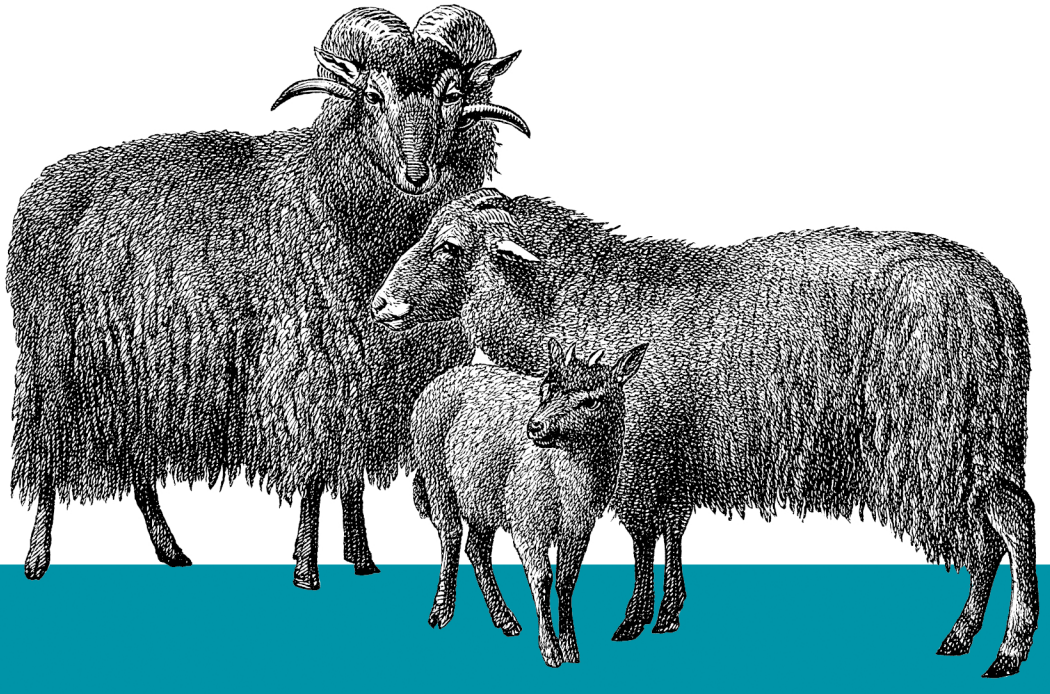
O'REILLY®



Основы разработки веб-приложений

ПИТЕР®

Сэмми Пьюривал

Semmy Purewal

Learning Web App Development

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Сэмми Пьюривал

Основы разработки веб-приложений

ПИТЕР®

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

2015

С. Пьюривал

Основы разработки веб-приложений

Серия «Бестселлеры O'Reilly»

Перевел с английского *О. Сивченко*

Заведующий редакцией	*Д. Виницкий*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художник	*В. Шимкевич*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*А. Засулевич*

ББК 32.988.02
УДК 004.738.5

Пьюривал С.

П96 Основы разработки веб-приложений. — СПб.: Питер, 2015. — 272 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-01226-3

Благодаря этой книге вы усвоите основы создания веб-приложений, построив простое приложение с нуля с помощью HTML, JavaScript и других свободно предоставляемых инструментов. Это практическое руководство на реальных примерах обучает неопытных веб-разработчиков тому, как создавать пользовательский интерфейс, строить серверную часть, организовывать связь клиента и сервера, а также применять облачные сервисы для развертывания приложения.

Каждая глава содержит практические задачи, полноценные примеры, а также ментальные модели процесса разработки. Эта книга поможет вам сделать первые шаги в создании веб-приложений, обеспечив глубокие знания по теме.

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

ISBN 978-1449370190 англ.

ISBN 978-5-496-01226-3

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 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 19.09.14. Формат 70×100/16. Усл. п. л. 21,930. Тираж 1000. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

Краткое содержание

Оглавление

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Об авторе

Сэмми Пьюривал около 10 лет преподавал компьютерные дисциплины и работал фрилансером-консультантом по JavaScript. За это время он успел поработать с разнообразными группами клиентов, включая стартапы, некоммерческие организации и исследовательские центры. Сейчас его основная работа — инженер-программист в Сан-Хосе, Калифорния.

Предисловие

В начале 2008 года, через шесть лет после окончания школы и работы учителем на полставки, мне очень хотелось стать преподавателем компьютерных дисциплин на полный день. Очень быстро выяснилось, что место преподавателя найти нелегко, а получение хорошей работы зависит от удачи в большей степени, чем от чего-либо еще. Ну что ж, я поступил так, как поступает любой уважающий себя академик, столкнувшись с удручающим положением на академическом рынке труда, а именно: решил повысить свою конкурентоспособность с помощью изучения разработки веб-приложений.

Это, конечно, звучит странно. Кроме всего прочего, к тому моменту я уже около девяти лет изучал компьютерные дисциплины и, более того, свыше шести лет учил студентов разрабатывать программное обеспечение (ПО). Разве я не должен был хорошо знать, как создавать веб-приложения? Похоже, что нет, так как существует определенный разрыв между практической ежедневной работой по разработке ПО и программированием как учебной дисциплиной, изучаемой в колледжах и университетах. Фактически мои знания по веб-разработке были ограничены HTML и в некоторой степени CSS, который я в то время изучал самостоятельно.

К счастью, у меня было несколько друзей, которые активно работали в компьютерном мире, и большинство из них в то время обсуждало (относительно) новый фреймворк[1], который назывался Ruby on Rails. Мне показалось, что это весьма подходящая область для развития, так что я купил несколько книг по этой теме и принялся читать обучающие материалы в Интернете, чтобы побыстрее освоиться.

А через несколько месяцев, пытаясь хоть чего-нибудь добиться на практике, я чуть было не сдался.

Почему? Да потому, что большинство книг и учебных статей начиналось с предположения, что я уже умею создавать веб-приложения и делаю это на протяжении нескольких лет! А между тем, несмотря на мой солидный теоретический багаж по компьютерному программированию, оказалось, что все эти материалы слишком лаконичны и очень сложны для понимания. Например, выяснилось, что можно пройти несколько классов по компьютерным дисциплинам, ни разу не столкнувшись с шаблоном проектирования Model — View — Controller, а в некоторых книгах уже в первой главе предполагается, что вы прекрасно с ним знакомы.

Тем не менее мне удалось изучить веб-разработку на уровне, достаточном для того, чтобы несколько раз провести консультации, которые оказались весьма кстати, пока я не получил должность преподавателя. Благодаря этому я заметил, что меня настолько увлекают практические стороны данной области, что я продолжил заниматься консультированием, одновременно работая учителем.

Через несколько лет занятий тем и другим мне предложили вести мой первый класс по разработке веб-приложений в Университете Северной Каролины в Эшвилле. Изначально я планировал *начать* с Ruby on Rails, но, взявшись за новейшие книги и обучающие материалы по ней, выяснил, что они никак не улучшились за все эти годы. Нет, они были хорошим подспорьем для людей, которые отлично знают основы, но для студентов, которые у меня учились, они определенно не годились.

Грустно, но неудивительно — академические книги по веб-разработке оказались еще хуже. Большинство из них содержали устаревшие концепции и не раскрывали важнейших тем, нужных для понимания платформ наподобие Ruby on Rails. Мне даже случилось выступить рецензентом одной книги, переизданной в 2011 году и до сих пор описывающей верстку с помощью таблиц и тег `<font>`!

Что ж, у меня не было другого выхода, кроме как создавать свой курс с нуля и писать все материалы самостоятельно. В то время я проводил небольшую консультационную работу по Node.js (адаптация JavaScript для стороны сервера) и подумал, что было бы интересно попробовать создать курс, обучающий одному и тому же языку и для клиента, и для сервера. Более того, я поставил себе цель дать моим студентам достаточно знаний для самостоятельного изучения Ruby on Rails, если они решат продолжить.

Эта книга содержит большую часть материалов, созданных мной во время преподавания этого курса в Университете Северной Каролины в Эшвилле. В ней

---

[1] Фреймворк — структура программной системы, а также специальное ПО, с помощью которого можно разрабатывать и объединять компоненты программного проекта. — *Примеч. пер.*

описано, как создать простое веб-приложение на основе базы данных с нуля, используя JavaScript. Сюда включены описание простейшего рабочего процесса (с использованием текстового редактора и системы контроля версий), основы технологий клиентской стороны (HTML, CSS, jQuery, Javascript), основы серверных технологий (Node.js, HTTP, базы данных), основы облачного развертывания (Cloud Foundry) и несколько примеров правильной практики написания кода (функции, MVC, DRY). Во время нашего пути мы исследуем фундаментальные основы языка JavaScript, научимся программировать, используя объекты и массивы, а также рассмотрим ментальные модели, которые соответствуют этому типу разработки ПО.

Выбор технологии

Для контроля версий я выбрал Git, потому что... ну хорошо, это Git, и он прекрасен. Кроме того, он дал моим студентам возможность изучить GitHub, который набирает все большую популярность. Хотя я не рассматриваю GitHub в этой книге, разобраться с ним совсем несложно, как только вы освоитесь с Git.

Я решил использовать для клиента jQuery, потому что он все еще остается популярным и мне очень нравится с ним работать. Я сам не использую никаких других фреймворков для клиента, хотя и упоминаю Twitter Bootstrap и Zurb Foundation в главе 3. Я решил не касаться современных клиентских фреймворков вроде Backbone или Ember, потому что считаю их слишком сложными для начинающих. А вот с Rails вы легко сможете начать работать после прочтения этой книги.

Для серверной стороны я выбрал Express, так как он (относительно) упрощенный и не догматический. Я решил также не рассматривать шаблоны клиентской и серверной стороны, поскольку считаю, что вначале важно научиться делать это вручную.

Я не рассматриваю и реляционные базы данных, так как вряд ли возможно полноценно раскрыть эту тему в течение времени, выделенного на нее в рамках курса. Реляционным базам данных я предпочел MongoDB из-за того, что они широко используются в сообществе Node.js и применяются JavaScript в качестве языка запросов. Кроме того, мне очень нравится Redis, поэтому его мы также изучим.

Я выбрал Cloud Foundry в качестве платформы для развертывания, потому что, как я выяснил (вместе с Heroku и Nodejitsu), она была одной из трех, предлагающих бесплатное использование и не требующих кредитной карты для настройки внешних сервисов. В любом случае различия между платформами незначительны и переход с одной на другую не потребует больших усилий.

Поможет ли вам эта книга

Эта книга не предназначена для того, чтобы сделать вас «ниндзя», или «суперзвездой», или даже хорошим компьютерным программистом. Она не подготовит вас к немедленному трудоустройству, и я даже не обещаю показать вам «правильный путь» в работе.

В то же время книга даст вам глубокое знание основ, которые необходимы для понимания того, как отдельные части современного веб-приложения взаимодействуют друг с другом, и дадут вам стартовую точку для дальнейшего изучения темы. Если вы как следует проработаете материал книги, то будете знать все, что в свое время нужно было мне для начала изучения Rails.

Вы извлечете больше всего пользы из этой книги, если у вас есть небольшой опыт программирования и нет никакого опыта в веб-разработке. Как минимум вы должны быть знакомы с основными программными конструкциями, такими как схемы if-else, циклы, переменные и типы данных. Впрочем, я не жду, что у вас есть какой-то опыт в объектно-ориентированном программировании или каком-то конкретном языке программмирования. Вы можете получить необходимые знания, изучив материалы в Khan Academy или Code Academy или пройдя курс программирования в ближайшем колледже.

Я надеюсь, что эта книга может быть использована не только для самостоятельного изучения, но и в качестве учебного материала в общественных классах по разработке веб-приложений или, возможно, как курс для одного семестра (14 недель) в колледже.

Занятия с книгой

Разработка веб-приложений — очень нужный вам навык. Держа это в уме, я писал книгу, рассчитывая, что ее будут читать активно. Это значит, что самого лучшего результата вы достигнете, читая ее около компьютера и по-настоящему набирая все примеры. Конечно, такой подход связан с некоторым риском — всегда есть опасность, что примеры кода не будут работать, если вы не наберете их точно так, как они написаны в книге. Чтобы снизить риск, я создал хранилище GitHub со всеми примерами из этой книги в рабочем состоянии. Вы можете посмотреть их в Интернете, перейдя по ссылке http://www.github.com/semmypurewal/LearningWebAppDev. Поскольку все полные примеры находятся там, я постараюсь избегать вставки на страницы книги избыточного кода.

Кроме того, значительную часть примеров я оставил незаконченными. По моему мнению, вам полезно будет закончить их самостоятельно. Я советую сделать это прежде, чем смотреть на законченные примеры, размещенные в Интернете. В каждой главе находится также некоторое количество практических задач и указателей на источники информации, так что рекомендую вам поработать и над ними.

Преподавание с этой книгой

Преподавая этот материал в 14-недельном курсе, я обычно уделял 2–3 недели материалу первых трех глав и 3–4 недели — материалу последних трех. Это значит, что больше всего времени я тратил на средние три главы, которые охватывают программирование на JavaScript, jQuery, AJAX и Node.js. Студенты, которых я учу,

испытывают больше всего трудностей с массивами и объектами, поэтому я уделяю дополнительное время этой теме, так как она очень важна для компьютерного программирования в целом.

Определенно, я объясняю свой предмет более компьютерно-научным способом, чем большинство книг на эти темы, поэтому данное издание хорошо подходит для курса компьютерного программирования. В частности, я уделяю внимание ментальным моделям, таким как дерево или иерархическая система, а также стараюсь выделить функциональные подходы к программированию, если это имеет смысл (хотя и стараюсь не подчеркивать этого в лекции). Если вы преподаете информатику, то можете уделить этим аспектам материала больше внимания.

Я не собираюсь публиковать решения практических задач (хотя, возможно, сделаю это, если получу очень много запросов), поэтому вы легко можете назначать их в качестве домашнего задания и внеклассной работы.

Куда обратиться в случае затруднений

Как я уже говорил, у нас есть хранилище GitHub, которое содержит все примеры кода, приведенные в этой книге. Кроме того, вы можете узнавать о найденных опечатках и необходимых обновлениях на http://learningwebappdev.com.

Я также попытаюсь быть доступным для контакта и буду очень рад оказать вам необходимую помощь. Пожалуйста, не стесняйтесь обращаться в мой Twitter (@semmypurewal) с короткими вопросами или комментариями, а также писать в любое время на электронный адрес me@semmy.me, если у вас более обширный вопрос. Я также предлагаю вам пользоваться функциональностью issues в нашем хранилище GitHub, чтобы задавать вопросы. Я приложу все усилия, чтобы отвечать как можно быстрее.

Общие комментарии к коду

Я старался писать код одновременно идиоматическим способом и настолько ясно, насколько это возможно. Так что у меня были две противоречащие друг другу цели, в результате чего в некоторых случаях мои решения в коде неидеальны — по педагогическим причинам. Надеюсь, что эти примеры очевидны для опытных разработчиков и не причинят каких-либо трудностей начинающим в долгосрочной перспективе.

Весь код работает корректно в современных браузерах, я протестировал его полностью в Google Chrome. Однако не могу гарантировать, что все будет работать хорошо в старых версиях Internet Explorer. Пожалуйста, дайте мне знать, если найдете дефекты браузерной совместимости в Internet Explorer 10+ или современной версии любого другого браузера.

В большинстве случаев я следовал идиомам JavaScript, однако в некоторых случаях пришлось отклониться. Например, для текстовых строк я предпочитаю двойные, а не одинарные кавычки в первую очередь потому, что предполагаю

наличие у моих студентов знания теоретических основ Java/C++. Я использую кавычки для названий свойств объектных констант, в результате чего JSON не очень отличается от объектов JavaScript. Кроме того, я ставлю `$` первым символом в переменных, которые указывают на объекты jQuery. Мне кажется, это делает код более ясным и легкочитаемым для новичков.

Условные обозначения

В книге применяются следующие условные обозначения.

Курсивный шрифт

Им обозначаются новые термины и понятия.

Шрифт для названий

Используется для обозначения URL, адресов электронной почты, а также сочетаний клавиш и названий элементов интерфейса.

`Шрифт для команд`

Применяется для обозначения программных элементов — переменных и названий функций, типов данных, переменных окружения, операторов, ключевых слов и т. д.

`Шрифт для листингов`

Используется в листингах программного кода.

`Шрифт для листингов полужирный`

Обозначает команды или другой текст, который должен быть введен пользователем.

`Шрифт для листингов курсивный`

Обозначает текст, который должен быть заменен величинами, введенными пользователем или определенными из контекста.

Этот элемент означает совет или рекомендацию.

Такой элемент означает общее замечание.

Этот элемент означает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступен для загрузки по адресу http://www.github.com/semmypurewal/LearningWebAppDev.

Эта книга создана, чтобы помочь вам выполнить свою работу. Если пример кода предлагается в книге, можете использовать его в своих программах и документации. Вам не нужно спрашивать разрешения на использование, если только вы не собираетесь скопировать значительный фрагмент кода. Например, написание программы с использованием нескольких фрагментов кода из этой книги не требует разрешения. Ответ на чей-либо вопрос с упоминанием этой книги и цитированием примера кода не требует разрешения. Для вставки же значительного количества кода из этой книги в документацию вашего продукта разрешение нужно. Мы не требуем указания ссылки на источник, но будем очень благодарны за это.

Если вы считаете, что использование примеров кода требует разрешения, так как отличается от приведенных ранее случаев, пожалуйста, напишите нам по адресу permissions@oreilly.com.

Safari® Books Online

Safari® Books Online — цифровая библиотека, работающая по запросу и предоставляющая экспертное собрание книг и видео ведущих авторов мира по технологиям и бизнесу.

Технологические специалисты, разработчики ПО, веб-дизайнеры и другие профессионалы в области бизнеса и творчества используют Safari® Books Online в качестве основного ресурса для исследований, решения проблем, обучения и сертификационных тренировок.

Safari® Books Online предлагает ряд наборов продуктов и программ оплаты для коммерческих организаций, государственных учреждений и частных лиц. Подписчики в одной удобной для поиска базе данных получают доступ к тысячам книг, обучающих видео и предпечатных рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и десятков других. Для получения более подробной информации о Safari® Books Online вы можете посетить сайт.

Как с нами связаться

Пожалуйста, адресуйте комментарии и вопросы, касающиеся этой книги, издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

У книги есть собственная веб-страница, где мы размещаем информацию о найденных опечатках, примеры и другую информацию. Она находится на http://oreil.ly/learning-web-app.

Чтобы прокомментировать или задать технический вопрос об этой книге, пожалуйста, пишите на bookquestions@oreilly.com.

Более подробная информация об этой книге, курсах, конференциях и новостях находится на нашем сайте по адресу http://www.oreilly.com.

Присоединяйтесь к нам на Facebook: http://facebook.com/oreilly.

Подписывайтесь на наш Twitter: http://twitter.com/oreillymedia.

Смотрите наш канал на YouTube: http://www.youtube.com/oreillymedia.

Выражения признательности

Огромное спасибо ребятам из департамента информатики в Университете Северной Каролины в Эшвилле за возможность провести этот курс дважды. И, конечно, спасибо студентам за терпение, поскольку материал постепенно эволюционировал.

Спасибо редактору Мег Бланшетт за огромные усилия по удержанию меня в нужном русле и, конечно, за бесконечное терпение из-за нарушения сроков. Я буду скучать по нашей еженедельной электронной переписке!

Спасибо Саймону Сент-Лорену за множество советов в самом начале работы и подсказки о том, что может подойти издательству O'Reilly.

Сильван Кавано и Марк Филипс внимательно прочитали каждую главу и высказывали множество очень полезных замечаний на протяжении всей работы. Эмили Уотсон прочитала первые четыре главы и внесла очень много дельных предложений. Майк Уилсон прочел последние четыре главы и дал бесценные технические советы. Я бесконечно благодарен всем вам и надеюсь когда-нибудь отплатить вам тем же.

Боб Бенитс, Уилл Бласко, Дэвид Браун, Ребекка Дэвид, Андреа Фей, Эрик Хоуи, Брюс Хауман, Джон Максвелл, Сюзан Рейсер, Бен Роузен и Вэл Скарлата прочитали множество версий материала и дали много полезных рекомендаций. Я искренне благодарен за потраченные вами время и усилия. Вы молодцы!

Несмотря на множество опытных рецензентов и друзей, которые просматривали материал, практически невозможно написать подобную книгу без каких-то технических ошибок, опечаток и неверных решений. Я принимаю на себя полную ответственность за все это.

1 Рабочий процесс

Создание веб-приложений — сложный процесс, включающий использование множества подвижных частей и интерактивных компонентов. Чтобы изучить, как это делается, мы должны разобрать эти части на отдельные элементы и постараться понять, как все они взаимодействуют друг с другом. А самое удивительное, что компонент, с которым мы будем сталкиваться чаще всего, не содержит ни строчки кода!

В этой главе мы исследуем рабочий процесс разработки веб-приложений — последовательность действий, которые выполняем для того, чтобы создать приложение. При этом мы изучим основы работы с некоторыми инструментами, которые делают этот процесс легко контролируемым и (в основном) безболезненным.

Эти инструменты — текстовый редактор, система контроля версий и браузер. Мы не будем изучать все это глубоко, но исследуем достаточно для того, чтобы оказаться готовыми начать веб-программирование на клиентской стороне. В главе 2 мы увидим рабочий процесс в действии, изучая HTML.

Если вы уже знакомы с этими инструментами, возможно, вам будет лучше просмотреть краткое изложение и примеры в конце этой главы, а затем двигаться дальше.

Текстовые редакторы

Инструмент, с которым вы будете сталкиваться чаще всего, — текстовый редактор. Об этом необходимом элементе технологии порой незаслуженно забывают, но на самом деле это самый важный инструмент в вашем рабочем чемоданчике, поскольку это программа, с помощью которой вы взаимодействуете со своим кодом. Поскольку код формирует конкретные элементы конструкции приложения, очень важно, чтобы было как можно проще его создавать и редактировать. К тому же вы, как правило, будете редактировать несколько файлов одновременно, так что очень важно, чтобы текстовый редактор предоставлял возможность быстро и просто перемещаться по файловой системе.

В прошлом вы, вероятно, тратили много времени на написание документов и редактирование текста в программах наподобие Microsoft Word или Google Docs. Это не лучшие примеры редакторов, о которых мы говорим. Эти редакторы концентрируются скорее на возможностях форматирования текста, чем на максимальной простоте его редактирования. Редактор, который будем использовать

мы, обладает очень скромными возможностями форматирования, но зато имеет множество функций, позволяющих эффективно манипулировать текстом.

На другом конце спектра находятся интегрированные среды разработки (Integrated Development Environments, IDE), такие как Eclipse, Visual Studio и XCode. Эти продукты, как правило, включают в себя функции простого управления кодом, но у них есть также очень много инструментов, важных при промышленной разработке ПО. Мы не будем рассматривать такие инструменты в этой книге, так как стремимся к простоте.

Так какие же текстовые редакторы будем рассматривать? В современной разработке веб-приложений обычно используются две основные категории. Первая — редакторы с графическим пользовательским интерфейсом (Graphical User Interface, GUI). Поскольку я предполагаю, что вы знакомы с основами программирования и информатики, то, скорее всего, у вас есть опыт работы с настольными приложениями с графическим пользовательским интерфейсом. Следовательно, эти редакторы будут для вас относительно удобными. Они отлично подходят для работы с мышью в качестве устройства ввода и имеют понятные меню, которые позволяют взаимодействовать с файловой системой как с любой другой программой. Примерами текстовых редакторов с графическим пользовательским интерфейсом могут послужить TextMate, Sublime Text и Coda.

Другая категория текстовых редакторов — терминальные редакторы (текстовые редакторы, работающие из командной строки). Эти редакторы разработаны еще до того, как были изобретены графический пользовательский интерфейс или мыши, поэтому их изучение — весьма нетривиальная задача для людей, привыкших к взаимодействию с компьютером посредством интерфейса и мыши. Однако эти редакторы могут быть весьма эффективными, если вы уделите их изучению достаточно времени. Наиболее популярные редакторы из этой категории — Emacs (рис. 1.1) и Vim (рис. 1.2).

Рис. 1.1. Документ HTML, открытый в Emacs

Рис. 1.2. Документ HTML, открытый в Vim

В этой книге мы будем пользоваться редактором с графическим интерфейсом, который называется Sublime Text, но я все же очень рекомендую вам поработать с Emacs или Vim. Если вы продолжите деятельность в сфере разработки веб-приложений, весьма возможно, что будете сталкиваться с их использованием очень часто.

Установка Sublime Text

Sublime Text (или для краткости Sublime) — популярный многофункциональный текстовый редактор, идеально подходящий для веб-разработки. Кроме этого, у него есть преимущество кросс-платформенности, то есть он работает одинаково эффективно с Windows, Mac OS и Linux. Это платная программа, но вы можете скачать бесплатную версию и использовать ее сколько захотите. Если этот редактор вам понравится и вы будете пользоваться им постоянно, рекомендую оплатить лицензию.

Чтобы установить Sublime, зайдите на http://www.sublimetext.com и щелкните на ссылке Download вверху страницы. Там вы найдете программы установки для всех основных платформ. Хотя (на момент написания книги) Sublime Text 3 находится на бета-тестировании, я рекомендую попробовать именно его. Я использовал его для всех примеров и скриншотов из этой книги.

Основы Sublime Text

После установки и запуска Sublime Text вы увидите экран примерно такого вида, как показано на рис. 1.3.

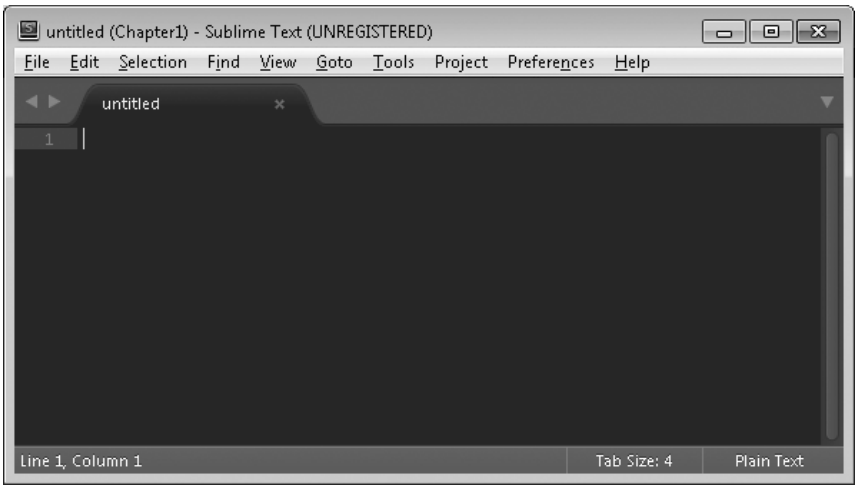


Рис. 1.3. Sublime Text, запущенный в первый раз

Вероятно, в первую очередь вы захотите создать новый файл. Это можно сделать с помощью меню File ▸ New. Вы можете также нажать Ctrl+N в Windows и Linux либо использовать Command+N в Mac OS. А сейчас наберите в редакторе `Hello, World!` Редактор будет выглядеть так, как показано на рис. 1.4.

Вы можете изменить внешний вид окружения Sublime, зайдя в меню Preferences ▸ Color Scheme. Попробуйте разные цветовые схемы и найдите ту, которая будет самой подходящей для ваших глаз. На самом деле будет полезно потратить какое-то время на исследование цветовых вариантов, поскольку вы будете проводить за работой с редактором много часов. Можете также изменить размер шрифта через подменю Font на вкладке Preferences, чтобы читать текст было еще удобнее.

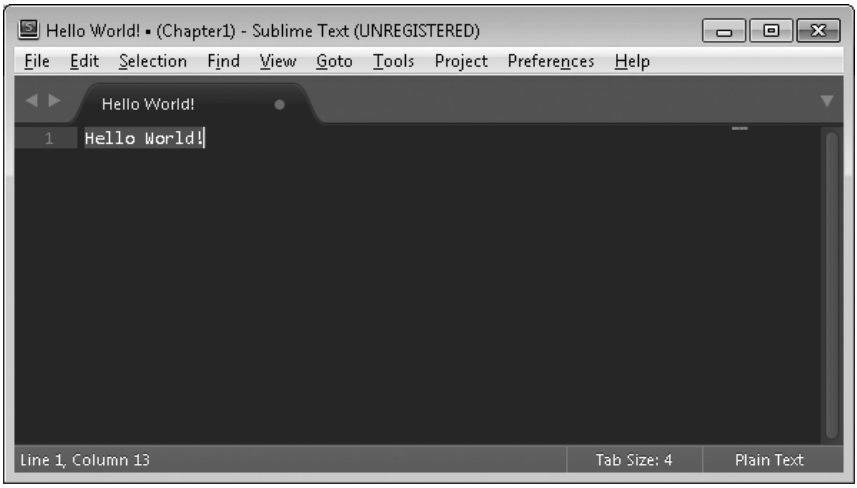


Рис. 1.4. Sublime после открытия нового файла и введения Hello, world!

Вы, наверное, заметили, что Sublime изменил название вкладки с untitled на Hello, world!, как вы набрали. Когда будете сохранять файл, в качестве имени по умолчанию будет предложено название вкладки, но вы, возможно, захотите изменить его так, чтобы он не содержал пробелов. После сохранения под другим именем название вкладки изменится на действительное имя файла. Отмечу, что после того как вы затем проделаете с содержимым файла что-то еще, вы увидите, что знак «×» справа от названия вкладки изменился на кружочек — это значит, что здесь есть несохраненные изменения.

После того как вы измените цветовую схему и сохраните файл под именем hello, редактор будет выглядеть так, как показано на рис. 1.5.

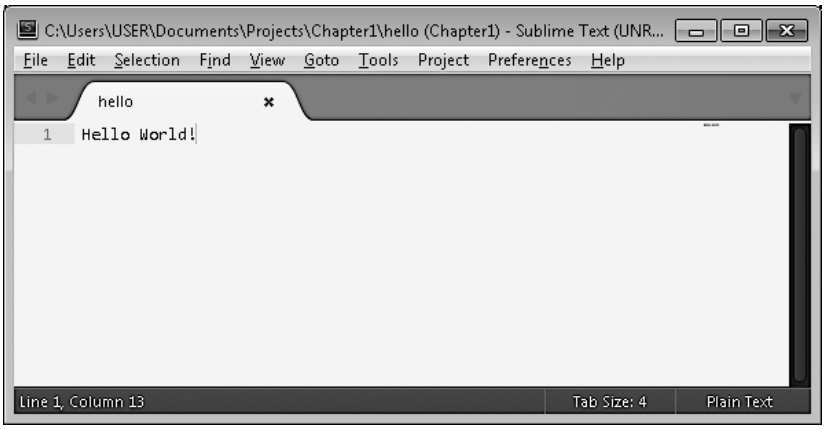


Рис. 1.5. Sublime после изменения цветовой схемы на Solarized (light) и сохранения файла под именем hello

Поскольку мы будем работать с командной строкой, лучше всего избегать пробелов и специальных символов в именах файлов. Можно сохранять файлы с использованием нижнего подчеркивания (_) вместо пробела, но постарайтесь не использовать никаких других нецифровых и неалфавитных символов.

Мы будем проводить много времени, редактируя код в Sublime, поэтому желательно иметь уверенность в том, что вы время от времени сохраняете свои изменения. Поскольку я ожидаю, что у вас есть небольшой опыт кодирования, думаю, вы уже знакомы с процессом edit — save — edit. Существует и необходимый смежный процесс, с которым начинающие программисты могли не сталкиваться. Он называется контролем версий.

Контроль версий

Представьте, что вы написали с помощью текстового редактора большой фрагмент художественной повести. Вы периодически сохраняете работу, чтобы случайно не

потерять ее. И вот однажды, достигнув важного момента в повествовании, вы понимаете, что значительной части истории главного героя не хватает. И решаете дополнить ее некоторыми деталями, вернувшись куда-нибудь в начало истории. Вы возвращаетесь к началу и понимаете, что существует два варианта развития событий для этого персонажа. Поскольку повествование еще не закончено, решаете написать черновики каждого из вариантов и посмотреть, к чему они приведут. Поэтому вы копируете файл в два различных места, один из вариантов называете «История А», другой — «История Б», а затем записываете в каждый из файлов различные варианты развития событий.

Можете мне не верить, но компьютерные программы создаются похожим образом гораздо чаще, чем романы. Фактически, как вы убедитесь, значительная часть времени тратится на что-то вроде *исследовательского кодирования*. Это значит, что вы разбираетесь, что нужно сделать, и пытаетесь заставить функциональность как-то работать, прежде чем действительно приступите к ее кодированию. Иногда фаза исследовательского кодирования может привести к необходимости внесения изменений, затрагивающих множество строк в различных кодовых файлах приложения. Даже начинающие программисты поймут это скорее рано, чем поздно и в конце концов придут к решению, подобному описанному. Например, начинающие могут скопировать рабочую папку с кодом в другую, немного изменить имя и продолжить работу. Если окажется, что допущена ошибка, они всегда могут вернуться к предыдущей копии.

Это весьма приблизительный подход к *контролю версий*. Контроль версий — это процесс, который позволяет вам создавать в коде помеченные контрольные точки, к которым вы всегда можете обратиться (или даже вернуться), если будет необходимо. Кроме того, контроль версий — незаменимый инструмент для взаимодействия с другими разработчиками. Мы не будем уделять данному аспекту много внимания в этой книге, но постарайтесь запомнить это.

Существует много профессиональных инструментов для контроля версий, и все они имеют собственный набор функций и особенностей. В качестве общих примеров можно привести Subversion, Mercurial, Perforce, CVS. В сообществе веб-разработчиков наибольшую популярность приобрела система контроля версий под названием Git.

Установка Git

У Git есть очень простые инсталляторы как для Windows, так и для Mac OS. Для Windows мы будем использовать проект msysgit, который доступен на GitHub (рис. 1.6). Инсталляторы доступны также на Google Code и связаны ссылками со страницей GitHub. Как только вы скачаете инсталлятор, щелкните на нем и следуйте инструкциям, чтобы установить Git в свою систему.

Для Mac OS я предпочитаю использовать инсталлятор Git for OS X Istaller (рис. 1.7). Вы просто скачиваете образ диска в архиве, монтируете его, а затем делаете двойной щелчок на инсталляторе. На момент написания книги инсталлятор сообщал, что версия предназначена для Mac OS Snow Leopard (10.5), но она вполне корректно работала и на моей системе Mountain Lion (10.8).

Рис. 1.6. Домашняя страница msysgit

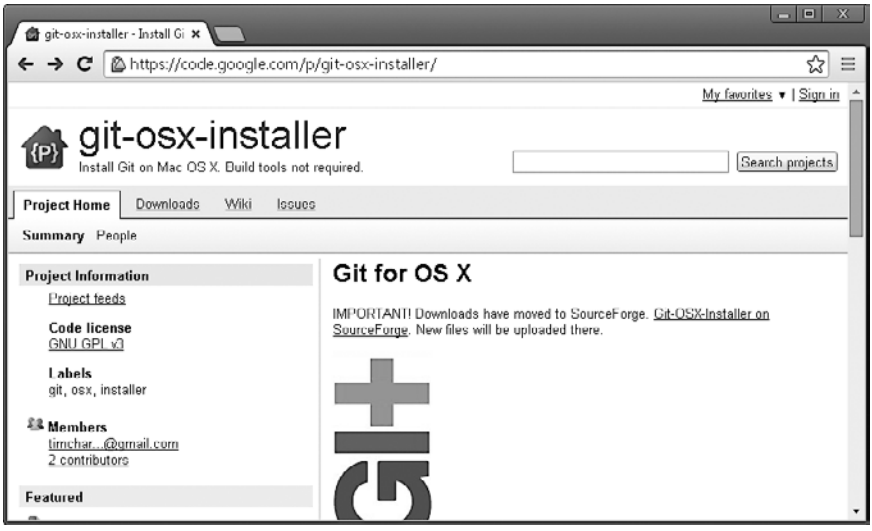


Рис. 1.7. Домашняя страница Git for OS X

Если вы пользуетесь Linux, то можете установить Git через систему управления пакетами.

Основы работы с командной строкой в UNIX

Для Git есть графические интерфейсы, но гораздо более эффективно научиться работать с ним с помощью командной строки. Перед тем как вы начнете учиться

этому, придется разобраться, как работать с файловой системой, используя некоторые основные команды UNIX.

Как я уже упоминал, я предполагаю, что вы немного знакомы с информатикой и программированием, но в основном взаимодействовали с окружением с графическим пользовательским интерфейсом. Это значит, что вы привыкли использовать настольное окружение для работы с файлами и папками, находящимися на вашем компьютере. Скорее всего, вы делаете это через менеджер файловой системы, например Finder в Mac OS или проводник в Windows.

Навигация в файловой системе с использованием командной строки очень похожа на аналогичный процес, осуществляемый с помощью системного файлового менеджера. Мы так же работаем с файлами, которые организованы в папки (каталоги). Вы можете выполнять те же самые операции, которые делаете с помощью файлового менеджера: перемещать файлы в каталоги или извлекать их оттуда, просматривать список файлов, находящихся в папке, и даже открывать и редактировать файлы, если знакомы с Emacs или Vim. Отличия состоят в том, что вы не получаете постоянной обратной связи с помощью графического пользовательского интерфейса и не можете взаимодействовать с мышью.

Если вы работаете в Windows, то будете использовать командную строку Git Bash, которую установили вместе с msysgit project, как описано в предыдущем разделе. Git Bash — это программа, симулирующая командную строку UNIX в Windows и дающая вам доступ к командам Git. Запустить командную строку Git Bash можно через меню Пуск. Если вы работаете с Mac OS, то будете использовать программу Terminal, которую можете найти в папке Utilities внутри каталога Applications. Если вы используете Linux, все может зависеть от конкретной конфигурации, которую вы предпочитаете, но, как правило, вполне успешно можно применять легкодоступную программу Terminal, которая находится в ваших приложениях. Терминальное окно по умолчанию в Mac OS показано на рис. 1.8.

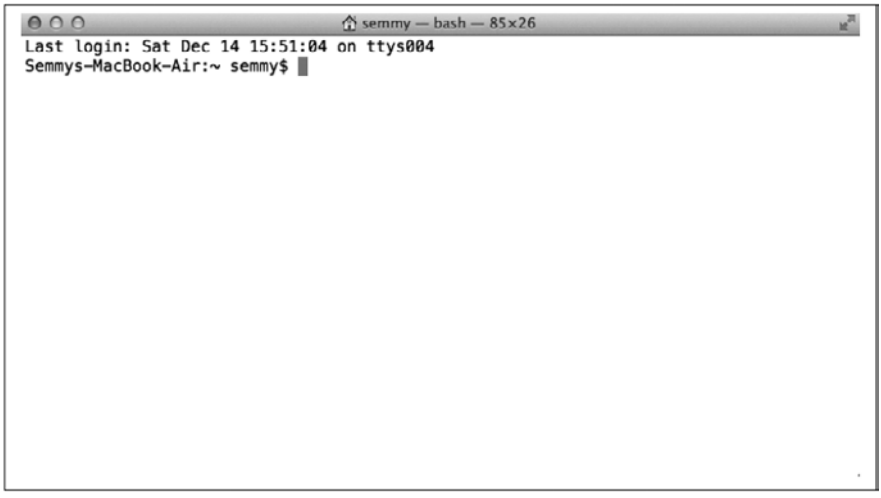


Рис. 1.8. Терминальное окно по умолчанию в Mac OS

Открыв командную строку, вы видите приветственное сообщение. Оно может выглядеть по-разному в зависимости от того, работаете вы в Mac OS или Windows, но, как правило, содержит какую-либо информацию о вашем рабочем окружении. В частности, оно может включать текущую папку или ваше пользовательское имя. Мое сообщение в Mac OS выглядит вот так:

```
Last login: Tue May 14 15:23:59 on ttys002
hostname $ _
```

Где я?

Очень важно помнить, что, работая в командной строке, вы всегда выполняете команды из какой-либо папки. Первый вопрос, который вы должны задать себе, попав в интерфейс командной строки: «В какой папке я нахожусь?» Получить ответ можно двумя способами. Первый — использовать команду `pwd`, что означает print working directory («вывести рабочую директорию» (то есть папку)). Вывод будет выглядеть примерно так:

```
hostname $ pwd
/Users/semmy
```

Хотя иногда я и использую `pwd`, но обычно предпочитаю команду `ls`, которую можно расшифровать как list the content of the current directory («перечисли содержимое текущей директории»). Это дает мне больше визуальных подсказок о том, где я нахожусь. В Mac OS вывод будет выглядеть примерно так:

```
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music
```

Таким образом, команда `ls` аналогична открытию папки в Finder или Проводника из вашего домашнего каталога. Результат выполнения команды показал мне, что я нахожусь в своем домашнем каталоге, поскольку вижу все подкаталоги, выведенные на экран. Если я не узнаю подкаталоги, хранящиеся в этой папке, то могу использовать `pwd` для получения дальнейшей информации.

Изменение папки

Следующее действие, которое вам понадобится, — это переход в следующую папку из той, где вы находитесь в настоящий момент. Если вы работаете с файловым менеджером с графическим пользовательским интерфейсом, переход осуществляется двойным щелчком на нужной папке.

Выполнить его из командной строки ничуть не сложнее: просто запомните название команды — `cd`, что значит change directory («изменить директорию»). Если, например, вы хотите перейти в вашу папку Documents, просто введите:

```
hostname $ cd Documents
```

Сейчас, если вы хотите получить визуальную обратную связь, можете использовать `ls`:

```
hostname $ ls
Projects
```

Мы видим один подкаталог в каталоге Documents, и этот подкаталог называется Projects (Проекты). Понятно, что если вы его предварительно не создали, то каталог Projects не отобразится. Вы также можете увидеть другие файлы или папки, созданные ранее в папке Documents. А сейчас, после перехода в новую папку, выполним команду pwd, чтобы уточнить наше текущее положение:

```
hostname $ pwd
/Users/semmy/Documents
```

Что произойдет, если вы хотите вернуться обратно в домашнюю папку? В файловом менеджере с графическим интерфейсом обычно есть кнопка Назад (Back), с помощью которой вы можете вернуться в предыдущую папку. В консоли, конечно, такой кнопки нет. Но вы можете снова использовать команду cd с небольшим изменением — двумя точками (. .) вместо названия каталога, чтобы вернуться в предыдущее место расположения:

```
hostname $ cd ..
hostname $ pwd
/Users/semmy
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music
```

Создание папки

И наконец, вам может понадобиться создать новую папку для хранения своих проектов, которые вы будете создавать с помощью этой книги. Для этого используйте команду mkdir, которая означает make directory («сделать директорию»):

```
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music
hostname $ mkdir Projects
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music Projects
hostname $ cd Projects
hostname $ ls
hostname $ pwd
/Users/semmy/Projects
```

При этом взаимодействии с командной строкой вы сначала смотрите содержимое текущей папки, чтобы удостовериться, где находитесь, с помощью команды ls. После этого используете команду mkdir для создания каталога Projects. Затем используете ls, чтобы убедиться, что каталог создан. После этого вводите cd, чтобы попасть в папку Projects, а потом — ls, чтобы вывести ее содержимое. Только что созданная папка, разумеется, пуста, поэтому в результате выполнения коман-

ды `ls` ничего выведено не будет. И наконец, последнее по очереди, но не по важности — вы используете `pwd`, чтобы убедиться, что находитесь именно в папке Projects.

Этих четырех основных команд UNIX вам будет вполне достаточно для начала, но вы изучите больше по мере нашего продвижения вперед. В конце главы я разместил удобную таблицу с перечнем и описанием этих команд. Будет очень хорошо, если вы постараетесь их запомнить.

Файловые системы и древовидные диаграммы

Веб-разработка (и программирование в целом) — очень абстрактная форма творчества. В целом это означает, что для эффективного и рационального труда вы должны развивать абстрактное мышление. Во многом эта способность требуется для быстрого создания ментальных моделей новых идей и структур. Одна из лучших ментальных моделей, которая применяется во множестве разных ситуаций, — древовидная диаграмма.

Древовидная диаграмма — это простой способ визуализации любого типа иерархической структуры. Поскольку файловая система в UNIX является иерархической структурой, будет естественно начать упражнения в ментальной визуализации с нее. Например, представьте себе, что папка Home (Домашняя) содержит три другие папки: Documents (Мои документы), Pictures (Мои рисунки) и Music (Моя музыка). Внутри папки Pictures находятся пять картинок. Внутри каталога Documents содержится другой каталог — Projects (Проекты) (рис. 1.9).

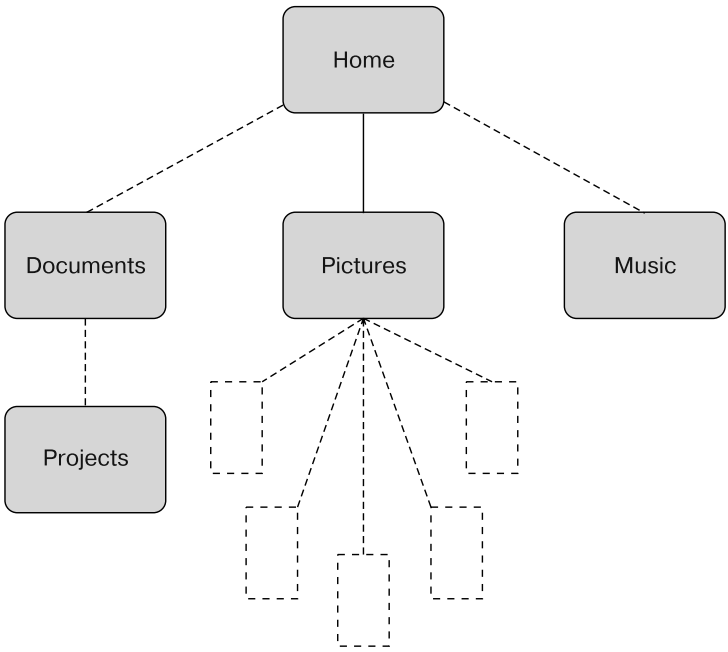


Рис. 1.9. Древовидная диаграмма, представляющая файловую иерархию

Постарайтесь держать эту ментальную модель в голове, перемещаясь по файловой системе. На самом деле я рекомендовал бы вам добавить звездочку (или аналогичный символ), которая означает текущую папку, и перемещать ее одновременно с движением по файловой системе.

Вообще говоря, если вы примените древовидную диаграмму к любой иерархической структуре, то, скорее всего, вам будет гораздо легче понять и анализировать ее. Поскольку способность стать хорошим программистом в значительной степени зависит от умения быстро строить ментальные модели, старайтесь практиковаться в применении древовидной диаграммы к иерархическим системам реального мира почаще — всегда, когда это имеет смысл. Мы вернемся к ним еще несколько раз в тексте книги.

Основы Git

Сейчас, когда мы умеем передвигаться по файловой системе с помощью командной строки, пора научиться управлять версиями нашего проекта с помощью Git.

Настройка Git при первом запуске

Как я уже говорил, Git был разработан в первую очередь для масштабного взаимодействия между большим количеством программистов. Хотя мы собираемся использовать его только для собственных проектов, нужно настроить его так, чтобы можно было отслеживать изменения с помощью некоторой дополнительной информации, например имени и адреса электронной почты. Откройте командную строку и наберите следующие команды (изменив, разумеется, мои имя и адрес на свои):

```
hostname $ git config --global user.name "Semmy Purewal"
hostname $ git config --global user.email semmy@semmy.me
```

Для нашей системы придется сделать это всего один раз! Другими словами, вам не нужно повторять эти действия каждый раз, когда вы хотите создать проект, используя Git.

А сейчас мы готовы начать отслеживание проекта с использованием Git. Начнем с перехода в наш каталог Projects, если еще не находимся там:

```
hostname $ pwd
/Users/semmy
hostname $ cd Projects
hostname $ pwd
/Users/semmy/Projects
```

Затем создадим каталог под названием Chapter1 (Глава 1)[1] и просмотрим содержимое, чтобы убедиться, что он тут. Затем перейдем в новую папку:

[1] При работе с Git возможны трудности с выводом кириллических символов в именах файлов и папок. Все сообщения, где есть русские символы, преобразуются в кодировку UTF-8 по latin1. Решение проблемы описано на http://habrahabr.ru/post/74839/, но на начальном этапе можно просто без крайней необходимости не использовать кириллические и другие специальные символы в названиях файлов и папок. — *Примеч. пер.*

```
hostname $ mkdir Chapter1
hostname $ ls
Chapter1
hostname $ cd Chapter1
hostname $ pwd
/Users/semmy/Projects/Chapter1
```

Создание хранилища Git

Сейчас мы можем установить контроль версий над папкой Chapter1, создав хранилище в Git с помощью команды `git init`. Git ответит нам, что создано новое пустое хранилище:

```
hostname $ pwd
/Users/semmy/Projects/Chapter1
hostname $ git init
Initialized empty Git repository in /Users/semmy/Projects/Chapter1/.git/[1]
```

А сейчас снова введите команду `ls`, чтобы увидеть файлы, которые Git создал в папке, и вы увидите, что там ничего нет! Это не совсем верно — здесь находится папка .git, но мы не можем ее увидеть, так как файлы, помеченные впереди точкой (.), считаются скрытыми. Чтобы решить эту проблему, можно использовать команду `ls` с включенным флагом `-a` (all — «все»), набрав следующее:

```
hostname $ ls -a
. .. .git
```

Таким образом выводится все содержимое папки, включая файлы, помеченные впереди точкой. Вы даже видите вывод своей текущей папки (это одинарная точка) и ее родительской папки (две точки).

Если вам интересно, вы можете вывести и содержимое каталога .git, увидев таким образом файловую систему, которую Git подготовил для вас:

```
hostname $ ls .git
HEAD config hooks objects
branches description info refs
```

Нам не придется ничего делать в этой папке, так что сейчас мы можем смело ее проигнорировать. Но со скрытыми файлами еще встретимся, так что постарайтесь запомнить полезный флаг `-a` для команды `ls`.

Определение статуса хранилища

Откроем Sublime Text (если он все еще открыт со времени чтения предыдущего раздела, закройте его и откройте снова). Затем откройте папку, версии в которой мы будем контролировать. Чтобы сделать это, просто выбираем папку в диалоговом окне Open программы Sublime вместо конкретного файла. Когда вы открываете

[1] Системное сообщение: «Создано пустое хранилище Git в /Users/semmy/Projects/Chapter1/.git». — *Примеч. пер.*

целую папку, файловая навигационная панель появляется с левой стороны окна редактора — это выглядит примерно как на рис. 1.10.

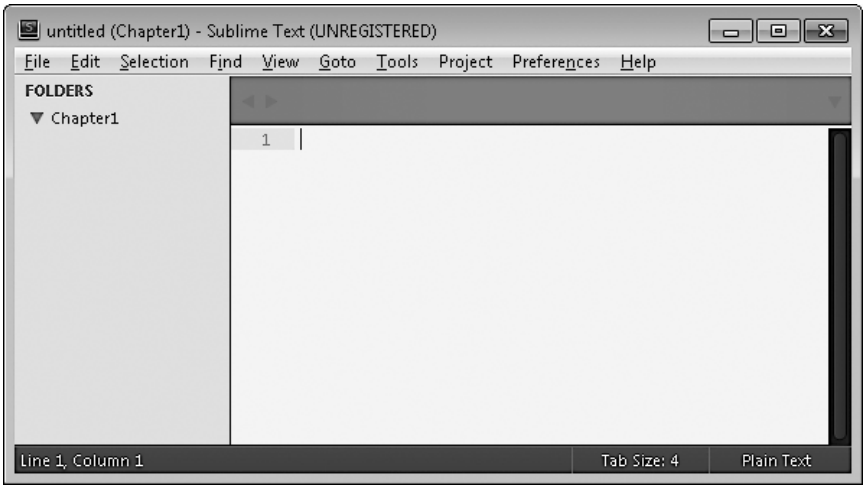


Рис. 1.10. Sublime с открытой папкой Chapter1

Чтобы создать новый файл в папке Chapter1, щелкните правой кнопкой (или выполните Command-щелчок в Mac OS) на названии Chapter1 в файловой навигационной панели, а затем выберите New File из контекстного меню. Как и ранее, будет создан новый файл, но, когда вы сохраните его, по умолчанию будет использована папка Chapter1. Назовем файл `index.html`.

После того как файл получил имя, сделайте на нем двойной щелчок и добавьте строку `Hello, World!` в верхнюю часть файла (рис. 1.11).

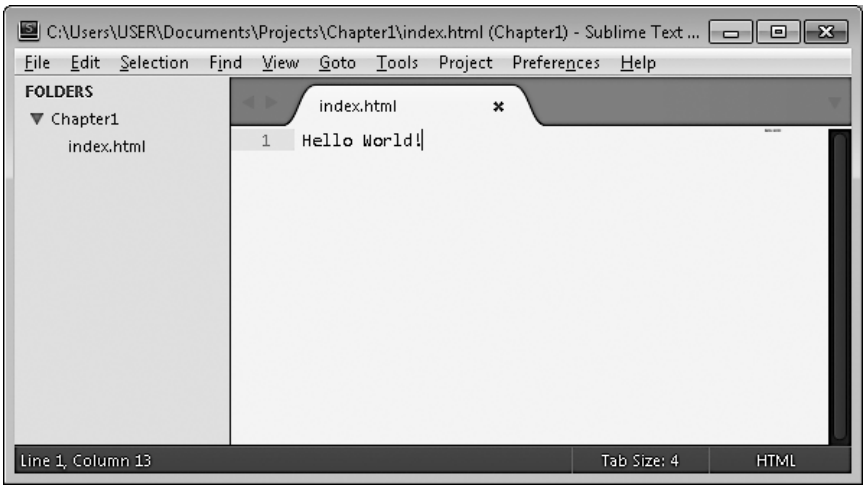


Рис. 1.11. Sublime после редактирования и сохранения файла index.html

Посмотрим, что произошло с хранилищем Git. Вернитесь к командной строке и убедитесь, что находитесь в нужном каталоге:

```
hostname $ pwd
/Users/semmy/Projects/Chapter1
hostname $ ls
index.html
```

А сейчас наберите `git status` и посмотрите на примерно такой ответ:

```
hostname $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# index.html[1]
```

Здесь довольно много информации! Больше всего нас интересует блок, помеченный `Untracked files` (Неотслеживаемые файлы). Здесь перечислены файлы, которые находятся в рабочей папке, но пока не включены в контроль версий.

Мы видим, что файл `index.html` здесь и он может быть зафиксирован (committed) в хранилище Git.

Наши первые коммиты[2]

Мы заинтересованы в отслеживании изменений в файле `index.html`. Чтобы иметь возможность делать это, последуем инструкциям, которые выдал Git, и добавим файл в хранилище, используя команду `git add`:

```
hostname $ git add index.html
```

Обратите внимание на то, что в этом случае мы не получаем никакого ответа. Так и должно быть. Можем убедиться, что команда сработала, набрав `git status` еще раз:

```
hostname $ git status
# On branch master
#
```

---

[1] Системное сообщение: «От бранч-мастера. Начальный коммит. Неотслеживаемые файлы (используйте "git add <file>..." для добавления их в список для фиксирования изменений): index.html». — *Примеч. пер.*

[2] Коммит (commit) — очередная версия отслеживаемого файла или нескольких файлов, отправленная в систему контроля версий. Само по себе изменение файла с кодом, например, коммитом не является, а становится таковым только после загрузки в систему контроля версий. Один или несколько коммитов могут составлять билд (сборку) — очередное обновление версии программы. Например, в новый билд могут входить три коммита: один с добавлением новой функциональности, другой — с изменением внешнего вида интерфейса, а третий — с исправлением дефекта. — *Примеч. пер.*

```
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: index.html[1]
#
```

Это и есть нужная нам обратная связь. Обратите внимание на то, что `index.html` теперь указан после заголовка `Changes to be commited` (Изменения, которые должны быть зафиксированы).

После добавления новых файлов в хранилище мы хотим зафиксировать его начальное состояние. Для этого используем команду `git commit` с флагом `-m` и смысловым сообщением о том, что изменилось со времени последнего коммита. Чаще всего начальный коммит будет выглядеть так:

```
hostname $ git commit -m "Initial commit"
[master (root-commit) 147deb5] Initial commit
1 file changed, 1 insertion(+)
create mode 100644 index.html[2]
```

Таким образом мы получаем мгновенный снимок состояния нашего проекта в определенный момент. Мы всегда можем откатить состояние системы к нему позднее, если решим, что где-то пошли неверным путем. Сейчас команда `git status` не покажет нам файла `index.html`, так как он уже отслеживается, а никаких изменений внесено не было. Если мы ничего не изменяли со времени последнего коммита, у нас сейчас, как говорят, актуальный рабочий каталог:

```
hostname $ git status
# On branch master
nothing to commit (working directory clean)[3]
```

Легко забыть включить флаг –m и сообщение для коммита в процессе заливки изменений в систему контроля версий. Если это произойдет, вы, скорее всего, обнаружите себя внутри текстового редактора Vim (который обычно по умолчанию является системным редактором). Если это так, вы можете покинуть его нажатием двоеточия (:), а затем ввести q! и нажать Enter, чтобы выйти.

Затем внесем в `index.html` небольшие изменения. Добавим вторую строку, которая говорит: Goodbye, World! («Прощай, мир!»). Сделайте это и сохраните файл,

[1] Системное сообщение: «От бранч-мастера. Начальный коммит. Изменения для коммита: (используйте "git rm --cached <file>..." для отката). Новый файл: index.html». — *Примеч. пер.*

[2] Системное сообщение: «[master (root-commit) 147deb5] Начальный коммит. 1 файл изменен, 1 добавлен (+). Создан модуль 100644 index.html». — *Примеч. пер.*

[3] Системное сообщение: «От бранч-мастера: нет изменений для коммита (рабочая директория актуальна)». — *Примеч. пер.*

использовав соответствующее клавиатурное сокращение. А сейчас посмотрим, как `git status` отреагирует на наши действия:

```
hostname $ git status
# On branch master
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified: index.html[1]
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Обратите внимание: Git предупреждает нас, что `index.html` был изменен, но не добавлен в следующий коммит. Чтобы добавить наши изменения в хранилище, мы должны выполнить команду `git add` для измененного файла, а затем — `git commit`, чтобы зафиксировать изменения в системе контроля версий. Мы можем убедиться в том, что добавление прошло успешно, набрав `git status` перед тем, как выполнять фиксацию изменений в системе контроля версий. Наши действия будут выглядеть примерно так:

```
hostname $ git add index.html
hostname $ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified: index.html
#
hostname $ git commit -m "Add second line to index.html"
[master 1c808e2] Add second line to index.html
1 file changed, 1 insertion(+)[2]
```

Как посмотреть историю в хранилище

Мы сделали целых два коммита проекта и можем откатить систему к их состояниям в любое время. В разделе «Больше теории и практики» далее я привожу ссылку, которая покажет вам, как откатить предыдущий коммит обратно и начать кодирование из этой точки. А сейчас изучим еще одну очень полезную команду. Мы можем посмотреть историю коммитов, используя `git log`:

---

[1] Системное сообщение: «От бранч-мастера: Изменения не зафиксированы в коммит. (используйте git add <file> для обновления списка файлов для коммита) (используйте "git checkout -- <file>..." для отмены изменений в рабочей директории) Изменен: index. html». — *Примеч. пер.*

[2] Системное сообщение: «От бранч-мастера: Изменения для коммита: (используйте "git rm --cached <file>..." для отката). Изменен: index.html. hostname $ git commit –m Добавлена вторая строка в index.html. 1 файл изменен, 1 добавление (+)». — *Примеч. пер.*

```
hostname $ git log
commit 1c808e2752d824d815929cb7c170a04267416c04
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:36:47 2013 -0400

Add second line to index.html

commit 147deb5dbb3c935525f351a1154b35cb5b2af824
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:35:43 2013 -0400
Initial commit
```

Очень полезно запомнить эти четыре команды Git, как и четыре команды UNIX, рассмотренные в предыдущем разделе. Удобная диаграмма в разделе «Подведем итоги» иллюстрирует эти команды.

Сохранение и коммиты

На всякий случай, чтобы избежать неясностей, я хотел бы подробно остановиться на разнице между сохранением файла в текстовом редакторе и фиксированием изменений в системе контроля версий. Когда вы сохраняете файл, по сути, вы переписываете его на диск компьютера. Это значит, что вы больше не получите доступа к старой версии файла, если только редактор не предоставляет возможности построения истории ревизий.

Фиксирование же в хранилище Git позволяет вам отслеживать все изменения, которые вы сделали с последнего раза, когда добавляли новую версию файла. Это значит, что вы всегда можете вернуться к предыдущей версии, если обнаружили, что сделали трудноисправимую ошибку в текущей версии файла.

В данный момент у вас, вероятно, сложилось впечатление, что Git сохраняет код как линейную последовательность коммитов. Сейчас это действительно так: мы изучили ту часть Git, которая позволяет создать такое хранилище, где один коммит следует строго за другим коммитом. Мы можем назвать первый коммит *родительским*, а второй — *дочерним*. Хранилище Git с четырьмя коммитами показано на рис. 1.12.

Следует отметить, однако, что коммит, по сути, — это серия инструкций по обновлению вашего проекта до следующей версии. Другими словами, Git на самом деле вовсе не сохраняет каждый раз содержимое каталога полностью, как если бы вы скопировали одну папку на компьютере в другую. Вместо этого он просто фиксирует, что именно должно быть изменено: например, коммит может хранить информацию вроде «добавь строку с текстом Goodbye, World!» вместо сохранения всего файла. Так что лучше представлять себе хранилище Git как серию инструкций. Вот почему мы пишем сообщения для коммитов в форме императива — вы можете представлять себе коммит как последовательность указаний для перевода проекта из одного состояния в другое.

Почему все это так важно? Дело в том, что хранилище Git может иметь гораздо более сложную структуру. Коммит может иметь более одного «ребенка» и фактически более одного «родителя». На рис. 1.13 показан пример более сложного хранилища Git, где иллюстрируются оба этих случая.

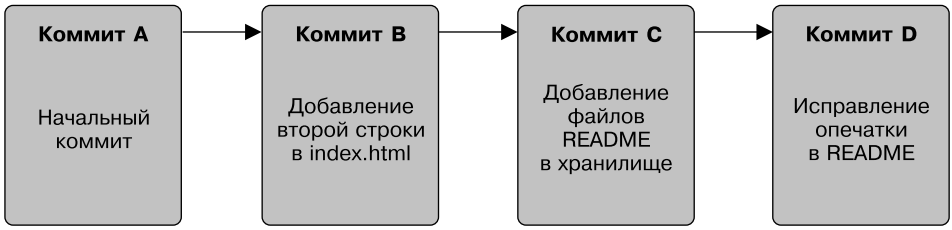


Рис. 1.12. Хранилище Git с четырьмя коммитами

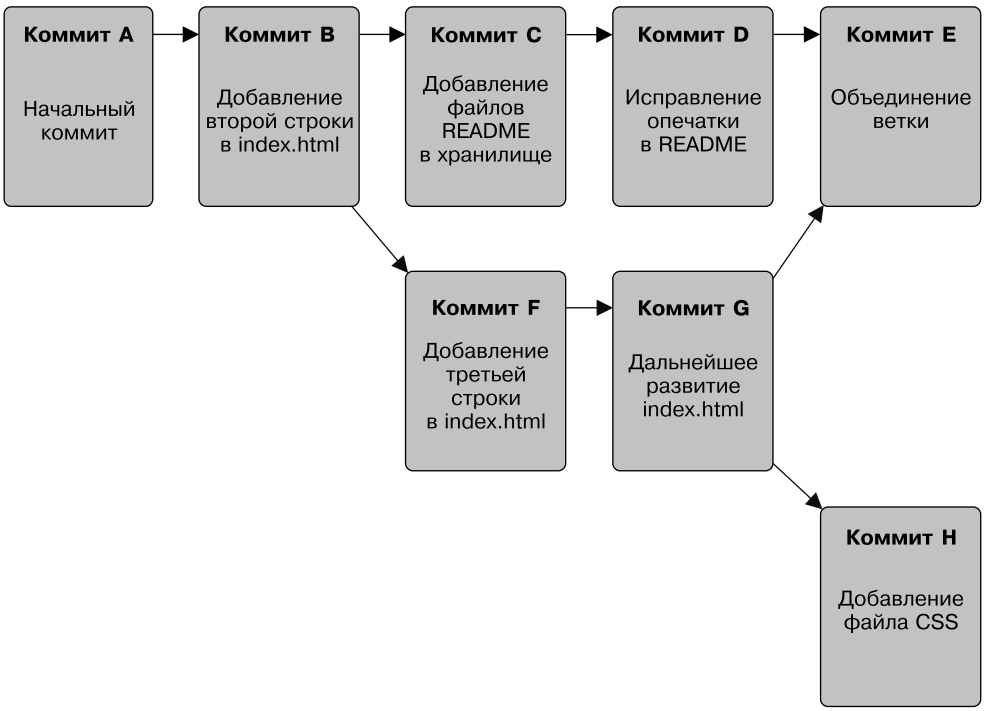


Рис. 1.13. Более сложное хранилище Git

Хотя мы еще не изучили команд Git для создания структуры наподобие этой, вам придется поближе познакомиться с ними, если вы захотите продолжить путешествие по миру веб-разработки. Я уверен, вы сделали вывод о необходимости составления визуальной схемы хранилища Git, чтобы не запутаться при возрастании сложности.

Браузеры

Последний инструмент, с которым мы будем регулярно сталкиваться, — браузер. Прежде чем изучать создание веб-приложений, которые будут запускаться в браузере,

следует узнать, как эффективно использовать его в качестве инструмента разработки, а не только как окно в Интернет.

Несколько лучших браузеров — Firefox, Safari и Chrome. Я бы рекомендовал вам приобретать опыт использования инструментов для разработки, доступных во всех трех. Но поначалу в целях простоты и синхронизации остановимся на Google Chrome.

Установка Chrome. Работаете ли вы в Windows, Mac OS или Linux, вы можете установить Google Chrome, просто зайдя на его веб-страницу. Процесс установки, конечно же, будет различаться, но инструкции очень просты. Как только вы установите Chrome и запустите его в первый раз, он будет выглядеть так, как показано на рис. 1.14.

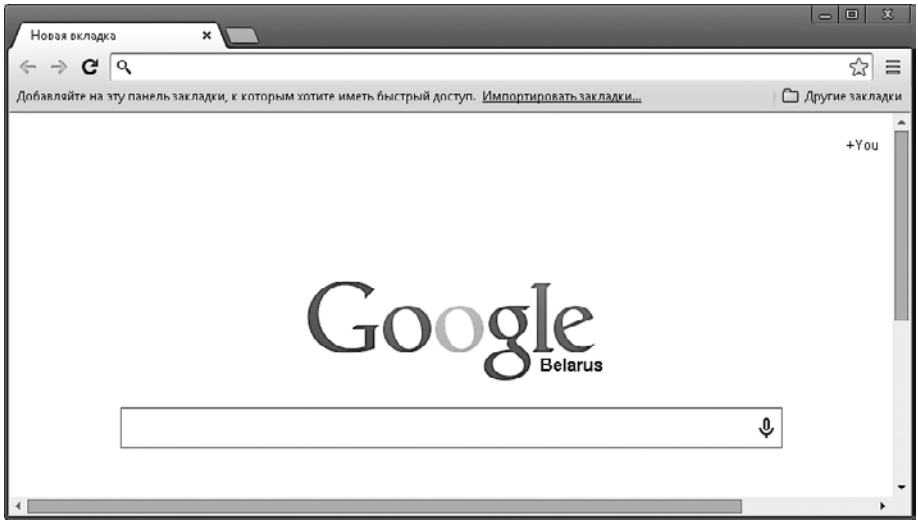


Рис. 1.14. Окно Chrome по умолчанию

Подведем итоги

Один из самых важных аспектов разработки веб-приложения — освоение рационального и эффективного рабочего процесса. Современный рабочий процесс включает в себя три важных инструмента: текстовый редактор, систему контроля версий и браузер. Sublime Text — популярный мультиплатформенный текстовый редактор, который отлично подходит для редактирования исходного кода. Git — широко используемая система контроля версий с управлением из командной строки. Chrome — прекрасный браузер для веб-разработки.

Прежде чем двигаться дальше, вы должны установить все эти инструменты на свой компьютер. Вы должны также запомнить команды из табл. 1.1 и 1.2, которые позволят вам передвигаться по файловой системе и взаимодействовать с Git из командной строки.

Таблица 1.1. Команды UNIX

Команда	Описание
pwd	Вывод на экран текущей папки
ls	Вывод на экран содержимого текущей папки
ls -a	Вывод на экран содержимого текущей папки, включая скрытые файлы
cd [dir]	Перейти в каталог [dir]
mkdir [dir]	Создать каталог [dir]

Таблица 1.2. Команды Git

Команда	Описание
git init	Инициализация хранилища
git status	Вывод на экран статуса хранилища
git add [file(s)]	Добавить файл(-ы) [file(s)] для фиксации в следующем коммите
git commit –m [msg]	Зафиксировать в системе контроля версий добавленные файлы с сообщением [msg — текст сообщения]
git log	Показать историю коммитов

Больше теории и практики

Заучивание

В сфере обучения слово «заучивание» обычно имеет негативную окраску. Я бы сказал, что это несправедливо, особенно в отношении компьютерного программирования. Если вы настраиваетесь так: «Ага, буду знать, где искать это, если понадобится», то проведете больше времени в поисках базовых инструментов, чем концентрируясь на интересных задачах. Представьте, например, насколько труднее были бы задачки на деление, если бы вы не помнили таблицу умножения наизусть!

Именно поэтому я собираюсь включить в конец первых нескольких глав раздел «Заучивание», который будет содержать основные понятия, которые вы должны хорошенько заучить, прежде чем двигаться дальше. Для этой главы все важное относится к командным строкам Git и UNIX. Просто повторяйте снова и снова следующие действия, пока не сможете выполнить их, не подглядывая в документацию.

1. Создайте новую папку с помощью командной строки.
2. Войдите в эту папку с помощью командной строки.
3. Создайте текстовый файл в текстовом редакторе и сохраните его как `index.html` в новой папке.
4. Инициализируйте хранилище Git из командной строки.
5. Добавьте и зафиксируйте этот файл в хранилище с помощью командной строки.

Каков же лучший способ запомнить эту последовательность задач? Все очень просто: делайте их снова и снова. На этих действиях основано много материала

следующих нескольких глав, поэтому очень важно хорошенько выучить их сейчас.

Sublime Text

Как я уже упоминал ранее, вы будете проводить много времени, работая с текстовым редактором, так что весьма неплохо будет как следует изучить его. На веб-сайте Sublime есть отличная страница поддержки, где находятся ссылки на документацию и видео, демонстрирующие продвинутые возможности редактора. Я рекомендую вам исследовать эту страницу и немного улучшить свои навыки работы с Sublime.

Emacs и Vim

Почти каждый веб-разработчик хоть раз да столкнется с необходимостью редактировать файл прямо на удаленном сервере. Это значит, что вы должны уметь использовать текстовый редактор, в котором отсутствует графический пользовательский интерфейс. Emacs и Vim — невероятно мощные редакторы, поднимающие эффективность процесса разработки до невиданных высот, но обучение работе с ними может быть крутовато для новичков. Если вы найдете время, постарайтесь изучить основы работы в обоих редакторах, хотя, кажется, популярность Vim у веб-разработчиков в последнее время возрастает (ладно, признаюсь: я пользуюсь Emacs).

На домашней странице GNU есть отличнейший обзор Emacs, включающий учебный материал для новичков. Издательство O'Reilly выпустило несколько книг по Emacs и Vim, включая *Learning the vi and Vim Editors* («Изучение vi и редакторов Vim») авторов Арнольда Роббинса, Эльберта Ханная и Линды Лэмб, а также *Learning GNU Emacs* («Изучение GNU Emacs») авторов Дебры Камерон, Джеймса Элиотта, Марка Лойя, Эрика Рэймонда и Билла Розенблатта.

Вам будет очень полезно изучить, как выполняются в каждом из редакторов следующие действия:

- открытие редактора и выход из него;
- открытие, редактирование и сохранение существующего файла;
- открытие нескольких файлов одновременно;
- создание нового файла и его сохранение;
- поиск файла по заданному слову или фразе;
- вырезание фрагментов текста из одного файла и вставка в другой.

Если вы уделите этому время, то постепенно определитесь, с каким редактором хотите продолжать работу и обучение.

Командная строка UNIX

Чтобы стать мастером работы с командной строкой UNIX, понадобятся долгие годы, но вы уже узнали достаточно, чтобы начать. По моему опыту, эффективным

становится изучение нового по мере надобности, при решении специальных задач, но существует еще несколько базовых команд, которые я применяю регулярно. Используя поиск в Google, изучите материал о следующих общих командах: `cp`, `mv`, `rm`, `rmdir`, `cat` и `less`. Все они пригодятся вам в разное время.

Узнайте больше о Git

Git — невероятно мощный инструмент. Мы только слегка коснулись его возможностей. К счастью, Скотт Чейкон написал *Pro Git* (Apress, 2009) — прекрасную книгу, где подробно рассматривается множество аспектов Git. В первых двух главах описывается несколько функций, которые помогут вам более эффективно продвигаться в изучении материала этой книги, включая откат к версиям, предварительно зафиксированным в хранилище.

В третьей главе книги Чейкона детально описана концепция бранчей. Бранчи несколько выходят за рамки моей книги, но я упоминал о них ранее. Рекомендую как следует изучить эту тему, так как возможность быстро и просто разветвить хранилище — одна из лучших функций Git.

GitHub

GitHub — это онлайн-сервис, в котором находится ваше хранилище Git. Если вы храните код открыто (open source), оно бесплатно. Если же хотите создать защищенное хранилище Git, самый дешевый тарифный план — $7 в месяц. Я рекомендую начать с бесплатного плана и исследовать хранилище Git на GitHub.

Страница помощи GitHub поможет вам настроить аккаунт GitHub и связать его с вашим хранилищем Git. Там также находятся тонны полезной информации и о Git, и о GitHub. Пользуйтесь ею, чтобы начать действовать.

2 Структура

В следующих двух главах мы рассмотрим две важные для разработки клиентской стороны темы: HTML и CSS. Поскольку рассмотреть их подробно в данной книге вряд ли возможно, эти главы представляют собой скорее сборник конспектов, которые позволят вам изучить HTML и CSS в достаточной степени для того, чтобы работать с примерами кода из книги. Раздел «Больше теории и практики» данной главы поможет вам найти информацию для дальнейшего изучения.

Если вы уже знакомы с HTML и CSS, вполне возможно, свободно можете перейти к главе 4, которая начинается с изучения JavaScript на клиентской стороне. Вы можете просмотреть главы и прочитать краткое содержание в конце, прежде чем решить.

Привет, HTML!

HTML, что значит Hyper Text Markup Language (язык гипертекстовой разметки), — это технология, которая позволяет задавать структуру расположения визуальных элементов (иногда ее называют пользовательским *интерфейсом*) веб-приложения. Что я подразумеваю под словом «структура»? Рассмотрим простой пример.

Для начала с помощью командной строки создадим папку Chapter2 (Глава 2) в папке Projects. Напомню, что для этого следует использовать команду `mkdir`. Затем откроем эту папку в Sublime Text с помощью меню File или клавиш быстрого доступа. Создайте новый файл под названием `hello.html` внутри этой папки. Наберите его содержимое точно так же, как показано далее:

```
<!doctype html>
<html>
<head>
<title>Мое первое веб-приложение</title>
</head>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Теги и содержание

Во время набора вы, наверное, заметили, что в документе содержатся два типа элементов. Один из них — обычный текст вроде «Мое первое веб-приложение» или традиционное сообщение «Hello, World!». Другие элементы, например `<html>` или `<head>`, заключенные в угловые скобки, называются *тегами*. Теги являются формой *метаданных*, которые используются для применения нашей структуры к содержимому страницы.

