dismiss`

Когда все будет готово, не забудьте добавить контроллер в массив конкатенируемых сценариев в файле Express `app.js`. Если вы сейчас перезагрузите страницу и нажмете кнопку **Add Review** (Добавить отзыв), то увидите отображение модального всплывающего окна (рис. 10.10). Нажатие кнопки **Cancel** (Отмена) или где-нибудь вне модального окна должно его закрыть.

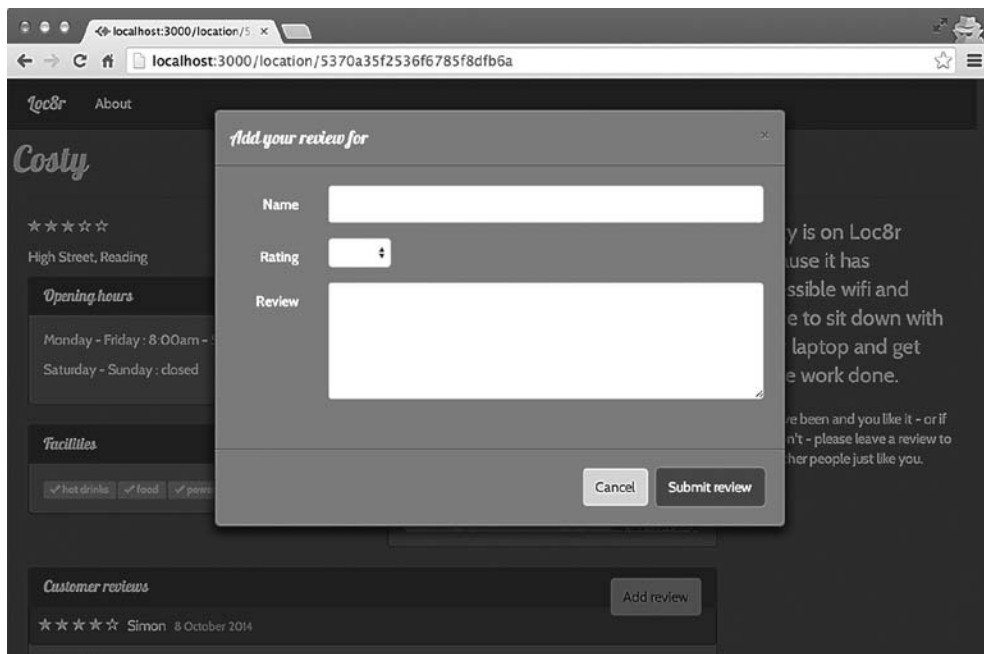


Рис. 10.10. Модальное всплывающее окно в действии — отображает форму для отзыва

Неплохое начало, но в заголовке модального окна не отображается название местоположения. Давайте исправим это, передав данные в модальное окно.

10.4.4. Передача данных в модальное окно

Передача данных из контроллера страницы в представление модели контроллера модального окна — процесс, состоящий из трех этапов.

1. В описании модального окна разрешаем переменные, которые собираемся использовать.
2. Внедряем их в качестве зависимостей в контроллер модального окна.

3. Устанавливаем их соответствие объектам модели представления модального окна.

Разрешаем переменные в описании экземпляра модального окна

Первый шаг при передаче данных контроллеру модального окна состоит в обеспечении доступности переменных в описании экземпляра модального окна. Это делается с помощью параметра `resolve`. Параметру `resolve` соответствует объект, содержащий один или несколько параметров, которые вы хотели бы использовать в модальном окне. Каждому параметру необходимо установить в соответствие функцию, возвращающую значение или объект.

Нам необходимо получить в модальном окне доступ к идентификатору и названию местоположения, поэтому мы разрешим параметр `locationData`, сделав так, чтобы он возвращал объект, содержащий как идентификатор, так и название местоположения. В листинге 10.22 показаны дополнения, которые нужно перенести в описание экземпляра модального окна.

Листинг 10.22. Использование параметра `resolve` для передачи значений переменных в модальное окно

```

Добавляем параметр resolve,
задавая соответствующий объект

var modalInstance = $modal.open({
  templateUrl: '/reviewModal/reviewModal.view.html',
  controller: 'reviewModalCtrl as vm',
  resolve : {
    locationData : function () {
      return {
        locationid : vm.locationid,
        locationName : vm.data.location.name
      };
    }
  }
});

```

Добавляем параметр, устанавливая ему в соответствие функцию

Функция должна возвращать объект или отдельное значение

Это даст контроллеру модального окна возможность использовать параметр `locationData`, если мы внедрим его как зависимость.

Внедряем разрешенные параметры в качестве зависимости и добавляем их в модель представления

Чтобы контроллер модального окна использовал только что созданный нами параметр, необходимо внедрить его как зависимость. Мы сделаем это точно так же, как внедряли бы любую другую зависимость, как показано в следующем фрагменте

кода. Воспользуемся также возможностью сохранить этот параметр в виде свойства модели представления.

```
reviewModalCtrl.$inject = ['$modalInstance', 'locationData'];
function reviewModalCtrl ($modalInstance, locationData) {
  var vm = this;
  vm.locationData = locationData; ←
```

Внедряем новый параметр из описания модального окна

Сохраняем параметр в модели представления

Теперь, когда это сделано, мы можем использовать в модальном окне значения из параметра `locationData`.

Использование передаваемых данных

Теперь, когда данные доступны в модели представления модального окна, мы можем использовать их в привязках в представлении модального окна. Следующий фрагмент кода показывает, как мы изменили название модального окна для отображения названия местоположения:

```
<h4 id="myModalLabel" class="modal-title">Add your review for {{
  vm.locationData.locationName }}</h4>
```

Перезагрузив страницу в браузере и снова нажав кнопку Add Review (Добавить отзыв), вы увидите работу этого кода (рис. 10.11).

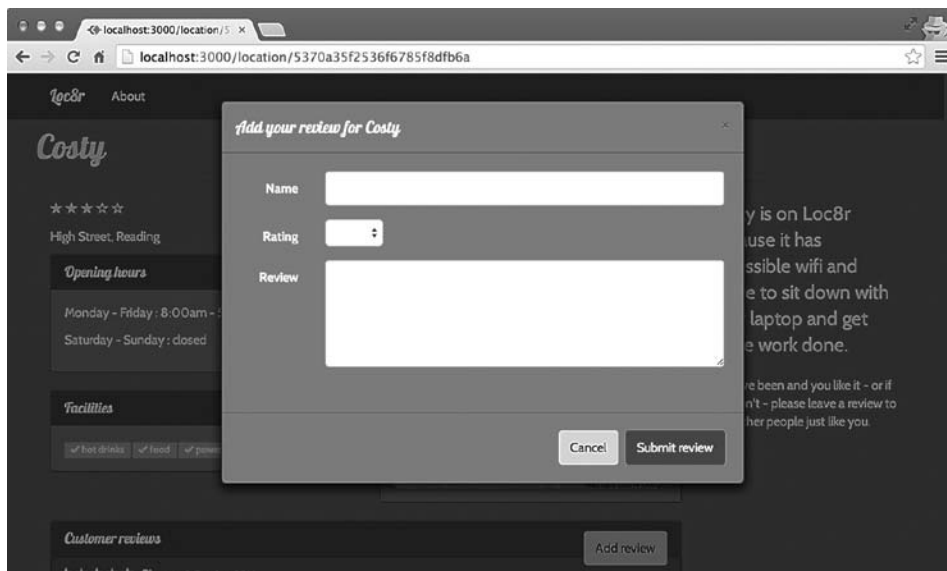


Рис. 10.11. Отображение названия местоположения в модальном всплывающем окне

Отлично, все выглядит как полагается. Последнее, что нам нужно сделать, — привязать форму, чтобы при подтверждении ее отправки добавлялся отзыв.

10.4.5. Форма для отправки отзыва

Теперь пришло время заставить форму для отзывов работать и действительно добавлять отзывы в базу данных при их отправке. Процесс состоит из нескольких шагов.

1. Сделать так, чтобы Angular обрабатывал форму при ее отправке.
2. Проверять форму, чтобы принимались только полные данные.
3. Добавить в сервис `loc8rData` обработчик POST.
4. Выполнить POST данных отзыва в сервис.
5. Вставить отзыв в список на странице `Details` (Подробности).

Добавляем обработчики форм `onSubmit`

При работе с формами в HTML у вас обычно есть действие и метод для извещения браузера о том, куда отправлять данные, а также используемый метод HTTP-запроса. Если вы хотите что-то делать с данными форм перед отправкой с помощью JavaScript, может существовать также обработчик событий `onSubmit`.

В SPA Angular нам не хотелось бы, чтобы при отправке формы на другой URL выполнялся переход на другую страницу. Желательно, чтобы все обрабатывал Angular. Для этой цели можно воспользоваться прослушивателем `ng-submit` фреймворка Angular, чтобы вызвать функцию из модели представления. Следующий фрагмент кода демонстрирует эту возможность, добавляя ее в описание формы и вызывая функцию контроллера, которую мы напишем буквально через минуту:

```
<form id="addReview" name="addReview" role="form" ng-submit="vm.onSubmit()"
  class="form-horizontal">
```

Далее нужно создать соответствующую функцию `onSubmit` в контроллере модального окна отзыва. Для проверки ее работы мы просто будем выводить в консоль данные формы, а затем выполнять функцию `return false`, чтобы предотвратить отправку формы. При создании представления для формы мы использовали свойство модели представления для каждого ввода. Ну на самом деле у нас получилось чуть лучше — мы сделали каждый элемент дочерним свойством `vm.formData`, что сильно облегчает получение всех данных одновременно. Следующий фрагмент кода показывает основу для функции `onSubmit`, которую нужно добавить в контроллер модального окна отзыва:

```
vm.onSubmit = function () {
  console.log(vm.formData);
  return false;
};
```

Выводим все данные формы в консоль для проверки работы функции

Возвращаем false, чтобы избежать отправки формы и перезагрузки страницы

Теперь, когда мы можем перехватить данные формы, пришло время добавить кое-какие проверки.

Проверяем отправляемые данные формы

Прежде чем отправлять вслепую все подтвержденные для отправки формы в API для сохранения в базе данных, хотелось бы выполнить хотя бы краткую проверку и убедиться, что все поля заполнены. Если какие-то поля не заполнены, мы будем отображать сообщение об ошибке. Ваш браузер может запрещать отправку форм с пустыми полями, в таком случае временно уберите атрибут `required` из полей формы для тестирования проверок Angular.

При подтверждении отправки формы мы начнем с того, что удалим все существующие сообщения об ошибках, прежде чем проверить правильность каждого элемента данных в форме. Если любая из проверок вернет `false`, то есть покажет, что данные отсутствуют, мы зададим сообщение об ошибке формы в модели представления и вернем `false`. Если все данные присутствуют, продолжим вывод в консоль, как и прежде.

Листинг 10.23 демонстрирует требуемые изменения в функции `onSubmit` в контроллере модального окна отзыва для обработки этой части проверок.

Листинг 10.23. Добавляем простейшие проверки в обработчик `onSubmit`

```
vm.onSubmit = function () {
  vm.formError = "";
  if(!vm.formData.name || !vm.formData.rating
    || !vm.formData.reviewText) {
    vm.formError = "All fields required, please try again";
    return false;
  } else {
    console.log(vm.formData);
    return false;
  }
};
```

Заменяем все существующие сообщения об ошибках

В противном случае заносим отправляемые данные формы в консоль

Если какие-либо поля формы не заполнены, задаем сообщение об ошибке

После создания сообщения об ошибке хотелось бы в случае его генерации показывать его пользователям. Для этого добавим новый `div` для предупреждения Bootstrap в шаблон представления модального окна и привяжем к нему сообщение в виде контента. Мы хотим показывать `div` только при наличии сообщения об ошибке, которое нужно отображать, так что добавим директиву Angular `ng-show`. Эта директива принимает выражение в качестве значения, и если результат вычисления этого выражения равен `true`, она отобразит элемент, в противном случае — скроет его.

Мы можем использовать его для проверки того, есть ли значение у переменной `vm.formError`, и отображать содержимое `div` предупреждения только тогда, когда

оно имеется. Следующий фрагмент кода демонстрирует дополнения, которые необходимо внести в шаблон представления модального окна отзыва (предупреждение добавляется вверху тела модального окна):

```
<div class="modal-body">
  <div role="alert" ng-show="vm.formError" class="alert alert-danger">{{
    vm.formError }}</div>
</div class="form-group">
```

Все это в действии можно увидеть на рис. 10.12.

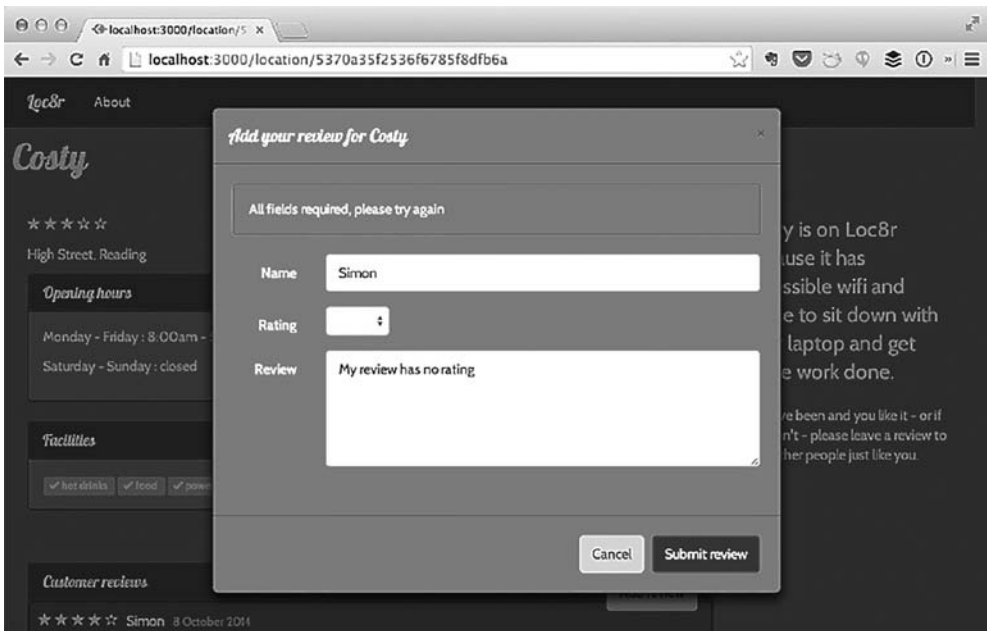


Рис. 10.12. При попытке отправки не до конца заполненной формы отображается сообщение об ошибке

Модификация сервиса данных для приема новых отзывов

Прежде чем использовать эту форму для отправки данных отзывов, необходимо добавить метод в сервис данных, который бы связывался с нужной конечной точкой API и мог отправить данные. Мы назовем новый метод `addReviewById`, он будет принимать на входе два параметра: ID местоположения и данные отзыва.

Содержимое метода ничем не будет отличаться от других, за исключением того, что для обращения к API мы используем `post` вместо `get`. В листинге 10.24 выделены полужирным шрифтом изменения, которые необходимо внести в функцию `loc8rData` в файле `loc8rData.service.js`.

Листинг 10.24. Добавляем в сервис данных новый метод `addReviewById`

```
function loc8rData ($http) {
  var locationByCoords = function (lat, lng) {
    return $http.get('/api/locations?lng=' + lng + '&lat=' + lat +
      '&maxDistance=20');
  };

  var locationById = function (locationid) {
    return $http.get('/api/locations/' + locationid);
  };

  var addReviewById = function (locationid, data) {
    return $http.post('/api/locations/' + locationid + '/reviews', data);
  };

  return {
    locationByCoords : locationByCoords,
    locationById : locationById,
    addReviewById : addReviewById
  };
}
```

Гениально! Теперь мы можем использовать этот сервис данных в нашем модальном окне.

Отправка данных формы сервису данных

Итак, мы добились того, чтобы данные формы отправлялись, а сервис данных был готов отправить их API. Свяжем это воедино. Будем использовать сервис данных так же, как делали раньше, — в этом новый метод ничем не отличается от предыдущих. Начнем с вызова метода, результатом которого будет промис `success` или `error`, поскольку сервис данных использует асинхронный метод `$http`.

Для того чтобы код оставался аккуратным, переместим эту функциональность в отдельную функцию под названием `doAddReview`. Все это показано в листинге 10.25, включая важный этап — внедрение сервиса `loc8rData` в контроллер.

Листинг 10.25. Отправляем заполненные данные формы сервису данных

```
reviewModalCtrl.$inject = ['$modalInstance', 'loc8rData', 'locationData'];
function reviewModalCtrl ($modalInstance, loc8rData, locationData) {
  var vm = this;
  vm.locationData = locationData;

  vm.onSubmit = function () {
    vm.formError = "";
    if (!vm.formData.name || !vm.formData.rating
      || !vm.formData.reviewText)
    {
      vm.formError = "All fields required, please try again";
    }
  };
}
```

Внедряем
сервис `loc8rData`
как зависимость

```

    return false;
  } else {
    vm.doAddReview(vm.locationData.locationid,
                  vm.formData);
  }
};
vm.doAddReview = function (locationid, formData) {
  loc8rData.addReviewById(locationid, {
    author : formData.name,
    rating : formData.rating,
    reviewText : formData.reviewText
  })
  .success(function (data) {
    console.log("Success!");
  })
  .error(function (data) {
    vm.formError = "Your review has not been saved, try again";
  });
  return false;
};

vm.modal = {
  cancel : function () {
    $modalInstance.dismiss('cancel');
  }
};
}

```

При успешной отправке формы передаем все подробности в новую функцию

Новая функция форматирует данные и отправляет их новому методу сервиса

Если сервис выполнен успешно, выводим сообщение в консоль

В противном случае используем formError для отображения предупреждающего сообщения в модальном окне

Теперь мы можем отправлять отзывы в базу данных. Остался последний штрих для красоты: после отправки отзыва нам хотелось бы закрыть модальное всплывающее окно и добавить отзыв в список.

