Переходим на сторону сервера

Изучаем Node.js

O'REILLY®

Шелли Пауэрс

Shelley Powers

Learning Node

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Шелли Пауэрс

Изучаем Node.js

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.988.02-018
УДК 004.737.5
П21

Пауэрс Ш.
П21 Изучаем Node.js. — СПб.: Питер, 2014. — 400 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-00356-8

Node.js является серверной технологией, которая основана на разработанном компанией Google JavaScript-движке V8. Это прекрасно масштабируемая система, поддерживающая не программные потоки или отдельные процессы, а асинхронный ввод-вывод, управляемый событиями. Она идеально подходит для веб-приложений, которые не выполняют сложных вычислений, но к которым происходят частые обращения. По целям использования Node сходен с фреймворками Twisted на языке Python и EventMachine на Ruby. В отличие от большинства программ JavaScript этот фреймворк исполняется не в браузере клиента, а на стороне сервера.

С помощью этого практического руководства вы сможете быстро овладеть основами Node. Книга понравится всем, кто интересуется новыми технологиями, например веб-сокетами или платформами создания приложений. Эти темы раскрываются в ходе рассказа о том, как использовать Node в реальных приложениях.

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

ББК 32.988.02-018
УДК 004.737.5

ISBN 978-1449323073 англ.

ISBN 978-5-496-00356-8

Краткое содержание

Содержание

Предисловие

Это не совсем JavaScript

Вы выбрали для изучения Node весьма удачный момент.

Технологии, развивающиеся вокруг Node, довольно свежи и полны жизни, постоянно появляются новые варианты и уточнения. В то же время технологическая база достигла достаточного уровня зрелости, гарантирующего, что время на изучение Node будет потрачено не зря: установка еще никогда не была такой простой, даже под Windows «лучшие в своем классе» модули начинают выделяться среди, пожалуй, сотен других доступных модулей, инфраструктура стала достаточно надежной для ее практического использования.

При работе с Node нужно помнить о двух важных обстоятельствах. Во-первых, в основе Node лежит JavaScript, причем это почти тот же язык, который используется при разработке сценариев на стороне клиента. По правде сказать, можно применять и другие языки сценариев, например CoffeeScript, но JavaScript является для этой технологии *общепринятым*.

Во-вторых, необходимо помнить, что Node — это не просто JavaScript, это серверная технология, а значит, некоторые функциональные средства (и защитные механизмы), привычно ожидаемые в браузере, здесь не нужны, зато нужны многие новые и потенциально совершенно незнакомые способности.

Но если Node — это почти то же самое, что JavaScript в браузере, то почему Node?

Почему именно Node?

Если исследовать исходный код Node, то в нем обнаружится исходный код V8 (с технической стороны — ECMAScript), то есть JavaScript-движка, разработанного в Google и используемого в ядре браузера Google Chrome. Одно из преимуществ

Node.js заключается в возможности разработки Node-приложений только для одной реализации JavaScript, а не для полудюжины различных браузеров и их версий.

Технология Node была задумана как платформа создания приложений, ориентированных на высокую интенсивность ввода-вывода и невысокую интенсивность вычислений. Что еще важнее, Node предлагает эту функциональность в полностью готовом виде. Вам не придется беспокоиться о том, что приложение заблокирует всю остальную работу, ожидая завершения загрузки файла или обновления базы данных, поскольку большая часть функциональности по умолчанию относится к *асинхронному вводу-выводу*. Вам также не нужно волноваться по поводу программных потоков, поскольку Node-приложение выполняется в единственном программном потоке.

Асинхронный ввод-вывод означает, что приложение не ждет завершения ввода-вывода перед переходом к следующему шагу в коде приложения. Асинхронная природа Node более подробно рассматривается в главе 1.

Особую важность имеет то, что исходный код Node написан на языке JavaScript, с которым знакомы многие рядовые веб-разработчики. Возможно, вам придется изучать новые технологии, например веб-сокеты или Express, но, по крайней мере, наряду с ними не придется изучать еще и новый язык. Когда язык уже знаком, проще сосредоточиться на новом материале.

Для кого предназначена эта книга

Почему-то считается, что большинство людей, пришедших к Node-разработке, прежде имели дело с Ruby, Python или Rails. Лично я так не думаю, поэтому, рассказывая о Node-компонентах, не буду говорить что-то вроде: «А это похоже на Синатру».

В этой книге предполагается лишь то, что вы, читатель, прежде программировали на JavaScript и хорошо знакомы с этим языком. Вам не нужно быть специалистом высшей квалификации, но вы должны знать, о чем идет речь, когда я говорю о *замыканиях*, у вас должен быть опыт работы с Ajax, вы должны понимать, что такое обработка событий в клиентской среде. Кроме того, книга будет вам понятнее, если вы занимались обычной веб-разработкой и знакомы с такими понятиями, как HTTP-методы (`GET` и `POST`), веб-сеансы, cookie-файлы и т. д. Вы также должны уметь работать либо с консолью в Windows, либо с командной строкой в Mac OS X или Linux (Unix).

Книга также должна понравиться тем, кто интересуется новыми технологиями, например веб-сокетами или платформами создания приложений. Эти темы раскрываются в ходе рассказа о том, как использовать Node в реальных приложениях.

Самое важное при чтении книги — не бояться столкнуться с непонятным. Будьте готовы, что вам периодически придется натыкаться на препятствия в виде альфа- и бета-версий и сталкиваться с ляпами динамической технологии. В конце концов, ведь главное — это изучение Node, что действительно интересно.

Если вы не уверены, что знакомы с JavaScript в достаточной степени, можете обратиться к моей книге «Learning JavaScript», второе издание (O'Reilly).

Как получить от этой книги максимальную пользу

Вы не обязаны читать все главы по порядку, тем не менее очередность чтения зависит от того, что вы собираетесь делать потом и каков опыт вашей работы с Node.

Если работать с Node вам еще не приходилось, то лучше начать с первой главы и прочитать, как минимум, все главы по пятую включительно. В этих главах описывается, как установить Node и диспетчер Node-пакетов, как их использовать, как создать свое первое приложение, как работать с модулями. В главе 5 рассматриваются также некоторые вопросы стилевого оформления и уникальный подход к разработке асинхронных приложений.

Если вы уже кое-что знаете о Node, работали как со встроенными в Node, так и с внешними модулями, использовали REPL (интерактивная консоль, поддерживающая цикл чтения, вычисления и вывода на экран), то можете спокойно пропускать главы с первой по четвертую, но начинать чтение я все же рекомендую не далее, чем с главы 5.

Во всех примерах, описываемых в книге, я использую платформу Express и связующий модуль Connect. Если работать с Express вам еще не приходилось, то, наверное, нужно изучить главы с шестой по восьмую, где рассматриваются вопросы маршрутизации, использования прокси-серверов, веб-серверов и связующего программного обеспечения, а также дается введение в Express. В частности, если вы интересуетесь вопросами применения Express на базе MVC (Model-View-Controller — модель-представление-контроллер), то обязательно прочтите седьмую и восьмую главы.

После этих базовых глав вы можете перейти к выборочному чтению. Например, если вы преимущественно работаете с парами ключ-значение, нужно прочитать материал по Redis в главе 9, а если вас интересуют данные в виде документов, обратитесь к главе 10, где кратко рассказывается об использовании MongoDB с Node. Если же вы собираетесь работать исключительно с реляционными базами данных, то можете сразу переходить к главе 11, пропустив главы, посвященные Redis и MongoDB, хотя лучше найти время и для них, поскольку они позволяют взглянуть на использование данных по-новому.

После этих трех глав, посвященных работе с данными, мы приступим к специализированным приложениям. Глава 12 целиком посвящена доступу к графике и медиаданным, включая передачу медиаданных новому HTML5-элементу, предназначенному для воспроизведения видео, обработку PDF-документов и использование модуля Canvas. В главе 13 рассматривается весьма популярный модуль Sockets.io, специально ориентированный на новую функциональность веб-сокетов.

После разделения на два специальных варианта применения Node в главах 12 и 13 мы вернемся в общее русло повествования и не покинем его до конца книги. После того как вы уделили время примерам в других главах, вам, наверное, потребуется уделить немного времени и примерам в главе 14, более подробно изучая практику отладки и тестирования Node-приложений.

Глава 15, наверное, одна из самых сложных, но и самых важных. В ней рассматриваются вопросы безопасности и прав доступа. Я не собираюсь рекомендовать читать ее одной из первых, но перед тем как выходить в свет со своим Node-приложением, вам нужно обязательно уделить время этой главе.

Заключительную главу 16 можно смело оставить напоследок, независимо от ваших интересов и навыков. Она посвящена подготовке приложения к эксплуатации, включая развертывание Node-приложения не только на вашей системе, но и на одном из облачных серверов, предназначенных для хостинга Node-приложений. Кроме того, в ней рассматриваются вопросы развертывания Node-приложения на сервере и обеспечения его совместной работы с другим веб-сервером, таким как Apache, а также вопросы поддержания живучести вашего приложения в условиях сбоев и его перезапуска при перезагрузке системы.

Технология Node тесно связана с предлагаемыми Git приемами управления исходным кодом, и большинство (если не все) Node-модулей размещены на сайте GitHub. В приложении вы найдете руководство по выживанию в среде Git/GitHub для тех, кто с этим еще не сталкивался.

Я уже упоминал, что вам не обязательно читать главы по порядку, но я все же рекомендую поступить именно так. В основе многих глав лежит материал предыдущих, и пропуская главы, вы можете упустить важные детали. Кроме того, хотя примеры в книге достаточно автономны, я использую одно относительно простое Express-приложение под названием фабрики виджетов, код которого упоминается, начиная с главы 7, и в той или иной степени используется во всех остальных главах. Я уверен, что для вас полезнее приступить к чтению с самого начала, а затем бегло просматривать те разделы, в которых содержится уже известная вам информация, а не пропускать сразу всю главу.

Помните Алису в стране чудес? «Начни с начала, — торжественно произнес Король, — и продолжай, пока не дойдешь до конца. Тогда остановись!»

Технология

Примеры в этой книге создавались в различных выпусках Node 0.6.x. Хотя большинство из них тестировалось в среде Linux, они должны работать без всяких изменений в любой Node-среде.

Когда книга уже была запущена в производство, вышла версия Node 0.8.x. Примеры, представленные в данной книге, в большинстве своем должны работать и в Node 0.8.x, а все моменты, где в код потребуется внести изменения, мною помечены, чтобы гарантировать, что приложение будет работать в самом последнем выпуске Node.

Примеры

Примеры можно найти в архивном файле на веб-странице издательства O'Reilly, посвященной этой книге (http://oreil.ly/Learning_node). После загрузки и распаковки и при

наличии установленной копии Node вы можете установить все библиотеки, от которых зависит работа примеров, перейдя в каталог examples и введя следующую команду:

```
npm install -d
```

Более подробно работа диспетчера Node-пакетов рассмотрена в главе 4.

Соглашения, используемые в этой книге

В данной книге используются следующие соглашения, связанные с типографским оформлением.

Шрифт без засечек

Служит признаком заголовков, пунктов и кнопок меню, клавиатурных комбинаций (с использованием клавиш Alt и Ctrl), а также URL-адресов, адресов электронной почты, имен файлов, расширений этих имен, путевых имен и каталогов.

Курсив

Служит признаком новых понятий.