Запустите браузер и откройте в нем созданный файл, используя пункт Open File (Открыть файл) в меню File (Файл). Вы увидите нечто похожее на рис. 2.1.

Рис. 2.1. Файл hello.html, открытый в Chrome

Очень полезно постараться выучить клавиатурные сокращения, так как это поможет работать быстрее и рациональнее. Для открытия файла в Chrome вам нужна команда Command+O, если вы работаете в Mac. В Windows эта же команда — Ctrl+O.

Как видите, теги не отображаются на странице, в отличие от остального текста. Заголовок «Мое первое веб-приложение» отображается как название вкладки, а содержимое «Hello, World!» появляется в основном пространстве окна.

Тег <p>: абзацы

А сейчас внесем небольшие изменения, добавив абзац текста *Lorem ipsum* — это простой текстовый наполнитель, который мы можем заменить нужным нам

текстом позднее. Вы можете скопировать и вставить его с вики-страницы Lorem ipsum[1]:

```
<!doctype html>
<html>
  <head>
    <title>Мое первое веб-приложение</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
     <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis
aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.</p>
  </body>
</html>
```

Сделав изменения в документе, мы должны их сохранить. Сейчас мы можем вернуться в браузер и переоткрыть страницу, щелкнув на круглой стрелке рядом с адресной строкой в Chrome. Вы увидите, что основное пространство браузера обновилось (рис. 2.2).

Рис. 2.2. Модифицированный файл hello.html, открытый в Chrome

[1] В Интернете много сервисов для генераторов текста lorem ipsum, в том числе на русском языке. Используя кириллицу, при сохранении файла выставляйте кодировку UTF-8, иначе символы могут отображаться некорректно. — *Примеч. пер.*

Вы можете обновлять страницы, используя сочетания клавиш Ctrl+R в Windows или Command+R в Mac OS.

Так и будет выглядеть обычный рабочий процесс при редактировании веб-страниц. Мы открываем файл в текстовом редакторе, вносим небольшие изменения, а затем обновляем браузер, чтобы увидеть их.

Комментарии

Комментарии — очень удобный способ делать пометки в HTML. Мы начинаем комментарий с <!-- и заканчиваем его -->. Вот простой пример, созданный на основе материала предыдущего раздела:

```
<!doctype html>
<html>
  <head>
    <title>Пример комментария</title>
  </head>
  <body>
    <!-- Это главный заголовок -->
    <h1>Hello, World!</h1>
    <!-- Это главный абзац текста -->
     <p>Я — главный абзац, скорее всего, я как-то связан с тегом h1, ведь я так
близко к нему расположен!</p>
  </body>
</html>
```

Поскольку компьютерные программы пишут в расчете на то, что их будут читать люди, всегда полезно делать пометки в сложном коде. Вы будете встречать комментарии в HTML на протяжении этой книги и, скорее всего, столкнетесь с комментированным HTML в Интернете.

Заголовки, ссылки и списки... ох!

Мы познакомились с некоторыми примерами основных тегов и комментариев, что же еще мы можем включить в разметку?

Сначала можем расширить тег <h1>, создав теги <h1>, <h2>, <h3>, <h4>, <h5> и <h6>. Они представляют собой различные уровни заголовков и обычно сохраняются для важных элементов содержимого страницы. *Самый* важный заголовок должен находиться в тегах <h1>, а менее значительные пусть появляются на нижних уровнях:

```
<!doctype html>
<html>
  <head>
    <title>Примеры тегов заголовков</title>
```

```
  </head>
  <body>
    <!-- Это самый главный заголовок -->
    <h1>Самое важное!</h1>
    <!-- Это сообщение, которое должно восприниматься как
     очень важное
    -->
    <p>Важный абзац</p>
    <h2>Менее важный заголовок</h2>
    <p>А это какое-то менее важное сообщение</p>
  </body>
</html>
```

Другой очень важный тег в HTML-документе — `<a>`, что означает anchor («якорь») и используется для создания ссылок. Теги ссылок — уникальная характеристика *гипертекста*, так как они могут связывать информацию на текущей странице и другой веб-странице. Чтобы использовать теги ссылок, нужно включить в HTML *атрибут* `href`, который скажет браузеру, что именно нужно открыть в случае щелчка на ссылке. Атрибут `href` помещается внутрь открывающего тега:

```
<!doctype html>
<html>
  <head>
    <title>Примеры ссылок</title>
  </head>
  <body>
    <!--
     Атрибут href указывает, куда идти при щелчке на ссылке
     -->
    <p>Это <a href="http://www.google.com">ссылка</a> на Google!</p>
    <p>
      <a href="http://www.example.com">
        Эта ссылка немного длиннее
      </a>
    </p>
    <p>
      А это ссылка на
      <a href="http://www.facebook.com">www.facebook.com</a>
    </p>
  </body>
</html>
```

Когда мы откроем веб-страницу в браузере, то увидим нечто похожее на рис. 2.3.

Подчеркнутый текст означает, что на нем можно щелкнуть и перейти таким образом на страницу, указанную в атрибуте `href`.

В этом примере есть только одна проблема — использование элементов абзаца для содержания, которое лучше выглядело бы в виде списка. Нет ли специальных тегов для создания списка? Есть, и не один, а целых два! Теги `<ol>` и `<ul>` служат для создания *нумерованных (ordered)* и *маркированных (unordered)* списков:

Рис. 2.3. Страница со ссылками с использованием тегов <a>

```
<!doctype html>
<html>
  <head>
    <title>Примеры списков</title>
  </head>
  <body>
    <h1>Примеры списков!</h1>
    <!-- Мы заключаем ссылки внутри тегов ul -->
    <ul>
      <li>
        Это <a href="http://www.google.com">ссылка</a> на Google!
      </li>
      <li>
        <a href="http://www.example.com">
          Эта ссылка немного длиннее
        </a>
      </li>
      <li>
        А это ссылка на
        <a href="http://www.facebook.com">
          www.facebook.com
        </a>
      </li>
    </ul>
    <!-- Мы можем также создать нумерованный список -->
    <h3>Как сделать нумерованный список</h3>
    <ol>
      <li>Начните с открывающего тега ol</li>
```

```
      <li>Затем добавьте несколько элементов в тегах li</li>
      <li>Добавьте закрывающий тег ol</li>
    </ol>
  </body>
</html>
```

А затем обновите страницу в браузере — и вы увидите нечто похожее на рис. 2.4.

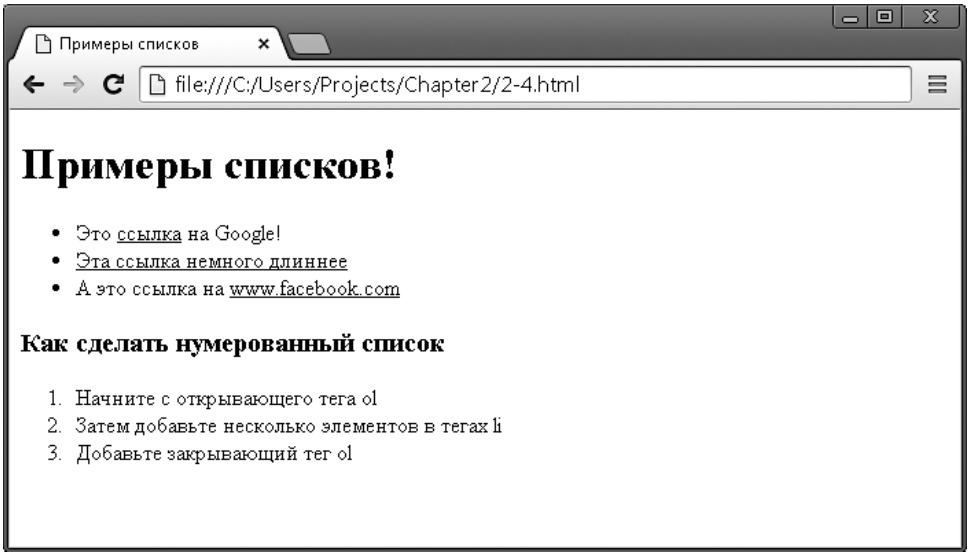


Рис. 2.4. Страница с маркированным и нумерованным списками

Подведем итоги

Подытожим информацию, полученную из первых увиденных вами примеров. Во-первых, все текстовое содержание помещается в HTML-теги.

Во-вторых, вы, наверное, заметили, что теги, заключенные в другие теги, смещены вправо с помощью отступов. Это делается потому, что HTML — иерархический метод структурирования документов. Мы используем отступы как визуальную пометку, напоминающую, что мы находимся на какой-либо ступени этой иерархии. Это означает, что теги `<head>` и `<body>` заключены внутрь тега `<html>`, а теги `<h1>` и `<p>`, в свою очередь, находятся внутри тега `<body>`. Случайным образом иногда мы помещали ссылки на той же строке, что и их содержимое, а в других случаях — разбивали строку. В HTML в большинстве случаев пробел не имеет значения.

И последнее по очереди, но не по важности: вы увидите, что в процессе создания HTML-документа мы будем понемногу добавлять или изменять содержимое страницы, сохранять файл, а затем переходить к окну браузера и обновлять страницу. Поскольку вы будете делать это очень часто, неплохо было бы попрактиковаться

несколько раз. Для начала добавьте несколько абзацев текста Lorem ipsum в тело документа, а затем перейдите в браузер и перезагрузите страницу.

Вы будете проделывать эту операцию довольно часто, поэтому очень полезно выучить клавиатурные сокращения для обновления страницы и переключения между активными окнами в рабочей среде. В Windows и большинстве оболочек Linux вы можете использовать Ctrl+Tab для переключения между активными окнами, а также Ctrl+R для обновления страницы. В Mac OS используйте Command+Tab и Command+R.

Объектная модель документа и древовидная модель

Теги HTML образуют иерархическую структуру, называемую *объектной моделью документа* (Document Object Model (DOM)). DOM — это способ представления объектов, которые определяются через HTML, а затем взаимодействуют интерактивно с помощью сценарного языка, например JavaScript. Теги HTML определяют *элементы* DOM — сущности, находящиеся в модели.

Мы уже написали HTML способом, позволяющим визуализировать структуру DOM. Вот почему мы отделяли отступами теги, находящиеся внутри других тегов, — это создает ощущение иерархии. Но хотя это и полезно для кода, не всегда работает так ясно, как бы нам хотелось. Посмотрите вот на такой HTML:

```
<!doctype html>
<html>
  <head>
    <title>Привет!</title>
  </head>
  <body>
    <h1>Привет!</h1>
    <div>
      <ol>
        <li>Элемент списка</li>
        <li> Элемент списка </li>
        <li> Элемент списка </li>
      </ol>
      <p>Это абзац</p>
      <p>Это <span>второй</span> абзац.</p>
    </div>
    <ul>
      <li>Элемент списка<span>1</span></li>
      <li>Элемент списка<span>2</span></li>
      <li>Элемент списка<span>3</span></li>
    </ul>
  </body>
</html>
```

В этом коде есть несколько тегов, с которыми вы еще не сталкивались, но пусть это вас не беспокоит: понимать их назначение пока не обязательно. Важно, чтобы вы обратили внимание: хотя этот код правильно разделен отступами, некоторые теги все равно не выделены в отдельные строки. Например, элементы `span` находятся на той же строке, что и элементы `li`. На самом деле это правильно, так как тег `<span>` содержит всего один символ, но эта структура не иллюстрирует взаимоотношения между элементами так же хорошо, как отступы. Следовательно, нужен другой способ визуализации этого примера.

В предыдущей главе мы обсуждали использование древовидной диаграммы для создания ментальной модели файловой системы компьютера. Что ж, нам ничего не мешает использовать ментальные модели для DOM! Эта ментальная модель очень пригодится позднее, когда мы будем взаимодействовать с DOM с помощью JavaScript. В качестве примера попробуем проиллюстрировать посредством древовидной диаграммы приведенный ранее код (рис. 2.5).

Эта диаграмма создает ясное представление о содержимом DOM. Кроме того, она упрощает представление некоторых взаимоотношений: мы рассматриваем элементы DOM, находящиеся в нижней части дерева, как потомки тех элементов, что находятся выше, если существует путь, связывающий их. Непосредственные потомки называются дочерними элементами, а элементы, находящиеся над дочерними, являются по отношению к последним *родительскими элементами*.

В этом примере все элементы являются потомками элемента `html`, а элемент `ul` является потомком элемента `body`. Элемент `ul` не является потомком элемента `head`, так как не существует пути, начинающегося от элемента `head` и заканчивающегося у элемента `ul` без продвижения вверх по иерархии. Элемент `ul` имеет три дочерних элемента (это пункты списка — элементы `li`), а каждый элемент `li` имеет дочерний `span`.

Мы изучим эти отношения подробнее по мере продвижения вперед, а пока постарайтесь потренироваться в восприятии DOM именно таким образом.

Использование валидации HTML для выявления проблем

Как я уже сказал в предыдущем разделе, текстовое содержание HTML-документа обычно заключено в пару тегов. Открывающий тег выглядит как `<html>`, а закрывающий — как `</html>`. Точное имя тега означает тип элемента DOM, который он собой представляет.

Если ваш документ становится слишком длинным, могут возникнуть некоторые проблемы. Например, представьте следующий HTML-документ, являющийся определенным обобщением предыдущего примера, с несколькими новыми тегами:

```
<!doctype html>
<html>
<head>
```

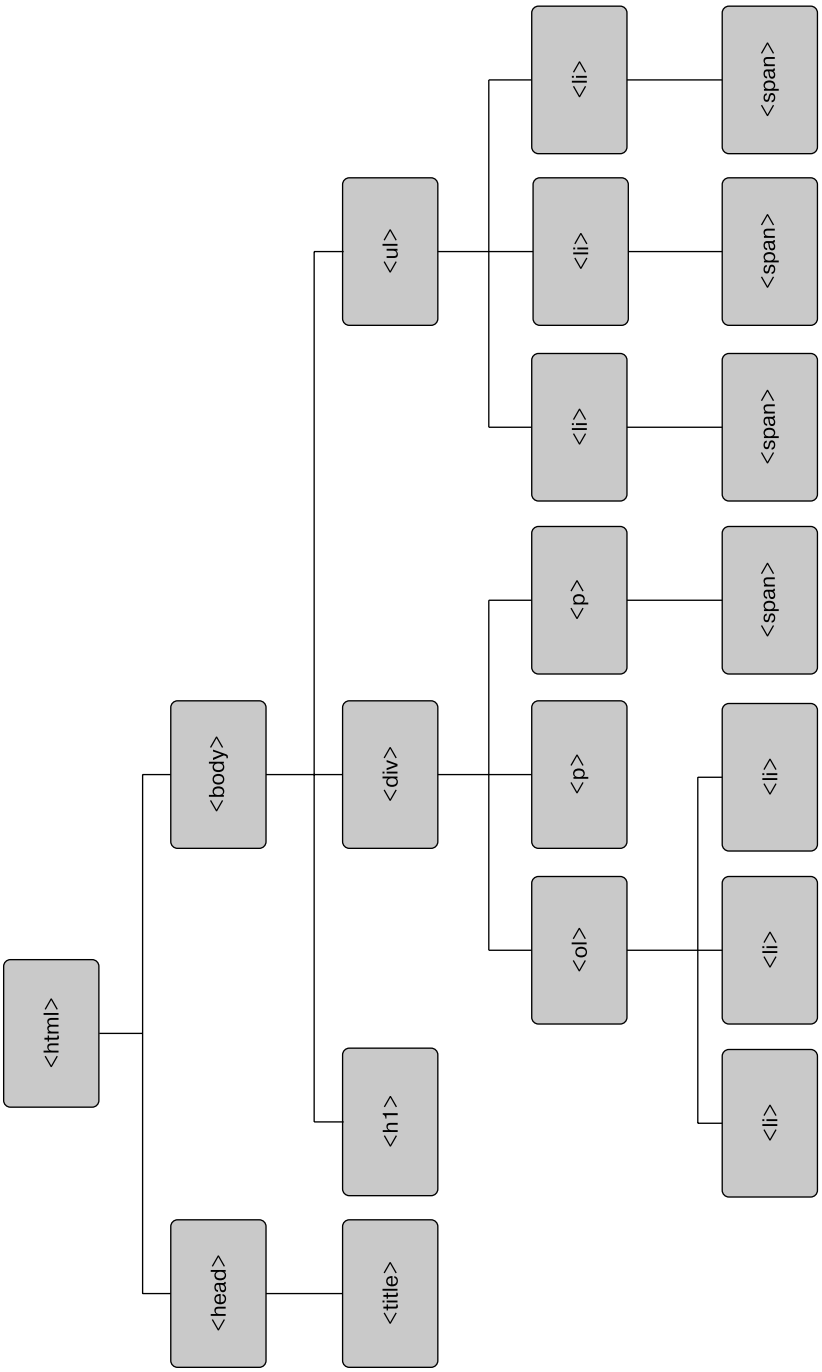


Рис. 2.5. Дерево, представляющее схему DOM

```
<title>Мое первое веб-приложение</title>
</head>
<body>
<h1>Привет!</h1>
<nav>
<div>Войти</div>
<div>ВиО</div>
<div>О нас</div>
</nav>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmodtempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, <span>quis
nostrud exercitation</span> ullamco laboris nisi ut aliquip ex ea commodo
consequat.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim <span>ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis
aute irure dolor in reprehenderit in voluptate <span>velit esse cillum dolore eu
fugiat</span> nulla pariatur.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.</p>
</body>
</html>
```

Этот HTML-документ содержит ошибку, но если вы откроете его в браузере, то не заметите ее. Попробуйте понять, где она, прежде чем читать дальше.

Нашли? Поздравляю — вы просто орлиный глаз! Если не нашли, ничего страшного. Дело в том, что во втором предложении есть открывающий тег `<span>`, но нет закрывающего. Большинство людей проводят много времени, пытаясь обнаружить такие ошибки в начале работы. К счастью, существует автоматизированный способ поиска ошибок в HTML-документах.

Программа *валидации* — это программа, которая автоматически проверяет, соответствует ли код определенным базовым стандартам. Если вы изучали когда-то язык программирования вроде Java или C++, вам, вероятно, приходилось работать с компилятором. Если в коде есть ошибки, компилятор сообщит о них, пропустив код через обработку. Языки вроде HTML немного «глупее» в том смысле, что браузер позволит вам допустить некоторое количество ошибок, но валидационная программа обнаружит то, что он пропустил.

Но почему мы вообще должны думать об ошибках, если браузер отображает страницу абсолютно одинаково, есть закрывающий тег `<span>` или его нет? Дело в том, что единственный путь гарантировать, что страница будет *всегда* выглядеть корректно в любом браузере, — соблюдение правил HTML. Вот почему HTML-валидатор является очень полезным инструментом.

Нам не нужно устанавливать никакого программного обеспечения для использования валидатора. Начнем с визита на домашнюю страницу W3C's Markup Validation Service validator.w3.org. На момент написания книги она выглядит так, как показано на рис. 2.6.

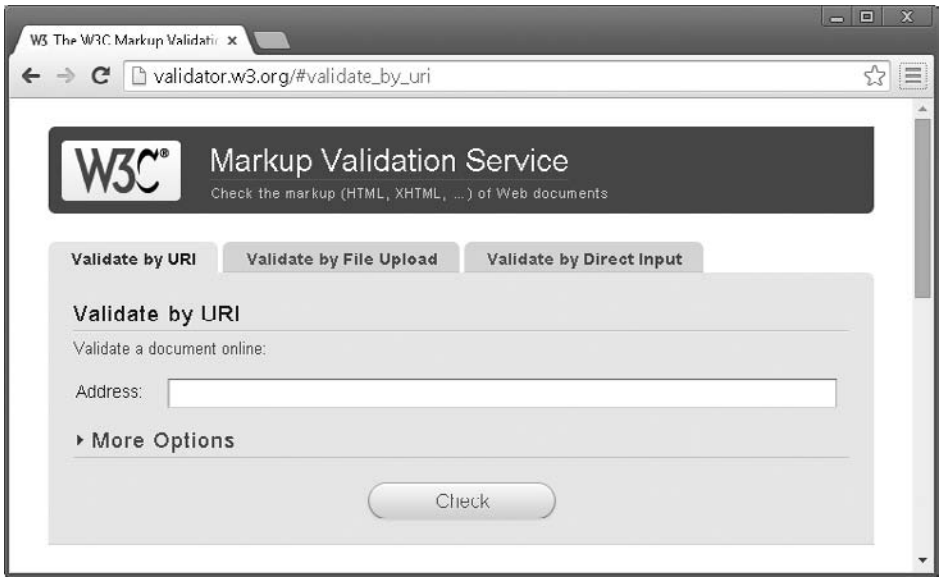


Рис. 2.6. Домашняя страница W3C's Markup Validation Service

Вы, наверное, заметили вкладку Validate by Direct Input. Щелкнув на ней, мы можем вырезать и вставить наш HTML-код в появившееся текстовое поле. После вставки кода нужно нажать большую кнопку Check.

Начнем с примера с Lorem ipsum, приведенного ранее. Если он не содержит никаких ошибок, мы увидим что-то похожее на рис. 2.7.

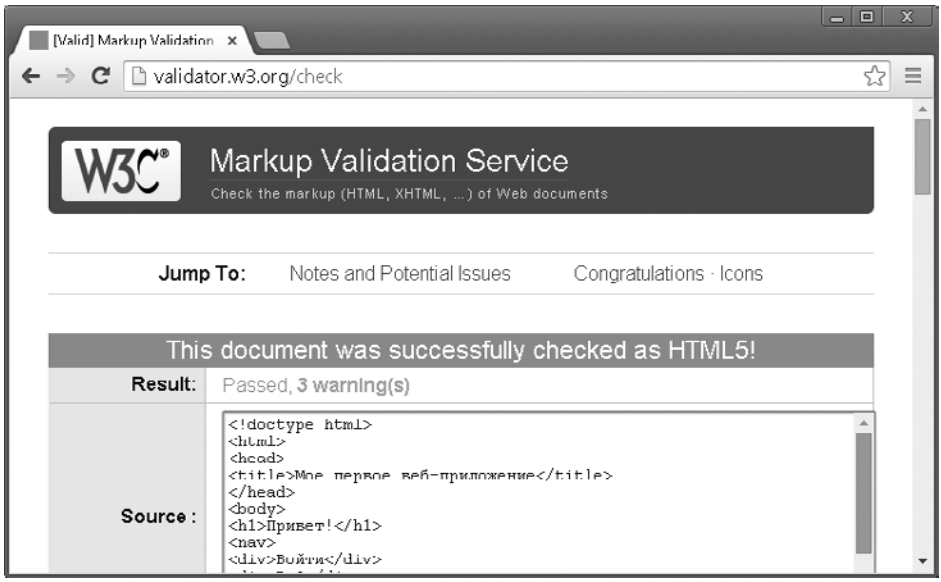


Рис. 2.7. Валидатор W3C HTML после проверки примера с Lorem ipsum

Валидируя HTML, вы, скорее всего, получите три предупреждения, даже если ошибок нет. Первое из них сообщает, что валидатор использует правила для проверки HTML5. Несмотря на то что стандарт HTML5 относительно стабилен, в нем все же могут быть изменения, о чем и уведомляет это сообщение.

Другие два относятся к кодировке символов и в данный момент могут быть проигнорированы. Если вам интересно, одно из предупреждений содержит ссылку на обучающий материал по кодировке символов, который научит вас изменять кодировку HTML-документа.

Если код не проходит валидацию, то мы увидим список найденных ошибок. Например, если запустим на проверку код с пропущенным закрывающим тегом элемента `span` во втором абзаце, то увидим нечто похожее на рис. 2.8.

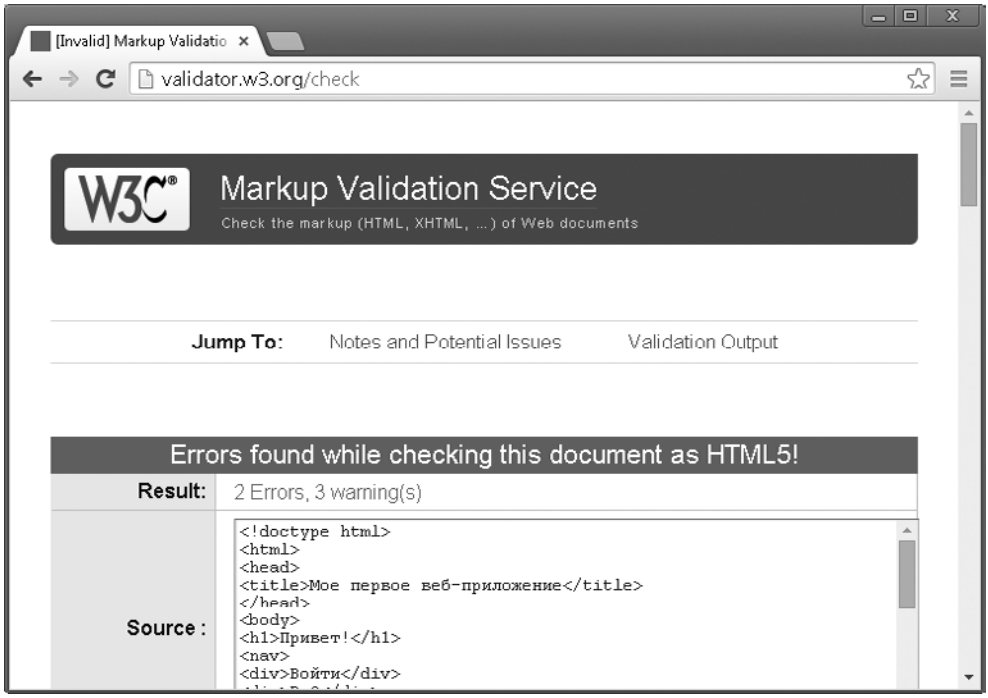


Рис. 2.8. Валидатор W3C HTML после проверки примера с ошибкой

Если мы прокрутим экран ниже, то увидим список найденных ошибок. Валидатор не способен точно указать, где проблемы, но если мы понимаем свой код, то и сами их найдем. На рис. 2.9 показано, как валидатор описывает ошибки.

Всегда полезно периодически прогонять через программу-валидатор HTML-код, чтобы убедиться в корректности последнего. Двигаясь к концу этой главы, я буду периодически напоминать вам еще разок проверить HTML с помощью валидатора. Пожалуйста, не забывайте делать это.

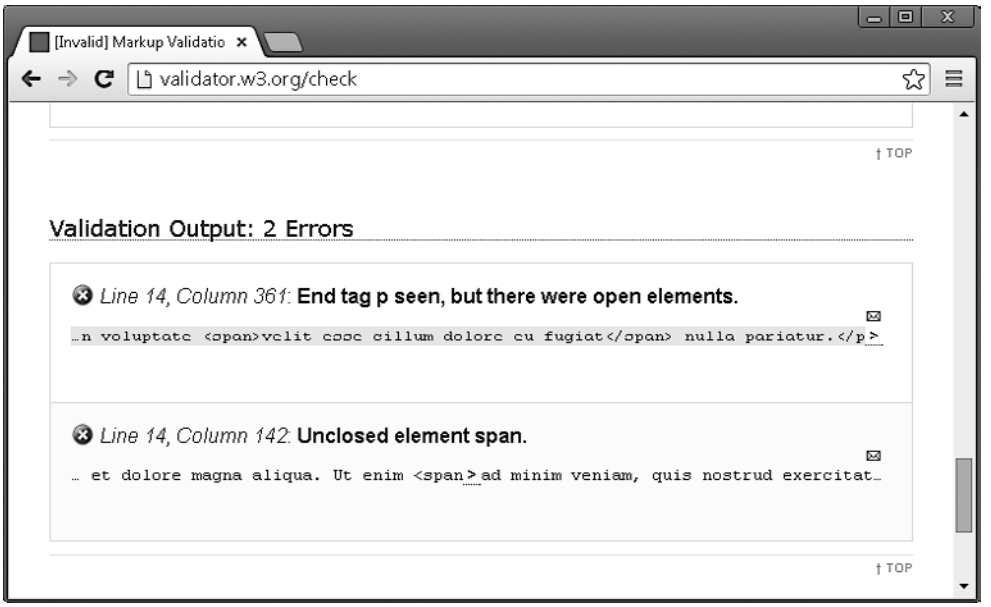


Рис. 2.9. Валидатор W3C HTML описывает допущенные ошибки

Amazeriffic

В оставшейся части главы мы с вами создадим HTML для учебного веб-приложения. Таким образом мы убьем сразу двух зайцев: опробуем на настоящем проекте рабочий процесс, изученный в предыдущей главе, а также рассмотрим еще несколько важных тегов HTML и отображаемых ими элементов.

Определение структуры

Учебное приложение называется Amazeriffic, от слов amazing — «изумительный» и terrific — «невероятный». Это название, конечно, немного глуповато, но не хуже названия любой другой компании, размещающейся в наши дни в Кремниевой долине. Предназначение продукта Amazeriffic состоит в отслеживании и категоризации набора задач (по сути, это список необходимых дел). Далее в этой книге мы и в самом деле поработаем над реализацией проекта, похожего на этот, но пока не освоимся достаточно с HTML, сконцентрируемся на главной странице продукта. Страница, которую мы создадим, будет похожа на рис. 2.10.

Помните, что HTML задает структуру документа. Это значит, что, даже если мы видим здесь множество стилистических элементов (разные шрифты, цвета и даже верстка элементов), по большей части их следует игнорировать, так как к HTML они не имеют никакого отношения. Пока мы с вами концентрируемся исключительно на структуре документа.

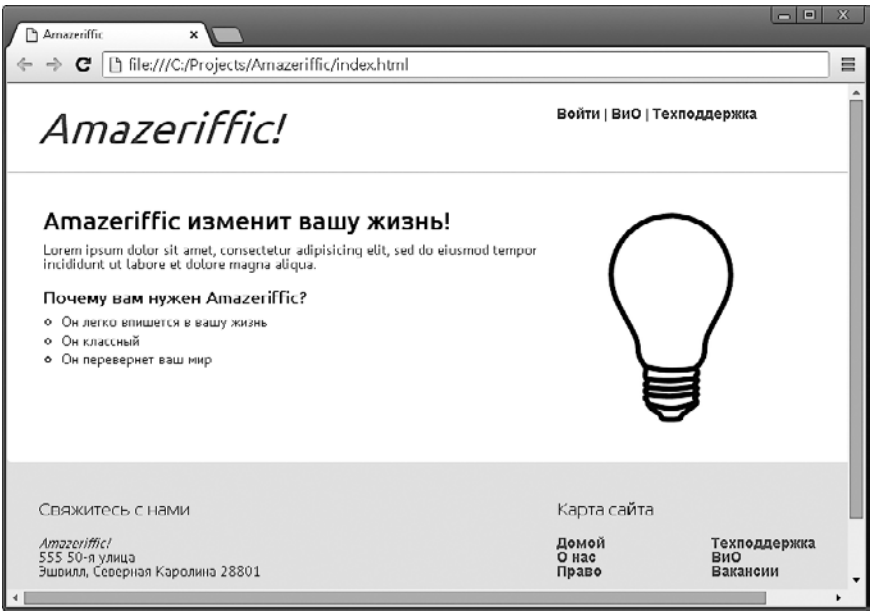


Рис. 2.10. Страница Amazeriffic, которую мы создадим в течение этой и последующих глав

Пока не перешли непосредственно к кодированию, даайте посмотрим, удастся ли выделить различные части структуры. Попробуйте нарисовать эскиз этой страницы на листке бумаги карандашом, а затем пометить все структурные элементы так тщательно, как только получится. Если вы не можете сообразить, что от вас требуется, просто обведите все большие и маленькие элементы на странице, а затем присвойте им описательные имена, которые показывают их роль в документе.

На рис. 2.11 вы можете посмотреть на версию предыдущего макета, где элементы обведены пунктирными линиями. Без труда можно заметить, что некоторые элементы находятся внутри других. Таким образом мы видим взаимоотношения, определяющие, какие элементы будут потомками других элементов в DOM, и это поможет нам примерно прикинуть, каким будет HTML-код.

Нетрудно и подписать обведенные элементы. Например, вполне очевидно, где находятся заголовок, логотип, навигационные ссылки, подвал, контактная информация, карта сайта, основное содержание, второстепенное содержание и изображение. Все они представляют собой какие-то структурные элементы страницы.

Визуализация структуры с помощью древовидной диаграммы

После идентификации всех структурных элементов нужно продумать, как они будут сочетаться друг с другом. Чтобы сделать это, построим древовидную диаграмму для структуры, которая определит содержимое разных элементов. На рис. 2.12 показано древовидное представление возможной структуры.

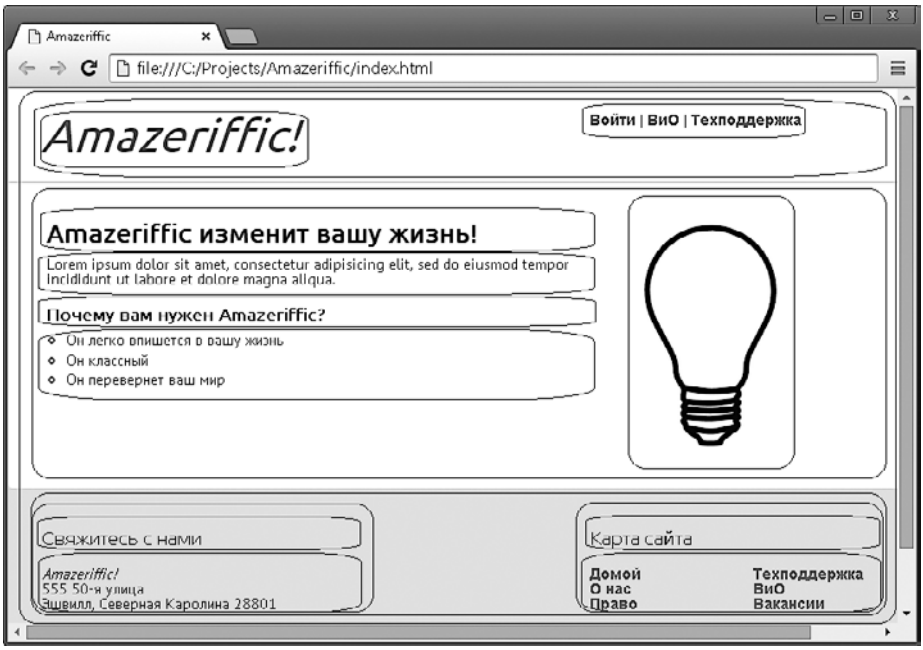


Рис. 2.11. Макет Amazerrific, размеченный для представления структуры

Реализация структуры в ходе рабочего процесса

Теперь, когда у нас есть древовидное представление страницы (как в голове, так и на бумаге), очень легко написать код HTML, если мы знаем теги, необходимые для всех элементов на странице. Поскольку вы еще не видели некоторые теги, которые здесь упомянуты, я буду объяснять их назначение по ходу дела.

Прежде чем что-либо делать, создадим каталог для хранения проекта. Если вы следовали инструкциям, приведенным в главе 1, у вас уже есть папка Projects в домашней папке (если вы в Mac OS или Linux) либо в каталоге Documents (если работаете в Windows). Перейдем в эту папку из командной строки Terminal application в Mac OS или из Git Bash в Windows. Используем команду `cd`:

```
hostname $ cd Projects
```

Находясь в этой папке, создадим каталог для хранения проекта Amazeriffic. Какую команду нужно использовать? Правильно! Команду `mkdir`:

```
hostname $ pwd
/Users/semmy/Projects
hostname $ mkdir Amazeriffic
```

Затем мы можем получить некоторую визуальную обратную связь, говорящую о том, что папка была успешно создана, и наконец перейти в нее с помощью команды `cd`:

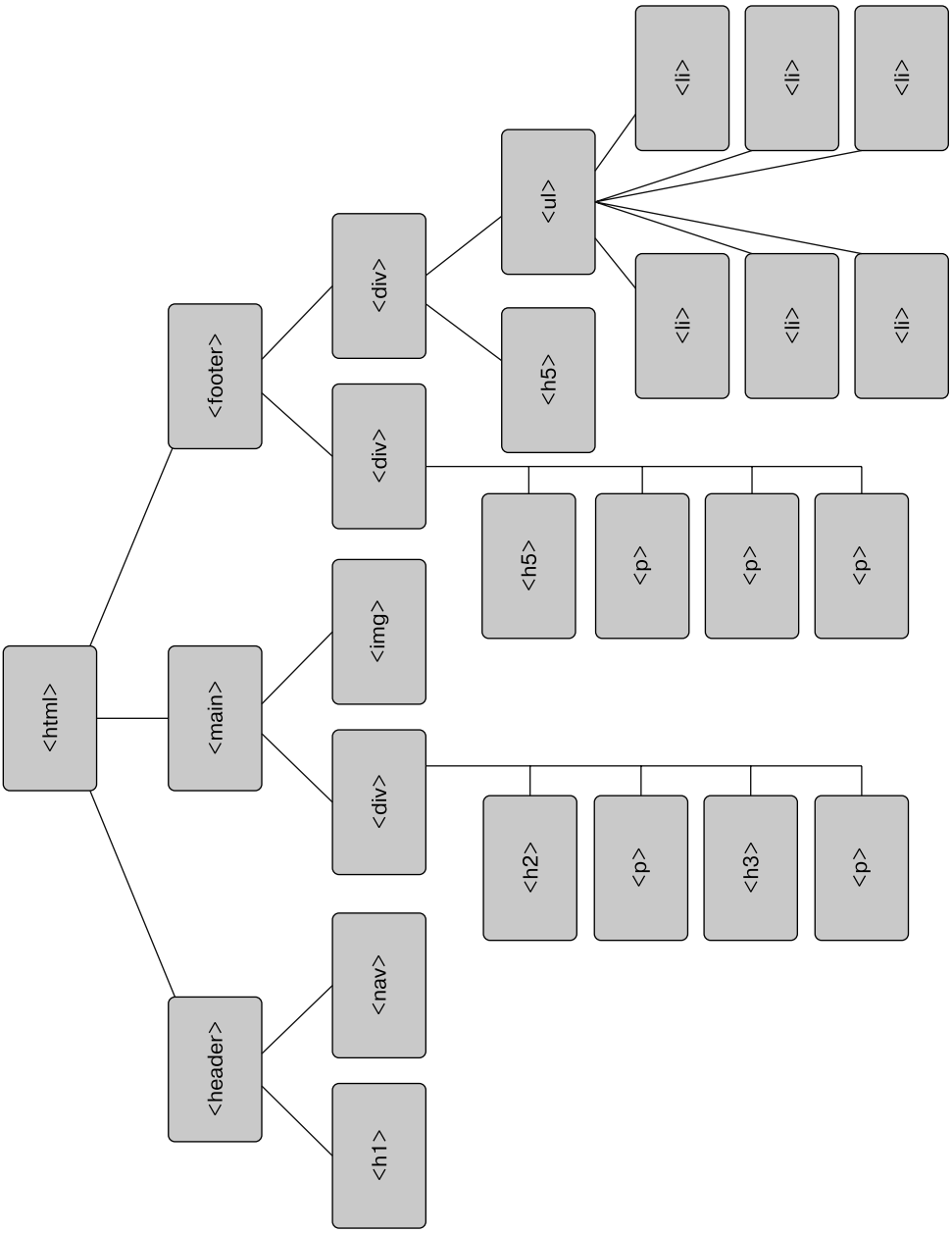


Рис. 2.12. Структура Amazeriffic в виде древовидной диаграммы

```
hostname $ ls
Amazeriffic
hostname $ cd Amazeriffic
```

Таким образом, сейчас мы находимся в новой папке проекта (в чем можем убедиться с помощью команды `pwd`). Следующая очень важная задача — поместить эту папку в систему контроля версий. Создадим проект Git с помощью команды `git init`:

```
hostname $ git init
Initialized empty Git repository in /Users/semmy/Projects/Amazeriffic/.git/
```

Теперь, когда мы создали рабочую папку и поместили ее в систему контроля версий, наконец-то можно приступить к кодированию! Запустите Sublime, а затем откройте папку Amazeriffic, используя клавиатурное сокращение, описанное в предыдущей главе.

Затем мы можем создать новый HTML-документ, щелкнув правой кнопкой мыши на навигационной панели и выбрав New File. В результате будет создан безымянный файл, который мы тут же переименуем, просто набрав `index.html`. После того как файл создан и переименован, можем открыть его двойным щелчком. Добавим «Hello, World!», чтобы в документе было какое-то видимое в браузере содержимое.

Убедившись, что у нас есть рабочая основа, можно запустить Chrome и открыть страницу. Если все идет хорошо, мы увидим в браузере «Hello, World!».

Теперь можно начать с построения HTML-скелета нашего документа. Замените «Hello, World!» в файле `index.html` следующим содержимым:

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
    <h1>Amazeriffic</h1>
  </body>
</html>
```

Сохраните файл и еще раз откройте страницу в браузере. Сделав это, увидите нечто похожее на рис. 2.13.

Проделав все это, можем выполнить наш первый коммит. Перейдите к командной строке. Сначала проверьте статус рабочей папки, а затем добавьте и зафиксируйте в системе контроля версий файл `index.html`:

```
hostname $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
```

Рис. 2.13. Страница Amazeriffic после добавления нескольких базовых элементов

```
#
#   index.html[1]
hostname $ git add index.html
hostname $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   index.html[2]
hostname $ git commit -m "Add default index.html to repository."
[master (root-commit) fd60796] Add default index.html to the repository.
 1 file changed, 10 insertions(+)
 create mode 100644 index.html[3]
```

Вы видите, что начали мы с проверки статуса. Это очень хорошая привычка — визуальная обратная связь всегда полезна. Кроме всего прочего, если мы случайно изменим файл, который не собирались менять, то получим подсказку об этом. Сейчас, однако, мы видим здесь только один файл — `index.html`.

---

[1] Системное сообщение: «От бранч-мастера. Начальный коммит. Неотслеживаемые файлы (используйте "git add <file>..." для добавления их в список для фиксирования изменений): новый файл index.html». — *Примеч. пер.*

[2] Системное сообщение: «От бранч-мастера. Начальный коммит. Будут зафиксированы изменения (используйте "git rm –cached <file>..." для отмены фиксации): новый файл index.html». — *Примеч. пер.*

[3] Системное сообщение: «1 файл изменен, 10 вставок (+). Создан модуль 100644 index.html». — *Примеч. пер.*

Добавляем index.html и смотрим, что получилось, с помощью еще одного вызова git status. Это покажет нам, какие файлы войдут в коммит с командой git commit. И наконец, выполним git commit, добавив соответствующее сообщение.

Теперь мы готовы по-настоящему приступить к построению структуры страницы. Вернувшись к древовидной диаграмме, можем заметить, что у нас есть заголовок, секция с основным контентом и подвал — нижняя часть страницы. Каждый из них является потомком элемента body. В HTML есть специальные теги для каждой из этих трех секций документа. Теги <header> и <footer> служат для элементов, находящихся в самом верху и в самом низу страницы, а тег <main> представляет секцию, где находится основное содержимое документа:

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
    <header>
      <h1>Amazeriffic</h1>
    </header>
    <main>
    </main>
    <footer>
    </footer>
  </body>
</html>
```

Обратите внимание: мы передвинули тег <h1>, содержащий название Amazeriffic, внутрь тега заголовка. Это вызвано тем, что в нашей древовидной диаграмме название является потомком заголовка.

Затем обратимся к верхнему правому углу страницы, где находится небольшой навигационный блок со ссылками на страницы Регистрация, ВиО, Техподдержка. Очень удобно: в HTML есть специальный тег для создания навигационных элементов, который так и называется — nav. Добавим эту секцию в тег <header>:

```
<header>
  <h1>Amazeriffic</h1>
  <nav>
    <a href="#">Регистрация</a> |
    <a href="#">ВиО</a> |
    <a href="#">Техподдержка</a>
  </nav>
</header>
```

Обратите внимание: элемент nav содержит несколько ссылок, разделенных символом |. Этот символ находится на вашей клавиатуре справа, над клавишей Enter, он совмещен с обратным слэшем. Нужно удерживать клавишу Shift, чтобы набрать его.

Ссылки в элементе nav заключены в теги <a>. Как уже упоминалось, теги <a> содержат атрибуты href, в которых обычно находится адрес страницы, куда мы должны перейти, щелкнув по ссылке. Поскольку пока что наш пример не будет содержать никаких настоящих ссылок, мы используем значок # как временную заглушку для места, куда нужно потом вставить ссылку.

Завершив работу над секцией <header>, будет весьма неплохо зафиксировать ее в хранилище Git. Сначала будет полезно выполнить команду git status, чтобы увидеть измененные файлы в хранилище. Затем мы выполняем git add и git commit с каким-то поясняющим сообщением об изменениях, содержащихся в коммите.

Структурирование основной части

После окончания работы над секцией <header> можем перейти к <main>. Из диаграммы видно, что эта секция разделена на две основные части, как и заголовок. Там есть содержимое, находящееся слева, и изображение справа. Текст слева разделен на две отдельные секции, и нам придется принять это во внимание.

Чтобы структурировать текст на странице слева, понадобятся четыре новых тега. Мы используем два заголовочных тега (<h2> и <h3>), которые представляют собой заголовки менее важные, чем <h1>. Появится и тег <p>, который означает содержимое абзаца. Кроме того, мы используем тег <ul>, чтобы создать маркированный список, внутри которого теги <li> будут содержать элементы списка.

И последнее по порядку, но не по важности: нам понадобится тег <img>, чтобы вставить изображение лампочки. Обратите внимание на то, что у тега <img> нет соответствующего закрывающего элемента. Это потому, что HTML5 включает набор элементов, относящихся к *пустым* (void). Пустые элементы обычно не подразумевают какого-либо содержимого внутри и, соответственно, не требуют закрывающего тега.

Зато, как вы сейчас увидите, у тега <img> есть необходимый атрибут alt. Он содержит текстовое описание изображения. Это сделано для обеспечения доступности нашей страницы для слабовидящих пользователей, которые часто применяют программы для чтения содержимого экрана при пользовании Интернетом.

Вы можете скачать изображение лампочки с сайта http://www.learningwebappdev.com/lightbulb.png. Чтобы оно появилось на вашей странице, необходимо сохранить его в ту же самую папку, где находится файл index.html.

После добавления структурированного содержания в тег <main> мы получим примерно такой код:

```
<h2>Amazeriffic изменит вашу жизнь!</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
<h3>Почему вам нужен Amazeriffic?</h3>
<ul>
```

```
  <li>Он легко впишется в вашу жизнь</li>
  <li>Он классный</li>
  <li>Он перевернет ваш мир</li>
</ul>
<img src="lightbulb.png" alt="Изображение лампочки">
```

На этом этапе неплохо будет прогнать код через валидатор и убедиться, что вы ничего случайно не пропустили. Если все хорошо, сделайте еще один коммит в наше хранилище Git, а затем перейдем к подвалу.

Структурирование подвала

Подвал, как и два предыдущих блока, содержит две основные логические части. Одна из них включает информацию о компании, а другая — набор ссылок, которые можно назвать *картой сайта*. Более того, сама карта сайта поделена на две колонки.

В первую очередь следует отметить, что элемента HTML специально для контактной информации не существует. Это нормально, так как HTML предоставляет два тега общего назначения, `<div>` и `<span>`, которые позволяют создавать разнообразные элементы, структуру которых мы определяем самостоятельно. Разницу между `div` и `span` рассмотрим в следующей главе.

В данном случае в подвале есть две разные структуры: контактная информация и карта сайта. Таким образом, нам понадобятся два элемента `<div>`, у каждого из которых есть свой атрибут `class`, с помощью которого указывается тип элемента. Пока запомните, что атрибут `class` — это свойство, с помощью которого вы указываете назначение элементов `<div>` и `<span>`.

Кроме того, мы используем еще один тег `<ul>` для создания маркированного списка элементов карты сайта. В результате у нас получится вот такой HTML для структуры подвала:

```
<footer>
      <div class="contact">
        <h5>Свяжитесь с нами</h5>
        <p>Amazeriffic!</p>
        <p>555 50-я улица</p>
        <p>Эшвилл, Северная Каролина 28801</p>
      </div>
      <div class="sitemap">
        <h5>Карта сайта</h5>
        <ul>
          <li><a href="#">Домой</a></li>
          <li><a href="#">О нас</a></li>
          <li><a href="#">Право</a></li>
          <li><a href="#">Техподдержка</a></li>
          <li><a href="#">ВиО</a></li>
          <li><a href="#">Вакансии</a></li>
        </ul>
      </div>
    </footer>
```

Добавьте содержание подвала в HTML-документ, проверьте его в HTML-валидаторе, чтобы убедиться в отсутствии ошибок, а затем отправьте в хранилище Git очередной коммит.

Мы вернемся к этому примеру в главе 3, когда займемся его стилевым оформлением.

Подведем итоги

В этой главе мы научились создавать структуру пользовательского интерфейса приложения с помощью HTML. HTML — это язык разметки, который позволяет нам использовать теги для определения структуры, называемой объектной моделью документа (Document Object Model (DOM)). Браузер использует DOM для создания визуального представления страницы.

DOM является иерархической структурой и легко может быть представлена в виде древовидной диаграммы. Иногда полезно представлять себе DOM в виде дерева, так как это более ясно показывает отношения между элементами: предками, потомками и родительскими.

Валидатор — полезный инструмент, помогающий нам избежать простых ошибок и просчетов при построении HTML.

В этой главе мы также изучили несколько тегов, которые перечислены в табл. 2.1. Они представляют собой специфические элементы структуры, за исключением тега `<div>`. К тегу `<div>` обычно добавляется атрибут `class`, с помощью которого указывается значение этого элемента.

Таблица 2.1. Теги HTML

Тег	Описание
<html>	Основной контейнер HTML-документа
<head>	Содержит метаинформацию о документе
<body>	Содержит то, что будет отображаться в браузере
<header>	Заголовочная часть страницы
<h1>	Самый важный заголовок (в документе обычно только один)
<h2>	Второй по важности заголовок
<h3>	Третий по важности заголовок
<main>	Область основного содержания документа
<footer>	Подвал (нижняя часть) документа
<a>	Ссылка на другой документ или страницу, куда переходят по щелчку
	Список элементов, для которых порядок перечисления неважен (маркированный)
	Список элементов, для которых порядок перечисления важен (нумерованный)
	Элемент списка (любого)
	Контейнер для подструктуры

Больше теории и практики

В коде Amazeriffic я оставил несколько ошибок. Рекомендую вам исправить их и заставить код работать настолько хорошо, насколько вы сможете. Помните: если

возникнут какие-то проблемы, вы всегда можете обратиться к финальному варианту HTML на нашей странице GitHub.

Заучивание

Выучив основы HTML, мы можем добавить несколько новых элементов к задачам на заучивание. В дополнение к пяти шагам, упомянутым в предыдущей главе, вы можете выполнять следующие дополнительные упражнения.

1. Откройте файл в Chrome, используя клавиатурные сокращения.
2. Измените файл `index.html`, включив в него теги `<!doctype>`, `<html>`, `<head>` и `<body>`.
3. Добавьте тег `<p>`, содержащий просто слова «Hello, World!».
4. Обновите файл в Chrome и убедитесь, что он отображается корректно (если это не так, исправьте ошибки).
5. Отправьте новый коммит в хранилище в Git из командной строки.
6. Добавьте в тег `<body>` теги `<header>`, `<main>` и `<footer>`.
7. Убедитесь, что все корректно отображается в Chrome.
8. Проверьте файл с помощью HTML-валидатора.
9. Отправьте изменения файла `index.html` в хранилище Git.

Древовидные диаграммы

Нарисуйте древовидную диаграмму для следующего HTML-документа. Мы еще раз используем этот документ в качестве практической задачи в конце глав 4 и 5:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Привет!</h1>
    <h2 class="important">Hi again</h2>
    <p class="a">Абзац любого текста</p>
    <div class="relevant">
      <p class="a">Первый</p>
      <p class="a">Второй</p>
      <p>Третий</p>
      <p>Четвертый</p>
      <p class="a">Пятый</p>
      <p class="a">Шестой</p>
      <p>Седьмой</p>
    </div>
  </body>
</html>
```

Составление страницы ВиО (FAQ) для Amazeriffic

В навигационной панели Amazeriffic находится ссылка на страницу ВиО (Вопросы и ответы), закрытая заглушкой. Создайте страницу с точно такими же заголовком и подвалом, как на главной, но содержащую список вопросов и ответов в основной части. Используйте текст Lorem ipsum (если, конечно, не хотите придумать текст вопросов и ответов сами).

Сохраните этот файл как `faq.html`. Можете связать наши две страницы с помощью атрибута `href` в теге `<a>`, установив его значение `faq.html`. Если вы поместите оба файла в одну и ту же папку, то можете щелкнуть на ссылке, находящейся на главной странице, и оказаться на странице ВиО. Можете установить и ссылку обратно на страницу `index.html` с `faq.html`.

Больше об HTML

На протяжении всей книги я буду рекомендовать вам Mozilla Developer Network documentation для дальнейшего изучения определенных тем. Этот сайт содержит замечательное описание HTML. Я рекомендую углубленно изучить документацию и расширенные возможности.

3 Стиль

В предыдущей главе мы изучили создание структуры HTML-документа и некоторые соответствующие ментальные модели. Но созданные нами страницы оставляют желать лучшего с точки зрения стиля и дизайна.

В этой главе мы постараемся решить некоторые из этих проблем, изучив, как можно изменить отображение HTML-документов с помощью *каскадных таблиц стилей* (Cascading Style Sheets (CSS)). Как упоминалось в предыдущей главе, это даст вам достаточно информации для начала работы с CSS, а из раздела «Больше теории и практики» данной главы вы узнаете, как эксплуатировать эти ресурсы.

Привет, CSS!

Чтобы сделать первые шаги, начнем с простого HTML-примера, похожего на начальные упражнения в предыдущей главе. Откройте командную строку и создайте каталог под названием Chapter3 (Глава 3) в папке Projects.

А сейчас запустите Sublime Text и откройте папку Chapter3 так, как мы это делали в предыдущей главе. Создайте следующий HTML-файл и сохраните его в этой папке как `index.html`:

```
<!doctype html>
<html>
  <head>
    <title>Мое первое веб-приложение</title>
    <link href="style.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>Всем привет!</h1>
    <p>Это абзац.</p>
  </body>
</html>
```

В этом файле составлена очень простая страница HTML с элементами `h1` и `p`, содержащимися в элементе `body`. Вы, конечно, обратили внимание на новый тег, находящийся внутри тега `<head>`. Тег `<link>` связывает наш файл с другим, в котором показан стиль отображения документа.

Вы можете создать файл `style.css` с помощью Sublime из навигационного файлового окна в редакторе. Содержание файла пусть будет следующим:

```
body {
    background: lightcyan;
    width: 800px;
    margin: auto;
}
h1 {
    color: maroon;
    text-align: center;
}
p {
    color: gray;
    border: 1px solid gray;
    padding: 10px;
}
```

Это простой пример файла CSS. В этом конкретном файле мы устанавливаем для элемента `body` светло-голубой фон (`lightcyan`) и указываем браузеру, что текст, содержащийся в элементе `h1`, должен быть светло-коричневым (`maroon`). Кроме того, мы задаем ширину области `body` 800 пикселов и устанавливаем ширину полей `auto`, благодаря чему область `body` будет выровнена по центру страницы. И наконец, мы задаем для текста, находящегося в элементе `p`, серый цвет и создаем вокруг него тонкую границу.

По сути дела, файл CSS описывает, как определенные элементы HTML будут отображаться в браузере. Например, на рис. 3.1 показано, как предыдущий HTML взаимодействует с файлом CSS.

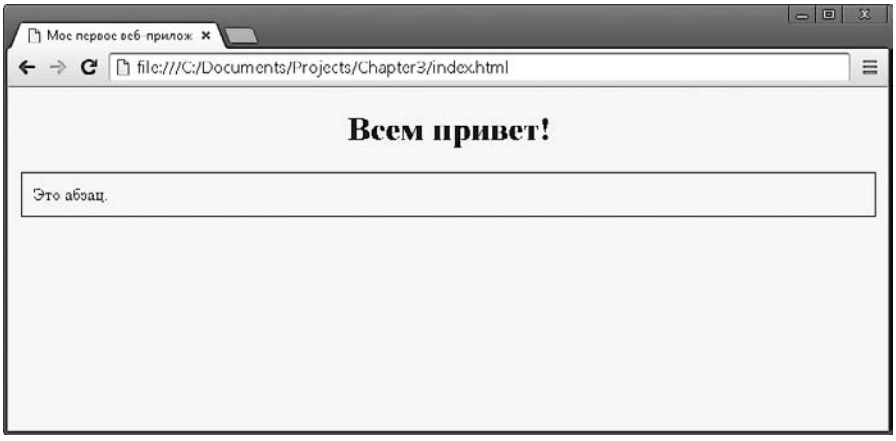


Рис. 3.1. Файл index.html, связанный с таблицей стилей и открытый в Chrome

Если бы мы не включили CSS, файл выглядел бы так, как продемонстрировано на рис. 3.2.

Рис. 3.2. Файл index.html, не связанный с таблицей стилей и открытый в Chrome

Если файлы `index.html` и `style.css` находятся в одной и той же папке, то вы можете открыть первый из них в браузере и увидеть, что он похож на рис. 3.1. Фактически если вы создадите еще одну страницу HTML с другим содержанием и свяжете ее с тем же самым файлом CSS, то обе страницы будут отображаться так, как указано в этой таблице стилей. Это одно из главных преимуществ CSS — он позволяет стилизовать сразу несколько страниц с помощью всего одного файла!

Наборы правил

Файл CSS — это коллекция *наборов правил*, а набор правил — это просто ряд указаний относительно стилевого оформления, которые применяются к какому-либо виду элементов в DOM (как вы помните, это иерархическая система объектов в HTML-документе). Набор правил состоит из *селектора* — это может быть, например, название тега, открывающей фигурной скобки, списка правил и закрывающей фигурной скобки. Каждое правило состоит из определенного *свойства*, после которого следуют двоеточие, значение этого свойства (или список значений, разделенных пробелами), после чего ставится точка с запятой, например:

```
body {
    width: 800px;
    background: lightcyan;
    color: #ff0000;
}
```

Это пример набора правил, применяемых в DOM к элементу `body`. В данном случае селектор — это `body`, то есть просто название элемента HTML, к которому мы хотим применить стиль. Эти правила будут применяться ко всему содержимому элемента `body`, то есть ко всем элементам, находящимся внутри него. В набор

входят три правила: первое определяет значение свойства `width` (ширина), второе — свойства `background` (фон), а третье — свойства `color` (цвет).

В CSS существует два способа указания цвета. Первый — названия наиболее часто употребляемых в CSS цветов. Второй — указание шестнадцатеричного цветового кода. Этот код состоит из шести цифр шестнадцатеричной системы счисления (0–9 или A–F). Первая пара определяет количество в цвете красного, вторая — зеленого, а третья — синего. Эти три цвета являются основными в цветовой модели RGB.

Опытный разработчик CSS хорошо знает типы свойств, которые могут быть установлены для какого-либо элемента, а также разбирается во всем разнообразии их значений. Большинство свойств, например цвет фона или шрифта, могут применяться к очень многим элементам HTML.

Комментарии

В наш файл CSS можно также добавлять комментарии. Как и в HTML, комментарии в CSS являются просто пометками к коду, которые браузер полностью игнорирует. Например, мы можем добавить комментарии к предыдущему набору правил следующим образом:

```
/* Здесь находится стиль для элемента body */
body {
width: 800px; /* Устанавливаем ширину body 800 пикселов */
background: lightcyan; /* Устанавливаем светло-голубой цвет фона */
color: #ff0000; /* Устанавливаем красный цвет для элементов*/
}
```

Некоторые советуют свободно писать в программах комментарии. Но я все-таки склоняюсь к мысли, что лучше позволить коду говорить самому за себя, насколько это возможно, и минимизировать необходимость комментариев. На самом деле опытные разработчики CSS, скорее всего, сочтут приведенные ранее комментарии избыточными, да и вы, когда узнаете обо всех этих элементах несколько больше, перестанете в них нуждаться. В то же время бывают ситуации, когда неочевидно, что должен делать код, тогда сделать некоторые пометки весьма полезно.

Если вы только начинаете работать с CSS, я бы рекомендовал вам писать комментарии чаще и подробнее. В оставшейся части главы я использую комментарии в свободной форме, чтобы сделать материал более понятным.

Отступы, границы и поля

В большинстве своем элементы HTML отображаются двумя способами. Первый — *внутристрочный*, который применяется, например, к элементам `span`. Это значит, кроме всего прочего, что содержимое тега будет появляться на той же строке, что и окружающие его элементы:

```
<div>
Этот абзац и это <span>слово</span> отображаются внутри строки. Эта
<a href="http://www.example.com">ссылка</a> тоже отображается внутри строки.
</div>
```

Если мы добавим этот элемент в тег `body` в нашем `index.html`, то страница будет отображаться, как показано на рис. 3.3.

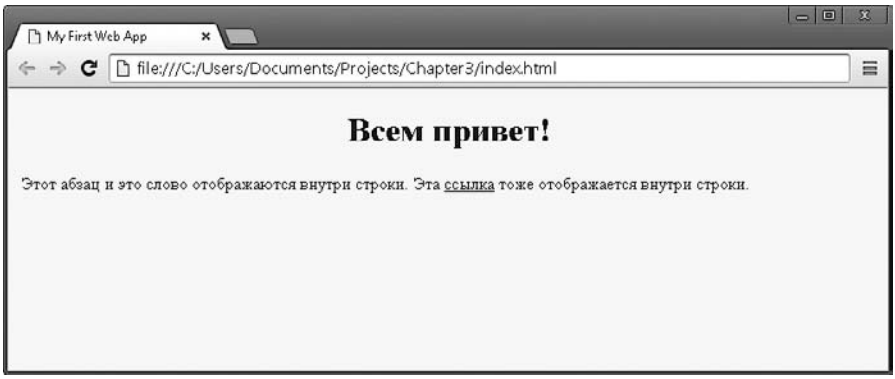


Рис. 3.3. Пример внутристрочного отображения элемента внутри блочного элемента

Но чаще элементы являются *блочными*, а не внутристрочными. Это значит, что содержимое, находящееся внутри элемента, будет отображаться с новой строки по отношению ко всему тексту. К блочным элементам, с которыми мы уже знакомы, относятся `p`, `nav`, `main` и `div`:

```
<div>
Этот абзац и это <div>слово</div> отображаются внутри строки. Эта
<a href="http://www.example.com">ссылка</a> тоже отображается внутри строки.
</div>
```

Этот код будет выглядеть несколько иначе (рис. 3.4).

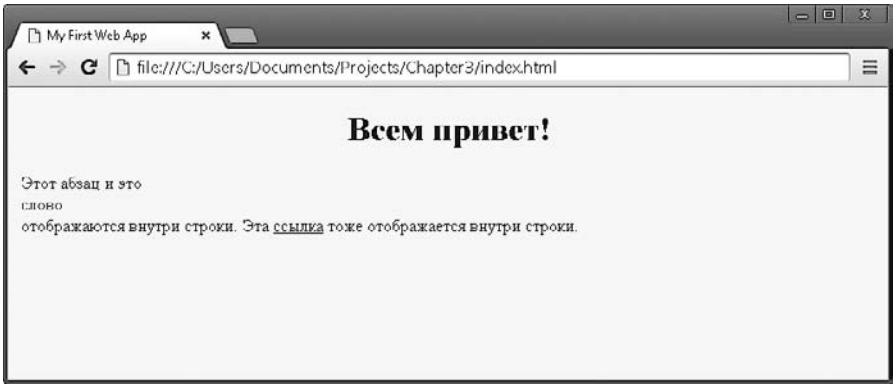


Рис. 3.4. Пример блочного элемента, находящегося внутри другого блочного элемента

Стили как блочных, так и внутристрочных элементов имеют свойства цветов фона и элемента. Но в стиле блочных элементов содержатся еще три важных свойства, которые очень полезны при настройке компоновки страницы: *отступы* (padding), *границы* (border) и *поля* (margin).

Отступ представляет собой пространство между содержанием элемента и его границей, а поля — пространство между самим элементом и его контейнером. Границей же является пространство между отступом и полями. Эта схема показана на рис. 3.5.

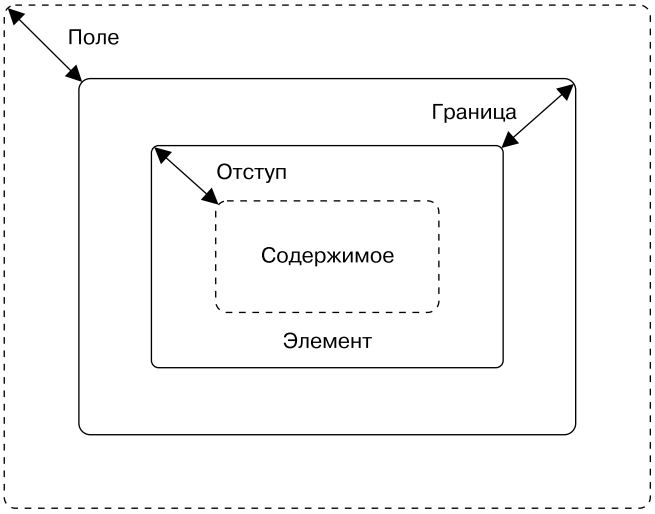


Рис. 3.5. Поля, граница и отступы блочного элемента DOM

Рассмотрим простой пример, который поможет изучить манипулирование размерами полей, отступов и границы для различных элементов. Создадим файл `margin_border_padding.html`:

```
<!doctype html>
<html>
<head>
<title>Глава 3 – Пример границ, отступов и полей</title>
<link href="margin_border_padding.css" rel="stylesheet" type="text/css">
</head>
<body>
<div>
<p>ЭТО АБЗАЦ, НАХОДЯЩИЙСЯ ВНУТРИ ЭЛЕМЕНТА DIV</p>
</div>
</body>
</html>
```

Еще нам понадобится файл `margin_border_padding.css`, на который мы ссылались в файле HTML ранее. Вы можете создать эти файлы в одной папке в Sublime, а затем открыть файл `margin_border_padding.html` в Chrome:

```
body {
  background: linen;
  width: 500px;
  margin: 200px auto;
}
div {
  border: 5px solid maroon;
  text-align: center;
  padding: 5px;
}
p {
  border: 2px dashed blue;
}
```

Если вы все набрали правильно, то страница будет выглядеть так, как показано на рис. 3.6.

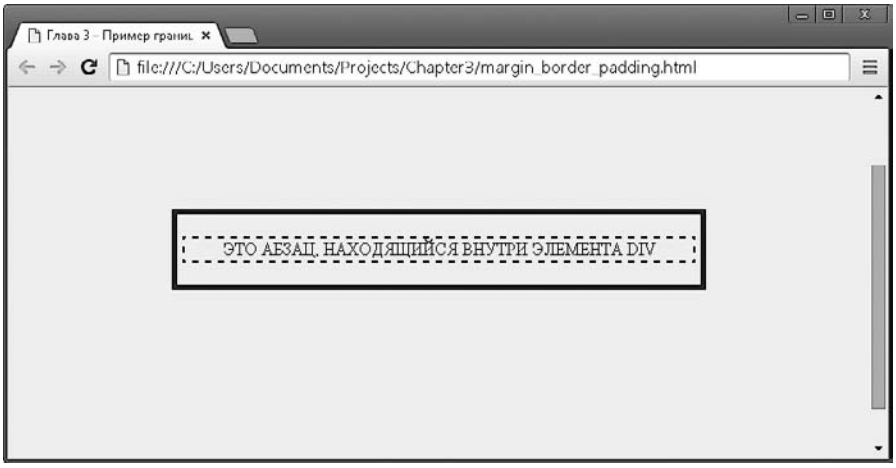


Рис. 3.6. Пример для изучения полей, отступов и границы

Вы можете видеть, что все элементы (и `div`, и `p`) являются блочными и имеют собственные границу, поля и отступы. Если вы этого еще не добились, попробуйте повторить HTML и CSS, показанные здесь. Затем поупражняйтесь немного в изменении свойств `padding`, `border` и `margin` каждого элемента, чтобы разобраться, как они влияют на отображение страницы.

Селекторы

Самый важный аспект в CSS — эффективное использование селекторов. Мы уже видели базовый тип селекторов, в качестве которых используется название тега — для применения стиля выбираются все теги с таким именем. Например, рассмотрим следующий HTML:

```
<body>
  <h1>Привет!</h1>
  <p>Это абзац.</p>
  <p>Это второй абзац.</p>
</body>
```

Предположим, что для применения к нему стиля используется следующий CSS:

```
h1 {
  background: black;
  color: white;
}
p {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

Первый набор правил оформляет элемент `h1`, а второй — *оба* элемента `p`. Во многих случаях нам будет требоваться именно это. Однако в других мы можем захотеть применить к первому и второму абзацам разные стили.