Закрытие модального окна и отображение отзыва

Закрытие модального окна и добавление нового отзыва в список — тесно связанные задачи. Мы можем просто воспользоваться методом `dismiss`, как сделали для кнопок `Close` (Закреть) и `Cancel` (Отмена), но существует лучший способ.

У экземпляра модального окна наряду с методом `dismiss` имеется метод `close`. Метод `close` может фактически передавать какие-то данные обратно в родительский контроллер. Мы можем воспользоваться этим для передачи данных отзыва из контроллера модального окна в контроллер представления местоположения. При отправке нового отзыва API мы настроим его так, чтобы ответ для успешной отправки возвращал объект отзыва из базы данных.

Нам известно, что эти данные будут находиться в нужном для отображения на странице формате, так что это идеальный источник данных для отправки обратно родительскому контроллеру. Итак, необходимо вызвать метод `close` модального окна в обратном вызове `success` вызова функции `addReviewById`. Вместо того

чтобы вызывать ее непосредственно, создадим вспомогательный метод аналогично тому, как делали для кнопки Cancel (Отмена). Все вместе это показано в листинге 10.26.

Листинг 10.26. Передаем данные отзыва в метод close модального окна

```
vm.doAddReview = function (locationid, formData) {
  loc8rData.addReviewById(locationid, {
    author : formData.name,
    rating : formData.rating,
    reviewText : formData.reviewText
  })
  .success(function (data) {
    vm.modal.close(data);
  })
  .error(function (data) {
    vm.formError = "Your review has not been saved, please try again";
  });
  return false;
};

vm.modal = {
  close : function (result) {
    $modalInstance.close(result);
  },
  cancel : function () {
    $modalInstance.dismiss('cancel');
  }
};
```

После успешного добавления нового отзыва в базу данных передаем возвращенные данные во вспомогательный метод close модального окна

Создаем вспомогательный метод для вызова метода close экземпляра модального окна, передавая предоставленные данные

Возникает вопрос: как мы будем использовать эти данные? Хороший вопрос! Метод close возвращает промис в родительский контроллер, туда, где мы описали экземпляр модального окна. Мы можем привязаться к этому промису и после его разрешения просто вставить новый отзыв в массив отзывов, как показано в листинге 10.27.

Листинг 10.27. Разрешаем промис экземпляра модального окна для обновления списка отзывов

```
vm.popupReviewForm = function () {
  var modalInstance = $modal.open({
    templateUrl: '/reviewModal/reviewModal.view.html',
    controller: 'reviewModalCtrl as vm',
    resolve : {
      locationData : function () {
        return {
          locationid : vm.locationid,
          locationName : vm.data.location.name
        };
      }
    }
  });
};
```

```

    }
  });

  modalInstance.result.then(function (data) {
    vm.data.location.reviews.push(data);
  });
};

```

При разрешении промиса модального окна...

...вставляем возвращенные данные в массив отзывов; привязка Angular выполнит всю остальную работу

Так как массив отзывов привязан к шаблону представления, Angular автоматически обновит отображаемый список отзывов. И поскольку мы задали порядок отображения, при котором вначале стоят самые новые отзывы, то этот отзыв появится вверху списка (рис. 10.13). Все просто, не правда ли?

Вот и все. SPA Angular создано. Давайте резюмируем, чему мы научились.



Рис. 10.13. Добавляем отзыв в модальном окне. При подтверждении отправки модальное окно закрывается и отзыв появляется вверху списка без перезагрузки страницы

10.5. Резюме

В этой главе мы рассмотрели следующее.

- ❑ Перенос всего кода приложения на сторону клиента.
- ❑ Создание «красивых» URL с помощью API HTML5.
- ❑ Добавление в приложение нескольких представлений.
- ❑ Безопасную привязку текста, содержащего элементы HTML.
- ❑ Использование параметров URL.
- ❑ Добавление логики в стиле `if` с помощью директив `ng-switch` и `ng-show`.
- ❑ Использование предварительно собранных компонентов Angular.
- ❑ Отправку данных API с помощью сервиса `$http`.

В последней главе планируется рассмотреть управление аутентифицированными сеансами путем добавления возможности для пользователей регистрироваться и вводить свои данные, перед тем как оставить отзыв.

ЧАСТЬ IV

Управление
аутентификацией
и пользовательскими
сеансами

Умение идентифицировать отдельных пользователей — ключевой элемент функциональности большинства веб-приложений. Посетители должны иметь возможность регистрироваться, чтобы позднее смочь войти повторно. Приложение должно уметь использовать данные зарегистрированных и выполнивших вход пользователей.

В главе 11 мы рассмотрим функционирование аутентификации в стеке MEAN. Основной акцент будет сделан на управлении аутентификацией тогда, когда весь код приложения находится в браузере, как в случае SPA Angular. Аутентификация затрагивает все технологические слои приложения — мы рассмотрим сохранение пользовательских данных в базе данных, обеспечение безопасности конечных точек API и управление сеансами в браузере.

К концу части IV данной книги мы добавим полностью работающие регистрацию пользователей и систему входа в наше приложение Loc8r, которое будет задействовать данные текущего пользователя во время сеанса.

Глава 11

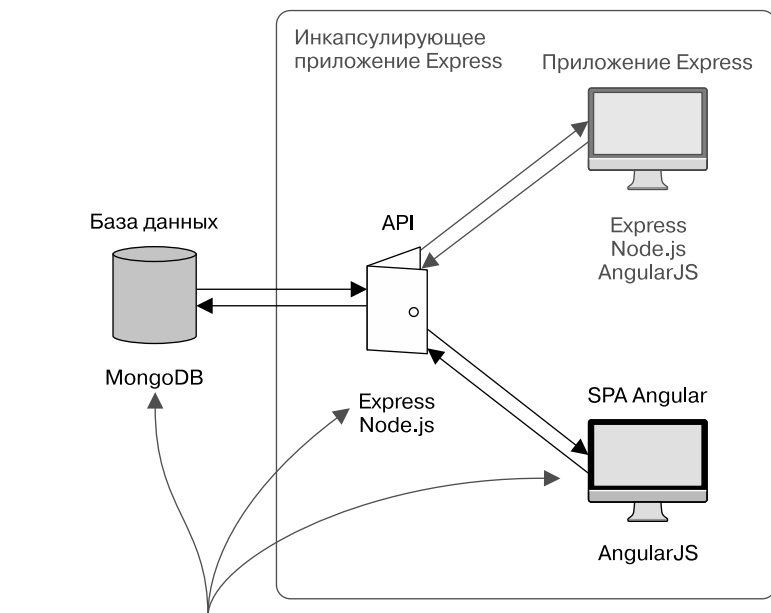
Аутентификация пользователей, управление сеансами и обеспечение безопасности API

В этой главе:

- ❑ добавление аутентификации в стеке MEAN;
- ❑ использование Passport для управления аутентификацией в Express;
- ❑ генерация веб-маркеров JSON в Express;
- ❑ регистрация и вход пользователей;
- ❑ обеспечение защищенности конечных точек в Express;
- ❑ использование локального хранилища и Angular для управления сеансами пользователей.

В этой главе мы собираемся усовершенствовать имеющееся приложение, заставив пользователей выполнять вход в него, прежде чем дать им возможность оставить отзыв. Это важная тема, поскольку многим веб-приложениям необходимо предоставлять пользователям возможность войти в приложение и управлять их сеансами.

Рисунок 11.1 демонстрирует наше местоположение в общем плане: мы теперь имеем дело с базой данных MongoDB, API Express и одностраничным приложением Angular.



Работает с базой данных MongoDB, API Express и Node.js, а также SPA Angular для предоставления приложению возможности аутентификации

Рис. 11.1. Эта глава добавляет в приложение систему аутентификации, затрагивающую большинство элементов архитектуры, таких как база данных, API и SPA клиентской части

Сначала мы сделаем обзор подхода к аутентификации в приложениях стека MEAN, а затем начнем модифицировать `loc8r` по одному элементу за раз, проходя по архитектуре от прикладной до клиентской части. Так, мы сначала модифицируем базу данных и схемы данных, затем поменяем API и, наконец, модифицируем клиентскую часть. К концу данной главы мы сможем регистрировать новых пользователей, обрабатывать их вход в приложение, поддерживать сеанс и выполнять действия, которые должны быть доступны только аутентифицированным пользователям.

11.1. Подход к аутентификации в стеке MEAN

Управление аутентификацией в приложении MEAN — одна из главных загадок стека, особенно при использовании SPA. Основная причина в том, что весь код приложения выдается браузеру, так что непонятно, как можно скрыть его часть. Как задать, кто что может видеть или делать?

11.1.1. Традиционный серверный подход

Основная путаница возникает из-за привычности для людей традиционного подхода к аутентификации и управлению пользовательскими сеансами.

При традиционных настройках код приложения находится и выполняется на сервере. Для входа пользователь вводит свои логин и пароль в форму, которая затем отправляется на сервер. После этого сервер обращается к базе данных для проверки учетных данных. Если учетные данные правильные, сервер устанавливает флаг или параметр сеанса в пользовательском сеансе на сервере, объявляя, что пользователь выполнил вход.

Сервер может устанавливать или не устанавливать в пользовательский браузер cookie-файл с информацией о сеансе. Это распространенная практика, но технически для управления аутентифицированными сеансами это не обязательно — жизненно важную информацию о сеансе обслуживает сервер. Этот поток выполнения показан на рис. 11.2.

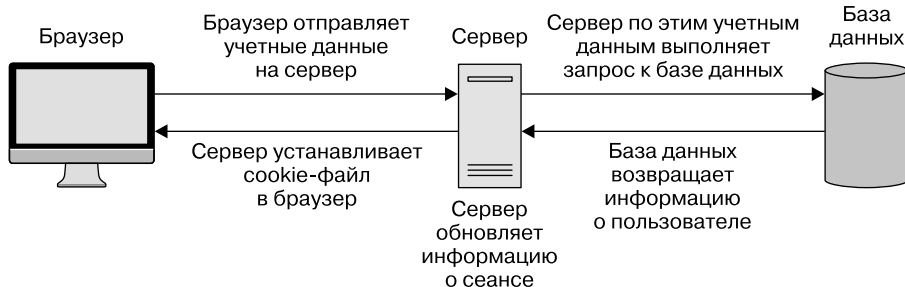


Рис. 11.2. В традиционном серверном приложении сервер и база данных проверяют учетные данные пользователя и добавляют их в пользовательские сеансы на сервере

Далее при обращении пользователя к защищенному ресурсу или попытке отправить какие-либо данные в базу данных сервер проверяет на основании его сеанса, может ли пользователь продолжить свои действия. Эти два потока выполнения показаны на рис. 11.3 и 11.4.

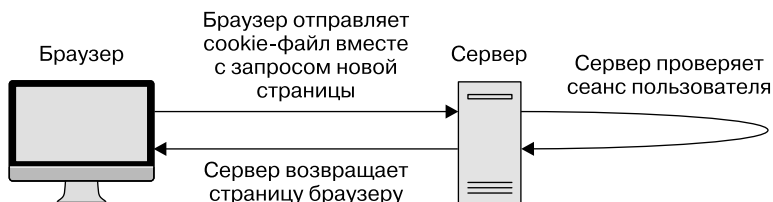


Рис. 11.3. В традиционном серверном приложении сервер проверяет сеансы пользователей, прежде чем продолжить обработку запроса к защищенному ресурсу

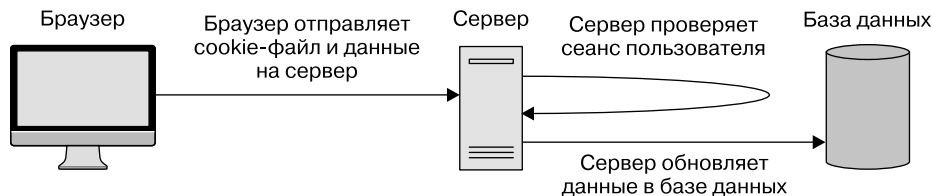


Рис. 11.4. В традиционном серверном приложении сервер проверяет сеансы пользователей, прежде чем отправить данные в базу данных

Использование традиционного подхода в стеке MEAN

Этот традиционный подход не совсем подходит для стека MEAN. Он основывается на резервировании сервером определенных ресурсов для каждого пользователя, чтобы можно было хранить сеансовую информацию. Как вы можете помнить из изложенного в главе 1, Node и Express не поддерживают сеансы для каждого пользователя — все приложение для всех пользователей выполняется в одном потоке.

Несмотря на это один из вариантов такого подхода можно применять в стеке MEAN, если вы используете серверное приложение, подобное тому, которое мы создавали в главе 7. Вместо применения серверных ресурсов для хранения информации о сеансе Express может задействовать базу данных. Можно использовать MongoDB, еще один популярный вариант — Redis — хранилище типа «ключ/значение» с исключительно высоким быстродействием.

Мы не станем рассматривать здесь этот подход, а взглянем на более сложный сценарий добавления аутентификации в SPA с обращением за данными к API.

11.1.2. Подход с использованием всего стека MEAN

Аутентификация в стеке MEAN ставит две проблемы.

- ❑ В API отсутствует сохранение состояния, так как в Express и Node нет понятия пользовательских сеансов.
- ❑ Логика приложения уже передана браузеру, так что у вас нет возможности ограничить передаваемый туда код.

Логичное решение этих проблем — поддерживать какую-либо разновидность состояния сеанса в браузере, оставляя браузеру решение, что он может отображать текущему пользователю, а что — не может. Это единственное существенное изме-

нение в подходе. Есть еще несколько технических различий, но это единственное серьезное изменение.

Отличный способ безопасного хранения пользовательских данных в браузере для целей поддержки сеансов — использовать JSON Web Token (веб-маркер JSON, JWT). Мы рассмотрим его подробнее дальше в данной главе, когда приступим к работе с ним. По сути JWT — объект JSON, зашифрованный в виде строки, бессмысленной для человеческого глаза, но поддающейся расшифровке и понятной как приложению, так и серверу.

Посмотрим, как это выглядит в высокоуровневом представлении, начав с процесса входа.

Управление процессом входа

Поток выполнения процесса входа иллюстрирует рис. 11.5. Пользователи отправляют свои учетные данные на сервер (через API), сервер проверяет их с помощью базы данных и возвращает идентификационный маркер браузеру. Браузер сохраняет этот маркер для дальнейшего повторного использования.

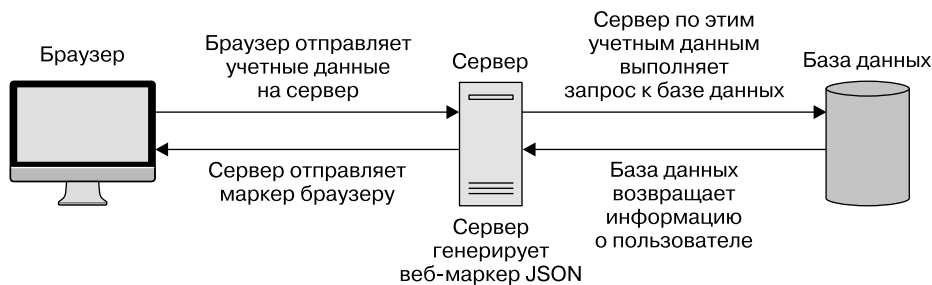


Рис. 11.5. Поток выполнения в приложении MEAN: после проверки сервером пользовательских учетных данных браузеру возвращается веб-маркер JSON

Это очень напоминает традиционный подход, но вместо хранения данных всех пользовательских сеансов на сервере они хранятся в браузере.

Меняем представления во время аутентифицированного сеанса

Во время действия сеанса пользователя последний должен иметь возможность сменить страницу или представление, а приложению будет необходима информация о том, что именно ему разрешено видеть. Так что тут приложение будет расшифровывать JWT и применять информацию из него для отображения пользователям соответствующих данных (рис. 11.6).

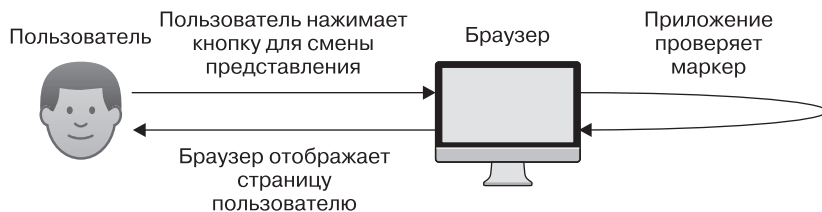


Рис. 11.6. Применяя находящиеся в JWT данные, SPA может определить, какие ресурсы пользователи могут использовать или видеть

В этом месте наиболее ярко проявляется отступление от традиционного подхода. Сервер не имеет представления, что пользователи что-либо делают, до тех пор, пока им не понадобится доступ к API и базе данных.

Безопасное обращение к API

Если доступ к каким-то частям приложения ограничен аутентифицированными пользователями, то вполне вероятно, что найдутся какие-либо действия с базой данных, которые разрешено выполнять только аутентифицированным пользователям. Поскольку API не сохраняет состояние, он не знает, кто выполняет какой вызов, если ему об этом не сообщить. Именно тут вступает в игру JWT. Как показывает рис. 11.7, маркер отправляется в конечную точку API, которая расшифровывает маркер перед проверкой того, разрешено ли пользователю выполнять данное обращение.

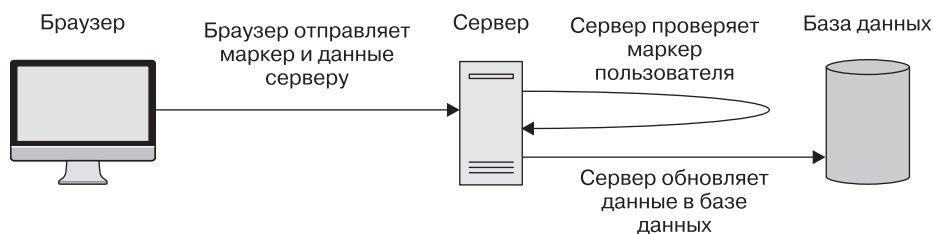


Рис. 11.7. При обращении к защищенной конечной точке API браузер отправляет вместе с данными JWT, сервер расшифровывает маркер для проверки допустимости запроса пользователя

Отлично, мы рассмотрели данный подход с высоты птичьего полета и неплохо понимаем, что хотим получить. Теперь сделаем первый шаг к встраиванию его в приложение Loc8r, настроив MongoDB для хранения информации о пользователе.

11.2. Создание схемы для пользователей в MongoDB

Имена пользователей и пароли, естественно, должны храниться в базе данных. Чтобы реализовать это в стеке MEAN, нам понадобится создать схему Mongoose. Пароли не должны никогда — ни при каких обстоятельствах! — храниться в базе данных в открытом виде, так как при утечке данных из базы это стало бы причиной колоссальной брешы в системе безопасности. Так что нам придется что-то с этим сделать на этапе генерации схемы.

11.2.1. Одностороннее шифрование паролей: хеши и соль

Нам нужно выполнить одностороннее шифрование пароля. Одностороннее шифрование делает невозможной расшифровку пароля, сохраняя возможность с легкостью проверить его правильность. Когда пользователь предпринимает попытку входа, приложение может зашифровать предоставленный пароль и проверить, соответствует ли он сохраненному значению.

Однако простого шифрования недостаточно. Если несколько пользователей выбрали себе пароль «пароль» (да, такое случается!), то зашифрованный вариант для всех них будет одним и тем же. Любой хакер, получивший доступ к базе данных, сможет увидеть закономерность и вычислить потенциально слабые пароли.

Здесь нам пригодится понятие *соли*. Соль — это случайная строка, генерируемая приложением для каждого пользователя и объединяемая с паролем перед шифрованием. Получающееся в итоге зашифрованное значение называется *хешем* (рис. 11.8).

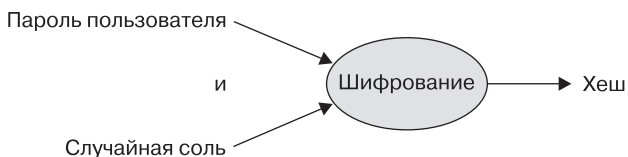


Рис. 11.8. Хеш создается объединением пользовательского пароля со случайной солью и последующим шифрованием

Обычно как соль, так и хеш хранятся в базе данных отдельно, а не в виде общего поля «пароль». При таком подходе все хеши будут уникальны, а пароли — хорошо защищены.

11.2.2. Создание схемы Mongoose

Начнем с создания файла, в котором будет храниться схема, и запросим его в приложении. Создадим новый файл `users.js` в каталоге `app_api/models/`.

Далее «подтянем» его в приложение, сославшись на него в файле `db.js` из того же каталога. Его необходимо запросить после уже существующей строки, вносящей в приложение модель для местоположений, как показано в следующем фрагменте кода:

```
// Внесите ваши схемы и модели
require('./locations');
require('./users');
```

Отлично, теперь мы готовы создать базовую схему.

Базовая схема для пользователей

Что нам необходимо в схеме для пользователей? Мы знаем, что нам необходимы отображаемое имя для показа в отзывах плюс хеш и соль для пароля. Добавим также адрес электронной почты, который будет уникальным идентификатором для входа пользователей.

В новом файле `user.js` запросим Mongoose и опишем новую `userSchema`, как показано в листинге 11.1.

Листинг 11.1. Базовая схема Mongoose для пользователей

```
var mongoose = require( 'mongoose' );

var userSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true
  },
  name: {
    type: String,
    required: true
  },
  hash: String,
  salt: String
});
```

Электронная почта должна быть обязательным и уникальным полем

Имя также должно быть обязательным, но, возможно, неуникальным полем

Хеш и соль — просто строки

Электронная почта и имя будут задаваться в регистрационной форме, но хеш и соль создаются системой. Хеш, конечно, будет вычисляться на основе соли и задаваемого через форму пароля.

Теперь рассмотрим, как можно задать соль и хеш с помощью методов — элемента функциональности Mongoose, который мы пока не трогали.

11.2.3. Задание зашифрованных путей с помощью методов Mongoose

Mongoose позволяет добавить в схему методы, которые становятся доступными в качестве методов модели. Подобные методы дают коду прямой доступ к атрибутам модели.

Идеальным результатом была бы возможность сделать что-то соответствующее следующим строкам псевдокода:

```
var User = mongoose.model('User');
var user = new User();
user.name = "User's name";
user.email = "test@example.com";
user.setPassword("myPassword");
user.save();
```

Создаем экземпляр модели

Создаем нового пользователя

Задаем значения для имени и адреса электронной почты

Вызываем метод setPassword для установки пароля

Сохраняем нового пользователя

Посмотрим, как добавить метод в Mongoose, чтобы добиться этого.

Добавление метода в схему Mongoose

Методы можно добавить в схему *после того*, как она описана, и *до того*, как модель скомпилирована. В коде приложения методы делаются в расчете на использование сразу после создания экземпляра модели.

Добавление метода в схему не представляет собой ничего сложного и выполняется путем подключения цепочкой к методу `.methods` схемы. Передача параметров также несложна. Например, смотрите следующий фрагмент кода, который будет эскизом для реального метода `setPassword`:

```
userSchema.methods.setPassword = function(password){
  this.salt = SALT_VALUE;
  this.hash = HASH_VALUE;
};
```

Что необычно для фрагмента кода JavaScript, применяя `this` в методах Mongoose фактически ссылается на саму модель. Так что в предыдущем примере переменные `this.salt` и `this.hash` в методе фактически задаются для модели.

Но прежде, чем мы сможем что-либо сохранить, необходимо сгенерировать случайное значение соли и зашифровать хеш. К счастью, существует предназначенный для этой цели модуль Node `crypto`.

Модуль `crypto` для шифрования

Шифрование — настолько распространенная потребность, что существует встроенный в Node модуль под названием `crypto`. Он включает несколько методов для управления шифрованием данных, мы рассмотрим следующие два:

- `randomBytes` — метод генерации криптографически стойкой строки данных для использования в качестве соли;
- `pbkdf2Sync` — метод создания из пароля и соли хеша. `pbkdf2` расшифровывается как *основанная на пароле функция формирования ключа 2* (password-based key derivation function 2), это промышленный стандарт.

Воспользуемся этими методами, чтобы создать случайную строку для соли, а потом для шифрования пароля и соли в хеш. Во-первых, выполним запрос модуля `crypto` вверху файла `users.js`:

```
var mongoose = require( 'mongoose' );
var crypto = require('crypto');
```

Во-вторых, модифицируем метод `setPassword` для задания соли и хеша для пользователей. Для задания соли воспользуемся методом `randomBytes`, чтобы сгенерировать случайную 16-байтную строку. Затем применим метод `pbkdf2Sync` для создания зашифрованного хеша из пароля и соли. Вот тут показано все вместе:

```
userSchema.methods.setPassword = function(password){
  this.salt = crypto.randomBytes(16).toString('hex');
  this.hash = crypto.pbkdf2Sync(password, this.salt,
    ↪ 1000, 64).toString('hex');
};
```

Создаем случайную строку для соли

Создаем зашифрованный хеш

Теперь при вызове метода `setPassword` с паролем в качестве параметра соль и хеш будут генерироваться для пользователей и вноситься в экземпляр модели. Пароль не будет сохраняться никогда и нигде, даже в оперативной памяти.

11.2.4. Проверяем введенный пароль

Еще один аспект хранения пароля — возможность его извлечения при попытке пользователей выполнить вход, ведь мы должны иметь возможность проверить их учетные данные. Пароль зашифрован таким образом, что мы не можем его дешифровать, так что нам нужно использовать то же шифрование для пароля, с которым пытается выполнить вход пользователь, чтобы сравнить с хранимым значением.

Выполнить хеширование и проверку можно в простом методе Mongoose. Добавьте в файл `users.js` следующий метод. Он будет вызываться из контрол-

лера при обнаружении пользователя с заданным адресом электронной почты и возвращать `true` или `false` в зависимости от совпадения или несовпадения хеша:

```
userSchema.methods.validPassword = function(password) {
  var hash = crypto.pbkdf2Sync(password,
    this.salt, 1000, 64).toString('hex');
  return this.hash === hash;
};
```

Вот и все. Довольно просто, не правда ли? Мы увидим эти методы в действии после генерации контроллеров API.

Последнее, что нужно сделать для контроллера, — сгенерировать веб-маркер JSON, который будет содержать кое-какие данные модели.

11.2.5. Генерация веб-маркера JSON

JWT (произносится «джот») применяется для передачи данных, в нашем случае между API на сервере и SPA в браузере. JWT может также использоваться сервером, генерирующим маркер для аутентификации пользователя, для его возвращения в последующем запросе.

Давайте вкратце рассмотрим основные части JWT.

Три составные части JWT

JWT состоит из трех выглядящих случайными разделенных точками частей. Они могут быть довольно длинными, вот реальный пример:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfawQiOiIiNTZiZWVmOTUzOTViMTlhNjc1ODgiLCJlbWVpbCI6InNpbW9uQGZ1bGxzZGFja3RyYWluaW5nLmNvbSIsIm5hbWUiOiJJTaw1b2xtZXMiLCJleHAiOjE0MzUwNDA0MTgsIm1hdCI6MTQzNDQzNTYxOH0.GD7UrfnLk295rwwIrCikbkAKctFFoRCHotLYZwZpd1E
```

Для человеческого взгляда он довольно бессмыслен, но вы могли заметить две точки, а значит, три отдельные части. Вот эти части.

- *Заголовок* (Header) — закодированный объект JSON, содержащий тип и алгоритм хеширования.
- *Содержимое* (Payload) — закодированный объект JSON, содержащий данные — собственно тело маркера.
- *Подпись* (Signature) — зашифрованный хеш заголовка и содержимого, использующий секрет, известный только формирующему его серверу.

Обратите внимание на то, что первые две части не зашифрованы, они *закодированы*. Это значит, что декодировать их браузеру — и, конечно, другим

приложениям — несложно. В наиболее современных браузерах имеется функция `atob()` для декодирования строки Base64. Родственная ей функция `btoa()` выполняет кодирование в формат строки Base64.

Третья часть, подпись, зашифрована. Для расшифровки ее вам необходимо воспользоваться секретом, задаваемым на сервере. Этот секрет следует хранить на сервере и никогда не обнародовать.

Хорошая новость заключается в том, что существуют библиотеки для всех сложных частей этого процесса. Так что установим одну из этих библиотек в наше приложение и создадим метод схемы для генерации JWT.

Генерация JWT из Express

Первый шаг к генерации JWT — включение из командной строки модуля `prn` под названием `jsonwebtoken`:

```
$ npm install jsonwebtoken --save
```

Далее нам необходимо выполнить его запрос вверху файла `users.js`:

```
var mongoose = require( 'mongoose' );
var crypto = require('crypto');
var jwt = require('jsonwebtoken');
```

Наконец, необходимо создать метод схемы, который мы назовем `generateJwt`. Для генерации JWT нужно предоставить содержимое, то есть данные, и значение секрета. В содержимом мы будем отправлять переменные `_id`, `email` и `name` с данными пользователя. Также зададим для маркера дату истечения срока действия, после которой пользователю вновь придется выполнить вход для генерации нового JWT. Мы воспользуемся зарезервированным для этого в JWT полем `exp`, в котором дата истечения срока действия должна указываться в формате числового значения Unix.

Для генерации JWT достаточно вызвать метод `sign` библиотеки `jsonwebtoken`, передав ему содержимое в виде объекта JSON и секрет в виде строки. Он вернет маркер, который мы затем сможем вернуть из метода. Листинг 11.2 демонстрирует все это.

Листинг 11.2. Создание метода схемы для генерации JWT

```
userSchema.methods.generateJwt = function() {
  var expiry = new Date();
  expiry.setDate(expiry.getDate() + 7);

  return jwt.sign({
    _id: this._id,
    email: this.email,
    name: this.name,
```

Создаем объект для даты истечения срока действия и задаем ее равной текущей дате плюс семь дней

Вызываем метод `jwt.sign` и задаем ее равной текущей дате плюс семь дней

Передает содержимое методу

и возвращаем то, что возвращает он

```

    exp: parseInt(expiry.getTime() / 1000),
  }, 'thisIsSecret' );
};

```

Включаем exp в виде времени Unix в секундах

Отправляем секрет для использования алгоритмом хеширования

При вызове метод `generateJwt` будет использовать данные из модели текущего пользователя для создания уникального JWT и его возвращения.

С этим кодом имеется только одна проблема: секрет не должен быть виден в коде, так что разберемся с этим прямо сейчас.

Держим секрет в секрете с помощью переменных среды

Если вы собираетесь поместить свой код в систему контроля версий, например GitHub, то вряд ли захотите публиковать секрет. Выставление секрета на всеобщее обозрение чрезвычайно ослабит вашу модель безопасности — узнав ваш секрет, любой желающий сможет генерировать фальшивые маркеры, которые приложение будет считать настоящими. Чтобы сохранить секреты в секрете, зачастую стоит задавать их в виде переменных среды.

Вот удобный способ выполнить это, позволяющий вам держать в поле зрения переменные среды в коде на своей машине. Сначала создайте файл в корневом каталоге проекта с названием `.env` и задайте секрет следующим образом:

```
JWT_SECRET=thisIsSecret
```

В данном случае секрет равен `thisIsSecret`, но он может быть таким, как вы пожелаете, поскольку представляет собой строку. Теперь нужно сделать так, чтобы этот файл не включался ни в какие коммиты Git, посредством добавления строки в файл `.gitignore` в проекте. Минимальное содержимое файла `.gitignore` должно быть следующим:

```
# Dependency directory
node_modules

# Environment variables
.env
```

Чтобы читать этот новый файл и использовать его для задания переменных среды, необходимо установить и использовать еще один модуль `npm` — `dotenv`. Сделаем это с помощью следующей команды терминала:

```
$ npm install dotenv --save
```

Необходимо выполнить запрос модуля `dotenv` в файле `app.js` в самой первой строке, как показано тут:

```
require('dotenv').load();
var express = require('express');
```

Теперь осталось только изменить схему для пользователей, чтобы заменить жестко зашитый секрет переменной среды (изменения выделены полужирным шрифтом в следующем фрагменте кода):

```
userSchema.methods.generateJwt = function() {
  var expiry = new Date();
  expiry.setDate(expiry.getDate() + 7);

  return jwt.sign({
    _id: this._id,
    email: this.email,
    name: this.name,
    exp: parseInt(expiry.getTime() / 1000),
  }, process.env.JWT_SECRET);
};
```

← Не держите секреты в коде, используйте вместо этого переменные среды

Конечно, ваша среда для промышленной эксплуатации тоже должна знать об этой переменной среды. Вы, вероятно, помните команду, которую мы использовали для указания URI базы данных на Heroku. Здесь все аналогично, так что выполните в терминале следующую команду:

```
$ heroku config:set JWT_SECRET=thisIsSecret
```

Вот и все. Мы рассмотрели всю относящуюся к MongoDB и Mongoose сторону, теперь взглянем на использование Passport для управления аутентификацией.

11.3. Создание API аутентификации с помощью Passport

Passport — написанный Джаредом Хенсоном модуль Node, разработанный для упрощения аутентификации в Node. Одна из самых сильных его сторон — умение согласовывать несколько методов аутентификации, называемых *стратегиями*. Примеры таких стратегий включают:

- Facebook;
- Twitter;
- OAuth;
- локальные имя пользователя и пароль.

Вы можете найти множество других стратегий, выполнив поиск по модулю passport на сайте npm. С помощью Passport вы сможете легко использовать один или несколько подобных подходов, чтобы предоставить пользователям возмож-

ность выполнить вход в ваше приложение. Для LocalStrategy будем использовать *локальную* (local) стратегию, так как мы храним имена пользователей и хеши паролей в базе данных.

Начнем с установки модулей.

11.3.1. Установка и конфигурация Passport

Passport разделяется на основной модуль и отдельные модули для каждой из стратегий. Поэтому установим основной модуль и модуль локальной стратегии посредством npm, выполнив в терминале следующие команды:

```
$ npm install passport --save
$ npm install passport-local --save
```

После того как оба они будут установлены, мы сможем создать конфигурацию для нашей локальной стратегии.

Создание конфигурационного файла Passport

Passport в нашем приложении будет использовать API, так что создадим конфигурационный файл в каталоге app_api. В app_api создайте подкаталог config, внутри которого создайте файл passport.js.

Вверху этого файла необходимо запросить Passport и модуль локальной стратегии, а также Mongoose и пользовательскую модель. Это показано в следующем фрагменте кода:

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var mongoose = require('mongoose');
var User = mongoose.model('User');
```

Теперь мы можем выполнить конфигурирование локальной стратегии.

Конфигурируем локальную стратегию

Для задания стратегии Passport необходимо использовать метод passport.use и передать ему конструктор новой стратегии. Этот конструктор принимает параметры в качестве первого аргумента и выполняющую почти всю работу функцию — в качестве второго. Каркас этого выглядит следующим образом:

```
passport.use(new LocalStrategy({
  function(username, password, done) {
  }
}));
```

По умолчанию локальная стратегия Passport ожидает и использует поля `username` и `password`. У нас есть `password`, так что с ним все в порядке, но вместо `username` мы будем использовать `email`. Passport предоставляет возможность переопределять поле для имени пользователя в объекте параметров, как показано в следующем фрагменте кода:

```
passport.use(new LocalStrategy(
  usernameField: 'email'
),
function(username, password, done) {
}
));
```

Далее следует основная функция, представляющая собой, по сути, просто вызов `Mongoose` для поиска пользователей с переданными функции именем пользователя и паролем. Нашей функции `Mongoose` нужно будет сделать следующее:

- ❑ найти пользователя с заданным адресом электронной почты;
- ❑ проверить правильность пароля;
- ❑ вернуть объект пользователя, если пользователь найден и пароль правилен;
- ❑ в противном случае вернуть сообщение, указывающее, что именно не так.

Так как адрес электронной почты должен быть уникальным для схемы, можно использовать метод `Mongoose findOne`. Интересно отметить, что мы будем использовать созданный ранее метод схемы `validPassword` для проверки правильности указанного пароля.