Классы

В главе 2 мы уже видели, что можно добавить атрибут `class` к тегам `<div>`, чтобы отличать один от другого. На самом деле мы можем добавить *класс* к любому элементу DOM. Например, можем переписать предыдущий HTML следующим образом:

```
<body>
  <h1>Привет!</h1>
  <p class="first">Это абзац.</p>
  <p class="second">Это второй абзац.</p>
</body>
```

Теперь мы можем применить стиль к определенному абзацу в соответствии с его классом:

```
h1 {
  background: black;
  color: white;
}
p.first {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

В этом примере набор правил `p.ruleset` будет применяться только к первому элементу `p`, то есть к первому абзацу. Если класс появляется только у определенного

типа элементов (как обычно и происходит), то мы можем опустить имя тега и просто использовать класс:

```
.first {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

Псевдокласс

В главе 2 мы убедились, что можем создавать кликабельные элементы DOM с помощью тега `<a>`:

```
<body>
  <a href="http://www.example.com">Click Me!</a>
</body>
```

Элементу `a` может быть присвоен стиль точно так же, как любому другому элементу DOM, — с помощью CSS. Например, мы можем создать файл CSS, который изменяет цвета всех ссылок:

```
a {
  color: cornflowerblue;
}
```

Часто бывает полезно менять цвет ссылки в зависимости от того, нажимал на нее пользователь или еще нет. CSS позволяет изменить цвет однажды нажатых ссылок добавлением нового набора правил:

```
a {
  color: cornflowerblue;
}
a:visited {
color: tomato;
}
```

В этом коде `visited` — пример псевдокласса CSS для элемента `a`. Он ведет себя почти так же, как и обычный класс, и мы можем присвоить элементам стили с его помощью, как будто он является обычным классом. Главное отличие состоит в том, что браузер самостоятельно добавляет этот класс.

Основной случай использования псевдоклассов CSS — изменение способа отображения ссылки после того, как пользователь наводит на нее указатель мыши. Этого можно достичь с помощью псевдокласса `hover` для элемента `a`. В следующем примере мы изменяем предыдущий пример так, чтобы ссылка становилась подчеркнутой, только когда пользователь наводит на нее указатель мыши:

```
a {
  color: cornflowerblue;
  text-decoration: none; /* убираем подчеркивание по умолчанию */
}
```

```
a:visited {
  color: tomato;
}
a:hover {
  text-decoration: underline;
}
```

Более сложные селекторы

По мере разрастания древовидной диаграммы DOM появляется необходимость в более сложных селекторах. Рассмотрим, например, такой HTML:

```
<body>
  <h1>Привет!</h1>
  <div class="content">
    <ol>
      <li>Элемент списка <span class="number">первый</span></li>
      <li>Элемент списка <span class="number">второй</span></li>
      <li>Элемент списка <span class="number">третий</span></li>
    </ol>
    <p>Это <span>первый</span> абзац.</p>
    <p>Это <span> первый</span> абзац.</p>
  </div>
  <ul>
    <li>Элемент списка <span class="number">1</span></li>
    <li>Элемент списка <span class="number">2</span></li>
    <li>Элемент списка <span class="number">3</span></li>
  </ul>
</body>
```

В этом HTML-коде два списка, несколько абзацев, несколько элементов списка. Мы можем выбрать основные элементы и присвоить им стили, как обсуждалось ранее. Например, если нужна закругленная граница вокруг нумерованного списка (`ol`), можем использовать следующий набор правил:

```
ol {
  border: 5px solid darksalmon;
  border-radius: 10px;
}
```

А теперь допустим, что мы хотим сделать элементы нумерованного списка коричневыми. Для этого можно было бы прибегнуть к следующему:

```
li {
  color: brown;
}
```

Но таким образом мы изменим все элементы во всех списках — и в маркированном, и в нумерованном. Можно поступить более конкретно и указать селектор только для элементов `li` в нумерованном списке:

```
ol li {
  color: brown;
}
```

Если на странице несколько нумерованных списков, может понадобиться еще больше конкретики. Тогда мы укажем, что стиль относится только к тем элементам `li`, которые являются потомками элемента `div` класса `content`:

```
.content li {
  color: brown;
}
```

А теперь допустим, что надо сделать фон первого элемента всех списков желтого цвета. Для этого существует псевдокласс `first-child`, указывающий на первый дочерний элемент каждого родителя:

```
li:first-child {
  background: yellow;
}
```

Аналогично для второго, третьего и четвертого потомков мы можем использовать псевдокласс `ntn-child`:

```
li:nth-child(2) {
  background: orange;
}
```

Каскадные правила

Что если два разных набора правил используют селекторы, указывающие на один и тот же элемент в HTML? Например, представим, что у нас есть элемент `p` с классом `greeting`:

```
<p class="greeting">Приветствуем каскадные правила!</p>
```

А затем напишем два правила, которые применяют к этому элементу разные стили:

```
p {
  color: yellow;
}
p.selected {
  color: green;
}
```

Какое же из правил будет применено к указанному ранее элементу? Оказывается, мы получили набор *каскадных правил*, который браузер будет применять в случае конфликта. В данном случае более конкретное правило (класс) получает преимущество. Но что произойдет, если мы напишем что-то подобное:

```
p {
  color: yellow;
}
```

```
p {
  color: green;
}
```

В этом случае будет применяться набор правил, указанный в списке CSS позднее. Следовательно, абзац или абзацы окажутся зеленого цвета. Если мы поменяем правила местами, абзац станет желтым.

Наследование

Если вы еще не заметили, потомки наследуют свойства своих родителей. Что это значит в отношении применения стилей к элементам? Если мы устанавливаем для элемента какой-либо стиль, все его потомки в DOM будут оформлены точно так же, если только для них специально не указан какой-либо другой стиль. Так, например, если мы изменяем свойство `color` для элемента `body`, все элементы, являющиеся потомками `body` (то есть все элементы, отображающиеся на странице), будут наследовать этот цвет. Это самая суть CSS — вот почему очень полезно визуализировать DOM и держать в голове иерархию во время присвоения стилей элементам:

```
body {
  background: yellow;
}
/**
* Поскольку h1 является потомком тега body,
* у него будет желтый фон
*/
h1 {
color: red;
}
/**
* h2 тоже потомок элемента body, но мы
* перекрываем правило для его фона,
* он не будет желтым
*/
h2 {
  background: green;
}
```

Большая часть CSS-свойств работает именно таким способом. Надо отметить, что не все свойства наследуются по умолчанию. Основная часть ненаследуемых свойств относится к блочным элементам (правила для полей, отступов, границ элементы не наследуют от своих родителей):

```
body {
  margin: 0;
  padding: 0;
}
/**
```

```
* h1 не унаследует правила для полей и отступов от body,
* даже если мы не укажем иного
*/
h1 {
}
```

Плавающая компоновка

Мы познакомились с элементами, влияющими на базовый стиль элементов DOM. Но существуют и другие, более общие свойства, которые влияют на всю компоновку страницы относительно единичного элемента. Эти свойства значительно расширяют возможности разработчика в части управления расположением элементов на странице. Одно из самых часто используемых свойств — `float`. Это свойство может помочь нам создать более гибкие компоновки, чем статичная разметка, которую HTML создает автоматически.

Свойство элементов DOM под названием `float` может иметь значения `left` и `right`. Оно устанавливает элемент вне рабочего потока (согласно которому, как правило, элементы располагаются сверху вниз один за другим) и выравнивает его по левой или правой стороне элемента-контейнера при условии, что есть достаточно места, чтобы это сделать. Например, рассмотрим такой фрагмент HTML:

```
<body>
  <main>
    <nav>
      <p><a href="link1">Ссылка 1</a></p>
      <p><a href="link2">Ссылка 2</a></p>
      <p><a href="link3">Ссылка 3</a></p>
      <p><a href="link4">Ссылка 4</a></p>
    </nav>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
       eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
       ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
       aliquip ex ea commodo consequat. Duis aute irure dolor in
       reprehenderit in voluptate velit esse cillum dolore eu fugiat
       nulla pariatur.
    </p>
  </main>
</body>
```

В этом примере элементы `nav` и `p` находятся внутри элемента `main`. Мы также создали отдельный элемент `p` для каждой ссылки, потому что хотим, чтобы элементы отображались как блочные (каждая на отдельной строке). Без присвоения им стилей эти элементы будут располагаться друг за другом вертикально (рис. 3.7).

А сейчас попробуем применить к этому HTML следующий CSS:

```
main {
  width: 500px;
  margin: auto;
```

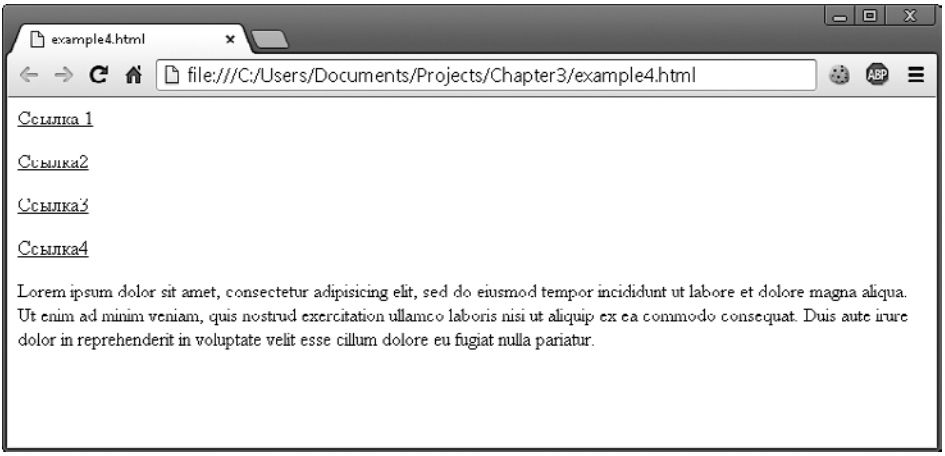


Рис. 3.7. Отображение страницы перед применением CSS

```
  background: gray;
}
nav {
  /* Уберите пометку комментария со следующей строки, если хотите иметь черную гра-
ницу */
  /* border: 3px solid black; */
  width: 200px;
  float: right;
}
p {
  margin: 0; /* Ноль от границы по умолчанию */
}
```

Результат будет похож на рис. 3.8.

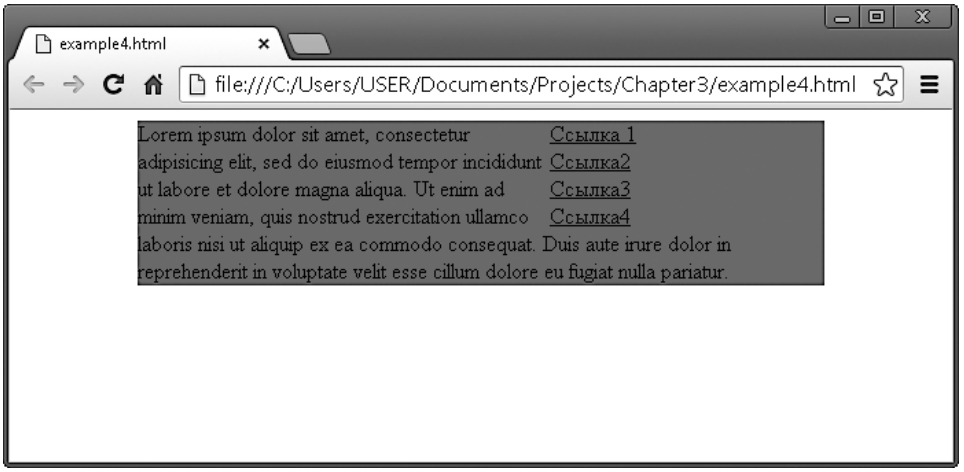


Рис. 3.8. Пример размещения элементов справа

Отметим, что мы задали ширину навигационного элемента равной 200 пикселам и разместили его справа, чтобы создать боковую панель. Обратите внимание, каким образом элемент `nav` был извлечен из стандартной вертикальной разметки и размещен справа. Содержание элемента `p` обтекает элемент `div`, содержащий в себе `nav`. Я также включил сюда линию — вы можете раскомментировать ее, чтобы вокруг плавающего элемента появилась визуальная граница. Попробуйте сделать это!

Хотя установка элемента плавающим справа отлично работает для изображений и других внутренних элементов, которые может обтекать текст, часто нужно создать компоновку, подобную сетке и состоящую из двух колонок. Эта задачка немного интереснее. В этом случае нужно сдвинуть элемент `p` влево и убедиться, что суммарный размер двух элементов не превышает размер контейнера. Вот как можно этого достичь с помощью CSS:

```
main {
  width: 500px;
  margin: auto;
  background: gray;
}
nav {
  width: 100px;
  float: right;
}
/* убираем значения по умолчанию для p, наследуемые от элемента nav */
nav p {
  margin: 0;
  padding: 0;
}
p {
  margin: 0; /* убираем значения полей для p по умолчанию */
  float: left;
  width: 400px;
}
```

Сейчас, если мы откроем страницу в браузере, то увидим, что у нас отлично вышли две колонки, но исчез серый фон. Это потому, что, когда все элементы, содержащиеся в контейнере, становятся плавающими, высота контейнера равняется нулю. Исправить это очень просто — устанавливаем для контейнера `div` свойство `overflow` со значением `auto`:

```
main {
  width: 500px;
  margin: auto;
  background: gray;
  overflow: auto;
}
```

Таким образом мы получили разметку, аналогичную показанной на рис. 3.9.

Отметим, что в этом примере мы установили суммарную ширину левого и правого элементов, равную 500 пикселам, что точно соответствует ширине контейнера `div`. Поэтому, если мы добавим какие-то отступы, поля, границы или другие

элементы, у нас возникнут проблемы. Например, мы можем захотеть слегка отодвинуть текст в элементе p от края. Это потребует применения отступов, но если мы добавим к элементу отступ 20 пикселов, то получим что-то похожее на рис. 3.10.

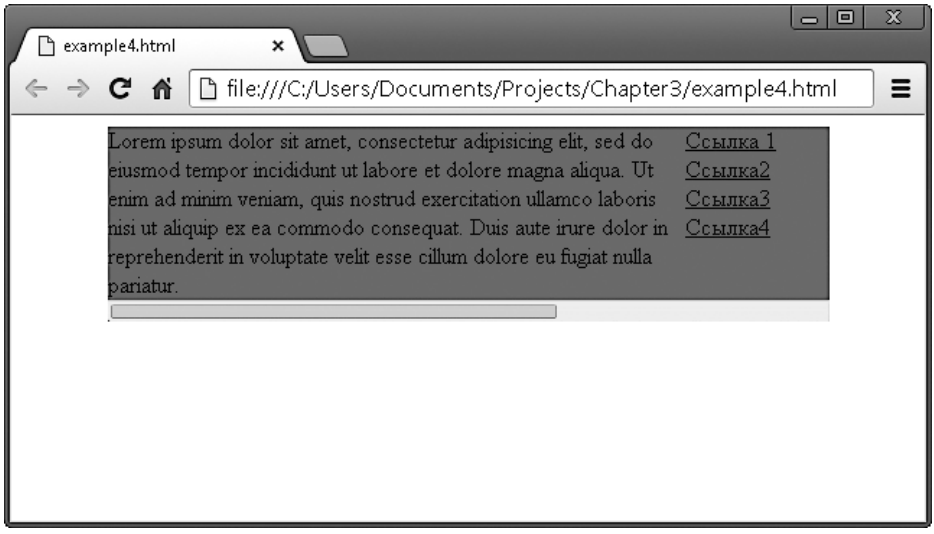


Рис. 3.9. Простая разметка в две колонки с использованием плавающих элементов

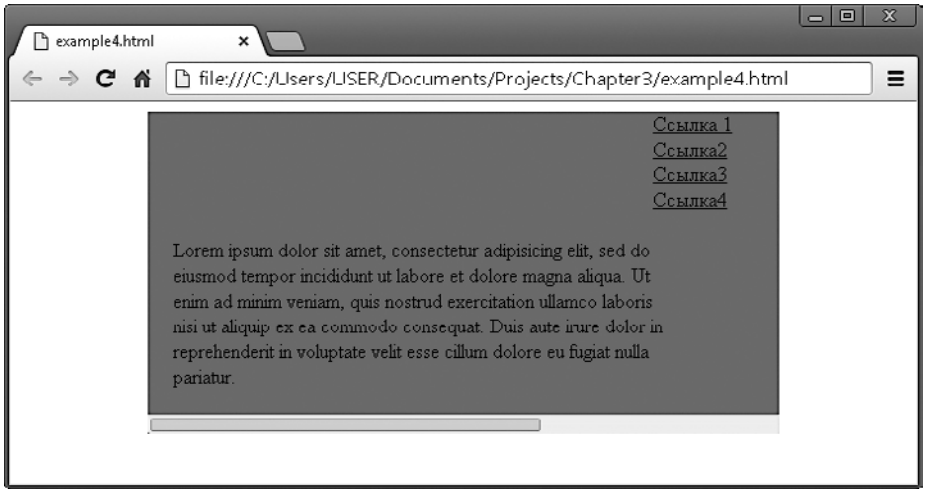


Рис. 3.10. Добавление отступов разбивает нашу разметку

Так произошло потому, что сумма пикселов в элементе `main` больше его ширины. Мы можем исправить это, уменьшив ширину с учетом двойной величины отступа (так как правый и левый отступы равны 10 пикселам). В результате CSS будет выглядеть так:

```
main {
  width: 500px;
  margin: auto;
  background: gray;
  overflow: auto;
}
nav {
  width: 100px;
  float: right;
}
p {
  margin: 0; /* убираем поля по умолчанию для элемента p */
  padding: 10px;
  float: left;
  width: 380px; /* 400 – 2*10 = 380 */
}
```

Тот же прием можно использовать при добавлении к элементам границы или полей ненулевой ширины.

Свойство clear

При создании разметки с помощью плавающих элементов можно столкнуться с интересной проблемой. Рассмотрим немного другой документ HTML. Сейчас наша цель — установить навигацию с левой стороны, а подвал оставить охватывающим обе колонки.

```
<body>
  <nav>
    <p><a href="link1">Ссылка 1</a></p>
    <p><a href="link2">Ссылка 2</a></p>
    <p><a href="link3">Ссылка 3</a></p>
    <p><a href="link4">Ссылка 4</a></p>
  </nav>
  <main>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
       eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
       ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
       aliquip ex ea commodo consequat. Duis aute irure dolor in
       reprehenderit in voluptate velit esse cillum dolore eu fugiat
       nulla pariatur.
    </p>
  </main>
  <footer>
    <p>Это подвал</p>
  </footer>
</body>
```

Следующий CSS-код отправляет элемент `nav` в правую часть страницы и обнуляет поля и отступы по умолчанию с помощью *универсального* селектора, который

выбирает все элементы DOM. Он также устанавливает для фона элемента разные оттенки серого цвета (рис. 3.11):

```
* {
  margin: 0;
  padding: 0;
}
nav {
  float: left;
  background:darkgray;
}
main {
  background:lightgray;
}
footer {
  background:gray;
}
```

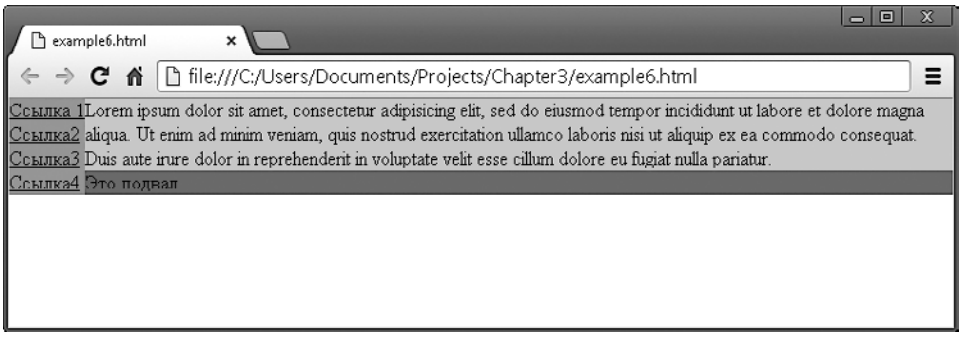


Рис. 3.11. Обратите внимание: подвал находится под секцией main, вместо того чтобы располагаться снизу всего содержимого страницы

По сути, подвал находится с правой стороны разметки, под главной секцией. А мы бы хотели, чтобы подвал был под обоими элементами. Вот для чего нужно свойство `clear`. Мы можем указать элементу `footer` расположение под плавающим элементом слева, справа или под обоими, левым и правым, элементами на странице, указав соответствующее значение свойства `clear`. Вот как можно изменить CSS для подвала:

```
footer {
  background:gray;
  clear: both; /* в данном случае можем использовать и 'clear: left'*/
}
```

Отображение получится похожим на рис. 3.12 — теперь подвал находится под обоими плавающими элементами, как мы и хотели.

Для начинающих плавающие элементы обычно являются самым трудным аспектом в CSS, поэтому я рекомендую вам потратить немного времени на эксперименты с созданием разметки.

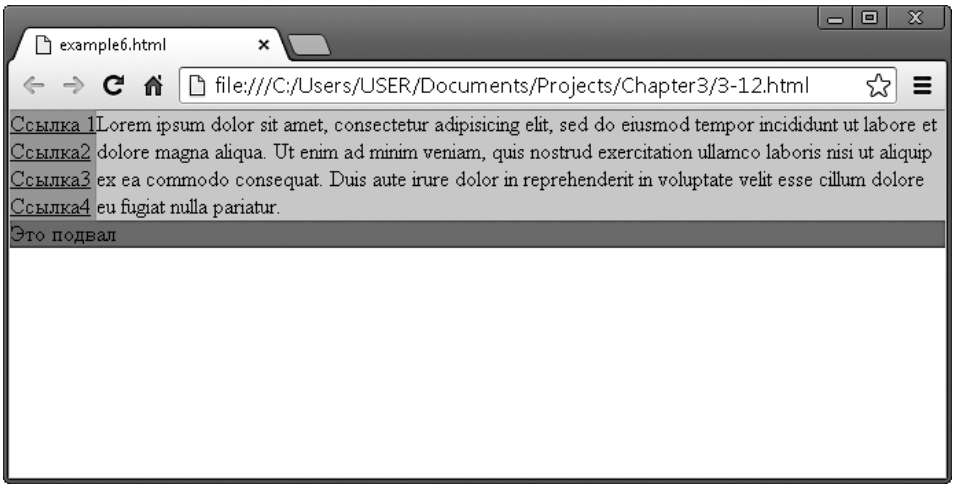


Рис. 3.12. Мы зафиксировали компоновку, установив свойство clear для подвала

Работа со шрифтами

В прошлом работа с нестандартными шрифтами на сайтах была очень проблематичной, потому что вы были ограничены теми из них, которые имелись на компьютерах ваших пользователей. Веб-шрифты значительно упростили эту задачу, так как вы можете запрашивать их прямо из Интернета, когда в них возникает необходимость. В Google сейчас хранится множество нестандартных шрифтов, использовать которые очень просто.

Вы можете начать с домашней страницы Google Fonts (рис. 3.13). Здесь мы видим длинный список шрифтов и сразу можем просмотреть их отображение.

В этом примере мы используем шрифт, который называется Denk One. Когда я зашел на домашнюю страницу Google Fonts, это был самый первый шрифт в списке. Чтобы использовать этот шрифт, нажмите кнопку Quick-use (Быстрое использование) — это одна из кнопок, расположенных слева от большой синей кнопки Add to Collection.

Нажав ее, мы увидим инструкции по использованию шрифта на нашей странице. Нужно добавить тег `<link>` в секцию `head` HTML. Если мы используем более ранний пример, то можем просто скопировать и вставить тег `<link>` со страницы быстрого использования или самостоятельно написать следующее:

```
<head>
  <title>Amazeriffic</title>
  <!-- Вставим пустую строку для удобства логического разделения -->
  <link href="http://fonts.googleapis.com/css?family=Denk+One"
        rel="stylesheet" type="text/css">
  <link href="style.css" rel="stylesheet" type="text/css">
</head>
```

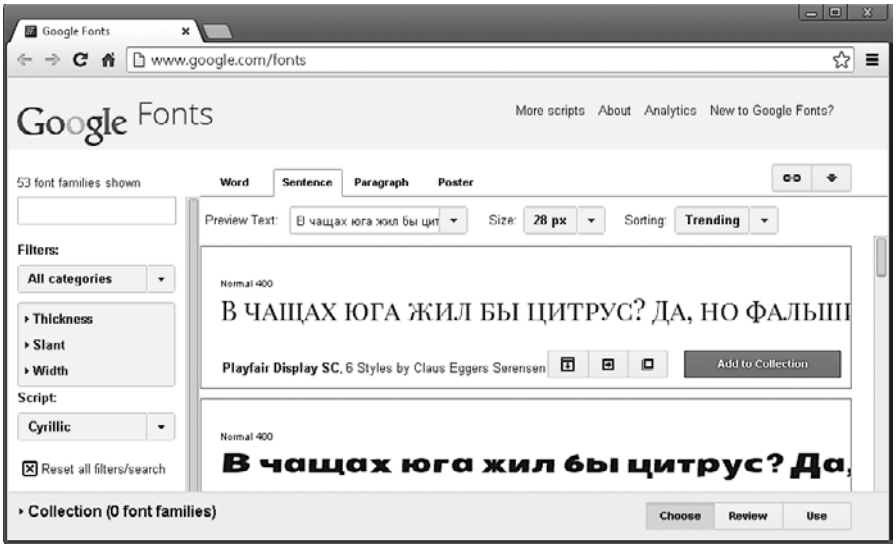


Рис. 3.13. Домашняя страница Google Fonts[1]

HTML допускает применение как одинарных ('), так и двойных (") кавычек для ограничения строк и значений. В примерах в этой книге я старался придерживаться использования двойных кавычек, но в скопированном и вставленном коде Google видим одинарные. Поскольку они эквивалентны друг другу, можете исправить их на двойные или оставить одинарными, значения это не имеет.

Таким образом этот веб-шрифт стал доступным на нашей странице. Теперь мы можем использовать его в файле CSS. Например, чтобы установить шрифт Denk One для элемента `h1`, надо добавить в набор правил свойство `font-family`:

```
h1 {
  font-family: 'Denk One', sans-serif;
}
```

Первая часть записи указывает желаемый шрифт (который стал доступным для применения с помощью тега `<link>`), а вторая — тот шрифт, который будет использоваться, если заданный нами окажется недоступным. То есть, по сути, эта запись говорит: «Применяй Denk One, если он доступен, а если нет, используй шрифт по умолчанию из семейства sans-serif, установленный в системе пользователя».

[1] На рисунке показаны шрифты, выбранные с применением фильтра Script:Cyrillic (находится на панели слева, значение по умолчанию — Latin). Большинство кириллических шрифтов, разумеется, поддерживают латинский алфавит, но, к сожалению, не наоборот. — *Примеч. пер.*

Но если отображать шрифты и изменять их цвет (с помощью свойства color соответствующего элемента) очень просто, то как быть с их размером? В прошлом существовало несколько способов это сделать, но в итоге остался один способ, признанный всеми лучшим.

В первую очередь нужно учесть, что пользователь может изменять шрифт по умолчанию в браузере и наши страницы должны позволять это сделать. Таким образом, мы можем установить основной размер шрифта в наборе правил для элемента body документа, а затем масштабировать все шрифты относительно его. К счастью, с помощью CSS сделать это очень просто. Для указания длины нужно использовать единицы em:

```
body {
  font-size: 100%; /* устанавливаем базовый размер шрифта для всего документа */
}
h1 {
  font-size: xx-large; /* установлен относительно базового размера шрифта */
}
h2 {
  font-size: x-large;
}
.important {
  font-size: larger; /* делает шрифт несколько больше, чем в родительском элементе
*/
}
.onePointTwo {
  font-size: 1.2em; /* устанавливает его в 1,2 раза больше базового размера */
}
```

В этом примере мы устанавливаем базовый размер шрифта равным 100 % размера, установленного в браузере пользователя, а затем масштабируем относительно его остальные шрифты. Можно использовать абсолютные величины CSS xx-large или x-large (а также аналогично x-small или xx-small), которые соответственно будут изменять масштаб. Аналогично можем использовать относительные размеры, larger или smaller, чтобы сделать размер шрифта больше или меньше для данного содержания.

Если требуется более точно определенное управление размерами шрифтов, мы используем величины em (что значит метаединицы) — множитель, на который умножается текущий размер шрифта. Например, если базовый размер шрифта элемента body равен, скажем, 12 пунктов, то установление для какого-то другого элемента размера шрифта 1,5 em обеспечит точный размер 18 пунктов. Это очень полезно, так как шрифты масштабируются соответственно базовому размеру шрифта. Благодаря этому мы можем поменять размер всех шрифтов на странице, изменяя размер одного только базового шрифта. Это значительно упрощает работу с сайтом для слабовидящих людей: они часто устанавливают больший размер базового шрифта в браузере, используя таким образом em вместо абсолютных величин, что может нарушить разметку страницы.

В предыдущем примере мы установили размер шрифта в абзаце в 1,2 раза больше базового шрифта. На рис. 3.14 показано, как данный набор правил определяет стили в документе.

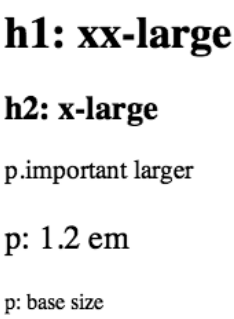


Рис. 3.14. Примеры шрифтов на основе упомянутой ранее таблицы стилей

Устранение браузерной несовместимости

С одним спорным инструментом, который называется сбросом CSS, вы будете сталкиваться довольно часто.

Вспомните «плавающий» пример, когда мы убирали поля по умолчанию и отступы в тегах абзацев? Оказывается, разные браузеры имеют разные базовые настройки CSS, в результате чего отображение стиля в этих браузерах может несколько различаться. Сброс CSS (CSS reset) был разработан для удаления всех браузерных настроек по умолчанию. Самый известный инструмент создал Эрик Мейер.

Почему я назвал сброс CSS спорным? По нескольким причинам. Одна из них — веб-доступность. Сброс настроек может, к примеру, вызывать проблемы у людей, которые управляют навигацией с помощью клавиатуры. Еще одна причина — производительность. Поскольку в инструментах часто используется универсальный селектор (*), они могут серьезно перегружать сайт. И наконец, эти инструменты означают огромное количество лишней работы, потому что на деле довольно часто браузерные настройки полностью соответствуют вашим нуждам.

В то же время я думаю, что сброс — отличный инструмент с точки зрения педагогики, вот почему я рассказываю о нем здесь. Он заставляет новичков в точности указывать, какими они хотят видеть отдельные аспекты страницы, а не полагаться на значения в браузере по умолчанию. Поскольку я предполагаю, что вы являетесь новичком в CSS, так как читаете сейчас эту книгу, то рекомендую вам поупражняться со сбросами.

Чтобы сделать объяснение более понятным, я включу инструмент сброса в отдельную таблицу стилей и добавлю дополнительный элемент `link` в мой HTML. Мы можем скопировать сброс Эрика Мейера в отдельный файл `reset.css` и связать его с нашим HTML:

```
<head>
  <title>Пример 8 – Использование сброса CSS</title>
  <link rel="stylesheet" type="text/css" href="reset.css">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

Увидеть, к чему это приведет, очень просто. Рассмотрим следующий HTML:

```
<body>
  <h1>Это заголовок</h1>
  <h3>Это менее важный заголовок</h3>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
     do eiusmod tempor incididunt ut labore et dolore magna aliqua.
     Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
     nisi ut aliquip ex ea commodo consequat.</p>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
     do eiusmod tempor incididunt ut labore et dolore magna aliqua.
     Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
     nisi ut aliquip ex ea commodo consequat.</p>
</body>
```

На рис. 3.15 показано, как страница отображается в Chrome при применении только стилей браузера.

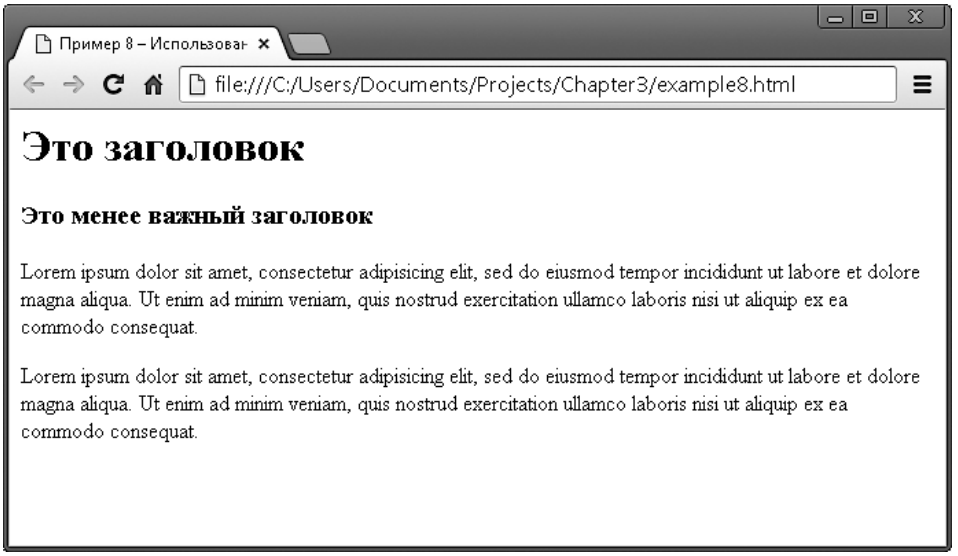


Рис. 3.15. Простая страница со стилем по умолчанию

Обратите внимание на то, что шрифт и поля элементов `h1` и `h3` уже заданы. У элемента `p` тоже изменены поля.

А сейчас сделаем сброс. Я скопировал свой файл сброса со страницы Эрика Мейера и сохранил его в файле `reset.css` в той же самой папке, что и мой HTML. Я также изменил тег `<head>` в HTML следующим образом:

```
<head>
  <title>Пример 8 – Использование сброса CSS</title>
  <link href="reset.css" rel="stylesheet" type="text/css">
</head>
```

Сейчас, перезагрузив страницу, мы увидим нечто похожее на рис. 3.16.

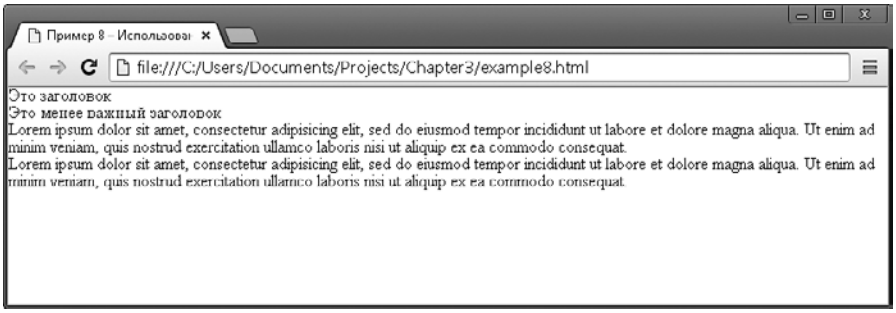


Рис. 3.16. Простая страница, где с помощью сброса удалены все стили по умолчанию

Поля, отступы и настройки шрифта по умолчанию были полностью удалены со страницы. Конечно, мы не собираемся оставлять сайт в таком виде, но запомните главное: включение сброса CSS полностью удаляет стили, использованные в браузере по умолчанию. Благодаря этому те правила, которые мы разрабатываем и сохраняем в своих CSS-файлах, всегда будут приводить к одному и тому же результату, независимо от того, какие настройки (и, следовательно, соответствующий набор правил по умолчанию) сделаны в браузере пользователя.

Использование CSS Lint для выявления возможных проблем

Как и в случае HTML, очень неплохо иметь инструмент, который поможет нам выявлять потенциальные проблемы с CSS. Например, можете ли вы заметить ошибку в этом примере?

```
body {
  background: lightblue;
  width: 855px;
  margin: auto;
  h1 {
    background: black;
    color: white;
  }
  p {
    color: red;
    margin: 10px
    padding: 20px;
  }
```

Если вы нашли ее, получите тройку с плюсом, потому что в этом примере целых две ошибки! Думаю, отсутствие закрывающей скобки после набора правил для `body` бросается в глаза сразу. Но есть и еще одна.

Вы, наверное, нашли ее, посмотрев внимательно еще раз. Но если нет, обратите внимание на набор правил для элемента `p`. После свойства `margin` пропущена точка с запятой.

Как и HTML-валидатор, CSS Lint — это инструмент, который может помочь выявить потенциальные проблемы с кодом. Как и онлайн-валидатор W3C HTML, он не требует установки на компьютер какого-либо программного обеспечения. Просто зайдите на страницу http://csslint.net (она показана на рис. 3.17), а затем скопируйте и вставьте туда CSS.

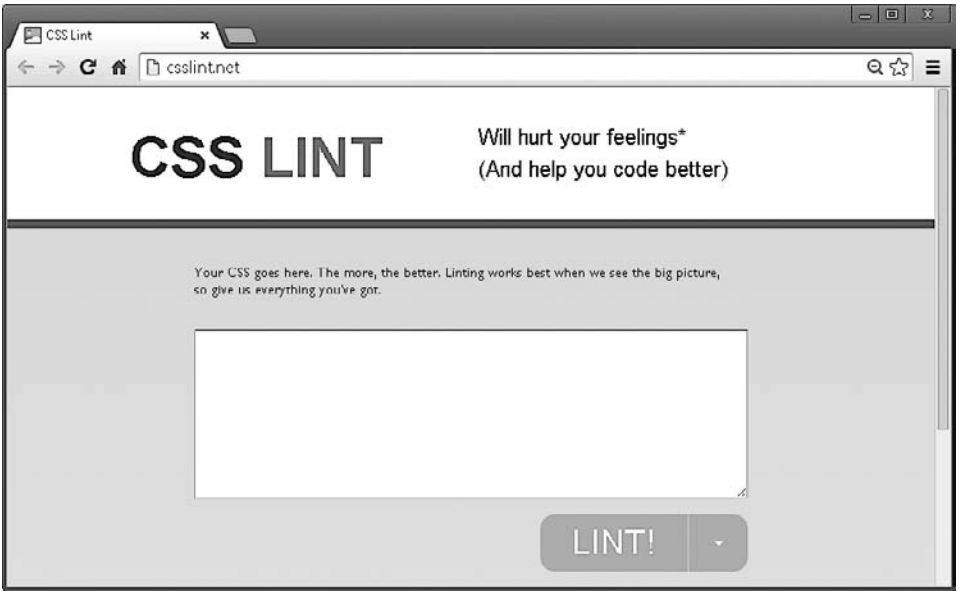


Рис. 3.17. Домашняя страница CSS Lint

CSS Lint дает несколько больше предупреждений, чем инструмент для HTML, изученный нами в главе 2, но здесь есть опции, позволяющие настроить уровень предупреждений. Вы можете увидеть эти опции, щелкнув на стрелке вниз справа на большой кнопке LINT! на главной странице. Опции разбиты на категории согласно причинам предупреждений. Например, опция Disallow universal selector (Запрещать универсальные селекторы) находится в категории Performance (Производительность). Это значит, что если вы готовы допустить чуть более медленную загрузку вашей страницы ради какой-то особенной функциональности CSS, то отключаете эту опцию и предупреждаете Lint об этом.

Обычно я оставляю все опции в CSS Lint выбранными, но вы можете настроить и свои варианты. Я бы рекомендовал не выключать ни одну из них, а, получив предупреждение, уделить немного времени поиску в Google, чтобы разобраться

в причине. Ведь это почти то же самое, что получить консультацию у настоящего эксперта по CSS!

Взаимодействие и решение проблем с Chrome Developer Tools

В абсолютно любом программном обеспечении очень часто наблюдается ситуация, когда все идет совсем не так, как надо. При использовании CSS это чаще всего означает, что страница выглядит в браузере совершенно иначе, чем вы ожидали, а в чем проблема — непонятно. К счастью, браузер Chrome включает в себя отличный набор инструментов, которые помогут решить проблемы как с CSS, так и с HTML.

Начнем с открытия файла `margin_border_padding.html` в Chrome. Затем откройте меню View, выберите подменю Developer и щелкните на опции Developer Tools[1]. В результате внизу страницы откроются Инструменты разработчика Chrome. Если вкладка Elements вверху панели неактивна, щелкните на ней — таким образом вы увидите нечто похожее на рис. 3.18.

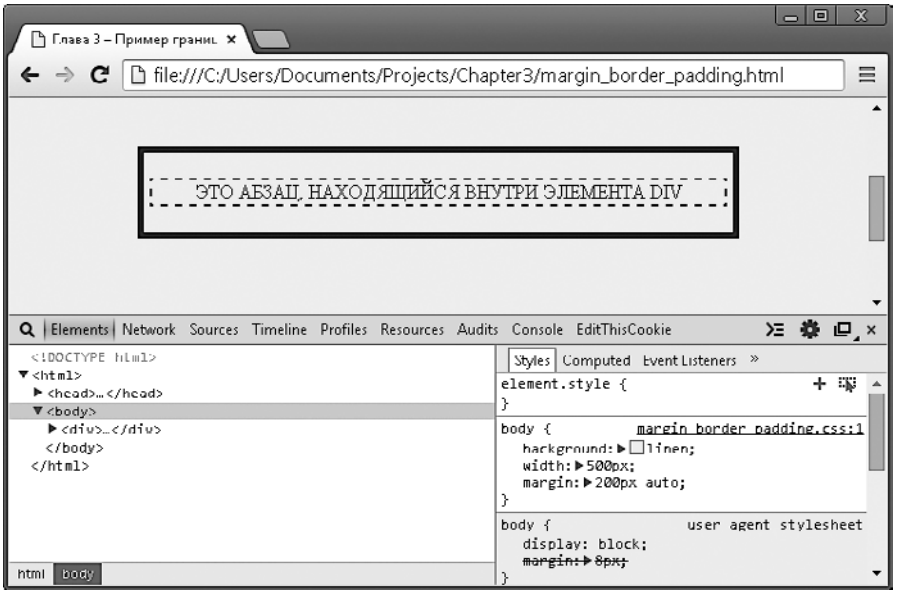


Рис. 3.18. Инструменты разработчика Chrome, открытые для примера margin_border_padding.html

[1] На сегодняшний момент: кнопка панели Chrome в правом верхнем углу ▸ Tools (Инструменты) ▸ Developer Tools (Инструменты разработчика) или сочетание клавиш Ctrl+Shift+I. — *Примеч. пер.*

С левой стороны вы видите какой-то фрагмент HTML-кода со стрелочками, которые могут сворачивать или разворачивать дочерние элементы. Если стрелка направлена вправо, это значит, что вы можете щелкнуть на ней, чтобы показать находящийся внутри нее HTML. Аналогично, если она направлена вниз, можете щелкнуть на ней, чтобы его скрыть.

Наводя мышь на HTML-теги, вы увидите, что соответствующие им отображаемые элементы подсвечиваются на странице. Можете выбрать определенный элемент HTML, щелкнув на его открывающем теге, и соответствующая ему строка подсветится голубым.

HTML, который вы видите на вкладке Elements, может точно или не совсем точно соответствовать HTML, который находится в вашем файле, но, если даже совпадение неполное, это именно тот HTML-код, который отображает браузер. Как я уже упоминал в главе 2, в некоторых случаях браузер совершенствует ваш код самостоятельно (например, закрывая теги), и HTML, отображаемый на вкладке Elements, содержит все предположения, сделанные Chrome. Например, если вы создадите HTML-документ, содержащий только тег <p> без <html>, <header> и <body>, Chrome допишет последние самостоятельно и покажет их вам на вкладке Elements (рис. 3.19). Помните об этом, если когда-нибудь вас озадачит разница между тем, что вы имели в виду по мнению Chrome, и тем, чего хотели на самом деле.

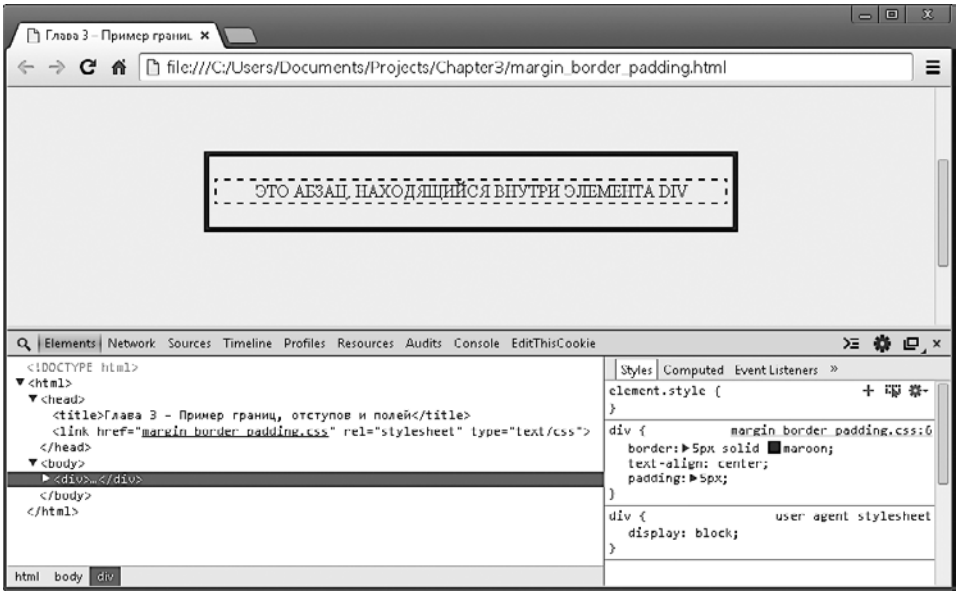


Рис. 3.19. Главный элемент div выделен, и вы можете видеть соответствующий ему набор правил

С правой стороны окна вы можете видеть CSS, связанный с выделенным HTML-элементом. Попробуйте выделить элемент `div`, в котором находится абзац. Справа на вкладке Styles вы увидите набор правил, связанный с этим элементом, а наведя на него указатель мыши, можно отобразить флажки, появляющиеся слева от

каждого из правил. Вы можете снять их, отменив таким образом какое-либо правило. Если возникли какие-то проблемы с вашим CSS, очень часто полезно посмотреть на эту вкладку, чтобы убедиться, что ожидаемые вами правила действительно применяются. Вы можете посмотреть, как правила отображаются на панели разработчика Chrome, на рис. 3.19.

Но это еще не все! Вы можете напрямую манипулировать правилами, щелкнув на каком-то из них и введя для него новое значение. Вы можете даже добавить новые правила прямо здесь, но помните, что изменения не отразятся на вашем CSS-файле. Так что, если вы найдете решение для применения стиля с помощью этого окна, вам нужно вернуться в Sublime и включить этот стиль в CSS.

Вы, наверное, уже пробовали манипулировать с отступами, границами и полями, изменяя код. Попробуйте проделать аналогичные действия на вкладке Elements на панели разработчика. Вы увидите, что намного проще попробовать что-то и тут же наблюдать визуальный эффект. Но помните, что после обновления страницы все изменения будут потеряны и Chrome снова будет отображать страницу, соответствующую вашим HTML- и CSS-файлам.

Стилизуем Amazeriffic!

А сейчас изучим все это на примере. В предыдущей главе мы составили структуру для домашней страницы приложения, которое называлось Amazeriffic. Финальный вариант HTML, определяющего структуру страницы, выглядел так:

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
    <header>
      <h1>Amazeriffic</h1>
      <nav>
        <a href="#">Войти</a> |
        <a href="#">ВиО</a> |
        <a href="#">Техподдержка</a>
      </nav>
    </header>
    <main>
      <h2>Amazeriffic изменит вашу жизнь!</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
         eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
      <h3>Почему вам нужен Amazeriffic?</h3>
      <ul>
        <li>Он легко впишется в вашу жизнь</li>
        <li>Он классный</li>
        <li>Он перевернет ваш мир</li>
      </ul>
      <img src="lightbulb.png" alt="Изображение лампочки">
```

```
    </main>
    <footer>
      <div class="contact">
        <h5>Свяжитесь с нами</h5>
        <p>Amazeriffic!</p>
        <p>555 50-я улица</p>
        <p>Эшвилл, Северная Каролина 28801</p>
      </div>
      <div class="sitemap">
        <h5>Карта сайта</h5>
        <ul>
          <li><a href="#">Домой</a></li>
          <li><a href="#">О нас</a></li>
          <li><a href="#">Право</a></li>
          <li><a href="#">Техподдержка</a></li>
          <li><a href="#">ВиО</a></li>
          <li><a href="#">Вакансии</a></li>
        </ul>
      </div>
    </footer>
</body>
</html>
```

Сначала мы должны зайти в каталог, где располагается проект Git для Amazeriffic. Если мы находимся в домашней папке, то можем перейти прямо в каталог Amazeriffic с помощью команды cd и указания полного пути в качестве аргумента:

```
hostname $ cd Projects/Amazeriffic
```

Можем использовать ls, чтобы увидеть содержимое папки, а затем выполнить git log, чтобы увидеть историю коммитов, созданную в главе 2. Моя выглядит примерно так:

```
hostname $ ls
index.html
hostname $ git log
commit efeb5a9a5f80d861119f5761df789f6bde0cda4f
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 1:41:52 2013 -0400
    Add content and structure to footer
commit 09a6ea9730521ed1effd135a243723a2745d3dc5
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 12:32:17 2013 -0400
    Add content to main section
commit f90c9a6bd896d1a303f6c3647a7475d6de9c4f9e
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 11:45:21 2013 -0400
    Add content to header section
commit 1c808e2752d824d815929cb7c170a04267416c04
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:36:47 2013 -0400
    Add skeleton of structure to Amazeriffic
commit 147deb5dbb3c935525f351a1154b35cb5b2af824
```

```
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:35:43 2013 -0400
    Initial commit[1]
```

Затем мы можем открыть папку и создать два новых CSS-файла. Сначала создадим файл `reset.css`, основанный на сбросе HTML Эрика Мейера. Вы можете набрать код (как описано ранее) или скопировать его с сайта Эрика. Создадим также файл `style.css`, где будет находиться наш собственный CSS.

Прежде чем отправить коммит, сразу свяжем наши CSS-файлы с файлом HTML, добавив в заголовок две ссылки:

```
<head>
  <title>Amazeriffic</title>
  <link rel="stylesheet" type="text/css" href="reset.css">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

Сейчас мы можем открыть этот файл в Chrome. Он будет полностью лишен стилей из-за наличия файла сброса. Мой выглядит так, как показано на рис. 3.20.

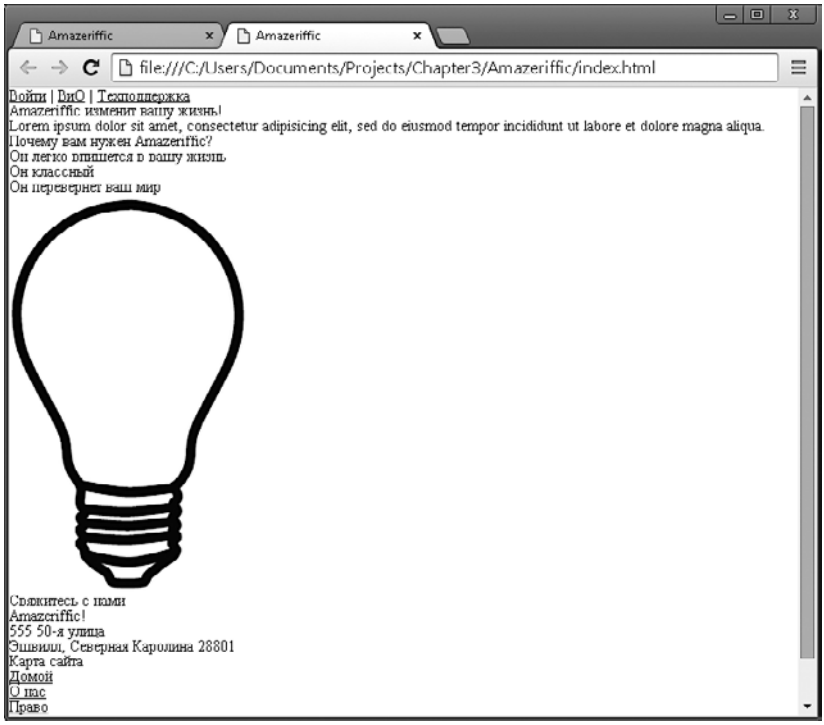


Рис. 3.20. Финальный HTML Amazeriffic, включающий сброс CSS

[1] Сообщения о коммитах сверху вниз: «Добавлены содержание и структура подвала», «Добавлено содержание секции main», «Добавлено содержание секции header», «Добавлен скелет структуры Amazeriffic», «Начальный коммит». — *Примеч. пер.*

А сейчас отправим изменения в хранилище Git:

```
hostname $ git add reset.css
hostname $ git add style.css
hostname $ git add index.html
hostname $ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified: index.html
# new file: reset.css
# new file: style.css
#
hostname $ git commit -m "Add reset.css and style.css, link in index.html"
[master b9d4bc9] Add reset.cssand style.css, link in index.html
3 files changed, 142 insertions(+), 2 deletions(-)
create mode 100644 reset.css
create mode 100644 style.css
```

Сетка

Исследуем стиль для Amazeriffic. Как вы можете видеть, содержимое страницы разделено на две колонки. Первая занимает около двух третей области содержания, а вторая — только одну треть. Затем обратите внимание на то, что содержание разбито по вертикали примерно по три строки. Это определяет стилистическую сетку для содержания, которая является одним из базовых подходов при разработке компоновки страницы.

Еще один аспект, который нелегко продемонстрировать в книге, — это дизайн фиксированной ширины. Это значит, что при изменении ширины окна браузера содержимое страницы остается выровненным по центру и сохраняет горизонтальный размер. На рис. 3.21 и 3.22 вы можете видеть Amazeriffiic в окнах браузера разного размера.

Это значит, что его дизайн не является *адаптивным* — так называется современный подход к верстке CSS, обеспечивающий перестроение макета в зависимости от ширины окна браузера. Это важно при веб-разработке, так как многие люди будут просматривать ваш сайт на мобильном телефоне или планшете. Мы не будем заниматься этим сейчас, но вы можете испробовать несколько инструментов, которые упрощают разработку адаптивного дизайна, выполнив упражнения в конце этой главы.

Наша цель — создать фиксированный дизайн в две колонки с помощью CSS. Для начала нужно, чтобы строки (в данном случае это элементы `header`, `main` и `footer`) имели фиксированную ширину — значение свойства `width`. Первое, что необходимо сделать, — создать наборы правил для этих элементов. Добавим в файл `style.css` набор правил, который выглядит следующим образом:

```
header, main, footer {
  min-width: 855px;
}
```

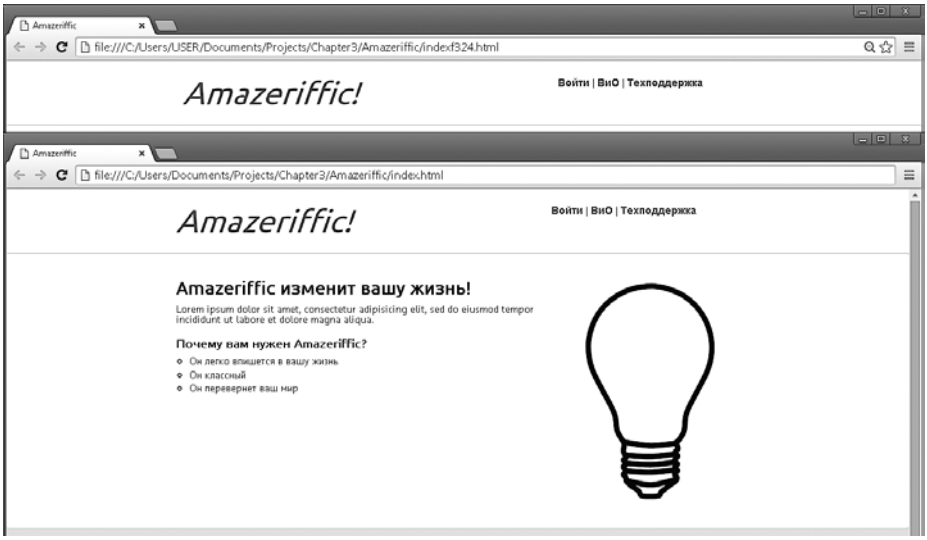


Рис. 3.21. Amazeriffic в окне браузера 1250 × 650

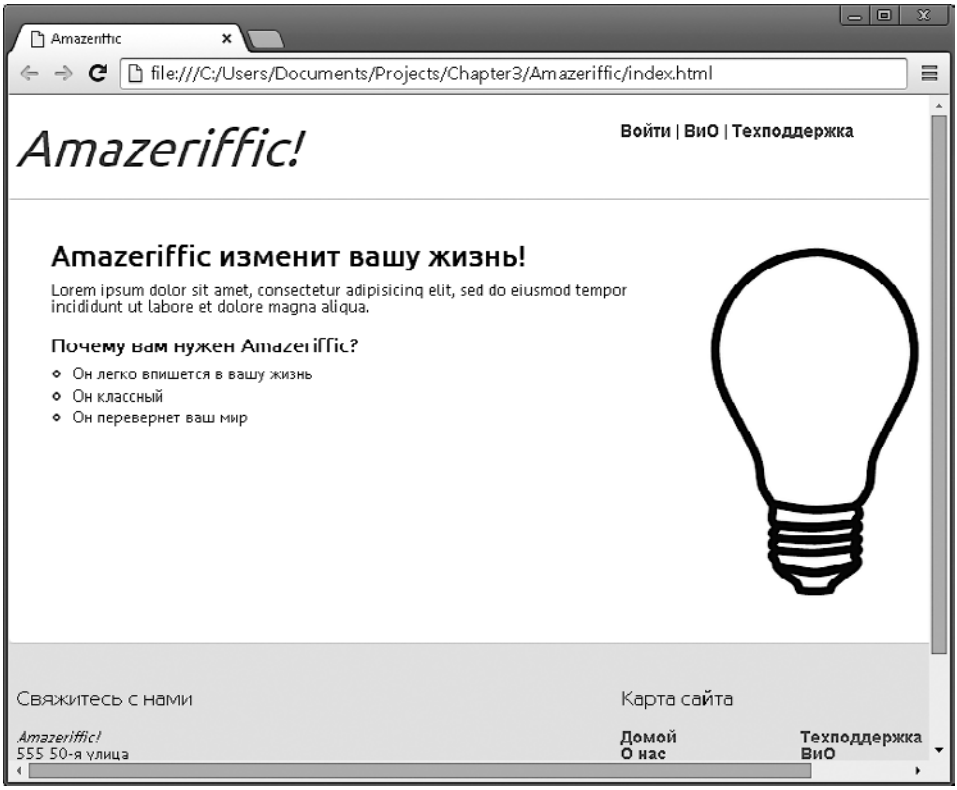


Рис. 3.22. Amazeriffic в окне браузера 800 × 575

Свойство min-width означает, что, если ширина окна браузера становится равной 855 пикселам или меньше, остальная часть страницы будет скрыта. Проверьте это.

Затем обратите внимание на то, что область подвала — желтого цвета. Он заполняет нижнюю часть страницы по мере того, как окно браузера вытягивается вертикально. Это значит, что желтый цвет должен быть фоновым цветом всей страницы, но нам придется изменить цвета элементов header и main позднее.

Мы можем добавить правило для body, которое устанавливает фоновое изображение желтого цвета. Отмечу, что, хотя и установил предыдущий набор правил первым, я добавил набор правил для body в файл style.css выше его. Это объясняется тем, что body расположен в HTML-файле до элементов header, main, footer. Если сохранять этот порядок, становится проще найти проблемы и произвести изменения:

```
body {
  background: #f9e933;
}
header, main, footer {
  min-width: 855px;
}
```

Сейчас мы видим, что вся страница стала желтого цвета, так что нужно изменить фоновый цвет элементов header и main на белый. Чтобы сделать это, мы должны добавить два новых набора правил для элементов header и main в дополнение к тому, который содержит свойство min-width. Разделим их, так как далее придется добавить отдельные стилистические правила в каждый набор, как мы увидим позднее:

```
header {
  background: white;
}
main {
  background: white;
}
```

В результате сейчас мы можем увидеть нечто похожее на рис. 3.23.

На момент написания книги тег <main> может отображаться в браузере Safari не совсем корректно. Если вы столкнулись с проблемами при использовании этого браузера, попробуйте добавить набор правил для элемента main, включающий правило display: block;.

А сейчас, когда у нас уже заданы какие-то базовые стили, было бы неплохо добавить эти изменения в хранилище Git. Начните с проверки статуса. Единственный поменявшийся файл — style.css. Добавьте этот файл для фиксации с помощью команды add, а затем отправьте коммит с поясняющим сообщением.

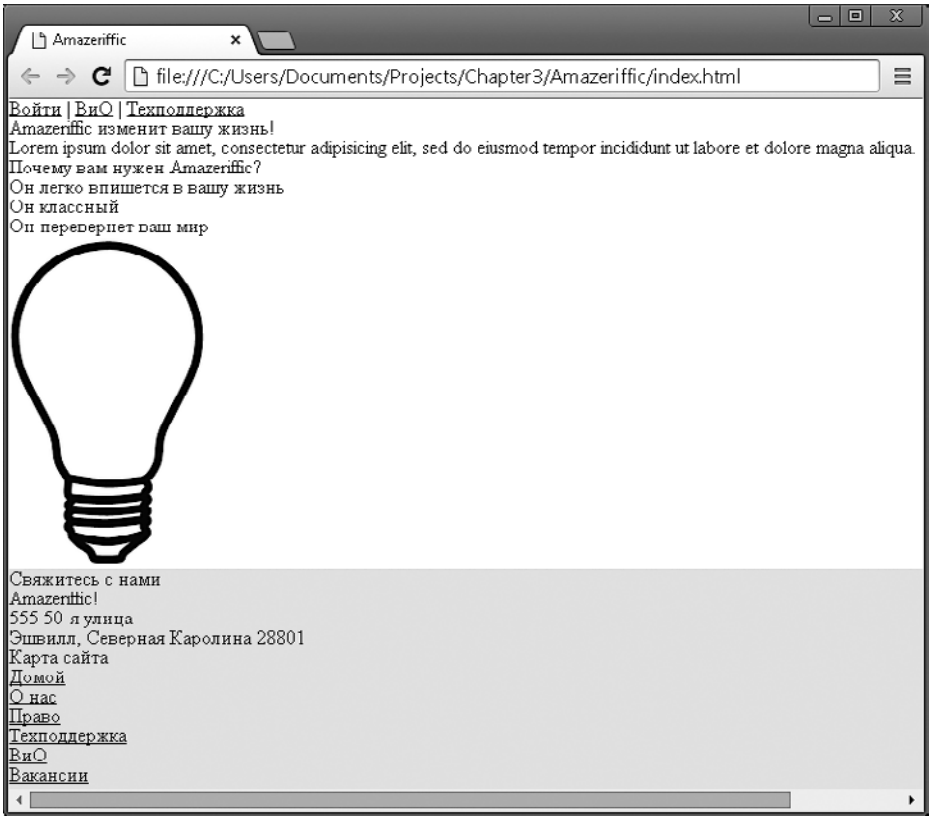


Рис. 3.23. Amazeriffic с (очень) начальными стилями

Чего нам хотелось бы дальше? Мы собирались выровнять содержимое страницы по центру и зафиксировать ширину 855 пикселов. Это значение объясняется тем, что нам требуется число, делимое на 3 (так как колонки имеют соотношение 2:1 по ширине). Поскольку 855 / 3 = 285, это дает возможность работать с целыми числами.

Вот как можно решить эту задачку. Мы можем добавить правило `max-width` в набор стилей `main`, `header` и `footer`, установить ширину элемента `body` 855 пикселов, а затем установить для поля элемента `body` значение `auto`. Прежде чем двигаться дальше, попробуйте сделать это и посмотрите, что получится.

Если вы действительно попробовали, то, наверное, заметили, что области заголовка и основной части сейчас не растягиваются на полную ширину страницы, из-за чего цвет фона заполняет области слева и справа. Это не совсем то, что нам нужно.

Как это исправить? Сейчас как раз тот случай, когда нужно добавить в DOM элемент `div`, чтобы задавать стили элементов независимо друг от друга. Мы хотим, чтобы элементы `header` и `main` занимали всю ширину страницы, но их содержимое оставалось в пределах 855 пикселов. Модифицируем HTML так, чтобы в элементах

header, main и footer содержался класс под названием conteiner, где находились бы элементы содержания. Например, вот какой div мы можем сделать для элемента header:

```
<header>
  <div class="container">
    <h1>Amazeriffic</h1>
    <nav>
      <a href="#">Войти</a> |
      <a href="#">ВиО</a> |
      <a href="#">Техподдержка</a>
    </nav>
  </div>
</header>
```

Таким образом, сейчас мы можем задать для этого контейнера максимальную ширину 855 пикселов и автоматическую ширину полей. Когда закончим, набор правил будет выглядеть так:

```
body {
  background: #f9e933;
}
header, main, footer {
  min-width: 855px;
}
header .container, main .container, footer .container {
  max-width: 855px;
  margin: auto;
}
header {
  background: white;
}
  main {
  background: white;
}
```

Сейчас, если мы изменим ширину окна браузера, ширина содержимого страницы окажется неизменной! Так что в этот момент очень полезно будет вернуться к командной строке и отправить изменения в хранилище Git. В этот раз мы изменили два файла, поэтому оба нужно добавить в хранилище Git перед коммитом.

Создание колонок

Следующее, что нам нужно, — создать колонки для содержимого элементов. Чтобы сделать это, прикрепим содержание, которое должно находиться справа, к правому краю страницы и укажем значение ширины для правой колонки 285 пикселов, а для левой — 570 пикселов. Добавим также свойство overflow со значением auto, чтобы элемент не сжимался до минимума, когда мы установим свойство flow для обоих дочерних элементов.

Закончив редактировать наборы правил, мы увидим нечто вроде этого:

```
header {
  background: white;
  overflow: auto;
}
header h1 {
  float: left;
  width: 570px;
}
header nav {
  float: right;
  width: 285px;
}
```

Элемент `header` несколько выделяется из-за того, что содержит только два дочерних элемента — `h1` и `nav`. Но для элементов `footer` и `main` это не так. Если вы понимаете все правильно, то, наверное, думаете, что стоило бы добавить два дополнительных элемента `div` к ним обоим, чтобы разбить плавающие левую и правую части на два отдельных элемента. А в правой части подвала нужно будет создать еще одну подразметку для оформления карты сайта. Все это соответствует дизайну — и вы должны быть способны сделать это, изучив материал этой главы.

Я бы создавал какую-то одну секцию за один раз: добился верного вида секции `header`, затем — основной области, а затем и подвала. Вы можете каждый раз делать коммиты в хранилище Git. Я бы рекомендовал также периодически прогонять код через CSS Lint (и HTML-валидатор, если вы также редактируете HTML), чтобы убедиться в отсутствии потенциальных проблем.

Закончив с этими секциями, вы увидите нечто похожее на рис. 3.24.

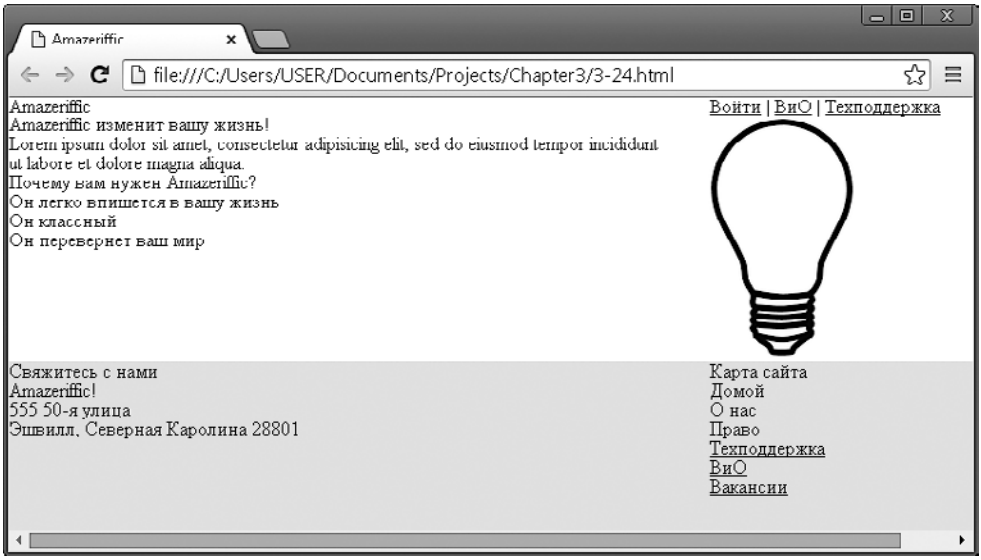


Рис. 3.24. Более стильный Amazeriffic

Если страница выглядит так же, как на рисунке, и вы выявили все возможные проблемы, пропустив CSS через сервис CSS Lint, то лучше всего сейчас сделать новый коммит в хранилище Git.

Добавление шрифтов и управление ими

Следующим шагом будет добавление двух шрифтов Google. Для этого примера я использовал шрифты Ubuntu и Droid Sans. Все теги заголовков (`<h1>`, `<h2>`, `<h3>` и `<h4>`) выводятся шрифтом Ubuntu, а остальная часть контента — шрифтом Droid Sans. Для этого нам понадобится добавить два тега `link` в HTML-файл и несколько правил — в CSS.

После установления шрифтов отправьте изменения в хранилище Git. Сделав это, установите базовый размер шрифта 100 %, а затем модифицируйте размеры шрифтов элементов так, чтобы они наиболее точно соответствовали оригинальному макету, приведенному на иллюстрации.

После того как размеры шрифтов будут выставлены корректно, провалидируйте HTML и пропустите CSS через сервис Lint, чтобы установить возможные проблемы. Сделав это, отправьте изменения в хранилище Git.

Еще несколько изменений

И наконец, нужно добавить поля и отступы, чтобы элементы отделялись друг от друга на странице. Это потребует экспериментов с полями и отступами для разных элементов. Я рекомендую проделать это с помощью Chrome Developer Tools. Вы также должны задать стили для имеющихся на странице ссылок. Все ссылки красные и не подчеркиваются, пока на них не наведен курсор. Это потребует от вас манипуляций с псевдоклассом `hover` для элементов `a`.

Подведем итоги

В этой главе мы использовали CSS, чтобы добавить стили к структуре пользовательского интерфейса и содержания. Основная идея CSS состоит в том, что мы связываем стилистические *наборы правил* с элементами DOM. Большинство из этих наборов правил *наследуются*: если мы добавляем набор правил для какого-либо элемента DOM, то они применяются также для всех его дочерних элементов, кроме случаев перекрытия набором правил, указанным для какого-либо определенного элемента. В случаях неопределенности работает принцип каскадных правил.

Вы можете указать наборы правил CSS строго для определенных элементов DOM с помощью CSS-селекторов. Работа с селекторами для отдельных элементов или их групп в DOM требует некоторой практики.

Мы можем выполнить разметку основы документа по сетке с помощью свойств CSS `float`. Можем закреплять элементы CSS с правого или левого края, извлекая

их из обычного последовательного построения. Если у элемента есть два дочерних элемента и мы закрепляем один дочерний элемент слева, а другой — справа, они будут отображаться рядом, если не перекрываются. Мы можем добиться этого, правильно указав ширину элементов.

CSS Lint — инструмент, который помогает нам избежать случайных ошибок или опечаток в CSS. Кроме того, богатый набор интерактивных инструментов встроен в Chrome.

Больше теории и практики

В Amazeriffic осталось еще несколько проблем, которые нужно исправить. Я рекомендую вам сделать это, чтобы добиться наиболее полного совпадения с макетом.

Заучивание

Добавим еще несколько шагов в упражнения. Помните, что ваша цель — *запомнить* эти шаги вместе с теми, которые перечислены в предыдущих главах.

1. Создайте простой файл `style.css`, в котором с помощью универсального селектора обнуляются поля и отступы для всех элементов, а фон страницы установлен светло-серым.
2. Свяжите файл `style.css` с вашим HTML-файлом в теге `<head>` и удостоверьтесь, что все корректно работает, обновив страницу.
3. Зафиксируйте изменения файлов `style.css` и `index.html` в хранилище Git.
4. Добавьте некоторые базовые стилевые правила для элементов `header`, `main` и `footer` и удостоверьтесь, что все корректно работает, обновив страницу.
5. Зафиксируйте изменения в новом коммите.

Упражнения в CSS-селекторах

Наберите следующий документ HTML и сохраните его в файле с названием `selectorpractice.html`:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Hi</h1>
    <h2 class="important">Снова привет</h2>
    <p class="a">Случайный несвязанный абзац</p>
    <div class="relevant">
      <p class="a">первый</p>
      <p class="a">второй</p>
      <p>третий</p>
```

```
      <p>четвертый</p>
      <p class="a">пятый</p>
      <p class="a">шестой</p>
      <p>seventh</p>
    </div>
  </body>
</html>
```

Затем присоедините к нему файл CSS с помощью тега `link`. В CSS добавьте единственный селектор:

```
* {
  color: red;
}
```

Это изменит цвет текста всех элементов на красный. Мы используем этот файл, чтобы практиковаться в применении CSS-селекторов. Заменяйте символ * селектором для элемента из следующего списка и проверяйте, что красными становятся только выбранные элементы:

- элемент `<h1>`;
- элемент `<h2>` через его класс;
- все абзацы класса `relevant`;
- первый абзац из `relevant`;
- третий абзац из `relevant`;
- седьмой абзац из `relevant`;
- все абзацы на странице;
- только случайный несвязанный абзац;
- все абзацы класса `a`;
- только абзацы `relevant` класса `a`;
- второй и четвертый абзацы (подсказка: используйте запятую).