Листинг 11.3 демонстрирует локальную стратегию целиком.

Листинг 11.3. Полное описание локальной стратегии Passport

```
passport.use(new LocalStrategy({
  usernameField: 'email'
}),
function(username, password, done) {
  User.findOne({ email: username }, function (err, user) {
    if (err) { return done(err); }
    if (!user) {
      return done(null, false, {
        message: 'Incorrect username.'
      });
    }
    if (!user.validPassword(password)) {

```

Ищем в MongoDB пользователя с заданным адресом электронной почты

Если пользователь не найден, возвращаем false и сообщение

Вызываем метод `validPassword`, передавая ему полученный пароль

```

        return done(null, false, {
            message: 'Incorrect password.'
        });
    }
    return done(null, user);
});
}
));

```

Если пароль неправильный, возвращаем false и сообщение

Если мы добрались до конца, то можем вернуть объект пользователя

Теперь, когда мы установили Passport и настроили стратегию, необходимо зарегистрировать ее в приложении.

Добавляем Passport и конфигурацию в приложение

Чтобы добавить наши настройки Passport в приложение, необходимо в `app.js`:

- ❑ выполнить запрос Passport;
- ❑ выполнить запрос конфигурации стратегии;
- ❑ инициализировать Passport.

Ничего сложного в этом нет, но важно то, *где* именно в файле `app.js` они будут находиться.

Passport необходимо запросить до моделей базы данных, а конфигурацию — после них. Причем и то и другое должно быть выполнено до описаний маршрутов. Слегка реорганизовав верхнюю часть файла `app.js`, мы можем вставить Passport и конфигурацию следующим образом:

```

require('dotenv').load();
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var uglifyJs = require("uglify-js");
var fs = require('fs');
var passport = require('passport');

require('./app_api/models/db');
require('./app_api/config/passport');

var routes = require('./app_server/routes/index');
var routesApi = require('./app_api/routes/index');

```

Выполняем запрос Passport до описания модели

Выполняем запрос стратегии после описания модели

Стратегию необходимо описать после описаний моделей, поскольку для нее нужно, чтобы модель пользователя уже существовала.

Passport следует инициализировать в файле `app.js` после описания статистических маршрутов, но до маршрутов, которые будут использовать аутентификацию, — в нашем случае маршрутов API. Следующий фрагмент кода демонстрирует это:

```
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'app_client')));

app.use(passport.initialize());

app.use('/api', routesApi);
```

Теперь Passport установлен, сконфигурирован и инициализирован в нашем приложении. А мы сейчас создадим конечные точки API, необходимые для того, чтобы пользователи могли регистрироваться и выполнять вход в приложение.

11.3.2. Создание конечных точек API для возврата веб-маркеров JSON

Чтобы предоставить пользователям возможность входа и регистрации через наш API, понадобятся две новые конечные точки. Для этого необходимо добавить два новых описания маршрутов и два новых соответствующих контроллера. Когда мы создадим конечные точки, мы сможем протестировать их с помощью Postman, а также убедиться в том, что конечная точка регистрации работает, воспользовавшись командной оболочкой Mongo, чтобы заглянуть внутрь базы данных. Прежде всего добавим маршруты.

Добавляем описания маршрутов для аутентификации

Описания маршрутов для API находятся в файле `index.js` в каталоге `app_api/routes`, так что именно оттуда мы и начнем. Наши контроллеры разделены по логическим наборам — в настоящее время это `locations` и `reviews`. Имеет смысл добавить третий набор для аутентификации. Следующий фрагмент кода показывает его добавление вверху файла:

```
var ctrlLocations = require('../controllers/locations');
var ctrlReviews = require('../controllers/reviews');
var ctrlAuth = require('../controllers/authentication');
```

Мы пока что не создали файл `controllers/authentication`, сделаем это позже, когда будем писать код соответствующих контроллеров. Далее добавим сами описания маршрутов — ближе к концу файла, но до строки `module.exports`. Их нам нужно

два, по одному для регистрации и входа, и сделаем мы их в папках `/api/register` и `/api/login` соответственно, как показано в следующем фрагменте кода:

```
router.post('/register', ctrlAuth.register);
router.post('/login', ctrlAuth.login);
```

Конечно, оба они должны быть действиями `post`, так как принимают данные. Напомню, что нам не нужно указывать часть маршрутов `/api`, поскольку они добавляются при запросе маршрутов в файле `app.js`. Теперь, прежде чем приступить к тестированию, нам нужно добавить контроллеры.

Создаем контроллер регистрации

Сначала рассмотрим контроллер регистрации, но для этого нам понадобится создать указанный в описании маршрута файл. Поэтому создайте в каталоге `app_api/controllers` файл `authentication.js` и введите приведенный в следующем фрагменте код для запроса всего, что нам понадобится, а также снова внедрите функцию `sendJSONresponse`:

```
var passport = require('passport');
var mongoose = require('mongoose');
var User = mongoose.model('User');

var sendJSONresponse = function(res, status, content) {
  res.status(status);
  res.json(content);
};
```

Процесс регистрации, по сути, вообще не использует Passport. Мы можем сделать все, что нам нужно, с помощью Mongoose, ведь мы уже настроили в схеме различные вспомогательные методы.

Контроллер регистрации должен будет выполнять следующее.

1. Проверять, все ли обязательные поля присутствуют.
2. Создавать новый экземпляр модели `User`.
3. Задавать имя и адрес электронной почты пользователя.
4. Использовать метод `setPassword` для создания и добавления соли и хеша.
5. Сохранять пользователя.
6. Возвращать JWT после сохранения.

Объем работы кажется значительным, но, к счастью, все довольно просто — мы уже сделали самую сложную часть работы, создав методы Mongoose. Теперь нам просто надо связать все воедино. В листинге 11.4 приведен весь код контроллера регистрации.

Листинг 11.4. Контроллер регистрации для API

```

module.exports.register = function(req, res) {
  if(!req.body.name || !req.body.email
    || !req.body.password) {
    sendJSONresponse(res, 400, {
      "message": "All fields required"
    });
    return;
  }

  var user = new User();

  user.name = req.body.name;
  user.email = req.body.email;

  user.setPassword(req.body.password);

  user.save(function(err) {
    var token;
    if (err) {
      sendJSONresponse(res, 404, err);
    } else {
      token = user.generateJwt();
      sendJSONresponse(res, 200, {
        "token" : token
      });
    }
  });
});
};

```

Возвращаем статус ошибки, если не все обязательные поля присутствуют

Создаем новый экземпляр пользователя и задаем имя и адрес электронной почты

Используем метод setPassword для задания соли и хеша

Сохраняем нового пользователя в MongoDB

Генерируем JWT с помощью метода схемы и отправляем его браузеру

В этом фрагменте кода нет ничего особо нового или сложного, но он действительно демонстрирует возможности методов Mongoose. Если бы контроллер регистрации был написан обычным образом, а не с помощью Mongoose, он был бы весьма сложен. Но в нынешнем виде контроллер удобен для чтения и понимания — как раз то, чего вам хотелось бы от своего кода. Переходим к контроллеру входа.

Создаем контроллер входа

Контроллер входа будет использовать Passport для наиболее сложных задач. Мы начнем с простой проверки заполнения обязательных полей, а затем передадим все данные Passport. Passport выполнит свою задачу — попытается аутентифицировать пользователя на основе заданной нами стратегии, а затем сообщит, была аутентификация удачной или нет. Если да, мы можем снова воспользоваться методом схемы generateJwt для создания JWT перед отправкой его браузеру.

Все это, включая также требуемый для запуска метода passport.authenticate синтаксис, показано в листинге 11.5. Его необходимо добавить в новый файл authentication.js.

Листинг 11.5. Контроллер входа для API

```

module.exports.login = function(req, res) {
  if(!req.body.email || !req.body.password) {
    sendJSONresponse(res, 400, {
      "message": "All fields required"
    });
    return;
  }

  passport.authenticate('local',
    function(err, user, info){
      var token;

      if (err) {
        sendJSONresponse(res, 404, err);
        return;
      }

      if(user){
        token = user.generateJwt();
        sendJSONresponse(res, 200, {
          "token" : token
        });
      } else {
        sendJSONresponse(res, 401, info);
      }
    })(req, res);
};

```

Проверяем, что переданы все обязательные поля

Передаем имя стратегии и обратный вызов методу аутентификации

Возвращаем ошибку, если Passport возвращает ошибку

Если Passport вернул экземпляр пользователя, генерируем и отправляем JWT

В противном случае возвращаем информационное сообщение (почему аутентификация не удалась)

Обеспечиваем для Passport доступность req и res

Мы опять видим, что для контроллера входа вся сложная работа выносится наружу, на этот раз в основном благодаря Passport. Это делает код легким для чтения, сопровождения и понимания — достигается обязательная цель любого программирования.

Что ж, мы создали эти две конечные точки, теперь их протестируем.

Тестирование конечных точек и сверка с базой данных

При создании основной части API в главе 6 мы тестировали конечные точки с помощью Postman. Здесь поступим аналогично. Тестирование конечной точки регистрации и возврат ей JWT демонстрирует рис. 11.9. URL для тестирования — localhost:3000/api/register, в форме создаются поля для переменных name, email и password. Не забудьте выбрать тип формы x-www-form-urlencoded.

Тестирование конечной точки для входа, включая возвращение сообщения об ошибке Passport, а также JWT в случае успешного входа демонстрирует рис. 11.10. URL для этого теста — localhost:3000/api/login, обязательные поля формы — email и password.

Помимо наблюдения в браузере возвращения JWT, тогда, когда нужно, мы можем заглянуть в базу данных и проверить, был ли создан пользователь. Вернемся для этого в командную оболочку Mongo, которую мы уже довольно давно не использовали:

```
$ mongo
> use Loc8r
> db.users.find()
```

Или можно выполнить поиск конкретного пользователя, задав адрес электронной почты:

```
> db.users.find({email : "simon@fullstacktraining.com"})
```

В любом случае вы должны увидеть один или несколько возвращенных из базы данных документов пользователей, которые будут выглядеть примерно так:

```
{ "hash" :
"1255e9df3daa899bee8d53a42d4acf3ab8739fa758d533a84da5eb1278412f7a7bdb36e-
888aeb80a9eec4fb7bbe9bcef038f01fbbf4e6048e2f4494be44bc3d5", "salt" :
"40368d9155ea690cf9fc08b49f328e38", "email" : "simon@fullstacktraining.com",
"name" : "Simon Holmes", "_id" : ObjectId("558b95d85f0282b03a603603"), "__v"
: 0 }
```

Я выделил имена путей полужирным шрифтом, чтобы они были заметнее в печатном виде, но вы в любом случае можете видеть здесь все данные, которые ожидали.

Теперь, когда мы создали конечные точки для обеспечения пользователям возможности регистрации и входа, следующее, что мы рассмотрим, — как сделать определенные конечные точки доступными только для аутентифицированных пользователей.

11.4. Защита конечных точек API

В веб-приложениях весьма часто бывает нужно ограничить доступ к конечным точкам API, сделав его возможным только для аутентифицированных пользователей. В Loc8r, например, мы хотели бы сделать так, чтобы только аутентифицированные пользователи могли оставлять отзывы. Эта задача состоит из двух частей:

- ❑ предоставления возможности обращения к API создания нового отзыва только пользователям, отправившим в своем запросе корректный JWT;
- ❑ проверки внутри контроллера существования пользователя и того, что ему разрешено создавать отзывы.

Начнем с добавления аутентификации в маршруты Express, прежде чем перейти к контроллеру.

11.4.1. Добавление промежуточного ПО аутентификации в маршруты Express

В Express можно добавить промежуточное ПО в маршруты, как вы увидите буквально через минуту. Это промежуточное ПО находится между маршрутом и контроллером. Поэтому после обращения к маршруту промежуточное ПО запускается до контроллера и может предотвратить запуск контроллера или изменить отправляемые данные.

Мы хотели бы использовать промежуточное ПО для проверки предоставляемых JWT с последующим извлечением данных из содержимого и добавления их в объект `req` для использования контроллером. Ничего удивительного в том, что для этой цели существует модуль `npm`, называющийся `express-jwt`, так что давайте установим его путем выполнения в терминале следующей команды:

```
$ npm install express-jwt --save
```

Теперь можно использовать его в файле маршрутов.

Настройка промежуточного ПО

Для использования модуля `express-jwt` необходимо выполнить его метод `require`, а также настроить его должным образом. После включения модуль `express-jwt` предоставляет функцию, которой можно передать объект параметров. Мы будем использовать его для отправки секрета и задания имени свойства, которое мы хотели бы добавить в объект `req` для хранения содержимого.

По умолчанию в `req` добавляется свойство `user`, но в нашем случае `user` — это экземпляр модели `Mongoose User`. Поэтому во избежание путаницы и ради поддержания согласованности мы используем имя `payload` — в конце концов, именно так оно называется в `Passport` и внутри самого `JWT`.

Откройте файл маршрутов `API app_api/routes/index.js` и добавьте настройки вверху файла. В следующем фрагменте кода изменения выделены полужирным шрифтом:

```
var express = require('express');
var router = express.Router();
var jwt = require('express-jwt');
var auth = jwt({
  secret: process.env.JWT_SECRET,
  userProperty: 'payload'
});
```

Выполняем `require` модуля `express-jwt`

Задаем секрет, используя ту же переменную среды, что и раньше

Указываем имя `payload` свойства для `req`

Теперь, когда промежуточное ПО сконфигурировано, можно добавить аутентификацию в маршруты.

Добавляем промежуточное ПО для аутентификации в конкретные маршруты

Добавить промежуточное ПО в описания маршрутов очень просто. Нужно только сослаться на него в командах маршрутизатора между маршрутом и контроллером. Оно на самом деле оказывается промежуточным!

Следующий фрагмент кода показывает, как добавить это ПО в методы отзывать `post`, `put` и `delete`, причем `get` остается общедоступным — предполагается, что отзывать должны быть доступны для чтения всеми посетителями:

```
router.post('/locations/:locationid/reviews', auth,
  ➔ ctrlReviews.reviewsCreate);
router.get('/locations/:locationid/reviews/:reviewid',
  ➔ ctrlReviews.reviewsReadOne);
router.put('/locations/:locationid/reviews/:reviewid', auth,
  ➔ ctrlReviews.reviewsUpdateOne);
router.delete('/locations/:locationid/reviews/:reviewid', auth,
  ➔ ctrlReviews.reviewsDeleteOne);
```

Итак, промежуточное ПО сконфигурировано и готово к использованию. Буквально через минуту мы посмотрим, как применить его в контроллере, но сначала — как поступить с неправильным маркером, отклоненным промежуточным ПО.

Обрабатываем случай отклонения аутентификации

Если предоставленный маркер неправильный или, возможно, его вообще не существует, промежуточное ПО сгенерирует ошибку, чтобы предотвратить дальнейшее выполнение кода. Поэтому нужно перехватить эту ошибку и вернуть сообщение о том, что действие не авторизовано, и соответствующий код состояния (401).

Наилучшее место для его добавления — рядом с другими обработчиками ошибок в `app.js`. Мы сделаем его первым обработчиком ошибок, чтобы обобщенные перехватчики ему не помешали. Следующий фрагмент кода демонстрирует новый обработчик ошибок, добавленный в файл `app.js`:

```
// Обработчики ошибок
// Перехватываем ошибку "не авторизовано"
app.use(function (err, req, res, next) {
  if (err.name === 'UnauthorizedError') {
    res.status(401);
    res.json({"message" : err.name + ": " + err.message});
  }
});
```

Сделав это и перезапустив приложение, мы можем проверить выполнение отклонения, снова воспользовавшись Postman и на этот раз подтверждая отправку отзыва. Мы можем взять тот же самый запрос `POST`, который использовали при первом тестировании API. Результат показан на рис. 11.11.

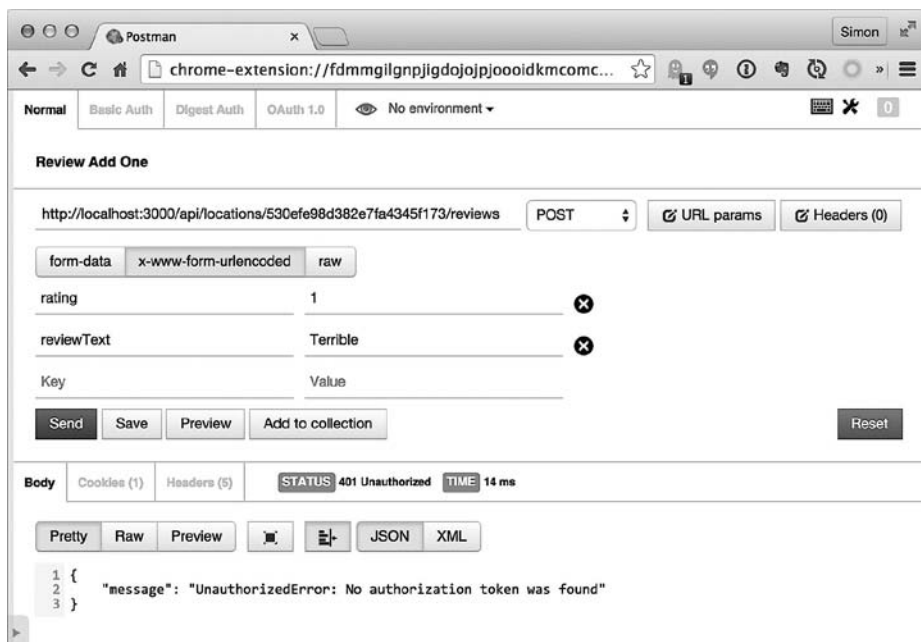


Рис. 11.11. Попытка добавить отзыв без правильного JWT теперь приводит к ответу 401

Как и ожидалось, попытка обратиться к теперь уже защищенному API без включения в запрос правильного JWT приводит к возврату сообщения о том, что действие не авторизовано, и соответствующего кода состояния. Это то, чего мы и хотели. Теперь перейдем к рассмотрению того, что будет происходить, когда запрос авторизован промежуточным ПО и выполнение продолжается в контроллере.

11.4.2. Использование информации из JWT в контроллере

В этом разделе мы рассмотрим, как использовать данные из JWT, извлеченные промежуточным ПО в Express и добавленные в объект `req`. Воспользуемся адресом электронной почты для получения имени пользователя из базы данных и добавления его в документ отзыва.

Выполняем код основного контроллера только тогда, когда пользователь существует

Первое, что нужно сделать, как показано в листинге 11.6, — это обернуть содержимое контроллера `reviewsCreate` в новую функцию, которую мы назовем `getAuthor`. Эта новая функция должна принимать на входе объекты `req` и `res`, возвращая код существующего контроллера в обратном вызове.

Основная задача функции `getAuthor` будет состоять в проверке того, существует ли пользователь в базе данных, и возвращении имени пользователя для применения в контроллере. Так что мы сможем передать его под именем `userName` в обратный вызов, который, в свою очередь, передаст его функции `doAddReview` (листинг 11.6).

Листинг 11.6. Модифицируем контроллер создания отзыва для получения сначала имени пользователя

```
module.exports.reviewsCreate = function(req, res) {
  getAuthor(req, res,
    function (req, res, userName) { ← Вызываем функцию
      if (req.params.locationid) {     getAuthor и передаем
        Loc                            код исходного контроллера
          .findById(req.params.locationid) в виде обратного вызова;
          .select('reviews')          передаем в обратном
          .exec(                       вызове имя пользователя
            function(err, location) {
              if (err) {
                sendJSONresponse(res, 400, err);
              } else {
                doAddReview(req, res, location, userName); ← Передаем
              }                               имя пользователя
            }                               в функцию
          );                               doAddReview
        } else {
          sendJSONresponse(res, 404, {
            "message": "Not found, locationid required"
          });
        }
      }
    }); ← Закрываем функцию getAuthor
  });
};
```

При взгляде на этот листинг мы замечаем две вещи, которые нам все еще осталось сделать: написать функцию `getAuthor` и модифицировать функцию `doAddReview`. Давайте напишем функцию `getAuthor`, чтобы понимать, как получить данные из JWT.

Проверяем пользователя и возвращаем его имя

Назначение функции `getAuthor` заключается в проверке того, что адрес электронной почты связан с пользователем системы, и возвращении имени этого пользователя для дальнейшего применения. Ей нужно будет:

- ❑ проверить наличие адреса электронной почты в объекте `req`;
- ❑ использовать адрес электронной почты для поиска пользователя;
- ❑ отправить имя пользователя в функции обратного вызова;
- ❑ перехватить ошибки и отправить соответствующие сообщения.

Полный код функции `getAuthor` приведен в листинге 11.7. Прежде всего мы проверяем свойство `payload` объекта `req`, у которого проверяем наличие свойства `email`. Напоминаем, что `payload` — свойство, которое мы задали при добавлении аутентификации в маршруты Express. После этого просто используем запись `req.payload.email` в запросе Mongoose, передавая имя пользователя обратному вызову в случае успеха.

Листинг 11.7. Используем данные из JWT для запроса к базе данных

```
var User = mongoose.model('User');
var getAuthor = function(req, res, callback) {
  if (req.payload && req.payload.email) {
    User
      .findOne({ email : req.payload.email })
      .exec(function(err, user) {
        if (!user) {
          sendJSONresponse(res, 404, {
            "message": "User not found"
          });
          return;
        } else if (err) {
          console.log(err);
          sendJSONresponse(res, 404, err);
          return;
        }
        callback(req, res, user.name);
      });
  } else {
    sendJSONresponse(res, 404, {
      "message": "User not found"
    });
    return;
  }
};
```

Убеждаемся, что модель User доступна

Проверяем наличие информации из JWT в объекте запроса

Используем адрес электронной почты для нахождения пользователя

Выполняем обратный вызов, передавая имя пользователя

Теперь при вызове функция обратного вызова будет выполнять исходный код контроллера, находя местоположение и передавая информацию функции `doAddReview`. Кроме того, она теперь передает этой функции имя пользователя, так что давайте побыстрее модифицируем функцию `doAddReview`, чтобы она использовала имя пользователя и добавляла его в документы отзывов.

Задаем имя пользователя для отзывов

Изменения в функции `doAddReview` очень просты, они показаны в листинге 11.8. Мы уже сохраняли автора (`author`) отзыва, получая данные из `req.body.author`. Теперь в функцию передается еще один параметр, который мы можем использовать взамен прежнего (листинг 11.8). Изменения выделены полужирным шрифтом.

Листинг 11.8. Сохраняем имя пользователя в отзыве

```

var doAddReview = function(req, res, location, author) {
  if (!location) {
    sendJSONresponse(res, 404, "locationid not found");
  } else {
    location.reviews.push({
      author: author,
      rating: req.body.rating,
      reviewText: req.body.reviewText
    });
    location.save(function(err, location) {
      var thisReview;
      if (err) {
        sendJSONresponse(res, 400, err);
      } else {
        updateAverageRating(location._id);
        thisReview = location.reviews[location.reviews.length - 1];
        sendJSONresponse(res, 201, thisReview);
      }
    });
  }
};

```

Добавляем параметр author в описание функции

Используем параметр author при создании поддокумента отзыва

Этими простыми изменениями завершаем работу с прикладной частью. Мы создали новую схему для пользователей, сгенерировали и использовали JWT, создали API аутентификации и защитили отдельные маршруты API. Это уже немало! И теперь мы собираемся перейти к клиентской части и сосредоточиться на интеграции сделанного в приложение Angular.

11.5. Создание сервиса аутентификации Angular

В приложении Angular, как и в любом другом, аутентификация, вероятно, будет необходима везде, в нескольких различных местах. Так что очевидным решением будет сделать сервис аутентификации таким, который можно будет использовать везде, где он понадобится.

Этот сервис будет отвечать за все имеющее отношение к аутентификации, включая сохранение и чтение JWT, возвращение информации о текущем пользователе и обращение к конечным точкам входа и регистрации нашего API.

Начнем с рассмотрения управления сеансами пользователей.

11.5.1. Управление сеансами пользователей в Angular

Ненадолго допустим, что пользователь только что выполнил вход и API вернул JWT. Как нам следует поступить с маркером? Поскольку мы выполняем SPA, то можем просто хранить его в памяти приложения. Это допустимо, но только если пользователь не решит обновить страницу, что приведет к перезагрузке приложения и потере всего находящегося в оперативной памяти. Отнюдь не идеально.

Итак, в дальнейшем мы поищем более надежное место для хранения маркера, которое дало бы приложению возможность читать его в любой момент, когда понадобится. Вопрос в том, что нам следует использовать — cookie-файлы или локальное хранилище?

Cookie-файлы в сравнении с локальным хранилищем

Традиционный подход к хранению данных пользователя в веб-приложении состоит в сохранении cookie-файла, и это, безусловно, заслуживающий рассмотрения вариант. Но cookie-файлы на самом деле предназначены для использования серверными приложениями: при каждом запросе к серверу cookie-файлы пересылаются в заголовке HTTP. В SPA это не требуется: конечные точки API не имеют состояния и не получают или не используют cookie-файлы.

Так что давайте поищем что-то еще и взглянем в сторону локального хранилища, разработанного в расчете на клиентские приложения. При использовании локального хранилища данные остаются в браузере и не передаются вместе с запросами.

Помимо этого, локальное хранилище очень удобно использовать с помощью JavaScript. Взгляните на следующий фрагмент кода, задающий и получающий определенные данные:

```
window.localStorage['my-data'] = 'Какая-то информация';  
window.localStorage['my-data']; // Возвращает 'Какая-то информация'
```

Отлично, решено: в LocalStorage мы будем использовать локальное хранилище для сохранения JWT.

Создаем сервис для сохранения и чтения JWT в локальном хранилище

Начнем построение сервиса аутентификации с создания методов для сохранения JWT в локальном хранилище и чтения его впоследствии. Мы только что видели, насколько удобно работать с локальным хранилищем в JavaScript, так что теперь просто нужно обернуть это в сервис Angular, предоставляющий два метода: `saveToken` и `getToken`.

Никаких особых сюрпризов нас тут не ожидает, просто метод `saveToken` должен принимать на входе значение, которое требуется сохранить, а `getToken` — возвращать значение. Во-первых, создадим файл `authentication.service.js` в каталоге `app_client/common/services`. В листинге 11.9 показано содержимое этого нового сервиса, включая первые два метода. Рекомендуемым решением в Angular будет использование `$window` вместо родного объекта `window` — в основном из соображений тестирования, так что внедрим его в сервис. Сервис должен быть зарегистрирован в приложении и должен возвращать методы.

Листинг 11.9. Создание сервиса аутентификации, включая первые два метода

```
(function () {
  angular
    .module('loc8rApp')
    .service('authentication', authentication);

  authentication.$inject = ['$window'];
  function authentication ($window) {

    var saveToken = function (token) {
      $window.localStorage['loc8r-token'] = token;
    };

    var getToken = function () {
      return $window.localStorage['loc8r-token'];
    };

    return {
      saveToken : saveToken,
      getToken  : getToken
    };
  }
})();
```

Регистрируем новый сервис в приложении

Внедряем сервис \$window

Создаем метод saveToken для сохранения значения в localStorage

Создаем метод getToken для чтения значения из localStorage

Делаем эти методы доступными приложению

Вот и все, довольно простой сервис для сохранения `loc8r-token` в хранилище `localStorage` и последующего его чтения. Не забудьте добавить ссылку на этот файл в массив `appClientFiles` в `app.js`!

Далее рассмотрим регистрацию и вход в приложение.

11.5.2. Предоставляем пользователям возможность регистрироваться, входить в приложение и выходить из него

Чтобы использовать этот сервис для предоставления пользователям возможности регистрироваться, входить в приложение и выходить из него, понадобится добавить еще три метода. Начнем с регистрации и входа.

Обращение к API для регистрации и входа в приложение

Нам понадобятся два метода для регистрации и входа в приложение, которые будут отправлять данные форм конечным точкам `register` и `login` API, созданным ранее в этой главе. В случае успешного выполнения обе эти конечные точки вернут JWT, так что мы можем использовать метод `saveToken` для его сохранения.

Следующий фрагмент кода показывает два новых метода, добавляемых в сервис. Не забудьте, что необходимо также добавить `register` и `login` в оператор `return`, чтобы сделать их доступными:

```
register = function(user) {
  return $http.post('/api/register', user).success(function(data) {
    saveToken(data.token);
  });
};
login = function(user) {
  return $http.post('/api/login', user).success(function(data) {
    saveToken(data.token);
  });
};
```

Не забывайте, что метод `$http` возвращает промисы, именно поэтому мы подключили метод `success` к цепочке вызовов запроса. Теперь рассмотрим выход из приложения.

Выполняем удаление из localStorage для выхода из приложения

Сеанс пользователя в приложении Angular управляется сохранением JWT в `localStorage`. Если маркер находится там, он правильный и срок его действия не истек, можем сказать, что пользователь в данный момент аутентифицирован. Мы не можем изменить дату истечения срока действия маркера изнутри приложения Angular, это может сделать только сервер. Мы лишь можем удалить его.

Таким образом, чтобы пользователи могли выходить из приложения, мы должны создать в сервисе аутентификации новый метод `logout` для удаления JWT `loc8r`, как показано в следующем фрагменте кода:

```
logout = function() {
  $window.localStorage.removeItem('loc8r-token');
};
```

Он просто удаляет элемент `loc8r-token` из `localStorage` браузера. Опять же не забудьте, что метод `logout` необходимо добавить в оператор `return`, чтобы он стал доступным приложению.

Теперь у нас имеются методы для извлечения JWT с сервера, сохранения его в хранилище `localStorage`, чтения из `localStorage`, а также удаления. Следующий вопрос: как мы будем использовать его в приложении, чтобы видеть, что пользователь выполнил вход, и при этом извлекать из него данные?

11.5.3. Использование данных из JWT в сервисе Angular

Сохраненный в `localStorage` браузера JWT — то, что мы используем для управления сеансом пользователя. Он будет применяться для проверки того, выполнил ли пользователь вход в приложение. Если да, то приложение может прочитать хранящуюся внутри него информацию о пользователе.

Вначале добавим метод для проверки того, аутентифицирован ли в настоящий момент кто-нибудь.

Проверка статуса входа в систему

Чтобы выяснить, аутентифицирован ли в настоящий момент пользователь в приложении, необходимо проверить, существует ли маркер `loc8r-token` в `localStorage`. Для этого можно использовать метод `getToken`. Но одного только существования маркера недостаточно. Не забывайте, что у JWT есть дата истечения срока действия, так что в случае существования маркера нужно проверить и ее.

Дата и время истечения срока действия JWT — часть содержимого, представляющего собой вторую часть данных. Как вы помните, эта часть — просто закодированный объект JSON, он не зашифрован, так что можно его декодировать. Собственно, мы уже обсуждали функцию для выполнения этого — `atob`.

Итак, резюмируем. Мы хотели бы создать метод, который:

- ❑ получает хранимый маркер;
- ❑ извлекает содержимое из маркера;
- ❑ декодирует содержимое;
- ❑ проверяет, не истек ли срок действия.

Этот метод будет просто возвращать `true`, если пользователь в данный момент аутентифицирован, и `false`, если нет. Следующий фрагмент кода демонстрирует все это вместе в методе `isLoggedIn`:

```
var isLoggedIn = function() {
  var token = getToken();

  if(token){
    var payload = JSON.parse($window.atob(token.split('.')[1]));

    return payload.exp > Date.now() / 1000;
  } else {
    return false;
  }
};
```

Получаем маркер из хранилища

Если маркер существует, извлекаем содержимое, декодируем его и выполняем синтаксический разбор в JSON

Проверяем, не истек ли срок действия

Здесь совсем немного кода, но он выполняет немало работы. Как только мы сошлемся на него в операторе `return` сервиса, приложение сможет быстро проверить, аутентифицирован ли в данный момент пользователь в любом нужном месте.

Следующий и последний метод, который нужно добавить в сервис аутентификации, будет извлекать из JWT информацию о пользователе.

Извлечение информации о пользователе из JWT

Нам хотелось бы, чтобы у приложения была возможность извлекать из JWT адрес электронной почты и имя пользователя. Мы только что видели в методе `isLoggedIn`, как извлекать данные из маркера, и наш новый метод будет делать то же самое.

Итак, создадим новый метод `currentUser`. Первое, что он будет делать, — проверить, имеются ли аутентифицированные пользователи, с помощью вызова метода `isLoggedIn`.

Если аутентифицированный пользователь нашелся, то метод будет получать его маркер посредством вызова метода `getToken`, прежде чем извлечь и декодировать содержимое и вернуть необходимые данные. Следующий фрагмент кода демонстрирует это:

```
var currentUser = function() {
  if(isLoggedIn()){
    var token = getToken();
    var payload = JSON.parse($window.atob(token.split('.')[1]));

    return {
      email : payload.email,
      name : payload.name
    };
  }
};
```

После реализации этого кода и ссылки на него в операторе `return` сервиса аутентификации Angular можно считать реализованным. Просматривая код, можно увидеть, что он получился довольно общим и удобным для копирования из одного приложения в другое. Все, что вам, вероятно, понадобится поменять, — это имя маркера и различные URL API, так что у нас получился отличный, допускающий повторное использование сервис Angular.

Теперь, когда в приложении уже есть этот сервис, можем его использовать. Так что продолжим и создадим страницы для входа и регистрации.

11.6. Создание страниц регистрации и входа в приложение

Все, что мы сделали до сих пор, замечательно, но без предоставления посетителям сайта способа действительно регистрироваться и входить на сайт будет совершенно бесполезно! Именно этим мы сейчас и займемся.

Говоря в терминах функциональности, нам нужны страница регистрации новых пользователей, на которой они могли бы ввести информацию о себе и зарегистрироваться, и страница входа для повторно посещающих сайт, где они могли бы ввести свои имя пользователя и пароль. После того как пользователь прошел через какой-либо из этих процессов и успешно аутентифицировался, приложение должно перенаправить его на страницу, где он был до начала процесса.

Начнем со страницы регистрации.

11.6.1. Создаем страницу регистрации

Для разработки работоспособной страницы регистрации необходимо:

- описать маршрут в конфигурации приложения Angular;
- создать для этой страницы представление;
- создать для этой страницы контроллер;
- добиться перенаправления на предыдущую страницу в случае успешного завершения.

Описываем маршрут в приложении Angular

Прежде всего опишем маршрут для страницы регистрации в конфигурации приложения Angular, находящейся в файле `app_client/app.js`. Маршрут будет `/register`, и мы разместим файл представления в новой иерархии каталогов `app_client/auth/register/`.

Итак, добавляемый в конфигурацию новый маршрут будет выглядеть следующим образом:

```
.when('/register', {  
  templateUrl: '/auth/register/register.view.html',  
  controller: 'registerCtrl',  
  controllerAs: 'vm'  
})
```

Тут нет ничего нового или интересного для нас, так что сохраним его и перейдем к созданию представления.

Создаем представление для регистрации

Что ж, теперь мы собираемся создать представление для страницы регистрации. Помимо обычного заголовка и нижнего колонтитула, в ней нам понадобится еще несколько вещей. В основном это форма, в которой посетители смогли бы ввести свои имя и адрес электронной почты и задать пароль. В этой форме нам также понадобится область для отображения ошибок, и еще вставим на страницу ссылку на форму входа на случай, если пользователь вспомнит, что он уже зарегистрирован.

Листинг 11.10 демонстрирует собранное воедино представление. Обратите внимание на то, как учетные данные привязаны к полям для ввода в представление модели с помощью команды `ng-model`.