В документации разработчика Mozilla (Mozilla developer documentation) содержится великолепный обучающий материал по селекторам. Этот учебник покрывает намного больше, чем вы изучили здесь. Как только вы как следует усвоите селекторы, с которыми я вас познакомил, очень рекомендую почитать этот материал.

Задайте стили для страницы ВиО для Amazeriffic

Если вы выполнили задания в конце главы 2, то у вас есть страница ВиО (FAQ) для Amazeriffic. Свяжите файл `style.css` со страницей `faq.html` и добавьте дополнительные стилевые правила, чтобы задать стили для области содержания на этой странице. Если структура заголовка и подвала такая же, то вам не придется редактировать эти правила. Чтобы задать стили для страницы ВиО, нужно будет просто добавить дополнительные правила для секции `main`.

Каскадные правила

В этой главе я осветил каскадные правила CSS довольно поверхностно. Я думаю, в большинстве случаев они вполне понятны интуитивно. В то же время знание правил в деталях очень часто оказывается полезным при решении проблем с CSS. Принципы четко определены в стандарте W3C. Я рекомендую вам прочесть их и хорошенько запомнить, особенно если придется много работать с CSS.

Адаптивность и библиотеки адаптивности

Обширная тема в CSS, которую мы не изучили, — адаптивный дизайн. Дизайн является *адаптивным*, если он изменяется в зависимости от высоты и ширины окна браузера, в котором отображается страница. Это очень важно в наши дни, когда огромное количество людей просматривает веб-приложения и веб-страницы с помощью мобильных телефонов и планшетов.

Адаптивность CSS можно обеспечить с помощью медиазапросов, но рассмотрение этой темы несколько выходит за рамки книги. Вы можете прочитать о них на сайте Mozilla для разработчиков. Я настоятельно рекомендую сделать это.

Кроме того, некоторые CSS-фреймворки предоставляют возможность построения адаптивного дизайна с минимальными усилиями. Я использую как Twitter Bootstrap, так и Zurb's Foundation. Если вам понравилось работать с компоновкой и CSS в этой главе, то советую прочесть документацию к обоим фреймворкам и попробовать сделать дизайн Amazeriffic адаптивным с помощью одного из них. Помните об этих фреймворках как о стартовых точках для будущих проектов.

4 Интерактивность

Итак, мы изучили три очень важных аспекта разработки веб-приложений. К сожалению, мы пока что не создали приложения, которое что-нибудь *делает*. Мы просто задали структуру его содержания и привлекательно оформили.

В этой главе мы с вами начнем исследование интерактивности. Это позволит нам наконец уйти от построения статичных веб-страниц — мы начнем создавать динамические приложения, реагирующие на действия пользователя.

К концу этой главы мы кое-что узнаем о JavaScript — сценарном языке программирования, который работает внутри каждого браузера, и jQuery — библиотеке JavaScript, которая очень полезна для управления элементами DOM (помимо всего прочего). Кроме того, уделим некоторое время изучению основ языка JavaScript, обратив основное внимание на использование массивов и управление ими.

Привет, JavaScript!

Начнем с примера. Как и прежде, создадим в папке Projects каталог под названием Chapter4. В нем сделаем еще один, Example1, и откроем его в Sublime. Затем создадим три файла: `index.html`, `style.css` и `app.js`. Начнем с `index.html`:

```
<!doctype html>
<html>
  <head>
    <title>Название приложения</title>
    <link href="style.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>Название приложения</h1>
    <p>Описание приложения</p>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

Все в этом файле должно быть вам знакомо, за исключением тегов `<script>` в нижней части элемента `body`. Как и теги `link`, элементы `script` позволяют вам связывать

внешние файлы с вашим HTML, но script, как правило, обращаются к файлам JavaScript. И еще одна важная особенность: в отличие от тега <link>, для <script> обязателен закрывающий тег.

Мы поместили теги <script> в body, а не в head по технической причине: браузер отображает страницу сверху вниз, создавая DOM-элементы по мере продвижения по HTML-документу. Если мы помещаем теги <script> в конце, то файлы JavaScript будут открыты одними из последних. Поскольку эти файлы часто довольно большие и требуют некоторого времени на загрузку, лучше открывать их в последнюю очередь, чтобы как можно быстрее отобразить для пользователя визуальные элементы.

Первый тег <script> включает широко используемую библиотеку под названием *jQuery*, которую мы подробно изучим в последующих секциях. Тег открывает jQuery с http://code.jquery.com — *сети доставки контента*. Это значит, что нет необходимости хранить копию jQuery на локальной машине. Вместо этого мы указываем браузеру загрузить ее с внешнего сайта (аналогичным образом мы обрабатывали Google Fonts в предыдущей главе). Так что ваш компьютер должен быть подключен к Интернету, чтобы этот пример работал правильно.

Второй элемент script включает сценарий app.js, который содержит написанный нами код JavaScript. Мы скоро создадим этот файл.

У нас есть также связанный файл CSS, в котором уберем поля и отступы, заданные по умолчанию, для всех элементов на странице. Использование универсального селектора вызовет предупреждение от CSS Lint, но мы пока проигнорируем его.

```
* {
  margin: 0;
  padding: 0;
}
```

Файл app.js наиболее интересен для нас. Его содержание показано в следующем фрагменте кода. Прежде чем мы поговорим о нем, наберите этот код в точности как в книге, а затем откройте файл index.html в браузере.

```
var main = function () {
  "use strict";
  window.alert("hello, world!");
};
$(document).ready(main);
```

Если вы все набрали правильно, то должны увидеть оповещение JavaScript с текстом «hello, world!» при загрузке страницы (рис. 4.1).

Это та самая база, которую мы будем использовать во всех наших программах JavaScript. Два ключевых момента, которые мы и будем рассматривать в остальной части книги, посвященной клиентской стороне, заключаются в следующем:

- определяется глобальная функция, называемая main, которая является входной точкой выполнения программы;
- для выполнения функции main используется jQuery после того, как документ HTML полностью загрузился и готов к использованию.

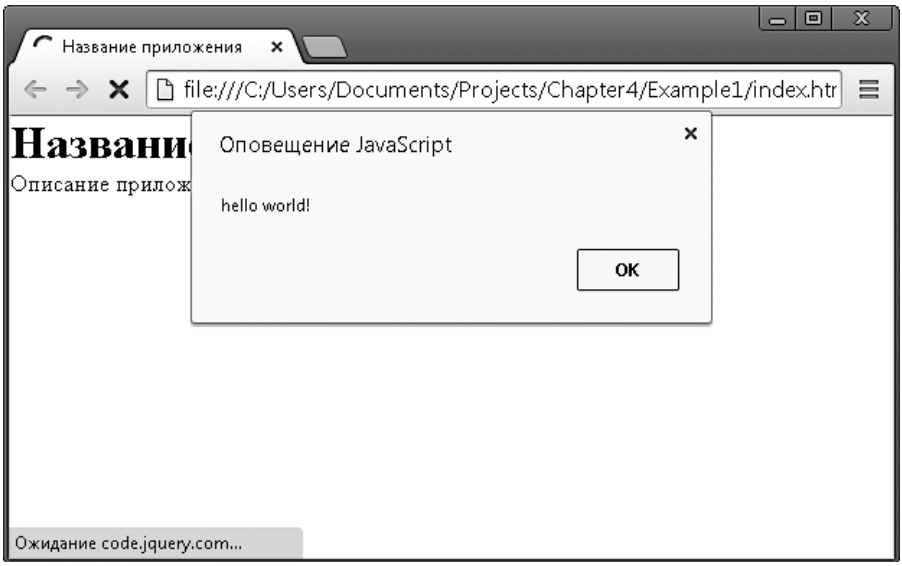


Рис. 4.1. Наше первое интерактивное приложение

Код настроен на запуск в строгом режиме (strict), который запрещает исполнение некоторых неудачных аспектов JavaScript, вызывавших проблемы у разработчиков в прошлом. В нашем коде это всегда будет первой строкой в функции `main`. Сценарий также создает сообщение «Hello, world!» — просто для примера.

Первое интерактивное приложение

А сейчас создадим с помощью JavaScript что-нибудь поинтереснее. Наша цель — создание простого приложения, позволяющего пользователю ввести какой-то текст, который затем появится на странице в другом месте. Этот пример продемонстрирует, как просто можно создавать интересные вещи с использованием всего нескольких строк кода.

Начнем с создания новой папки Example2 в каталоге Chapter4. Создайте пустое хранилище Git и откройте папку в Sublime. Создайте заново три файла из примера, приведенного в предыдущем разделе (`index.html`, `style.css` и `app.js`). Как мы уже обсуждали в предыдущих главах, полезно запомнить эту основную структуру, чтобы вы могли создавать ее сразу, без необходимости копировать код.

Как только у вас появилась основа проекта, выполните начальный коммит в хранилище Git.

Структура

Посмотрим, к какому пользовательскому интерфейсу нам нужно стремиться. Он показан на рис. 4.2.

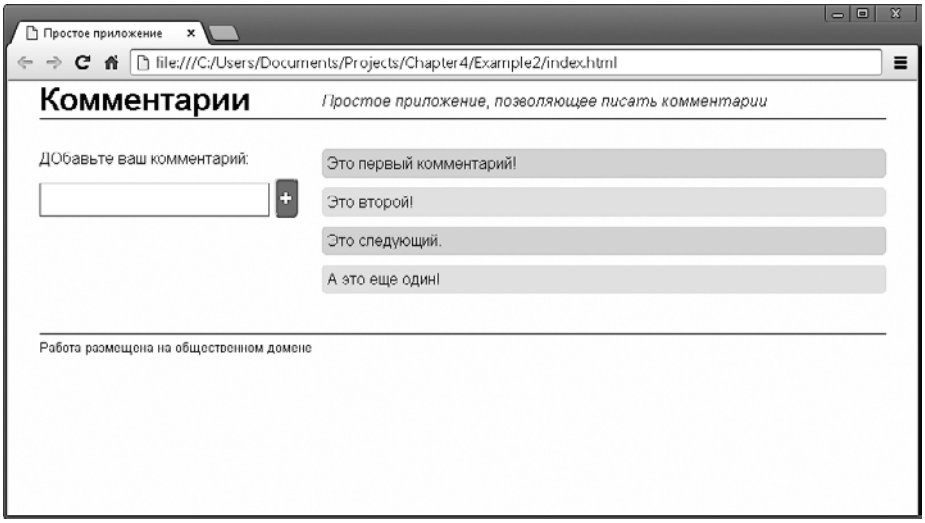


Рис. 4.2. Пользовательский интерфейс нашего интерактивного приложения

Вы видите, что структура состоит из трех основных элементов: заголовка `header`, подвала `footer` и основной части `main`. Помня об этом, можем модифицировать HTML для отображения этого интерфейса:

```
<!doctype html>
<html>
  <head>
    <title>Простое приложение</title>
    <link href="style.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <header>
    </header>
    <main>
      <!-- обратите внимание: здесь для названия класса css, содержащего
      несколько слов, мы используем тире, это соглашение CSS -->
      <section class="comment-input">
      </section>
      <section class="comments">
      </section>
    </main>
    <footer>
    </footer>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

Как только вы соответствующим образом изменили HTML, отправьте его в хранилище Git. Затем заполните элементы `header` и `footer` вместе с основной частью

содержания main с левой стороны в соответствии с рис. 4.2. Вы, наверное, заметили, что некоторые из этих блоков будут содержать теги <h1>, <h2> и <h3> с определенным содержимым. Добавьте их сейчас.

В блоке comment-input нам понадобится новый тег HTML <input>. Элемент input представляет собой текстовое поле ввода и не требует закрывающего тега. Добавьте пустой тег <input> в блок comment-input. Вы можете также добавить абзац с подсказкой «Добавьте ваш комментарий»:

```
<section class="comment-input">
  <p>Добавьте ваш комментарий:</p>
  <input type="text">
</section>
```

В блоке .comments добавьте несколько комментариев — просто в тегах абзацев. Это поможет в ходе присвоения стилей. Например, мы можем модифицировать секцию с комментариями примерно так:

```
<section class="comments">
  <p>Это первый комментарий!</p>
  <p>Это второй!</p>
  <p>Это следующий.</p>
  <p>А это еще один!</p>
</section>
```

И наконец, нужно добавить элемент button — кнопку с пометкой +. Вы еще не видели его раньше, но этот элемент бесполезен, пока мы не добавим для него код JavaScript. Чтобы добавить кнопку, вставим тег <button> непосредственно за тегом <input>. Код, находящийся в моей секции comment-input, выглядит так:

```
<section class="comment-input">
  <p>Add Your Comment:</p>
  <input type="text"><button>+</button>
</section>
```

Закончив работать со структурой, отправим ее в хранилище Git, а затем перейдем к заданию стилей на странице.

Стиль

Оформление будет совсем несложным. Полезно будет использовать сброс CSS, но это не обязательно для данного примера. Но как минимум вы должны создать простую стартовую точку для CSS, обнулив поля и отступы.

Ширина основной части в данном примере — 855 пикселов, а левая колонка занимает одну треть ширины. Другие две трети содержат секцию комментариев.

Один интересный аспект секции комментариев — меняющиеся цвета. Другими словами, фон четных комментариев — одного цвета, а нечетных — другого. Мы легко можем достичь этого эффекта с помощью CSS — просто добавим псевдокласс ntn-child со значениями even (четные) и odd (нечетные). Например, если мы хотим, чтобы чередовались цвета lavender и gainsboro, то можем создать вот такие правила CSS:

```
.comments p:nth-child(even) {
  background: lavender;
}
.comments p:nth-child(odd) {
  background: gainsboro;
}
```

Если вы добавили несколько комментариев для примера, то можете увидеть, что цвет их фона меняется в соответствии с новыми правилами CSS. Обратите внимание на то, что мы также добавили для комментариев свойство `border-radius`, чтобы уголки фоновой подложки были слегка закругленными.

Кроме этого, и остальная часть оформления должна соответствовать макету как можно точнее. Закончите присваивать стили интерфейсу настолько хорошо, насколько сможете, и отправьте код в хранилище Git, чтобы мы могли перейти к интерактивности. Если столкнетесь с трудностями, посмотрите на примеры, находящиеся в хранилище Git.

Интерактивность

А сейчас мы с вами готовы работать с интерактивностью. Построим наш пример шаг за шагом, по ходу дела выполняя коммиты в Git.

Обработка щелчков кнопкой мыши

Начнем со следующего: когда пользователь нажимает кнопку +, мы вставляем новый комментарий в секцию комментариев. Для этого изменим код `app.js` вот так:

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
   console.log("Hello, World!");
  });
};
$(document).ready(main);
```

Загрузите вашу страницу в браузере и откройте инструменты разработчика Chrome, как описано в предыдущей главе. После этого щелкните на вкладке Console (Консоль) на верхней панели инструментов разработчика. Сейчас, щелкнув на кнопке +, вы увидите надпись «Hello, World!», появляющуюся в консоли. Если все настроено правильно и работает корректно, это отличный момент для того, чтобы отправить вашу работу в хранилище Git.

Что же здесь происходит? Код связывает слушатель событий с элементом DOM, указанным в вызове функции `$`. Обратите внимание на то, что аргумент функции `$` очень похож на селектор CSS — и не случайно! Именно так и обстоит дело. jQuery позволяет вам легко обращаться к элементам DOM с помощью селекторов CSS, а затем управлять ими, используя JavaScript.

Событие, которое мы ожидаем на элементе `button`, — это щелчок кнопкой мыши. Слушатель сам по себе — это функция, которая просто выводит «Hello, World!»

в консоли. Так что на естественном языке мы можем пересказать этот код следующим образом: *когда пользователь нажимает кнопку +, набирай «Hello, World!» в консоли.*

Конечно же, на самом деле нам вовсе не нужно вводить «Hello, World!» в консоли — мы хотим добавить комментарий в секцию комментариев. Чтобы прийти к этому, мы должны немного модифицировать наш код, заменив `console.log` другой строкой jQuery, которая добавляет элемент DOM в раздел комментариев:

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
    $(".comments").append("<p>Это новый комментарий</p>");
  });
};
$(document).ready(main);
```

Обновите свою страницу в браузере, и, если все хорошо, вы должны увидеть, что в секции комментариев появился новый: «Это новый комментарий». Вы заметите, что код jQuery (начинающийся с `$`) выбирает секцию комментариев таким же образом, как и мы выбирали ее в файле CSS, а затем применяет к ней какой-то HTML-код. Если же это не работает, вернитесь обратно и убедитесь в том, что вы все набрали правильно.

Динамическое управление элементами DOM

Мы значительно продвинулись к нашей цели, но пока щелчок кнопкой мыши на кнопке приводит к появлению одного и того же комментария. А нужно изменять текст в тегах абзаца на тот, что находится в поле ввода. Начнем с создания переменной для хранения элемента DOM, который мы собираемся добавить:

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
    var $new_comment = $("<p>");
    $new_comment.text("Это новый комментарий");
    $(".comments").append($new_comment);
  });
};
$(document).ready(main);
```

В результате этой модификации не изменилось ничего, что делает наш код, но сам код *реструктурирован*: теперь настраивать текст в тегах абзаца куда проще. В частности, мы добавили объявление и назначение *переменной*. Имя переменной `$new_comment` может быть любым, каким хотите, но если в ней будет храниться объект jQuery, разумно было бы выделять ее, используя `$` в качестве первого символа.

Первая строка нового кода создает новый элемент абзаца как объект jQuery, а вторая изменяет текстовое содержание этого абзаца на «Это новый комментарий». Поскольку jQuery позволяет объединять вызовы функций в *цепочки*, мы можем, если так удобнее, сделать из двух строк одну:

```
var $new_comment = $("<p>").text("this is a new comment");
```

Но даже если мы используем в jQuery цепочку, важно помнить, что здесь происходят две вещи — создается новый элемент абзаца, а его текстовое содержание заменяется на «Это новый комментарий».

Затем нам нужно сделать так, чтобы содержимое текстового поля ввода хранилось в переменной, которую мы создали. Перед этим, однако, остановимся на том, как определить элемент `input`, находящийся в секции `.comment-input` в jQuery.

Можно попробовать сделать это, используя селекторы CSS! Вот решение:

```
$(".comment-input input");
```

Как и в CSS, эта строка ищет элементы класса `.comment-input`, а затем просматривает их дочерние элементы в поисках элемента `input`. Поскольку только один элемент отвечает этим критериям, он и будет выбран в результате.

Итак, теперь мы знаем, что можем получить содержимое извне. Оказывается, что в jQuery есть функция, позволяющая вернуть содержимое текстового поля ввода. Она называется `val` — сокращение от value («значение»). Мы можем получить доступ к содержимому текстового поля, использовав вот такой код:

```
$(".comment-input input").val();
```

А сейчас надо просто его обработать! В данном случае необходимо, чтобы содержимое текстового поля ввода стало текстовым содержанием нового элемента абзаца. Следовательно, нужно реструктурировать код примерно так:

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
    var $new_comment = $("<p>"),
    comment_text = $(".comment-input input").val();
    $new_comment.text(comment_text);
    $(".comments").append($new_comment);
  });
};
$(document).ready(main);
```

Если нужно, мы можем поместить все это в одну строку без использования промежуточной переменной для хранения результата вызова функции `val`:

```
var $new_comment = $("<p>").text($(".comment-input input").val());
```

А сейчас откройте приложение в вашем браузере. Если все хорошо, вы сможете набрать какой-либо текст в текстовом поле, щелкнуть на кнопке и увидеть, что набранный текст добавлен в секцию комментариев. Если что-то не работает, вы можете проверить код еще раз, чтобы удостовериться, что все двоеточия и скобки на своих местах, а затем открыть Инструменты разработчика Chrome и проверить, нет ли каких-либо ошибок в консоли. Сообщение об ошибке может намекнуть вам, где именно что-то не так.

Как только вы добились корректной работы приложения, отправьте изменения в хранилище Git с поясняющим комментарием.

В данной точке все должно работать правильно. Для пользователей, однако, приложение оставляет желать лучшего, и мы можем внести несколько мелких изменений, которые сделают работу куда более приятной.

Исправление ошибки

Вы, вероятно, не подозреваете, что в нашем коде уже есть баги, но, как ни странно, это именно так! Полезно будет уделить некоторое время продумыванию ожидаемого поведения нашего приложения и проверке того, соответствует ли ему действительное положение вещей.

Если вы все еще его не нашли, вот подсказка: когда мы щелкаем на кнопке добавления при отсутствии текста в поле ввода, программа jQuery добавляет в DOM пустой элемент `p`. Из-за этого нарушается цветовая схема четных/нечетных элементов, которую мы разработали в CSS.

Чтобы увидеть, как баг делает свое черное дело, прежде всего перезагрузите страницу. Затем напишите один комментарий с помощью поля ввода, очистите поле ввода, щелкните на кнопке добавления, а затем добавьте еще один комментарий. Если вы в точности следовали этим инструкциям, то увидите, что два комментария подряд отображаются с одним и тем же цветом фона! Это потому, что пустой комментарий, который не отображается, должен был появиться на фоне «нечетного» цвета.

Мы можем убедиться, что баг проявляется именно из-за пустого элемента `p`, также с помощью вкладки Elements инструментов разработчика Chrome. Начните с открытия инструментов разработчика Chrome и перехода к вкладке Elements. Затем перейдите от элемента `main` к секции комментариев. Имея перед глазами эту секцию, вы можете сейчас нажать кнопку добавления, не написав никакого комментария в поле ввода, и увидите, что был добавлен пустой элемент `p`.

Что же делать? Вообще говоря, следует перед добавлением комментария выполнять проверку того, есть ли какое-либо содержимое в поле ввода. Мы можем сделать это с помощью оператора `if`:

```
$(".comment-input button").on("click", function (event) {
  var $new_comment;
  if ($(".comment-input input").val() !== "") {
    $new_comment = $("<p>").text($(".comment-input input").val());
    $(".comments").append($new_comment);
  }
})
```

Сочетание `!==` подтверждает, что содержимое поля ввода не должно быть пустой строкой, так что, по сути, это и есть проверка того, что поле ввода не пустое. Таким образом, условие `if` допускает выполнение кода, добавляющего новый комментарий, только в том случае, если поле ввода не пустое. Так простое изменение позволило нам исправить баг, и сейчас было бы неплохо отправить коммит в хранилище Git.

Очистка поля ввода

Еще одна серьезная проблема пользовательского взаимодействия — то, что поле ввода не очищается после того, как пользователь щелкнет на кнопке. Если он хочет

добавить еще один комментарий, то вынужден вручную удалять текст, добавленный предварительно. Очистить поле ввода очень просто: мы обращаемся к объекту в jQuery с помощью метода `val`, непосредственно указывая его значение, поле ввода примет это значение. Иными словами, мы можем очистить текущее содержание поля, отправив в него пустую строку с помощью метода `val`:

```
$(".comment-input input").val("");
```

Чтобы реализовать эту функциональность в коде, потребуется добавить всего одну строку:

```
$(".comment-input button").on("click", function (event) {
  var $new_comment;
  if ($(".comment-input input").val() !== "") {
    var $new_comment = $("<p>").text($(".comment-input input").val());
    $(".comments").append($new_comment);
    $(".comment-input input").val("");
  }
});
```

Ожидаемое действие кнопки Enter

Еще одна вещь, которой обязательно будут ожидать пользователи, — отправка комментария при нажатии клавиши Enter на клавиатуре. Это привычное положение вещей, например, при работе с мессенджерами.

Как мы можем этого добиться? Нужно добавить дополнительный обработчик событий, который улавливает нажатие клавиши непосредственно в элементе ввода. Мы можем добавить его сразу после обработчика нажатия:

```
$(".comment-input input").on("keypress", function (event) {
  console.log("hello, world!");
});
```

Обратите внимание на два важных различия между этим и предыдущим обработчиками. Первое заключается в том, что новый обработчик настроен на улавливание нажатия клавиши клавиатуры, а не щелчка кнопкой мыши. Второе — в том, что мы перехватываем события в другом элементе: в данном случае внимание направлено на само поле ввода, тогда как в предыдущем обрабатывали элемент `button`.

Если вы успели испробовать этот код, то увидели, что надпись «Hello, World!» появляется в логе консоли разработчика Chrome каждый раз, когда мы нажимаем какую-то клавишу. Но нам нужно, чтобы нажатие большинства клавиш было проигнорировано и реакция наблюдалась только тогда, когда пользователь нажимает клавишу Enter. Чтобы добиться этого, мы можем использовать локальную переменную, которая не понадобилась нам в предыдущем разработчике, — она будет хранить значение с информацией о нажатой клавише. Как можно увидеть это значение? Немного поменяем код:

```
$(".comment-input input").on("keypress", function (event) {
  console.log("Это значение keyCode " + event.keyCode);
});
```

Обратите внимание на то, что в слове keyCode буква C прописная. Это пример camelCase — «верблюжьего регистра» («горбатый Регистр», «стиль Верблюда»): когда мы даем переменной имя, объединяющее несколько слов, каждое из них, кроме первого, нужно писать с большой буквы.

Для вывода в лог мы используем знак «+» для связывания значения `keyCode` со строкой, которая начинается с «Это значение keyCode». Запустив код, увидим действительное значение `keyCode`, выведенное в лог.

Сейчас, открыв браузер и начав что-то набирать в поле ввода, мы увидим значения `keyCode`, выводимые на экран одно за другим. Воспользовавшись этим, мы узнаем значение `keyCode` для клавиши Enter. Сделав это, мы можем переработать код в условии `if` так, чтобы он реагировал только на нажатие клавиши Enter:

```
$(".comment-input input").on("keypress", function (event) {
  if (keyCode === 13) {
    console.log("Это значение " + event.keyCode);
  }
});
```

Теперь код выводит значение `keyCode` только при нажатии клавиши Enter. И наконец, надо скопировать код из другого обработчика событий, который добавляет новый комментарий:

```
$(".comment-input input").on("keypress", function (event) {
  var $new_comment;
  if (event.keyCode === 13) {
    if ($(".comment-input input").val() !== "") {
      var $new_comment = $("<p>").text($(".comment-input input").val());
      $(".comments").append($new_comment);
      $(".comment-input input").val("");
    }
  }
});
```

Плавное появление комментария

Ну вот, сейчас все важнейшие функциональности должны работать. Но добавим еще один аспект взаимодействия: пусть комментарий отображается на странице постепенно вместо простого немедленного появления. К счастью, с помощью jQuery сделать это очень просто, так как в каждый элемент jQuery встроен метод `fadeIn`. Но для того, чтобы элемент плавно отобразился, нужно, чтобы изначально он был скрыт. Поэтому перед добавлением элемента в DOM мы должны применить к нему метод `hide`. Это сделает следующий код:

```
$(".comment-input button").on("click", function (event) {
  var $new_comment;
  if ($(".comment-input input").val() !== "") {
    $new_comment = $("<p>").text($(".comment-input input").val());
    $new_comment.hide();
    $(".comments").append($new_comment);
```

```
    $new_comment.fadeIn();
    $(".comment-input input").val("");
  }
});
```

А сейчас добавим новый комментарий с помощью кнопки. Вы увидите, что он отображается на странице постепенно вместо немедленного появления. Таким же образом нужно модифицировать и события, наступающие после нажатия клавиши Enter, но перед тем, как заняться этим, отправьте новый коммит в хранилище.

Переработка для упрощения

Сейчас мой (и ваш тоже) файл `app.js` должен выглядеть так:

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
    var $new_comment;
    if ($(".comment-input input").val() !== "") {
      $new_comment = $("<p>").text($(".comment-input input").val());
      $new_comment.hide();
      $(".comments").append($new_comment);
      $new_comment.fadeIn();
      $(".comment-input input").val("");
    }
  });
  $(".comment-input input").on("keypress", function (event) {
    var $new_comment;
    if (event.keyCode === 13) {
      if ($(".comment-input input").val() !== "") {
        $new_comment = $("<p>").text($(".comment-input input").val());
        $new_comment.hide();
        $(".comments").append($new_comment);
        $new_comment.fadeIn();
        $(".comment-input input").val("");
      }
    }
  });
};
$(document).ready(main);
```

Вы, наверное, сразу заметили, что код дублируется. В частности, код, добавляющий комментарий, повторяется в обоих обработчиках событий, а когда мы меняем один из них, чтобы заставить текст появляться постепенно, то вынуждены таким же образом изменить и другой. Это противоречит одному из принципов разработки, известному как DRY, что значит Don't Repeat Yourself («Не повторяйся»). Когда вы копируете и вставляете код, в вашем сознании должен прозвучать тревожный звоночек: подумайте, нет ли лучшего пути, чтобы достичь нужного результата? В данном случае можем переписать код вот так:

```
var main = function () {
  "use strict";
  var addCommentFromInputBox = function () {
    var $new_comment;
    if ($(".comment-input input").val() !== "") {
      $new_comment = $("<p>").text($(".comment-input input").val());
      $new_comment.hide();
      $(".comments").append($new_comment);
      $new_comment.fadeIn();
      $(".comment-input input").val("");
    }
  };
  $(".comment-input button").on("click", function (event) {
    addCommentFromInputBox();
  });
  $(".comment-input input").on("keypress", function (event) {
    if (event.keyCode === 13) {
      addCommentFromInputBox();
    }
  });
};
$(document).ready(main);
```

В этом примере мы делаем дублирующийся код абстрактной функцией, а затем вызываем ее в каждом обработчике событий. Мы можем добиться этого с помощью объявления переменной для хранения функций и затем определения функции. Таким образом, код становится гораздо более удобным в обслуживании: решив внести изменения в механизм комментирования, мы должны проделать это только в одном месте!

Общие сведения о jQuery

Каким насыщенным был этот пример, но, надеюсь, вы все-таки с ним справились! С его помощью мы немного познакомились с множеством возможностей jQuery. А сейчас сделаем шаг назад и поговорим о том, с чем мы сейчас работали. Вообще jQuery предоставляет нам три основные возможности:

- упрощенный и четко организованный способ манипулирования элементами DOM;
- способ последовательной обработки событий DOM;
- упрощенный подход к AJAX.

Изучим две первые из них в этой главе, а третью — в следующей. Нужно отметить также, что в jQuery есть огромное количество сторонних плагинов, с помощью которых можно быстро и просто улучшить любой сайт.

Создание проекта

Перед тем как приступить, уделим немного времени обсуждению рабочего процесса. В целом файлы клиентской стороны приложения будут делиться на HTML, CSS

и JavaScript. Таким образом, всегда будет полезно каким-то образом их организовать.

Начиная с этого момента будем хранить файлы HTML в корневом каталоге проекта; кроме того, там будут находиться три подкаталога: stylesheets (для таблиц стилей), images (для изображений), javascripts (для сценариев). Таким образом, мы всегда будем знать, где находятся нужные файлы.

В соответствии с этим придется немного изменить привязку к таблицам стилей или загрузку сценариев. Вот пример того, как такие изменения повлияют на HTML:

```
<!doctype html>
<html>
  <head>
    <link href="stylesheets/reset.css" rel="stylesheet">
    <link href="stylesheets/style.css" rel="stylesheet">
  </head>
  <body>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="javascripts/app.js"></script>
  </body>
</html>
```

Обратите внимание: мы ясно указываем, где размещаются сценарии и таблицы стилей, с помощью слэша и наименования папки.

Комментарии

Прежде чем начать работать над реальным кодом, поговорим о комментариях в JavaScript. Как и в HTML или CSS, мы можем добавлять комментарии в код JavaScript для удобства его понимания человеком. Комментарии могут быть двух типов: однострочные и многострочные. Однострочные комментарии создаются с помощью двух слэшей, следующих один за другим, а многострочные выглядят точно так же, как комментарии CSS:

```
// это однострочный комментарий, продолжающийся до конца строки
var a = 5; // это однострочный комментарий, располагается на одной строке с кодом
/* А это многострочный комментарий, который будет продолжаться,
пока мы не закроем его. Он выглядит точно так же, как комментарий в CSS. */
```

Селекторы

Как мы уже убедились в предыдущем примере, селекторы в jQuery очень похожи на селекторы CSS. Фактически можно использовать любой селектор CSS в качестве селектора для jQuery. Например, вот эти селекторы jQuery будут действовать точно так, как мы ожидаем:

```
$("*");              // выбираются все элементы в документе
$("h1");             // выбираются все элементы h1
```

```
$("p");              // выбираются все элементы p
$("p .first");       // выбираются все абзацы класса 'first'
$(".first");         // выбираются все элементы класса 'first'
$("p:nth-child(3)"); // выбираются все абзацы, являющиеся дочерними элементами
                     третьего уровня
```

Но на самом деле селекторы jQuery идентичны селекторам CSS не всегда. В jQuery существует богатый набор псевдоклассов и псевдоэлементов, которые в CSS недоступны. Кроме того, некоторые верные для CSS идентификаторы должны представляться в jQuery иначе (специальные символы, например «.», должны отделяться двумя обратными слэшами). Но для решения наших задач вполне допустимо принять, что селекторы jQuery — то же самое, что селекторы CSS, которые указывают на элементы DOM для последующего управления ими с помощью JavaScript. Если вы хотите изучить эту тему подробнее, можете обратиться к документации по селекторам в jQuery.

Управление элементами DOM

После того как мы с помощью селекторов успешно добавили элементы в DOM, можно начать управление ими. В jQuery это очень просто.

Добавление элементов в DOM

Для начала очень полезно запомнить древовидную ментальную модель DOM. Например, рассмотрим следующий фрагмент HTML:

```
<body>
  <h1>Это пример!h1>
  <main>
  </main>
  <footer>
  </footer>
</body>
```

Обратите внимание на то, что элементы `main` и `footer` пусты. Мы можем нарисовать для такого примера диаграмму, похожую на показанную на рис. 4.3.

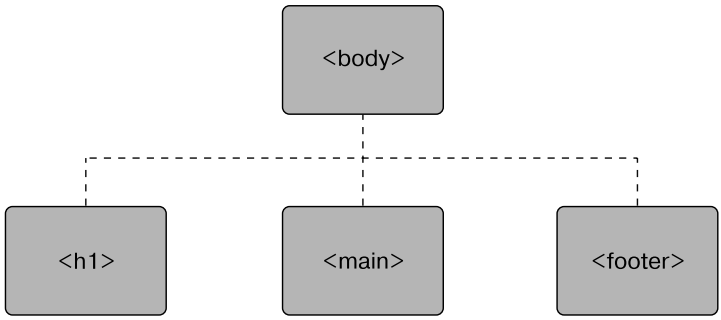


Рис. 4.3. Древовидная диаграмма элементов DOM перед переходом к управлению jQuery

Чтобы создать элемент, мы должны использовать тот же самый оператор $, что и для выбора элемента. Но вместо селектора указываем в нем тег, представляющий элемент, который нужно создать:

```
// Как договорились, я даю переменным объектов jQuery названия, начинающиеся с $
var $newUL = $("<ul>"); // создание нового элемента списка
var $newParagraphElement = $("<p>"); // создание нового абзаца
```

После того как мы создали элемент, можем добавлять туда разные объекты, включая другие HTML-элементы. Например, можем добавить текст в элемент абзаца вот так:

```
$newParagraphElement.text("Это абзац");
```

После исполнения кода абзац в DOM будет эвивалентен следующему коду HTML:

```
<p>Это абзац</p>
```

Новые элементы `ul` и `p` еще не являются частью DOM, которая отображается на странице, поэтому они будут выведены из древовидной диаграммы, но необходимо учесть их с помощью соответствующих переменных jQuery (рис. 4.4).

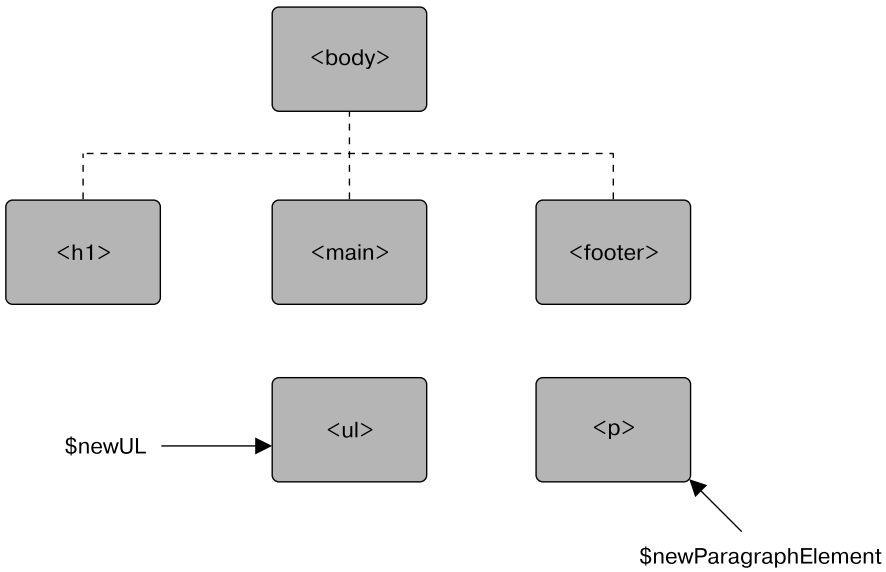


Рис. 4.4. Элементы ul и p уже созданы, но еще не добавлены в DOM

Мы можем добавить их на страницу, выбрав элемент, который будет для них родительским, а затем вызвав функцию `append` для объекта jQuery. Например, если мы хотим сделать элемент дочерним для `footer`, следует написать вот такую строку:

```
$("footer").append($newParagraphElement);
```

Сейчас новый элемент будет находиться в подвале и, следовательно, присоединяется к древовидной диаграмме! Это показано на рис. 4.5.

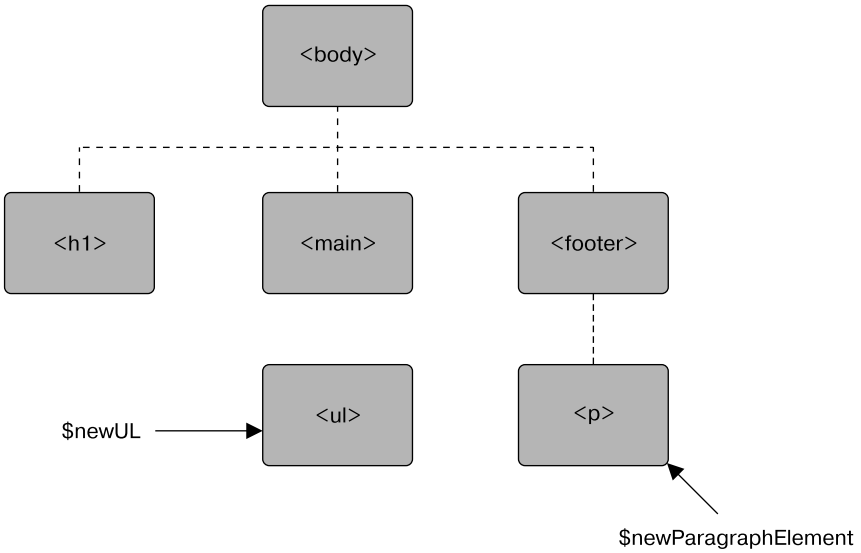


Рис. 4.5. DOM после того, как элемент p был добавлен в подвал

На самом деле мы можем создавать куда более сложные структуры DOM до того, как они отобразятся на странице. Например, можем добавить элементы `li` в элемент `ul`, созданный ранее:

```
// мы можем соединить наше творение и обратиться к нему, чтобы добавить текст
var $listItemOne = $("<li>").text("Это первый элемент списка");
var $listItemTwo = $("<li>").text("Второй элемент списка");
var $listItemThree = $("<li>").text("Боже, третий элемент списка");
// а сейчас мы привяжем эти элементы к ul, созданному ранее
$newUL.append($listItemOne);
$newUL.append($listItemTwo);
$newUL.append($listItemThree);
```

В этой точке у нас получилось *поддерево*, не связанное с остальной частью дерева (рис. 4.6).

А сейчас предположим, что мы хотим присоединить это поддерево к основной части в качестве потомка элемента `main`. Это делается аналогично показанному выше, но связывать нужно только *корень* поддерева!

```
$("main").append($newUL);
```

Сейчас наша DOM выглядит так, как показано на рис. 4.7.

Как только у нас появляется выбранный элемент DOM, работа с jQuery допускает некоторую гибкость. Например, мы можем поставить элементы на более высокий уровень вместо более низкого, что сделает их первым дочерним элементом родительского узла:

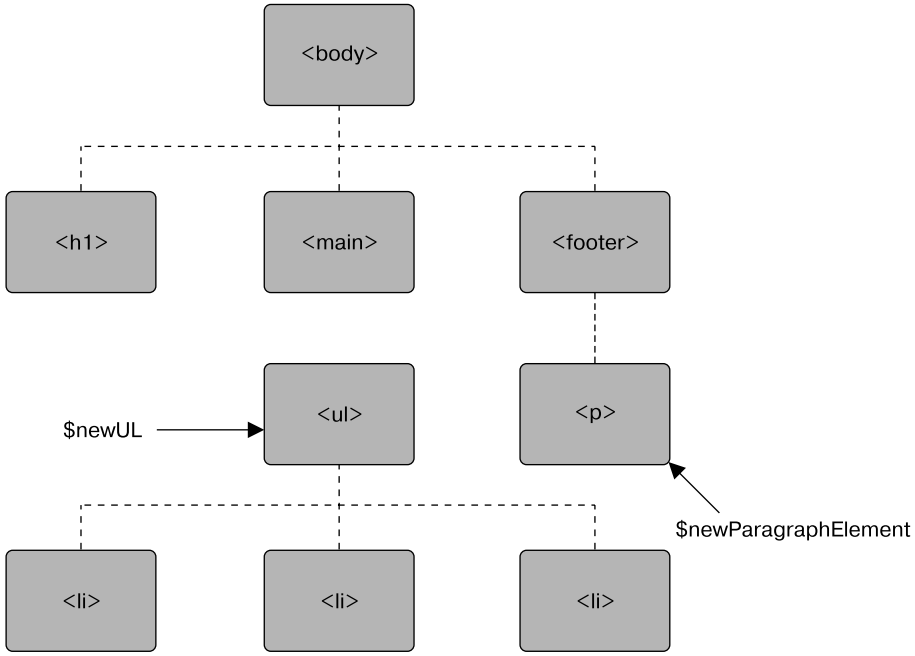


Рис. 4.6. DOM, содержащая поддерево с элементом ul, не связанное с остальной частью

Рис. 4.7. DOM, куда входит связанное поддерево, представляющее собой маркированный список

```
var $footerFirstChild = $("<p>").text("Я первый дочерний элемент подвала!");
$("footer").prepend($footerFirstChild);
```

Кроме того, мы можем использовать appendTo или prependTo, чтобы изменить порядок чтения кода:

```
// это эквивалент предыдущей строки
$footerFirstChild.appendTo($("footer"));
```

Не забывайте о возможности использования инструментов разработчика Chrome для подверждения того, что ваша DOM обновлена так, как описано в предыдущем разделе.

Удаление элементов из DOM

Удалять элементы очень просто, и jQuery предоставляет несколько способов сделать это. Один из них — просто использовать функцию remove вместе с селектором, что приведет к удалению из DOM выбранного элемента:

```
// удаляем первый пункт из списка,
// который мы создали ранее
$("ul li:first-child").remove();
```

Мы можем также удалить все потомки какого-либо элемента с помощью функции empty для объекта jQuery. Так, например, чтобы очистить элементы списка, которые мы создали в предыдущем примере, нужно проделать следующее:

```
// удаляем все элементы
// предварительно созданного списка
$newUL.empty();
```

В первом примере мы выучили, как использовать fadeIn для постепенного, плавного отображения скрытого элемента. В дополнение мы можем, используя функции fadeOut или slideUp, заставить элементы плавно исчезать перед тем, как удалить их из DOM!

```
// это удалит абзац в подвале из DOM
$("footer p").fadeOut();
```

На самом деле fadeOut не удаляет элементы из DOM, а лишь скрывает их. Для удаления мы должны применить к элементу функцию remove, но нужно сделать это *после* полного исчезновения. А это означает, что нужно синхронизировать между собой эти события, и в следующем разделе описано, как это сделать.

События и асинхронность

В первом примере в этой главе мы научились обрабатывать как щелчки кнопкой мыши, так и нажатие клавиши на клавиатуре. В общем, мы можем обработать любое событие с помощью шаблона on. Например, можем реагировать на событие dbclick, которое происходит, когда пользователь делает двойной щелчок кнопкой мыши:

```
$(".button").on("dblclick", function () {
  alert("Эй! Ты дважды щелкнул кнопкой мыши!");
});
```

В общем случае такой стиль программирования называется *событийно-ориентированным* (event-driven) или *асинхронным* (asynchronous) программированием. В чем его отличие от традиционного программирования? Наш код, вероятно, работал бы примерно так:

```
console.log("это выводится первым");
console.log("это выводится вторым");
console.log("это выводится третьим");
```

При асинхронном программировании функции, связанные с событиями, принимают обратные вызовы или функции, которые должны исполняться позднее, в качестве аргументов. Из-за этого понять порядок исполнения не всегда легко:

```
console.log("это выводится первым");
$("button").on("click", function () {
  console.log("это выводится, когда кто-то щелкает кнопкой мыши");
});
console.log("это выводится вторым");
```

Вот еще один пример, где для выполнения обратного вызова пользовательский ввод не требуется. Мы уже видели функцию `ready`, которая ожидает, пока DOM не будет готова выполнить обратный вызов. Функция `setTimeout` ведет себя аналогично, но выполняет обратный вызов по прошествии определенного количества миллисекунд:

```
// Это событие jQuery, выполняющее обратный вызов,
// как только DOM готова. В этом примере мы используем
// анонимную функцию вместо отправки функции main аргумента
$(document).ready(function () {
  console.log("Это будет выведено, как только документ будет готов");
});
// А это функция, встроенная в JavaScript, которая
// выполняется по прошествии определенного количества миллисекунд
setTimeout(function () {
  console.log("Это будет выведено через 3 секунды");
}, 3000);
// это будет выведено перед чем-либо еще, даже если
// появится последним
console.log("Это будет выведено первым");
```

На данный момент событийно-управляемое программирование, базирующееся на пользовательском взаимодействии, вполне может иметь место, а в примерах, включающих функции `setTimeout` и `ready`, разобраться несложно. Проблемы начинаются, когда мы хотим расположить события последовательно. Например, представьте ситуацию, в которой мы используем функцию jQuery `slideDown` для того, чтобы отобразить скользящее раскрытие какого-то текста:

```
var main = function () {
  "use strict";
  // создаем и скрываем содержимое в div
  var $content = $("<div>Hello, World!</div>").hide();
```

```
  // делаем его потомком элемента body
  $("body").append($content);
  // добавляем скользящее отображение за 2 секунды
  $content.slideDown(2000);
};
$(document).ready(main);
```

А теперь допустим, что мы хотим добавить постепенное отображение второго сообщения после того, как все содержимое появится. Скорее всего, мы испробуем что-то вроде следующего:

```
var main = function () {
  "use strict";
  // создаем и скрываем содержимое в div
  var $content = $("<div>Hello, World!</div>").hide();
  var $moreContent = $("<div>Goodbye World!</div>").hide();
  // делаем его потомком элемента body
  $("body").append($content);
  // добавляем скользящее отображение за 2 секунды
  $content.slideDown(2000);
  // добавляем еще один элемент в body
  $("body").append($moreContent);
  // и плавно отображаем его
  $moreContent.fadeIn();
}
$(document).ready(main);
```

Попробуйте набрать этот код и запустить его. Вы увидите, что текст «Goodbye, World!» постепенно отображается одновременно с тем, как раскрывается текст «Hello, World!». Это определенно не то, что нам нужно. Остановитесь на секунду и подумайте, почему так произошло.

Вы, наверное, догадались, что функция `slideDown` работает асинхронно. Это значит, что код, который за ней следует, выполняется *одновременно* с раскрытием текста! К счастью, jQuery предлагает обходной путь — большинство асинхронных функций принимают обратный вызов в качестве опционального параметра как свой последний аргумент, который позволяет располагать асинхронные события одно за другим. Таким образом, мы можем добиться нужного эффекта, изменив код вот так:

```
var main = function () {
  "use strict";
  // создаем и тут же скрываем комментарий в элементе div
  var $content = $("<div>Hello, World!</div>").hide();
  var $moreContent = $("<div>Goodbye, World!</div>").hide();
  // отправляем содержимое в элемент body
  $("body").append($content);
  // заставляем содержимое постепенно раскрываться вниз за 2 секунды,
  // а затем выполняем обратный вызов, в котором
  // находится другое содержимое
  $content.slideDown(2000, function () {
```

```
    // отправляем новое содержимое в body
    $("body").append($moreContent);
    // заставляем новое содержимое постепенно отобразиться
    $moreContent.fadeIn();
  });
};
$(document).ready(main);
```

Чтобы закончить пример из предыдущей секции, нужен тот же подход. Для удаления элемента p из подвала после того, как он постепенно исчез, нужно проделать следующее:

```
$("footer p").fadeOut(1000, function () {
  // это происходит, когда элемент p
  // постепенно исчезает
  $("footer p").remove();
});
```

Позднее в этой книге мы познакомимся с другими примерами асинхронного программирования с помощью Node.js, но сейчас нужно хорошенько разобраться с этими функциями и изучить шаблоны.

Общие характеристики JavaScript

Большое количество разработчиков клиентской стороны имеют глубокие знания в области HTML и CSS, а также jQuery, которые помогают им организовать работу плагинов и выполнять базовые манипуляции с элементами DOM. Знание этих трех вещей может сделать вас весьма компетентными в области веб-разработки на клиентской стороне. В то же время очень полезно может быть изучить JavaScript как можно глубже, чтобы быть высокоэффективным разработчиком клиентской стороны и создавать более сложные веб-приложения.

В этом разделе мы поговорим о некоторых основных возможностях JavaScript. Мой план состоит в краткости и фокусировании на самых важных аспектах языка. Мы увидим пример, который сводит вместе эти идеи, в конце главы.

Работа с JavaScript в Chrome JavaScript Console

Оказывается, не нужно создавать целый проект, чтобы попробовать работу с JavaScript. Прекрасный интерактивный интерпретатор JavaScript встроен прямо в инструменты разработчика Chrome!

Начнем с открытия нового окна или вкладки в браузере и перехода к инструментам разработчика, как описано в предыдущей главе (кнопка Настройки ▸ Инструменты ▸ Инструменты разработчика). Затем щелкните на вкладке Console в верхней части панели. Вы увидите подсказку, которая выглядит, как показано на рис. 4.8.

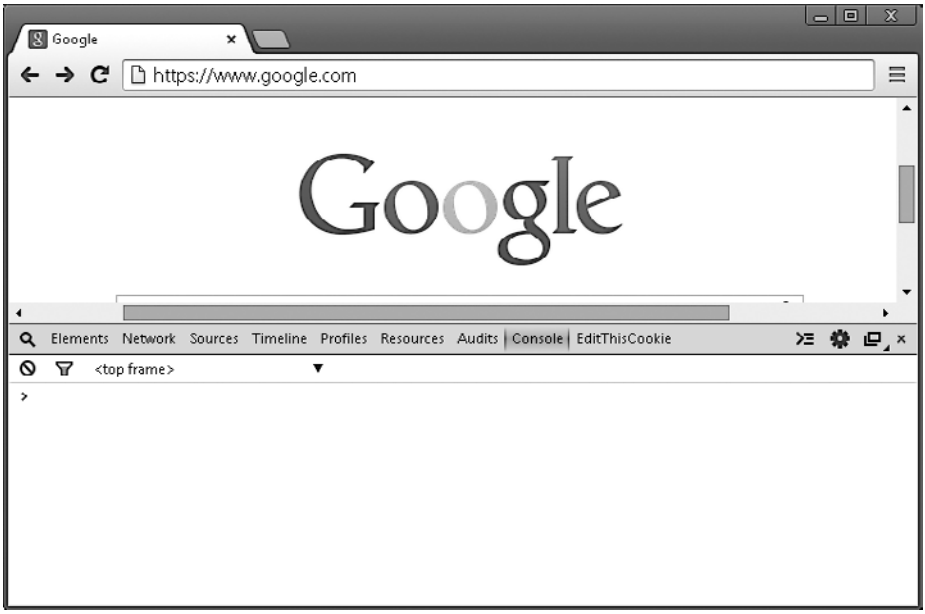


Рис. 4.8. Консоль JavaScript в Chrome

Сейчас мы можем работать с консолью JavaScript. Например, наберите вот такой код, нажимая Enter после каждой строки:

```
5+2;
//=> 7
Math.random();
//=> 0.3896130360662937
console.log("hello, world!");
//=> hello, world!
```

Первая строка должна вывести `7`, так как цифровое выражение равно 7. Вторая строка должна вывести случайное дробное число между 0 и 1.

Последняя строка должна вывести `hello, world!`, а затем — строку, которая сообщает: `undefined` (не определено). Все хорошо: Chrome всегда показывает результат вычисления последней строки. Команда `console.log` выполняет какие-то действия, но не имеет никакого результата оценки или вычислений, поэтому Chrome и сообщает, что ничего не было определено.

Далее на протяжении всего раздела вы можете вводить примеры прямо в JavaScript Console — с одной небольшой поправкой. Если вам нужно ввести что-то, помещающееся на нескольких строках (например, условие `if` для цикла `for`), следует в конце каждой строки вместо Enter нажимать Shift+Enter. Потренируйтесь на следующем коде:

```
var number = 5;
if (number >= 3) {
  console.log("Введенное число больше 3!");
```

```
}
//=> Введенное число больше 3!
```

Помните, что вы можете нажимать стрелки вверх или вниз на клавиатуре, чтобы передвигаться между набранными ранее фрагментами кода (аналогично тому, как передвигались по истории команд в командной строке).

Переменные и типы

JavaScript — не строго типизированный язык. Это означает, что переменные могут хранить любые данные любого типа (целочисленные, строки, десятичные числа и т. п.). В этом его отличие от языков наподобие Java и C++, где переменные должны быть строго определенного типа:

```
// Здесь хранится строка
var message = "Всем привет!";
// Здесь – числа
var count = 1;
var pi = 3.1415;
// Здесь хранится логическая переменная
var isFun = true;
console.log(message);
//=> Всем привет!
console.log(pi);
//=> 3.1415
console.log(isFun);
//=> true
```

Объявления и определения переменных могут быть одной записью, разделенной запятыми. Как я упоминал ранее, лучше всего поместить эти записи перед определением функций. Это правило создает последовательность, которая добавляет вашему коду четкости и облегчает чтение:

```
var main = function () {
  "use strict";
  var message = "hello, world!",
  count = 1,
  pi = 3.1415,
  isFun = true;
  console.log(message);
};
```

Функции

Мы уже видели много примеров работы функций. Здесь, в отличие от C++ или Java, функции являются пассажирами *первого класса*. Это значит, что мы можем назначать функциям переменные, отправлять функции в качестве аргументов для других функций и определять анонимные функции (которые являются попросту функциями без имен):

```
// определим функцию и сохраним ее
// в переменной под названием sayHello
var sayHello = function () {
  console.log("hello, world!");
}
// выполним функцию, находящуюся в переменной sayHello
sayHello();
//=> "hello, world!"
```

Как и в других языках, в JavaScript функции имеют *входные* и *выходные* данные. Входные данные часто называют *аргументами* или *параметрами* и указывают в скобках при определении функции. Выходные данные часто называют *возвращаемой величиной* и всегда сопровождают ключевым словом `return`:

```
// определяем функцию add, которая
// принимает два входных значения: num1 и num2,
// и выдает одно выходное: сумму
// двух чисел
var add = function (num1, num2) {
  // добавим входные данные и сохраняем результат в sum
  var sum = num1 + num2;
  // возвращаем сумму
  return sum;
}
// выполняем функцию add с входными данными 5 и 2
add(5,2);
//=> 7
```

Одно из интересных последствий того, что в JavaScript функции являются объектами первого класса, — возможность использовать одни функции в качестве аргументов для других. Мы используем этот шаблон для отправки анонимных функций в качестве обратных вызовов, но в этом качестве можно использовать и функции с названиями. Например, мы отправляли функцию `main` в качестве аргумента для функции `ready` в jQuery:

```
// входная точка main в программе
var main = function () {
  "use strict";
  console.log("hello, world!");
};
// функция main запускается, как только DOM готова
$(document).ready(main);
```

Хотя в C++ есть функциональные указатели, нельзя сказать, что шаблоны наподобие показанного ранее используются часто. Аналогично, по крайней мере на момент написания, в Java нет простого механизма для создания функций, принимающих другие функции в качестве параметров.

В следующих главах мы будем иметь возможность писать функции, которые принимают другие функции в качестве параметров. Это будет очень полезно, когда мы приступим к более сложному асинхронному программированию.

Условия

Одна из первых структур управления, которую мы изучаем в каждом языке, — это условие `if`. Оно позволяет нам сообщить интерпретатору, что какой-либо блок кода должен быть выполнен только в том случае, если некоторое условие правдиво:

```
var count = 101;
if (count > 100) {
  console.log("Значение count больше 100");
}
//=> the count is bigger than 100
count = 99;
if (count > 100) {
  console.log("Значение count больше 100");
}
//=> Ничего не введено
```

Оператор `else` позволяет нам сделать что-то другое в случае, если условие ложно:

```
var count = 99;
if (count > 100) {
  console.log("Значение count больше 100");
} else {
  console.log("Значение count меньше 100");
}
//=> Значение count меньше 100
```

Иногда необходимо сделать несколько взаимоисключающих вещей в зависимости от разных условий. В этом случае пригодится шаблон `if — else — if`:

```
var count = 150;
if (count < 100) {
  console.log("Значение count меньше 100");
} else if (count <= 200) {
  console.log("Значение count между 100 и 200 включительно");
} else {
  console.log("Значение count больше 200");
}
//=> Значение count между 100 и 200 включительно
```

А лучше всего то, что условия не обязательно должны быть простыми. Мы можем использовать операторы `&&` (логическое «И»), `||` (логическое «ИЛИ») и `!` («НЕ») для создания более сложных условий (как упоминалось в главе 2, символ `|` находится выше и правее клавиши Enter на клавиатуре — вы должны нажать Shift, чтобы набрать его, а не обратный слэш):

```
// Проверка того, что *любые* условия правдивы
if (cardRank === "король" || cardRank === "дама" || cardRank === "валет") {
  console.log("Это лицевая карта!");
} else {
  console.log("Ничего крупного!");
}
```

```
// проверка того, является ли карта тузом пик
if (cardRank === "туз" && cardSuit === "пик") {
  console.log("ЭТО ПИКОВЫЙ ТУЗ!");
} else {
  console.log("К сожалению, это не пиковый туз");
}
// проверка того, что карта *не* является пиковым тузом
// с помощью переворота выходных данных и оператора !
if (!(cardRank === "ace" && cardSuit === "пик")) {
  console.log("Эта карта — не пиковый туз!");
}
```

В JavaScript единичный знак равенства означает назначение, а тройной — сравнение, которое возвращает true или false, если левая и правая стороны выражения эквивалентны. В дальнейшем поговорим об этом более подробно.

Повторение

Часто нам приходится повторять какие-то действия несколько раз. Например, допустим, что нужно вывести первые 100 чисел. Конечно, можно сделать это примитивным способом:

```
console.log(0);
console.log(1);
console.log(2);
// ... ох…
console.log(99);
console.log(100);
```

Очень много кода! Гораздо проще использовать циклическую структуру, чтобы ввести все 100 чисел:

```
var num; // здесь будет число, которое мы вводим
// введем 101 цифру, начав с нуля
for (num = 0; num <= 100; num = num + 1) {
  console.log(num);
}
```

Здесь достигается тот же самый результат, что и в предыдущем коде, но гораздо более разумным способом. Это и есть пример цикла `for`.

Цикл `for` состоит из четырех элементов. Три из них — скобки, которые следуют за словом for. Обычно их называют *оператором инициализации* (initialization statement), *условием продолжения* (continuation condition) и *оператором приращения* (update statement). Четвертый элемент — это тело цикла, код, находящийся в фигурных скобках. Оператор инициализации выполняется, только когда тело цикла запускается в первый раз. Оператор приращения — каждый раз, когда тело

цикла выполнилось. А условие продолжения всегда проверяется на истинность перед запуском тела цикла (даже в первый раз).

Следующие два цикла `for` позволяют достичь одного и того же результатата — вывести все четные цифры менее 100:

```
var i;
// инициализация: присваиваем i значение 0
// условие продолжения: цикл выполняется, пока i строго меньше 100
// приращение: прибавляем 2 к i
// тело цикла: выводим i
// другими словами, вводим только целые числа, начиная с нуля
// и заканчивая 98
for (i = 0; i < 100; i = i + 2) {
console.log(i);
}
// инициализация: присваиваем i значение 0
// условие продолжения: цикл выполняется, пока i строго меньше 100
// приращение: прибавляем 2 к i
// выводим i, только если остаток деления на 2 равен 0
// вводим только четные числа, начиная с 0 и заканчивая 98
for (i = 0; i < 100; i = i + 1) {
  if (i%2 === 0) {
    console.log(i);
  }
}
```

Во втором примере мы использовали оператор остатка (`%`), который получает остаток от деления целых чисел. Например, результат `5%2` равен 1, так как 5/2 будет 2 и 1 в остатке. В примере мы используем этот оператор, чтобы проверить делимость на 2 (в этом случае число четное). Даже если этот оператор *выглядит* непривычно, во многих случаях он очень полезен. В частности, он полезен для опытных программистов, так как вопросы о нем нередко задают на собеседованиях соискателям должности программиста (см. практические задачи FizzBuzz в конце главы). Если вы знакомы с циклами `while` и `do...while` в Java или C++, то вам будет приятно узнать, что и JavaScript их поддерживает. Но я не хочу усложнять материал, поэтому в оставшейся части книги буду использовать только циклы `for` (а также `forEach` для массивов).

Массивы

Циклы — не только интересная, но и очень полезная в контексте массивов тема. Массивы — это индексированные собрания объектов JavaScript. Один из вариантов массивов — это единичная переменная, в которой мы можем хранить несколько значений. Вот простой пример создания массива:

```
var greetings = ["здравствуйте", "привет", "здорово", "салют", "алоха"];
//приветствия
```

Мы можем очень просто обобщить этот пример: создадим литерал массива с помощью фигурных скобок, а затем перечислим его элементы, разделяя их запятыми.

В этом массиве пять элементов, каждый из которых является текстовой строкой. Вот пример массива, где все элементы — целые числа:

```
var primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
```

Мы можем индексировать этот массив и взаимодействовать с отдельными элементами с помощью имени переменной и квадратных скобок. Нумерация элементов массива всегда начинается с 0 и заканчивается числом на 1 меньшим, чем длина массива:

```
console.log(greetings[1]);
//=> 'привет'
console.log(greetings[0]);
//=> 'здравствуйте'
console.log(primes[4]);
//=> 11
console.log(greetings[4]);
//=> 'алоха'
```

Аналогичным образом присваиваются значения отдельным элементам массива:

```
greetings[3] = "приветствую"; // меняет "салют" на "приветствую"
```

А что, если нужно набрать все элементы массива? Один из способов сделать это — использовать свойства массива `length` для создания условия продолжения цикла `for`:

```
var index;
for (index = 0; index < greetings.length; index = index + 1) {
  console.log(greetings[index]);
}
```

Это очень хороший способ, и он часто используется как в JavaScript, так и в других языках, например Java. Однако в JavaScript есть лучший, *джаваскриптовый*, способ достичь того же результата. С каждым массивом связан цикл `forEach`, который заставляет функцию работать с каждым элементом массива:

```
// Цикл forEach loop принимает функцию в качестве аргумента
// и запускает ее для каждого элемента массива
greetings.forEach(function (element) {
  console.log(element);
});
```

Это гораздо лучше, так как позволяет избежать использования лишних переменных, например `index`, как в примере ранее. А ведь нередко бывает, что удаление объявления переменных снижает количество ошибок в нашем коде.

В дополнение к включению функции `forEach` массивы JavaScript имеют еще несколько преимуществ над массивами в C++ или Java. Во-первых, они могут расти или уменьшаться динамически. Во-вторых, у них есть несколько встроенных функций, что позволяет быстро выполнять какие-то общие операции. Например, часто встречается необходимость добавить элементы в конец массива. Для этого можно использовать функцию `push`:

```
// создаем пустой массив
var cardSuits = [];
cardSuits.push("трефы");
console.log(cardSuits);
//=> ["трефы"]
cardSuits.push("бубны");
console.log(cardSuits);
//=> ["трефы", "бубны"]
cardSuits.push("червы");
console.log(cardSuits);
//=> ["трефы", "бубны", "червы"]
cardSuits.push("пики");
console.log(cardSuits);
//=> ["трефы", "бубны", "червы", "пики"]
```

В массивах JavaScript очень много других встроенных функций, но пока мы остановимся на `push`. Несколько других изучим в дальнейшем.

Массивы и циклы — очень нужные инструменты независимо от того, с каким языком программирования вы работаете. Единственный способ хорошо изучить их — это практиковаться в их создании и использовании. Несколько практических задач в конце главы позволят вам поупражняться в создании массивов и управлении ими с помощью циклов.

Использование JSLint для выявления возможных проблем

Как и в случае HTML и CSS, очень просто написать код JavaScript, который *работает*, но не соответствует принятым стандартам качества. Особенно легко это удается, когда вы только начинаете работу с языком. Например, посмотрите на следующий код:

```
cardRank = "король";
if (cardRank = "король") {
  console.log("карта – король!");
} else {
  console.log("карта – не король!");
}
//=> карта – король!
```

На первый взгляд код выглядит хорошо и вроде бы должен работать. Наберите его в Google Chrome в консоли JavaScript — и вы увидите тот результат, который ожидался. Но немного изменим код — пусть значение `cardRank` будет «дама»:

```
cardRank = "дама";
if (cardRank = "король") {
  console.log("карта – король!");
} else {
  console.log("карта – не король!");
}
//=> карта – король!
```

Теперь-то вы точно заметили ошибку. Дело в том, что условие `if` содержит оператор назначения вместо сравнения, в результате чего переменной всегда назначается псевдоправдивое значение[1]. Пока не обязательно понимать, что именно означает *псевдоправдивое* (truthy) и почему оно заставляет выполняться блок кода, но надо разобраться, из-за чего происходит ошибка: случайно было использовано = (назначение) вместо === (сравнение).

Исправим ее, поменяв назначение на сравнение:

```
if (cardRank === "король") {
  console.log("карта – король!");
} else {
  console.log("карта – не король!");
}
```

Но в этом коде есть еще одна скрытая проблема. К счастью для нас, существует аналог CSS Lint для JavaScript, называемый вполне логично JSLint. Домашняя страница JSLint показана на рис. 4.9.

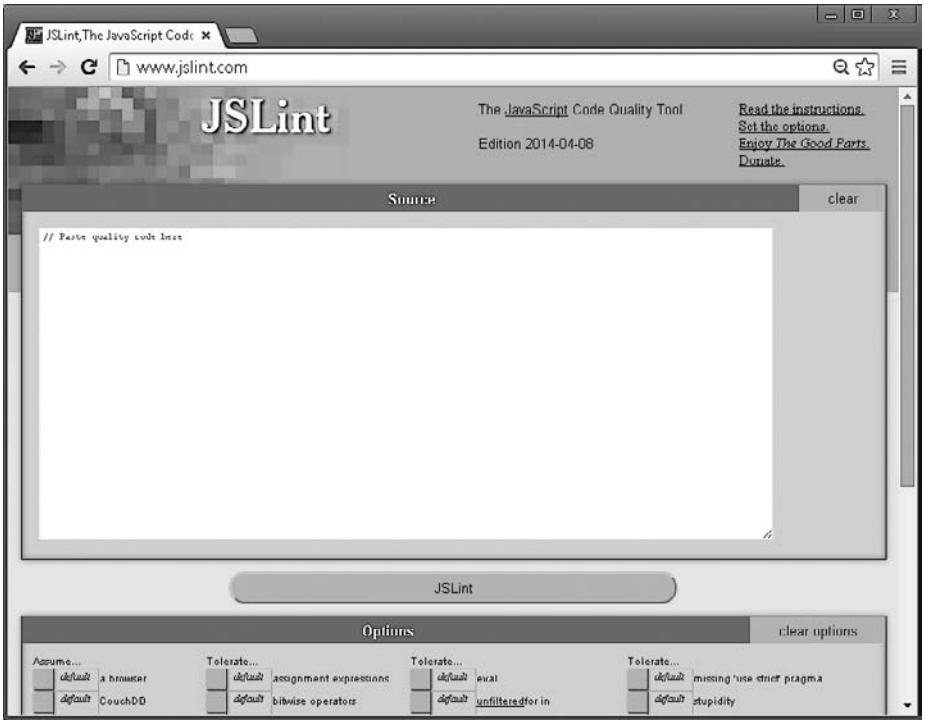


Рис. 4.9. Домашняя страница JSLint

[1] Автор использует жаргонные слова truthy и falsy для обозначения ситуации, когда значение переменной заранее преобразуется в логическое «истинно» или «ложно» соответственно. Емкий перевод обнаружился на Habrahabr и звучит как «значения правдочки и кривдочки». — *Примеч. пер.*

Вставим наш код в JSLint и посмотрим, что нам скажут. Сервис должен ответить так, как показано на рис. 4.10.

Рис. 4.10. Первые ошибки в JSLint

Первые ошибки, которые мы видим, относятся к скрытой ошибке, которую я упоминал ранее. JavaScript не требует обязательного объявления переменных перед использованием, но это часто может привести к непредсказуемым последствиям. Лучше все же объявить все переменные с помощью ключевого слова `var`. Изменим код так, чтобы он объявлял переменную в дополнение к ее определению:

```
var cardRank = "король";
```

Сделав это и перезапустив проверку, мы увидим, что первые две ошибки исчезли. Следующая — ошибка назначения, о которой мы уже говорили. Исправив ее, перейдем к двум последним.

Последние две ошибки связаны с глобальной переменной `console`, которая была использована до своего объявления. Мы не вправе просто добавить слово `var` перед этой переменной, поскольку не создаем ее. Но можно перейти к опциям JSLint, включить глобальную опцию `console`, `alert`... после чего эти ошибки исчезнут.

Как и CSS Lint, JSLint эмулирует присутствие рядом с вами опытного наставника, профессионала в JavaScript, который проверяет, как вы пишете код. Порой его замечания будут не совсем понятными и вам придется какое-то время разбираться, в чем состоит ошибка и как ее исправить (отличная стартовая точка для этой работы — Google), но все же оно того стоит, и постепенно вы научитесь писать код JavaScript более эффективно.

Добавление интерактивности Amazeriffic

А сейчас рассмотрим все наши идеи на примере. Работа с приложением Amazeriffic началась с определения цели — приложение должно выполнять функции списка задач. Для ее достижения построим интерфейс, состоящий из трех вкладок. Первая будет отображать список задач, начиная с самых новых (рис. 4.11).

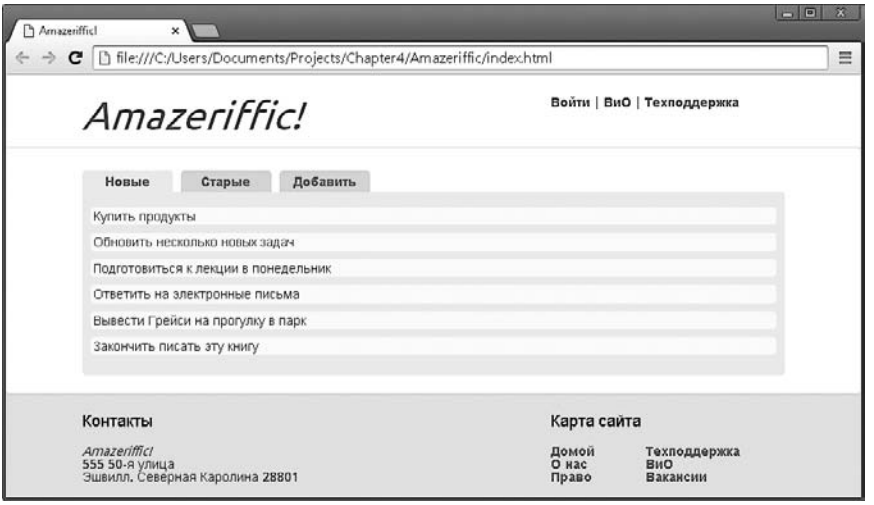


Рис. 4.11. Первая вкладка, где задачи отсортированы от новых к старым

Вторая вкладка будет отображать тот же самый список, но начиная с самых старых задач (рис. 4.12).

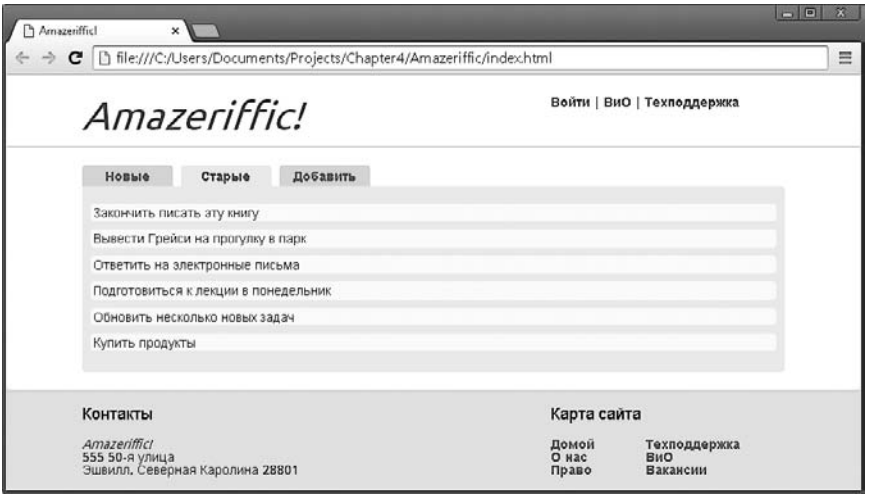


Рис. 4.12. Вторая вкладка, где задачи отсортированы от старых к новым

А на последней вкладке будет находиться поле ввода, с помощью которого можно добавлять новые задачи (рис. 4.13).

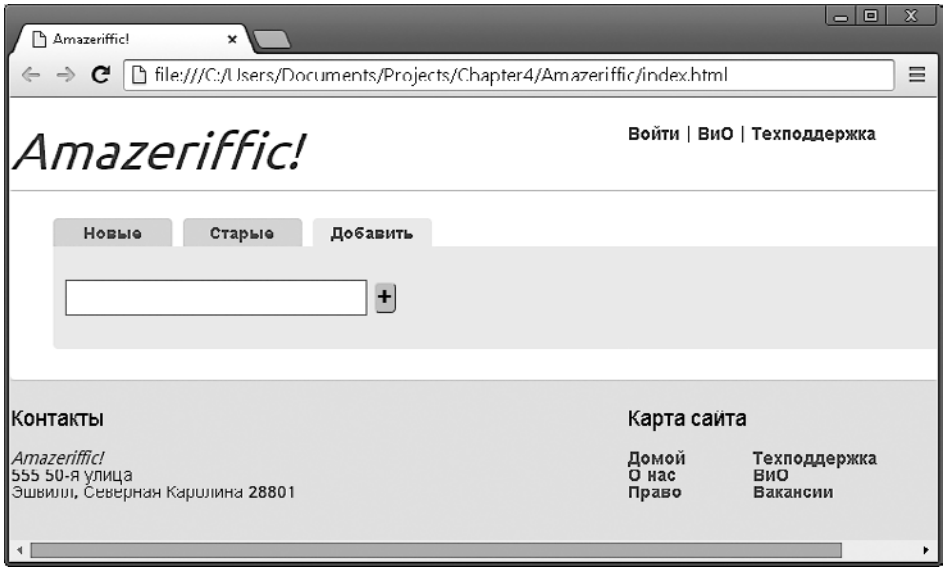


Рис. 4.13. Вкладка позволяет добавить новую задачу

Приступим

Поскольку мы уже много работали с Git к этому моменту, в этом примере я предоставляю вам возможность самостоятельно выбирать подходящие моменты для обновления хранилища. Просто убедитесь, что вы создали хранилище Git и регулярно обновляете его по мере рабоы с этим примером.

Вы также увидите, что большая часть дизайна повторяет пример из предыдущей главы. Поэтому вы можете просто скопировать свою работу (а именно HTML и CSS) из предыдущей главы. Только в этом случае нужно создать каталоги scripts и stylesheets для лучшей организации работы.

Структура и стиль

Вы можете начать с изменения элемента `main`, который содержит наш интерфейс. Мой HTML-код для этого элемента выглядит вот так:

```
<main>
  <div class="container">
    <div class="tabs">
      <a href=""><span class="active">Новые</span></a>
      <a href=""><span>Старые</span></a>
      <a href=""><span>Добавить</span></a>
```

```
    </div>
    <div class="content">
      <ul>
        <li>Купить продукты</li>
        <li>Обновить несколько новых задач</li>
        <li>Подготовиться к лекции в понедельник</li>
        <li>Ответить на письма нанимателей в LinkedIn</li>
        <li>Вывести Грейси на прогулку в парк</li>
        <li>Закончить писать книгу</li>
      </ul>
    </div>
  </div>
</main>
```

Вы можете использовать данную базовую структуру элемента `main`, а также свои наработки по стилю из главы 3. Вот как выглядят стилевые правила для моих вкладок:

```
.tabs a span {
  display: inline-block;
  border-radius: 5px 5px 0 0;
  width: 100px;
  margin-right: 10px;
  text-align: center;
  background: #ddd;
  padding: 5px;
}
```

Вы видите здесь один новый элемент: я использую свойство `display` со значением `inline-block`. Напомню, что внутристрочные элементы наподобие `span` не имеют свойства ширины (`width`), в отличие от блочных, но последние всегда начинаются с новой строки. Свойство `inline-block` создает гибридный элемент, давая элементу `span` свойство `width`, для которого я установил значение 100 пикселов. Таким образом, все вкладки у нас будут одной ширины.

Конечно, у меня есть дополнительный набор правил для активной вкладки. С его помощью цвет верхней части активной вкладки делается таким же, как у ее содержимого, в результате чего создается иллюзия объема интерфейса:

```
.tabs a span.active {
  background: #eee;
}
```

Интерактивность

А сейчас перейдем к действительно интересной части — JavaScript! Конечно, к этому моменту jQuery и `/javascripts/app.js` через элемент `script` уже должны быть включены в нижнюю часть элемента `body` в нашем HTML. Поэтому можем начать работу над файлом `app.js` с создания основной структуры программы:

```
var main = function () {
  "use strict";
```

```
  console.log("hello, world!");
};
$(document).ready(main);
```

Напомню, что если все было сделано правильно, то, открыв ее в окне браузера, мы увидим `hello, world!` в консоли JavaScript. Проверьте это.

Работа вкладок

Если все работает правильно, займемся переключением вкладок. Начнем с кода, который обрабатывает щелчки кнопкой мыши на каждой вкладке:

```
var main = function () {
  "use strict";
  $(".tabs a:nth-child(1)").on("click", function () {
    // делаем все вкладки неактивными
    $(".tabs span").removeClass("active");
    // делаем активной первую вкладку
    $(".tabs a:nth-child(1) span").addClass("active");
    // очищаем основное содержание, чтобы переопределить его
    $("main .content").empty();
    // возвращается false, так как мы не переходим по ссылке
    return false;
  });
  $(".tabs a:nth-child(2)").on("click", function () {
    $(".tabs span").removeClass("active");
    $(".tabs a:nth-child(2) span").addClass("active");
    $("main .content").empty();
    return false;
  });
  $(".tabs a:nth-child(3)").on("click", function () {
    $(".tabs span").removeClass("active");
    $(".tabs a:nth-child(2) span").addClass("active");
    $("main .content").empty();
    return false;
  });
};
```

Внимательно изучив код, вы заметите, что мы нарушили принцип DRY («Не повторяйся») — определенно здесь присутствует некоторое дублирование. Придется перевести какое-то количество кода в абстракции с помощью функции. Попробуйте сделать это прямо сейчас, не заглядывая вперед. Продолжайте. Я подожду.

Преобразование кода с помощью функции

Надеюсь, что какое-то время вы размышляли над решением, прежде чем прочесть его! Пришло ли вам в голову применить функцию, где в качестве аргумента используется номер вкладки? Если да — отлично! Если нет, все хорошо — скоро вы с этим разберетесь. Нужно всего лишь немного практики.

Это нормально, если ваше решение отличается от моего. Самое главное — постарайтесь разобраться в принципе работы этого решения:

```
var main = function () {
  "use strict";
  var makeTabActive = function (tabNumber) {
    // сконструируем селектор из tabNubmer
    var tabSelector = ".tabs a:nth-child(" + tabNumber + ") span";
    $(".tabs span").removeClass("active");
    $(tabSelector).addClass("active");
  };
  $(".tabs a:nth-child(1)").on("click", function () {
    makeTabActive(1);
    return false;
  });
  $(".tabs a:nth-child(2)").on("click", function () {
    makeTabActive(2);
    return false;
  });
  $(".tabs a:nth-child(3)").on("click", function () {
    makeTabActive(3);
    return false;
  });
};
```

Возможно, вы попробовали поместить `return false` внутрь функции `makeTabActive`, но это может привести к проблемам. Нужно оставить этот оператор внутри обработчика щелчков, так как обработчик должен вернуть `false`, иначе браузер будет пытаться перейти по ссылке.

Переработка кода с помощью цикла

Итак, мы немного сократили количество строк в коде и, более того, сделали это способом, позволяющим избежать ошибок. Но можно пойти еще дальше. Обратите внимание на то, как мы пометили вкладки: с помощью цифр 1, 2, 3. Если мы преобразуем их в цикл `for`, который будет перебирать все эти номера, то сможем настроить слушатели совсем просто! Попробуйте сделать это.

Таким образом мы даже избавимся от необходимости использования функции `makeTabActive`, так как поместим важные строки кода непосредственно в цикл. Мое решение выглядит вот так:

```
var main = function () {
  "use strict";
  var tabNumber;
  for (tabNumber = 1; tabNumber <= 3; tabNumber++) {
    var tabSelector = ".tabs a:nth-child(" + tabNumber + ") span";
    $(tabSelector).on("click", function () {
      $(".tabs span").removeClass("active");
      $(this).addClass("active");
      return false;
    });
  }
};
```

Переработка кода с помощью цикла forEach

Оказывается, есть еще одно решение! jQuery позволяет создать набор элементов, а затем перебрать их через массив. В этом упрощении мы переберем все элементы span внутри вкладок, создав обработчик щелчков (click) для каждой:

```
var main = function () {
  "use strict";
  $(".tabs a span").toArray().forEach(function (element) {
    // создаем обработчик щелчков для этого элемента
    $(element).on("click", function () {
      $(".tabs a span").removeClass("active");
      $(element).addClass("active");
      $("main .content").empty();
      return false;
    });
  });
};
```

Массив, который создает jQuery, является массивом элементов DOM, а не объектов jQuery. Нам нужно преобразовать их в объекты jQuery, поместив внутрь функционального вызова $().

Как видите, есть три способа решения подобных задач. Могут быть, конечно, и другие, но, думаю, два последних неплохи — они коротки, понятны (если, конечно, вы умеете работать с циклами), а кроме того, с их помощью легко добавлять дополнительные вкладки.

Управление содержимым

Теперь нужно решить другую проблему. Внутри вкладок элемент main .content должен заполняться по-разному в зависимости от того, какая вкладка выбрана. Это очень легко, если мы знаем, на каком потомке элемента .tabs находимся. Например, если активен первый потомок элемента .tabs, мы делаем одно, если второй — что-то другое. Но на самом деле это не так-то просто, потому что внутри обработчика событий мы имеем доступ только к элементу span, который является потомком интересующего нас элемента. Древовидная диаграмма этой ситуации изображена на рис. 4.14.

Оказывается, что в jQuery существует отличный способ выбрать предок объекта jQuery — функция, которая так и называется — parent. Но затем нужно определить, с которым по счету потомком мы имеем дело. Для этого в jQuery есть функция is, которая позволяет проверить селектор относительно текущего объекта jQuery. Наверное, понять эту абстракцию нелегко, но на практике все это вполне читаемо. Вот короткий пример:

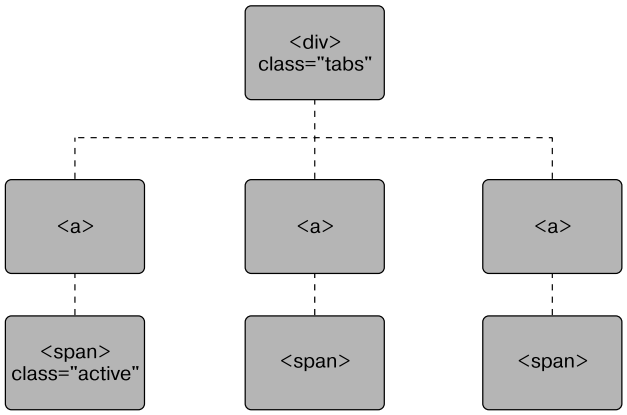


Рис. 4.14. Нам нужно знать индекс элемента span, на котором мы щелкнули

```
// проверим, является ли предок объекта jQuery $me
// первым потомком своего собственного предка
if ($me.parent().is(":first-child")) {
  console.log("МОЙ ПРЕДОК — ПЕРВЫЙ ПОТОМОК!!");
} else {
  console.log("Мой предок не является первым потомком.");
}
```

Мы можем использовать этот паттерн и селектор `:nth-child`, чтобы определить необходимые действия для примера с вкладками:

```
var main = function () {
  "use strict";
  $(".tabs a span").toArray().forEach(function (element) {
    // создаем обработку щелчков для этого элемента
    $(element).on("click", function () {
      // поскольку мы используем версию элемента jQuery,
      // нужно создать временную переменную,
      // чтобы избежать постоянного обновления
      var $element = $(element);
      $(".tabs a span").removeClass("active");
      $element.addClass("active");
      $("main .content").empty();
      if ($element.parent().is(":nth-child(1)")) {
        console.log("Щелчок на первой вкладке!");
      } else if ($element.parent().is(":nth-child(2)")) {
        console.log("Щелчок на второй вкладке!");
      } else if ($element.parent().is(":nth-child(3)")) {
        console.log("Щелчок на третьей вкладке!");
      }
      return false;
    });
  });
};
```

Запустив этот код с открытой панелью JavaScript в Chrome, вы увидите соответствующее сообщение, появляющееся при щелчке на определенной вкладке! Супер-пупер просто!

Настройка содержимого вкладки

Настройка содержимого вкладки требует несколько более сложной работы. Прежде всего нужно создать сами списки задач в виде массива строковых элементов. Для этого добавим переменную внутрь функции `main`, где будет храниться массив текстовых констант с задачами:

```
var main = function () {
  "use strict";
  var toDos = [
    "Закончить писать эту книгу",
    "Вывести Грейси на прогулку в парк",
    "Ответить на электронные письма",
    "Подготовиться к лекции в понедельник",
    "Обновить несколько новых задач",
    "Купить продукты"
  ];
  //... все остальное, относящееся к вкладкам
};
```

Теперь, когда появятся новые задачи, все, что необходимо сделать, — добавить их в конец массива. Таким образом получается, что старые задачи стоят в массиве первыми, а новые — последними. Вторая вкладка (Старые) будет перечислять задачи в том же порядке, что и в массиве, а первая вкладка (Новые) — в обратном.

Для начала я покажу, как будет перестраиваться контент в случае щелчка на второй вкладке, а затем предоставлю вам возможность создать содержимое остальных двух. Содержимое первой и второй вкладок создать очень просто: нужно добавить элемент `ul`, а затем с помощью циклической работы с массивом добавить элементы `li` для каждого элемента массива. Обратите внимание на то, что для первой вкладки нужно будет работать с элементами в обратном порядке, поэтому в данном случае мы используем традиционный цикл `for`. Для второй же можем воспользоваться более удобным циклом `forEach`. Код может выглядеть примерно так:

```
$(element).on("click", function () {
  var $element = $(element),
  $content;
  $(".tabs a span").removeClass("active");
  $element.addClass("active");
  $("main .content").empty();
  if ($element.parent().is(":nth-child(1)")) {
    console.log("Щелчок на первой вкладке!");
  } else if ($element.parent().is(":nth-child(2)")) {
    $content = $("<ul>");
```

```
    toDos.forEach(function (todo) {
      $content.append($("<li>").text(todo));
    });
    $("main .content").append($content);
  } else if ($element.parent().is(":nth-child(3)")) {
    console.log("Щелчок на третьей вкладке!");
  }
})
```

Как я уже упоминал, построение содержимого для первой вкладки выглядит аналогичным образом. После того как мы это сделаем, содержание, жестко прописанное в HTML, больше не нужно! Мы можем отправить (`trigger`) фальшивый щелчок на первой вкладке, добавив одну строку в конце функции `main`, сразу после обработчиков щелчков. Это динамически сконструирует содержимое:

```
$(".tabs a:first-child span").trigger("click");
```

Если все работает правильно, можно удалить жестко прописанные в `index.html` элементы.

Для третьей вкладки нужно проделать кое-что другое — создать элементы `input` и `button`, такие же, с которыми мы работали в примере с комментариями в начале главы. Но в данном случае нужно поупражняться в создании поддерева DOM с помощью jQuery вместо внедрения их в HTML. Следует также добавить обработчик событий для кнопки. В данном случае вместо добавления элемента в DOM, как мы делали в более раннем примере, нужно просто поместить его в массив `ToDo` с помощью функции `Push`.

Завершенный пример вы можете видеть в хранилище Git.

Подведем итоги

В этой главе мы изучили, как сделать веб-приложение интерактивным с помощью jQuery и JavaScript. jQuery — широко используемая библиотека, которая берет на себя некоторые трудности управления DOM и обработки событий, а также многое другое. Из-за того что запущенный в браузере пользовательский интерфейс является событийно-управляемым, появляется необходимость создания *асинхронного* кода. Самый простой способ начать — это прикрепление функций *обратных вызовов* к событиям.

JavaScript является полнофункциональным языком программирования, поддерживаемым всеми браузерами. Для начинающего веб-разработчика и программиста очень важно приобрести опыт в работе с несколькими фундаментальными концепциями, включающими переменные, условия `if` и `if-else`, а также разнообразные циклические структуры. Массивы — еще один необходимый элемент во всех языках программирования, поэтому научиться создавать их и управлять ими также очень важно.

JSLint — сервис, аналогичный CSS Lint или HTML Validator, — поможет вам избежать самых распространенных ошибок в JavaScript. Очень полезно время от времени прогонять через него свой код.

Больше теории и практики

Заучивание

Добавим несколько шагов к упражнениям на запоминание.

1. Создайте папку под названием javascripts для хранения файлов `.js`.
2. Создайте простую программу JS и сохраните ее в файле с названием `app.js`. Файл должен или выдавать предупреждение «hello, world!», или вводить его в консоль. Это должно быть организовано в функции `main`, которая должна вызываться функцией jQuery `document.ready`.
3. Импортируйте сценарий вместе с jQuery в свой документ HTML в нижней части тега `<body>`.
4. Заново откройте страницу в Chrome и убедитесь, что все работает хорошо.
5. Добавьте файл и изменения в HTML в хранилище Git.

Плагины jQuery

Одна из причин популярности jQuery — существование крупного сообщества веб-разработчиков, которые создают плагины, позволяющие добавлять на вашу страницу невероятные эффекты. Я рекомендую посетить страницу http://plugins.jquery.com и посмотреть, какие плагины там предлагают. Кроме того, советую поэкспериментировать с некоторыми плагинами. Придется почитать документацию по плагинам, чтобы научиться работать с ними, но, поверьте, это отличное вложение вашего времени.

Один из моих любимых плагинов называется *colorbox*. Он позволяет легко добавить на свою страницу анимированную фотогалерею. Автор написал очень простую документацию по работе с этим плагином.

Селекторы jQuery

Наберите следующий код HTML и сохраните его в файле `selectorpractice.html`:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Привет</h1>
    <h2 class="important">Снова привет</h2>
    <p>Случайные несвязанные абзацы</p>
    <div class="relevant"> //relevant — релевантный (см.ниже)
      <p class="a">first</p>
      <p class="a">second</p>
```

```
      <p>third</p>
      <p>fourth</p>
      <p class="a">fifth</p>
      <p class="a">sixth</p>
      <p>seventh</p>
    </div>
  </body>
</html>
```

Затем создайте простой файл `app.js` с нижеследующим содержимым. Используйте два тега `<script>`, чтобы импортировать jQuery и вновь созданный файл в нижнюю часть элемента `body`:

```
var main = function () {
  "use strict;"
  $("*").css("color","red");
};
$(document).ready(main);
```

Функция `css` позволяет менять стиль выбранных элементов с помощью jQuery. По умолчанию код использует универсальный селектор CSS и меняет цвет каждого элемента DOM на красный. Изменим его так, чтобы красными становились только определенные элементы. Например, если мы хотим сделать красным только тег `<h1>`, нужно использовать следующий селектор:

```
$("h1").css("color","red");
```

Следующие несколько шагов будут куда проще, если вы просмотрите документацию jQuery по селекторам. Кроме селекторов, с которыми вы уже знакомы из этой главы, обратите особое внимание на селекторы `:not` и `:gt`.

1. Выберите элементы `h2` по их классу.
2. Выберите первый абзац из релевантных абзацев.
3. Выберите третий абзац из релевантных абзацев.
4. Выберите все релевантные абзацы на странице.
5. Выберите все релевантные абзацы.
6. Выберите второй, четвертый и шестой релевантные абзацы.
7. Выберите седьмой релевантный абзац.
8. Выберите пятый, шестой и седьмой релевантные абзацы.
9. Выберите все релевантные абзацы, у которых нет класса `a`.

Задача FizzBuzz

Если вы подумываете о том, чтобы работать программистом, вам обязательно нужно уметь решать задачку FizzBuzz. Насколько я знаю, эта задача стала популярной после поста в блоге Джеффа Этвуда под названием «Почему программисты не могут... программировать?» *(Why Can't Programmers... Program?)*. В посте Джефф

указывает на тот факт, что многие люди, работающие программистами, не могут решить эту простую задачу.

Вот как выглядит эта задача: *напишите программу, которая выводит числа от 1 до 100. Но вместо тех, что делятся на 3, пусть будет выведено Fizz, а вместо тех, что делятся на 5, — Buzz. Вместо тех же, что делятся и на 3, и на 5, пусть будет выведено FizzBuzz.*

Вы вполне можете решить эту задачку с помощью цикла `for`, а также других инструментов, которым я научил вас в этой главе. Используйте серии условий `if-else` и оператор остатка от деления.

Я должен добавить, что не считаю эту задачу особенно хорошим тестом на способность кого-либо программировать, и не думаю, что ненайденное решение может быть поводом для увольнения программиста из-за несоответствия занимаемой должности. Существует множество причин, по которым кто-либо может не справиться с этой задачей, не имеющих никакого отношения к написанию компьютерных программ на уровне начинающего. Но все-таки что есть, то есть, и, возможно, когда-нибудь вам зададут этот вопрос. Поэтому постарайтесь найти решение и запомнить его.

Упражнения в работе с массивами

Еще один часто встречающийся источник вопросов на собеседованиях — массивы. Это легко объяснимо: определенно это самая часто встречающаяся структура данных в компьютерных программах, в той или иной форме они присутствуют в каждом языке программирования. Попробуйте ответить на все вопросы в этом разделе с помощью материала данной главы. Начнем с простого вопроса с несколькими вариантами ответа: напишите функцию, которая принимает массив чисел в качестве аргумента и возвращает их сумму.

Самое простое решение будет выглядеть вот так:

```
var sum = function (nums) {
  var sumSoFar = 0,
  i;
  // цикл перебирает все элементы массива, по очереди прибавляя их к сумме
  for (i = 0; i < nums.length; i++) {
    sumSoFar = sumSoFar + nums[i];
  }
  // Сейчас, после перебора всего массива,
  // в переменной sumSoFar должна находиться
  // сумма всех элементов
  return sumSoFar;
};
sum([1,2,3,4]);
//=> 10
```

Аналогично мы можем использовать цикл `forEach`:

```
var sum = function (nums) {
  var sumSoFar = 0;
```

```
  // use a forEach loop
  nums.forEach(function (value) {
    sumSoFar = sumSoFar + value;
  });
  return sumSoFar;
};
sum([1,2,3,4]);
//=> 10
```

Цикл `forEach` более предпочтителен, так как с ним нет необходимости в использовании переменной `i`. В общем случае это неплохо — удаление переменной, регулярно изменяющей свое состояние в программе, снижает вероятность появления в нашем коде ошибок. В самом деле, если мы хотим отделаться от всех временных локальных переменных, то можем использовать функцию `reduce`, которая выполнит то же самое более красиво:

```
var sum = function (nums) {
  return nums.reduce(function (sumSoFar, value) {
    return sumSoFar + value;
  }, 0);
};
sum([1,2,3,4]);
//=> 10
```

Я не собираюсь больше уделять время функции `reduce`, но если она вас заинтересовала, то вы можете почитать о ней подробнее в Интернете.

Следует также отметить, что мы никак не проверяем корректность вводимых данных для нашей функции. Например, что, если мы попробуем отправить нечто, не являющееся массивом?

```
sum(5);
//=> TypeError!
sum("hello, world");
//=> TypeError!
```

Существует множество способов это исправить, но пока не стоит тратить на это время. Но, если вы пишете код, например, на собеседовании, будет очень неплохо проверять вводные данные для каждой функции.

Вот еще несколько вопросов, которые помогут вам поупражняться в работе с массивами.

1. Напишите функцию, которая принимает в качестве аргумента массив чисел, а затем возвращает их среднее.
2. Напишите функцию, которая принимает в качестве аргумента массив чисел, а затем возвращает наибольшее число в массиве.
3. Напишите функцию, которая принимает массив чисел, а затем возвращает `true`, если он содержит хотя бы одно четное число, и `false` — в ином случае.
4. Напишите функцию, которая принимает массив чисел, а затем возвращает `true`, если *каждое* число в массиве четное, и `false` — в противном случае.

5. Напишите функцию, которая принимает два аргумента: массив строк и отдельную строку — и возвращает `true`, если строка уже содержится в массиве, и `false` — в противном случае. Например, функция может работать вот так:

```
arrayContains(["привет", "мир"], "привет");
//=> true
arrayContains(["привет", "мир"], "пока");
//=> false
arrayContains(["привет", "мир", "пока"], "пока");
//=> true
```

6. Напишите функцию, которая похожа на предыдущую, но возвращает `true` только в случае, если массив содержит данную строку по крайней мере дважды:

```
arrayContainsTwo(["a","b","a","c"], "a");
//=> true
arrayContainsTwo(["a","b","a","c"], "b");
//=> false
arrayContainsTwo(["a","b","a","c","a"], "a");
//=> true
```

После того как вам удалось это сделать, напишите функцию под названием `arrayContainsThree`, которая работает аналогичным образом, но для трех вместо двух. Словом, мы можем обобщить задачу. Напишите функцию, которая принимает три аргумента и возвращает `true`, если массив содержит указанный элемент *n* раз, где `n` — третий аргумент:

```
arrayContainsNTimes(["a","b","a","c","a"], "a", 3);
//=> true
arrayContainsNTimes(["a","b","a","c","a"], "a", 2);
//=> true
arrayContainsNTimes(["a","b","a","c","a"], "a", 4);
//=> false
arrayContainsNTimes(["a","b","a","c","a"], "b", 2);
//=> false
arrayContainsNTimes(["a","b","a","c","a"], "b", 1);
//=> true
arrayContainsNTimes(["a","b","a","c","a"], "d", 0);
//=> true
```

Проект Эйлера (Project Euler)

Еще один отличный источник практических задач — проект Эйлера (Project Euler). Там находится множество практических задач, для корректного решения которых не требуется глубоких знаний. Большинство из них сводятся к нахождению определенного числа. Получив решение, вы вводите его на сайте и, если ответ правильный, получаете доступ к обсуждениям, где другие люди публикуют свои решения и ищут лучшие способы.

Однажды на собеседовании в известной IT-компании мне задали вопрос, который почти слово в слово повторял задачу из проекта Эйлера.

Другие материалы по JavaScript

JavaScript — популярный язык, и поэтому о нем написано множество книг. К сожалению, Дуг Крокфорд клеймит большинство из них как «совершенно кошмарные. Они содержат ошибки, плохие примеры и прививают неправильные навыки». Я склонен согласиться с ним (и надеюсь, что он не поместит в эту категорию и мою книгу!).

Как я уже говорил, мне не удалось найти хорошую книгу, которая должным образом освещала бы программирование на JavaScript для новичков или тех, кто только приступает к изучению программирования. В то же время существует несколько великолепных книг для программистов среднего и продвинутого уровня. Я очень рекомендую книгу Крокфорда *JavaScript: The Good Parts* (O'Reilly, 2008), как одну из лучших по программированию на JavaScript в целом. Я также думаю, что в книге Дэвида Германа *Effective JavaScript* (Addison-Wesley, 2012) содержится множество хороших практических советов для программистов на JavaScript среднего уровня.

Если же вы ищете общую практическую информацию по разработке ПО с помощью JavaScript, я очень рекомендую вам книгу Ника Закаса *Maintanable JavaScript* (O'Reilly, 2012). И как только вы освоитесь с объектно-ориентированным программированием на JavaScript, обратитесь к книге Майкла Фогаса *Functional JavaScript* (O'Reilly, 2013), которая предлагает иные, но очень увлекательные перспективы.

5 Мост

Мы уже почти закончили увлекательное путешествие по клиентской стороне веб-приложения. Хоть я не уточнял этого, когда мы обсуждали *клиентскую сторону*, но я говорю сейчас о той части приложения, которая запускается в браузере. Другая сторона медали — серверная часть приложения, которая запускает и хранит информацию, находящуюся вне вашего браузера, обычно на удаленном компьютере.

Эта глава не о серверной стороне нашего веб-приложения, а о многообразии технологий, которые позволяют клиенту и серверу обмениваться информацией самым простым путем. Я обычно представляю себе этот набор технологий как *мост* между клиентом и сервером.

Говоря конкретнее, мы с вами изучим объекты JavaScript, JSON (JavaScript Object Notation) и AJAX (Asynchronous JavaScript And XML — не совсем верное название). Изучение этих тем подготовит нас к работе с Node.js, который мы будем изучать в следующей главе.

Привет, объекты JavaScript!

Прежде чем начать разговор о передаче данных между компьютерами, нужно обсудить один важный примитив JavaScript — объекты. Вы, может быть, слышали ранее об *объектно-ориентированном программировании*, и если вам приходилось программировать на C++ или Java, то вы хорошо знакомы с этой темой.

Хотя эти принципы очень важны для разработки программного обеспечения в целом, объектно-ориентированное программирование на JavaScript реализовано несколько иначе, так что лучше всего забыть о них, начиная изучать объекты. Пока нам стоит рассматривать их несколько упрощенно, полагая, что объекты — это просто коллекции переменных, относящихся к одной конкретной сущности. Начнем с примера.

Представление карточной игры

В предыдущей главе мы рассмотрели несколько примеров с игрой в карты. Игральная карта имеет два основных свойства: масть (пики, трефы, бубны или червы)

и номинал (цифровые от 2 до 10, а также старшие карты — валет, дама, король и туз). Углубимся в этот пример и создадим веб-приложение для игры в покер. Это потребует от нас представления пяти карт в руке с помощью JavaScript.

Самый простой подход с использованием инструментов, с которыми я вас познакомил, потребует использования десяти переменных, по одной для каждого номинала и по одной для каждой масти:

```
var cardOneSuit = "червей",
    cardOneRank = "двойка",
    cardTwoSuit = "пик",
    cardTwoRank = "туз",
    cardThreeSuit = "пик",
    cardThreeRank = "пятерка",
    // ...
    cardFiveSuitRank = "семерка";
```

Надеюсь, вы понимаете, что это очень громоздкое решение. И если как следует изучили предыдущую главу, то, наверное, думаете, что массив из пяти элементов мог бы все упростить! Но проблема в том, что у каждой карты по два атрибута. И что же нам делать с этим массивом?

Можно предположить решение, которое выглядит так:

```
var cardHandSuits = ["червей", "пик", "пик", "треф", "бубен"],
    cardHandRanks = ["двойка", "туз", "пятерка", "король", "семерка"];
```

Определенно уже лучше, но основная проблема предыдущего решения осталась: нет конкретной связи между отдельным номиналом и отдельной мастью — мы должны удерживать все связи в голове. Всякий раз, когда в программе есть какие-либо сущности, тесно связанные между собой, но эти связи существуют только в сознании программиста, надо сделать вывод, что решение неоптимально. А на самом деле это и не обязательно, если мы будем использовать объекты!

Объект — это просто собрание разнообразных переменных, каким-то образом связанных между собой. Для его создания нужно использовать фигурные скобки, после чего можно получить доступ к внутренним переменным объекта с помощью оператора «точка» (`.`). Вот пример создания отдельной карты:

```
// создадим объект "карта" номиналом 2 (two)
// червовой масти (hearts)
var cardOne = { "rank":"двойка", "suit":"червей" };
// Выведем номинал cardOne
console.log(cardOne.rank);
//=> двойка
//Выведем масть cardOne
console.log(cardOne.suit);
//=> червей
```

Создав объект, мы всегда можем изменить его в дальнейшем. Например, изменить масть `suit` и достоинство `rank` для `cardOne`:

```
// изменим карту на туз пик
cardOne.rank = "туз";
```

```
cardOne.suit = "пик";
console.log(cardOne);
//=> Object {rank: "туз", suit: "пик"}
```

Как и в случае с массивом, можем создать пустой объект, а позже добавить ему атрибуты:

```
// создаем пустой объект
var card = {};
// присвоим ему номинал туза
card.rank = "туз";
console.log(card);
//=> Object {rank: "туз"}
// присвоим масть червей
card.suit = "червей";
console.log(card);
//=> Object {rank: "туз", suit: "червей"}
```

Сейчас, если нам требуется представить раздачу карт, мы можем создать массив и заполнить его карточными объектами вместо создания двух отдельных массивов!

```
// создаем пустой массив
var cards = [];
// отправляем в массив двойку червей
cards.push( {"rank": "двойка", "suit":"червей"} );
cards.push( {"rank": "туз", "suit":"пик"} );
cards.push( {"rank": "пятерка", "suit":"пик"} );
cards.push( {"rank": "король", "suit":"треф"} );
cards.push( {"rank": "семерка", "suit":"бубен"} );
// открываем первую и третью карты в раздаче
console.log(cards[0]);
//=> Object {rank: "двойка", suit: "червей"}
console.log(cards[2]);
//=> Object {rank: "пятерка", suit: "пик"}
```

Можно создать и один большой массив, в буквальном смысле представляющий раздачу карт:

```
// создаем раздачу карт
// с помощью большого массива
var cards = [
    {"rank": "двойка", "suit":"червей"},
    {"rank": "туз", "suit":"пик"},
    {"rank": "пятерка", "suit":"пик"},
    {"rank": "король", "suit":"треф"},
    {"rank": "семерка", "suit":"бубен"}
]
```

Подведем итоги

Как я упоминал ранее, мы можем думать об объектах JavaScript как о собрании переменных, каждая из которых обладает именем и значением. Чтобы создать

пустой объект, просто вводим две фигурные скобки — открывающую и закрывающую:

```
// создаем пустой объект
var s = {};
```

Затем можем добавить переменные к объекту с помощью оператора «точка» (.):

```
s.name = "Сэмми";
```

Переменные внутри объекта могут быть любого типа, включая строки, которые мы видели во всех предыдущих примерах, а также массивы или даже объекты:

```
s.age = 36;  // число, означающее возраст
s.friends = [ "Марк", "Эмили", "Брюс", "Сильван" ]; // друзья
s.dog = { "name":"Грейси", "breed":"Австралийская овчарка" };
/ собака по имени Грейси породы австралийская овчарка*/
console.log(s.age);
//=> 36
console.log(s.friends[1]);
//=> "Эмили"
console.log(s.dog);
//=> Object {name: "Грейси", breed: " Австралийская овчарка"}
console.log(s.dog.name);
//=> "Грейси"
```

Мы также можем создавать литералы, которые являются, по сути, просто объектами, полностью определенными в коде:

```
var g = {
    "name": "Гордон",
    "age": 36,
    "friends": ["Сара", "Энди", "Роджер", "Брендон"],
    "dog": {"name":"Пи", "breed":"Метис лабрадора" }
}
console.log(g.name);
//=> "Гордон"
console.log(g.friends[2]);
//=> "Роджер"
console.log(g.dog.breed);
//=> "Метис лабрадора"
```

При необходимости мы можем использовать специальную величину `null`, что значит «нет объекта»:

```
var b = {
    "name": "Джон",
    "age" : 45,
    "friends" : [ "Сара", "Джим" ],
    "dog" : null
}
```

В этом примере у Джона нет собаки. Мы можем использовать `null` также для того, чтобы обозначить окончание работы с этим объектом:

```
// назначаем currentPerson объекту g
var currentPerson = g;
// ... выполняем какие-то действия с currentPerson
// устанавливаем currentPerson на null
currentPerson = null;
```

Будем использовать величину `null` в качестве заполнителя для объекта, особенно часто — в главе 6, когда будем говорить об ошибках. В частности, функции обратного вызова будут активизироваться величиной `null` в том случае, когда запрос не содержит ошибок.

Вообще говоря, объекты предоставляют нам множество возможностей для хранения данных и управления ими. Надо также отметить, что мы можем хранить в объектах переменные, так как функциональные переменные в JavaScript работают так же, как и любые другие. Например, мы уже знаем, как можно получить доступ к функциям, которые связаны с объектами jQuery:

```
// получаем элемент DOM
var $headerTag = $("h1");
// $headerTag – объект, у которого есть связанная с ним функция,
// называемая fadeOut
$headerTag.fadeOut();
```

Объекты со связанными функциями — очень мощный инструмент для абстракций (что и привело к идее объектно-ориентированного программирования), но мы, пожалуй, отложим их обсуждение на некоторое время. Вместо этого в данной главе мы сконцентрируемся на использовании объектов для обмена информацией с другими веб-приложениями.

Обмен информацией между компьютерами

В наши дни почти невозможно создать веб-приложение, не затрагивая других уже существующих приложений. Например, вы хотите реализовать возможность авторизации ваших пользователей через аккаунт Twitter. Или ваши пользователи должны получать обновления от вашего приложения в свою ленту в Facebook. Это значит, что у вашего приложения должна быть возможность обмена информацией с этими сервисами. Стандартный формат, используемый в веб-приложениях сегодня, называется JSON, и, если вы хорошо освоили работу с объектами JavaScript, значит, уже можете работать с JSON!

JSON

Объект JSON — это не что иное, как объект JavaScript в виде строки (с некоторыми техническими нюансами, но большинство из них мы пока что можем игнорировать). Это значит, что, если нам нужно отправить какую-либо информацию на другой

сервис, мы просто создаем в коде объект JavaScript, преобразуем его в строку и отправляем. В большинстве случаев библиотеки AJAX сделают за нас основную работу, и для программиста это выглядит так, будто программы просто обмениваются объектами!

Например, предположим, что я хочу определить объект JSON как внешний файл — я могу просто кодировать его, словно внутри программы JavaScript:

```
{
  "rank":"десятка",
  "suit":"червей"
}
```

Оказывается, что в большинстве языков программирования очень просто взять строку JSON и сконвертировать ее в объект, который может использовать компьютерная программа! А в JavaScript это и вовсе не представляет никаких сложностей, так как сам по себе этот формат — просто строковая версия литерала объекта. Большинство окружений JavaScript поддерживают объекты JSON, с которыми мы можем взаимодействовать. Например, откройте консоль Google Chrome и введите следующее:

```
// создаем строку JSON в единичных кавычках
var jsonString = '{"rank":"десятка", "suit":"червей"}'
// JSON.parse преобразует ее в объект
var card = JSON.parse(jsonString);
console.log(card.rank);
//=> "ten" десятка
console.log(card.suit);
//=> "червей"
```

Обратите внимание на то, что мы создали строку jsonString с использованием одинарных кавычек вместо двойных. Дело в том, что для JavaScript не имеет значения, какие именно вы используете, но для JSON это важно. Поскольку нужно создать строку внутри строки, мы используем одинарные кавычки снаружи и двойные внутри.

Можно также преобразовать объект JSON в строку с помощью функции `stringify`:

```
console.log(JSON.stringify(card));
//=> {"rank":"ten","suit":"червей"}
```

А сейчас, когда мы выяснили, как создавать объекты JSON во внешних файлах или в виде строк в наших программах, посмотрим, как можно обмениваться ими между компьютерами.

AJAX

AJAX расшифровывается как Asynchronous JavaScript And XML (асинхронный JavaScript и XML, что, как я уже говорил, не совсем верно). Общий формат обмена

данными, предшествовавший JSON, назывался XML и больше напоминал HTML. Но несмотря на то, что XML по-прежнему широко используется, продвижение к JSON — очень серьезный шаг с момента изобретения AJAX.

Несмотря на то что аббревиатура полна технических терминов, на самом деле она вовсе не сложна. Основная идея AJAX — возможность для приложения получать информацию и отдавать ее другим компьютерам без перезагрузки веб-страницы. Один из первых примеров этой технологии (и пока что один из лучших) — приложение Google для электронной почты, Gmail, появившееся на сцене около 10 лет назад! Если вы пользуетесь им, то, наверное, замечали, как новое письмо *магическим образом* появляется в ваших входящих. Другими словами, вам не нужно перезагружать страницу, чтобы получить свежую почту. Это и есть пример AJAX.

Доступ к внешнему файлу JSON

Ну, пожалуй, хватит теории. Посмотрим на работу AJAX в действии. Начнем с создания структуры приложения. На этом этапе обойдемся без CSS и просто выполним пример для работы. Надеюсь, к этому моменту вы уже способны создать по памяти страницу, которая приветствует нас фразой «Hello, World!» с помощью элемента `h1` и имеет пустой элемент `main`. Нам также нужно включить элемент `script` со ссылкой на jQuery из CDN (мы говорили об этом в главе 4), а также тег `<script>`, содержащий ссылку на файл `app.js` — основное приложение JavaScript. Этот файл должен находиться в папке **javascripts** и выглядеть примерно так:

```
var main = function () {
    "use strict";
    console.log("Hello, World!");
}
$(document).ready(main);
```

Запустите базовое приложение в Chrome и откройте консоль JavaScript. Как обычно, если все идет хорошо, вы увидите в логе слова «Hello, World!».

Затем нужно создать новую папку внутри каталога с нашим примером и назвать ее **cards**. Внутри нее мы создадим файл `aceOfSpades.json`, который будет выглядеть как одно из наших предыдущих определений объектов JavaScript:

```
{
  "rank" : "туз",
  "suit" : "пик"
}
```

Сейчас мы хотим получить доступ к этому файлу из нашей программы с помощью AJAX, что в данный момент составляет определенную проблему.

Ограничения браузера по безопасности

На момент появления JavaScript был предназначен для работы в браузере. Это означало, что по соображениям безопасности доступ к локальным файлам, хранящимся

на вашем компьютере, запрещается. Можете себе представить проблемы, которых нельзя было бы избежать в противном случае, — любой сайт, посещенный вами, получал бы полный доступ к вашему компьютеру!

Само собой разумеется, что это ограничение по безопасности актуально и до сих пор. В то же время у нас есть право доступа к определенным типам файлов с того же самого сервера, откуда был получен сам файл JavaScript. Файлы JSON являются отличным примером: я могу сделать любой запрос AJAX на сервер и получить любые файлы JSON, которые он делает доступными. К сожалению, мы не будем работать с сервером до начала чтения следующей главы, так что пока придется обойти эти ограничения с помощью Chrome.

Не следует выходить в Интернет при отключенных настройках безопасности. Убедитесь, что вы перезапустили браузер перед тем, как заходить на сайты, за исключением созданного вами самостоятельно.

Чтобы запустить Chrome без этого ограничения, следует сначала закрыть его. Затем, если вы работаете в Mac, откройте командную строку и введите следующую команду:

```
hostname $ open -a Google\ Chrome --args --allow-file-access-from-files
```

Если вы работаете в Windows, то можете сделать то же самое, щелкнув на кнопке Пуск, набрав `run` в строке поиска, а затем введя в окне запуска следующую команду:

```
%userprofile%\AppData\Local\Google\Chrome\Application\chrome.exe
-allow-file-access-from-files
```

Таким образом, Chrome должен открыться с отключенным ограничением по безопасности, упомянутым ранее. Как я уже сказал, мы используем данный прием только для примера в этой главе — в следующей главе будем запускать приложение с реального сервера, и в отключении ограничения по безопасности необходимости не будет. А сейчас убедимся, что все сработало.

Функция getJSON

Если вам удалось успешно отключить ограничение по безопасности в Chrome между сайтами, вы легко получите доступ к локальному файлу JSON из нашей программы. Чтобы сделать это, надо использовать функцию в jQuery, которая называется `getJSON`. Как и большинство примеров JavaScript, этот запрос jQuery будет асинхронным, поэтому нужно добавить обратный вызов:

```
var main = function () {
    "use strict";
    // getJSON сразу интерпретирует JSON, поэтому
    // нет необходимости вызывать JSON.parse
    $.getJSON("cards/aceOfSpades.json", function (card) {
```

```
        // вводим карту в консоль
        console.log(card);
    });
};
$(document).ready(main);
```

Если все было сделано верно, то, открыв страницу в Chrome (с отключенными ограничениями по безопасности), мы увидим, что карта появилась в консоли. Это значит, что отныне мы можем использовать ее в нашей программе, как любой другой объект JavaScript:

```
var main = function () {
    "use strict";
    console.log("Hello, World!");
    $.getJSON("cards/aceOfSpades.json", function (card) {
        // создаем элемент для хранения карты
        var $cardParagraph = $("<p>");
        // создаем
        $cardParagraph.text(card.rank + " of " + card.suit);
        // связываем абзац с картой с main
        $("main").append($cardParagraph);
    });
}
$(document).ready(main);
```

Массив JSON

Мы можем создавать в файле и более сложные объекты JSON. Например, объект может состоять из массива объектов, а не единичного объекта. Чтобы проверить это на практике, создайте следующий файл и назовите его `hand.json`:

```
[
  { "suit" : "пик",       "rank" : "туз" },
  { "suit" : "червей",    "rank" : "десятка" },
  { "suit" : "пик",       "rank" : "пятерка" },
  { "suit" : "треф",      "rank" : "тройка" },
  { "suit" : "бубен",     "rank" : "тройка" }
]
```

Добавьте следующий `getJSON` сразу за предыдущим:

```
var main = function () {
    "use strict";
    console.log("Hello, World!");
    $.getJSON("cards/aceOfSpades.json", function (card) {
        // создаем элемент для хранения карты
        var $cardParagraph = $("<p>");
        // создаем текст для карты
        $cardParagraph.text(card.rank + " of " + card.suit);
        // присоединяем абзац с картой к элементу main
        $("main").append($cardParagraph);
```

```
    });
    $.getJSON("cards/hand.json", function (hand) {
        var $list = $("<ul>");
        // hand – массив, поэтому мы можем применить к нему итерационный процесс
        // с помощью цикла forEach
        hand.forEach(function (card) {
            // создаем элемент списка для хранения карты
            // и присоединяем его к списку
            var $card = $("<li>");
            $card.text(card.rank + " of " + card.suit);
            $list.append($card);
        });
        // присоединяем список к элементу main
        $("main").append($list);
    });
};
$(document).ready(main);
```

В этом примере мы повторно используем переменную card в цикле forEach во втором вызове JSON. Так происходит потому, что переменная выходит из работы в конце первого вызова к JSON. Мы не будем сейчас углубляться в правила охвата переменных в JavaScript, но я подумал, что следует объяснить этот момент во избежание недопонимания.

После запуска приложения мы получим страницу, выглядящую как показано на рис. 5.1.

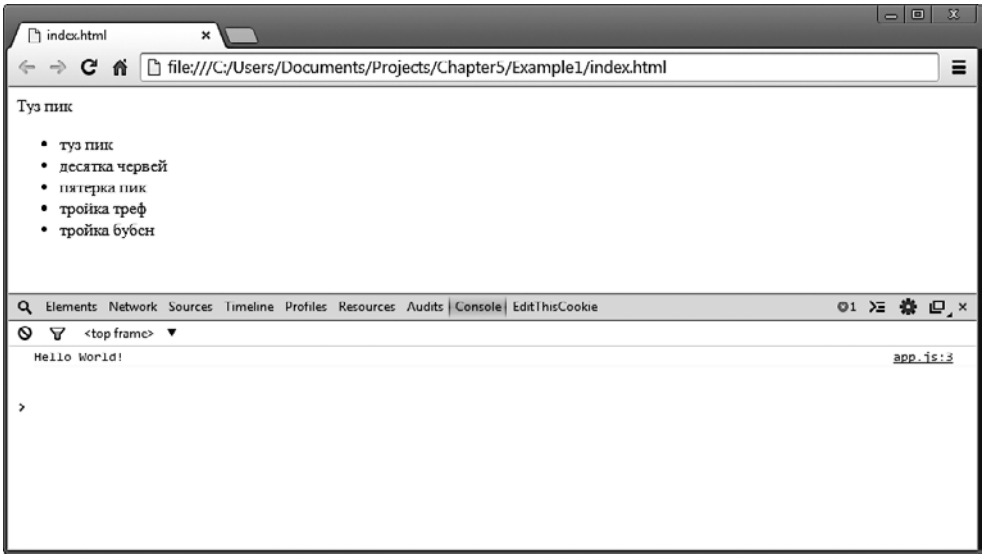


Рис. 5.1. Наше первое приложение AJAX

Что же дальше?

Я понимаю, что этот пример не слишком впечатляющий. На самом деле вы, наверное, подумали, что было бы гораздо проще определить эти объекты прямо внутри кода. Зачем же понадобился отдельный файл?

Что ж, представьте, что данные поступают к вам с отдельного компьютера. В этом случае очень пригодится возможность доступа к ним из нашего кода. Фактически изначально мы можем даже не знать, как выглядит объект! Например, представьте, что мы хотим получить самые свежие изображения собак, загруженные на Flickr.

Получение изображений с Flickr

Сделаем еще один скелет приложения, на этот раз с CSS. Откройте Sublime и перейдите в вашу папку Chapter5, которая находится в Projects. Создайте каталог под названием Flickr. Внутри него создайте привычные папки javascripts и stylesheets, а также файлы `app.js` и `style.css`. Сделайте так, чтобы `app.js` выводил «Hello, World!» в консоль JavaScript. Создайте также файл `index.html`, который свяжет все файлы вместе для создания основы веб-приложения.

Вау! Если вы смогли сделать все это по памяти, то вы далеко пойдете! Если же пару раз заглядывали в книгу, то все равно неплохо, но попробуйте все-таки запомнить этапы основной подготовки и поупражняться выполнять их. Чем больше основной работы вы сможете делать по памяти, тем лучше сможете фокусировать свой мозг на сложных и интересных задачах.

Вот как выглядит мой HTML-файл:

```
<!doctype html>
<html>
  <head>
    <title>Приложение для Flickr</title>
    <link rel="stylesheet" href="stylesheets/style.css">
  </head>
  <body>
    <header>
    </header>
    <main>
     <div class="photos">
     </div>
    </main>
    <footer>
    </footer>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="javascripts/app.js"></script>
  </body>
</html>
```

А сейчас мы просто обобщим пример из документации jQuery, которая находится по адресу http://api.jquery.com. Немного изменим его, чтобы он лучше соответствовал

нашим целям. В этом примере мы используем JavaScript для получения изображений с Flickr, сервиса Yahoo! для публикации изображений. Но перед тем как сделать это, откройте браузер и введите в адресную строку следующую ссылку: http://api.flickr.com/services/feeds/photos_public.gne?tags=dogs&format=json.

Ответ должен появиться прямо в вашем браузере в виде JSON. Мой, например, выглядит вот так:

```
jsonFlickrFeed({
   "title": "Recent Uploads tagged dogs",
   "link": "http://www.flickr.com/photos/tags/dogs/",
   "description": "",
   "modified": "2013-10-06T19:42:49Z",
   "generator": "http://www.flickr.com/",
   "items": [
       {
           "title": "Huck and Moxie ride stroller",
           "link": "http://www.flickr.com/photos/animaltourism/10124023233/",
           "media": {"m":"http://bit.ly/1bVvkn2"},
           "date_taken": "2013-09-20T09:14:25-08:00",
           "description": "...description string...",
           "published": "2013-10-06T19:45:14Z",
           "author": "nobody@flickr.com (animaltourism.com)",
           "author_id": "8659451@N03",
           "tags": "park dog beagle dogs brooklyn ride stroller prospect hounds"
       },
       {
           "title": "6th Oct Susie: \"You know that thing you\'re eating?\"",
           "link": "http://www.flickr.com/photos/cardedfolderol/10123495123/",
           "media": {"m":"http://bit.ly/1bVvbQw"},
           "date_taken": "2013-10-06T14:47:59-08:00",
           "description": "...description string...",
           "published": "2013-10-06T19:14:22Z",
           "author": "nobody@flickr.com (Cardedfolderol)",
           "author_id": "79284220@N08",
           "tags": "pets dogs animal mammal canine"
       },
       {
           "title": "6th Oct Susie ready to leave",
           "link": "http://www.flickr.com/photos/cardedfolderol/10123488173/",
           "media": {"m":"http://bit.ly/1bVvpXJ"},
           "date_taken": "2013-10-06T14:49:59-08:00",
           "description": "...description string...",
           "published": "2013-10-06T19:14:23Z",
           "author": "nobody@flickr.com (Cardedfolderol)",
           "author_id": "79284220@N08",
           "tags": "pets dogs animal mammal canine"
       }
   ]
});
```

Это выглядит несколько сложнее, чем примеры, виденные вами ранее, но основная структура и формат остаются теми же самыми. Он начинается с основной информации о запросе, а затем выводит свойство `items`, которое представляет собой массив изображений. Каждый элемент массива содержит другой объект, `media`, у которого есть свойство `m`, включающее в себя ссылку на изображение. В главе 2 мы изучили, как добавить изображение в HTML-документ с помощью тега `<img>` — сейчас он нам и пригодится.

Проделаем все необходимое шаг за шагом. Начнем с добавления строки в функцию `main`, которая объявляет URL как переменную, после чего можем вызвать функцию jQuery `getJSON` точно таким же образом, что и раньше:

```
var main = function () {
    "use strict";
    // на самом деле это всего одна строка,
    // но я разделил ее на несколько
    // для улучшения восприятия
    var url = "http://api.flickr.com/services/feeds/photos_public.gne?" +
              "tags=dogs&format=json&jsoncallback=?";
    $.getJSON(url, function (flickrResponse) {
        //пока мы просто выводим ответ в консоль
        console.log(flickrResponse);
    });
};
$(document).ready(main);
```

Сейчас, если все хорошо, запустив этот код, мы увидим, что выдаваемый Flickr объект выводится в консоли и можно изучить его с помощью прокрутки вверх-вниз. Это поможет выявить источник проблем, если таковые появятся в дальнейшем.

Затем изменим код так, чтобы вместо вывода целого объекта программа выводила каждый URL по отдельности. Другими словами, мы проходим по всем элементам объекта с помощью цикла `forEach`:

```
$.getJSON(url, function (flickrResponse) {
    flickrResponse.items.forEach(function (photo) {
        console.log(photo.media.m);
    });
});
```

Таким образом, в консоли будет выведена последовательность URL — и вы даже сможете щелкать на них и видеть изображения! Теперь наконец мы можем по-настоящему отправить их в DOM. Чтобы сделать это, используем функцию jQuery `attr`, с которой мы пока не сталкивались. Используем ее, чтобы вручную установить атрибут `src` для тега `<img>`:

```
$.getJSON(url, function (flickrResponse) {
    flickrResponse.items.forEach(function (photo) {
        // создаем новый элемент jQuery для помещения в него изображения
        var $img = $("<img>");
        // помещаем в атрибут URL,
```

```
        // хранящийся в ответе Flickr
        $img.attr("src", photo.media.m);
        // прикрепляем тег img к элементу
        // main.photos
        $("main .photos").append($img);
    });
});
```

Сейчас, перезагрузив страницу, мы и в самом деле увидим изображения с Flickr! И если изменим в первоначальном URL тег на какой-нибудь другой (вместо dog — «собака»), целая страница изменится! И как обычно, мы можем значительно улучшить отображение страницы с помощью эффектов jQuery:

```
$.getJSON(url, function (flickrResponse) {
    flickrResponse.items.forEach(function (photo) {
        // создаем новый элемент jQuery для хранения изображений
        // но пока скрываем его
        var $img = $("<img>").hide();
        // устанавливаем атрибут для URL,
        // находящегося в ответе
        $img.attr("src", photo.media.m);
        // прикрепляем тег к функции main
        // элемента photos, а затем отображаем его
        $("main .photos").append($img);
        $img.fadeIn();
    });
});
```

Сейчас при повторном открытии страницы изображения будут появляться постепенно. В практических задачах в конце главы мы изменим этот код так, чтобы изображения появлялись на странице по очереди, одно за другим.

Добавление теговой функциональности в Amazeriffic

Сейчас, когда мы научились работать с объектами JavaScript и JSON, может быть полезно немного попрактиковаться и внедрить некоторые изученные функциональности в приложение Amazeriffic. В этом примере добавим теги для каждого элемента задачи. Мы можем использовать эти теги для сортировки списка задач другим, более удобным способом. В дополнение мы можем инициализировать список задач из файла JSON вместо массива, жестко помещенного в код.

Для начала скопируем всю папку Amazeriffic из каталога Chapter4 в Chapter5. Это даст нам отличную стартовую точку: не придется переписывать весь код.

Затем добавим файл JSON, в котором будет находиться список задач. Мы можем сохранить файл под названием `todos.json` в основной папке проекта (в той, где находится файл `index.html`):

```
[
    {
        "description" : "Купить продукты",
        "tags"  : [ "шопинг", "рутина" ]
    },
    {
        "description" : "Сделать несколько новых задач",
        "tags"  : [ "писательство", "работа" ]
    },
    {
        "description" : "Подготовиться к лекции в понедельник",
        "tags"  : [ "работа", "преподавание" ]
    },
    {
        "description" : "Ответить на электронные письма",
        "tags"  : [ "работа" ]
    },
    {
        "description" : "Вывести Грейси на прогулку в парк",
        "tags"  : [ "рутина", "питомцы" ]
    },
    {
        "description" : "Закончить писать книгу",
        "tags"  : [ "писательство", "работа" ]
    }
]
```

Вы видите, что файл JSON содержит массив с задачами и каждая задача имеет свой массив с текстовыми элементами, которые и являются тегами. Наша цель — добиться того, чтобы теги работали как вспомогательный способ организации списка задач.

Чтобы достичь этого, нужно добавить немного кода jQuery, который будет читать файл JSON. Но, поскольку функция `main`, рассмотренная в предыдущей главе, зависит от списка задач, нужно модифицировать ее так, чтобы вызов к `getJSON` происходил до вызова `main`. Для этого добавим анонимную функцию в вызов `document.ready`, который будет вызывать `getJSON`, а затем вызывать `main` с результатом общей работы:

```
var main = function (toDoObjects) {
    "use strict";
    // как main имеет доступ к списку задач!
};
$(document).ready(function () {
    $.getJSON("todos.json", function (toDoObjects) {
        // вызов функции main с аргументом в виде объекта toDoObjects
        main(toDoObjects);
    });
});
```

Появилась небольшая проблема: код не будет работать, так как мы изменили структуру объекта со списком задач. Ранее это был массив со строками, содержащими

описание задачи, а сейчас это массив объектов. Если нам нужно, чтобы код работал как раньше, можно создать массив старого типа из нового с помощью функции `map`.

Функция map

Функция `map` берет массив и делает из него новый, применяя функцию к каждому элементу. Запустите консоль в Chrome и попробуйте следующее:

```
// создаем массив из чисел
var nums = [1, 2, 3, 4, 5];
// применим функцию map
// для создания нового массива
var squares = nums.map(function (num) {
    return num*num;
});
console.log(squares);
//=> [1, 4, 9, 16, 25]
```

В этом примере функция, которая возвращает `num*num`, применяется к каждому элементу для создания нового массива. Этот пример может показаться экзотическим, но посмотрите на другой, более интересный:

```
// создаем массив имен
var names = [ "эмили", "марк", "брюс", "андреа", "пабло" ];
// а сейчас мы создадим массив,
// где первая буква каждого имени прописная
var capitalizedNames = names.map(function (name) {
    // получаем первую букву
    var firstLetter = name[0];
    // возвращаем ее в виде прописной
    // в строку, начинающуюся с индекса 1
    return firstLetter.toUpperCase() + name.substring(1);
});
console.log(capitalizedNames);
//=> ["Эмили", "Марк", "Брюс", "Андреа", "Пабло"]
```

Как видите, мы создали массив из имен с первой прописной буквой, даже не перебирая элементы самого исходного массива!

Сейчас, если вы поняли, как работает функция `map`, очень легко создать новый массив из старого:

```
var main = function (toDoObjectss) {
    "use strict";
    var toDos = toDoObjects.map(function (toDo) {
          // просто возвращаем описание
          // этой задачи
          return toDo.description;
    });
    // сейчас весь старый код должен работать в точности как раньше!
    // ...
};
$(document).ready(function () {
```

```
    $.getJSON("todos.json", function (toDoObjects) {
        // вызываем функцию main с задачами в качестве аргумента
        main(toDoObjects);
    });
});
```

А сейчас, когда весь старый код работает, как раньше, мы можем создать вкладку Теги.

Добавление вкладки Теги

Начнем с добавления вкладки Теги в пользовательский интерфейс. Поскольку мы сделали код (относительно) гибким, это будет не очень сложно. Начнем с открытия файла `index.html` и добавления кода для вкладки под названием Теги между вкладками Старые и Добавить:

```
<div class="tabs">
  <a href=""><span class="active">Новые</span></a>
  <a href=""><span>Старые</span></a>
  <a href=""><span>Теги</span></a>
  <a href=""><span>Добавить</span></a>
</div>
```

С помощью этой дополнительной строки мы добавляем вкладку в пользовательский интерфейс. Теперь (почти) все будет работать так, как нужно. Единственная проблема состоит в том, что создание вкладок основывается на их расположении в списке, поэтому, щелкнув на вкладке Теги, мы увидим интерфейс для вкладки Добавить. Вот это точно не то, что нам нужно, но, к счастью, требует лишь небольших изменений.

Все, что необходимо сделать, — добавить дополнительный блок `else-if` в середину кода, обеспечивающего работу вкладок, и слегка переставить цифры. Когда я проделал это в моем коде, соответствующий фрагмент стал выглядеть вот так:

```
} else if ($element.parent().is(":nth-child(3)")) {
    // ЭТО КОД ДЛЯ ВКЛАДКИ ТЕГИ
    console.log("Щелчок на вкладке Теги");
} else if ($element.parent().is(":nth-child(4)")) {
    $input = $("<input>"),
    $button = $("<button>").text("+");
    $button.on("click", function () {
        toDos.push($input.val());
        $input.val("");
    });
    $content = $("<div>").append($input).append($button);
}
```

Создание пользовательского интерфейса

Сейчас, когда мы знаем, как все устроено, подумаем, к чему нужно стремиться на этой вкладке. Посмотрите на рис. 5.2.

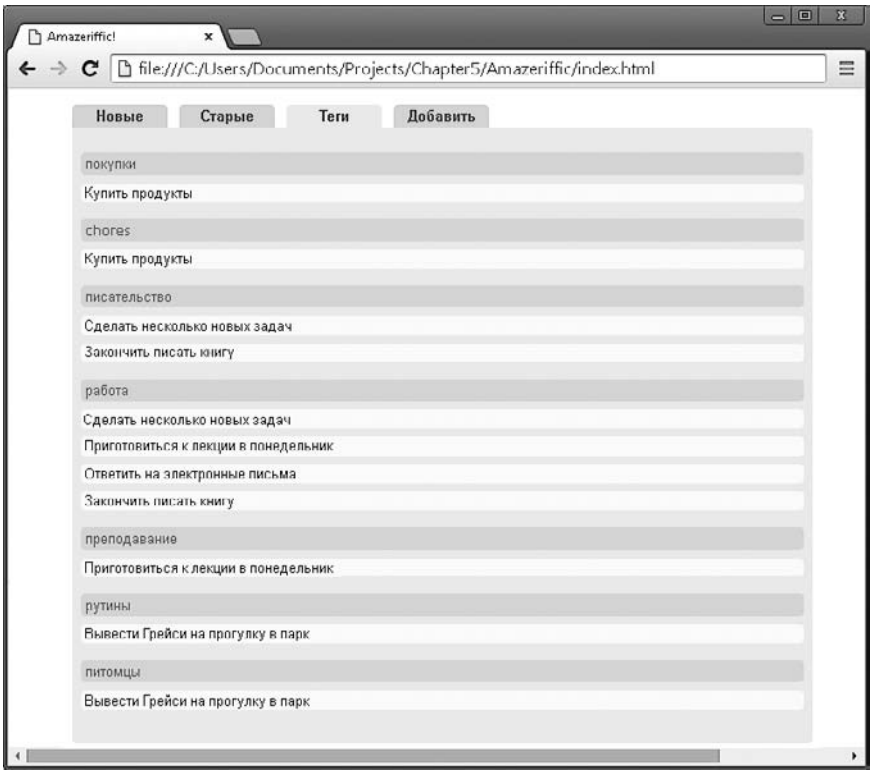


Рис. 5.2. Наша цель для вкладки Теги

На этой вкладке все имеющиеся теги должны быть перечислены как заголовки, а после них добавлены все описания задач, подходящие под эту категорию. Это значит, что некоторые задачи могут появляться в некоторых местах.

Проблема здесь, однако, еще и в том, что способ сохранения объекта JSON несколько затрудняет задачу. Было бы лучше, если бы `toDoObjects` хранились в формате, организованном с помощью тегов:

```
[
    {
        "name": "покупки",
        "toDos": ["Купить продукты"]
    },
    {
        "name": "рутина",
        "toDos": ["Купить продукты", "Вывести Грейси на прогулку в парк"]
    },
    {
        "name": "писательство",
        "toDos": ["Сделать несколько новых задач", "Закончить писать книгу"]
    },
    {
```

```
        "name": "работа",
           "toDos": ["Сделать несколько новых задач", "Подготовиться к лекции
в понедельник","Ответить на электронные письма", "Закончить писать книгу"]
    },
    {
        "name": " преподавание",
        "toDos": ["Подготовиться к лекции в понедельник"]
    },
    {
        "name": "питомцы",
        "toDos": ["Вывести Грейси на прогулку в парк "]
    }
]
```

К счастью, мы без труда можем изменить изначальный объект `toDoObjects` и привести его к такому виду с помощью серий вызовов функций `map` и `forEach`! Но пока оставим эту трансформацию для следующего раздела и сконцентрируемся на создании пользовательского интерфейса. Пока пропишем этот формат (или его упрощенную версию) жестко в коде для вкладки Теги как переменную под названием `organizedByTag`:

```
} else if ($element.parent().is(":nth-child(3)")) {
    // ЭТО КОД ДЛЯ ВКЛАДКИ ТЕГИ
    console.log("щелчок на вкладке Теги");
    var organizedByTag = [
        {
            "name": "покупки",
            "toDos": ["Купить продукты "]
        },
        {
            "name": "рутина",
            "toDos": ["Купить продукты", "Вывести Грейси на прогулку в парк "]
        },
        /* и т. д. */
    ];
}
```

Сейчас наша цель — выполнить итерации с этим объектом, добавив новую секцию в элемент страницы `.content`, как мы уже делали. Для этого просто добавим элемент `h3` вместе с `ul` и `li` для каждого тега. Элементы `ul` и `li` организованы так же, как в предыдущей главе, поэтому стиль может остаться тем же самым. Я также добавил стили для элементов `h3` в файле `style.css`, чтобы они выглядели так же, как в предыдущем примере:

```
} else if ($element.parent().is(":nth-child(3)")) {
    // ЭТО КОД ДЛЯ ВКЛАДКИ ТЕГИ
    console.log("щелчок на вкладке Теги");
    var organizedByTag = [
        /*  и т. д.*/
    ]
    organizedByTag.forEach(function (tag) {
```

```
        var $tagName = $("<h3>").text(tag.name),
            $content = $("<ul>");
        tag.toDos.forEach(function (description) {
            var $li = $("<li>").text(description);
            $content.append($li);
        });
        $("main .content").append($tagName);
        $("main .content").append($content);
    });
}
```

Осталось проделать всего две вещи. Нужно модифицировать вкладку Добавить так, чтобы вместе с новой задачей добавлялся и список категорий, а также решить, как преобразовать текущий список задач в список, организованный с помощью тегов. Начнем с последнего. Таким образом наше приложение будет динамически обновляться при добавлении новых элементов.

Создание промежуточной структуры данных о тегах

Когда я сталкиваюсь с проблемой, которую не получается решить в несколько строк кода, что-то подсказывает мне о необходимости применить абстракцию в виде функции. В данном случае нужно создать функцию с названием `organizeByTags`, которая принимает объект, хранящийся в нашей программе, в таком виде:

```
[
    {
        "description" : "Сделать покупки",
        "tags"  : [ "шопинг", "рутина" ]
    },
    {
        "description" : "Сделать несколько новых задач",
        "tags"  : [ "писательство", "работа" ]
    },
    /* и т. д. */
]
```

и преобразует его в объект, выглядящий примерно так:

```
[
        {
            "name": "шоппинг",
            "toDos": ["Сделать покупки"]
        },
        {
            "name": "рутина",
            "toDos": ["Сделать покупки", "Вывести Грейси на прогулку в парк"]
        },
        /* и т. д. */
]
```

После того как у нас появилась функция, мы можем легко модифицировать ее и поместить на место жестко прописанного в коде объекта:

```
} else if ($element.parent().is(":nth-child(3)")) {
    // ЭТО КОД ДЛЯ ВКЛАДКИ ТЕГИ
    console.log("щелчок на вкладке Теги");
    var organizedByTag = organizeByTag(toDoObjects);
}
```

Настройка тестовой платформы

Мы можем проверить результаты нашей работы в консоли Chrome, но постепенно, по мере разрастания и усложнения функций сделать это будет становиться труднее. В таких случаях я предпочитаю создать внешнюю программу JavaScript, которая содержит функцию и тестирует ее, выводя в консоль результат работы. Если я доволен тем, что вижу, то преобразую результат в рабочий код.

Чтобы сделать это, лучше всего иметь какой-нибудь неиспользуемый файл HTML, находящийся где-то вне вашей файловой иерархии. Этот файл понадобится только для того, чтобы запустить сценарий, в котором мы и будем ставить эксперименты, поэтому HTML не нуждается в нашей стандартной «рыбе», а может содержать лишь следующее:

```
<script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
<script src="test.js"></script>
```

Сейчас, сохранив этот файл как `index.html` и создав файл `test.js` следующего содержания:

```
var toDoObjects = [
    {
        "description" : "Сделать покупки",
        "tags"  : [ "шопинг", "рутина" ]
    },
    {
        "description" : "Сделать несколько новых задач",
        "tags"  : [ "писательство", "работа" ]
    },
    /* etc */
];
var organizeByTags = function (toDoObjects) {
    console.log("organizeByTags вызвана");
};
var main = function () {
    "use strict";
    var organizeByTags = function () {
        console.log("organizeByTags вызвана");
    };
    organizeByTags(toDoObjects);
};
$(document).ready(main);
```

мы должны увидеть в консоли сообщение «organizeByTag вызвана», а также пример объекта, который настроили.

Я рекомендую вам, после того как вы справитесь с этим, уделить немного времени самостоятельному решению задачи. Она довольно хитрая, но, увидев решение, вы, скорее всего, в нем разберетесь. Однако попытка найти его покажет вам, как вы можете справляться самостоятельно.

Добавлю, что есть несколько способов решения, как часто бывает в компьютерном программировании. Я покажу вам свое решение, но лишь в качестве ориентира, чтобы вы поняли, насколько оно отличается от вашего.

Мое решение

Мое решение относительно просто объяснить в два этапа. Сначала я создаю массив, который содержит все возможные теги, с помощью перебора изначальной структуры с помощью `forEach`. После того как я получу необходимый результат, я использую функцию `map` для связи своего массива с тегами с желаемым объектом с помощью перебора списка задач и выделения тех, у которых есть такой тег.

Начнем с первого этапа. Единственная новая для вас вещь здесь — функция `indexOf`, которая работает с любыми массивами. Мы можем видеть, как это работает, из консоли Chrome:

```
var nums = [1, 2, 3, 4, 5];
nums.indexOf(3);
//=> 2
nums.indexOf(1);
//=> 0
nums.indexOf(10);
//=> -1
```

Упрощенно это можно изложить так: если объект содержится в массиве, функция возвращает индекс этого объекта (все индексы начинаются с `0`). Если же в массиве его нет, функция возвращает `-1`. Это работает в том числе для строк:

```
var msgs = ["hello", "goodbye", "world"];
msgs.indexOf("goodbye");
//=> 1
msgs.indexOf("hello");
//=> 0
msgs.indexOf("HEY!");
//=> -1
```

Эта функция понадобится нам, чтобы избежать добавления в массив дубликатов:

```
var organizeByTags = function (toDoObjects) {
    // создание пустого массива для тегов
    var tags = [];
    // перебираем все задачи toDos
    toDoObjects.forEach(function (toDo) {
        // перебираем все теги для каждой задачи
```

```
        toDo.tags.forEach(function (tag) {
            // убеждаемся, что этого тега
            // еще нет в массиве
            if (tags.indexOf(tag) === -1) {
                tags.push(tag);
            }
        });
    });
    console.log(tags);
};
```

Запустив этот код, увидим выведенные наименования тегов без дубликатов. Значит, мы уже на полпути к успеху! Во второй части решения я использую функцию `map`:

```
var organizeByTags = function (toDoObjects) {
    /* первая часть, приведенная выше */
    var tagObjects = tags.map(function (tag) {
        // здесь мы находим все задачи,
        // содержащие этот тег
        var toDosWithTag = [];
        toDoObjects.forEach(function (toDo) {
            // проверка, что результат
            // indexOf is *не* равен -1
            if (toDo.tags.indexOf(tag) !== -1) {
                toDosWithTag.push(toDo.description);
            }
        });
        // мы связываем каждый тег с объектом, который
        // содержит название тега и массив
        return { "name": tag, "toDos": toDosWithTag };
    });
    console.log(tagObjects);
};
```

Сейчас, когда у нас есть функция и она корректно работает, можем вставить ее в функцию `main` кода приложения, и он будет работать! Как я уже говорил, эта проблема может быть решена различными способами, так что будет здорово испробовать какие-нибудь еще.

Теги как часть входных данных

Итак, мы добились того, чтобы элементы списка задач организовывались и отображались с помощью тегов, но как мы будем добавлять теги к новым элементам, которые вводим с помощью вкладки Добавить? Это требует модификации кода, который отвечает за отображение интерфейса вкладки. Я предлагаю преобразовать его так, чтобы интерфейс выглядел как на рис. 5.3.

Рис. 5.3. Цель для вкладки Добавить

Как вы видите, там находятся два поля ввода. Во втором из них пользователь может ввести список тегов, разделенных запятыми, которые затем ассоциируются с добавляемым объектом.

Если вы до конца разобрались с примером из предыдущей главы, то ваш код, вероятно, выглядит примерно так:

```
} else if ($element.parent().is(":nth-child(4)")) {
    var $input = $("<input>"),
        $button = $("<span>").text("+");
    $button.on("click", function () {
        toDos.push($input.val());
        $input.val("");
    });
    $content = $("<div>").append($input).append($button);
}
```

Чтобы наш пользовательский интерфейс выглядел так, как показано на рис. 5.3, нужно добавить дополнительное поле ввода и метки. Затем мы модифицируем обработчик `$button` для создания массива с тегами и вставки его в объекты.

Единственное, что нам нужно для этого знать, — как разделить строковый объект. Оказывается, что у всех строк есть встроенная функция, называемая `split` (разделить), которая именно это и делает — разделяет единичную строку на массив строк. Как и в случае с большинством изученных нами функций, ее работу можно наблюдать внутри консоли Chrome:

```
var words = "hello,world,goodbye,world";
// это разделяет массив в соответствии с запятыми
```

```
var arrayOfWords = words.split(",");
console.log(arrayOfWords);
//=> ["hello","world","goodbye","world"]
arrayOfWords[1];
//=> "world"
arrayOfWords[0];
//=> "hello"
```

Таким образом, пользователь может ввести строку слов, разделенных запятыми, которые являются тегами и могут быть добавлены в объект в виде массива.

И наконец, нужно переработать создание массива `toDos` из нового массива `toDoObjects`. Для этого я скопировал и вставил код, который мы использовали в верхней части функции `top` (нарушив таким образом принцип DRY — подумайте, как можно избежать этого), в результате чего мой код стал выглядеть вот так:

```
} else if ($element.parent().is(":nth-child(4)")) {
    var $input = $("<input>").addClass("description"),
        $inputLabel = $("<p>").text("Description: "),
        $tagInput = $("<input>").addClass("tags"),
        $tagLabel = $("<p>").text("Tags: "),
        $button = $("<span>").text("+");
    $button.on("click", function () {
        var description = $input.val(),
        // разделение в соответствии с запятыми
        tags = $tagInput.val().split(",");
        toDoObjects.push({"description":description, "tags":tags});
        // обновление toDos
        toDos = toDoObjects.map(function (toDo) {
              return toDo.description;
        });
        $input.val("");
        $tagInput.val("");
    });
}
```

Подведем итоги

В этой главе мы изучили основы трех очень важных тем: объекты JavaScript, JSON и AJAX. Я представил вам объекты JavaScript как способ хранения связанных данных в программе в виде единого целого. Объекты создаются с использованием фигурных скобок, а их свойства добавляются и связываются с ними с помощью оператора «точка» (`.`).

Вы можете рассматривать JSON как строковое представление объектов JavaScript, которые могут быть обработаны на любом языке программирования. JSON используется для передачи данных между компьютерными программами — в нашем случае между браузером (клиентом) и сервером. Кроме того, многие веб-сервисы (включая Flickr, Twitter и Facebook) предлагают API — интерфейсы программирования

приложений, которые позволяют отправлять информацию в эти сервисы (или запрашивать из них) с помощью формата JSON.

Как правило, если нужно отправить информацию в программу или получать сведения из нее на серверной стороне, мы используем технологию, которая называется AJAX. Это позволяет динамически обновлять информацию на веб-странице без необходимости перезагрузки. jQuery предлагает несколько функций AJAX, которые мы изучим в следующих главах, а в этой мы уже познакомились с функцией `getJSON`.

Больше теории и практики

Слайд-шоу Flickr

В этой задаче вам нужно создать простое приложение, которое позволяет пользователю ввести параметры поиска, а затем отображает серию изображений из Flickr, содержащих этот тег. Изображения должны появляться и исчезать по очереди.

Для этого нам понадобится функция `setTimeout`, с которой мы познакомились в главе 4. Эта функция позволяет нам запланировать событие, которое должно произойти спустя определенное время. Допустим, например, что мы хотим вывести сообщение «Hello, World!» через 5 секунд. Можем сделать это следующим образом:

```
// вывод "hello, world!" через 5 секунд
setTimeout(function () {
    console.log("hello, world!");
}, 5000);
```

Обратите внимание на то, что аргументы этой функции указаны не так, как мы привыкли (обычно обратный вызов идет последним), а наоборот. Это особенность, которую нужно просто запомнить. Обратите внимание также на то, что второй аргумент представляет собой количество *миллисекунд*, которое необходимо выждать перед вызовом функции.

А сейчас представим себе, что нужно просто отобразить и скрыть массив сообщений:

```
var messages = ["первое сообщение", "второе сообщение", "третье", "четвертое"];
```

Можем начать с простой основы HTML:

```
<body>
  <div class="message"></div>
</body>
```

Затем нужно настроить функцию `main` в файле `app.js` следующим образом:

```
var main = function () {
    var messages = ["первое сообщение", "второе сообщение", "третье", "четвертое"];
    var displayMessage = function (messageIndex) {
        // создаем и скрываем элемент DOM
```

```
        var $message = $("<p>").text(messages[messageIndex]).hide();
        // очищаем текущее содержимое
        // лучше всего будет выделить текущий параграф
        // и постепенно скрыть его
        $(".message").empty();
        // добавляем сообщение с messageIndex вDOM
        $(".message ").append($message);
        // постепенное отображение сообщения
        $message.fadeIn();
        setTimeout(function () {
            // через 3 секунды вызываем displayMessage снова со следующим индексом
            messageIndex = messageIndex + 1;
            displayMessage(messageIndex);
        }, 3000);
    };
    displayMessage(0);
}
$(document).ready(main);
```

Таким образом, сообщения будут отображаться по порядку каждые 3 секунды. Здесь есть, однако, небольшая проблема: достигнув конца списка сообщений, программа начнет выводить слово `undefined`, так как в массиве больше ничего нет. Мы можем исправить это, добавив оператор `if` для проверки того, достигли ли мы конца массива, и, если это так, сбрасывая индекс на 0.

На самом деле, скорее всего, не нужно добавлять этот эффект на реальные веб-страницы. Это современный эквивалент тега <blink>, который был удален из HTML, так как ужасно всех раздражал.

А сейчас попробуем все это обобщить. Итак, сначала нужно сделать запрос AJAX для получения данных об изображениях с Flickr, а затем вместо добавления в DOM элементов абзаца вставить изображение (`img` вместо `p`). Для элемента `img` используем атрибут `src`, чтобы указать изображение с Flickr.

После того как все это начало работать, создадим интерфейс, позволяющий пользователю ввести тег для поиска, а затем генерирующий слайд-шоу из изображений, соответствующих указанному тегу. Это прекрасное упражнение, которое займет у вас определенное время, особенно если вы решите добавить простейшие стили и действительно хорошо скоординировать отображение картинок. Я рекомендую вам попробовать!

Упражняемся в работе с объектами

Одна из задач, которые я очень люблю давать начинающим программистам, связана с определением покерных раздач. Если вы не знакомы с карточными играми, основанными на покере, ничего страшного. Все они базируются на комбинациях,

которые могут появляться в раздаче пяти игральных карт. Комбинации могут быть такие:

- пара — две карты одного номинала;
- две пары — две карты одного номинала и еще две карты другого;
- тройка — три карты одного номинала;
- стрит — пять карт, идущих по порядку, причем туз может считаться как тузом (следующим после короля), так и единицей;
- флеш — пять карт одной и той же масти;
- фулл-хауз — две карты одного номинала и три карты другого номинала;
- каре — четыре карты одного номинала;
- стрит-флеш — пять карт одной масти, номиналы которых идут по порядку;
- роял-флеш — стрит-флеш, в которой последовательность начинается с десятки и заканчивается тузом.

Любая другая раздача, где нет таких комбинаций, называется «нет игры». Обратите внимание на то, что комбинации не эксклюзивны — например, раздача, содержащая тройку, автоматически содержит и пару, а фулл-хауз содержит две пары. В этой серии задач мы напишем функции JavaScript, которые проверяют, есть ли в массиве из пяти карт одно из этих свойств.

Решать эти задачи можно разными способами, часть из которых более эффективны или требуют меньше кода, чем другие. Но наша цель сейчас — поупражняться в работе с массивами, объектами и условными операторами. Поэтому выберем подход, который позволяет создать серию вспомогательных функций, которые мы можем использовать для определения того, содержит ли набор карт определенный тип комбинации.

Для начала посмотрим, как может выглядеть раздача:

```
var hand = [
   { "rank":"двойка", "suit":"пик" },
   { "rank":"четверка", "suit":"червей" },
   { "rank":"двойка", "suit":"треф" },
   { "rank":"король", "suit":"пик" },
   { "rank":"восьмерка", "suit":"бубен" }
];
```

В нашем примере раздача содержит пару двоек. Было бы неплохо создать примеры для всех остальных типов комбинаций, прежде чем двигаться дальше.

Как же определить, соответствует ли раздача критериям комбинации? Нужно свести эту задачу к той, которую мы решали в конце главы 4! Если вы решали те задачи, то, наверное, помните, что нужно было написать функцию под названием `containsNTimes`, принимающую три аргумента: массив, элемент для поиска и минимальное количество раз повторения этого элемента в массиве. А сейчас представьте, что мы отправляем этой функции массив карточных номиналов:

```
// тут у нас две двойки
containsNTimes(["двойка","четверка","двойка","король","восьмерка"], "двойка", 2);
//=> true
```

Это говорит нам, что в массиве с номиналами есть две двойки! Мы можем использовать тот же принцип для определения наличия тройки или каре.

```
// тройки из двоек здесь нет
containsNTimes(["двойка","четверка","двойка","король","восьмерка"], "двойка", 3);
//=> false
```

Таким образом, мы свели задачу к представлению массива карточных объектов в виде массива карточных номиналов. Для этого очень легко использовать функцию `map`, с которой мы познакомились в этой главе:

```
// "hand" – массив раздачи, определенный ранее
hand.map(function (card) {
    return card.rank;
});
//=> ["двойка","четверка","двойка","король","восьмерка"]
```

Результат работы вызова функции `map` мы можем обработать как переменную, а затем отправить его в другую функцию `containsNTimes`, чтобы определить, есть ли там пара двоек:

```
var handRanks,
    result;
handRanks = hand.map(function (card) {
    return card.rank;
});
//в результате получаем 'true'
result = containsNTimes(handRanks, "two", 2);
```

И наконец, мы создаем массив для всех возможных номиналов, а затем используем цикл `forEach` для определения того, содержит ли раздача пару любых из них:

```
// это все возможные номиналы
var ranks = ["двойка","тройка","четверка","пятерка","шестерка","семерка","восьмерка",
             "девятка","десятка","валет","дама","король","туз"];
var containsPair = function (hand) {
    // предположим, что пар нет,
    // пока не доказано обратного
    var result = false,
        handRanks;
    // создаем массив карточных номиналов
    handRanks = hand.map(function (card) {
        return card.rank;
    });
    // поиск пары любого номинала
    ranks.forEach(function (rank) {
        if (containsNTimes(handRanks, rank, 2)) {
            // мы нашли пару
            result = true;
```

```
        }
    });
    // в результате получаем true, если пара найдена,
    // и false, если нет
    return result;
};
```

После рассмотрения этого примера попробуйте написать функцию для всех остальных комбинаций (например, `containsThreeOfAKind` для проверки наличия тройки). Для двух пар и фулл-хауза понадобится отслеживать значения номиналов элементов, которые вы обнаружили. А для флеша нужно будет извлекать из массива объектов масти, а не номиналы.

Случай стрита несколько сложнее, но я дам вам подсказку: стрит не содержит пар (что вы можете определить в результате работы функции `containsPair`, использовав оператор `!`), а разница между низшей и высшей картами составляет 4. Так что лучше всего будет написать набор вспомогательных функций для нахождения высшего и низшего номинала карт в виде чисел (валет в данном случае можно считать за 11, даму — за 12, короля — за 13, а туз — за 14). После того как у вас будут эти функции, вы можете определить, является ли раздача стритом, подтвердив, что отсутствуют пары, а разница между низшей и высшей картами составляет 4.

Затем можно создать и остальные функции, используя те, что у вас уже есть. Например, стрит-флеш автоматически содержит стрит, а роял-флеш является стрит-флешем, где высшая карта — туз (или низшая — 10).

Другие API

Теперь, когда вы убедились, как легко получать данные из Flickr и работать с ними, может быть интересно получить данные из других API. Оказывается, очень многие API позволяют получать различные данные точно так же, как Flickr.

Портал Programmable Web содержит целый список API. Для сайта, использующего функцию jQuery `getJSON`, необходима поддержка технологии под названием JSONP. Вы можете получить список API, содержащих JSONP, а также почитать об этой технологии в «Википедии».

Вам придется почитать документацию для определенных API, чтобы понять, как форматировать URL запроса, но упражнения с различными API — это отличный способ попрактиковаться в работе с AJAX.

6 Сервер

К этому моменту мы уже изучили основные технологии, связанные с клиентской частью веб-приложения. А также немного познакомились со взаимодействием браузера и сервера с помощью JSON и AJAX. А сейчас начнем знакомиться с программированием на серверной стороне!

Понимание серверной стороны приложения потребует от вас более глубого изучения модели «клиент — сервер», протокола HTTP и Node.js. Замечу, что Node.js — относительно новая (и превосходная!) технология, которая позволяет создавать событийно-управляемые серверы с помощью JavaScript.

Настройка рабочего окружения

Настройка рабочего окружения с поддержкой приложений, управляемых базами данных, может быть непростой задачей, и описание этой процедуры для Windows, Mac OS и Linux выходит за рамки книги. Для упрощения процесса я создал набор сценариев, которые помогут вам приступить к работе с использованием *VirtualBox* и *Vagrant* достаточно быстро.

Vagrant — это инструмент для настройки рабочего окружения на виртуальной машине. Вы можете думать о виртуальных машинах как об отдельных компьютерах, работающих полностью в пределах вашего компьютера. Мы поговорим о них подробнее в разделе «Ментальные модели». А сейчас просто запомните, что мы будем использовать Vagrant и VirtualBox для настройки виртуального рабочего окружения на серверной стороне. Это окружение будет включать в себя большинство инструментов, которые понадобятся нам в оставшейся части книги.

В общем-то, часть этого процесса связана в большей степени с удобством: хотя установка рабочего окружения Node.js на вашей локальной машине совсем не сложна (и в конце главы я буду рекомендовать вам сделать это самостоятельно), добавление набора инструментов для сервера, включая MongoDB и Redis, довольно нетривиально и требует много времени и терпения.

Другая причина, почему я так поступаю, — стабильность. Поскольку я сам написал сценарии, я получу несколько больше контроля над установленными версиями Node.js, Redis и MongoDB. Более того, использование Vagrant позволит иметь одно и то же рабочее окружение независимо от того, работаете ли вы в Windows,

Mac OS или Linux. Это минимизирует проблемы, возникающие из-за различий в версиях операционных систем.

Я также думаю, что рабочее окружение внутри VirtualBox отделяет серверную часть приложения очень ясно. Это создает отличную педагогическую абстракцию для всех изучающих основы разработки веб-приложений.

Установка Virtual Box и Vagrant

Прежде всего вы должны установить последнюю версию VirtualBox. На момент написания книги это версия 4.2.18. Вы можете зайти на http://www.virtualbox.org (рис. 6.1), щелкнуть на ссылке Downloads в левой части страницы и скачать соответствующую версию для вашей операционной системы. Загрузка, установка и настройка будут зависеть от операционной системы, поэтому внимательно следуйте инструкциям.

Рис. 6.1. Домашняя страница VirtualBox

Затем нужно установить последнюю версию Vagrant (на момент написания книги это версия 1.4). Чтобы получить последнюю версию, зайдите на http://www.vagrantup.com (рис. 6.2), щелкните на ссылке Downloads в верхнем правом углу и скачайте последнюю версию для вашей платформы. После загрузки дважды щелкните на пакете и установите его так же, как и VirtualBox.

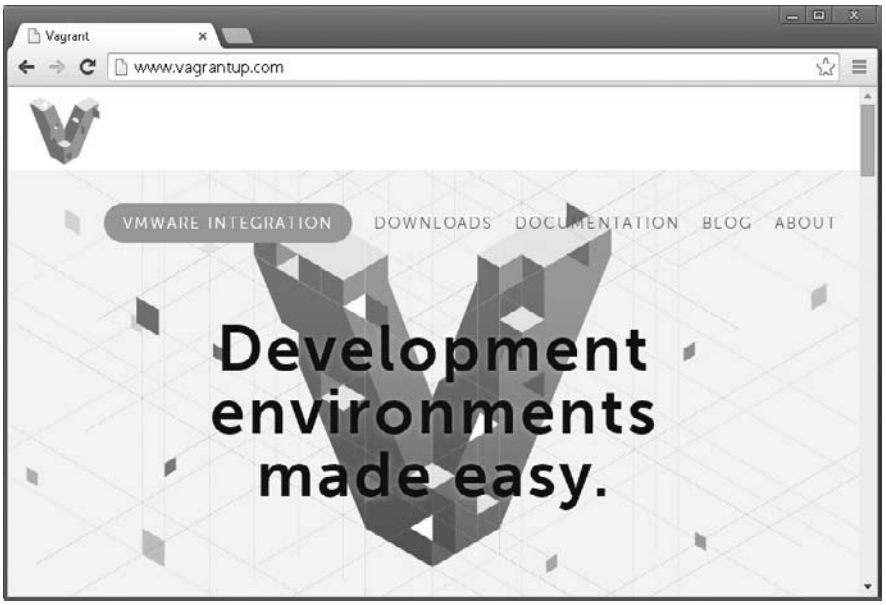


Рис. 6.2. Домашняя страница Vagrant

Если вы работаете в Windows, понадобится перезагрузка компьютера, чтобы убедиться в том, что путь ко всем файлам настроен корректно. Вы можете проверить это, открыв командную строку (нажмите кнопку Пуск и введите `cmd` в строку поиска) и введя команду `vagrant -version`:

```
C:\Users\semmy>vagrant --version
Vagrant version 1.4.0
```

Если все получилось, вы готовы к работе. Если нет, перезагрузите компьютер и попробуйте ввести команду `vagrant` снова.

Создание виртуальной машины

Если все получилось, вы готовы к скачиванию хранилища Git под названием `node-dev-bootsrap` с моей страницы GitHub. Это значит, что вам нужно скачать все содержимое хранилища Git, которое я создал, на свой компьютер. Если вы работаете в Windows, это потребует от вас открытия командной строки `git-bash`. В любом случае пройдите в свою папку Projects и скопируйте туда хранилище `node-dev-bootstrap` с помощью следующей команды:

```
hostname $ git clone https://github.com/semmypurewal/node-dev-bootstrap Chapter6
```

Таким образом будет создана папка под названием Chapter6, а хранилище `node-dev-bootstrap` — скопировано в нее. Затем зайдите в папку и введите команду `vagrant up`:

```
hostname $ cd Chapter6
hostname $ vagrant up
```

Если вы работаете в Windows, то можете увидеть предупреждение брандмауэра, спрашивающего, может ли программа vboxheadless получить доступ к сети. Вы смело можете ответить, что да.

Если все прошло хорошо, Vagrant создаст для вас виртуальную машину. Это займет несколько минут. Будьте терпеливы.

Подключение к виртуальной машине с помощью SSH

После того как машина создана, ее можно запустить. Как это проверить? У вас должна быть сетевая технология, называемая SSH (от Secure Shell), с помощью которой можно подключиться к серверу через командную строку.

Если вы используете Mac OS или Linux, это очень просто, так как в вашей платформе есть встроенный клиент SSH. Если же работаете в Windows, придется дополнительно установить клиент SSH.

В любом случае вам следует ввести:

```
hostname $ vagrant ssh
```

Если вы работаете в Mac OS, это подключит вас к виртуальной машине. Если же в Windows, это может сработать или не сработать в зависимости от версии. Если способ не работает, Vagrant даст вам данные для доступа (скорее всего, ваш хост будет 127.0.0.1 и порт 2222). Нужно будет скачать клиент SSH, чтобы подключиться, — я рекомендую PuTTY, который можно установить со страницы загрузки PuTTY. Домашняя страница PuTTY показана на рис. 6.3.

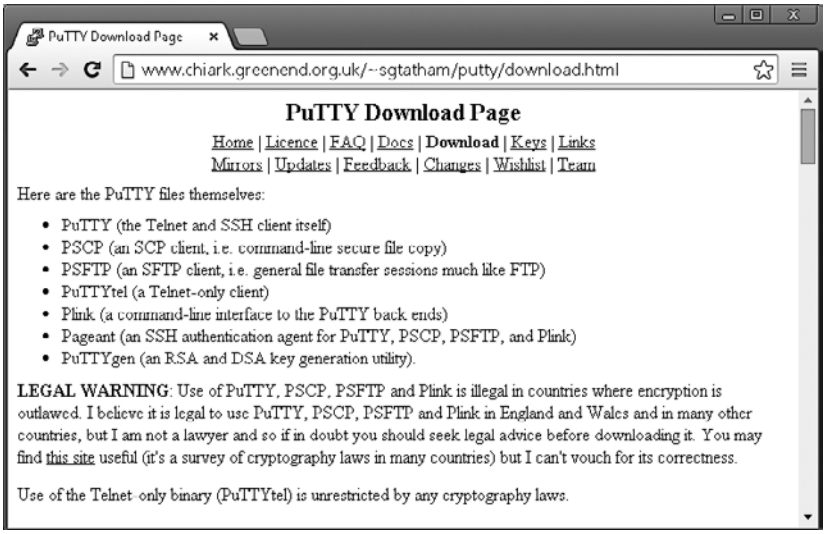


Рис. 6.3. Домашняя страница (страница загрузок) PuTTY

Запустив PuTTY, введите имя хоста и порт, которые вы получили от Vagrant, в соответствующие поля ввода Host Name и Port (рис. 6.4). Затем щелкните на Open (Открыть). PuTTY подключит вас к виртуальному серверу и запросит имя пользователя и пароль. То и другое установлено в Vagrant.

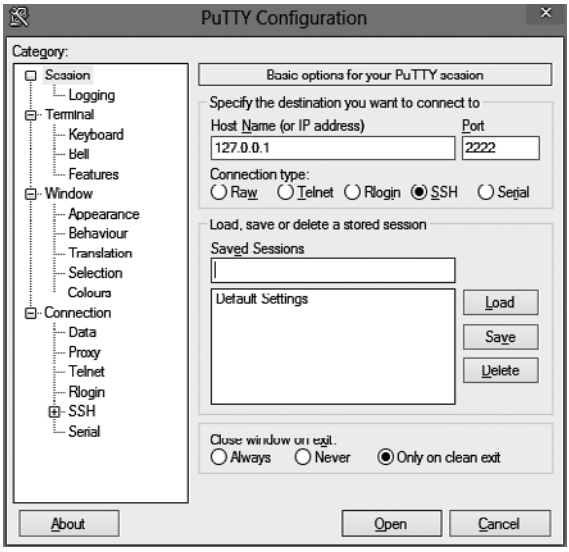


Рис. 6.4. Установка PuTTY для подключения к виртуальной машине

Привет, Node.js!

После того как вы подключились, можете работать в командной строке виртуального компьютера точно так же, как в командной строке на локальном. Например, вы можете вывести на экран содержимое какой-либо папки на виртуальной машине тем же способом, каким пользовались на локальной, — с помощью команды `ls`:

```
vagrant $ ls
app postinstall.sh
```

Пока не обращайте внимания на файл `postinstall.sh`. С помощью навыков работы с командной строкой, которые вы получили в предыдущих главах, перейдите в каталог app на вашей виртуальной машине:

```
vagrant $ cd app
vagrant $ ls
server.js
```

Здесь вы должны увидеть файл `server.js`. Проверим, что все работает правильно, начав с команды `node`:

```
vagrant $ node server.js
Server listening on port 3000
```

Вы увидите сообщение Server listening on port 3000 («Сервер слушает на порте 3000») и больше не сможете работать с командной строкой из-за того, что сервер запущен. Вы можете убедиться в этом, открыв Chrome и набрав `localhost:3000` в адресной строке. Если все работает правильно, вы должны увидеть сообщение «Hello, World!». Оно должно выглядеть так, как показано на рис. 6.5.

Рис. 6.5. Сервер Node по умолчанию, отображаемый в Chrome

Самое главное: вы можете остановить программу из командной строки, нажав Ctrl+C. Окончив работу, можете покинуть виртуальный сервер, введя `exit`:

```
vagrant $ exit
logout
Connection to 127.0.0.1 closed.
```

Если вы авторизованы на своей виртуальной машине с помощью PuTTY в Windows, это действие закроет программу. Затем можете вернуться к командной строке, которую вы использовали, чтобы запустить виртуальную машину. Если же вы работаете в MacOS, то немедленно вернетесь к командной строке. В любом случае вы всегда можете остановить сервер, набрав `vagrant halt`:

```
hostname $ vagrant halt
[default] Attempting graceful shutdown of VM..[1]
```

Остановка виртуальной машины может понадобиться, если вы не работаете с ней в какой-то момент, но она запущена в фоновом режиме. Виртуальная машина потребляет очень много ресурсов компьютера. После остановки можете запустить ее снова командой `vagrant up`.

Чтобы полностью удалить виртуальную машину, созданную в Vagrant, со своего компьютера, нужно использовать команду vagrant destroy. В результате виртуальная машина будет создана снова в следующий раз, когда вы введете vagrant up из этой папки. Поэтому пока я рекомендую вам работать с командой vagrant halt.

[1] Сообщение: «Мягкое завершение работы виртуальной машины...». — *Примеч. пер.*

Ментальные модели

В данном разделе обсудим различные способы понимания природы клиентов, серверов, хостов и гостей.

Клиенты и серверы

В области компьютерных сетей о программах обычно говорят либо как о *серверных*, либо как о *клиентских*. Традиционно *серверная* программа рассматривает какой-либо ресурс как сеть, в которую могут получить доступ несколько *клиентских* программ. Например, кому-либо может понадобиться переместить файлы с удаленного компьютера на локальный. FTP-сервер — это программа, которая предоставляет протокол передачи файлов (File Transfer Protocol), позволяющий пользователям сделать это. А FTP-клиент — это программа, которая устанавливает связь с FTP-сервером и передает файлы. Самая общая модель «клиент — сервер» представлена на рис. 6.6.

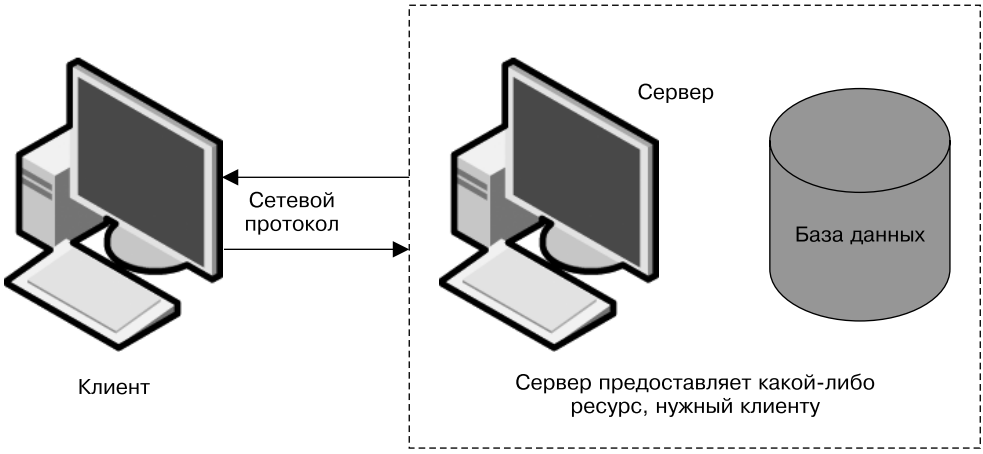


Рис. 6.6. Модель «клиент — сервер»

Многое из происходящего сейчас в мире компьютерных сетей является второстепенным для разработки веб-приложений, но нужно разобраться в некоторых важных вещах.

Во-первых, тот факт, что (в большинстве случаев) клиент — это браузер, а сервер — удаленная машина, которая опосредованным образом предоставляет ресурсы через протокол передачи гипертекста HTTP (Hypertext Transfer Protocol). Хотя изначально он был разработан для обмена HTML-документами между компьютерами, этот протокол может использоваться также для передачи различных типов ресурсов с удаленного компьютера (в частности, документы, базы данных или другие типы представлений). В дальнейшем нам придется немало говорить об HTTP, а пока просто запомните: это протокол, который мы используем, чтобы с помощью браузера связаться с удаленным компьютером.

Наши серверы HTTP будут использоваться для доставки клиентской части приложения, которую сможет интерпретировать браузер. В частности, все HTML, CSS и JavaScript, изученные нами к этому моменту, будут доставляться браузеру через сервер. Клиентская часть программы, запущенная в браузере, будет отвечать за получение информации с сервера и отправку информации на него, как правило, с помощью AJAX.

Хосты и гости

Обычно HTTP-сервер запускается на удаленной машине (позднее в этой книге мы проделаем это самостоятельно). Из-за этого у разработчиков возникают некоторые проблемы — если код запускается на удаленном сервере, приходится или редактировать код на удаленном сервере, а затем перезагружать его, или редактировать код локально и отправлять его на сервер всякий раз, когда нужно попробовать изменения. Это может быть нерационально.

Мы можем обойти эту проблему, запустив сервер на своей машине локально. Фактически мы делаем таким образом шаг вперед. Вместо того чтобы локально запустить программу сервера и все программное обеспечение, мы создаем внутри своего компьютера виртуальную машину, на которой запускается сервер. Для человека это выглядит так же, как удаленная машина (в том смысле, что мы подключаемся к ней через SSH, тем же способом, что и к удаленной машине). Но поскольку машина работает внутри нашего компьютера, можем просто предоставить локальной машине права доступа к папке с рабочими файлами и таким образом редактировать их.

Локальный компьютер, на котором запущена виртуальная машина, является *хостом* (хозяином). Виртуальная машина является *гостем*. В оставшейся части главы они оба будут запущены, и я буду четко разделять их.

Практические вопросы

Эта абстракция может привести к недопониманию, которое будет прояснено позже. Прежде всего поговорим о приложениях, которые будут запущены во время процесса разработки, и об окнах, с которыми они будут связаны.

- Браузер. Вы будете использовать Chrome для тестирования своего приложения.
- Текстовый редактор. Если вы до сих пор делали все правильно, это, скорее всего, будет Sublime. Он будет редактировать файлы, находящиеся в папке с разделенными правами доступа как на хостовой, так и на гостевой машине.
- Git. Командная строка Git должна быть открыта всегда. Используйте Git с машины-хоста. Если вы работаете в Mac OS или Linux, это будет командное окно, если же в Windows — командная строка `git-bash`.
- Vagrant. Скорее всего, вы будете взаимодействовать с Vagrant через то же окно, которое используете для работы с Git. Это значит, что вы можете ввести `vagrant up` или `vagrant halt` в том же самом месте, где набираете `git commit`.

- SSH. Он будет запущен на хостовой машине, но вы будете использовать его для взаимодействия с гостевой. Если вы работаете в Windows, понадобится PuTTY или дополнительное окно `git-bash` в зависимости от того, каким образом `vagrant ssh` осуществляет связь. Если же вы работаете в Mac OS, лучше всего будет открыть дополнительное командное окно.

Обычно у меня открыто не менее двух командных окон в дополнение к браузеру и текстовому редактору. Частично это иллюстрирует рис. 6.7, где, правда, не показано окно текстового редактора.

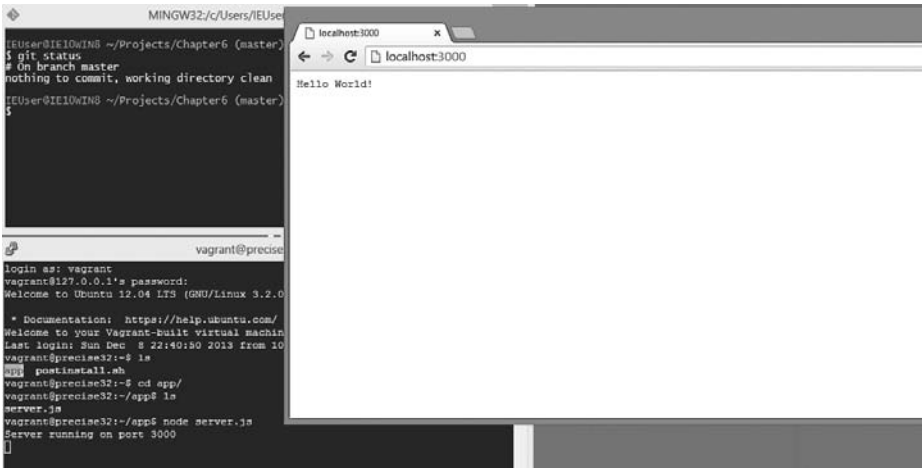


Рис. 6.7. Запущенные в Windows окна git-bash, PuTTY и Chrome

Привет, HTTP!

А сейчас, когда мы немного разобрались в отношениях между виртуальной машиной-гостем и машиной-хостом, попробуем понять сам код. Как я уже упоминал, главное преимущество запуска кода на виртуальной машине, работающей на нашем локальном компьютере (в отличие от удаленного запуска), — это доступ к редактированию кода прямо с локального компьютера через любой текстовый редактор по нашему выбору.

Откройте в Sublime папку app внутри каталога Chapter6. Вы найдете там файл `server.js`, который выглядит примерно так:

```
var http = require("http"),
server;
server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Hello, World!");
});
server.listen(3000);
console.log("Server listening on port 3000");
```

Этот код не должен показаться вам совершенно незнакомым — вы обязаны немедленно заметить некоторые объявления переменных, оператор `console.log`, анонимную функцию, объект JSON, а затем шаблон обратного вызова, который мы использовали при построении элементов UI в предыдущих главах.

Но что же делает этот код? Оказывается, довольно много — он создает сервер HTTP, который отвечает на запросы браузера! Как я уже говорил, HTTP означает HyperText Transfer Protocol (протокол передачи гипертекста), и это основная технология, которая стоит за www (World Wide Web — Всемирная паутина). Вы, возможно, слышали о программах вроде Apache или Nginx — это настраиваемые серверные программы HTTP промышленного класса, которые используются для хостинга больших веб-сайтов. Наш HTTP-сервер гораздо проще: он просто принимает запрос браузера и выдает текстовый ответ «Hello, World!».

Этот код можно понять по аналогии с обработчиком щелчков в jQuery — разница состоит лишь в том, что обратная связь возникает не в ответ на щелчки кнопкой мыши, выполняемые пользователем, а в результате выхода на связь клиента (в данном случае браузера). Параметр `req` — это объект, представляющий запрос HTTP, который приходит на наш сервер с клиента, а `res` — это параметр, который представляет ответ HTTP, отправляемый обратно клиенту. Функция `req.writeHead` создает заголовок HTTP, который устанавливает атрибуты ответа, а функция `res.end` завершает ответ, добавляя `Hello, World!`.

После создания сервера мы начинаем *слушание* на порте 3000 (вот почему после `localhost` в браузере нужно было набрать `3000`). Оператор `console.log` отображает сообщение в логе, когда мы запускаем программу.

Модули и Express

Уф! Не слишком ли много для маленькой программы? А ведь это еще не все — сервер HTTP требует хороших навыков и знаний для корректного написания кода. К счастью, нет необходимости заботиться о деталях, так как мы импортировали код через модуль http Node.js. Модуль — это просто какое-то количество кода, которое мы можем использовать без необходимости полностью понимать его работу — нужно знать лишь API, который предоставляет код. Первая строка нашего кода — оператор `require`, который импортирует модуль http и хранит его в переменной `http`.

Этот серверный модуль http интересен, если нам нужен самый примитивный, базовый HTTP-сервер, который просто принимает запросы клиента и отвечает на них. Поскольку мы хотим начать с отправки файлов HTML или CSS с сервера, все становится несколько сложнее. Мы можем создать более сложный сервер поверх базового, который предоставляет нам Node.js, но, к счастью, решение уже создано до нас! Модуль Express создает поверх ядра модуля http надстройку, которая обрабатывает большинство сложных вещей, о которых нам не нужно заботиться самостоятельно, в частности, обслуживает статичный HTML, CSS и файлы JavaScript на клиентской стороне.

В самом деле, одно из преимуществ программирования на Node — наличие множества модулей наподобие Express, среди которых можно обнаружить очень полезные. Модуль http, который мы рассматривали ранее, является частью основного дистрибутива Node, поэтому не нужно делать ничего особенного, чтобы использовать его. Express не входит в основной дистрибутив, поэтому придется приложить некоторые усилия, прежде чем мы сможем получить доступ к нему так же легко, как к модулю http.

К счастью, каждый дистрибутив Node поставляется со специальной программой, называемой *Node Package Manager* (NPM) — менеджер пакетов Node. Этот инструмент позволяет легко устанавливать модули, а затем немедленно использовать их в коде.

Установка Express с помощью NPM

Использование NPM очень просто. Начнем с создания новой папки внутри каталога app и назовем ее Express. Вы можете сделать это из Sublime или из командной строки. В любом случае, как только у вас появится эта папка, свяжитесь с машиной Vagrant через SSH и перейдите в папку Express. Затем можете набрать в `npm install express`:

```
vagrant $ ls
app postinstall.sh
vagrant $ cd app
vagrant $ ls
Express server.js
vagrant $ cd Express
vagrant $ npm install express
```

Иногда NPM и Virtual Box не работают корректно вместе на компьютере с Windows. Если вы получаете сообщения об ошибках во время установки Express, попробуйте выполнить команду npm install express –no-bin-links — это должно помочь.

NPM сообщит об окончании установки модуля Express и выведет длинный список его взаимоотношений. Куда же он устанавливается? Оказывается, модуль установлен в текущей папке, в подпапке под названием node_modules. В этом легко убедиться:

```
vagrant $ pwd
/home/vagrant/app/Express
vagrant $ ls
server.js node_modules
vagrant $ ls node_modules
express
```

Первый сервер Express

Сейчас, когда модуль Express установлен, мы можем создать простой сервер Express. Создадим внутри каталога Express файл, назовем его `server.js` и добавим следующее содержание:

```
var express = require("express"),
http = require("http"),
app;
// Создаем http-сервер на основе Express
// и заставляем его слушать на порте 3000
app = express();
http.createServer(app).listen(3000);
// настраиваем маршруты
app.get("/hello", function (req, res) {
  res.send("Hello, World!");
});
app.get("/goodbye", function (req, res) {
  res.send("Goodbye, World!");
});
```

Этот пример кажется очень похожим на предыдущий, но на самом деле здесь есть несколько существенных различий. Во-первых, нет необходимости прописывать заголовки HTTP в обратных вызовах, так как Express уже проделал это самостоятельно. Во-вторых, мы просто используем `res.send` вместо `res.write` или `res.end`. И последнее, но, наверное, самое важное: мы настроили два маршрута, `hello` и `goodbye`. Мы еще увидим, что произойдет, когда будет открыт браузер, но сначала запустим сервер.

Напомню, что мы можем сделать это, введя `node server.js` из папки Express на гостевой машине. Затем, перейдя к браузеру, нужно набрать `localhost:3000`, как мы уже делали ранее. В этот раз, однако, получим сообщение об ошибке с текстом `'Cannot GET /'`. Но если вместо этого мы введем `localhost:3000/hello` или `localhost:3000/goodbye`, то получим сообщения, которые указали в обратных вызовах. Попробуйте прямо сейчас.

Как видите, добавление `hello` или `goodbye` в конце основного URL нашего приложения определяет, какая функция сработает. Кроме того, получается, что Express не устанавливает маршрута по умолчанию. Если нужно, чтобы адрес `localhost:3000` работал как прежде, нужно просто настроить главный маршрут, добавив к файлу `server.js` еще три строки:

```
app.get("/", function (req, res) {
  res.send("Это основной маршрут!");
});
```

Если остановить сервер (нажав Ctrl+C) и запустить его снова, мы сможем получить доступ к `localhost:3000`, как уже было до этого! Express обрабатывает маршруты самостоятельно — мы просто говорим ему, что должно быть сделано в случае запроса определенного маршрута.

Отправка клиентского приложения

Вы уже видели, как можно отправлять информацию в браузер от сервера. Но что, если мы хотим послать нечто вроде базовой HTML-страницы? Задание

значительно усложняется, и очень быстро. Допустим, например, что надо отправить нечто такое:

```
app.get("/index.html", function (req, res) {
res.send("<html><head></head><body><h1>Hello, World!</h1></body></html>");
});
```

Несмотря на то что это работает, отправка HTML-кода, значительно большего, чем этот простенький пример, может стать невероятно продолжительной. К счастью, в Express эта проблема решена — мы можем использовать *статичный файловый сервер*. С его помощью можно создавать файлы HTML, CSS и JavaScript на клиентской стороне точно так же, как и раньше в этой книге! Кроме того, оказывается, что для этого нужна всего одна дополнительная строка кода.

Начнем с создания новой папки в каталоге app и назовем ее client. Создадим файл `server.js` такого содержания:

```
var express = require("express"),
http = require("http"),
app = express();
// настроим статическую файловую папку
// для маршрута по умолчанию
// (см. также замечание о маршрутах далее)
app.use(express.static(__dirname + "/client"));
// создадим HTTP-сервер на базе Express
http.createServer(app).listen(3000);
// настроим маршруты
app.get("/hello", function (req, res) {
res.send("Hello, World!");
});
app.get("/goodbye", function (req, res) {
res.send("Goodbye, World!");
});
```

Если вы попробуете запустить этот пример в Windows вместо своей виртуальной машины, то, скорее всего, столкнетесь с проблемой, вызванной именами файлов. Для простоты изложения я отказался от использования базового модуля path, который придал бы папке свойство кросплатформенной совместимости. В то же время я использовал его в коде в хранилище GitHub, поэтому, если у вас возникнут какие-то трудности, ознакомьтесь с хранящимися там примерами.

В предыдущем примере мы использовали `app.use` для создания статической файловой серверной папки. Это значит, что любой запрос, отправленный на сервер, будет немедленно обработан статической файловой папкой (client), прежде чем попадет на маршруты. Таким образом, если у нас есть файл `index.html` в папке client и мы заходим на `localhost:3000/index.html`, то в результате увидим содержимое

файла. Если же файла не существует, то будет произведена проверка того, есть ли другие совпадения на маршрутах.

Здесь легко можно запутаться — если у нас есть маршрут с тем же названием, что и файл в папке client, как же ответит Express? Сначала он ищет ответ в клиентской папке и, если найдет совпадение, даже не будет просматривать маршруты. Поэтому нельзя создавать файлы и маршруты с одним именем — это не приведет ни к чему хорошему практически никогда.

Посмотрим, что будет, если мы скопируем одно из клиентских приложений в каталог client. Возьмите одно из приложений, которые мы уже создали (а еще лучше — поупражняйтесь и создайте по памяти новое), и скопируйте его в папку client, находящуюся внутри Express.

Сейчас можно запустить сервер с гостевой машины с помощью команды `node server.js`. Поскольку основная страница HTML находится в файле `index.html`, когда мы открываем страницу `localhost:3000/index.html` в браузере, можем увидеть ее — включая CSS и JavaScript!

Общие принципы

В дальнейшем будем создавать все приложения, следуя этому общему шаблону. Браузерный код будет храниться в каталоге client, а сервер Express — в файле под названием `server.js`. Мы также будем импортировать и/или создавать модули для поддержки серверной стороны нашей программы.

Единственная вещь, с которой мы пока не сталкивались, — это настройка коммуникации между сервером и клиентом. Как упоминалось в предыдущей главе, будем использовать AJAX для связи и формат JSON — для представления сообщений. Следующий пример продемонстрирует это, а также настройку модулей для получения более интересных вещей — например, подключения к Twitter API.

Считаем твиты

В этом примере мы подключимся к Twitter API и извлечем некоторые данные на наш сервер. Начнем с запуска виртуальной машины, если она еще не работает. Если вы пока этого не сделали, введите `vagrant up` из папки app, находящейся внутри Chapter6. Затем подключимся по SSH к гостевой машине. Если вы работаете в Windows, это может потребовать использования PuTTY, а если в Mac OS или Linux, просто введите `vagrant ssh`.

Теперь создадим новую папку внутри каталога app и назовем ее Twitter. Помните, что, даже если мы создаем эту папку на гостевой машине, она отображается и на хостовой, так что мы легко можем открыть ее (через каталог app внутри Projects) в Sublime.

Получение данных для входа в Twitter

Чтобы получить доступ к потоковому API Twitter, нужно установить приложение, а затем авторизоваться, используя логин и пароль для Twitter. Конечно, если у вас нет учетной записи в Twitter, нужно ее создать. Но не беспокойтесь, я не собираюсь заставлять вас писать твиты!

После того как вы авторизуетесь на странице разработчиков Twitter, нажмите кнопку Create a new application (Создать новое приложение) в верхнем правом углу. Сделав это, вы увидите форму, похожую на рис. 6.8. Заполните все поля и отправьте запрос.

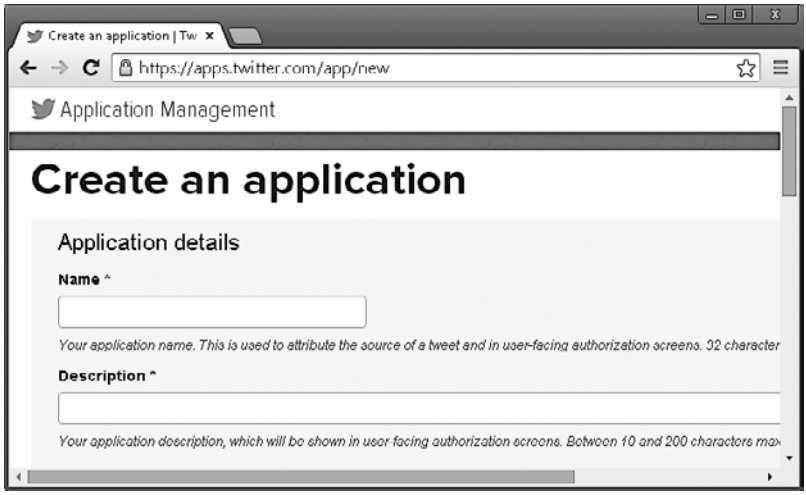


Рис. 6.8. Форма, которую вы будете использовать для создания первого приложения Twitter

В форме есть обязательное поле, где требуется указать веб-сайт вашего приложения. У вас, наверное, его пока нет, поэтому можете использовать временную «заглушку» (я, например, указываю www.example.com). Не беспокойтесь об URL обратного вызова для этого примера.

Если запрос сработал успешно, вы окажетесь на странице, которая содержит всю информацию о вашем приложении, включая Consumer key (ключ абонента) и Consumer secret (секретные данные абонента). Чтобы получить доступ к потоковому API, нужно создать маркер доступа для вашего приложения. Вы можете получить его, нажав кнопку Create my access token (Создать мой маркер доступа). Когда страница обновится, у вас будут все данные, необходимые для работы следующего примера.

А сейчас создадим простой файл под названием `credentials.json`, к которому наша программа должна иметь доступ. Файл будет содержать один объект JSON, в котором находятся данные для входа в Twitter (не забудьте оставить вокруг них кавычки, так как они должны сохраниться как строковые переменные):

```
{
  "consumer_key": "your consumer key here",
  "consumer_secret": "your consumer secret here",
  "access_token_key": "your access token key here",
  "access_token_secret": "your access token secret here"
}
```

Если вы создаете приложения, которые требуют данных для входа, следует изолировать их в хранилище Git на тот случай, если вы захотите поделиться своим кодом с кем-нибудь еще. Вот почему лучше всего хранить свои данные для входа в отдельном файле. Git позволяет создавать файлы специального типа .gitignore — локальные файлы, которые изолированы от остальной части хранилища.

Подключение к Twitter API

Начнем с установки модуля ntwitter через NPM:

```
vagrant $ mkdir Twitter
vagrant $ cd Twitter
vagrant $ npm install ntwitter
vagrant $ ls node_modules
express ntwitter
```

После этого создадим файл под названием `tweet_counter.js`, где будет находиться код, обеспечивающий получение доступа к Twitter API. Отмечу, что мы запрашиваем модуль ntwitter с использованием того же пути, который использовали при запросе модуля http в примере с сервером. Запрашивая файл `credentials.json`, мы должны сообщить Node, где его найти, так как, в отличие от express и ntwitter, не устанавливали его через NPM. Вот почему перед именем файла находится `./` — это указание Node искать файл в текущей папке:

```
var ntwitter = require("ntwitter"),
credentials = require("./credentials.json"),
twitter;
// настроим наш объект twitter
twitter = ntwitter(credentials);
// настроим поток twitter с тремя параметрами,
// разделенными запятыми
twitter.stream(
  // первый параметр – строка
  "statuses/filter",
  // второй параметр – объект, содержащий массив
  { "track": ["awesome", "cool", "rad", "gnarly", "groovy" }[1], //см.перевод в сноске
```

[1] «Потрясающе», «круто», «супер», «великолепно», «классно». В данном примере использование кириллических символов требует изменения кодировки для правильного их отображения в командной строке. — *Примеч. пер.*

```
  // третий параметр – обратный вызов, срабатывающий, когда поток создан
  function(stream) {
    stream.on("data", function(tweet) {
      console.log(tweet.text);
    });
  }
);
```

Если все настроено правильно, мы можем сейчас ввести команду `node tweet_counter` в виртуальной машине и увидим, что в командном окне отображаются твиты, содержащие слова из массива! Когда вам надоест смотреть, как компьютер выражает свое восхищение, просто нажмите Ctrl+C, чтобы остановить поток лести.

Как это получилось?

Посмотрим на код еще раз. Вы увидите несколько знакомых элементов. Во-первых, мы объявили три переменные и импортировали модуль ntwitter вместе с файлом `credentials.json`. Затем создали переменную под названием `twitter` и сохранили в ней результат вызова функции `ntwitter` с нашими данными входа в качестве аргумента. Таким образом инициализируется объект `twitter`, в результате чего запускается поток.

После этого мы определяем поток, вызывая функцию `stream` для объекта `twitter`. Эта функция принимает три аргумента. Первый — строка, определяющая тип потока (мы представляем твиты в виде списка слов). Второй — объект, в котором содержится информация о правилах фильтрации (в данном случае мы просто наблюдаем за возникновением отдельных слов, а вообще их можно отфильтровать, например по расположению). И наконец, обратный вызов, означающий, что создан поток, который мы отправляем.

Аргумент `stream` сам по себе является объектом, с помощью которого мы можем улавливать события (аналогично элементам DOM в jQuery). Событие, которое нужно поймать, — данные (data), и оно запускается, когда новый твит приходит из потока. Что же мы делаем, получив новый твит? Просто вводим его в консоль! Вот все, что заставляет Twitter отправлять поток твитов в ваше командное окно.