Листинг 11.10. Полное представление для страницы регистрации

```

<navigation></navigation>

<div class="container">
  <page-header content="vm.pageHeader"></page-header>

  <div class="row">
    <div class="col-md-6 col-sm-12">
      <p class="lead">Already a member? Please
      ➔ <a href="/#login">log in</a> instead.</p>
      <form ng-submit="vm.onSubmit()"
      <div role="alert" ng-show="vm.formError" class="alert
      ➔ alert-danger">{{ vm.formError }}</div>
      <div class="form-group">
        <label for="name">Full name</label>
        <input type="text" class="form-control" id="name"
        ➔ placeholder="Enter your name" ng-model=
        ➔ "vm.credentials.name">
      </div>
      <div class="form-group">
        <label for="email">Email address</label>
        <input type="email" class="form-control"
        ➔ id="email" placeholder="Enter email"
        ➔ ng-model="vm.credentials.email">
      </div>
      <div class="form-group">
        <label for="password">Password</label>
        <input type="password" class="form-control"
        ➔ id="password" placeholder="Password"
        ➔ ng-model="vm.credentials.password">
      </div>
      <button type="submit" class="btn btn-default">Register!</button>
    </form>
  </div>
</div>
<footer-generic></footer-generic>
</div>

```

Ссылка на переход на страницу входа

div для отображения ошибок

Поле ввода имени пользователя

Поле ввода адреса электронной почты

Поле ввода пароля

И снова важно отметить, что имя пользователя, адрес электронной почты и пароль привязаны к модели представления в объекте `vm.credentials`. Теперь взглянем на зеркальное отображение этого и запрограммируем соответствующий контроллер.

Создаем каркас контроллера для регистрации

Исходя из этого представления нужно будет сделать несколько вещей в контроллере регистрации. Понадобятся текст названия для заголовка страницы и функция `vm.onSubmit` для обработки подтверждения отправки формы. Мы также зададим всем свойствам учетных данных значение по умолчанию в виде пустой строки.

Листинг 11.11 демонстрирует все это. Но есть еще один нюанс. После регистрации пользователей мы хотели бы перенаправить их обратно на страницу, где они находились прежде. Для этого воспользуемся строчным параметром запроса под названием `page`. Так что во все страницы, где мы создадим ссылки на URL регистрации, мы включим также строку запроса с указанием текущего URL — что-то вроде `/#register?page=/about`.

В контроллере, включенном в листинг 11.11, мы будем обрабатывать эту ситуацию также через функцию `vm.returnPage`, используя домашнюю страницу в качестве значения по умолчанию для случая, если строку запроса найти не удалось. Для получения строки запроса понадобится внедрить в контроллер сервис Angular `$location`.

Листинг 11.11. Каркас контроллера регистрации

```
(function () {

  angular
    .module('loc8rApp')
    .controller('registerCtrl', registerCtrl);

  registerCtrl.$inject = ['$location', 'authentication'];
  function registerCtrl($location, authentication) {
    var vm = this;

    vm.pageHeader = {
      title: 'Create a new Loc8r account'
    };

    vm.credentials = {
      name : "",
      email : "",
      password : ""
    };

    vm.returnPage = $location.search().page || '/';

    vm.onSubmit = function () {
    };
  }
})();
```

Внедряем `$location` и сервисы аутентификации в контроллер

Создаем экземпляр учетных данных

Получаем из строки запроса страницу, на которую нужно будет возвращаться

Создаем заглушку для функции `onSubmit`

Обратите внимание на то, как мы получаем параметр `page` из строки запроса посредством выполнения функции `search` по `location`. Теперь, когда у нас есть значение `returnPage`, нужно позаботиться о передаче его в том случае, если пользователь решит выбрать на странице ссылку для входа. Для этого просто надо изменить ссылку в представлении (изменения выделены полужирным шрифтом в следующем фрагменте кода):

```
<p class="lead">Already a member? Please <a href="/#login?page={{
vm.returnPage }}">log in</a> instead.</p>
```

Теперь, когда мы позаботились об этом, можем перейти к написанию кода функции `onSubmit`.

Обрабатываем отправку формы регистрации

При подтверждении отправки формы первое, что должен выполнить код, — проверить, все ли поля заполнены. Если в каких-то нет данных, можно отобразить ошибку аналогично тому, как мы делали при добавлении отзывов. Если же эта простейшая проверка пройдена, можно перейти к регистрации пользователя.

Для регистрации пользователя мы вызовем метод `register` из сервиса `authentication`, передав ему учетные данные. Помните, что метод `register` использует сервис `$http`, так что он будет возвращать промисы, которые можно присоединить в цепочке вызовов. Но если регистрация прошла успешно, мы можем очистить объект строки запроса, а затем задать путь приложения, равный перехваченной ранее переменной `returnPage`. Это действие перенаправит пользователя по данному пути.

Все это показано в следующем фрагменте кода, который необходимо добавить в контроллер регистрации:

```
vm.onSubmit = function () {
  vm.formError = "";
  if (!vm.credentials.name || !vm.credentials.email
  ➔ || !vm.credentials.password) {
    vm.formError = "All fields required, please try again"; ←
    return false;
  } else {
    vm.doRegister();
  }
};

vm.doRegister = function() {
  vm.formError = "";
  authentication
  .register(vm.credentials)
  .error(function(err){
    vm.formError = err;
  })
}
```

Если какие-либо учетные данные отсутствуют, отображаем сообщение об ошибке

В противном случае продолжаем регистрацию

Вызываем аутентификационный метод регистрации, передавая учетные данные

Отображаем ошибку формы, если произошел сбой регистрации


```

    .then(function(){
      $location.search('page', null);
      $location.path(vm.returnPage);
    });
  });
};

```

Если регистрация прошла успешно, очищаем строку запроса и перенаправляем пользователя

Не забудьте добавить этот файл контроллера в массив `appClientFiles` в файле `app.js`. После этого можно будет протестировать страницу и функциональность регистрации, запустив приложение и перейдя по адресу `http://localhost:3000/register`.

Когда вы проделаете это и успешно зарегистрируетесь в качестве пользователя, откройте инструменты разработчика для своего браузера и взгляните на ресурсы. Вы должны увидеть сервис `loc8r-token` в папке Local Storage (рис. 11.12).

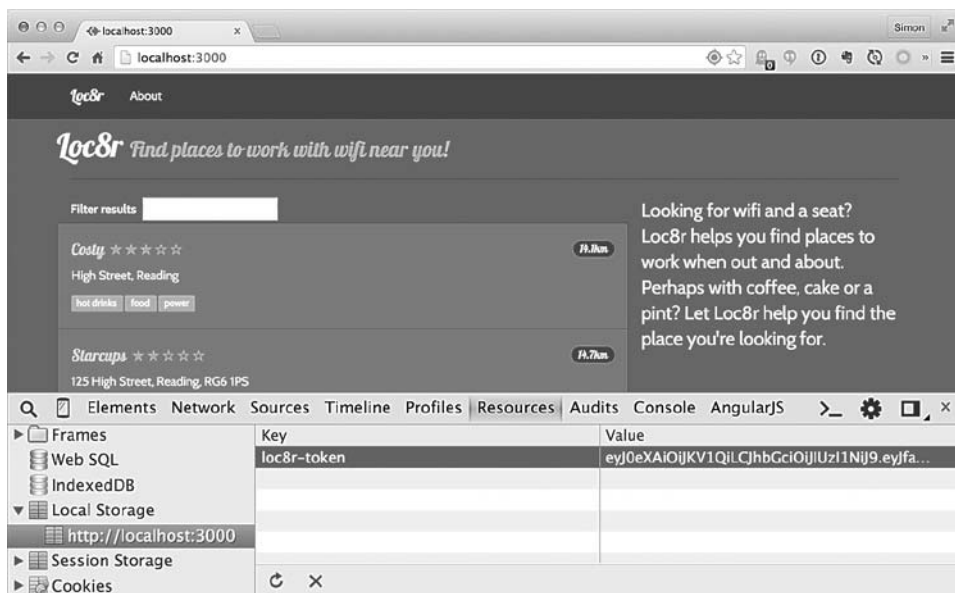


Рис. 11.12. Находим `loc8r-token` в браузере

Отлично. Итак, мы добавили возможность регистрации новых пользователей. Теперь дадим повторно посещающим сайт пользователям возможность выполнить вход.

11.6.2. Создаем страницу входа

Наш подход к созданию страницы входа очень похож на способ создания страницы регистрации. В основном это будет копирование и вставка, так что рассмотрим все шаги очень кратко.

Прежде всего нужно добавить новый маршрут в файл `app.js`:

```
.when('/login', {
  templateUrl: '/auth/login/login.view.html',
  controller: 'loginCtrl',
  controllerAs: 'vm'
})
```

Затем необходимо создать файл представления `login.view.html`. Из маршрута видно, где ему желательно находиться. Он очень похож на представление для регистрации, так что, вероятно, проще всего его скопировать и отредактировать. Все, что нужно, — убрать поле ввода имени и поменять несколько кусков текста. Изменения выделены полужирным шрифтом в следующем фрагменте кода:

```
<div class="col-md-6 col-sm-8">
  <p class="lead">Not a member? Please <a href="/#register?page={{
  vm.returnPage }}">register</a> first.</p>
  <form ng-submit="vm.onSubmit()">
    <div role="alert" ng-show="vm.formError" class="alert alert-danger">{{
    vm.formError }}</div>
    <div class="form-group">
      <label for="email">Email address</label>
      <input type="email" class="form-control" id="email" placeholder="Enter
      email" ng-model="vm.credentials.email">
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" id="password"
      placeholder="Password" ng-model="vm.credentials.password">
    </div>
    <button type="submit" class="btn btn-default">Sign in!</button>
  </form>
</div>
```

Меняем ссылку для входа на ссылку для регистрации

Обратите внимание, что поле ввода имени убрано

Меняем текст на кнопке

И наконец, контроллер для входа, который опять же очень похож на контроллер для регистрации. Необходимо сделать следующие изменения:

- ❑ поменять имя контроллера;
- ❑ поменять название страницы;
- ❑ убрать ссылки на поле ввода имени;
- ❑ переименовать `doRegister` на `doLogin`;
- ❑ вызвать метод `login` сервиса `authentication` вместо метода `register`.

Так что давайте создадим файл `login.controller.js`, скопировав `register.controller.js` и внося соответствующие изменения. Следующий фрагмент кода

демонстрирует содержимое этого файла (для краткости — без оберток ИФЕ), выполненные изменения выделены полужирным шрифтом:

```
angular
  .module('loc8rApp')
  .controller('loginCtrl', loginCtrl);

loginCtrl.$inject = ['$location', 'authentication'];
function loginCtrl($location, authentication) {
  var vm = this;

  vm.pageHeader = {
    title: 'Sign in to Loc8r' ← Меняем название страницы
  };

  vm.credentials = {
    email : "",
    password : ""
  };

  vm.returnPage = $location.search().page || '/';

  vm.onSubmit = function () {
    vm.formError = "";
    if (!vm.credentials.email || !vm.credentials.password) { ←
      vm.formError = "All fields required, please try again";
      return false;
    } else {
      vm.doLogin();
    }
  };

  vm.doLogin = function() {
    vm.formError = "";
    authentication
      .login(vm.credentials)
      .error(function(err){
        vm.formError = err;
      })
      .then(function(){
        $location.search('page', null);
        $location.path(vm.returnPage);
      });
  };
};
}
```

Меняем название на loginCtrl

Убираем ссылки на имя

Меняем doRegister на doLogin

Вызываем метод login вместо register

Это было легко! Нет нужды подробно останавливаться на этом, ведь функционально все работает так же, как и контроллер для регистрации. Просто не забудьте добавить его в массив `appClientFiles` в файле `app.js`, чтобы включить в код приложения.

Теперь перейдем к последнему этапу и воспользуемся аутентифицированным сеансом в приложении Angular.

11.7. Работаем с аутентификацией в приложении Angular

Как только у вас появляется возможность аутентифицировать пользователей, очевидный следующий шаг — использовать эту информацию. В `Loc8r` мы планируем сделать две вещи:

- менять навигацию в зависимости от того, аутентифицирован посетитель или нет;
- использовать информацию о пользователе при создании отзывов.

Сначала разберемся с навигацией.

11.7.1. Меняем навигацию

В навигации сейчас отсутствует ссылка для входа в приложение. Поэтому добавим ее в подходящем месте — справа вверху экрана. Но если пользователь уже аутентифицирован, отображать сообщение о входе не нужно, лучше отображать имя пользователя и предоставлять возможность выхода.

Именно этим мы и займемся в данном разделе, начав с добавления раздела на панель навигации с правой стороны.

Добавляем раздел с правой стороны панели навигации

Навигация для `Loc8r` сделана в виде директивы, включаемой в каждую страницу, файлы находятся в папке `common/directives/navigation`. В следующем фрагменте кода выделена полужирным шрифтом разметка, которую необходимо добавить в шаблон, чтобы поместить с правой стороны ссылку для входа в приложение:

```
<div id="navbar-main" class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li><a href="#/about">About</a></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="#/login/">Sign in</a></li>
  </ul>
</div>
```

Это будет отправной точкой. Но чтобы за этим стояла какая-то логика — например, для отображения имени аутентифицированного пользователя, — нам понадобится добавить к директиве контроллер.

Используем контроллер с директивой

До сих пор используемые нами директивы состояли из описания директивы и шаблона HTML. Но существует возможность привязать к директиве контроллер для получения еще большей функциональности. Именно это мы здесь и сделаем.

Это выполняется добавлением в описание директивы свойства `controller`, которое может задаваться с помощью встроенной версии синтаксиса `controllerAs`. Поскольку директива вложена внутрь других страниц с уже заданной моделью представления, во избежание конфликтов понадобится отличное от `vm` имя модели представления. В следующем фрагменте кода показано, как добавить контроллер в описание директивы и задать имя модели представления `navvm`:

```
function navigation () {
  return {
    restrict: 'EA',
    templateUrl: '/common/directives/navigation/navigation.template.html',
    controller: 'navigationCtrl as navvm'
  };
}
```

Отлично, теперь создадим соответствующий контроллер.

Создаем контроллер навигации

Мы создадим новый файл в том же каталоге, что и описание директивы и шаблон `navigation.controller.js`, — добавим его в массив `appClientFiles` в `app.js`. Этот контроллер должен будет выполнять несколько действий, в том числе довольно много взаимодействовать с сервисом `authentication`. Но первое, что ему понадобится сделать, — получить текущий путь URL, чтобы можно было добавить его в ссылку Sign in (Вход). Не забудьте, что мы хотели бы, чтобы пользователь после входа перенаправлялся обратно на исходную страницу.

Итак, создаем новое описание контроллера и передаем в него родной сервис `$location` и наш сервис `authentication`, так как знаем, что понадобятся оба. Ради согласованности кода назовем модель представления в функции `vm` — это внутренняя ссылка для функции контроллера, которая не должна соответствовать шаблону.

Следующий фрагмент кода показывает отправную точку для контроллера навигации и делает доступным текущий путь под именем `vm.currentPath`:

```
(function () {
  angular
    .module('loc8rApp')
    .controller('navigationCtrl', navigationCtrl);

  navigationCtrl.$inject = ['$location', 'authentication'];
  function navigationCtrl($location, authentication) {
    var vm = this;

    vm.currentPath = $location.path();
  }
})();
```

Сделав доступным текущий путь, мы можем использовать его в шаблоне, чтобы добавить ссылку для входа в виде строки запроса, следующим образом:

```
<li><a href="/#login/?page={{ navvm.currentPath }}">Sign in</a></li>
```

Теперь при успешном входе или регистрации пользователи будут возвращены на исходные позиции, чтобы продолжить делать то, что они делали. Следующий шаг предназначен для аутентифицированных пользователей — это отображение их имени и пункта меню для выхода вместо ссылки для входа.

Отображаем имя пользователя и ссылку для выхода: контроллер

Вспомогательное назначение этой части навигации — сообщать пользователям, что они авторизованы, отображая их имя вместо ссылки для входа. В процессе этого мы добавим также раскрывающийся пункт меню, чтобы пользователи могли уйти со страницы.

Именно при этом мы по-настоящему будем использовать методы сервиса аутентификации и некоторые данные из JWT, сохраненные в локальном хранилище. В контроллере навигации мы хотели бы знать, аутентифицирован ли пользователь или нет, для того чтобы отобразить правильный элемент навигации, а также имя пользователя. Листинг 11.12 демонстрирует контроллер, модифицированный добавлением вызовов методов `isLoggedIn` и `currentUser` сервиса аутентификации.

Нам также хотелось бы, чтобы у выполнившего вход пользователя была возможность выхода. Для этого создадим в сервисе аутентификации метод, так что нам нужен будет только способ вызывать его до перенаправления пользователей на страницу, на которой они находились до начала процесса. Мы могли бы организовать перенаправление на специальную страницу подтверждения, но пока что для этой цели подойдет домашняя страница. Это также является частью кода контроллера и показано в следующем листинге.

Листинг 11.12. Использование методов сервиса аутентификации в контроллере навигации

```
function navigationCtrl($location, authentication) {
  var vm = this;

  vm.currentPath = $location.path();
  vm.isLoggedIn = authentication.isLoggedIn();
  vm.currentUser = authentication.currentUser();

  vm.logout = function() {
    authentication.logout();
    $location.path('/');
  };
}
```

Определяем, авторизован ли посетитель

Получаем имя текущего пользователя

Создаем функцию для выхода, по завершении перенаправляем на домашнюю страницу

Теперь, когда функции готовы, можем соответствующим образом модифицировать представление.

Отображаем имя пользователя и ссылку для выхода: шаблон HTML

Для отображения правильного элемента навигации в зависимости от статуса входа в систему (**logged-in status**) **пользователя мы будем использовать родные директивы** Angular `ng-show` и `ng-hide` и получать у контроллера значение `isLoggedIn`. Если метод `isLoggedIn` возвращает `true`, будем скрывать ссылку для входа и отображать другую разметку.

Создание раскрывающегося меню с помощью Bootstrap потребует значительного количества разметки, так что в следующем фрагменте кода важные моменты выделены полужирным шрифтом:

```

<ul class="nav navbar-nav navbar-right">
  <li ng-hide="navvm.isLoggedIn"><a href="/#login/?page={{
  ➔ navvm.currentPath }}>Sign in</a></li>
  <li ng-show="navvm.isLoggedIn" class="dropdown"> ← Отображаем
    <a href="" class="dropdown-toggle" data-toggle=
  ➔ "dropdown">{{ navvm.currentUser.name }} <span
  ➔ class="caret"></span></a> ← Отображаем
    <ul class="dropdown-menu" role="menu">
      <li><a href="" ng-click="navvm.logout()"> ←
  ➔ Logout</a></li>
      </ul>
    </li>
</ul>

```

Скрываем ссылку для входа для уже выполнивших вход пользователей

Отображаем имя пользователя

Отображаем раскрывающийся элемент навигации для выполнивших вход пользователей

Добавляем ссылку для выхода в раскрывающееся меню

Завершив это, мы получаем в `Loc8r` полностью функционирующую систему аутентификации, проверяющую статус текущего посетителя и отображающую сохраненную информацию в браузере. Перейдем к последней части функциональности — добавлению имени пользователя в отзыв.

11.7.2. Добавляем данные пользователя в отзыв

Основной сценарий использования для аутентификации в `Loc8r` — прием отзывов только от зарегистрированных и выполнивших вход пользователей. Здесь понадобится делать кое-какие вспомогательные вещи, например отображать аутентифицированным пользователям только кнопку `Add Review` (Добавить отзыв) и убрать из формы поле ввода имени. Новое и важное в данном разделе — передача `JWT` из сервиса `loc8rData` конечной точке `API`. Начнем с самого интересного, а затем расставим все по своим местам.

Передаем JWT защищенной конечной точке API

JWT передается вместе с запросом конечной точке API в виде HTTP-заголовка `Authorization`. Обращение для добавления отзыва в `Loc8r` находится в сервисе `loc8rData`, располагающемся в папке `app_client/common/services`. Сам вызов использует метод `$http.post`.

Для добавления HTTP-заголовка методу достаточно просто добавить в вызов объект `headers` в виде части параметра `options`, как показано в листинге 11.13. Параметр `options` следует за параметрами `URL` и `data`, а содержимым заголовка `Authorization` должно быть слово `Bearer`, за которым следуют пробел и JWT. Нам придется внедрить наш сервис аутентификации в этот сервис, чтобы получить маркер. Все это выделено в листинге полужирным шрифтом.

Листинг 11.13. Модификация сервиса данных для передачи JWT

```
loc8rData.$inject = ['$http', 'authentication'];
function loc8rData ($http, authentication) {
    var locationByCoords = function (lat, lng) {
        return $http.get('/api/locations?lng=' + lng + '&lat=' + lat +
        ➔ '&maxDistance=20');
    };

    var locationById = function (locationid) {
        return $http.get('/api/locations/' + locationid);
    };

    var addReviewById = function (locationid, data) {
        return $http.post('/api/locations/' + locationid + '/reviews', data, {
            headers: {
                Authorization: 'Bearer ' + authentication.getToken()
            }
        });
    };

    return {
        locationByCoords : locationByCoords,
        locationById : locationById,
        addReviewById : addReviewById
    };
}
```

Внедряем сервис аутентификации

Добавляем параметр options для передачи нового заголовка HTTP, содержащего JWT

После создания этого заголовка конечная точка для добавления отзыва будет способна читать ожидаемый ею JWT и проверять пользователя. Теперь мы быстро соберем все вместе, сначала организовав показ кнопки `Add Review` (Добавить отзыв) только тогда, когда пользователь аутентифицирован.

Отображаем различные кнопки в зависимости от статуса текущего пользователя

Мы хотим показывать на странице различный контент в зависимости от того, аутентифицирован текущий посетитель или нет. Мы хотим показывать пользователям кнопку Add Review (Добавить отзыв) на странице подробностей о местоположении только тогда, когда они аутентифицированы. Если же пользователь не аутентифицирован, мы можем поменять эту кнопку на приглашение войти в приложение.

Основанный на некоторых проделанных ранее в этой главе вещах, таких как использование методов `currentPath` и `isLoggedIn`, а также `ng-show` и `ng-hide`, следующий фрагмент кода выделяет полужирным шрифтом изменения, которые нужно сделать в файле `locationDetail.view.html`:

```
<div class="panel-heading">
  <a ng-show="vm.isLoggedIn" ng-click="vm.popupReviewForm()" class="btn
  ➤ btn-default pull-right">Add review</a>
  <a ng-hide="vm.isLoggedIn" href="/#/login?page={{ vm.currentPath }}"
  ➤ class="btn btn-default pull-right">Login to add review</a>
  <h2 class="panel-title">Customer reviews</h2>
</div>
```

С точки зрения представления тут ничего сложного нет, просто необходимо обеспечить, чтобы у соответствующего контроллера были методы `isLoggedIn` и `currentPath`. Как показано в следующем фрагменте кода, нам просто нужно передать сервисы `$location` и `authentication` и описать методы. Обратите внимание на то, что в этом фрагменте показана только часть контроллера, остальную часть кода необходимо оставить неизменной:

Внедряем сервисы

`$location` и `authentication`

```
locationDetailCtrl.$inject = ['$routeParams', '$location', '$modal',
➤ 'loc8rData', 'authentication'];
function locationDetailCtrl ($routeParams, $location, $modal, loc8rData,
➤ authentication) {
  var vm = this;
```

```
  vm.locationid = $routeParams.locationid;
```

```
  vm.isLoggedIn = authentication.isLoggedIn(); ←
```

```
  vm.currentPath = $location.path(); ←
```

Создаем метод `isLoggedIn` для получения статуса текущего посетителя

Получаем текущий путь URL посетителя

После сохранения этих изменений кнопка будет вести себя по-другому для аутентифицированных пользователей. Оба статуса состояния показаны на рис. 11.13.



Рис. 11.13. Два различных состояния кнопки Add Review (Добавить отзыв) в зависимости от того, аутентифицирован ли текущий пользователь

Это хороший способ, и работает он отлично. Но при нажатии пользователем кнопки для добавления отзыва форма в модальном всплывающем окне по-прежнему содержит поле ввода для имени пользователя.

Убираем поле ввода для имени из формы отзыва

Нам больше не требуется, чтобы пользователи вводили их имена в форму, так как API извлечет их из JWT. Поэтому можно *удалить* из файла `reviewModal.view.html` следующий фрагмент кода:

```
<div class="form-group">
  <label for="name" class="col-xs-2 col-sm-2 control-label">Name</label>
  <div class="col-xs-10 col-sm-10">
    <input id="name" name="name" required="required"
      ng-model="vm.formData.name" class="form-control"/>
  </div>
</div>
```

В отсутствие этого поля формы нам больше не нужно его проверять или управлять его значением API, так что можно также удалить ссылки на него из контроллера. В следующем фрагменте кода выделены полужирным шрифтом места, которые необходимо удалить из файла `reviewModal.controller.js`:

```
vm.onSubmit = function () {
  vm.formError = "";
  if (!vm.formData.name || !vm.formData.rating || !vm.formData.reviewText) {
    vm.formError = "All fields required, please try again";
    return false;
  } else {
    vm.doAddReview(vm.locationData.locationid, vm.formData);
  }
};

vm.doAddReview = function (locationid, formData) {
  loc8rData.addReviewById(locationid, {
    author : formData.name,
```

```

    rating : formData.rating,
    reviewText : formData.reviewText
  })
  .success(function (data) {
    vm.modal.close(data);
  })
  .error(function (data) {
    vm.formError = "Your review has not been saved, please try again";
  });
return false;
};
};

```

Рисунок 11.14 демонстрирует, как выглядит форма отзыва без поля ввода имени.

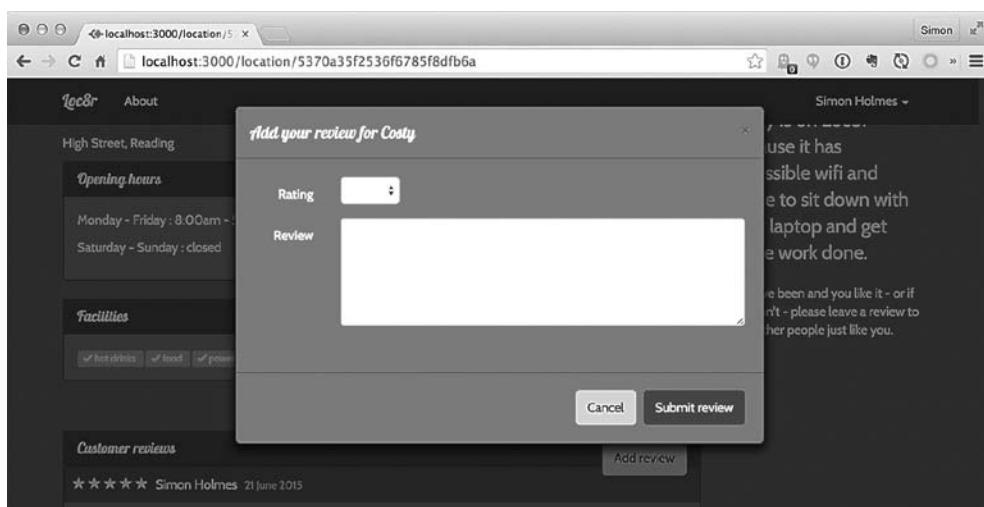


Рис. 11.14. Окончательная форма отзыва без поля ввода имени. Обратите внимание на имя пользователя в правом верхнем углу и вариант кнопки Add Review (Добавить отзыв) для случая аутентифицированного пользователя

На этом изменении мы завершаем относящийся к аутентификации раздел. Пользователи должны быть аутентифицированы для добавления отзывов, а благодаря системе аутентификации отзыву будет присвоено правильное имя пользователя.

11.8. Резюме

В этой главе мы рассмотрели следующее.

- Работу с аутентификацией в стеке MEAN.
- Шифрование паролей с помощью хешей и соли.
- Использование методов модели Mongoose для добавления функций в схемы.

- ❑ Создание веб-маркеров JSON с помощью Express.
- ❑ Управление аутентификацией на сервере с помощью Passport.
- ❑ Обеспечение доступности маршрутов в Express только аутентифицированным пользователям.
- ❑ Использование локального хранилища для управления сеансами пользователей в браузере.
- ❑ Использование данных JWT в Angular.
- ❑ Добавление контроллеров в директивы Angular.
- ❑ Передачу JWT из Angular в API через заголовки HTTP.

На этом мы завершаем нашу книгу. Теперь у вас уже должно быть хорошее представление о возможностях стека MEAN и вам должно быть по силам создание многих замечательных вещей!

У вас теперь есть платформа для создания различных API REST, серверных веб-приложений и браузерных одностраничных приложений. Вы умеете создавать ориентированные на работу с базой данных сайты, различные API и приложения, а также публиковать их по реальному адресу в Интернете.

Начиная следующий проект, не забудьте выделить немного времени на то, чтобы продумать наилучшие архитектуру и алгоритм взаимодействия с пользователем. Потратьте немного времени на планирование, чтобы разработка проходила продуктивнее и доставила вам удовольствие. И никогда не бойтесь переделывать и улучшать свой код и приложение в ходе работы.

Если вы размышляете, за изучение чего вам взяться дальше, взгляните на Gulp на сайте www.gulpjs.com — отличную систему сборки для автоматизации таких действий, как линтинг, сокращение и конкатенация кода. В зависимости от того, над какими проектами вы трудитесь, у вас может появиться желание углубиться в Angular, Express, Node, Mongo или Mongoose.

На самом деле мы затронули лишь вершину айсберга возможностей, обеспечиваемых этими замечательными технологиями. Так что не робейте, погружайтесь в них, создавайте что-то, пробуйте, продолжайте их изучение и, самое главное, наслаждайтесь!

Приложения

Приложение А

Установка стека

В этом приложении:

- ❑ установка Node и npm;
- ❑ глобальная установка Express;
- ❑ установка MongoDB;
- ❑ установка Angular.

Прежде чем создавать что-либо на стеке MEAN, необходимо установить программное обеспечение для него. Сделать это в Windows, Mac OS X и наиболее распространенных дистрибутивах Linux, таких как Ubuntu, совсем несложно.

Поскольку в основе стека лежит Node, лучше всего начать с него. Помимо этого, Node поставляется с системой управления пакетами npm, которая очень пригодится при установке некоторых других частей программного обеспечения.

А.1. Установка Node и npm

Наилучший способ установки Node и npm зависит от вашей операционной системы. При возможности рекомендуется скачать программу установки с сайта Node по адресу <http://nodejs.org/download/>. По этому адресу, поддерживаемому командой разработчиков ядра Node, всегда находится самая свежая версия.

А.1.1. Установка Node в Windows

Пользователям Windows рекомендуется просто скачать программу установки с сайта Node.

А.1.2. Установка Node в Mac OS X

Наилучшим вариантом для пользователей Mac OS X будет скачать программу установки с сайта Node. В качестве альтернативы можете установить Node и npm с помощью системы управления пакетами Homebrew, как описано на посвященном Node вики-сайте компании Joyent по адресу <https://github.com/nodejs/node/wiki/Installing-Node.js-via-package-manager>.

А.1.3. Установка Node в Linux

Для пользователей Linux программ установки нет, но вы можете скачать исполняемые файлы с сайта Node, если вам удобно с ними работать.

В качестве альтернативы пользователи Linux могут также установить Node из систем управления пакетами. Учтите, версия систем управления пакетами не всегда самая свежая, особенно это касается популярной системы apt на Ubuntu. Инструкции по использованию множества систем управления пакетами, включая исправление для apt, можно найти на посвященном Node вики-сайте компании Joyent по адресу <https://github.com/nodejs/node/wiki/Installing-Node.js-via-package-manager>.

А.1.4. Проверяем установку путем сверки версий

После установки Node и npm можно проверить версии с помощью двух команд терминала:

```
$ node --version  
$ npm --version
```

При этом будут выведены версии имеющихся на вашей машине Node и npm. Код в данной книге создавался с использованием Node 4.2.1 и npm 2.2.0.

А.2. Глобальная установка Express

Чтобы иметь возможность создавать новые приложения Express на лету из командной строки, необходимо установить генератор приложений Express. Это можно сделать из командной строки с помощью npm. Просто выполните в терминале следующую команду:

```
$ npm install -g express-generator
```

Если она выдаст сбой из-за связанной с правами ошибки, необходимо будет выполнить ее от имени администратора. В Windows щелкните правой кнопкой мыши на значке программы командной строки и выберите **Run As Administrator** (Запустить от имени администратора). Теперь попробуйте выполнить предыдущую команду еще раз в новом окне. В Mac OS X и Linux можно указать перед командой `sudo`, как показано в следующем фрагменте кода, после этого вам будет предложено ввести пароль:

```
$ sudo npm install -g express-generator
```

Когда установка генератора приложений будет завершена, вы сможете проверить ее из терминала путем сверки номера версии:

```
$ express --version
```

В примерах кода в данной книге использовалась версия 4.9.0.

Если вы столкнетесь с какими-либо проблемами в процессе установки, документация для Express доступна на его сайте по адресу: <http://expressjs.com/>.

A.3. Установка MongoDB

MongoDB тоже доступен в Windows, Mac OS X и Linux. Подробные инструкции обо всех возможных опциях доступны в онлайн-документации MongoDB по адресу <http://docs.mongodb.org/manual/installation/>.

A.3.1. Установка MongoDB в Windows

По адресу <http://docs.mongodb.org/manual/installation/> имеется возможность напрямую скачать программы установки в Windows в зависимости от используемой вами версии Windows.

A.3.2. Установка MongoDB в Mac OS X

Простейший способ установить MongoDB в Mac OS X — использовать систему управления пакетами Homebrew, но, если хотите, можете установить MongoDB вручную.

A.3.3. Установка MongoDB в Linux

Для нескольких дистрибутивов Linux также доступны пакеты, как подробно описано на сайте <http://docs.mongodb.org/manual/installation/>. Если вы используете версию Linux, для которой MongoDB недоступна в виде пакета, можете установить ее вручную.

А.3.4. Запуск MongoDB в качестве сервиса

После установки MongoDB, вероятно, вы захотите запустить ее в качестве сервиса, чтобы при перезагрузке компьютера она перезапускалась автоматически. В документации по установке MongoDB имеются инструкции относительно выполнения этой процедуры.

А.3.5. Проверка номера версии MongoDB

MongoDB устанавливает не только себя, но и командную оболочку Mongo, так что у вас есть возможность взаимодействовать со своими базами данных MongoDB через командную строку. Вы можете проверить номера версий MongoDB и командной оболочки Mongo независимо друг от друга. Для проверки версии командной оболочки выполните в терминале следующее:

```
$ mongo --version
```

Для проверки версии MongoDB выполните:

```
$ mongod --version
```

Данная книга использует версию 2.4.6 как MongoDB, так и командной оболочки Mongo.

А.4. Установка Angular

Angular не требует особой установки, поскольку это всего лишь файл библиотеки, который нужно скачать и поместить в соответствующее место в структуре каталогов. Скачать Angular можно с его домашней страницы по адресу <http://angularjs.org/>.

При начале загрузки у вас будет возможность выбрать из нескольких вариантов. Необходимо скачать сборку *minified* ветви *stable*. Используемая на протяжении данной книги версия — 1.2.19.

После завершения скачивания создайте новый каталог `angular` внутри каталога `public` своего приложения и поместите туда файл JavaScript.

Приложение Б

Установка и подготовка вспомогательного программного обеспечения

В этом приложении:

- ❑ добавление Bootstrap и пользовательских тем;
- ❑ установка Git;
- ❑ установка подходящего интерфейса командной строки;
- ❑ подписка на Heroku;
- ❑ установка набора инструментов Heroku.

Существует несколько технологий, которые могут быть полезны при разработке программ для стека MEAN, начиная с макетов клиентской части и заканчивая управлением исходным кодом и утилитами для развертывания. Данное приложение охватывает установку и настройку вспомогательных технологий, используемых на протяжении данной книги. Так как сами инструкции по установке склонны меняться со временем, это приложение укажет вам наилучшие места, где можно получить инструкции и все, что вам может понадобиться найти.

Б.1. Twitter Bootstrap

Bootstrap как таковой, по существу, не устанавливается, а скорее добавляется в приложение. Это сводится к скачиванию файлов библиотек, их разархивированию и помещению в соответствующее место приложения.