Хранение счетчиков

Вместо того чтобы просто набирать твиты в консоли, начнем сохранять количество появлений определенного слова. Это значит, что для каждого слова нужна отдельная переменная. Таким образом получится несколько переменных, хранящих нечто похожее, следовательно, имеет смысл использовать объект, представляющий собой счетчик. Атрибут, связанный с каждым из счетчиков, — слово, а значение атрибута — количество появлений каждого слова.

Мы можем изменить начало кода, чтобы объявить и создать такой объект:

```
var ntwitter = require("ntwitter"),
credentials = require("./credentials.json"),
twitter,
```

```
counts = {};
counts.awesome = 0;
counts.cool = 0;
counts.rad = 0;
counts.gnarly = 0;
counts.groovy = 0;
```

Таким образом, начальное значение всех счетчиков будет равно нулю, потому что пока мы не видели ни одного из этих слов.

Использование функции indexOf для поиска слов

Когда мы получаем данные из потока Twitter, то проверяем, содержит ли свойство `text` нашего объекта `tweet` слово, которое мы ищем. В предыдущей главе мы видели, что у массивов есть метод, называемый `indexOf`, который проверяет, находится ли в массиве некоторый объект. Оказывается, такая же функциональность существует и для строк! Функция `indexOf` для строковых объектов будет возвращать индекс для первого появления подстроки внутри строки или –1, если подстрока не появляется. Например, попробуйте набрать в консоли JavaScript в Chrome следующее:

```
var tweet = "Это твит! В нем много слов, но меньше 140 символов."
// Используем indexOf для проверки того, содержатся ли в строке отдельные слова
tweet.indexOf("твит");
//=> 10
tweet.indexOf("Это");
//=> 0
tweet.indexOf("слов");
//=> 32
tweet.indexOf("симв");
//=> 56
tweet.indexOf("привет");
//=> -1
// обратите внимание: indexOf чувствительна к регистру!
tweet.indexOf("Твит");
//=> -1
```

Вы видите, что `indexOf` будет искать твит для данного слова, и, если слово появляется в подстроке, результат будет больше –1. Это позволяет убрать оператор `console.log` и модифицировать код примерно так:

```
function(stream) {
  stream.on("data", function(tweet) {
    if (tweet.indexOf("awesome") > -1) {
      // приращение счетчика для слова awesome
      counts.awesome = counts.awesome + 1;
    }
  });
}
```

Использование функции setInterval для планирования задач

Но как мы узнаем, работает ли код, без оператора `console.log`? Одно из возможных решений — введение счетчиков через каждые несколько секунд. Это более рацио-

нально, чем вывод каждого твита, который мы видим. Чтобы сделать это, можно добавить вызов функции setInterval в нижней части потокового кода:

```
// вводить счетчик для
setInterval(function () {
  console.log("awesome: " + counts.awesome);
}, 3000);
```

Аналогично функции setTimeout, с которой мы уже сталкивались, функция setInterval устанавливает функцию, которая будет вызвана в дальнейшем. Разница состоит в том, что она повторяет вызов функции каждый раз, когда проходит указанное количество миллисекунд. Так что в данном случае программа будет выводить счетчик для слова awesome каждые 3 секунды.

Если нам требуется всего лишь подсчитать количество появлений слова awesome, то целиком файл twitter.js будет выглядеть следующим образом:

```
var ntwitter = require("ntwitter"),
credentials = require("./credentials.json"),
twitter,
counts = {};
// настроим объект twitter
twitter = ntwitter(credentials);
// обнуляем счетчики
counts.awesome = 0;
twitter.stream(
  "statuses/filter",
  { "track": ["awesome", "cool", "rad", "gnarly", "groovy"] },
  function(stream) {
    stream.on("data", function(tweet) {
      if (tweet.indexOf("awesome") > -1) {
        // приращение счетчика для слова awesome
        counts.awesome = counts.awesome + 1;
      }
    });
  }
);
// вводим счетчик каждые 3 секунды
setInterval(function () {
  console.log("awesome: " + counts.awesome);
}, 3000);
```

Запустите этот код на своей виртуальной машине, введя node tweet_counter.js, и посмотрите на него в действии.

Разделение счетчиков Twitter на модули

Наблюдать за появлением твитов в командном окне, конечно, забавно, но было бы лучше каким-то образом подключить счетчик твитов к серверу HTTP, чтобы мы могли отображать их в браузере. Есть два способа сделать это. Первый заключается в создании внутри кода счетчика твитов сервера, который и будет генерировать страницу. Это решение вполне подходит для начинающих.

Второй способ заключается в преобразовании кода счетчика твитов в модуль, а затем импорте этого модуля в программу, где уже есть сервер HTTP (таков, например, наш основной серверный пример). В данном случае второе решение лучше, так как оно позволяет использовать этот код счетчика твитов и в других проектах. Я бы сказал, что это решение соответствует одному из ключевых принципов философии Node.js: создание маленьких модулей, каждый из которых делает какую-то одну вещь, но делает ее хорошо.

Как мы можем преобразовать счетчик твитов в модуль? Оказывается, это очень просто. В каждом модуле есть специальная переменная под названием `module.exports`, где хранится все, что нужно представлять окружающему миру. В данном случае нам следует представить объект `counts`, который будет обновляться во время работы нашей программы. В нижней части кода `tweet_counter.js` мы можем просто добавить следующую строку для экспорта объекта:

```
module.exports = counts;
```

Здесь мы экспортируем объект, где хранятся несколько целочисленных переменных, но вы можете экспортировать любой тип объекта JavaScript, включая функции. В практических упражнениях в конце этой главы я дам вам возможность сделать этот модуль более универсальным с помощью экспортирования функции вместо объекта

Импорт модуля в Express

Напомню, что наш несложный сервер Express выглядит примерно так:

```
var express = require("express"),
http = require("http"),
app = express();
// настраиваем приложение для использования клиентской папки для статичных файлов
app.use(express.static(__dirname + "/client"));
// создаем HTTP-сервер на базе Express и начинаем слушать
http.createServer(app).listen(3000);
// настраиваем маршруты
app.get("/hello", function (req, res) {
res.send("Hello, World!");
});
```

Немного изменим этот код так, чтобы импортировать модуль для подсчета твитов и использовать полученные результаты, возвращая их в браузер через JSON:

```
var express = require("express"),
http = require("http"),
tweetCounts = require("./tweet_counter.js"),
app = express();
// настраиваем приложение для использования клиентской папки для статичных файлов
app.use(express.static(__dirname + "/client"));
```

```
// создаем HTTP-сервер на базе Express и начинаем слушать
http.createServer(app).listen(3000);
// настраиваем маршруты
app.get("/counts.json", function (req, res) {
// res.json возвращает весь объект целиком в виде файла JSON
res.json(tweetCounts);
});
```

Если мы авторизуемся на гостевой машине, то можем ввести папку app и запустить сервер из node app. Сделав это, мы увидим вывод счетчиков в командной строке каждые 3 секунды, но сейчас можем открыть браузер и подключиться к localhost:3000/counts.json, после чего увидим отображение JSON объекта!

Но как взаимодействовать с объектом JSON в клиентском коде? Конечно же, через AJAX!

Настройка клиента

Напишем основу клиентского приложения, которая, как обычно, будет включать в себя файлы index.html, styles/style.css и javascripts/app.js, к которым мы уже привыкли при программировании клиентской стороны. Создайте их непосредственно в папке client или в любом другом месте, а затем скопируйте в client с помощью команды cp. Запустите сервер из каталога app на гостевой машине:

```
vagrant $ node app.js
```

Если в файле javascript/app.js уже настроен вывод hello, world, то, перейдя на localhost:3000/index.html в браузере, мы увидим слова «Hello, World!», выведенные в консоли JavaScript, как и раньше.

Если все работает правильно, изменим файл клиентской стороны app.js для получения объекта counts.json с вызовом функции jQuery getJSON:

```
var main = function () {
  "use strict";
  $.getJSON("/counts.json", function (wordCounts) {
    // Сейчас "wordCounts" становится объектом, возвращаемым
    // маршрутом counts.json, который мы
    // настроили в Express
    console.log(wordCounts);
  });
}
$(document).ready(main);
```

Таким образом, вызов getJSON подключается к маршруту counts.json, который мы настроили в Express, в результате чего нам возвращаются счетчики. Если все эти связи работают правильно, в консоль теперь будет выводиться объект counts. Мы можем просмотреть результат вывода объекта сверху вниз с помощью кнопок прокрутки. А если перезагрузим страницу, то увидим обновленные значения!

Сейчас мы можем модифицировать код для вставки счетчиков в DOM с помощью инструментов jQuery для управления объектами DOM. Вот простой способ добиться этого:

```
var main = function () {
  "use strict";
  var insertCountsIntoDOM = function (counts) {
    // здесь код для управления элементами DOM
  };
  // обратите внимание: insertCountsIntoDOM вызывается
  // с параметрами счетчиков, когда getJSON возвращает их
  $.getJSON("counts.json", insertCountsIntoDOM);
}
$(document).ready(main);
```

Почему это сделано именно так? Потому, что такой способ реализации позволяет нам проделать нечто похожее на недавние операции с кодом на серверной стороне — использовать функцию `setInterval`, чтобы обновлять страницу динамически:

```
var main = function () {
  "use strict";
  var insertCountsIntoDOM = function (counts) {
    // здесь код для управления элементами DOM
  };
  // проверка значения counts каждые 5 секунд
  // и вставка обновленных значений в DOM
  setInterval(function () {
    $.getJSON("counts.json", insertCountsIntoDOM);
  }, 5000);
}
$(document).ready(main);
```

Попробуйте сами поиграть с этим кодом, перенастроить его и действительно разобраться, как это работает. Это первый пример полноценного веб-приложения, которое имеет и клиентский, и серверный компоненты, связанные друг с другом с помощью AJAX.

Создание сервера для Amazeriffic

Если вы как следует поработали с примером из предыдущей главы, то вполне справитесь с организацией работы приложения Amazeriffic так, чтобы оно обрабатывало список задач, передаваемый файлом JSON, а не жестко прописанный в программном коде. На этом примере я предлагаю вам начать с работы с хранилищем Git, чтобы отслеживать изменения, появляющиеся по ходу работы.

Настройка папок

Начнем с создания папки внутри каталога Chapter5/app, которую назовем Amazeriffic. Напомню, что не имеет значения, сделаем мы это на гостевой или на хостовой машине. Внутри папки создадим еще одну для хранения клиентского кода. Назовем ее client и скопируем туда содержимое предыдущего примера с Amazeriffic из главы 5.

Очень важное различие между этим примером и примером из главы 5 состоит в том, что мы будем хранить список задач на сервере. Это значит, что нам больше не нужен файл todo.json с начальным содержанием списка задач. Но, поскольку через некоторое время нам потребуется скопировать содержание этого файла на сервер, лучше всего сохранить его под другим именем. Чтобы переименовать файл из командной строки, используйте простую команду mv, что значит move («переместить»):

```
hostname $ mv client/todos.json client/todos.OLD.json
```

Создание хранилища Git

Затем заложим основу хранилища Git, отправив туда скопированные файлы. Помните, что взаимодействовать с Git нужно с хостовой машины. С этого момента я предоставляю вам выбирать подходящие моменты для выполнения коммитов:

```
hostname $ git init
Initialized empty Git repository ...
```

После этого мы можем проверить статус хранилища и отправить туда файлы клиентской стороны, как мы это делали в предыдущих главах. Затем можем переключиться на гостевую машину, чтобы установить модуль Express. Перейдите в каталог Amazeriffic и наберите следующую команду npm:

```
vagrant $ pwd
/home/vagrant/app/Amazeriffic
vagrant $ npm install express
```

Создание сервера

Сейчас наше приложение работает отлично, если исходить из предположения, что пользователь никогда не переходит в другой браузер или на другой компьютер и никогда не перезагружает страницу. К сожалению, если пользователь предпримет одно из этих действий, он потеряет все созданные задачи и вернется к исходной точке.

Мы решаем эту проблему, отправляя список задач на хранение на сервере и перенастраивая клиент для обработки данных с сервера. Для начала создадим список задач в переменной на сервере и настроим маршрут JSON для их доставки:

```
var express = require("express"),
http = require("http"),
app = express(),
toDos = {
  // настраиваем список задач копированием
  // содержимого из файла todos.OLD.json
};
app.use(express.static(__dirname + "/client"));
http.createServer(app).listen(3000);
// этот маршрут замещает наш файл
// todos.json в примере из главы 5
```

```
app.get("todos.json", function (req, res) {
  res.json(toDos);
});
```

Запуск сервера

Проверьте, что находитесь на гостевой машине, и запустите `app.js` через Node:

```
vagrant $ node app.js
```

После запуска мы сможем зайти на страничку нашего приложения, открыв Chrome и введя `localhost:3000/index.html` в адресную строку! На самом деле мы можем просто набрать `localhost:3000` в адресной строке, после чего сервер автоматически перенаправит нас к файлу `index.html`, который, как правило, считается страницей по умолчанию.

На этот момент все должно работать точно так же, как раньше, но мы пока не решили ни одну из заявленных ранее проблем. Это потому, что, хотя список задач и доставляется с сервера, клиент пока не посылает никаких обновлений. А мы пока не изучили, как отправлять информацию на сервер в случае, если что-то меняется на клиенте.

Размещение информации на сервере

До этого момента в программировании на клиентской стороне мы ограничивались получением данных с сервера. Это значит, что наши приложения поддерживали только одностороннюю коммуникацию между сервером и клиентом. Для обеспечения этой коммуникации мы пользовались функцией jQuery под названием `getJSON`. Оказывается, в jQuery существует и такая функция, которая позволяет легко отправлять на сервер объекты JSON, но для этого нужно модифицировать серверную сторону и подготовить ее к приему данных.

Процесс отправки данных с клиента на сервер через протокол HTTP называется `post`. Начнем с настройки маршрута `post` на сервере Express. Следующий пример настроит маршрут `post`, который просто выводит строку в терминале сервера:

```
app.post("/todos", function (req, res) {
  console.log("Данные были отправлены на сервер!");
  // простой объект отправлен обратно
  res.json({"message":"Вы размещаетесь на сервере!"});
});
```

Это очень похоже на маршруты `get`, за исключением того, что мы заменили вызов функции `get` вызовом функции `call`. На практике разница состоит в том, что мы не можем просто проложить маршрут, используя браузер, как делали с маршрутами `get`. Вместо этого нужно изменить JavaScript на клиентской стороне, разместив в нем маршрут, чтобы увидеть сообщение. Мы делаем это, когда пользователь щелкает на кнопке Добавить. Модифицируем клиентский код следующим образом:

```
// этот код находится внутри функции main,
// где изначально был определен toDoObjects
$button.on("click", function () {
  var description = $input.val(),
  tags = $tagInput.val().split(",");
  toDoObjects.push({"description":description, "tags":tags});
  // здесь мы отправляем быстрое сообщение на маршрут списка задач
  $.post("todos", {}, function (response) {
    // этот обратный вызов выполняется при ответе сервера
    console.log("Мы отправили данные и получили ответ сервера!");
    console.log(response);
  });
  // обновление toDos
  toDos = toDoObjects.map(function (toDo) {
    return toDo.description;
  });
  $input.val("");
  $tagInput.val("");
});
```

Первый аргумент функции `$.post` — маршрут, на который мы хотим отправить данные, второй — сами данные (представленные в виде объекта), которые нужно послать, и третий — обратный вызов для ответа сервера. Сейчас наш код должен работать точно так же, как раньше, но при щелчке на кнопке Добавить мы увидим, что сервер выводит сообщение в своей консоли, а клиент выводит ответ сервера. Так что мы на правильном пути, но пока что не отправили список задач на сервер. Как я уж говорил, второй аргумент функции `post` — это объект, который отправляется на сервер. Оказывается, очень просто заменить пустой объект новым объектом со списком задач, но на сервере придется проделать несколько больше работы для его использования.

Самый простой путь добиться этого — использовать URL-шифрованный плагин Express, который будет превращать JSON, отправленный jQuery, в объект JavaScript, который сможет использовать сервер:

```
var express = require("express"),
http = require("http"),
app = express()
toDos = {
  // ...
};:
app.use(express.static(__dirname + "/client"));
// командуем Express принять поступающие
// объекты JSON
app.use(express.urlencoded());
app.post("/todos", function (req, res) {
  // сейчас объект сохраняется в req.body
  var newToDo = req.body;
  console.log(newToDo);
  toDos.push(newToDo);
  // отправляем простой объект
  res.json({"message":"Вы разместили данные на сервере!"});
});
```

На этом этапе сервер будет выводить объект, когда он отправлен, в серверную консоль. После этого он будет добавлять новую задачу в список. А сейчас сделаем еще одно небольшое изменение, чтобы наше клиентское приложение действительно добавляло новый элемент в список задач:

```
$button.on("click", function () {
  var description = $input.val(),
  tags = $tagInput.val().split(","),
  // создаем новый элемент списка задач
  newToDo = {"description":description, "tags":tags};
  $.post("todos", newToDo, function (result) {
    console.log(result);
    // нужно отправить новый объект на клиент
    // после получения ответа сервера
    toDoObjects.push(newToDo);
    // обновляем toDos
    toDos = toDoObjects.map(function (toDo) {
      return toDo.description;
    });
    $input.val("");
    $tagInput.val("");
  });
});
```

Таким образом новый элемент списка задач будет отправлен на сервер, а сообщение с ответом сервера — выведено в консоли JavaScript. После всего этого новый объект `toDo` обновит список задач на клиентской стороне.

Теперь мы можем открыть приложение в двух разных окнах браузера, добавить новый элемент списка задач в одном окне, а затем перезагрузить страницу во втором, чтобы увидеть обновленное содержание!

Подведем итоги

В этой главе мы изучили основы программирования на серверной стороне с помощью Node.js. Node.js — это платформа, которая позволяет программировать сервер на языке JavaScript. В частности, мы используем его для написания серверов HTTP, которые связываются с клиентским приложением, запущенным в браузере.

Программы Node.js, как правило, состоят из нескольких отдельных модулей. Мы можем создавать собственные модули, написав какой-либо код, а затем выделив нужные части в объекте `module.exports`. После этого мы можем импортировать их в другую программу с помощью функции `require`.

NPM означает *Node.js Package Manager* (менеджер пакетов Node.js) и служит для установки модулей Node.js, которые мы не писали самостоятельно и которые не включены в дистрибутив Node.js. Например, модуль Express мы установили с помощью NPM. Express — это оболочка модуля http в Node.js, включающая в себя большинство наиболее частых требований к серверу HTTP. В частности, она позволяет легко доставлять файлы клиентской стороны — HTML, CSS и JavaScript.

Кроме того, с ее помощью можно настраивать специфические маршруты для обработки данных сервером.

В качестве базового окружения мы использовали VirtualBox и Vagrant. Эти инструменты не являются необходимыми для понимания работы веб-приложения, но облегчают запуск процесса разработки. Кроме того, с их помощью можно четко разделить клиент (браузер, запущенный на хостовой машине) и сервер (программу Node.js, запущенную на гостевой машине).

И последнее по порядку, но не по значимости: мы научились настраивать запросы HTTP post с помощью jQuery, а также заставлять сервер отвечать на запросы post через Express.

Больше теории и практики

Локальная установка Node.js

В этой главе мы установили Node.js на виртуальной машине с помощью Vagrant. Основная причина этого состоит в том, что сценарии Vagrant одновременно настроили программное обеспечение, которое понадобится нам в главах 7 и 8. Однако установить Node.js на локальном компьютере очень, очень просто, и все примеры из этой главы будут работать на вашей локальной машине с минимальными изменениями. Если вы зайдете на http://nodejs.org/download, то сможете найти установочные пакеты для своей платформы.

Вам очень полезно будет попробовать сделать это. Если вы установите ПО на своей хостовой машине, то сможете инсталлировать и несколько других полезных инструментов, включая JSHint, CSS Lint и валидатор HTML5.

JSHint и CSS Lint через NPM

В главе 4 мы познакомились с JSLint — инструментом для контроля качества кода, который проверяет соответствие кода стандартам качества JavaScript. Мы использовали сетевой инструмент, доступный по адресу http://jslint.com, где нужно было копировать и вставлять код в браузер. Пока это не вызывало никаких трудностей, но с разрастанием кода работать с сайтом станет неудобно.

Оказывается, что если Node.js установлен на локальной машине, то с помощью NPM можно установить другие инструменты с интерфейсом командной строки, которые могут сделать работу очень удобной, например JSHint. JSHint очень похож на JSLint и также может использоваться для проверки вашего кода. Пройдите в командную строку и установите его с помощью NPM. Мы можем использовать флажок `-g`, чтобы указать NPM, что хотим установить пакет глобально. Таким образом, мы сможем использовать JSHint как стандартное приложение с интерфейсом командной строки вместо установки его в качестве библиотеки в каталоге node_modules:

```
hostname $ sudo npm install jshint -g
```

В Mac OS и Linux нужно использовать sudo для установки пакетов глобально. В Windows это не обязательно.

Сделав это, мы можем запустить JSHint непосредственно для обработки каких-либо файлов. Допустим, у нас есть файл под названием `test.js`:

```
var main = function () {
  console.log("hello, world");
}
$(document).ready(main);
```

Этот код, конечно, будет отлично работать в браузере, но в нем пропущена точка с запятой после определения функции `main`. JSHint предупредит об этом:

```
hostname $ jshint test.js
test.js: line 3, col 2, Missing semicolon.[1]
1 error
```

Если же с кодом все в порядке, JSHint не даст никакого ответа. Если мы добавим пропущенную точку с запятой и запустим JSHint снова, то не увидим никакого сообщения:

```
hostname $ jshint test.js
hostname $
```

Установка JSHint на локальной машине делает проверку качества кода (как серверного, так и клиентского) гораздо более простой, так что нет причин этого не делать. Вы также обнаружите, что JSHint может быть очень гибким, позволяя легко менять опции проверки с помощью командной строки. Документация находится по адресу http://jshint.com/docs.

Аналогично NPM позволяет установить и запустить CSS Lint из командной строки, что делает проверку CSS на наличие ошибок и соответствие стандартам качества очень простой:

```
hostname $ sudo npm install csslint -g
```

Для валидаторов и проверочных средств HTML тоже есть несколько функций. Если вам придется много работать с HTML, вы легко найдете необходимые для этого инструменты.

Обсудим код счетчика твитов

Наш счетчик твитов отслеживает массив слов в Twitter. Однако в коде мы должны перечислять слова в нескольких разных местах. Например, в начале файла `tweet_counter.js` видим следующее:

[1] Сообщение: «test.js: строка 3, столбец 2, пропущена точка с запятой». — *Примеч. пер.*

```
counts.awesome = 0;
counts.cool = 0;
counts.rad = 0;
counts.gnarly = 0;
counts.groovy = 0;
```

А позже видим указание модулю ntwitter отслеживать эти слова:

```
{ "track": ["awesome", "cool", "rad", "gnarly", "groovy"] },
```

Если вы уже пытались оптимизировать этот код, то, возможно, сделали нечто похожее для приращения счетчиков:

```
if (tweet.indexOf("awesome") > -1) {
  // приращение счетчика awesome
  counts.awesome = counts.awesome + 1;
}
if (tweet.indexOf("cool") > -1) {
  // приращение счетчика cool
  counts.cool = counts.cool + 1;
}
```

Что здесь плохого? Если нужно удалить или добавить какое-нибудь слово, то придется изменять код в трех местах! Как можно решить эту проблему? Для начала можно определить массив в одном определенном месте — в начале модуля tweet_counter:

```
var trackedWords = ["awesome", "cool", "rad", "gnarly", "groovy"];
```

Таким образом, при создании счетчика твитов мы можем использовать переменную:

```
{ "track": trackedWords };
```

В оставшейся части кода нам пригодится полезнейшее свойство JavaScript — возможность использовать объект как *схему* (map) структуры данных, где атрибутами являются строки. Эквивалентными будут два следующих способа индексации в объект:

```
// доступ к счетчику awesome
// получен через оператор точка (dot)
counts.awesome = 0;
counts.awesome = counts.awesome + 1;
// доступ к счетчику awesome
// получен через строку
counts["awesome"] = 0;
counts["awesome"] = counts["awesome"] + 1;
```

В общем случае стоит применить способ с точкой, так как большинство программистов JavaScript сочтут его лучше читаемым (да и JSLint будет с этим согласен). Но у способа со строкой и квадратными скобками есть более весомое преимущество — мы можем использовать переменные для доступа к значениям:

```
var word = "awesome";
```

```
counts[word] = 0;
counts[word] = counts[word] + 1;
```

Вы догадываетесь, к чему я веду? Мы можем добиться того, чтобы объект счетчика зависел исключительно от массива:

```
// создается пустой объект
var counts = {};
trackedWords.forEach(function (word) {
  counts[word] = 0;
});
```

Цикл перебирает каждое слово и устанавливает начальное значение счетчика равным нулю. Аналогичным образом с помощью цикла `forEach` мы можем оптимизировать код, проверяющий, содержит ли твит какое-либо слово. Сделав это, мы можем легко добавлять слова в изначальный массив и удалять их из него, а приложение будет обновляться!

Это сделало наш код лучше управляемым, но мы можем улучшить его еще немножко. Представьте, что мы позволим пользователю нашего модуля вместо экспорта счетчиков для определенных слов самостоятельно решать, какие слова будут отслеживаться. Например, пользователь модуля может захотеть использовать его как-то так:

```
var tweetCounter = require("./tweet_counter.js"),
counts;
// таким образом, наш tweetCounter запускается для
// определенных слов вместо жестко прописанного списка
counts = tweetCounter(["hello", "world"]);
```

Мы можем добиться этого, экспортировав функцию вместо объекта `counts`:

```
var setUpTweetCounter = function (words) {
  // настраиваем объект counts
  // и поток ntwitter
  // для использования массива слов
  // ...
  // а в конце возвращаем счетчик
  return counts;
}
module.exports = setUpTweetCounter;
```

Таким образом, в перспективе наш модуль становится более гибким в применении. Программа, которая использует модуль, может решать самостоятельно, какие слова будут отслеживаться, даже не заглядывая в код. Эта настройка — отличное упражнение, и вам стоит попробовать выполнить ее!

API покерного приложения

Вот несложный проект, который позволит вам попрактиковаться в создании API с помощью Express. Создайте приложение Express, которое отвечает на единичный маршрут получения данных `/hand`. Этот маршрут должен принимать объект, озна-

чающий раздачу карт, а затем отвечать объектом JSON, который содержит лучшую из имеющихся комбинаций. Например, получен следующий объект:

```
[
  { "rank":"двойка", "suit":"пик" },
  { "rank":"четверка", "suit":"червей" },
  { "rank":"двойка", "suit":"треф" },
  { "rank":"король", "suit":"пик" },
  { "rank":"восьмерка", "suit":"бубен" }
]
```

Наше API с помощью сообщения `null` (мы говорили о нем в главе 5), означающего отсутствие объекта, может ответить вот так:

```
{
  "handString":"пара",
  "error": null
}
```

Другими словами, в этом примере мы отправили корректную раздачу (пять карт), а максимальная комбинация, которую можно из нее состряпать, — пара. Мы установили для свойства `error` значение `null`, означающее, что ошибок здесь нет. В случае же, если кто-то отправит некорректную раздачу (например, с неправильными номиналами карт или слишком большим количеством карт), мы установим значение `handString` на `null` и отправим обратно сообщение об ошибке:

```
{
  "handString": null,
  "error": "Раздача некорректна!"
}
```

Для достижения всего этого написанные функции следует выделить в модуль Node.js, как описано в главе 5. Для использования в приложении его нужно импортировать на Express-сервер:

```
var poker = require("./poker.js");
```

А затем обработать функции следующим образом:

```
var hand = [
  { "rank":"двойка", "suit":"пик" },
  { "rank":"четверка", "suit":"червей" },
  { "rank":"двойка", "suit":"треф" },
  { "rank":"король", "suit":"пик" },
  { "rank":"восьмерка", "suit":"бубен" }
]
var hasPair = poker.containsPair(hand);
// hasPair сейчас будет true
```

Как это сделать? Создадим в определении модуля объект `poker` с несколькими функциями:

```
var poker = {};
poker.containsPair = function (hand) {
```

```
  // ... определение функции
}
poker.containsThreeOfAKind = function (hand) {
  // ... определение функции
}
module.exports = poker;
```

Модуль `poker` будет иметь несколько встроенных функций, которые мы не станем включать в экспортированный объект. Например, мы включили функцию `containsNTimes` в модуль, но не экспортируем ее:

```
var poker = {},
containsNTimes; // объявление функции
// здесь мы определяем наши "приватные" функции,
// которые не будут добавлены в объект poker
containsNTimes = function (array, item, n) {
  // ... определение функции
};
poker.containsPair = function (hand) {
  // ... используйте здесь containsNTimes
}
poker.containsThreeOfAKind = function (hand) {
  // ... используйте здесь containsNTimes
}
// экспортируем только функции, относящиеся к покеру
module.exports = poker;
```

Может понадобиться включить еще несколько функций для проверки того, что раздача действительно является покерной, и даже специальную функцию для возврата объекта, указанного API. Таким образом, обратный вызов маршрута, по сути, уместится в одной строке:

```
app.post("/hand", function (req, res) {
  var result = poker.getHand(req.body.hand);
  res.json(result);
});
```

Чтобы все это работало, нужно написать клиентское приложение, создающее несколько раздач и отправляющее их на сервер с помощью jQuery. Если вам удастся заставить все это работать, ваше обучение действительно начинает приносить плоды!

7 Хранение данных

В нескольких предыдущих главах мы научились создавать несложные серверы с помощью Node.js, а также обмениваться данными между сервером и клиентом с помощью AJAX. Одно из заданий, которые мы выполняли, включало в себя отслеживание количества упоминаний различных слов в ленте Twitter и отображение счетчика на клиентской стороне.

Одна из основных проблем этого приложения заключается в том, что всю информацию, необходимую для работы, оно хранит в памяти программы Node.js. Это значит, что если мы прервем серверный процесс, то одновременно исчезнут и счетчики чисел. Это весьма нежелательно, потому что сервер придется останавливать часто, например, чтобы обновить его, а скорее, сервер может остановиться сам по причине имеющегося бага. Было бы хорошо, если бы полученные нами счетчики не повреждались, когда происходит что-то из названного.

Чтобы решить эту проблему, нужно специальное приложение для хранения данных, работающее независимо от программы. В этой главе мы изучим два различных способа решения данной проблемы с использованием хранилищ данных в формате не-SQL. В частности, познакомимся с Redis и MongoDB, а также рассмотрим, как взаимодействовать с ними в приложениях Node.js.

SQL и не-SQL

Если вы изучали базы данных, то, наверное, вам знакома аббревиатура SQL (иногда произносится как sequel[1]), что значит Structured Query Language (структурированный язык запросов). Это язык, используемый для запросов к базе, в которой данные хранятся в *реляционном* формате. Реляционные базы данных хранят данные в ячейках таблиц, в них таблицы легко связываются с другими таблицами с помощью перекрестных ссылок. Например, в реляционной базе данных может храниться

[1] Читается «сиквел», на русском языке тоже так говорят, но редко. — *Примеч. пер.*

таблица фамилий актеров и актрис, а также отдельная таблица с названиями фильмов. А реляционная таблица будет использоваться для связи актеров и актрис с фильмами, в которых они играли.

Данный подход имеет определенные преимущества, но главное, почему такие базы очень широко используются, — это то, что хранение информации таким способом минимизирует дублирование данных и, следовательно, требует меньше пространства, чем альтернативные подходы. В прошлом это было очень важно, так как хранение большого количества данных стоило очень дорого. Но в последние годы цена хранения данных многократно снизилась, поэтому минимизация избыточности данных уже не так важна, как раньше.

По этой причине исследователи и инженеры пересмотрели свои убеждения относительно хранения данных и начали экспериментировать с хранением в нереляционном виде. Новые способы хранения данных, иногда называемые не-SQL (NoSQL), обеспечили компромисс: так, можно хранить избыточную информацию, но при этом повысить простоту использования с точки зрения программирования. Более того, некоторые из этих хранилищ данных были разработаны специально для особых сценариев использования, например для приложений, где чтение данных должно быть эффективнее записи.

Redis

Redis — великолепный пример хранения данных в формате не-SQL. Он был разработан для быстрого доступа к данным, которые часто используются. Это ускорение достигается за счет надежности, так как данные хранятся в памяти, а не на диске (технически Redis периодически выполняет сохранение на диск, но в основном вы можете думать о нем как о хранилище данных в памяти).

Redis хранит информацию в формате «ключ — значение» — вы, возможно, увидите здесь аналогию со способом, который использует JavaScript для хранения свойств объекта. Таким образом разработчик получает возможность организовать хранение данных традиционной структуры (хэш, списки, наборы и т. д.), то есть в виде естественного продолжения хранения данных в программе. Это отличное решение для данных, к которым необходим быстрый доступ, или для временного хранения часто используемой информации (это называется *кэшированием*), оно улучшает время отклика наших приложений.

Мы не будем изучать эти сценарии в данной книге, но не забывайте о них во время чтения. Использовать Redis мы будем довольно примитивным образом, так как нам нужно всего лишь сохранить счетчики Twitter отдельно от сервера Node.js. Для этого присвоим каждому слову ключ, а все значения будут целочисленными переменными, представляющими собой количество раз использования каждого слова. Перед тем как научиться программировать это, поговорим о взаимодействии с Redis с помощью командной строки.

Взаимодействие с Redis через клиентскую командную строку

Чтобы провести черту под предыдущей работой, начнем с копирования проекта `node-dev-bootstrap` в папку Projects, создав папку Chapter7. Если вам нужно вспомнить, как это сделать, обратитесь к главе 6.

Покончив с копированием, войдите в папку, запустите виртуальную машину и подключитесь к гостевой машине. Поскольку вы будете восстанавливать машину с нуля, это займет некоторое время:

```
hostname $ cd Projects/Chapter7
hostname $ vagrant up
...vagrant build stuff...
hostname $ vagrant ssh
```

Сейчас вы должны быть авторизованы на виртуальной машине, где уже инсталлирован и настроен Redis. Можно использовать команду `redis-cli` для старта интерактивного клиента Redis:

```
vagrant $ redis-cli
redis 127.0.0.1:6379>
```

Хранить данные в Redis так же просто, как использовать команду `set`. Здесь мы создадим ключ для слова `awesome` и установим его значение равным 0:

```
redis 127.0.0.1:6379> set awesome 0
OK
```

Если все идет хорошо, Redis должен ответить «OK». Мы можем проверить значение ключа с помощью команды `get`:

```
redis 127.0.0.1:6379> get awesome
"0"
```

Получив сохраненное значение, мы можем делать с ним очень многое. Конечно, больше всего мы заинтересованы в его приращении на единицу, когда встречается слово, использованное в твите. К счастью, оказывается, что в Redis есть команда `incr` (от increment — «прирастить»), которая делает именно это:

```
redis 127.0.0.1:6379> incr awesome
(integer) 1
redis 127.0.0.1:6379> incr awesome
(integer) 2
redis 127.0.0.1:6379> get awesome
"2"
```

Команда `incr` добавляет 1 к величине, которая в настоящее время ассоциирована с определенным ключом (в данном случае ключ — `awesome`). Чтобы выйти из интерактивного клиента Redis, мы можем набрать `exit`.

Эти три команды (`set`, `get`, `incr`) — все, что нам нужно знать, чтобы приступить к организации хранения счетчиков, но на самом деле возможности Redis гораздо

шире, чем описанные мной. Чтобы попробовать некоторые из функций, изучите великолепный интерактивный учебник по Redis, домашняя страница которого изображена на рис. 7.1.

Установка модуля Redis через файл package.json

К этому моменту мы уже знаем, как создать ключ для каждого слова, а затем интерактивно наращивать его значение, но хотелось бы проделать это с помощью программирования Node.js из наших приложений.

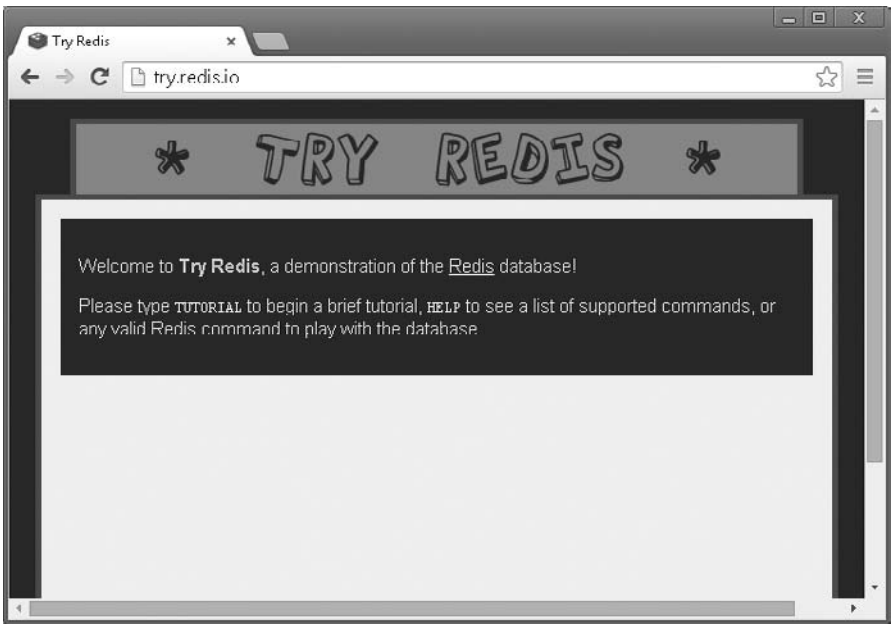


Рис. 7.1. Домашняя страница интерактивного учебника по Redis

Будем работать с проектом о Twitter из предыдущей главы, поэтому скопируйте папку Twitter из Chapter6/app в Chapter7/app. Вы можете использовать команду `cp` для выполнения этой задачи из командной строки или просто воспользоваться обычным способом через графический интерфейс.

Чтобы заставить программу Node.js взаимодействовать с Redis, нужно установить модуль `node-redis`. Раньше мы из командной строки использовали команду `npm` для установки модуля `ntwitter`. В этом был смысл в начале работы, но, продвигаясь вперед, важно отслеживать взаимозависимости более формальным путем. Для этого используем специальный файл `package.json`, который отслеживает изменения различных аспектов вашего проекта, включая их взаимозависимости.

В папке app создайте файл с именем `package.json` и скопируйте в него следующий код:

```
{
  "name": "tutorial",
  "description": "a tutorial on using node, twitter, redis, and express",
  "version": "0.0.1",
  "dependencies": {
    "ntwitter": "0.5.x",
    "redis": "0.8.x"
  }
}
```

NPM предлагает интерактивный способ создания файла package.json, но вам придется ответить на множество вопросов о нем. Если хотите попробовать, наберите npm init, находясь в папке вашего проекта.

Как видите, таким образом указаны несколько деталей о нашем проекте, включая его взаимозависимости (поскольку для Node.js рекомендуется клиент `node-redis`, он просто обращается к `redis` в NPM). Мы используем этот способ в главе 8, чтобы поместить приложение и все его взаимозависимости в Cloud Foundry. А сейчас `package.json` нужен нам только для упрощения установки. В частности, мы можем сейчас инсталлировать все взаимозависимости, набрав следующую команду на гостевой машине:

```
vagrant $ cd app
vagrant $ npm install
```

Взаимодействие с Redis в коде

На хостовой машине с помощью текстового редактора откройте файл `tweet_counter.js`, который мы создали в предыдущей главе. Он должен находиться в папке Twitter в каталоге app. Если вы работали с примерами и задачами, ваш код, должно быть, лучше проработан, чем мой. В любом случае следуйте за мной и модифицируйте код для импорта и использования модуля Redis:

```
var ntwitter = require("ntwitter"),
redis = require("redis"), // требование модуля redis
credentials = require("./credentials.json"),
redisClient,
twitter,
// объявление объектов счетчиков для хранения счетчиков
counts = {};
twitter = ntwitter(credentials);
// создание клиента для подключения к Redis
client = redis.createClient();
// инициализация счетчиков
counts.awesome = 0;
twitter.stream(
```

```
  "statuses/filter",
  { track: ["awesome", "cool", "rad", "gnarly", "groovy"] },
  function(stream) {
    stream.on("data", function(tweet) {
      if (tweet.text.indexOf("awesome") >= -1) {
        // приращение ключа на клиенте
        client.incr("awesome");
        counts.awesome = counts.awesome + 1;
      }
    });
  }
);
module.exports = counts;
```

Мы использовали функцию `incr` аналогично тому, как делали это с интерактивным клиентом Redis. Фактически мы можем использовать любую из команд Redis тем же способом — клиент предоставляет функцию с тем же самым названием команды.

Мы можем запустить этот код обычным способом. Дайте ему поработать немного — он должен продолжать отслеживать количество твитов, содержащих слово awesome в Redis, и, если код `setInterval` остался на месте, мы увидим, что периодически он выводит значения. Убедившись, что код обработал несколько твитов, мы можем проверить, что счетчики хранятся в Redis, остановив программу (нажатием Ctrl+C), а затем повторно подключившись к `redis-cli`. После этого можно использовать команду `get`, чтобы увидеть сохраненное значение упоминаний слова awesome:

```
vagrant $ redis-cli
redis 127.0.0.1:6379> get awesome
(integer) "349"
```

Если вдруг вы захотите очистить хранилище данных Redis, можете сделать это в любой момент. Нужно набрать команду flushall в командной строке redis-cli.

Установление начального значения счетчиков из хранилища Redis

Сейчас наши данные сохраняются в течение работы программы, но, перезапуская сервер, мы все еще обнуляем объект `count`. Для окончательного решения этой проблемы нужно, чтобы счетчик твитов устанавливал в качестве начального значения данные, которые сохранены в Redis. Мы можем использовать команду `get`, чтобы получить их перед тем, как запустим поток. Но, как и многое в Node, функция `get` является асинхронной, поэтому работать с ней надо осторожно:

```
var ntwitter = require("ntwitter"),
redis = require("redis"), // требование модуля Redis
credentials = require("./credentials.json"),
redisClient,
counts = {},
twitter;
twitter = ntwitter(credentials);
client = redis.createClient();
// обратный вызов получает два аргумента
client.get("awesome", function (err, awesomeCount) {
  // проверка, что ошибки здесь нет
  if (err !== null) {
    console.log("ERROR: " + err);
    // выход из функции
    return;
  }
  // установление счетчиков в целое
  // значение, хранящееся в Redis, или к нулю,
  // если оно не установлено
  counts.awesome = parseInt(awesomeCount,10) || 0;
  twitter.stream(
    "statuses/filter",
    { track: ["awesome", "cool", "rad", "gnarly", "groovy"] },
    function(stream) {
      stream.on("data", function(tweet) {
        if (tweet.text.indexOf("awesome") >= -1) {
          // приращение ключевого значения клиента
          client.incr("awesome");
          counts.awesome = counts.awesome + 1;
        }
      });
    }
  );
});
module.exports = counts;
```

Вы заметили, что обратный вызов для `get` принимает два параметра: `err` и `awesomeCount`? Параметр `err` представляет собой условие отсутствия ошибки и будет объектом `error`, если в запросе есть какая-либо проблема. Если же с запросом все хорошо, он будет равен `null`. Обычно, выполняя запрос к хранилищу данных, первое, что мы делаем в ответ, — проверяем на отсутствие ошибок и обрабатываем запрос тем или иным образом. В противном случае просто выводим информацию о появлении какой-либо проблемы. Но в случае вывода ваших приложений на рынок ошибки нужно обрабатывать более корректно.

Затем вы видите, что необходимо каким-то образом обработать значение `awesomeCount`. Поскольку Redis хранит все величины как строки, мы должны преобразовать значение в целое число, чтобы выполнять с ним арифметические операции в JavaScript. В данном случае мы использовали глобальную функцию `parseInt`, которая извлекает числовое значение из строки, возвращенной Redis. Второй

параметр, запрошенный parseInt, называется radix и означает, что нам нужны цифры, составляющие число. Если же величина не число, parseInt возвращает значение NaN, что означает — думаю, это понятно — Not a Number, то есть не число.

Помните, что || относится к оператору JavaScript «ИЛИ». Этот оператор вернет первое значение в списке величин, которые не являются ложными, что значит — они не равны false, 0 или NaN. Если ложные все величины, оператор вернет последнее.

В общем, данная строка кода переводится как «используй величину awesomeCount, если она определена, или 0 в ином случае». Это позволяет нам обнулить код, когда awesome не определена в отдельной строке кода. В этот момент вам стоит переработать код с учетом всего сказанного ранее, но сначала лучше изучить еще одну команду Redis.

Использование mget для получения нескольких величин

Команда get прекрасно подходит для единичной пары «ключ — величина», но что произойдет, если нам нужно запросить величину, связанную с несколькими ключами? Это почти так же просто, если использовать функцию mget. Мы можем преобразовать код следующим образом:

```
client.mget(["awesome", "cool"], function (err, results) {
  if (err !== null) {
    console.log("ERROR: " + err);
    return;
  }
  counts.awesome = parseInt(results[0], 10) || 0;
  counts.cool = parseInt(results[1], 10) || 0;
}
```

С помощью mget сможем преобразовать код для обработки всех слов, которые нужно отслеживать.

Redis прекрасно справляется с хранением простых данных, которые могут представляться в виде строк, включая объекты JSON. Но если нам нужен несколько больший контроль над данными JSON, лучше использовать хранилище, разработанное специально для JSON. MongoDB — отличный пример такой технологии.

MongoDB

MongoDB (или для краткости Mongo) — база данных, которая позволяет хранить данные на диске, но не в реляционном формате. Mongo является документоориентированной базой данных, которая концептуально позволяет хранить данные в виде коллекций в формате JSON (технически MongoDB хранит свои данные в формате BSON, но в нашем случае мы вполне можем думать о них как о JSON). Кроме того, она позволяет полноценно взаимодействовать с ними с помощью JavaScript!

Mongo может быть использована для более сложных задач хранения данных, например хранения учетных записей пользователей или комментариев к постам в блогах. Или даже для хранения бинарных данных, например изображений! Mongo отлично подходит для независимого хранения объектов задач Amazeriffic на нашем сервере.

Взаимодействие с MongoDB из клиента с интерфейсом командной строки

Как и Redis, Mongo предлагает клиент с интерфейсом командной строки, который позволяет напрямую взаимодействовать с хранилищем данных. Вы можете запустить клиент Mongo, набрав `mongo` в командной строке на вашей гостевой машине:

```
vagrant $ mongo
MongoDB shell version: 2.4.7
connecting to: test
>
```

При первом запуске вы можете увидеть несколько начальных предупреждений, но это совершенно нормально.

Одно из основных различий между Redis и Mongo состоит в том, что мы можем взаимодействовать с Mongo с помощью JavaScript! Например, здесь мы создаем переменную под названием `card` (карта) и сохраняем в ней объект:

```
> var card = { "rank":"туз", "suit":"треф" };
> card
{ "rank" : "туз", "suit" : "треф" }
>
```

Аналогичным образом можем создавать массивы и управлять ими. Отмечу, что в этом примере мы не завершаем операторы в конце каждой строки. После нажатия Enter Mongo выдает три точки, давая знать, что предыдущий оператор не завершен. Mongo автоматически выполнит первый оператор JavaScript:

```
> var clubs = [];
> ["двойка", "тройка", "четверка", "пятерка"].forEach(
... function (rank) {
  ... cards.push( { "rank":rank, "suit":"треф" } )
... });
> clubs
[
  {
    "rank" : "двойка",
    "suit" : "треф"
  },
  {
```

```
    "rank" : "тройка",
    "suit" : "треф"
  },
  {
    "rank" : "четверка",
    "suit" : "треф"
  },
  {
    "rank" : "пятерка",
    "suit" : "треф"
  }
]
```

Другими словами, клиент Mongo с интерфейсом командной строки работает почти как консоль JavaScript в Chrome. Но это сходство исчезнет, как только мы примемся работать с хранением данных. Mongo организует данные как *документы*, которые мы можем рассматривать как объекты JSON. *Коллекции* документов хранятся в *базах данных*. Мы можем увидеть базы данных MongoDB с помощью команды `show dbs`:

```
> show dbs
local 0.03125GB
```

Локальная база данных всегда находится здесь. Мы можем переключиться на другую базу данных с помощью команды `use`:

```
> use test
switched to db test
```

(ответ: переключились на базу данных `test`).

Выбрав для использования нужную базу данных, мы можем получить доступ к ней через объект `db`. Мы можем сохранять объекты в коллекции, вызвав функцию `save` для коллекции. Если коллекции еще не существует, Mongo ее создаст. Здесь мы сохраняем карту, созданную ранее, в нашу коллекцию:

```
> show collections;
> db.cards.save(card);
> show collections;
cards
system.indexes
```

Вы видите, что коллекция `cards` не существовала перед тем, как в нее сохранили объект. Мы можем использовать для коллекций функцию `find`, не требующую аргументов, чтобы увидеть, какие документы там хранятся:

```
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"),"rank":"туз","suit":"треф" }
```

В дополнение к свойствам `rank` (номинал) и `suit` (масть) карта имеет также `_id`, связанный с ней. В большинстве случаев каждому документу в коллекции MongoDB присваивается идентификационный номер.

Мы можем не только сохранить единичный элемент, но и добавить несколько документов в коллекцию, используя один вызов `save`. Проделаем это на примере массива `clubs` (трефы), созданного ранее:

```
> db.cards.save(clubs);
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"),"rank":"туз","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"),"rank":"двойка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"),"rank":"тройка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"),"rank":"четверка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"),"rank":"пятерка","suit":"треф" }
```

Мы можем также добавить в базу данных больше объектов:

```
>hearts = [];
> ["двойка", "тройка", "четверка", "пятерка"].
... forEach(function (rank)
  ... { hearts.push( { "rank":rank, "suit":"треф" } )
... });
> db.cards.save(hearts);
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"),"rank":"туз","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"),"rank":"двойка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"),"rank":"тройка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"),"rank":"четверка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"),"rank":"пятерка","suit":"треф" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558de"),"rank":"двойка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558df"),"rank":"тройка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e0"),"rank":"четверка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e1"),"rank":"пятерка","suit":"червей" }
```

После того как в коллекции появится много разнообразных документов, мы можем получить их, создавая запросы из объектов JSON, которые представляют собой свойства нужного нам документа. Например, мы можем запросить все документы с картами номиналом двойка и сохранить их в переменной под названием `twos`:

```
> var twos = db.cards.find({"rank":"двойка"});
> twos
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"),"rank":"двойка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558de"),"rank":"двойка","suit":"треф" }
```

Или выбрать все тузы:

```
> var aces = db.cards.find({"rank":"туз"});
> aces
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"),"rank":"туз","suit":"треф" }
```

Можем также удалить элементы из коллекции, вызвав метод `remove` и отправив его в запросе:

```
> db.cards.remove({"rank":"двойка"});
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"),"rank":"тройка","suit":"треф" }
```

```
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"),"rank":"четверка","suit":"треф" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"),"rank":"пятерка","suit":"треф" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558df"),"rank":"тройка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e0"),"rank":"четверка","suit":"червей" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e1"),"rank":"пятерка","suit":"червей" }
```

Или же удалить все документы из коллекции, вызвав `remove` с пустым запросом:

```
> db.cards.remove();
> db.cards.find();
>
```

Аналогично Redis MongoDB представляет интерактивный учебник (рис. 7.2), который вы можете опробовать в своем браузере. Я рекомендую вам самостоятельно поработать с этим учебником, чтобы узнать немного больше о функциональности MongoDB, а также доступных типах запросов.

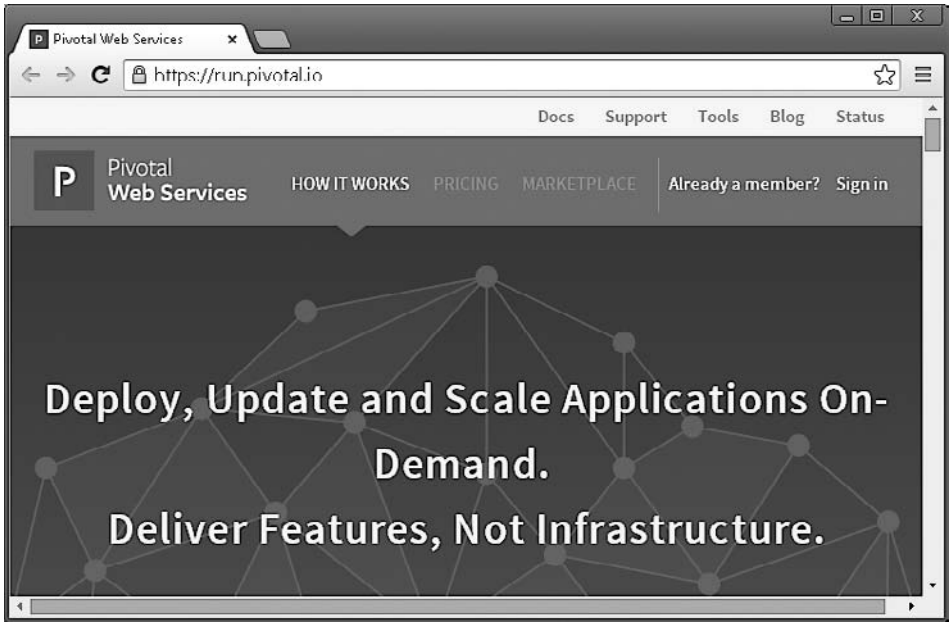


Рис. 7.2. Домашняя страница интерактивного учебника MongoDB

В конечном итоге, однако, мы не будем использовать особенно много команд Mongo для Node.js по умолчанию. Вместо этого будем моделировать данные как объекты с помощью модуля Node.js, называемого Mongoose.

Моделирование данных с Mongoose

Mongoose — это модуль Node.js, который служит для двух основных целей. Прежде всего он работает как клиент для MongoDB аналогично тому, как модуль `node-redis`

работает в качестве клиента Redis. Но кроме этого, Mongoose работает как инструмент для *моделирования данных*, позволяющий представлять документы как объекты в наших программах. В этом разделе мы изучим основы моделирования данных и используем Mongoose для создания модели данных для задач Amazeriffic.

Модели

Модель данных — это всего лишь объектное представление коллекции документов в хранилище данных. Кроме указания полей, которые присутствуют в каждом документе коллекции, она добавляет для базы данных MongoDB операции наподобие `save` или `find` для соответствующих объектов.

В Mongoose модель данных содержит схему, которая описывает структуру всех объектов такого типа. Например, предположим, что мы хотим создать модель данных для коллекции игральных карт. Начнем с указания схемы для карты — в частности, непосредственно укажем, что каждая карта обладает номиналом и мастью. В нашем файле JavaScript это будет выглядеть примерно так:

```
var CardSchema = mongoose.Schema({
  "rank" : String,
 "suit" : String
});
```

Имея схему, построить модели очень просто. По договоренности мы используем заглавную букву для объектов моделей данных:

```
var Card = mongoose.model("Card", CardSchema);
```

Схемы могут быть более сложными. Например, мы можем построить схему для записей в блоге, которая будет содержать даты и комментарии. В этом примере атрибут `comments` (комментарии) будет представлять собой массив строк вместо единичной строки:

```
var BlogPostSchema = mongoose.Schema({
  title: String,
  body : String,
  date : Date,
  comments : [ String ]
});
```

Как же модель может нам помочь? Имея модель, мы можем создать объект соответствующего ей типа очень просто, используя оператор `JavaScript new`. Например, такая строка кода создает туз пик и сохраняет его в переменной под названием `c1`:

```
var c1 = new Card({"rank":"туз", "suit":"пик"})
```

Отлично, но не проще ли проделать то же самое с помощью вот такой строки кода:

```
var c2 = { "rank":"туз", "suit":"пик" };
```

Разница состоит в том, что объект Mongoose позволяет нам взаимодействовать с базой данных с помощью некоторых встроенных функций:

```
// сохраняем эту карту в хранилище данных
c1.save(function (err) {
  if (err !== null) {
    // объект не был сохранен
    console.log(err);
  } else {
    console.log("Объект не был сохранен!");
  }
});
```

Мы также можем взаимодействовать с моделью напрямую, получая объекты из базы данных с помощью функции `find` для данной части модели. Аналогично функции `find` в интерактивном клиенте MongoDB эта функция принимает произвольные запросы MongoDB. Разница состоит в том, что она ограничивает саму себя типами, которые определены моделью:

```
Card.find({}, function (err, cards) {
  if (err !== null) {
    console.log("ОШИБКА: " + err);
    // возвращаемся из функции
    return;
  }
  // если мы попали сюда, значит, ошибки нет
  Cards.forEach(function (card) {
    // это введет все карты в базу данных
    console.log (card.rank + " of " + card.suit);
  });
});
```

Мы можем также находить какие-либо элементы (через их идентификаторы или другие запросы), обновлять их и снова сохранять. Например, допустим, что нужно изменить масть всех карт червей на пик:

```
Card.find({"suit" : " червей"}, function (err, cards) {
  cards.forEach(function (card) {
    // обновляем карты пик
    card.suit = "пик";
    // сохраняем обновленную карту
    card.save(function (err) {
      if (err) {
        // если объект не был сохранен
        console.log(err);
      }
    });
  });
});
```

И наконец, мы можем удалять элементы из базы данных, вызывая функцию `remove` для модели:

```
Card.remove({ "rank":"туз", "suit":"треф" }, function (err) {
  if (err !== null) {
```

```
    // если объект не был успешно удален
    console.log(err);
  }
});
```

К этому моменту мы рассмотрели примеры четырех основных операций над данными с помощью Mongoose: создание, чтение, обновление и удаление. Но мы все еще не видели работу Mongoose в веб-приложении! Рассмотрим ее сейчас.

Хранение списка задач для Amazeriffic

Для начала скопируем текущую версию кода Amazeriffic (и клиентскую и серверную часть) в папку Chapter7. Нам понадобится файл `package.json`, где присутствует взаимосвязь с Mongoose:

```
{
  "name": "amazeriffic",
  "description": "The best to-do list app in the history of the world",
  "version": "0.0.1",
  "dependencies": {
    "mongoose": "3.6.x"
  }
}
```

Сделав это, запустим `npm install` для установки модуля `mongoose`. Затем можем добавить код в `server.js` для импорта модуля:

```
var express = require("express"),
http = require("http"),
// импортируем библиотеку mongoose
mongoose = require("mongoose"),
app = express();
app.use(express.static(__dirname + "/client"));
app.use(express.urlencoded());
// подключаемся к хранилищу данных Amazeriffic в Mongo
mongoose.connect('mongodb://localhost/amazeriffic');
```

Затем определим схему и, следовательно, модель для списка задач в серверном коде:

```
// Это модель Mongoose для задач
var ToDoSchema = mongoose.Schema({
  description: String,
  tags: [ String ]
});
var ToDo = mongoose.model("ToDo", ToDoSchema);
// начинаем слушать запросы
http.createServer(app).listen(3000);
```

Мы можем обновить маршрут для получения задач из базы данных и возвращения туда:

```
app.get("/todos.json", function (req, res) {
  ToDo.find({}, function (err, toDos) {
    // не забудьте о проверке на ошибки!
    res.json(toDos);
  });
});
```

И наконец, нужно обновить маршрут post для добавления элементов в базу данных. Это немного интереснее, потому что для сохранения совместимости с клиентом, модифицированным в главе 2, придется возвращать весь список задач:

```
app.post("/todos", function (req, res) {
  console.log(req.body);
  var newToDo = new ToDo({"description":req.body.description,
  "tags":req.body.tags});
  newToDo.save(function (err, result) {
    if (err !== null) {
      console.log(err);
      res.send("ERROR");
    } else {
      // клиент ожидает, что будут возвращены все задачи,
      // поэтому для сохранения совместимости сделаем дополнительный запрос
      ToDo.find({}, function (err, result) {
        if (err !== null) {
          // элемент не был сохранен
          res.send("ERROR");
        }
        res.json(result);
      });
    }
  });
});
```

Сейчас мы можем запустить сервер и подключиться к нему через браузер. Помните, что мы еще не добавили в нашу коллекцию никаких задач, поэтому следует добавить несколько штук, прежде чем мы увидим в приложении какие-либо элементы.

Подведем итоги

В этой главе мы научились хранить данные независимо от приложения. Мы рассмотрели два решения для хранения данных, которые являются примерами баз данных формата не-SQL, — Redis и MongoDB.

Redis — хранилище типа «ключ — значение». Он предоставляет быструю и гибкую технологию для хранения простых данных. Модуль node-redis позволяет взаимодействовать с Redis почти точно так же, как мы делали это из командной строки.

MongoDB — более надежное решение для хранения данных, которое организует базы данных в коллекции. Как пользователи базы данных, мы можем думать о ней как о хранилище данных в виде объектов JavaScript. Кроме того, интерфейс командной строки для Mongo может быть использован как базовый интерпретатор JavaScript, позволяющий нам взаимодействовать с нашими коллекциями.

Mongoose — это модуль Node.js, который позволяет нам взаимодействовать с MongoDB, а кроме того, еще и создавать модели данных. Модели данных не только указывают, как должны выглядеть все элементы какой-либо коллекции, но и позволяют нам взаимодействовать с базой данных через объекты в программе.

Больше теории и практики

Покерное API

Если вы проработали все практические разделы в предыдущих главах, то построили покерное API, которое принимает покерную раздачу и возвращает ее комбинацию (то есть пару, фулл-хауз и др.). Было бы интересно добавить в ваш сервер Express код, который хранит результат для каждой раздачи и ее тип в Redis или Mongo. Другими словами, сделайте так, чтобы приложение продолжало отслеживать каждую корректную раздачу, размещенную в API, вместе с результатом. Не следует отслеживать элементы с ошибками.

Если вы сделаете это, то можете настроить маршрут `get`, который будет возвращать объект JSON, включающий в себя все сохраненные раздачи. Вы можете даже пожелать настроить `get` так, чтобы он возвращал только пять последних раздач.

Если же хотите сделать этот пример по-настоящему интересным, попробуйте найти библиотеку с изображениями игральных карт и заставьте ваш клиент показывать изображения карт в раздаче вместо объектов JSON или текстового описания. Чтобы сделать это, нужно будет создать подпапку `images` в клиентской части вашего приложения и хранить изображения там.

Другие источники информации о базах данных

Базы данных — очень обширная тема, материала которой хватит на несколько книг. Если вы хотите узнать больше о разработке веб-приложений и о программировании в целом, обязательно попытайтесь изучить основы реляционных баз данных и SQL.

Coursera — прекрасная возможность пройти бесплатные онлайн-классы по темам баз данных среди прочего. Например, там есть курс для самостоятельного обучения, составленный Дженнифер Видом из Стэнфордского университета!

Если же вы предпочитаете книги, Дженнифер Видом является соавтором (вместе с Гектором Гарсия-Молина и Джеффри Уллманом) книги «Системы баз данных: Полная книга» (*Database Systems: The Complete*, Prentice Hall, 2008). Она представляет довольно-таки академический подход к теме, но я думаю, что это великолепное, легко читаемое произведение.

Еще одна прекрасная книга, освещающая различные типы баз данных (включая не-SQL), — «Семь баз данных за семь недель» Эрика Рэдмонда и Джима Уилсона (*Seven Databases in Seven Weeks*, Pragmatic Bookshelf, 2012). Кроме Mongo и Redis, в книге описаны такие популярные варианты, как PostgreSQL и Riak. Авторы проделали большую работу по описанию плюсов и минусов каждой базы данных, а также разнообразных сценариев их использования.

8 Платформа

К этому моменту вы уже имеете представление о создании веб-приложения с использованием клиентских и серверных технологий, но приложение, очевидно, не будет широко использоваться, пока существует только на вашем компьютере. Следующий фрагмент головоломки — размещение приложения в Сети и запуск его в Интернете.

В прошлом для этого требовался огромный объем работы: нужно было оплатить место на сервере с фиксированным IP-адресом, установить и настроить необходимое ПО, оплатить доменное имя, а затем связать его со своим приложением. К счастью, времена изменились и в наши дни существует категория хостинговых сервисов под названием «платформа-сервис» (Platforms-as-a-Service — PaaS), которые могут выполнить за нас всю рутинную работу.

Вы, может быть, слышали термин *«облачные вычисления»*? Эта концепция основана на том, что низкоуровневые детали содержания программного обеспечения и вычислений могут и должны быть перенесены на локальные компьютеры и в Интернет. С PaaS нам не нужно беспокоиться о любых деталях обслуживания и настройки веб-сервера, так что мы можем сфокусироваться только на обеспечении корректной работы своего веб-приложения. Другими словами, PaaS — это тип технологии, обеспечивающий работу облачной вычислительной модели.

В этой главе мы изучим, как загрузить свои веб-приложения на открытый ресурс PaaS, который называется Cloud Foundry. Хотя мы сконцентрируемся на Cloud Foundry, большинство изученных нами концепций будут справедливы и для других сервисов PaaS, например Heroku или Nodejitsu.

Cloud Foundry

Cloud Foundry — это открытый ресурс PaaS, изначально разработанный VMWare. Вы можете прочитать больше об этом сервисе на его домашней странице, расположенной по адресу http://cloudfoundry.com (рис. 8.1). Ресурс предлагает бесплатное использование на 60 дней, в течение которых вы можете попробовать загрузить туда некоторые из примеров-приложений из этой книги.

Рис. 8.1. Домашняя страница Cloud Foundry[1]

Регистрация

Чтобы начать, вам нужно создать учетную запись на http://www.cloudfoundry.com. Вы можете получить 60 дней бесплатного использования, щелкнув на ссылке в верхнем правом углу и набрав свой адрес электронной почты. В ответ получите информацию о том, как настроить учетную запись.

Подготовка приложений к развертыванию в Сети

Чтобы развернуть приложение, вам понадобится инструмент, называемый cf. Инсталляция его на локальную машину требует предварительной установки языка Ruby и пакет-менеджера RubyGems. Я включил приложение cf в проект node-dev-bootstrap, так что вы можете начать развертывание прямо с гостевой машины без всяких дополнительных установок!

Чтобы начать, создайте папку Chapter8 клонированием проекта node-dev-bootstrap, как вы уже делали в предыдущих двух главах. Покончив с этим, запустите гостевую машину и подключитесь к ней по SSH.

---

[1] На момент перевода с адреса, указанного в предыдущем абзаце, идет перенаправление на тот, что вы видите на скриншоте в URL. Однако этот сервис предлагает примерно те же функции, что и Cloud Foundry, поэтому вы можете и далее пользоваться инструкциями из книги. — *Примеч. пер.*

Начнем с развертывания базовой программы server.js, которая появилась у нас вместе с node-dev-bootstrap, но перед этим нам придется проделать еще две вещи. Вначале понадобится добавить файл `package.json`, так как Cloud Foundry ожидает найти его во всех приложениях, которые мы будем разворачивать. Поскольку наше серверное приложение по умолчанию не зависит от каких-либо внешних модулей Node.js, файл будет очень коротким:

```
{
  "name":"sp_example",
  "description": "Мое первое приложение Cloud Foundry!"
}
```

Затем нужно сделать небольшое изменение в файле по умолчанию `server.js`, который поставляется вместе с проектом node-dev-bootstrap. Это изменение затрагивает номер порта, по которому производится слушание сервера: по техническим причинам Cloud Foundry сам должен назначить нам номер порта для слушания, но мы не будем знать этот номер до запуска кода. К счастью, Cloud Foundry доставляет номер порта через *переменную окружения*, которая называется `PORT`. Мы можем получить к ней доступ через переменную `process.env.PORT` в программе Node.js:

```
var http = require("http"),
// если переменная окружения PORT настроена,
// слушать по ней, если же нет, слушать
// по порту 3000
port = process.env.PORT || 3000;
var server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Привет от Cloud Foundry!");
});
server.listen(port);
console.log("Сервер слушает по порту " + port);
```

Здесь мы снова встречаем идиому `||`, с которой уже столкнулись в главе 7. А вообще этот код делает следующее: если `process.env.port` определена, использовать ее, если нет, слушать по порту 3000. Это позволяет настроить номер порта так, что приложение будет корректно работать, будучи как размещенным на гостевой машине, так и развернутым на Cloud Foundry.

Развертывание приложения

К этому моменту мы уже можем запустить виртуальную машину и подключиться к ней через браузер, как и раньше. А сейчас, создав файл `package.json` и настроив приложение на слушание по корректному порту, мы также готовы загрузить его на Cloud Foundry.

Как я упоминал ранее, это потребует использования программы cf, которая уже установлена на гостевой машине. Чтобы начать, перейдите на гостевой машине в папку, где находится файл `server.js`.

API платформы Cloud Foundry располагается на `api.run.pivotal.io`. Поэтому начнем с указания cf, где находится целевая платформа Cloud Foundr, с помощью субкоманды `target`:

```
vagrant $ cf target api.run.pivotal.io
Setting target to https://api.run.pivotal.io... OK
```

Затем нужно авторизоваться с помощью команды `login` и ввести данные для входа в учетную запись Cloud Foundry. После авторизации cf спросит, какое пространство для развертывания мы хотим использовать. Я обычно использую пространство `development`, когда экспериментирую:

```
vagrant $ cf login
target: https://api.run.pivotal.io
Email> me@semmy.me
Password> ************
Authenticating... OK
1: development
2: production
3: staging
Space> 1
Switching to space development... OK
```

Если все сделано правильно, следующий шаг — отправка приложения на Cloud Foundry! Как это сделать? Это так же просто, как использовать команду `push`. Вместе с субкомандой `push` нужно включить команду для запуска приложения:

```
vagrant $ cf push --command "node server.js"
```

Субкоманда `push` отправляет нас к диалогу с Cloud Foundry. Нужно будет ответить на несколько вопросов о приложении и об окружении, которое мы будем использовать. Вот диалог, который я провел с Cloud Foundry, — ваш, скорее всего, будет очень похожим:

```
Name> sp_example
Instances> 1
1: 128M
2: 256M
3: 512M
4: 1G
Memory Limit> 256M
Creating sp_example... OK
1: sp_example
2: none
Subdomain> sp_example
1: cfapps.io
2: none
Domain> cfapps.io
Creating route sp_example.cfapps.io... OK
Binding sp_example.cfapps.io to sp_example... OK
Create services for application?> n
Save configuration?> n
Uploading sp_example... OK
```

```
Preparing to start sp_example... OK
-----> Downloaded app package (4.0K)
-----> Resolving engine versions
       WARNING: No version of Node.js specified in package.json, see:
       https://devcenter.heroku.com/articles/nodejs-versions
       Using Node.js version: 0.10.21
       Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
       npm WARN package.json sp_example@ No repository field.
       npm WARN package.json sp_example@ No readme data.
       Dependencies installed
-----> Building runtime environment
-----> Uploading droplet (15M)
Checking status of app 'sp_example'...
1 of 1 instances running (1 running)
Push successful! App 'sp_example' available at sp_example.cfapps.io
```

Имя вашего приложения должно быть уникальным среди всех имен в этом домене. Это значит, что, если вы выберете имя наподобие example, скорее всего, получите сообщение о невозможности развертывания. Я стараюсь избегать этой проблемы, добавляя мои инициалы и знак подчеркивания в начале имени. Это не всегда работает, поэтому попробуйте найти собственные способы создания уникальных имен для приложений.

Если все прошло хорошо, ваше приложение сейчас запущено в Веб! Можете убедиться в этом, открыв браузер и перейдя по URL, который выдала вам cf (в моем примере это http://sp_example.cfapps.io). Сделав это, вы должны увидеть ответ от вашего приложения.

Получение информации о приложениях

Сейчас, когда ваше приложение загружено и работает, можете использовать другие субкоманды cf для получения информации о статусе приложений. Например, можете использовать субкоманду `apps` для получения списка ваших приложений, находящихся на Cloud Foundry, и их статусов:

```
vagrant $ cf apps
name       status    usage     url
sp_example running   1 x 256M  sp_example.cfapps.io
```

Одна из главных проблем запуска приложений на PaaS — то, что вы не можете видеть в консоли результаты работы операторов `console.log` так же просто, как это было при запуске приложений локально. Это может быть серьезной проблемой, если ваша программа сбоит и надо понять почему. К счастью, Cloud Foundry предоставляет субкоманду `logs`, которую вы можете использовать для запущенных приложений, чтобы просмотреть программные логи:

```
vagrant $ cf logs sp_example
Getting logs for sp_example #0... OK
Reading logs/env.log... OK
TMPDIR=/home/vcap/tmp
VCAP_APP_PORT=61749
USER=vcap
VCAP_APPLICATION= { ... }
PATH=/home/vcap/app/bin:/home/vcap/app/node_modules/.bin:/bin:/usr/bin
PWD=/home/vcap
VCAP_SERVICES={}
SHLVL=1
HOME=/home/vcap/app
PORT=61749
VCAP_APP_HOST=0.0.0.0
MEMORY_LIMIT=256m
_=/usr/bin/env
Reading logs/staging_task.log... OK
-----> Downloaded app package (4.0K)
-----> Resolving engine versions
       WARNING: No version of Node.js specified in package.json, see:
       https://devcenter.heroku.com/articles/nodejs-versions
       Using Node.js version: 0.10.21
       Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
       npm WARN package.json sp_example@ No repository field.
       npm WARN package.json sp_example@ No readme data.
       Dependencies installed
-----> Building runtime environment
Reading logs/stderr.log... OK
Reading logs/stdout.log... OK
Server listening on port 61749
```

Вы видите, что Cloud Foundry выводит содержимое четырех сохраненных логов. Первый — `env.log`, где находятся все переменные окружения, к которым вы можете получить доступ через переменную `process.env` в программе. Второй — `staging_task.log`, в который записывается все, что происходит во время первого запуска программы (вы увидите, что там записано то же самое, что было выведено на экран, когда вы в первый раз запустили `cf push`). Последние два — `stderr.log` и `stdout.log`. Вы увидите, что `stdout.log` включает в себя оператор `console.log`, который вы использовали в своей программе. Если вместо него был указан `console.err`, сообщение появится в `stderr.log`.

Обновление приложения

Вы легко можете послать новейшую версию приложения в Cloud Foundry, развернув его заново. Модифицируем `server.json` так, чтобы он возвращал несколько больше информации:

```
var http = require("http"),
    port = process.env.PORT || 3000;
var server = http.createServer(function (req, res) {
    res.writeHead(200, {"Content-Type": "text/plain"});
    res.write("Сервер слушает по порту " + port);
    res.end("Привет от Cloud Foundry!");
});
server.listen(port);
console.log("Сервер слушает по порту " + port);
```

Как только вы внесете это изменение, можете запустить субкоманду `push` заново вместе с именем приложения, которое хотите обновить. Соответствующие изменения будут внесены без необходимости заново отвечать на все вопросы:

```
vagrant $ cf push sp_example
Save configuration?> n
Uploading sp_example... OK
Stopping sp_example... OK
Preparing to start sp_example... OK
-----> Downloaded app package (4.0K)
-----> Downloaded app buildpack cache (4.0K)
-----> Resolving engine versions
       WARNING: No version of Node.js specified in package.json, see:
       https://devcenter.heroku.com/articles/nodejs-versions
       Using Node.js version: 0.10.21
       Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
       npm WARN package.json sp_example@ No repository field.
       npm WARN package.json sp_example@ No readme data.
       Dependencies installed
-----> Building runtime environment
-----> Uploading droplet (15M)
Checking status of app 'sp_example'...
1 of 1 instances running (1 running)
Push successful! App 'sp_example' available at sp_example.cfapps.io
```

Удаление приложений из Cloud Foundry

Иногда, оказывается, нужно удалить приложения из Cloud Foundry, особенно если мы просто экспериментируем. Чтобы сделать это, можно использовать субкоманду `delete`:

```
vagrant $ cf delete sp_example
Really delete sp_example?> y
Deleting sp_example... OK
```

Взаимозависимости и package.json

В предыдущем примере наши приложения имели внешние взаимозависимости наподобие модулей express, redis, mongoose и ntwitter. Использование базовых модулей, которые не подключаются к внешним сервисам (например, Express или Twitter), — весьма незамысловатое решение. Поскольку обычно мы не отправляем в облачное хранилище каталог node_modules, нужно просто удостовериться в том, что все взаимозависимости перечислены в файле package.json.

Например, вспомните одно из наших первых приложений Express. После выполнения небольших модификаций оно начинает слушание по корректному порту Cloud Foundry, что выглядит следующим образом:

```
var express = require("express"),
    http = require("http"),
    app = express(),
    port = process.env.PORT || 3000;;
http.createServer(app).listen(port);
console.log("Express is listening on port " + port);
app.get("/hello", function (req, res) {
  res.send("Hello, World!");
});
app.get("/goodbye", function (req, res) {
  res.send("Goodbye World!");
});
```

Нужно будет включить взаимозависимость с модулем Express в файл package.json, что мы и сделали в предыдущем примере с package.json:

```
{
  "name": "sp_express",
  "description": "a sample Express app",
  "dependencies": {
    "express": "3.4.x"
  }
}
```

Как видите, я указал, что приложение зависит от модуля Express, в частности от всех версий, номер которых начинается с 3.4 (x — подстановочный символ). Таким образом, CLoud Foundry понимает, какую версию нужно установить, чтобы приложение работало корректно. Поскольку взаимозависимости включены в файл package.json, мы можем отправить его в Cloud Foundry с помощью той же самой команды, что и раньше:

```
vagrant $ ~/app$ cf push --command "node server.js"
Name> sp_expressexample
```

Сделав это, мы можем зайти на http://sp_expressexample.cfapps.io/hello или http://sp_expressexample.cfapps.io/goodbye, чтобы увидеть ответ приложения!

Но заставить работать наши приложения Twitter или Amazeriffic несколько сложнее, так как они зависят от других хранилищ данных — в частности, Redis или MongoDB. Это значит, что нужно создать сервисы, а затем заставить приложения их использовать.

Привязка Redis к приложению

Когда мы запускаем приложение на виртуальной машине, такие сервисы, как Redis или MongoDB, работают локально. Но это будет не так, если мы запустим приложение из PaaS. Иногда сервисы работают на том же самом хосте, но в некоторых случаях могут запускаться и на другом.

В любом случае начать следует с настройки сервиса, который требуется запустить при взаимодействии с cf. В этой секции настроим Redis в Cloud Foundry, а затем подключим к нему счетчик твитов.

Начнем с копирования приложения Twitter из главы 7 в папку Chapter8. Убедитесь в том, что файл `Package.json` существует и включает взаимозависимости `ntwitter` и `redis`. Мой выглядит вот так:

```
{
  "name": "tweet_counter",
  "description": "tweet counter example for learning web app development",
  "dependencies": {
    "ntwitter":"0.5.x",
    "redis":"0.9.x"
  }
}
```

Нужно также обновить файл `server.js` так, чтобы сервер слушал по порту, указанному в `process.env.PORT`. Сделав это, мы можем попробовать развернуть приложение! Я так и сделаю (забегая вперед, скажу, что попытка будет неудачной, но даст возможность развернуть сервис Redis):

```
vagrant $ cf push --command "node server.js"
Name> sp_tweetcounter
Instances> 1
1: 128M
2: 256M
3: 512M
4: 1G
Memory Limit> 256M
Creating sp_tweetcounter... OK
1: sp_tweetcounter
2: none
Subdomain> sp_tweetcounter
1: cfapps.io
2: none
Domain> cfapps.io
Binding sp_tweetcounter.cfapps.io to sp_tweetcounter... OK
Create services for application?> y
1: blazemeter n/a, via blazemeter
2: cleardb n/a, via cleardb
3: cloudamqp n/a, via cloudamqp
4: elephantsql n/a, via elephantsql
5: loadimpact n/a, via loadimpact
```

```
6: mongolab n/a, via mongolab
7: newrelic n/a, via newrelic
8: rediscloud n/a, via garantiadata
9: sendgrid n/a, via sendgrid
10: treasuredata n/a, via treasuredata
11: user-provided , via
```

Обратите внимание: мы сказали Cloud Foundry, что хотим создать сервис для приложения:

```
What kind?> 8
Name?> rediscloud-dfc38
1: 20mb: Lifetime Free Tier
Which plan?> 1
Creating service rediscloud-dfc38... OK
Binding rediscloud-dfc38 to sp_tweetcounter... OK
Create another service?> n
Bind other services to application?> n
Save configuration?> n
```

Закончив диалог, вы обнаружите, что попытка развертывания не удалась. Так случилось потому, что мы еще не указали приложению, как подключиться к внешнему сервису Redis. Логи должны дать нам подсказку об этом. И в самом деле, запустив `cf logs sp_tweetcounter`, я увижу что-то похожее в моем `logs/stderr.log`:

```
Reading logs/stderr.log... OK
events.js:72
    throw er; // Unhandled 'error' event
          ^
Error: Redis connection to 127.0.0.1:6379 failed – connect ECONNREFUSED
  at RedisClient.on_error (/home/vcap/app/node_modules/redis/index.js:189:24)
  at Socket.<anonymous> (/home/vcap/app/node_modules/redis/index.js:95:14)
  at Socket.EventEmitter.emit (events.js:95:17)
  at net.js:441:14
  at process._tickCallback (node.js:415:13)
```

В наших приложениях мы не отправляем никаких аргументов Redis, поэтому она пытается подключиться к локальному серверу (127.0.0.1), используя порт по умолчанию (6379). Нужно дать указание подключаться к облачному сервису Redis по хосту и порту, которые предоставляет Cloud Foundry. Где же найти эту информацию? Как и номер порта, все о связях сервисов указано в переменной `process.env`!

Если вы уже ввели `cf logs sp tweetcounter`, то можете прокрутить окно вниз, чтобы просмотреть переменные окружения и содержание переменной `VCAP_SERVICES`. В логах она представлена в виде одной длинной строки JSON, поэтому здесь я привел ее к более удобочитаемому виду:

```
VCAP_SERVICES = {
  "rediscloud-n/a": [{
    "name": "rediscloud-dfc38",
    "label": "rediscloud-n/a",
```

```
    "tags": ["redis", "key-value"],
    "plan": "20mb",
    "credentials": {
      "port": "18496",
      "hostname": "pub-redis-18496.us-east-1-4.2.ec2.garantiadata.com",
      "password": "REDACTED"
    }
  }]
}
```

Здесь находится вся информация, необходимая для подключения к удаленному сервису Redis, включая URL, порт и пароль. Можно было бы напрямую прописать данные для входа в Redis в нашей программе, но, скорее всего, лучше будет сделать это программным способом, аналогично тому как мы установили номер порта.

Есть небольшое затруднение — переменная окружения VCAP_SERVICES хранится в виде строки, поэтому нужно использовать команду JSON.parse так же, как в главе 5, чтобы конвертировать переменную в объект, с которым мы можем работать как с любым другим объектом JavaScript.

Если говорить коротко, мы можем заставить Redis работать, модифицировав первую часть модуля tweet_counter.js следующим образом:

```
var ntwitter = require("ntwitter"),
    redis = require("redis"), // требуется модуль Redis
    credentials = require("./credentials.json"),
    redisClient,
    counts = {},
    twitter,
    services,
    redisCredentials;
// создание клиента Twitter
twitter = ntwitter(credentials);
// настройка сервисов, если существует переменная окружения
if (process.env.VCAP_SERVICES) {
  // анализ строки JSON
  services = JSON.parse(process.env.VCAP_SERVICES);
  redisCredentials = services["rediscloud-n/a"][0].credentials;
} else {
  // в ином случае мы используем значения по умолчанию
  redisCredentials = {
    "hostname": "127.0.0.1",
    "port": "6379",
    "password": null
  };
}
// создаем клиент для подключения к Redis
client = redis.createClient(redisCredentials.port, redisCredentials.hostname);
// аутентификация
client.auth(redisCredentials.password);
```

В этом фрагменте кода мы проверяем, существует ли переменная окружения VCAP_SERVICES, и, если да, анализируем связанную с ней строку, чтобы сгенерировать объект JavaScript из представления JSON. Затем отправляем данные для входа, связанные со свойством rediscloud-n/a в объекте services. Сам по себе этот объект является массивом (так как наше приложение должно быть связано с несколькими экземплярами Redis), поэтому нужно получить первый элемент массива.

Если переменная окружения VCAP_SERVICES не определена, мы настраиваем объект redisCredentials, куда заложены значения по умолчанию. После этого подключаемся к Redis, указав порт и имя хоста, а затем отправляем пароль. Если мы подключены к Redis локально на виртуальной машине, в качестве пароля отправляется null, так как аутентификации в данном случае не требуется.

Если предыдущая попытка развернуть приложение не удалась, мы можем предпринять новую. В этот раз я использую предварительно заданное имя. Если забуду его, то всегда могу использовать команду cf apps, чтобы увидеть список моих приложений:

```
vagrant $ cf apps
Getting applications in development... OK
name            status   usage     url
sp_example      running  1 x 256M  sp_example.cfapps.io
sp_express      running  1 x 256M  sp_express.cfapps.io
sp_tweetcounter running  1 x 256M  sp_tweetcounter.cfapps.io
vagrant $ cf push sp_tweetcounter
```

Если все настроено правильно, то вы увидите, что приложение успешно развернуто, и можете получить доступ к нему через URL, который получите от Cloud Foundry.

Привязка MongoDB к приложению

Привязка MongoDB к приложению и ее запуск в Cloud Foundry почти так же просты, как те же операции с Redis. Начнем с копирования приложения Amazeriffic из главы 7 в текущую папку. Придется сделать несколько небольших изменений по уже знакомому образцу.

Итак, начнем с указания слушать по порту process.env.PORT, если эта величина существует. Код для этого идентичен приведенному в примерах в предыдущем разделе.

После этого нужно получить данные входа из process.env.VCAP_SERVICES. Код будет очень похожим на код для Redis. Основное отличие состоит в том, что данные для входа в MongoDB содержатся в единичной строке — uri:

```
// не забудьте объявить переменную mongoUrl где-нибудь выше
// настройка наших сервисов
if (process.env.VCAP_SERVICES) {
  services = JSON.parse(process.env.VCAP_SERVICES);
  mongoUrl = services["mongolab-n/a"][0].credentials.uri;
} else {
```

```
  // нам понадобится эта часть для запуска не из Cloud Foundry
  mongoUrl = "mongodb://localhost/amazeriffic"
}
```

После того как все это заработает, можем развернуть приложение в Cloud Foundry точно так же, как в предыдущем примере. Одно незначительное изменение вызвано необходимостью настроить сервис MongoLab:

```
Create services for application?> y
1: blazemeter n/a, via blazemeter
2: cleardb n/a, via cleardb
3: cloudamqp n/a, via cloudamqp
4: elephantsql n/a, via elephantsql
5: loadimpact n/a, via loadimpact
6: mongolab n/a, via mongolab
7: newrelic n/a, via newrelic
8: rediscloud n/a, via garantiadata
9: sendgrid n/a, via sendgrid
10: treasuredata n/a, via treasuredata
11: user-provided, via
What kind?> 6
Name?> mongolab-8a0f4
1: sandbox: 0.5 GB
Which plan?> 1
Creating service mongolab-8a0f4... OK
```

После того как сервис Mongo создан, просто связываем его с нашим приложением так же, как делали это с Redis. Как только код будет готов к связи с удаленным сервером через URL, который предоставляется нам в переменной окружения `VCAP_SERVICES`, программа должна будет работать именно так, как мы ожидаем.

Подведем итоги

В этой главе мы изучили, как использовать Cloud Foundry, чтобы выложить приложения в Интернете. Cloud Foundry является примером платформы-сервиса (Platform as a Service). PaaS — это компьютерная технология, которая абстрагирует настройку сервера, администрацию и хостинг, обычно с помощью программы с интерфейсом командной строки или веб-интерфейсом.

Чтобы заставить наши приложения работать из Cloud Foundry (или любого другого PaaS), как правило, нужно внести небольшие модификации в код, так как его поведение различно в зависимости от того, запускается ли он локально из рабочего окружения или на серверах Cloud Foundry.

Cloud Foundry представляет также внешние сервисы, такие как Redis или MongoDB. Чтобы наши приложения работали с этими сервисами, необходимо сначала создать сервис с помощью программы cf, а затем привязать к нему наше приложение.

Больше теории и практики

Покерное API

В разделах с задачами в предыдущих главах мы создали простое покерное API, которое должно идентифицировать покерные раздачи. В последней главе мы добавили к нему компонент базы данных. Если у вас все получилось и приложение работает, попробуйте модифицировать его и развернуть на Cloud Foundry.

Другие платформы

В наши дни нет недостатка в облачных платформах. Одна из самых популярных называется Heroku. Она позволяет зарегистрироваться бесплатно без использования кредитной карты. Если вы хотите добавить использование Redis или Mongo, то карта все же понадобится, несмотря на то что у них есть варианты бесплатного использования обоих этих сервисов.

Если вы хотите больше практики, попробуйте почитать документацию Heroku по развертыванию с Node.js, а затем загрузить одно или несколько предложений на этот сервис. Я бы также рекомендовал вам попробовать и другие сервисы, в том числе Nodejitsu или Microsoft's Windows Azure. Все они немного отличаются друг от друга, так что сравнение их функциональности будет отличным практическим упражнением.

9 Приложение

Итак, если вы продвинулись по пути так далеко, то наверняка уже можете работать и с клиентской, и с серверной стороной приложения, используя HTML, CSS и JavaScript. А если у вас есть идея для веб-приложения, то, скорее всего, вы можете использовать изученное и написать код, который будет делать то, что вам нужно.

Эта глава в основном о том, почему вам не надо этого делать, по крайней мере прямо сейчас. Дэвид Парнас сказал однажды, что «один плохой программист создает две новые вакансии в год». Хотя лично я полностью с этим согласен, на самом деле я не думаю, что «плохие» программисты в самом деле плохие, просто у них пока недостаточно опыта и практики, чтобы позаботиться о хорошей организации своей базы кода и пригодности ее к дальнейшему развитию. Код (особенно JavaScript) в руках неопытного программиста может очень быстро стать монолитной массой, непригодной к работе.

Как я упоминал в предисловии, будучи новичком, вполне нормально «взламывать» код, пока вы учитесь, но с опытом вы быстро поймете, что это не лучший способ решения более сложных и серьезных инженерных проблем. Другими словами, вы еще многому должны научиться, и так будет всегда.

В то же время разработчики программного обеспечения создавали веб-приложения, основанные на базах данных, много лет, в результате чего было неизбежно появление определенных шаблонов оптимизации кода. Фактически целые фреймворки для разработки, такие как Ruby on Rails, предназначены для того, чтобы заставить разработчиков создавать приложения с использованием этих шаблонов (или, во всяком случае, настойчиво рекомендовать им это). Поскольку наш путь по основам подходит к концу, мы вернемся к Amazeriffic и постараемся понять, как можно сделать имеющийся у нас сейчас код более гибким. Мы также увидим, как некоторая часть кода может быть модифицирована, чтобы соответствовать этим шаблонам.

Переработка клиента

Начнем с клиентской части. По моему опыту, клиент — одна из частей, которую легче всего быстро сделать непригодной к развитию. Думаю, это может быть связано с тем, что код клиентской стороны часто ассоциируется с обманчиво простыми визуальными действиями, в результате чего *кажется*, что и реализовать его

очень просто. Например, реализация специфического поведения, когда пользователь выполняет тривиальное действие, допустим нажимает кнопку, может показаться простейшей задачей кодирования. Но это не всегда так.

В нашем примере создание интерфейса с вкладками кажется довольно прямолинейным действием. И в данном конкретном случае это действительно так. Но мы можем сделать его значительно более пригодным к дальнейшему развитию, слегка переработав и следуя основным советам.

Обобщение основных принципов действия

Вы всегда должны стараться как можно сильнее обобщать связанные понятия. Я уже несколько раз упоминал в этой книге, что если какие-либо сущности в вашей программе логически соотносятся друг с другом, лучше всего, если они будут определенным образом связаны через реальные программные конструкции. Но наши вкладки не соответствуют этому принципу: у них много общего, но логика и структура разбросаны в нескольких местах. Например, мы определяем название вкладки в HTML:

```
<div class="tabs">
  <a href=""><span class="active">Новые</span></a>
  <a href=""><span>Старые</span></a>
  <a href=""><span>Теги</span></a>
  <a href=""><span>Добавить</span></a>
</div>
```

Затем в JavaScript используем расположение вкладки в DOM, чтобы определить, какое действие следует предпринять:

```
if ($element.parent().is(":nth-child(1)")) {
  // генерация содержимого вкладки "Новые"
} else if ($element.parent().is(":nth-child(2)")) {
  // генерация содержимого вкладки "Старые"
} else if ($element.parent().is(":nth-child(3)")) {
  // генерация содержимого вкладки "Теги"
} else if ($element.parent().is(":nth-child(4)")) {
  // генерация содержимого вкладки "Добавить"
}
```

Обратите внимание на то, что нигде в этих действиях название вкладки не используется. Вот пример двух программных сущностей, которые соотносятся друг с другом, но только в голове программиста и нигде в конструкции кода. Самый первый симптом непорядка — каждый раз, когда мы хотим добавить в пользовательский интерфейс новую вкладку, нужно перерабатывать как HTML, так и JavaScript. В главе 5 я упоминал, что из-за этого повышается вероятность возникновения ошибок в нашем коде.

Как же можно это исправить? Один из способов — обобщить вкладку в виде объекта, точно так же, как мы поступили в примере с картами в главе 5. Разница состоит в том, что объект вкладки будет иметь строковое значение, связанное

с названием вкладки, а также функциональное значение, связанное с действием, создающим содержимое. Например, мы можем обобщить вкладку Новые в объект такого вида:

```
var newestTab = {
  // название новой вкладки
  "name":"Новые",
  // функция, создающая содержимое вкладки
  "content": function () {
    var $content;
    // генерация и отображение содержимого вкладки Новые
    $content = $("<ul>");
    for (i = toDos.length-1; i >= 0; i--) {
      $content.append($("<li>").text(toDos[i]));
    }
    return $content;
  }
}
```

Оказывается, наличие этой структуры решает множество проблем. Наш пользовательский инструмент содержит набор вкладок, и вместо того, чтобы хранить имя вкладки и соответствующее действие в разных местах кода, мы можем создать массив вкладочных объектов, который будет держать их вместе:

```
var main = function (toDoObjects) {
  "use strict";
  var toDos,
  tabs;
  toDos = toDoObjects.map(function (toDo) {
    return toDo.description;
  });
  // создание пустого массива с вкладками
  tabs = [];
  // добавляем вкладку Новые
  tabs.push({
    "name":"Новые",
    "content":function () {
      // создание $content для Новые
      return $content;
    }
  });
  // добавляем вкладку Старые
  tabs.push({
    "name":"Старые",
    "content":function () {
      // создание $content для Старые
      return $content;
    }
  });
  // добавляем вкладку Теги
  tabs.push({
```

```
    "name":"Теги",
    "content":function () {
      // создание $content для Теги
      return $content;
    }
  });
  // Создаем вкладку Добавить
  tabs.push({
    "name":"Добавить",
    "content":function () {
      // создание $content для Добавить
      return $content;
    }
  });
};
```

Имея массив, где содержатся все вкладочные объекты, мы можем многократно упростить подход к построению пользовательского интерфейса. Начнем с удаления вкладок из HTML `<div class="tabs">`:

```
<!-- здесь определялись вкладки -->
<!-- а сейчас мы динамически создадим их с помощью JavaScript -->
</div>вообще (но оставляем элемент div):
```

А сейчас нужно циклически обработать массив `tabs`. Для каждой вкладки мы сделаем обработчик щелчков и добавление в DOM:

```
tabs.forEach(function (tab) {
  var $aElement = $("<a>").attr("href","");
  $spanElement = $("<span>").text(tab.name);
  $aElement.append($spanElement);
  $spanElement.on("click", function () {
    var $content;
    $(".tabs a span").removeClass("active");
    $spanElement.addClass("active");
    $("main .content").empty();
    // здесь мы получаем содержимое из функции,
    // определенной в объекте tab
    $content = tab.content();
    $("main .content").append($content);
    return false;
  });
});
```

Введение AJAX для работы с вкладками

Другая проблема приложения заключается в том, что, если кто-либо посетит его из другого браузера и добавит какую-нибудь задачу в наш список, мы не увидим этот новый элемент, если щелкнем на другой вкладке. Для этого нужно будет перезагрузить всю страницу.

Это происходит потому, что наше приложение хранит и открывает элементы списка задач, когда страница загружается впервые, и не изменяет их до того, как загрузка не произойдет снова. Чтобы решить эту проблему, можно сделать так, чтобы каждая вкладка выполняла запрос AJAX при открытии. Мы можем достичь этого, переместив действие в вызовы функции jQuery `get`:

```
tabs.push({
  "name":"Новые",
  "content":function () {
    $.get("todos.json", function (toDoObjects) {
      // создание $content для вкладки Новые
      // 'return' $content больше не требуется
    });
  }
});
```

Другое решение — можно заставить программу выполнять обновления клиента в реальном времени, то есть добиться того, чтобы, когда другой пользователь обновляет страницу, на которую мы смотрим, сервер выдавал новые данные. Это несколько выходит за рамки данной книги, но реализовать такое решение можно с использованием модуля Node.js, который называется socket.io.

Обратите внимание на то, что таким образом меняется поведение функций, так как теперь мы собираемся делать асинхронный запрос внутри вызова функции. Это значит, что код теперь не будет корректно возвращать содержимое при вызове, так как мы должны дождаться завершения вызова AJAX:

```
$content = tab.content();
$("main .content").append($content);
```

Это можно исправить несколькими способами. Самый простой — перенести обновление DOM внутрь самой функции `content`:

```
tabs.push({
  "name":"Новые",
  "content":function () {
    $.get("todos.json", function (toDoObjects) {
      // создание содержимого вкладки Новые $content
      // обновление DOM здесь
      $("main .content").append($content);
    });
  }
});
```

Но это решение мне не нравится по двум причинам. Первая из них скорее эстетическая: функция `content` должна создавать и возвращать содержимое вкладки, а не влиять на DOM. Иначе мы должны переименовать ее в `getContentAndUpdateTheDOM`.

Другая причина немного важнее: если в итоге мы хотим добиться чего-то большего, а не просто обновления DOM, нужно по такому же принципу изменить каждую функцию content для каждой вкладки.

Этих двух зайцев можно убить одним выстрелом: реализацией протяженного подхода, который мы использовали ранее для асинхронных операций. Пусть вызывающая функция включает в себя и обратный вызов, а обращаться к ней мы будем внутри функции content:

```
// создаем функцию content
// так, что она принимает обратный вызов
tabs.push({
  "name":"Новые",
  "content":function (callback) {
    $.get("todos.json", function (toDoObjects) {
      // создаем $content для Новые
      // создаем обратный вызов с $content
      callback($content);
    });
  }
});
// ...
// а сейчас отправляем обратный вызов внутри вызывающей функции
tab.content(function ($content) {
  $("main .content").append($content);
});
```

Это, наверное, самый известный способ решения, который вы можете найти в сообществе Node.js, но и другие тоже набирают популярность, например Promises (Потенциалы) и Reactive JavaScript (Реактивный JavaScript). Если асинхронные операции становятся сложнее и вы обнаруживаете себя в *омуте обратного вызова*[1] (как часто называют эту ситуацию), то лучше всего хорошенько изучить эти варианты решения.

Избавление от костылей совместимости

Сейчас, когда вкладки создаются с помощью AJAX, мы можем избавиться от болтающихся повсюду костылей совместимости. Раньше серверу приходилось возвращать список задач ToDo целиком, так как именно этого ожидал клиент, независимо от времени добавления. Сейчас вместо этого мы будем переключаться на вкладку Новые при добавлении какого-либо объекта ToDo.

Кнопка для добавления кода в настоящее время выглядит вот так:

```
$button.on("click", function () {
  var description = $input.val(),
  tags = $tagInput.val().split(","),
```

[1] В оригинале callback hell, то есть буквально «ад обратного вызова». Но омут как нечто крутящееся для данной ситуации подходит больше, а черти есть и тут, и там. — *Примеч. пер.*

```
  newToDo = {"description":description, "tags":tags};
  $.post("todos", newToDo, function (result) {
    // это остатки всякого хлама после
    // хранения списка задач на клиенте
    toDoObjects = result;
    // обновление клиентского списка задач
    toDos = toDoObjects.map(function (toDo) {
      return toDo.description;
    });
    $input.val("");
    $tagInput.val("");
  });
});
```

В действительности больше нет необходимости собирать список задач на клиенте, поэтому мы можем избавиться от большинства ненужных строк. Фактически, все, что нам на самом деле нужно проделать при успешном добавлении новой задачи, — это очистить все поля ввода, а затем *перейти* к вкладке Новые. Это и сделает запрос AJAX, расставив результаты, начиная с новейшего объекта.

Мы можем использовать функцию jQuery `trigger` для обработки щелчков на вкладке Новые, после чего код в конце концов примет такой вид:

```
$button.on("click", function () {
  var description = $input.val(),
  tags = $tagInput.val().split(","),
  newToDo = {"description":description, "tags":tags};
  $.post("todos", newToDo, function (result) {
    // очистка полей ввода
    $input.val("");
    $tagInput.val("");
    // щелчок на вкладке Новые
    $(".tabs a:first span").trigger("click");
  });
});
```

Небольшое изменение позволит нам вдобавок значительно упростить код на серверной стороне, который сохраняет новый объект ToDo:

```
app.post("/todos", function (req, res) {
  var newToDo = new ToDo({"description":req.body.description,
  "tags":req.body.tags});
  newToDo.save(function (err, result) {
    console.log(result);
    if (err !== null) {
      // этот элемент не сохранен!
      console.log(err);
      res.json(err);
    } else {
      res.json(result);
    }
  });
};
```

Обработка ошибок AJAX

В основном мы с вами игнорировали проблему обработки ошибок в коде, но как только вы начнете писать крупномасштабные приложения, то обнаружите, что заботиться об ошибках необходимо. Будут возникать, например, моменты, когда ваш клиентский код загрузился, но сервер оказался недоступным (или внезапно вышел из строя). Что тогда?

В большинстве случаев обстоятельства возникновения ошибки будут означать, что обратные вызовы не могут быть обработаны корректно. Внешне приложение будет выглядеть зависшим. Хуже, когда код станет реагировать на добавление задачи в клиент, но элемент не будет попадать в хранилище данных. Это может привести к потере данных пользователя, в то время как они будут думать, что данные успешно добавлены.

К счастью, jQuery позволяет легко обработать этот сценарий с помощью функции `fail`, которая может быть получена из вызова AJAX. Это пример API потенциального типа, который был упомянут в предыдущем разделе (не будем вдаваться в детали, что именно это значит). Я думаю, что лучший способ обработки этой ситуации — следовать шаблону, который мы использовали в работе с модулями Mongoose и Redis для Node.js. Мы позволим вызывающему коду отправить обратный вызов, который принимает ошибку и актуальные данные. Затем обратный вызов обработает ошибку, если ее значение не установлено на `null`:

```
// создаем функцию content так,
// чтобы она принимала обратный вызов
tabs.push({
  "name":"Новые",
  "content":function (callback) {
    $.get("todos.json", function (toDoObjects) {
      // создаем $content для 'Новые'
      // отправляем обратный вызов с ошибкой,
      // установленной на null, и $content в качестве
      // второго аргумента
      callback(null, $content);
    }).fail(function (jqXHR, textStatus, error) {
      // в этом случае мы отправляем ошибку вместе
      // с null для $content
      callback(error, null);
    });
  }
});
```

Обратный вызов функции `call` принимает три аргумента. Ошибка — самый интересный из них для нас, его мы и отправляем в обратный вызов нашей функции.

Сейчас в обратном вызове вызывающей функции мы обрабатываем ошибку точно так же, как делали это в примерах с Mongoose и Redis:

```
tab.content(function (err, $content) {
  if (err !== null) {
    alert("Упс, возникла проблема при обработке запроса: " + err);
```

```
  } else {
    $("main .content").append($content);
  }
});
```

Можно проверить поведение программы, изменив маршрут вызова на несуществующий (отличающийся от `todos.json`). Сообщение об ошибке в этом случае будет «Not found».

Переработка серверного кода

Вы познакомились с несколькими примерами переработки серверного кода. Можно легко применить такой же подход, но для сервера существуют некоторые дополнительные решения. В этом разделе мы научимся организовывать серверный код, используя шаблон разработки «модель — представление — контроллер» (Model — View — Controller).

Организация кода

Сейчас весь наш код серверной стороны находится в одном файле `server.js`. Это не создает особых проблем для небольших приложений вроде нашего, но по мере того, как приложение будет расти и станет включать в себя другие сущности, а не только списки задач, переполнение наступит очень быстро. Поэтому наведем порядок.

Разделение задач: модели

Начнем с перемещения определения модели Mongoose из файла `server.js` в независимый модуль Node.js. Я предпочел бы поместить определение модели в ее собственную независимую папку, так как по мере роста приложения количество создаваемых моделей тоже будет увеличиваться. Поэтому создадим файл под названием `todo.js` в папке **models**, находящейся внутри проекта Amazeriffic. Внутри этого файла определим модель так же, как и ранее, а затем экспортируем ее:

```
var mongoose = require("mongoose");
// Это модель mongoose для списка задач
var ToDoSchema = mongoose.Schema({
  description: String,
  tags: [ String ]
});
var ToDo = mongoose.model("ToDo", ToDoSchema);
module.exports = ToDo;
```

Сейчас мы можем запросить (require) этот модуль в файле `server.js` и удалить код ToDo, находящийся в файле:

```
var express = require("express"),
http = require("http"),
```

```
mongoose = require("mongoose"),
// импорт модели ToDo
ToDo = require("./models/todo.js"),
app = express();
```

Если мы запустим сейчас наш сервер, то все будет работать точно так же, как раньше.

Разделение задач: представление

Перемещение моделей в свою собственную папку позволяет нам сделать разделение ответственности в программе более четким. Теперь мы знаем, что, если нужно изменить способ хранения задач в базе данных, следует отредактировать файл `todo.js` в папке **models**. Аналогично, если понадобится изменить способ ответа программы на запросы клиента, мы можем отредактировать соответствующий файл `controllers`.

В данный момент запросы `get` и `post` приходят на сервер Express, где мы отвечаем на них с помощью анонимных функций. Дадим этим функциям имена и выделим их в специальный модуль. Этот модуль будет содержать единый контроллер объекта с определенным *набором* действий, управлять им можно через маршруты Express. В данном случае для элементов списка задач мы предусмотрим два действия: `index` и `create`.

Для этого нужно создать папку **controllers** там же, где находится **models**, а внутри нее — файл под названием `todos_controller.js`. Внутри этого модуля импортируем модель ToDo и создадим объект `ToDosController`. К нему прикрепим функции, выполняющие то же, что и анонимные функции в файле `server.js`:

```
// обратите внимание на то, что нужно перейти в папку,
// в которой находится каталог models
var ToDo = require("../models/todo.js"),
ToDosController = {};
ToDosController.index = function (req, res) {
  ToDo.find({}, function (err, toDos) {
    res.json(toDos);
  });
};
ToDosController.create = function (req, res) {
  var newToDo = new ToDo({"description":req.body.description,
    "tags":req.body.tags});
  newToDo.save(function (err, result) {
    console.log(result);
    if (err !== null) {
      // элемент не был сохранен!
      console.log(err);
      res.json(500, err);
    } else {
      res.json(200, result);
    }
  });
};
module.exports = ToDosController;
```

Точно так же, как и в случае моделей, мы можем импортировать этот код в `server.js` с помощью `require`. Сделав это, мы обновим действия `route` так, чтобы они обращались к этим функциям, а не к анонимным. Обратите внимание на то, что, поскольку доступа к модели ToDo в файле `server.js` больше нет, мы убираем оператор `require` для модели:

```
var express = require("express"),
http = require("http"),
mongoose = require("mongoose"),
// импорт представления ToDoController
ToDosController = require("./controllers/todos_controller.js"),
app = express();
// наш setup/Cloud Foundry/mongoose код здесь…
// маршруты
app.get("/todos.json", ToDosController.index);
app.post("/todos", ToDosController.create);
```

Итак, теперь код несколько лучше организован и более гибок для изменений, так как мы разделили выполняемые им задачи. Файл `server.js` в основном контролирует базовые настройки сервера и маршруты, контроллер представлений — действия, которые нужно проделать при появлении запроса, а модель — задачи, связанные с базой данных. Такое разделение ответственности значительно упрощает модификацию кода по мере роста приложения, а также оптимизирует связь между HTTP-запросами от клиента и действиями на сервере.

Выражения HTTP, CRUD и REST

В главе 6 мы в общих чертах обсудили протокол HTTP. В последующем изучили, как выполняются два типа запросов HTTP: GET и POST. Они соответствуют маршрутам `get` и `post` на сервере Express, которые, в свою очередь, соотносятся с действиями, которые мы настраиваем в нашем контроллере. Но, оказывается, в HTTP есть два других глагола, которые мы не использовали до сих пор: PUT и DELETE. Они отлично соотносятся с операциями хранения данных, которые мы изучили в главе 7 (табл. 9.1).

Таблица 9.1. Соотношения команд HTTP, CRUD и контроллера

HTTP	CRUD	Действие	Поведение
POST (отправить)	Create (создать)	create (создать)	Создает новый объект и возвращает его ID
GET (получить)	Read (прочитать)	show (показать)	Возвращает объект с данным ID
PUT (поместить)	Update (обновить)	update (обновить)	Обновляет объект с данным ID
DELETE (удалить)	Delete (удалить)	destroy (уничтожить)	Удаляет объект с данным ID

Эти взаимосвязи позволяют нам создавать API, которые предоставляют простой и ясный интерфейс ресурсам, доступным на сервере. API, работающие таким об-

разом, обычно называются программными интерфейсами веб-приложений типа REST (RESTful web APIs). REST означает Representational State Transfer (передача репрезентативного состояния) и, грубо говоря, является описанием представления ресурсов веб-сервера в протоколе HTTP.

Настройка маршрутов через ID

Отличная функциональность, которую предлагает нам Express, — способность создавать переменные на маршрутах. Это позволяет создавать единичное правило, которое относится к целому набору запросов. Например, допустим, что мы хотим получить какой-то один элемент списка задач по его ID в MongoDB:

```
// Маршруты
app.get("/todos.json", ToDosController.index);
// базовые маршруты CRUD
app.get("/todos/:id", ToDosController.show);
app.post("/todos", ToDosController.create);
```

Здесь мы создали маршрут `get` к определенному элементу списка задач. Он соотносится с функцией `show` в контроллере. Обратите внимание на то, что в маршруте использовано двоеточие (`:`) для создания переменной. Таким образом, маршрут будет соответствовать любому начинающемуся с `/todos/`. Так что, если мы установим в браузере `/todos/helloworld`, запрос будет отправлен на действие `show` в контроллере. Это действие контроллера, которое отвечает за нахождение элемента с ID, равным `helloworld`.

Как же мы можем получить доступ к переменной, которая отправлена контроллеру? Оказывается, что объект запроса отслеживается в его свойстве `params`. Код для действия может выглядеть примерно так: мы запрашиваем модель и возвращаем ответ. Если не нашли соответствующего ID, то возвращаем строку «Не найдено»:

```
ToDosController.show = function (req, res) {
  // это ID, который мы отправляем через URL
  var id = req.params.id;
  // находим элемент списка задач с соответствующим ID
  ToDo.find({"_id":id}, function (err, todo) {
    if (err !== null) {
      res.json(err);
    } else {
      if (todo.length > 0) {
        res.json(todo[0]);
      } else {
        res.send("Не найдено");
      }
    }
  });
};
```

Предположим, мы уже добавили какие-то задачи в нашу базу данных; таким образом, мы можем проверить работу этого кода, запустив клиент MongoDB из командной строки и получив ID:

```
vagrant $ mongo
MongoDB shell version: 2.4.7
connecting to: test
> show dbs
amazeriffic 0.0625GB
local 0.03125GB
> use amazeriffic
switched to db amazeriffic
> show collections
system.indexes
todos
> db.todos.find()
{ "description" : "first", "_id" : ObjectId("5275643e0cff128714000001"), ... }
{ "description" : "second", "_id" : ObjectId("52756de289f2f5f014000001"), ... }
{ "description" : "test", "_id" : ObjectId("5275722a8d735d0015000001"), ... }
{ "description" : "hello", "_id" : ObjectId("5275cbdcd408d04c15000001"), ... }
```

Теперь, видя поле `_id`, мы можем запустить сервер, открыть Chrome и попробовать набрать что-то вроде http://localhost:3000/5275643e0cff128714000001 в адресной строке. Если все сработало, мы увидим JSON, возвращенный сервером.

Если вы набрете неверный ID объекта (то есть строку, которая не содержит 24 цифры или буквы от a до f), приведенный ранее код выдаст сообщение об ошибке, предупреждающее о неверном ID. На данный момент это нормально — мы исправим это в разделе «Больше теории и практики».

Использование jQuery для прокладки и удаления маршрутов

У jQuery есть возможность выполнять запросы `put` и `delete` через общую функцию `$.ajax`. Фактически тем же способом можно выполнять и запросы `get` и `post`:

```
// пример PUT с jQuery
// Здесь мы обновляем описание элемента ToDo
// у которого id равно 1234
$.ajax({
  "url" : "todos/1234",
  "type": "PUT",
  "data": {"description":"это новое описание"},
}).done(function (response) {
  // успешно!
}).fail(function (err) {
  // ошибка!
```

```
});
// Удалить пример с jQuery
// Удаляем объект ToDo с id, равным 1234
$.ajax({
  "url" : "todos/1234",
  "type": "DELETE",
}).done(function (response) {
  // успех!
}).fail(function (err) {
  // ошибка!
});
```

Можно легко связать эти действия с событиями щелчков на кнопках при удалении или обновлении объектов в базе данных из клиента (имея в виду, что существуют соответствующие маршруты к серверу).

Коды ответов HTTP

Кроме наличия в HTTP серий выражений, четко соответствующих операциям CRUD, в этом протоколе заложена серия стандартных кодов ответов, которые представляют собой возможные результаты запросов HTTP. Вы, наверное, сталкивались с известной ошибкой 404, означающей, что страница не найдена, пытаясь зайти на несуществующий сайт или страницу. Таким образом, 404 и является одним из кодов ответов, определенных в протоколе HTTP.

Другие коды ответов HTTP — это 200, означающий, что запрос был успешным, и 500, означающий общую внутреннюю ошибку сервера. Express позволяет отсылать эти значения клиенту обратно вместе со значением запроса. Таким образом, чтобы сделать маршрут `show` несколько более надежным, мы можем наряду с отсылаемыми данными включить в него соответствующие коды ответов HTTP:

```
ToDosController.show = function (req, res) {
  // это id, который был отправлен через URL
  var id = req.params.id;
  ToDo.find({"_id":id}, function (err, todo) {
    if (err !== null) {
      // возвращаем внутреннюю серверную ошибку
      res.json(500, err);
    } else {
      if (todo.length > 0) {
        // возвращаем успех!
        res.json(200, todo[0]);
      } else {
        // мы не нашли элемент списка задач с этим ID!
        res.send(404);
      }
    }
  });
};
```

Если сейчас вы запустите сервер и наберете неверный ID объекта, то увидите, что браузер Chrome автоматически отреагирует сообщением «Не найдено», получив код ошибки 404 через HTTP.

Шаблон «модель — представление — контроллер»

Переходим к одной из самых важных концепций во всей разработке веб-приложений — шаблону проектирования «модель — представление — контроллер» (Model — View — Controller (MVC)). Это подход к архитектуре приложения, определяющий проектирование приложения, управляемого данными, в целом и по сути являющийся стандартом написания веб-приложений. Этот шаблон настолько широко известен как лучший способ написания веб-приложений, что он был закреплен в большинстве популярных фреймворков соответствующей направленности.

Мы уже перерабатывали код Amazeriffic для соответствия этому шаблону, но будет полезно сделать шаг назад и понять, в чем заключаются сфера деятельности каждого из компонентов и их совместная работа в приложении.

Контроллер, как правило, самая простая часть из трех. Когда браузер делает HTTP-запрос к серверу, маршрутизатор обрабатывает запрос с соответствующим действием контроллера. Затем контроллер преобразует запрос в действие, что, как правило, соотносится с действием базы данных через модель, а после этого отправляет (или генерирует) ответ через представление (`view`). Вы можете понаблюдать это в действии `show` — контроллер находит элемент списка задач с нужным ID (если он существует), а затем формирует представление (в нашем случае JSON) объекта, чтобы отправить ответ.

Модель — это объектная абстракция элементов в базе данных. К счастью, большинство фреймворков включают средство моделирования объектов наподобие Mongoose. В Rails по умолчанию используется инструмент, который называется Active Record и работает схожим с Mongoose образом. В нашем случае модель ToDo в основном состоит из определения схемы, но, как правило, модель может включать в себя гораздо больше. Например, она может определять отношения с другими моделями, а также указывать функции, которые запускаются в случае изменения определенных аспектов модели.

И последним по очереди, но не по значимости рассмотрим компонент *представление*. В нашем случае представление означает код HTML и CSS на клиентской стороне. Контроллер просто отправляет данные в виде JSON клиенту, а клиент решает, куда их поместить. Представление, однако, зачастую может быть намного интереснее. Например, большинство фреймворков MVC включают шаблоны серверной стороны, позволяющие контроллеру сконструировать HTML в момент запроса. В мире JavaScript и Node.js известны два широко используемых шаблонных движка: Jade и EJS.

Вообще говоря, приложение работает примерно так: клиент отправляет запрос через выражение HTTP и маршрут. Маршрутизатор (в нашем случае *server.js*) решает, какому контроллеру и для какого действия отправить этот запрос. Затем контроллер использует этот запрос для взаимодействия с моделью таким же образом, а после этого определяет, как должно быть сконструировано представление. Проделав все это, он реагирует на запрос. Схематически этот процесс представлен на рис. 9.1.

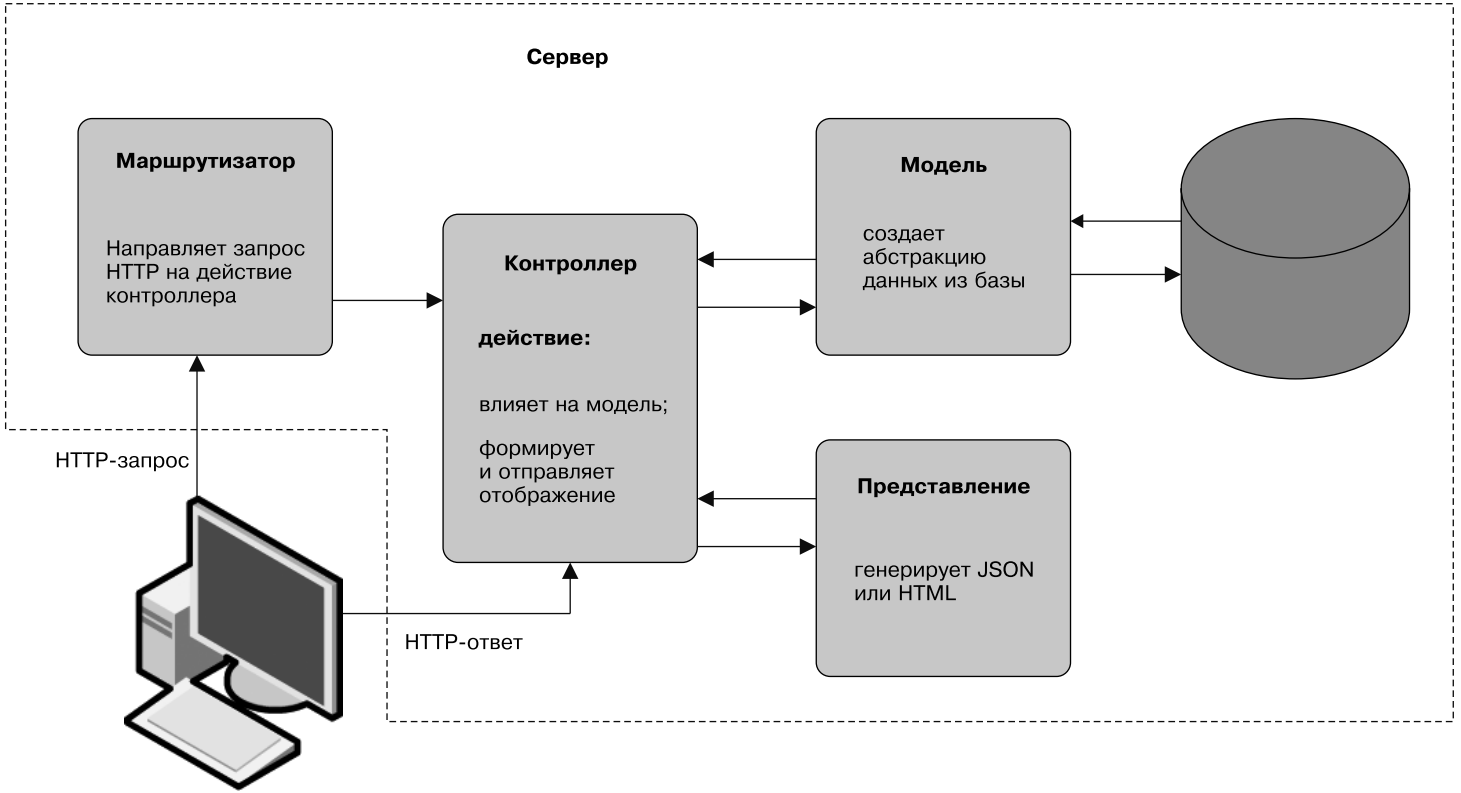


Рис. 9.1. Шаблон «модель — представление — контроллер»

Проверим его в действии. Добавим в наше приложение пользователей, в результате чего каждый пользователь сможет иметь собственный список задач.

Добавление пользователей в Amazeriffic

К этому моменту в Amazeriffic есть ровно одна сущность: объекты ToDo — списка задач. Это хорошо, если, кроме нас с вами, никто не будет их отслеживать или изменять, но таким образом мы никогда не построим бизнес на миллион долларов. Что же можно сделать, чтобы позволить большому количеству людей использовать наше приложение, контролируя свои задачи? Похоже, нужно добавить в приложение еще одну сущность — пользователей.

Построение модели пользователей

Начнем с создания модели для пользователей. Модель User будет состоять из строки, представляющей собой имя пользователя, а также ID объекта в MongoDB по умолчанию. Мы свяжем ID пользователя с принадлежащими ему объектами списка задач ToDo, а затем установим маршруты, которые будут возвращать только те задачи, которые связаны с данным пользователем.

При использовании Mongoose настроить простую модель вроде этой совсем не трудно. Создадим файл под названием `user.js` в папке **models**. В этом файле построим схему и модель Mongoose. И, как и раньше, экспортируем модель Mongoose в программу Node.js, использующую оператор `statement`:

```
var mongoose = require("mongoose");
// Это модель Mongoose для пользователей
var UserSchema = mongoose.Schema({
  username: String,
});
var User = mongoose.model("User", UserSchema);
module.exports = User;
```

Имело бы смысл включить сюда и массив объектов ToDo, но пока не будем усложнять. Немного позже мы займемся взаимосвязями между объектами ToDo и User.

Построение контроллера пользователей

Затем мы можем создать скелет контроллера пользователей. Это значит, что мы создаем пустые функции-«заглушки», которые заполним позднее по мере надобности. Настроим действие для каждой из операций CRUD, а также действие `index`, возвращающее список всех объектов-пользователей:

```
var User = require("../models/user.js"),
mongoose = require("mongoose");
var UsersController = {};
UsersController.index = function (req, res) {
  console.log("вызвано действие: индекс");
  res.send(200);
```

```
};
// Отобразить пользователя
UsersController.show = function (req, res) {
  console.log("вызвано действие: показать");
  res.send(200);
};
// Создать нового пользователя
UsersController.create = function (req, res) {
  console.log("вызвано действие: создать");
  res.send(200);
};
// Обновить существующего пользователя
UsersController.update = function (req, res) {
  console.log("вызвано действие: обновить");
  res.send(200);
};
// Удалить существующего пользователя
UsersController.destroy = function (req, res) {
  console.log("destroy action called");
  res.send(200);
};
module.exports = UsersController;
```

Надеюсь, вы можете сделать выводы об этих действиях на основе примера ToDoController. В разделе «Больше теории и практики» далее вы создадите пользовательский интерфейс, позволяющий взаимодействовать с моделью User таким же образом, как мы до этого общались с моделью ToDo.

А сейчас создадим тестового пользователя для работы с ним в следующих разделах, добавив следующий код в начало нашего контроллера пользователей:

```
// проверка, не существует ли уже пользователь
User.find({}, function (err, result) {
  if (err !== null) {
    console.log("Что-то идет не так");
    console.log(err);
  } else if (result.length === 0) {
    console.log("Создание тестового пользователя…");
    var exampleUser = new User({"username":"Сэмми"});
    exampleUser.save(function (err, result) {
      if (err) {
        console.log(err);
      } else {
        console.log("Тестовый пользователь сохранен");
      }
    });
  }
});
```

Мы используем этого пользователя для тестирования маршрутов. Можем создать другого тестового пользователя или пользователя по умолчанию, используя этот же базовый формат.

Настройка маршрутов

В этом примере маршруты — точка, где все становится куда интереснее. В первую очередь мы можем настроить базовую взаимосвязь «HTTP/действие» для модели User так же, как и раньше. Откройте файл `server.js` и добавьте следующие базовые маршруты, которые связывают запросы HTTP с действиями контроллера:

```
app.get("/users.json", usersController.index);
app.post("/users", usersController.create);
app.get("/users/:username", usersController.show);
app.put("/users/:username", usersController.update);
app.del("/users/:username", usersController.destroy);
```

Обратите внимание на то, что Express использует del вместо delete в запросах HTTP DELETE. Это потому, что delete в JavaScript имеет иное значение.

Сейчас было бы неплохо настроить приложение для работы с маршрутами и действиями, представленными в табл. 9.2.

Таблица 9.2. Маршруты и действия

Выражение	Маршрут	Действие
GET	/users/semmy/	Показать мою страницу Amazeriffic
GET	/users/semmy/todos.json	Получить все мои элементы ToDo в виде массива
POST	/users/semmy/todos	Создать для меня новую задачу ToDo
PUT	/users/semmy/todos/[id]	Обновить мой элемент списка задач ToDo с данным ID
DELETE	/users/semmy/todos/[id]	Удалить мой элемент списка задач ToDo с данным ID

Как видите, эти маршруты работают практически так же, как существующие маршруты для ToDo, за исключением того, что они ассоциированы с определенным именем пользователя. Как мы можем реализовать это в существующем коде? Мы просто настроим новый набор маршрутов, включающих префикс `users/:username`, а затем укажем для них соответствующие действия в ToDoController:

```
app.get("/users/:username/todos.json", toDosController.index);
app.post("/users/:username/todos", toDosController.create);
app.put("/users/:username/todos/:id", toDosController.update);
app.del("/users/:username/todos/:id", toDosController.destroy);
```

Сейчас, имея настройку базовых маршрутов, мы можем начать внедрять изменения. Теперь при запросе пользовательской страницы нужно отправлять существующий файл для клиентской стороны `index.html` вместо создания другого. Этого легко добиться небольшими изменениями действия `show`. Нужно, чтобы теперь действие `show` отправляло пользователю `view`. К счастью, пользовательское отображение будет совпадать с отображением по умолчанию, так что мы можем просто отправить файл `index.html` с помощью реагирующей функции Express, которая называется `sendfile`:

```
UsersController.show = function (req, res) {
  console.log("show action called");
  // отправляем базовый файл HTML, представляющий отображение
  res.sendfile("./client/index.html");
};
```

Если сейчас мы запустим сервер и пройдем по адресу localhost:3000/users/semmy/, то увидим тот же самый интерфейс, что и прежде, включая объекты списка задач, находящиеся в данный момент в нашем хранилище данных. Это потому, что мы пока не выделили набор задач для конкретного пользователя.

Есть и небольшая проблема. Откройте браузер и зайдите на localhost:3000/users/hello/. Вы увидите в точности то же самое, хотя `hello` вовсе не является пользователем! Вместо этого нужно возвращать ошибку 404, если маршрут указывает на несуществующего пользователя. Мы можем добиться этого, указывая в качестве параметра пользовательскую модель, а не имя пользователя. Поэтому действие `show` следует изменить вот таким образом:

```
UsersController.show = function (req, res) {
  console.log("show action called");
  User.find({"username":req.params.username}, function (err, result) {
    if (err) {
      console.log(err);
      res.send(500, err);
    } else if (result.length !== 0) {
      // мы нашли пользователя
      res.sendfile("./client/index.html");
    } else {
      // пользователя с таким именем не существует,
      // поэтому возвращаем ошибку 404
      res.send(404);
    }
  });
};
```

Таким образом, сейчас мы будем получать ответ на этом маршруте только в случае, если запрашиваемый пользователь существует.

Совершенствуем действия контроллера ToDo

Прежде чем двигаться дальше, нужно получить способ связать модель пользователя User и модель элементов ToDo друг с другом. Для этого немного изменим модель ToDo так, чтобы каждый объект ToDo отныне имел владельца. Владелец будет представлен в собрании ToDo через ObjectID, соответствующий определенному пользователю в собрании Users. В классической терминологии баз данных это примерно то же самое, что введение внешнего ключа, связывающего данную таблицу с какой-либо другой.

С этой целью модифицируем модель ToDo и схему, включив туда отсылку к модели User:

```
var mongoose = require("mongoose"),
ToDoSchema,
ObjectId = mongoose.Schema.Types.ObjectId;
ToDoSchema = mongoose.Schema({
  description: String,
  tags: [ String ],
  owner : { type: ObjectId, ref: "User" }
});
module.exports.ToDo = mongoose.model("ToDo", ToDoSchema);
```

Теперь, создавая объект ToDo, мы указываем владельца или значение `null`, если владелец не существует (если, например, мы добавляем элемент списка задач с маршрута по умолчанию).

Затем можно начать работу над контроллером действий ToDo, которые будут учитывать возможность вызова их с маршрута, включающего имя пользователя. Для начала модифицируем действие `index` так, чтобы оно реагировало с учетом пользовательского списка задач, если пользователь указан:

```
ToDosController.index = function (req, res) {
  var username = req.params.username || null,
  respondWithToDos;
  // вспомогательная функция, получающая элементы ToDo,
  // основанные на запросе
  respondWithToDos = function (query) {
    ToDo.find(query, function (err, toDos) {
      if (err !== null) {
        res.json(500,err);
      } else {
        res.json(200,toDos);
      }
    });
  };
  if (username !== null) {
    // получение элементов списка задач, основанных на имени пользователя
    User.find({"username":username}, function (err, result) {
      if (err !== null) {
        res.json(500, err);
      } else if (result.length === 0) {
        // не найдено пользователя с таким ID!
        res.send(404);
      } else {
        // ответ с использованием элементов списка задач для этого пользователя
        respondWithToDos({ "owner" : result[0].id });
      }
    });
  } else {
    // ответ с использованием всех элементов списка задач
    respondWithToDos({});
  }
};
```

И наконец, модифицируем действие `create` так, чтобы оно добавляло информацию о пользователе к новому элементу ToDo, если пользователь указан:

```
ToDosController.create = function (req, res) {
  var username = req.params.username || null,
  newToDo = new ToDo({"description":req.body.description,
    "tags":req.body.tags});
  User.find({"username":username}, function (err, result) {
    if (err) {
      res.send(500);
    } else {
      if (result.length === 0) {
        // пользователь не найден, поэтому
        // создаем элемент ToDo без владельца
        newToDo.owner = null;
      } else {
        // пользователь найден, поэтому
        // устанавливаем владельца для этого списка задач
        // с пользовательским ID
        newToDo.owner = result[0]._id;
      }
      newToDo.save(function (err, result) {
        if (err !== null) {
          res.json(500, err);
        } else {
          res.json(200, result);
        }
      });
    }
  });
};
```

Как видите, здесь мы устанавливаем значение `owner` равным `null`, если пользователь не существует, или соответствующим пользовательскому `_id`, если такой пользователь у нас есть. Сейчас, запустив сервер, мы можем зайти на маршрут для каждого из тестовых пользователей и создать задачи, связанные с пользователями на этих маршрутах. А зайдя на главную страницу localhost:3000, мы должны увидеть все элементы списка задач для всех пользователей.

Подведем итоги

Написать веб-приложение совсем не сложно, если вы знаете, как работают все его части. В то же время написание *гибкого в развитии* приложения требует несколько больших навыков дальновидности и планирования. Гибкость кода в веб-приложениях крепко связана с принципом разделения ответственности различных аспектов приложения. Иными словами, программа должна состоять из множества маленьких частей, которые делают какую-то одну вещь, но делают хорошо. Взаимодействие между этими частями должно быть минимизировано настолько, насколько это возможно.

В этой главе мы познакомились с несколькими базовыми приемами поддержания кода в чистом и упорядоченном состоянии. Одна из главных изученных нами вещей — это обобщение значимых понятий, которое довольно тесно связано с *объектно-ориентированным программированием*.

На серверной стороне существует шаблон проектирования, который указывает структуру кода. Шаблон проектирования «модель — представление — контроллер» — общепринятая практика для организация гибких в развитии клиент-серверных приложений. Эта практика распространилась так широко, что были созданы целые фреймворки, стимулирующие разработчиков писать приложения именно так. В число таких фреймворков входит Ruby on Rails.

В шаблоне MVC модель представляет абстракцию базы данных, выполненную с помощью какого-либо средства моделирования данных. В наших примерах представление — это просто клиентская часть приложения (HTML, CSS и JavaScript на клиентской стороне). Контроллер связывает запросы от представления с действиями в модели и возвращает данные, которые используются для нового отображения.

Еще одна известная практика организации серверной стороны веб-приложения — веб-сервис RESTful. Он открывает программные ресурсы (такие как модели данных) для клиентской стороны приложения через URL прямого действия. Клиент выполняет запрос через определенный URL, какие-либо данные и выражение HTTP, которые маршрутизатор приложения связывает с действием контроллера.

Больше теории и практики

Удаление элементов списка задач

Одна из функций, пока не затронутых нами, — удаление элементов списка задач из приложения. Вызов действия `destroy` в контроллере ToDo удалит элемент списка с соответствующим ID. Кроме того, требуется возможность удалять элемент списка из пользовательского интерфейса.

Как можно этого добиться? Если мы определили маршруты и действия правильно, то возвращаем целый объект ToDo по маршруту `todos.json`. Затем мы используем только поле объекта `description` для создания пользовательского интерфейса. Кроме того, нужен `_id` для создания и удаления ссылки.

Например, допустим, что вкладка, отображающая элементы списка задач, выглядит вот так:

```
"content":function (callback) {
  $.get("todos.json", function (toDoObjects) {
    var $content = $("<ul>");
    // создаем $content для "Старые"
    toDoObjects.forEach(function (toDo) {
      $content.append($("<li>").text(todo.description));
    });
    // выполняем обратный вызов с $content
```

```
    callback($content);
  });
}
```

Здесь мы добавляем элемет `li` для каждого объекта ToDo. Вместо того чтобы просто добавлять в `li` текст, нужно включать туда и возможность удаления элемента. Например, в результате элемент DOM мог бы быть представлен с помощью HTML следующим образом:

```
<li>Это объект ToDo <a href="todos/5275643e0cff128714000001">Удалить</a></li>
```

Добиться этого просто, немного поработав с объектом `$content`:

```
var $todoListItem = $("<li>").text(todo.description),
$todoRemoveLink = $("<a>").attr("href", "todos/"+todo._id);
// добавляем якорь для удаления
$todoListItem.append($todoRemoveLink)
$content.append($todoListItem);
```

Затем преобразуем его, прикрепив обработчик щелчков к ссылке на удаление:

```
var $todoListItem = $("<li>").text(todo.description),
$todoRemoveLink = $("<a>").attr("href", "todos/"+todo._id);
$todoRemoveLink.on("click", function () {
  $.ajax({
    // вызов запроса delete HTTP для данного объекта
  }).done(function () {
    // успешно удалив этот элемент из приложения,
    // можно удалить его из DOM
  });
  // возвращаем false, если мы не перешли по ссылке
  return false;
});
// прилагаем ссылку на удаление
$todoListItem.append($todoRemoveLink)
$content.append($todoListItem);
```

Добиться корректной работы не так-то легко, так что приготовьтесь потратить некоторое время!

Добавление пользовательской панели администратора

Кстати, у нас нет никакого способа добавлять или удалять пользователей приложения. Создадим пользовательскую администраторскую панель, где будут отображены все пользователи, а также поле ввода, позволяющее добавить новых пользователей. Это потребует создания дополнительной страницы на клиентской стороне, а также дополнительного файла JavaScript для обработки пользовательского интерфейса на этой странице. Например, вы можете создать файл `users.html`, который импортирует файл `users.js`, хранящийся в каталоге **javascripts**. Можете также использовать заново большую часть кода CSS из существующего приложения,

поэтому лучше всего просто привязать файл `style.css` из папки **stylesheets**. Там нужно будет добавить всего несколько отдельных стилистических элементов.

После того как все это заработает, любопытно будет для каждого пользователя добавить кнопку, позволяющую удалить его из приложения. Конечно, нажатие кнопки должно приводить к действию `destroy` в пользовательской модели. В дополнение к удалению пользователя из собрания пользователей было бы логично удалить и все принадлежащие ему элементы списка задач.

Это несколько сложнее, чем удалить одну задачу, поэтому попробуйте сперва справиться с этим заданием.

Представления с использованием EJS и Jade

Одна из обширных тем, которую я обошел вниманием, говоря о MVC и Node.js, — это использование шаблонных движков. Это не потому, что я отрицаю их важность или полезность, просто я попытался сделать содержание книги более гибким и дать вам возможность поработать с клиентской стороной вызовов AJAX.

Таким образом, я вам очень рекомендую уделить некоторое время изучению Jade и EJS. Использованные в них подходы к динамической генерации HTML на серверной стороне весьма значительно различаются, но оба движка отлично интегрируются с Express.

Создание нового приложения

Мы построили простое приложение для хранения списка задач. Попробуйте создать другое с чистого листа. Обобщите идеи, которые вы выучили, и постройте новое приложение. Если вы обратитесь к другим учебникам, то столкнетесь с примерами, затрагивающими дневниковые платформы или клоны Twitter. Эти незамысловатые гибкие проекты помогут вам закрепить знания, полученные в результате прочтения этой книги.

Ruby on Rails

Одна из главных целей, которые я преследовал при написании этой книги, — дать читателям хорошую основу для начала изучения Ruby on Rails, одного из самых популярных веб-фреймворков. К этому моменту, я надеюсь, вы получили достаточно знаний, чтобы приступить к учебнику по Rails Майкла Хартла. В этом учебнике он проведет вас по пути управляемой тестированием разработки простого приложения Rails. Многие из концепций, о которых я говорил, естественным образом переходят в его уроки.