Первый этап заключается в скачивании Bootstrap. Его можно скачать с <http://www.getbootstrap.com/>. Удостоверьтесь, что вы скачиваете ZIP-архив дистрибутива, а не исходного кода. На момент написания данной книги текущая версия Bootstrap 3.0.2, а ZIP-архив дистрибутива содержит три каталога: `css`, `fonts` и `js`.

Когда вы его скачаете и разархивируете, необходимо переместить файлы в каталог `public` вашего приложения Express. Чтобы все файлы находились рядом и не загромождали верхний уровень, создайте новый каталог `Bootstrap` в каталоге `public` и скопируйте туда разархивированные файлы. Каталог `public` в вашем приложении должен выглядеть так, как показано на рис. Б.1.

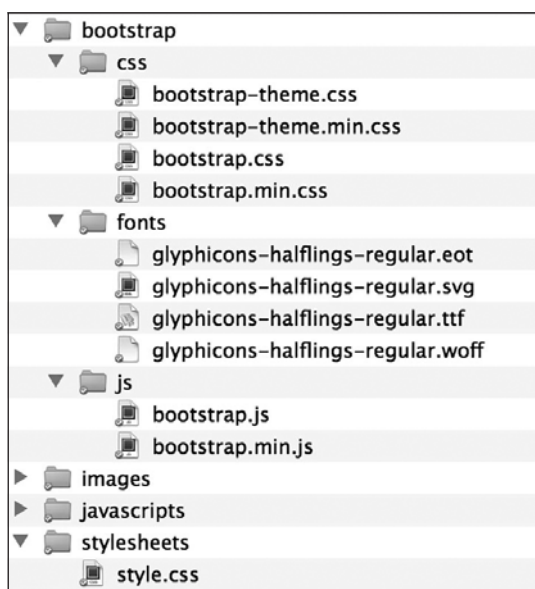


Рис. Б. 1. Структура и содержимое каталога `public` после добавления Bootstrap

Это даст вам доступ к внешнему виду и возможностям Bootstrap в варианте по умолчанию, но, вероятно, вы захотите, чтобы ваше приложение немного отличалось от других. Этого можно добиться с помощью добавления темы.

Б.1.1. Получение темы **Amelia**

Приложение `Loc8r` в данной книге использует тему Bootstrap из `Bootwatch` под названием `Amelia`. Со времени написания данной книги `Bootwatch` убрал тему `Amelia` со своего сайта, но вы можете скачать как исходный файл CSS, так и сокращенную версию из моего репозитория на GitHub по адресу <https://github.com/simonholmes/amelia>. После скачивания файлов `amelia.bootstrap.css` и `amelia.bootstrap.min.css` вы сможете скопировать их в каталог `/public/bootstrap/css` своего приложения.

Б.1.2. Чистка каталогов

Если хотите, можете почистить свой каталог Bootstrap, удалив часть дубликатов. Вы увидите, что там есть как удобочитаемые, так и сокращенные версии файлов CSS и JavaScript, а также пара CSS-файлов темы Bootstrap по умолчанию. Если вы не собираетесь влезать в файлы, то вам нужна фактически только сокращенная версия.

Б.2. Установка Git

Исходный код для этой книги организован с помощью Git, так что простейший способ доступа к нему — через Git. Помимо этого, Heroku использует Git для управления процессом развертывания и отправки кода с вашей машины для разработки в промышленную среду в Интернет. Так что необходимо установить Git, если он у вас еще не установлен.

Проверить, установлен ли он у вас, можно с помощью простой команды терминала:

```
$ git --version
```

Если она выдает номер версии, то Git у вас уже установлен и можно переходить к следующему разделу. Если же нет, нужно его установить.

Хорошим отправным пунктом для пользователей Mac OS X и Windows, которые не имели дела с Git, будет скачать и установить пользовательский интерфейс GitHub с сайта <https://help.github.com/articles/set-up-git>.

Однако GUI не обязателен, и вы можете установить сам Git, следуя инструкциям, размещенным на основном сайте Git по адресу <http://git-scm.com/downloads>.

Б.3. Установка подходящего интерфейса командной строки

Добиться наилучших результатов от Git можно с помощью CLI, даже если вы скачали и установили GUI. Некоторые из них лучше других, и использовать родную командную строку Windows вы не сможете, так что, если вы работаете в Windows, вам придется применять что-то другое. Вот список того, что я использую в нескольких различных средах:

- ❑ Mac OS X Mavericks и более поздние версии: родной терминал;
- ❑ Mac OS X до Mavericks (10.8.5 и более ранние версии): iTerm;
- ❑ Windows: командная оболочка GitHub (устанавливается вместе с GUI GitHub);
- ❑ Ubuntu: родной терминал.

Если вы предпочитаете что-то другое и команды Git там работают, конечно, используйте то, что уже используете и к чему привыкли.

Б.4. Настройка Heroku

Данная книга использует Heroku для размещения приложения Loc8r в промышленной среде в Интернете. Вы тоже можете это делать — бесплатно, — если только зарегистрируетесь, установите набор инструментов и войдете через терминал.

Б.4.1. Подписка на Heroku

Для использования Heroku вам необходимо зарегистрироваться для получения учетной записи. Для создания приложения, над которым вы будете работать во время чтения данной книги, достаточно будет бесплатной учетной записи. Просто перейдите по адресу <http://www.heroku.com/> и следуйте инструкциям для регистрации.

Б.4.2. Установка набора инструментов Heroku

Набор инструментов Heroku содержит оболочку командной строки Heroku и утилиту Foreman. Командная оболочка — это то, что вы будете использовать из терминала для управления развертыванием на Heroku, а Foreman исключительно удобна при настройке правильного запуска всего созданного на вашей машине на Heroku. Скачать набор инструментов для Mac OS X, Windows и Linux можно с сайта toolbelt.heroku.com.

Б.4.3. Вход в Heroku с помощью терминала

Когда вы зарегистрируете учетную запись и установите на своей машине набор инструментов, останется последнее — войти в учетную запись из терминала. Введите следующую команду:

```
$ herokulogin
```

Она запросит ваши учетные данные для входа в Heroku и, вероятнее всего, сгенерирует новый открытый ключ SSH и загрузит его на сервер. Теперь все настроено и готово для работы с Heroku.

Приложение В

Разбираемся со всеми представлениями

В этом приложении:

- ❑ удаляем данные из всех представлений, кроме домашней страницы;
- ❑ перемещаем данные в контроллеры.

В главе 4 рассмотрена настройка контроллеров и представлений для работоспособного статического прототипа. Вопросы «Как?» и «Почему?» рассмотрены в этой главе более детально, так что данное приложение будет посвящено тому, каким должен оказаться конечный результат.

В.1. Перемещение данных из представлений в контроллеры

Эта задача включает перемещение данных обратно по потоку выполнения MVC — из представлений в контроллеры. В главе 4 в качестве примера эта задача была выполнена для домашней страницы `Loc8r`, но необходимо сделать это и для остальных страниц. Начнем со страницы `Details` (Подробности).

В.1.1. Страница `Details` (Подробности)

Страница `Details` (Подробности) — самая большая и сложная из всех, с наибольшими информационными потребностями, но после домашней страницы более логично будет заняться ею. Первое, что нужно сделать, — настроить контроллер.

Настройка контроллера

Контроллер для этой страницы называется `locationInfo` и находится в файле `locations.js` в каталоге `app_server/controllers`. После анализа данных в представлении и объединения их в JavaScript-объект ваш контроллер должен выглядеть примерно так, как показано в листинге В.1.

Листинг В.1. Контроллер `locationInfo`

```
module.exports.locationInfo = function(req, res){
  res.render('location-info', {
    title: 'Starcups',
    pageHeader: {title: 'Starcups'},
    sidebar: {
      context: 'is on Loc8r because it has accessible wifi and space to sit
      down with your laptop and get some work done.',
      callToAction: 'If you\'ve been and you like it - or if you don\'t -
      please leave a review to help other people just like you.'
    },
    location: {
      name: 'Starcups',
      address: '125 High Street, Reading, RG6 1PS',
      rating: 3,
      facilities: ['Hot drinks', 'Food', 'Premium wifi'],
      coords: {lat: 51.455041, lng: -0.9690884},
      openingTimes: [{
        days: 'Monday - Friday',
        opening: '7:00am',
        closing: '7:00pm',
        closed: false
      },{
        days: 'Saturday',
        opening: '8:00am',
        closing: '5:00pm',
        closed: false
      },{
        days: 'Sunday',
        closed: true
      }],
      reviews: [{
        author: 'Simon Holmes',
        rating: 5,
        timestamp: '16 July 2013',
        reviewText: 'What a great place. I can\'t say enough
        good things about it.'
      },{
        author: 'Charlie Chaplin',
        rating: 3,
        timestamp: '16 June 2013',
```

Включаем координаты широты и долготы для использования в изображении из Google Map

Добавляем массив часов работы с возможностью указания различных данных для разных дней [недели]

Массив для отзывов, оставленных другими пользователями

```

    reviewText: 'It was okay. Coffee wasn\'t great,
    but the wifi was fast.'
  }
}
});
};

```

Отметим передачу координат широты и долготы. Получить текущие значения вашей широты и долготы можно с сайта <http://www.where-am-i.net/>.

Преобразовать адрес в географические координаты, то есть широту и долготу, можно на сайте <http://www.latlong.net/convert-address-to-lat-long.html>. Ваши представления будут использовать координаты `lat` и `lng` для изображения нужного местоположения из Google Map, так что имеет смысл сделать это на стадии прототипа.

Модифицируем представление

Поскольку это наиболее сложная и насыщенная данными страница, логично, что у нее будет самый большой шаблон представления. Вы уже видели в макете домашней страницы большую часть технических деталей, таких как организация циклов по массивам, внедрение включаемых файлов, описание и вызов примесей. В этом шаблоне, однако, есть еще два дополнительных нюанса, на которые стоит обратить внимание, они оба прокомментированы и выделены полужирным шрифтом в листинге.

Во-первых, этот шаблон использует условный оператор `if-else`. Он напоминает аналог из JavaScript, только без скобок. Во-вторых, шаблон использует функцию JavaScript `replace` для замены всех разрывов строк в тексте отзывов на теги `
`. Это выполняется с помощью простого регулярного выражения с поиском всех случаев употребления символов `\n` в тексте. Листинг В.2 демонстрирует шаблон представления `location-info.jade` полностью.

Листинг В.2. Шаблон представления `location-info.jade` из каталога `app_server/views`

```
extends layout
```

```
include _includes/sharedHTMLfunctions
```

Внедряем включаемый файл `sharedHTMLfunctions`, в котором находится примесь `outputRating`

```
block content
```

```
  .row.page-header: .col-lg-12
    h1= pageHeader.title
```

```
  .row
```

```
    .col-xs-12.col-md-9
```

```
      .row
```

```
        .col-xs-12.col-sm-6
```

```
          p.rating
```

```
            +outputRating(location.rating)
```

Вызываем примесь `outputRating`, передавая ей оценку текущего местоположения

```
            p= location.address
```

```
          .panel.panel-primary
```

```
            .panel-heading
```

```
              h2.panel-title Opening hours
```



```

.panel-body
  each time in location.openingTimes
    p
      | #{time.days} :
      if time.closed
        | closed
      else
        | #{time.opening} - #{time.closing}
.panel.panel-primary
  .panel-heading
    h2.panel-title Facilities
  .panel-body
    each facility in location.facilities
      span.label.label-warning
      span.glyphicon.glyphicon-ok
      | &nbsp;#{facility}
      | &nbsp;
.col-xs-12.col-sm-6.location-map
  .panel.panel-primary
  .panel-heading
    h2.panel-title Location map
  .panel-body
    img.img-responsive.img-rounded(src=
      "http://maps.googleapis.com/maps/api/
      staticmap?center=#{location.coords.lat},
      #{location.coords.lng}&zoom=17&size=400x350
      &sensor=false&markers=#{location.coords.lat},
      #{location.coords.lng}&scale=2")
  .row
    .col-xs-12
      .panel.panel-primary.review-panel
      .panel-heading
        a.btn.btn-default.pull-right(href="/location/review/new")
        Add review
      h2.panel-title Customer reviews
      .panel-body.review-container
        each review in location.reviews
          .row
            .review
              .well.well-sm.review-header
                span.rating
                  +outputRating(review.rating)
                span.reviewAuthor #{review.author}
                small.reviewTimestamp #{review.timestamp}
              .col-xs-12
                p !{{(review.reviewText).replace(/\n/g, '<br/>')}}
.col-xs-12.col-md-3
  p.lead #{location.name} #{sidebar.context}
  p= sidebar.callToAction

```

Организуем цикл по часам работы, проверяя, закрыто ли местоположение, с помощью встроенного оператора if-else

Формируем URL для статического изображения из Google Maps, вставляя lat и lng через переменные Jade

Выполняем цикл по всем отзывам, снова вызывая примесь outputRating для генерации разметки для звезд

Этот код заменяет все разрывы строк в тексте отзыва на тег
, чтобы он визуализировался так, как хотел автор

Может возникнуть вопрос: а зачем каждый раз заменять разрывы строк тегами `
`? Почему просто не сохранять сразу данные с тегами `
`? При этом нужно было бы выполнять функцию `replace` только один раз, при сохранении данных. Ответ: HTML — лишь один из методов визуализации текста, просто так случилось, что тут мы используем именно его. В дальнейшем вы можете захотеть извлечь эту информацию в родное мобильное приложение. При этом вам вряд ли захочется, чтобы исходные данные были замусорены разметкой HTML, которую вы не будете использовать в данной среде. Так что, по сути, это делается для сохранения чистоты данных.

В.1.2. Страница Add Review (Добавление отзыва)

Страница Add Review (Добавление отзыва) в настоящий момент очень проста: в ней есть только один элемент данных — название в заголовке страницы. Так что модификация контроллера не должна вызвать особых сложностей. См. полный код контроллера `AddReview`, находящегося в файле `locations.js` из каталога `app_server/controllers`, в листинге В.3.

Листинг В.3. Контроллер AddReview

```
module.exports.addReview = function(req, res){
  res.render('location-review-form', {
    title: 'Review Starcups on Loc8r',
    pageHeader: { title: 'Review Starcups' }
  });
};
```

Здесь особо нечего обсуждать: мы просто поменяли текст в названиях. Листинг В.4 демонстрирует соответствующее представление, `location-review-form.jade`, в папке `app_server/views`.

Листинг В.4. Шаблон location-review-form.jade

```
extends layout
block content
  .row.page-header
    .col-lg-12
      h1= pageHeader.title
  .row
    .col-xs-12.col-md-6
      form.form-horizontal(action="/location", method="get", role="form")
        .form-group
          label.col-xs-2.col-sm-2.control-label(for="name") Name
          .col-xs-10.col-sm-10
            input#name.form-control(name="name")
```

```

label.col-xs-10.col-sm-2.control-label(for="rating") Rating
.col-xs-12.col-sm-2
  select.form-control.input-sm(name="rating")
    option 5
    option 4
    option 3
    option 2
    option 1
.form-group
  label.col-sm-2.control-label(for="review") Review
.col-sm-10
  textarea#review.form-control(name="review", rows="5")
  button.btn.btn-default.pull-right Add my review
.col-xs-12.col-md-4

```

Опять же тут нет ничего сложного или нового, так что займемся страницей About (О нас).

В.1.3. Страница About (О нас)

Страница About (О нас) тоже не содержит больших объемов данных, только название и какой-то контент. Так что извлечем его из представления и переместим в контроллер. Обратите внимание на то, что в контенте из представления в настоящий момент имеется несколько тегов `
`, так что замените все теги `
` на `\n` при помещении его в контроллер. Это выделено полужирным шрифтом в листинге В.5. Контроллер `about` находится в файле `app_server/controllers/others.js`.

Листинг В.5. Контроллер `about`

```

module.exports.about = function(req, res){
  res.render('generic-text', {
    title: 'About Loc8r',
    content: 'Loc8r was created to help people find places to sit down and
➤ get a bit of work done.\n\nLorem ipsum dolor sit amet, consectetur
➤ adipiscing elit. Nunc sed lorem ac nisi dignissim accumsan. Nullam
➤ sit amet interdum magna. Morbi quis faucibus nisi. Vestibulum mollis
➤ purus quis eros adipiscing tristique. Proin posuere semper tellus, id
➤ placerat augue dapibus ornare. Aenean leo metus, tempus in nisl eget,
➤ accumsan interdum dui. Pellentesque sollicitudin volutpat ullamcorper.'
  });
};

```

Кроме удаления HTML из контента, ничего особенного тут нет. Так что взглядом на представление, и на этом наша работа будет завершена. Листинг В.6 демонстрирует итоговое представление `generic-text`, используемое на странице About

(О нас), в каталоге `app_server/views` представление будет применять тот же код, который мы видели в относящемся к отзывам разделе, для замены разрывов строк `\n` на HTML-теги `
`.

Листинг В.6. Шаблон `generic-text.jade`

```
extends layout
block content
  #banner.page-header
    .row
      .col-md-6.col-sm-12
        h1= title
    .row
      .col-md-6.col-sm-12
        p !{{(content).replace(/\n/g, '<br/>')}} ←
```

Заменяет разрывы строк на теги `
` при визуализации HTML-кода

Это очень простой, маленький и допускающий повторное использование шаблон, который вы можете применять всякий раз, когда захотите вывести какой-то текст на странице.

Саймон Холмс
Стек MEAN. Mongo, Express, Angular, Node

Перевел с английского И. Пальти

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Рощина</i>
Художник	<i>С. Заматевская</i>
Корректор	<i>Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 27.12.16. Формат 70×100/16. Бумага офсетная. Усл. п. л. 39,990. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, гоб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гоб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гоб. 6217;
e-mail: kuznetsov@piter.com





КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: www.piter.com
- по электронной почте: books@piter.com
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

- БЕСПЛАТНАЯ ДОСТАВКА:**
- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
 - почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничает с крупнейшими книжными магазинами.

Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com

Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com