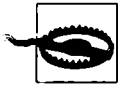
`Моноширинный шрифт`

Служит признаком команд, переменных, атрибутов, ключей, функций, типов, классов, пространств имен, методов, модулей, свойств, параметров, значений, объектов, событий, обработчиков событий, XML-тегов, HTML-тегов, макросов, контента файлов и результатов выполнения команд, а также кода, который должен набираться пользователем буквально.

Моноширинный наклонный шрифт

Текст, который должен быть заменен значениями, предоставляемыми пользователями.

Этими значками отмечены советы, примечания или общие замечания.



Этими значками отмечены предупреждения или предостережения.

Использование примеров кода

Эта книга призвана помочь вам в решении задач. По большей части вы можете использовать код из книги в своих программах и документации. Вам не нужно связываться с нами по поводу получения разрешения на это, если только вы не начнете копировать достаточно существенные фрагменты кода. Например, написание программы, в которой используется несколько фрагментов кода из этой

книги, не требует разрешения. А вот продажа или распространение компакт-дисков с примерами из книг издательства O'Reilly требует разрешения. Ответы на вопросы с использованием цитат из этой книги и приведением примеров не требуют получения разрешения. А вставка существенных объемов кода примеров из этой книги в документацию потребует разрешения.

Благодарности

Как всегда, я благодарен своим друзьям и семье за поддержку при работе над этой книгой. Особые благодарности моему редактору Симону Сен-Лорану (Simon St. Laurent), которому приходилось терпеть мои излияния.

Мои благодарности также коллективу издательства, помогавшему воплотить идею книги в осязаемый конечный продукт: Рэйчел Стили (Rachel Steely), Рэйчел Монохан (Rachel Monaghan), Килю Ван-Хорну (Kiel Van Horn), Аарону Хазелтону (Aaron Hazelton) и Ребекке Демарест (Rebecca Demarest).

Помните, что работая с Node, вы используете труд его разработчиков, начиная от создателя Node.js, Райана Дала (Ryan Dahl), и создателя диспетчера Node-пакетов, Исаака Шлютера (Isaac Schlueter), который теперь отвечает также и за Node.js.

Поставщиками весьма полезного кода и примеров для этой книги стали Берт Белдер (Bert Belder), Т. Дж. Холовайчук (TJ Holowaychuk), Джереми Ашкенас (Jeremy Ashkenas), Микеал Роджерс (Mikeal Rogers), Гильермо Раух (Guillermo Rauch), Джаред Хансон (Jared Hanson), Феликс Гейзендорфер (Felix Geisendörfer), Стив Сандерсон (Steve Sanderson), Мэтт Рэнни (Matt Ranney), Каолан МакМахон (Caolan McMahon), Реми Шарп (Remy Sharp), Крис О'Хара (Chris O'Hara), Мариано Иглесиас (Mariano Iglesias), Марко Аурелио (Marco Aurélio), Дамиан Суарез (Damián Suárez), Натан Райлич (Nathan Rajlich), Кристиан Амор Квалхейм (Christian Amor Kvalheim) и Джианни Чиапетта (Gianni Chiappetta). Мои извинения тем разработчикам модулей, которых я случайно пропустил.

И чем бы была технология Node без тех прекрасных людей, предоставивших нам учебники, ответы на вопросы и полезные руководства? Спасибо Тиму Касвелу (Tim Caswell), Феликсу Гейзендорферу (Felix Geisendörfer), Микато Такада (Mikato Takada), Гео Полу (Geo Paul), Мануэлю Кислингу (Manuel Kiessling), Скотту Хансельману (Scott Hanselman), Петеру Круминсу (Peter Krumins), Тому Хьюз-Кроучеру (Tom Hughes-Croucher), Бену Наделю (Ben Nadel) и всему коллективу Nodejitsu и Joyent.

От издательства

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

Мы будем рады узнать ваше мнение!

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

1 Установка и запуск Node.js

Node.js является серверной технологией, которая основана на разработанном компанией Google JavaScript-движке V8. Это прекрасно масштабируемая система, поддерживающая не программные потоки или отдельные процессы, а асинхронный ввод-вывод, управляемый событиями. Она идеально подходит для веб-приложений, которые не выполняют сложных вычислений, но к которым происходят частые обращения.

Если вы используете обычный веб-сервер, например Apache, то при каждом запросе веб-ресурса для обслуживания этого запроса Apache создает отдельный программный поток или вызывает новый процесс. Даже если Apache реагирует на запросы достаточно быстро, а после удовлетворения запроса все приводит в порядок, при таком подходе задействуется множество ресурсов. В результате у наиболее популярных веб-приложений возникают предпосылки для серьезных проблем производительности.

В отличие от этого Node не создает новый программный поток или процесс для каждого запроса, а прослушивает конкретные события, и когда эти события происходят, соответствующим образом на них реагирует. Node не блокирует никаких запросов, дожидаясь завершения действий, инициируемых событием, а сами события обрабатываются в относительно простом *цикле обработки событий* по принципу «первым пришел — первым обслужен».

Node-приложения создаются с помощью языка JavaScript (или альтернативных языков, компилирующихся в JavaScript), который ничем не отличается от языка, применяемого в приложениях на стороне клиента. Однако в отличие от языка JavaScript, используемого в браузере, для Node нужно создать среду разработки.

Node можно установить на платформе Unix/Linux, Mac OS или Windows. Эта глава проведет вас через всю процедуру создания среды разработки для Node в Windows 7 и Linux (Ubuntu). Установка на Mac аналогична установке на Linux. Кроме того, мы рассмотрим все требования и подготовительные действия, которые необходимо выполнить перед установкой.

Как только ваша среда разработки будет готова, я продемонстрирую простейшее Node-приложение и покажу самое главное — упомянутый ранее цикл обработки событий.

Создание среды разработки для Node

На большинстве платформ при установке Node можно использовать несколько подходов. Какой из них выбрать, зависит от имеющейся у вас среды разработки, вашей квалификации в плане работы с исходным кодом и того, как вы планируете задействовать Node в имеющихся у вас приложениях.

Установочные пакеты предоставляются как для Windows, так и для Mac OS, но вы также можете установить Node, используя копию исходного кода и скомпилировав приложение. Можно также задействовать распределенную систему управления версиям Git для *клонирования* (выгрузки) Node-*репозитория* во всех трех средах.

В этом разделе я собираюсь показать, как заставить Node работать в системе Linux на виртуальном выделенном сервере (Virtual Private Server, VPS) Ubuntu 10.04 путем непосредственного извлечения и компиляции исходного кода. Кроме этого, я покажу, как установить Node-сервер, чтобы его можно было использовать с помощью WebMatrix компании Microsoft на персональном компьютере, работающем под управлением Windows 7.

Загрузите установщики исходного и основного пакетов Node с веб-сайта http://nodejs.org/#download. Некоторые основные инструкции по установке Node в различных средах предоставлены на вики-странице https://github.com/joyent/node/wiki/Installing-Node-via-package-manager. Я также настоятельно советую вам поискать самые свежие руководства по установке Node в вашей среде, поскольку Node является очень динамичной технологией.

Установка Node на платформе Linux (Ubuntu)

Перед установкой Node на платформу Linux нужно подготовить среду. Как отмечено в документации, предоставленной на вики-странице Node, сначала убедитесь в том, установлен интерпретатор языка Python, а затем установите libssl-dev, если планируете использовать протокол SSL/TLS. В зависимости от имеющегося у вас варианта установки Linux, интерпретатор языка Python уже может быть установлен. Если это не так, можете воспользоваться установщиком пакета для установки наиболее стабильной для вашей системы версии Python, коль скоро его версия 2.6 или 2.7 (требуется для последней версии Node).

Предполагается, что у читателей книги уже есть опыт работы с JavaScript и опыт обычной веб-разработки. Учитывая это, я опускаю некоторые предупреждения и подробности при описании того, что вам нужно делать для установки Node.

Как для Ubuntu, так и для Debian вам нужно также установить другие библиотеки. Используя инструментарий Advanced Packaging Tool (APT), доступный в большинстве систем Debian GNU/Linux, вы можете обеспечить установку нужных библиотек с помощью следующих команд:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential openssl libssl-dev pkg-config
```

Команда `update` всего лишь гарантирует, что индекс пакета на вашей системе находится в актуальном состоянии, а команда `upgrade` заменяет все устаревшие пакеты на новые. Установка всех необходимых пакетов осуществляется с помощью третьей командной строки. Любые имеющиеся зависимости пакетов друг от друга обрабатываются диспетчером пакетов.

После подготовки системы загрузите на вашу систему tarball-дистрибутив Node (сжатый архивный файл исходного кода). Чтобы получить доступ к tarball-дистрибутивам, я использую команду `wget`, хотя вы можете задействовать команду `curl`. Когда я работал над книгой, самой последней версией исходного кода для Node была 0.8.2:

```
wget http://nodejs.org/dist/v0.8.2/node-v0.8.2.tar.gz
```

После загрузки файл нужно разархивировать и распаковать:

```
tar -zxf node-v0.8.2.tar.gz
```

Теперь у вас есть каталог с именем `node-v0.6.18`. Перейдите в этот каталог и для компиляции и установки Node выполните следующие команды:

```
./configure
make
sudo make install
```

Если раньше вы никогда не пользовались в Unix утилитой `make`, эти три команды подготавливают *сборочный файл* на основе варианта установки и среды вашей системы, запускают подготовительную команду `make` для проверки зависимостей, а затем выполняют завершающую команду `make` с установкой. После обработки этих команд сервер Node должен быть установлен и доступен из командной строки.

Самым забавным в программировании является то, что двух абсолютно одинаковых систем не существует. Эта последовательность действий должна привести к успеху в большинстве сред Linux, но ключевым здесь является слово «должна».

Обратите внимание, что последней командой, необходимой для установки Node, является команда `sudo`. Для установки Node этим способом требуются root-привилегии (см. следующее примечание). Но вы можете установить Node локально, используя следующие команды, с помощью которых Node устанавливается в заданный локальный подкаталог:

```
mkdir ~/working
./configure --prefix=~/working
make
make install
echo 'export PATH=~/working/bin:${PATH}' >> ~/.bashrc
. ~/.bashrc
```

Итак, здесь можно увидеть, что установка ключа конфигурации `prefix` для указания пути в вашем исходном каталоге приводит к локальной установке Node. Нужно не забыть соответствующим образом обновить переменную окружения `PATH`.

Чтобы использовать команду sudo, вы должны обладать root-привилегиями, или привилегиями суперпользователя, и ваше имя пользователя должно входить в список в специальном файле, который находится в каталоге /etc/sudoers.

Хотя Node можно установить локально, если вы собираетесь опробовать этот подход для применения Node в общей среде хостинга, то тут есть над чем задуматься. Установка Node — это еще не все, что требуется для использования Node в среде. Вам еще нужны привилегии для компиляции приложения, а также для запуска приложений посредством определенных портов (например, порта 80). Большинство общих сред хостинга не позволят вам установить собственную версию Node.

Если только у вас нет на то особых причин, я рекомендую устанавливать Node с помощью команды `sudo`.

В свое время при запуске рассматриваемого в главе 4 диспетчера Node-пакетов (Node Package Manager, npm) с root-привилегиями возникали проблемы безопасности. Но на данный момент вопросы безопасности уже решены.

Совместное использование Node и WebMatrix на платформе Windows 7

Установить Node на платформу Windows можно с помощью весьма простой последовательности действий, описанной на упомянутой ранее вики-странице. Но если вы собираетесь использовать Node в среде Windows, то, скорее всего, это будет происходить в составе инфраструктуры Windows, предназначенной для веб-разработки.

В настоящее время Node можно использовать с двумя различными инфраструктурами Windows. Одной из них является новая облачная платформа Windows Azure, позволяющая разработчикам размещать приложения на удаленной службе

(называемой *облаком*). Инструкции по установке Windows Azure SDK для Node предоставляются компанией Microsoft, поэтому я не хочу давать описание этого процесса в данной главе (об SDK мы поговорим чуть позже).

Windows Azure SDK для Node и инструкции по установке можно найти по адресу https://www.windowsazure.com/en-us/develop/nodejs/.

Еще одним подходом к использованию Node на платформе Windows (в данном случае Windows 7) является включение Node в разработанное Microsoft средство WebMatrix, поддерживающее объединение технологий с открытым кодом для веб-разработчиков. Чтобы установить и запустить Node вместе с WebMatrix в Windows 7, нужно выполнить следующие действия:

1. Установите WebMatrix.
2. Установите Node, используя самый последний установочный пакет для Windows.
3. Установите iisnode для IIS Express 7.x, что позволит Node-приложениям работать с IIS под управлением Windows.
4. Установите Node-шаблоны для WebMatrix, что позволит упростить Node-разработку.

Установка WebMatrix производится, как показано на рис. 1.1, с помощью Microsoft Web Platform Installer. Это средство также установит программу IIS Express, являющуюся версией веб-сервера Microsoft для разработчика. Загрузить WebMatrix можно с веб-сайта http://www.microsoft.com/web/webmatrix/.

Когда установка WebMatrix будет завершена, установите самую последнюю версию Node, используя установщик, предоставляемый основным сайтом Node (http://nodejs.org/#download). Установка производится одним щелчком, и когда она завершится, вы сможете, как показано на рис. 1.2, открыть окно командной строки и набрать команду node, чтобы проверить работоспособность приложения.

Чтобы система Node работала с IIS в Windows, установите iisnode, исходный модуль IIS 7.x, созданный и поддерживаемый Томашем Янчуком (Tomasz Janczuk). Его установка, как и установка Node, проходит с помощью одного щелчка с использованием предварительно собранного пакета установки, доступного по адресу https://github.com/tjanczuk/iisnode. Имеются варианты установки для x86 и для x64, но для x64 придется провести обе установки.

В процессе установки iisnode может появиться окно, показанное на рис. 1.3 и сообщающее об отсутствии пакета Microsoft Visual C++ 2010 Redistributable Package. В таком случае нужно установить этот пакет, убедившись в том, что он соответствует версии устанавливаемого модуля iisnode, это будет либо пакет x86 (доступный по адресу http://www.microsoft.com/download/en/details.aspx?id=5555), либо пакет x64 (доступный по адресу http://www.microsoft.com/download/en/details.aspx?id=14632), либо тот и другой. После установки необходимого пакета повторите установку iisnode.

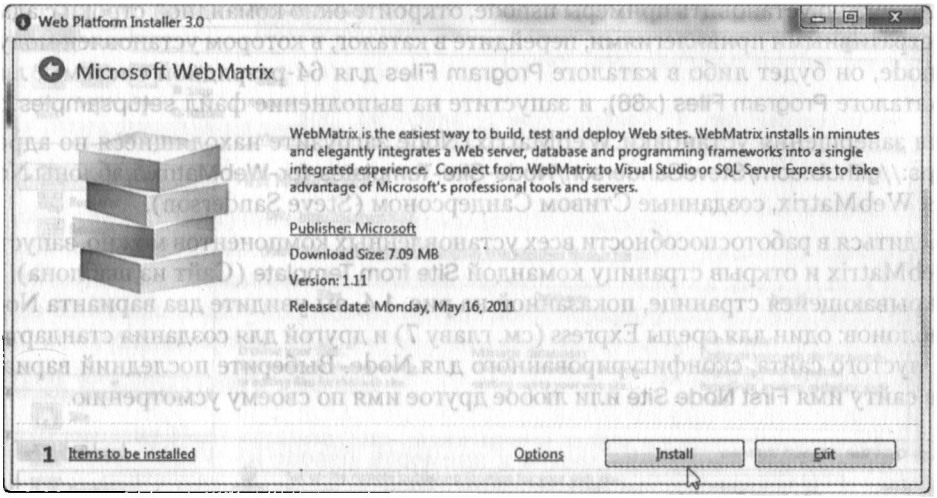


Рис. 1.1. Установка WebMatrix в Windows 7

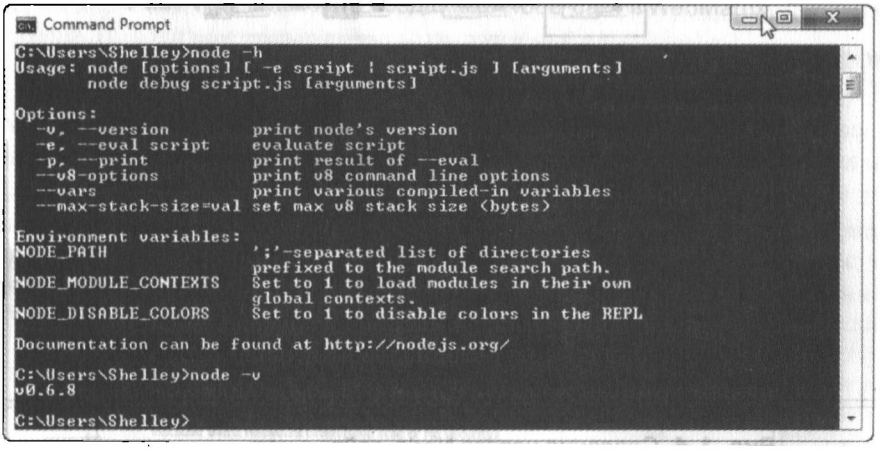


Рис. 1.2. Проверка с помощью окна командной строки правильности установки Node

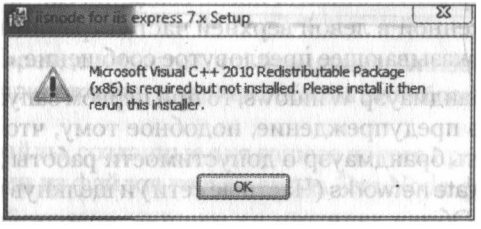


Рис. 1.3. Уведомление о необходимости установки свободно распространяемого пакета Microsoft Visual C++ 2010 Redistributable Package

Если нужно установить примеры iisnode, откройте окно командной строки с административными привилегиями, перейдите в каталог, в котором установлен модуль iisnode, он будет либо в каталоге Program Files для 64-разрядной системы, либо в каталоге Program Files (x86), и запустите на выполнение файл setupsamples.bat.

Для завершения установки WebMatrix/Node загрузите находящиеся по адресу https://github.com/SteveSanderson/Node-Site-Templates-for-WebMatrix шаблоны Node для WebMatrix, созданные Стивом Сандерсоном (Steve Sanderson).

Убедиться в работоспособности всех установленных компонентов можно, запустив WebMatrix и открыв страницу командой Site from Template (Сайт из шаблона). На открывающейся странице, показанной на рис. 1.4, вы увидите два варианта Node-шаблонов: один для среды Express (см. главу 7) и другой для создания стандартного, пустого сайта, сконфигурированного для Node. Выберите последний вариант, дав сайту имя First Node Site или любое другое имя по своему усмотрению.

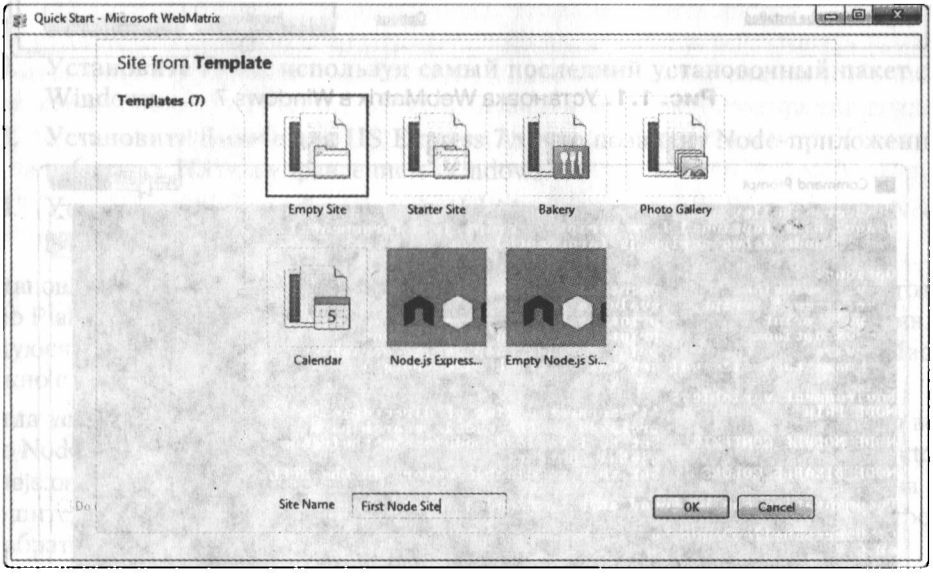


Рис. 1.4. Создание нового Node-сайта с использованием шаблона в WebMatrix

На рис. 1.5 показана среда WebMatrix после создания сайта. Щелкните на кнопке Run (Пуск), расположенной в левой верхней части страницы. В результате откроется окно браузера, показывающее пресловутое сообщение «Hello, world!».

Если у вас работает брандмауэр Windows, то при первом запуске Node-приложения может быть получено предупреждение, подобное тому, что показано на рис. 1.6. Вы должны оповестить брандмауэр о допустимости работы данного приложения, установив флажок Private networks (Частные сети) и щелкнув на кнопке Allow access (Разрешить доступ). Обмен данными на машине, используемой для разработки, следует ограничить рамками своей частной сети.

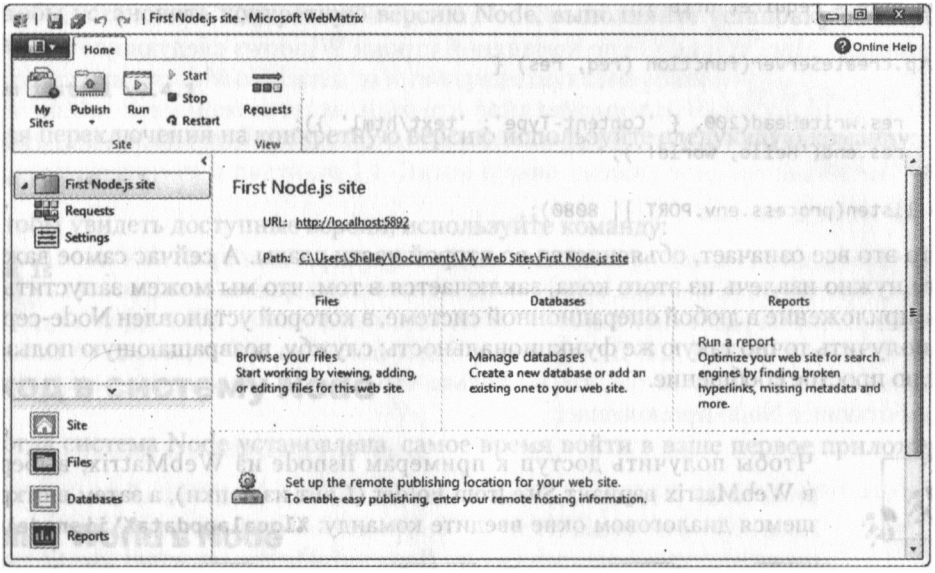


Рис. 1.5. Только что созданный Node-сайт в WebMatrix

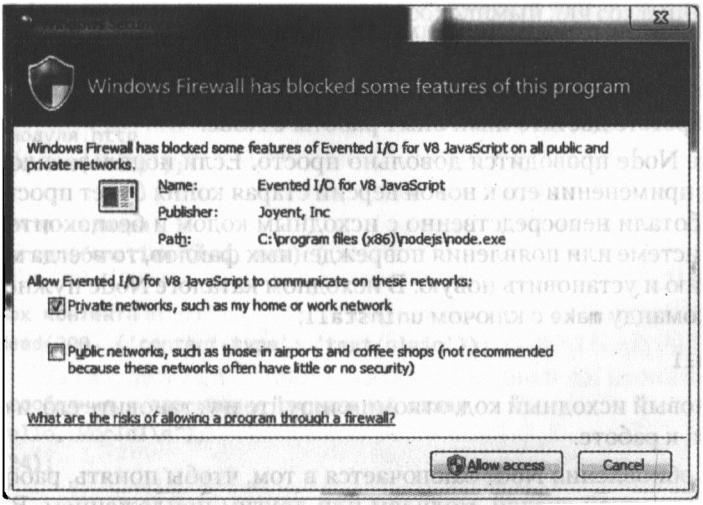


Рис. 1.6. Предупреждение о том, что брандмауэр Windows заблокировал Node-приложение, и вариант обхода этой блокировки

Если посмотреть на файлы, созданные для вашего нового проекта WebMatrix Node, то вы увидите, что один из файлов назван app.js. Это Node-файл, в котором содержится следующий код:

```
var http = require('http');

http.createServer(function (req, res) {

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('Hello, world!');

}).listen(process.env.PORT || 8080);
```

Что это все означает, объясняется во второй части главы. А сейчас самое важное, что нужно извлечь из этого кода, заключается в том, что мы можем запустить это же приложение в любой операционной системе, в которой установлен Node-сервер, и получить точно такую же функциональность: службу, возвращающую пользователю простое сообщение.

Чтобы получить доступ к примерам iisnode из WebMatrix, выберите в WebMatrix вариант Site from Folder (Сайт из папки), а затем в открывшемся диалоговом окне введите команду: `%localappdata%\iisnode\www`.

Обновление Node

Все стабильные выпуски Node имеют четные номера (текущий выпуск 0.8.x), а разрабатываемые выпуски — нечетные (текущий выпуск 0.9.x). Я рекомендую останавливать свой выбор только на стабильных выпусках, по крайней мере, пока вы не приобретете достаточный опыт работы с Node.

Обновление Node проводится довольно просто. Если использовался установщик пакета, при применении его к новой версии старая копия будет просто переписана. Если вы работали непосредственно с исходным кодом и беспокоитесь насчет беспорядка в системе или появления поврежденных файлов, то всегда можно удалить старую копию и установить новую. В исходном каталоге Node нужно будет просто запустить команду `make` с ключом `uninstall`:

```
make uninstall
```

Загрузите новый исходный код, откомпилируйте и установите его, и система снова будет готова к работе.

Сложность обновления Node заключается в том, чтобы понять, работает ли новая версия с конкретной средой, модулем или другим приложением. В большинстве случаев проблемы возникнуть не должны. Однако если они все же возникнут, есть приложение, которым можно воспользоваться для «переключения» версий Node. Это приложение называется диспетчером версий Node (Node Version Manager, Nvm).

Nvm можно загрузить с GitHub (https://github.com/creationix/nvm). Как и Node, Nvm нужно откомпилировать и установить в вашей системе.

Чтобы установить конкретную версию Node, выполняйте установку с помощью Nvm:

```
nvm install v0.4.1
```

Для переключения на конкретную версию используйте следующую команду:

```
nvm run v0.4.1
```

Чтобы увидеть доступные версии, используйте команду:

```
nvm ls
```

Вход в систему Node

Когда система Node установлена, самое время войти в ваше первое приложение.

Hello, World в Node

Как обычно, для тестирования любой новой среды разработки, языка или инструментария первым создаваемым приложением становится «Hello, World» — простая программа, выводящая приветствие всем, кто к ней обращается.

В листинге 1.1 показан весь исходный код, необходимый для создания приложения Hello, World в Node.

Листинг 1.1. Hello, World в Node

```
// загрузка модуля http
var http = require('http');

// создание http-сервера
http.createServer(function (req, res) {

  // заголовок контента
  res.writeHead(200, {'content-type': 'text/plain'});

  // запись сообщения и завершение сигнальной связи
  res.end("Hello, World!\n");
}).listen(8124);

console.log('Server running on 8124');
```

Код сохраняется в файле helloworld.js. С точки зрения функциональности на стороне сервера в этом Node-приложении нет ни большого объема кода, ни какой-либо таинственности, так что любому, даже тому, кто ничего не понимает в Node, под силу предположить, что произойдет. А самое лучшее в этом коде то, что он нам знаком, поскольку написан на языке JavaScript, который нам хорошо известен.

Чтобы запустить приложение, нужно набрать в командной строке Linux, в окне терминала MacOS или в окне командной строки Windows следующую команду:

```
node helloworld.js
```

После успешного запуска программы в командной строке будет выведена следующая строка:

```
Server running at 8124
```

Теперь обратитесь к сайту, используя любой браузер. Если приложение запущено на вашей локальной машине, нужно воспользоваться адресом `localhost:8124`. Если оно запущено на удаленной системе, нужно воспользоваться URL-адресом удаленного сайта с портом 8124. В результате будет показана веб-страница со словами «Hello, World!». Получается, что вы только что создали свое первое полноценное и работающее Node-приложение.

Если Node устанавливается в системе Fedora, следует иметь в виду, что Node переименовывается из-за конфликта имен с уже существующими там программными конструкциями. Дополнительные сведения можно найти по адресу http://nodejs.tchol.org/.

Поскольку после команды `node` не был указан амперсанд (`&`), предписывающий запуск приложения в фоновом режиме, приложение запускается, но не возвращает управление командной строке. Вы можете продолжать обращения к приложению, и при этом будут выводиться те же самые слова. Работа приложения продолжится до тех пор, пока вы не нажмете комбинацию клавиш Ctrl+C, чтбы прекратить его выполнение, или не прервете процесс каким-нибудь другим способом.

Если требуется запустить приложение в фоновом режиме на Linux-системе, нужно воспользоваться следующей командой:

```
node helloworld.js &
```

Но затем вам придется найти идентификатор процесса, используя команду `ps -ef`, и вручную прекратить выполнение нужного процесса — в данном случае процесса, которому присвоен идентификатор 3747, используя команду `kill`:

```
ps -ef | grep node
kill 3747
```

Работа процесса будет также прервана при выходе из окна терминала.

Вопросу создания надежного Node-приложения посвящена глава 16.

Запустить еще одно Node-приложение, слушающее тот же самый порт, не удастся: для одного и того же порта можно одновременно запустить только одно Node-приложение. Если у вас запущен сервер Apache, использующий порт 80, то Node-при-

ложение для этого порта запустить не удастся. Для каждого приложения нужен свой порт.

Если используется WebMatrix, то к уже существующему ранее созданному веб-сайту WebMatrix вы можете также добавить файл helloworld.js. Нужно будет просто открыть сайт, выбрав в панели меню пункт New File (Новый файл), и добавить к файлу текст, показанный в листинге 1.1. Затем нужно щелкнуть на кнопке Run (Пуск).

WebMatrix переписывает порт в приложении. Когда приложение будет запущено, обращаться к нему нужно будет через порт, определенный для проекта, а не через порт, указанный в методе http.Server.listen.

Hello, World с самого начала

Углубиться в анатомию Node-приложений нам предстоит в следующих двух главах, а сейчас давайте присмотримся к приложению Hello, World.

Если вернуться к листингу 1.1, первая строка имеет следующий вид:

```
var http = require('http');
```

Основная часть функциональных возможностей Node предоставляется через внешние приложения и библиотеки, называемые модулями. Эта строка JavaScript-кода загружает модуль HTTP, присваивая его локальной переменной.

Модуль HTTP предоставляет базовую HTTP-функциональность, обеспечивающую сетевой доступ к приложению.

Следующая строка выглядит так:

```
http.createServer(function (req, res) { ...
```

В этой строке кода с помощью функции `createServer` и безымянной функции, передаваемой вызову первой функции в виде параметра, создается новый сервер. Эта безымянная функция является функцией `requestListener` (приемником запроса) и имеет два параметра: серверный запрос (`http.ServerRequest`) и серверный ответ (`http.ServerResponse`).

В безымянной функции есть следующая строка:

```
res.writeHead(200, {'content-Type': 'text/plain'});
```

У объекта `http.ServerResponse` имеется метод, `writeHead`, который отправляет заголовок ответа с кодом статуса ответа (`200`), а также предоставляет тип контента ответа — `content-type`. В объект заголовка можно включить другую заголовочную информацию, например длину контента — `content-length` или состояние подключения — `connection`:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

Вторым необязательным параметром для `writeHead` является `reasonPhrase`, представляющий собой текстовое описание кода статуса.

За кодом создания заголовка следует команда для написания сообщения «Hello, World!»:

```
res.end("Hello, World!\n");
```

Метод `http.ServerResponse.end` подает сигнал о завершении передачи данных, то есть о том, что все заголовки и тело ответа были отправлены. Этот метод *должен* использоваться с каждым объектом `http.ServerResponse`.

Метод `end` имеет два параметра:

- Блок данных, который может быть либо строкой, либо буфером.
- Если блок данный является строкой, второй параметр указывает на кодировку.

Оба параметра являются необязательными, а второй параметр требуется, только если кодировка строки отличается от применяемой по умолчанию кодировки utf8.

Вместо передачи текста в функции `end` я мог бы использовать другой метод — `write`:

```
res.write("Hello, World!\n");
```

И далее:

```
res.end();
```

Обе функции, и безымянная, и `createServer`, завершаются в следующей строке кода:

```
}).listen(8124);
```

Метод `http.Server.listen`, пристроенный в цепочку сразу за методом `createServer`, прослушивает входящие подключения к заданному порту — в данном случае к порту 8124. Необязательными параметрами являются имя хоста и функция *обратного вызова*. Если имя хоста не предоставляется, сервер принимает подключения к веб-адресам, например http://oreilly.com или http://examples.burningbird.net.

Функции обратного вызова рассмотрены в этой главе чуть позже.

Метод `listen` является *асинхронным*. Это означает, что приложение не блокирует выполнение программы в ожидании подключения. Обрабатывается тот код, который следует за `listen`, а функция обратного вызова `listen` активизируется, когда происходит прослушиваемое событие — подключение к порту.

Последняя строка кода имеет следующий вид:

```
console.log('Server running on 8124/');
```

Объект `console` является одним из объектов мира браузеров, встроенных в Node. Эта конструкция знакома большинству JavaScript-разработчиков — она обеспечивает вывод текста в командной строке (или в среде разработки), а не отправку его клиенту.

Асинхронные функции и цикл обработки событий в Node

В основе Node лежит выполнение приложения в одном программном потоке (или процессе), а также асинхронная обработка всех событий.

Рассмотрим работу обычного веб-сервера, например Apache. Apache поддерживает две *модели мультипроцессорной обработки* (Multiprocessing Model, MPM) поступающих запросов. В первой для каждого запроса выделяется отдельный процесс, продолжающийся до тех пор, пока запрос не будет обслужен, во второй для каждого запроса выделяется отдельный программный поток.

В первой MPM-модели, известной как модель *prefork*, может создаваться столько дочерних процессов, сколько указано в конфигурационном файле Apache. Преимущество создания отдельного процесса состоит в том, что приложения, к которым обращаются посредством запроса, например PHP-приложения, не обязательно должны быть многопоточными. Недостаток заключается в том, что каждый процесс расходует память, и эта модель характеризуется неважной масштабируемостью.

Во второй MPM-модели, известной как модель *worker*, реализуется гибридная схема процесс-поток, когда каждый поступающий запрос обрабатывается с помощью нового программного потока. С точки зрения расхода памяти этот подход более эффективен, но он требует, чтобы все приложения были многопоточными (то есть были безопасными в отношении потоков). Хотя популярный язык создания веб-приложений PHP теперь безопасен в отношении потоков, нет никаких гарантий, что множество различных библиотек, используемых с интерпретатором этого языка, также безопасно в отношении потоков.

Независимо от используемой модели, запросы обрабатываются в параллельном режиме. Если к веб-приложению в одно и то же время обращается пять человек и сервер имеет соответствующую настройку, все пять запросов обрабатываются одновременно.

В Node все происходит по-другому. При запуске Node-приложения создается единственный программный поток. Node-приложение выполняется в этом потоке в ожидании, что некое приложение сделает запрос. Когда Node-приложение получает запрос, никакие другие запросы не обрабатываются до тех пор, пока не завершится обработка текущего запроса.

Все это кажется не слишком эффективным, если бы не то обстоятельство, что Node работает в асинхронном режиме, используя цикл обработки событий и функции обратного вызова. Цикл обработки событий просто опрашивает конкретные события и в нужное время вызывает обработчики событий. В Node таким обработчиком событий является функция обратного вызова.

В отличие от других однопоточных приложений, когда к Node-приложению делается запрос, оно должно, в свою очередь, запросить какие-то ресурсы (например, обратиться к базе данных или получить доступ к файлу). В этом случае Node инициирует запрос, но не ожидает ответа на этот запрос. Вместо этого запросу назначается некая функция обратного вызова. Когда запрошенное будет готово (или завершено), генерируется событие, активизирующее соответствующую функцию обратного вызова, призванную что-то сделать либо с результатом запрошенного действия, либо с запрошенными ресурсами.

Если пять человек обращаются к Node-приложению в одно и то же время и приложению нужно обратиться к ресурсам из файла, для каждого запроса Node назначает свою функцию обратного вызова событию ответа. Когда для каждого из них ресурс становится доступен, вызывается нужная функция обратного вызова, и запрос удовлетворяется. В промежутке Node-приложение может обрабатывать другие запросы либо для того же приложения, либо для какого-нибудь другого.

Хотя приложение не обрабатывает запросы в параллельном режиме, в зависимости от своей загруженности и конструкции можно даже не заметить задержки в ответе. А что лучше всего, приложение очень экономно относится к памяти и к другим ограниченным ресурсам.

Чтение файла в асинхронном режиме

Чтобы продемонстрировать асинхронную природу Node, в листинге 2.1 представлена модифицированная версия ранее приведенного в этой главе приложения Hello, World. Вместо простого вывода фразы «Hello, World!» оно открывает ранее созданный файл helloworld.js и выводит его контент на экран клиента.

Листинг 1.2. Асинхронное открытие файла и чтение его контента

```
// загрузка модуля http
var http = require('http');
var fs = require('fs');
// создание http-сервера
http.createServer(function (req, res) {
   // открытие helloworld.js и чтение контента
   fs.readFile('helloworld.js', 'utf8', function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      if (err)
         res.write('Could not find or open file for reading\n');
      else
         // при отсутствии ошибки вывод файла на экран клиента
         res.write(data);
      res.end();
     });
}).listen(8124, function() { console.log('bound to port 8124');});
console.log('Server running on 8124/');
```

В этом листинге используется новый модуль, файловая система (File System, fs). Модуль fs поддерживает стандартную файловую функциональность POSIX, включая открытие файла и обращение к его контенту. Здесь использовался метод `readFile`. В листинге 1.2 ему передается имя открываемого файла, кодировка и безымянная функция.

В листинге 1.2 хочется отметить два экземпляра с асинхронным поведением — это функции обратного вызова для методов `readFile` и `listen`.

Как уже отмечалось, метод `listen` предписывает объекту HTTP-сервера приступить к прослушиванию подключений к заданному порту. Node не блокирует выполнение программы, дожидаясь подключения, поэтому если нам нужно что-то сделать, как только подключение будет установлено, мы, как это показано в листинге 1.2, предоставляем функцию обратного вызова.

Когда подключение устанавливается, происходит событие `listening`, которое вызывает функцию обратного вызова, выводящую сообщение на консоль. Второй, более важный экземпляр функции обратного вызова назначается методу `readFile`. Доступ к файлу является, собственно говоря, длительной операцией, поэтому однопоточное приложение, к которому обращается несколько клиентов, блокирующих его на время доступа к файлу, довольно скоро зависнет, став непригодным к использованию.

Вместо этого файл открывается и его контент считывается в асинхронном режиме. Только когда контент окажется в буфере данных или же в процессе открытия или чтения возникнет ошибка, вызывается функция обратного вызова, назначенная методу `readFile`. Ей передается ошибка, если таковая имела место, или данные, если ошибки не было.

В функции обратного вызова проводится проверка на наличие ошибки, и при ее отсутствии данные возвращаются клиенту в виде ответа.

Более пристальный взгляд на асинхронное выполнение программы

Большинство программистов, пишущих на JavaScript, делает это в рамках клиентских приложений, что означает их запуск в браузере в одно и то же время одним человеком. Даже простое использование JavaScript на стороне сервера может показаться необычным, а создание JavaScript-приложения, к которому одновременно обращается сразу несколько человек, может показаться еще более необычным.

Наша задача упростилась благодаря поддержке в Node цикла обработки событий и асинхронных вызовов функций. Однако теперь мы уже не находимся в привычной для нас обстановке и ведем разработку для совершенно другой среды.

Чтобы показать различия, присущие этой новой среде, я создал два новых приложения: одно в виде службы, а другое для тестирования новой службы. В листинге 1.3 представлен код для служебного приложения.

В этом коде синхронно вызывается функция, записывающая числа от 1 до 100. Затем, как и в листинге 1.2, открывается файл, но на этот раз имя файла передается

в виде строкового параметра запроса. Кроме того, файл открывается только после таймерного события.

Листинг 1.3. Новая служба записывает последовательность чисел, а затем контент файла

```
var http = require('http');
var fs = require('fs');

// запись чисел
function writeNumbers(res) {
  var counter = 0;

  // глобальное приращение значения, запись для клиента
  for (var i = 0; i<100; i++) {
    counter++;
    res.write(counter.toString() + '\n');
   }
}

// создание http-сервера
http.createServer(function (req, res) {

   var query = require('url').parse(req.url).query;
   var app = require('querystring').parse(query).file + ".txt";

   // заголовок контента
   res.writeHead(200, {'Content-Type': 'text/plain'});

   // запись чисел
   writeNumbers(res);

   // установка таймера на открытие файла и чтение его контента
   setTimeout(function() {

      console.log('opening ' + app);
      // открытие файла и чтение его контента
      fs.readFile(app, 'utf8', function(err, data) {
         if (err)
            res.write('Could not find or open file for reading\n');
         else {
            res.write(data);
         }
         // ответ дан
         res.end();
      });
   },2000);
}).listen(8124);

console.log('Server running at 8124');
```

Цикл написания чисел служит для задержки приложения, подобной той, которая происходит при выполнении интенсивного вычислительного процесса, блокирующей программу до завершения этого процесса. Функция `setTimeout` является примером еще одной асинхронной функции, которая в свою очередь вызывает вторую асинхронную функцию `readFile`. В приложении сочетаются асинхронные и синхронные процессы.

Создайте текстовый файл с именем main.txt, содержащий любой текст на ваше усмотрение. Запуск приложения и обращение к странице из браузера Chrome со строкой запроса `file=main` приводит к следующему выводу на консоль:

```
Server running at 8124/
opening main.txt
opening undefined.txt
```

Первые две строки вполне ожидаемы. Первая является результатом запуска файла `console.log` в конце приложения, а вторая является распечаткой имени открываемого файла. А что это за файл `undefined.txt` в третьей строке?

При обработке веб-запроса в браузере нужно быть в курсе, что браузер может отправить более одного запроса. Например, браузер может также отправить второй запрос на поиск значка сайта favicon.ico. Поэтому при обработке строки запроса нужно проверить предоставление нужных данных и проигнорировать запросы без данных.

Браузер, отправляющий несколько запросов, может повлиять на ваше приложение, если вы ожидали значение, передаваемое через строку запроса. Во избежание этого вы можете внести в приложение соответствующие коррективы. И конечно же, вам по-прежнему понадобится протестировать свое приложение на нескольких различных браузерах.

До сих пор мы занимались только тестированием своих Node-приложений из браузера. Учитывая асинхронную природу Node-приложения, это не накладывало большой нагрузки.

В листинге 1.4 содержится код для очень простой проверки приложения. Все, что он делает, сводится к использованию модуля HTTP для многократно повторяющегося запроса в цикле, созданного в примере для сервера. Но мы будем также обращаться к службе, используя браузер, поскольку мы запускаем тестовую программу. И все это вместе взятое обеспечивает асинхронное тестирование приложения.

Создание асинхронных тестовых приложений рассмотрено в главе 14.

Листинг 1.4. Простое приложение для вызова нового Node-приложения 2000 раз

```
var http = require('http');
// требуемый url-адрес плюс путь и нужные нам параметры
var options = {
   host: 'localhost',
   port: 8124,
   path: '/?file=secondary',
   method: 'GET'
};
var processPublicTimeline = function(response) {
   // готово? хорошо, запись данных в файл
   console.log('запрос окончен');
};
for (var i = 0; i < 2000; i++) {
   // совершение запроса с последующим его завершением
   // для закрытия соединения
   http.request(options, processPublicTimeline).end();
}
```

Создайте второй текстовый файл с именем secondary.txt. Поместите в него что-нибудь на свое усмотрение, но сделайте так, чтобы его контент полностью отличался от контента файла main.txt.

Убедившись, что Node-приложение запущено, запустите тестовое приложение:

```
node test.js
```

Как только тестовое приложение будет запущено, обратитесь к приложению, воспользовавшись своим браузером. Если вы посмотрите на консольные сообщения, выводимые приложением, то увидите, что оно обрабатывает как ваши запросы, посланные вручную, так и запросы, автоматически посылаемые тестовым приложением. Тем не менее результаты согласуются с ожидаемыми — это веб-страница, содержащая:

- числа от 1 до 100;
- контент текстового файла (в данном случае, main.txt).

Теперь давайте все немного перемешаем. Сделайте счетчик `counter` из листинга 1.3 глобальным, а не локальным по отношению к функции цикла и запустите приложение еще раз. Затем запустите тестовую программу и обратитесь к странице в браузере.

Результаты явно изменились. Вместо того чтобы числа начинались с 1 и доходили до 100, получается нечто похожее на 2601 и 26 301. В результатах по-прежнему выводится последовательность из 99 чисел, но стартовое значение становится другим.

Причина, конечно же, в использовании глобальной переменной `counter`. Вы обращаетесь к этому же приложению в браузере вручную, в то время как тестовая программа делает это автоматически, но и вы, и она обновляете значение переменной `counter`. И ручные, и автоматические запросы к приложению обрабатываются по очереди, следовательно, конкуренции за общие данные не существует (что является

главной проблемой безопасности в отношении потоков в многопоточных средах), и если вы ожидали постоянного начального значения этой переменной, вас может ждать неприятный сюрприз.

А теперь снова измените приложение, но на этот раз удалите ключевое слово `var` перед переменной `app`, «случайно» сделав эту переменную глобальной. Все мы порой забываем ставить ключевое слово `var` в наших JavaScript-приложениях, работающих на стороне клиента. Единственным случаем, когда ощущались негативные последствия от этой ошибки, было наличие в какой-нибудь используемой библиотеке переменной с точно таким же именем.

Запустите тестовое приложение и обратитесь несколько раз к Node-службе в своем браузере. Скорее всего, в окне браузера появится текст из файла secondary.txt, а не из запрошенного файла main.txt. Причина в том, что во время между обработкой запроса имени файла и фактическим открытием файла автоматическое приложение изменило значение переменной `app`. Тестовое приложение может это сделать, поскольку вы осуществляете запрос асинхронной функции, по сути, уступая управление программой другому запросу, в то время как ваш запрос еще оставался в стадии обработки.

В этом примере показано, насколько важно при работе с Node использование ключевого слова var.

Преимущества Node

Теперь у вас есть работоспособная копия Node, возможно даже, что не одна. У вас также была возможность создать несколько Node-приложений и протестировать их, оценив разницу между синхронным и асинхронным кодом (и узнать, что произойдет, если случайно забыть поставить ключевое слово `var`).

Не все вызываемые в Node функции относятся к асинхронным. Некоторые объекты могут предоставлять как синхронные, так и асинхронные версии одной и той же функции. Но лучше всего Node работает, когда асинхронный код используется по возможности по максимуму.

Используемые в Node цикл обработки событий и функции обратного вызова имеют два основных преимущества.

Первое из них состоит в том, что приложение лучше масштабируется, поскольку у одного программного потока не так уж и много накладных расходов. Если бы мы создавали PHP-приложение, подобное Node-приложению из листинга 1.3, пользователь бы увидел точно такую же страницу, но ваша система, несомненно, заметила бы разницу. Если PHP-приложение запущено в среде Apache с используемой по умолчанию MPM-моделью prefork, то при каждом поступлении запроса к приложению этот запрос должен обрабатываться в отдельном дочернем процессе. Вероятно, при наличии сильно загруженной системы вы сможете запустить параллельно пару

сотен дочерних процессов максимум. При превышении этого количества запросов клиенту придется ждать ответа.

Второе преимущество Node заключается в минимизации расходования ресурсов, не прибегая для этого к многопоточной разработке, то есть не создавая многопоточное приложение. Если прежде вам уже приходилось разрабатывать приложения, безопасные в отношении потоков, то вы, вероятно, очень обрадуетесь этому преимуществу Node.

Однако как было показано в последнем примере, вы больше уже не ведете разработку JavaScript-приложений, предназначенных для запуска в браузере отдельными пользователями. Когда вы работаете с асинхронными приложениями, нужно убедиться в том, что вы не порождаете зависимости, требующие, чтобы вызов одной асинхронной функции завершался до вызова другой асинхронной функции, поскольку в этом случае никаких гарантий быть не может (исключая вариант, когда вызов второй функции находится внутри кода первой функции). Кроме того, глобальные переменные чрезвычайно опасны в Node, точно так же опасно забывать ставить ключевое слово `var`.

И все же со всеми этими проблемами можно справиться, особенно если учитывать преимущества Node, касающиеся низкой требовательности к ресурсам и отсутствия беспокойства относительно программных потоков.

Хотите знать о последней причине, по которой нужно отдать предпочтение Node? Вы можете спокойно программировать на JavaScript, не переживая за то, что у пользователя браузер IE6.

2 Интерактивный режим работы с Node с использованием REPL

В процессе изучения работы с Node и анализа кода для создаваемого вами собственного нестандартного модуля или Node-приложения вам не придется набирать JavaScript-код в файле и запускать его с помощью Node для тестирования. Вместе с Node поставляется интерактивный компонент REPL (read-eval-print loop — цикл чтения, вычисления и вывода на экран), о котором и пойдет речь в данной главе.

REPL (произносится «рипл») поддерживает упрощенный Emacs-стиль строкового редактирования и небольшой набор основных команд. То, что вы набираете в REPL, проходит обработку точно так же, как при вводе JavaScript-кода в файл и запуска этого файла в Node. Фактически, с помощью REPL можно создать программный код для всего вашего приложения, тестируя приложение буквально «на лету».

В данной главе будут также рассмотрены некоторые странности REPL, наряду со способами их обхода. Эти обходные пути предусматривают подмену нижележащего механизма поддержки команд, а также редактирование командной строки.

В конце концов, если встроенный компонент REPL не предоставляет всего того, что вам нужно для интерактивной среды, можно создать собственную нестандартную версию REPL с помощью соответствующего прикладного программного интерфейса (Application Program Interface, API), о котором рассказывается в заключительной части этой главы.

Полезное руководство по использованию REPL можно найти по адресу http://docs.nodejitsu.com/articles/REPL/how-to-use-nodejs-repl. На сайте Nodejitsu имеется также неплохой учебник по созданию собственной версии REPL; этот учебник находится по адресу http://docs.nodejitsu.com/articles/REPL/how-to-create-a-custom-repl.

Первое знакомство с REPL и неопределенные выражения

Чтобы приступить к работе с REPL, нужно просто набрать команду node, не указывая в качестве ее аргумента никакого файла, являющегося Node-приложением:

```
$ node
```

После этого REPL предоставляет приглашение командной строки, символом которого по умолчанию является угловая скобка (>). После этого все, что набирается в этой строке, обрабатывается основным JavaScript-движком V8.

Пользоваться REPL предельно просто. Просто набирайте свой JavaScript-код, как будто добавляете его к содержимому файла:

```
> a = 2;
2
```

REPL выводит на экран результат обработки только что набранного выражения. В приведенном фрагменте значением выражения является 2. В следующем фрагменте результатом выражения является массив из трех элементов:

```
> b = ['a','b','c'];
[ 'a', 'b', 'c' ]
```

Для доступа к последнему выражению используется символ подчеркивания (_). В следующем фрагменте переменной a присваивается значение 2, после чего результирующее выражение увеличивается на единицу, а затем еще раз на единицу:

```
> a = 2;
2
> _ ++;
3
> _ ++;
4
```

Можно даже получать доступ к свойствам или вызывать методы обозначенных подчеркиванием выражений:

```
> ['apple','orange','lime']
[ 'apple', 'orange', 'lime' ]
> _.length
3
> 3 + 4
7
> _.toString();
'7'
```

Работая с REPL, можно использовать ключевое слово var для последующего обращения к выражению или к значению, но при этом могут быть получены неожиданные результаты. Например, следующая строка в REPL возвращает значение undefined, а не 2, как предполагается:

```
var a = 2;
```

Результатом выражения является неопределенное значение, поскольку присваивание значения переменной при вычислении результата не возвращает.

Вместо этого выражения давайте рассмотрим следующий код, который отчасти дает понять, что происходит «за кулисами» REPL:

```
console.log(eval('a = 2'));
console.log(eval('var a = 2'));
```

Наберите эти строки, сохраните их в файле и запустите этот файл на выполнение, чтобы воспользоваться значениями, возвращаемыми Node. Вот результат:

```
2
undefined
```

Второй вызов `eval` не возвращает результат, поэтому возвращаемым значением является `undefined`. Вспомним, что REPL — это цикл чтения, вычисления и вывода на экран, с ударением на слове «вычисления».

И все же переменную можно использовать в REPL так, как это будет делаться в Node-приложении:

```
> var a = 2;
undefined
> a++;
2
> a++;
3
```

У последних двух командных строк есть результат, выводимый на экран с помощью REPL.

В разделе «Использование собственной нестандартной версии REPL» показано, как создать версию REPL, которая не будет выводить значение undefined.

Чтобы завершить REPL-сеанс, нужно либо дважды нажать комбинацию клавиш Ctrl+C, либо один раз — комбинацию Ctrl+D. Чуть позже, в разделе «REPL-команды» вы познакомитесь с другими способами завершения сеанса.

Преимущества REPL: представление о закулисной работе JavaScript

Вот как выглядит обычная демонстрация REPL:

```
> 3 > 2 > 1;
false
```

Этот фрагмент кода является неплохим примером того, насколько полезным может быть компонент REPL. На первый взгляд, можно было бы ожидать, что набранное

нами выражение дает в результате `true`, поскольку число 3 больше числа 2, а то, в свою очередь, больше числа 1. Но в JavaScript выражения вычисляются слева направо, и для каждого следующего вычисления возвращается результат предыдущего.

Проще всего понять, что происходит в предыдущем фрагменте кода, можно с помощью следующего REPL-сеанса:

```
> 3 > 2 > 1;
false
> 3 > 2;
true
> true > 1;
false
```

Теперь смысл результата стал понятнее. При вычислении выражения `3 > 2` возвращается значение `true`. А затем значение `true` сравнивается с числовым значением 1. JavaScript обеспечивает автоматическое преобразование данных, после которого `true` и 1 становятся одинаковыми значениями. Следовательно, `true` не больше единицы, и в результате получается `false`.

REPL предоставляет нам возможность исследования этих интересных особенностей языка JavaScript. Будем надеяться, что после тестирования кода в REPL в ваших приложениях не будет возникать таких неприятных неожиданностей (наподобие получения в качестве результата значения `false`, когда ожидалось значение `true`).

Многострочный и более сложный JavaScript-код

JavaScript-код можно набирать в REPL точно так же, как и в файле, включая необходимые инструкции для импорта модулей. В следующем тексте повторяется сеанс эксперимента с модулем Query String (qs):

```
$ node
> qs = require('querystring');
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
> val = qs.parse('file=main&file=secondary&test=one').file;
[ 'main', 'secondary' ]
```

Пока не используется ключевое слово `var`, результат выражения выводится на экран, в данном случае результатом является интерфейс объекта `querystring`. Так в чем же здесь бонус? Вы не только получаете доступ к объекту, но заодно еще узнаете о подробностях интерфейса этого объекта. Но если нужно отказаться от потенциально длинного вывода текста, воспользуйтесь ключевым словом `var`:

```
> var qs = require('querystring');
```

Как при одном, так и при другом подходе вы сможете с помощью переменной `qs` получить доступ к объекту `querystring`.

Кроме возможности включения внешних модулей, REPL легко справляется с обработкой многострочных выражений, предоставляя текстовый индикатор вложенного кода, который следует за открывающей фигурной скобкой ({):

```
> var test = function (x, y) {
... var val = x * y;
... return val;
... };
undefined
> test(3,4);
12
```

С помощью многоточия REPL показывает, что все приведенные далее инструкции идут за открывающей фигурной скобкой, и следовательно, команда не завершена. То же самое делается и для открывающей круглой скобки:

```
> test(4,
... 5);
20
```

Повышение уровня вложенности приводит к генерированию большего количества точек; это необходимо в интерактивной среде, где в процессе набора текста можно утратить представление о том, где вы находитесь:

```
> var test = function (x, y) {
... var test2 = function (x, y) {
..... return x * y;
..... }
... return test2(x,y);
... }
undefined
> test(3,4);
12
>
```

Вы можете не только набирать код вручную, но и скопировать и вставить в строку целое Node-приложение, запустив его затем из REPL:

```
> var http = require('http');
undefined
> http.createServer(function (req, res) {
...
...   // заголовок контента
...   res.writeHead(200, {'Content-Type': 'text/plain'});
...
...   res.end("Hello person\n");
... }).listen(8124);
{ connections: 0,
```

продолжение ⇘

```
  allowHalfOpen: true,
  _handle:
   { writeQueueSize: 0,
     onconnection: [Function: onconnection],
     socket: [Circular] },
  _events:
   { request: [Function],
     connection: [Function: connectionListener] },
  httpAllowHalfOpen: false }
>
undefined
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
Undefined
```

Вы можете получить доступ к этому приложению из браузера точно так же, как при вводе текста в файл и запуска его с помощью Node. И в этом случае REPL позволяет по-новому взглянуть на код, выделяемый полужирным шрифтом.

На мой взгляд, лучше всего использовать REPL для быстрого просмотра объектов. Например, основной Node-объект `global` документирован на веб-сайте Node.js довольно слабо. Чтобы получить о нем более полное представление, я открываю REPL-сеанс и передаю объект методу `console.log`:

```
> console.log(global)
```

Следующее действие приведет к аналогичному результату:

```
> gl = global;
```

Я не воспроизвожу то, что будет выведено в REPL, давая вам возможность попробовать сделать то же самое, поскольку интерфейс для `global` слишком объемный. Из этого упражнения главное вынести, что мы можем в любой момент быстро и просто увидеть интерфейс объекта. Это удобный способ вспомнить о вызываемом методе или о доступных свойствах.

Дополнительные сведения об объекте global можно найти в главе 3.

Для перехода по командам, набранным в REPL, можно воспользоваться клавишами с направленными вверх и вниз стрелками. Это может стать удобным способом просмотра и редактирования введенного кода, хотя и с некоторыми ограничениями.

Рассмотрим следующий REPL-сеанс:

```
> var myFruit = function(fruitArray,pickOne) {
... return fruitArray[pickOne - 1];
... }
undefined
> fruit = ['apples','oranges','limes','cherries'];
[ 'apples',
  'oranges',
```

```
  'limes',
  'cherries' ]
> myFruit(fruit,2);
'oranges'
> myFruit(fruit,0);
undefined
> var myFruit = function(fruitArray,pickOne) {
... if (pickOne <= 0) return 'invalid number';
... return fruitArray[pickOne - 1];
... };
undefined
> myFruit(fruit,0);
'invalid number'
> myFruit(fruit,1);
'apples'
```

Хотя это и не показано в данном коде, когда я изменил функцию для проверки введенного значения, на самом деле я с помощью клавиши ↑ прошел вверх по контенту к начальному объявлению функции, а затем нажал клавишу ввода для перезапуска функции. Я добавил новую строку, а затем опять воспользовался клавишами со стрелками для повторения ранее набранных записей до тех пор, пока функция не была завершена. Я также использовал клавишу ↑ для повторения вызова функции, который привел к результату `undefined`.

А не слишком ли много работы просто для того, чтобы избежать повторного набора подобных простых команд? Но давайте, к примеру, рассмотрим следующие регулярные выражения:

```
> var ssRe = /^\d{3}-\d{2}-\d{4}$/;
undefined
> ssRe.test('555-55-5555');
true
> var decRe = /^\s*(\+|-)?((\d+(\.\d+)?)|(\.\d+))\s*$/;
undefined
> decRe.test(56.5);
True
```

При работе с регулярными выражениями у меня сразу же ухудшается самочувствие, поскольку для получения приемлемого результата мне приходится заниматься их многократной настройкой. В то же время тестирование регулярных выражений с помощью REPL приводит к весьма впечатляющим результатам, а повторный набор длинных регулярных выражений может потребовать огромного объема работы.

К счастью, в REPL нам достаточно нажимать клавишу ↑ для поиска строки, в которой было создано регулярное выражение, вносить в строку поправки и продолжать работу, проводя последующую проверку.

Кроме клавиш со стрелками для автоматического завершения ввода текста можно использовать клавишу Tab. В качестве примера наберите в командной строке символы `va`, а затем нажмите клавишу Tab. REPL автоматически завершит ввод, введя ключевое слово `var`. Клавишу Tab можно также использовать для автоматического

завершения имени любой глобальной или локальной переменной. В табл. 2.1 предлагается краткая сводка клавиатурных команд, доступных в REPL.

Таблица 2.1. Управление REPL с помощью клавиатурных команд

Клавиатурный ввод	Его действие
Ctrl+C	Завершает выполнение текущей команды. Повторное нажатие клавиш Ctrl+C приводит к выходу из REPL
Ctrl+D	Выход из REPL
Tab	Автоматическое завершение имени глобальной или локальной переменной
Стрелка вверх	Проход вверх по списку введенных команд
Стрелка вниз	Проход вниз по списку введенных команд
Подчеркивание (_)	Ссылка на результат вычисления последнего выражения

Если вас беспокоит, что потратив много времени на ввод кода в REPL в итоге будет нечего показать, ваши волнения напрасны: получившийся текущий контекст можно сохранить с помощью команды `.save`. Эта и другие REPL-команды рассматриваются в следующем разделе.

REPL-команды

REPL располагает простым интерфейсом с небольшим набором полезных команд. В предыдущем разделе была упомянута команда `.save`. Эта команда сохраняет в файле все, что было вами введено в текущий объект контента. Пока не будет специально создан новый объект контента или не будет использована команда `.clear`, контент включает в себя все, что было введено в текущем REPL-сеансе:

```
> .save ./dir/session/save.js
```

Сохраняется только введенный вами код, как будто вы набирали его непосредственно в файле, используя текстовый редактор.

Вот полный перечень REPL-команд с описанием их назначения:

`.break`

> Если вы запутались в многострочном вводе, команда `.break` вернет вас к самому началу введенного кода, но весь многострочный ввод при этом будет потерян.

`.clear`

> Перезапуск объекта контента и очистка любого многострочного выражения. Эта команда, по сути, запускает сеанс с самого начала.

`.exit`

Выход из REPL.

`.help`

Вывод всех доступных REPL-команд.

`.save`

Сохранение текущего REPL-сеанса в файле.

`.load`

Загрузка файла в текущий сеанс (`.load` *`/путь/к/файлу`*`.js`).

Если вы создаете приложение, используя REPL в качестве редактора, могу дать совет: почаще сохраняйте свою работу с помощью команды `.save`. Хотя текущие команды остаются в истории ввода, попытка воссоздать из них свой код — занятие весьма непростое.

Поскольку затронут вопрос сохранения и истории ввода, давайте перейдем к настройке этих аспектов работы с REPL.

REPL и утилита rlwrap

В документации на веб-сайте Node.js, относящейся к REPL, упоминается о переменной окружения, позволяющей использовать REPL с утилитой rlwrap. Что это за утилита и как работать с ней в REPL?

Утилита rlwrap является оболочкой, добавляющей к командным строкам GNU-функциональность библиотеки readline, позволяющую повысить гибкость клавиатурного ввода. Она перехватывает клавиатурный ввод и предоставляет такую дополнительную функциональность, как усовершенствованное строковое редактирование и надежное сохранение истории ввода команд.

Для использования этой возможности в REPL нужно установить rlwrap и readline, тем более что большинство дистрибутивов Unix предлагают простую пакетную установку. Например, на моей системе Ubuntu установка rlwrap свелась к простой команде:

```
apt-get install rlwrap
```

Пользователи Mac должны использовать для этих приложений соответствующий установщик. Пользователям Windows придется задействовать какой-нибудь эмулятор Unix, например Cygwin.

Вот краткая и наглядная демонстрация использования REPL с утилитой rlwrap для изменения цвета REPL-приглашения на фиолетовый:

```
env NODE_NO_READLINE=1 rlwrap -ppurple node
```

Если нужно придать REPL-приглашению фиолетовый цвет на постоянной основе, можно к своему bashrc-файлу добавить псевдоним (alias):

```
alias node="env NODE_NO_READLINE=1 rlwrap -ppurple node"
```

Для изменения как знака приглашения, так и цвета я используют следующую команду:

```
env NODE_NO_READLINE=1 rlwrap -ppurple -S "::>" node
```

В результате мое фиолетовое приглашение будет выглядеть так:

```
::>
```

Особенно полезной в rlwrap является возможность сохранять историю ввода между REPL-сеансами. По умолчанию у нас есть доступ к истории командной строки только в рамках REPL-сеанса. Однако благодаря rlwrap при следующем обращении к REPL у нас будет доступ не только к истории ввода команд в рамках текущего сеанса, но и к истории ввода команд в предыдущих сеансах (и к другим записям командной строки). Показанные далее команды не были набраны в командной строке в текущем сеансе, а были извлечены из истории ввода с помощью клавиши ↑:

```
# env NODE_NO_READLINE=1 rlwrap -ppurple -S "::>" node
::>e = ['a','b'];
[ 'a', 'b' ]
::>3 > 2 > 1;
false
```

Но как бы ни была полезна утилита rlwrap, при наборе выражения, не возвращающего значения, мы по-прежнему будем получать неопределенное значение (`undefined`). Однако эту и другие странности среды можно исправить, создав собственную нестандартную версию REPL.

Использование собственной нестандартной версии REPL

Node дает возможность создать собственную версию REPL. Для этого сначала нужно подключить REPL-модуль (`repl`):

```
var repl = require("repl");
```

Для создания новой версии REPL для объекта `repl` вызывается метод `start` со следующим синтаксисом:

```
repl.start([prompt], [stream], [eval], [useGlobal], [ignoreUndefined]);
```

Все параметры не обязательны. Если они отсутствуют, используются значения, предлагаемые по умолчанию:

`prompt`

: Значение по умолчанию >.

`stream`

: Значение по умолчанию `process.stdin`.

`eval`

: Значение по умолчанию — оболочка `async` для `eval`.

`useGlobal`

Значение по умолчанию `false` служит для запуска нового контекста вместо использования глобального объекта.

`ignoreUndefined`

Значение по умолчанию `false` означает запрет на игнорирование неопределенных значений (`undefined`) в ответах.

Я считаю, что в REPL получение в качестве результата выражения значения `undefined` неприемлемо, поэтому я создал собственную нестандартную версию REPL. Для этого понадобилось буквально две строки кода (исключая комментарий):

```
repl = require("repl");
// запуск REPL с параметром ignoreUndefined, установленным в true
repl.start("node via stdin> ", null, null, null, true);
```

Я запустил файл repl.js, используя Node:

```
node repl.js
```

Затем я воспользовался нестандартной версией REPL точно так же, как делал это со встроенной версией. Теперь я больше не получаю раздражающего меня результата `undefined` после первого присваивания значения переменной. В то же время я по-прежнему получаю ответы, отличные от `undefined`:

```
node via stdin> var ct = 0;
node via stdin> ct++;
0
node via stdin> console.log(ct);
1
node via stdin> ++ct;
2
node via stdin> console.log(ct);
2
```

В своем коде я решил оставить предлагаемые по умолчанию значения для всего, кроме параметров `prompt` и `ignoreUndefined`. Присваивание другим параметрам значения `null` заставляет Node использовать для каждого из них значения, предлагаемые по умолчанию.

Благодаря собственной версии REPL можно заменить функцию `eval`. Единственным требованием является наличие у нее конкретного формата:

```
function eval(cmd, callback) {
   callback(null, result);
}
```

Параметр `stream` также вызывает определенный интерес. Можно запустить несколько версий REPL, получая данные посредством как стандартного ввода (по умолчанию), так и сокетов. В документации по REPL на сайте Node.js приводится следующий пример кода, прослушивающего TCP-сокет:

```
var repl = require("repl"),
    net = require("net");
// запуск REPL с параметром ignoreUndefined, установленным в true
repl.start("node via stdin> ", null, null, null, true);
net.createServer(function (socket) {
  repl.start("node via TCP socket> ", socket);
}).listen(8124);
```

При запуске приложения вы получаете стандартное приглашение ввода, как у запущенного Node-приложения. Но также вы можете получить доступ к REPL через TCP. Для обращения к этой версии REPL я использовал в качестве Telnet-клиента PuTTY. Эта версия работает, но до определенного момента. Сначала мне пришлось выдать команду `.clear` и отключить форматирование, а когда я попытался воспользоваться символом подчеркивания для ссылки на последнее выражение, среда Node, как показано на рис. 2.1, меня не поняла.

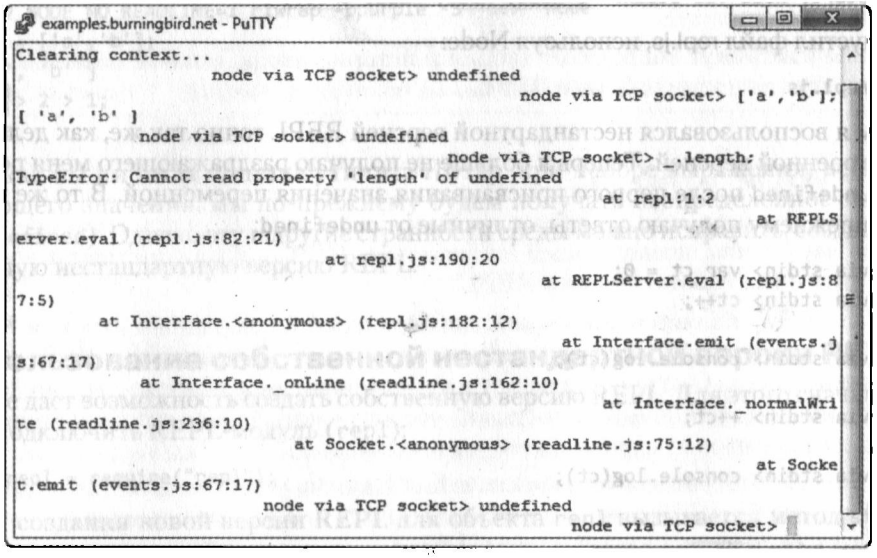


Рис. 2.1. Вариант PuTTY не полностью идентичен REPL через TCP

Я попробовал поработать с Telnet-клиентом Windows 7, но ответ был еще хуже. А вот при использовании Linux Telnet-клиент работал без нареканий.

Проблема, как вами, возможно, и ожидалось, заключается в настройке Telnet-клиента. Но я не стал рассматривать этот вопрос, поскольку не собирался запускать REPL из открытого Telnet-сокета или рекомендовать такой запуск, по крайней мере, без надежной защиты. Это сродни использованию функции `eval()` в вашем коде на стороне клиента без очистки текста, отправляемого вашими пользователями на выполнение, но еще хуже.

Вы можете поддерживать запущенный REPL-модуль и осуществлять обмен данными через Unix-сокет с помощью чего-нибудь вроде GNU-утилиты Netcat:

```
nc -U /tmp/node-repl-sock
```

Команды можно вводить точно так же, как и с использованием `stdin`. Но при этом нужно иметь в виду, что как для TCP, так и для Unix-сокета любые команды `console.log` выводят информацию на консоль сервера, а не на консоль клиента:

```
console.log(someVariable); // на самом деле информация выводится на сервер
```

Я считаю, что полезнее будет создать REPL-приложение с предварительной загрузкой модулей. Как показано в листинге 2.1, после запуска REPL в приложение загружаются модули `http`, `os` и `util`, контент которых присваивается свойствам объекта `context`.

Листинг 2.1. Создание собственной нестандартной версии REPL с предварительной загрузкой модулей

```
var repl = require('repl');
var context = repl.start(">>", null, null, null, true).context;
// предварительная загрузка модулей
context.http = require('http');
context.util = require('util');
context.os = require('os');
```

Запуск приложения с помощью Node приводит к выводу REPL-приглашения, позволяющего получить доступ к модулям:

```
>>os.hostname();
'einstein'
>>util.log('message');
5 Feb 11:33:15 - message
>>
```

Если нужно запустить REPL-приложение как исполняемый файл в Linux, добавьте в приложение первой строкой следующий код:

```
#!/usr/local/bin/node
```

Внесите изменения в файл, который нужно сделать исполняемым, и запустите его на выполнение:

```
# chmod u+x replcontext.js
# ./replcontext.js
>>
```

Частые изменения — частые сохранения

Node-компонент REPL является удобным интерактивным средством, упрощающим разработку. REPL позволяет не только проверять JavaScript-код перед его включением в файлы, но и создавать приложения в интерактивном режиме, сохраняя результаты после завершения.

Еще одним полезным свойством REPL является возможность создания собственной нестандартной версии, позволяющей исключить бесполезные ответы в виде

неопределенных значений, предварительно загружать модули, изменить используемое приглашение или функцию `eval` и т. д.

Я настоятельно рекомендую присмотреться к использованию REPL с утилитой rlwrap, позволяющей сохранять команды между сеансами. Это может помочь экономить время. Кроме того, кто же откажется от дополнительных инструментов редактирования?

В рамках освоения REPL очень важно вынести из материалов данной главы одно важное правило:

При внесении изменений чаще сохраняйте код.

Если вы собираетесь тратить много времени на разработку в REPL, то даже при использовании утилиты rlwrap, запоминающей историю ввода, нужно чаще сохранять свою работу. Работа в REPL не отличается от работы в других средах редактирования, поэтому я повторю: *при внесении изменений чаще сохраняйте код.*

В Node 0.8 компонент REPL претерпел существенные изменения. Например, простой набор имени встроенного модуля, например fs, приводит к загрузке этого модуля. Другие усовершенствования упомянуты в новой документации по REPL на основном веб-сайте Node.js.

3 Ядро Node

В главе 1 с помощью традиционного (и всегда забавного) приложения Hello, World состоялось наше первое знакомство с Node-приложением, причем в примерах использовался ряд модулей того, что известно как *ядро Node* — прикладной программный интерфейс, предоставляющий основную функциональность для создания Node-приложений.

В данной главе я собираюсь рассмотреть ядро Node более подробно. Не следует ожидать, что я представлю его исчерпывающий обзор, поскольку этот прикладной программный интерфейс слишком велик по объему и динамичен по своей природе. Вместо этого мы сфокусируемся на его ключевых элементах и более пристально рассмотрим те их них, которые упоминаются в последующих главах и (или) из-за своей сложности нуждаются в более подробном описании.

В этой главе рассматриваются следующие темы:

- Глобальные Node-объекты, такие как `global`, `process` и `Buffer`.
- Таймерные методы, например `setTimeout`.
- Краткий обзор модулей и функциональности сокетов и потоков ввода-вывода.
- Объект `Utilities` и особенно его роль в Node-наследовании.
- Объект `EventEmitter` и события.

Документация по текущему стабильному выпуску Node.js доступна по адресу http://nodejs.org/api/.

Глобальные объекты global, process и Buffer

Существует ряд объектов, которые доступны всем Node-приложениям без подключения каких-либо модулей. На веб-сайте Node.js они помечены как *глобальные*.

Мы уже использовали один глобальный объект, `require`, предназначенный для включения модулей в приложения. Также для вывода контрольных сообщений на консоль часто использовался еще один глобальный объект, `console`. Другие глобальные объекты требуются для нижележащей реализации Node, но не относятся к категории объектов, к которым обязательно нужен доступ из любого места приложения или даже вообще знание об их существовании. Тем не менее некоторые из них достаточно важны для нас и требуют более пристального изучения, поскольку помогают понять ключевые принципы работы Node.

В частности, мы собираемся исследовать следующие объекты:

- Объект `global` представляет собой глобальное пространство имен.
- Объект `process` поддерживает такую важную функциональность, как оболочки для трех стандартных потоков ввода-вывода, а также функциональность для преобразования синхронных функций в асинхронные функции обратного вызова.
- Класс `Buffer` — это глобальный объект, предоставляющий простое хранилище данных и средства управления этим хранилищем.
- Дочерние процессы.
- Модули, используемые для разрешения доменных имен и обработки URL-адресов.

Объект global

Объект `global` является объектом глобального *пространства имен*. В некоторой степени он похож на объект `windows` браузера тем, что предоставляет доступ к глобальным свойствам и методам и на него не нужно ссылаться непосредственно по имени.

Объект `global` можно вывести на консоль из REPL:

```
> console.log(global)
```

Все, что выводится на экран, является интерфейсом для всех остальных глобальных объектов, а также полезной информацией о системе, в которой вы работаете.

Я уже упоминал, что объект `global` похож на объект `windows` браузера, но здесь есть ряд ключевых отличий, причем касающихся не только доступности свойств и методов. Объект `windows` браузера по своей природе является настоящим глобальным объектом. Если в JavaScript вы определите глобальную переменную на стороне клиента, то к ней будет доступ как из веб-страницы, так и из каждой отдельной взятой библиотеки. Однако если вы создадите переменную в области видимости верхнего уровня Node-модуля (переменную, находящуюся за пределами функции), она станет глобальной только для данного модуля, но не для других модулей.

Вы можете на практике посмотреть, что происходит с объектом `global`, когда вы определяете в REPL глобальную переменную на уровне модуля. Сначала определим переменную верхнего уровня[1]:

```
> var test = "This really isn't global, as we know global";
```

[1] Сообщение гласит: «Эта переменная не является по-настоящему глобальной в том смысле, как мы это себе представляем». — *Примеч. перев.*

Затем выведем объект на консоль:

```
> console.log(global);
```

Вы должны увидеть свою переменную в самом низу в качестве нового свойства объекта **global**. Теперь присвоим объект **global** переменной, но без использования ключевого слова **var**:

```
gl = global;
```

Интерфейс глобального объекта выводится на консоль, и в самом низу вы увидите локальную переменную, определенную в качестве *циклической ссылки* (circular reference):

```
> gl = global;
...
  gl: [Circular],
  _: [Circular] }
```

Любой другой глобальный объект или метод, включая **require**, является частью интерфейса объекта **global**.

Когда Node-разработчики говорят о *контексте*, то на самом деле они имеют в виду объект **global**. В листинге 2.1 при создании собственного нестандартного REPL-объекта код обращался к объекту **context**. Объект контекста является глобальным. Когда приложение создает нестандартную версию REPL, она существует в среде нового контекста, что в данном случае означает наличие у него собственного объекта **global**. Для изменения такого порядка действий и использования при создании собственной версии REPL существующего объекта **global** требуется установка для флага **useGlobal** значения **true** вместо предлагаемого по умолчанию значения **false**.

Модули находятся в собственном глобальном пространстве имен, следовательно, если определить переменную верхнего уровня в одном модуле, она будет недоступна в других модулях. Более того, это означает, что частью приложения, содержащего модуль, становится только то, что экспортируется из этого модуля в явном виде. Фактически, вы не сможете получить доступ из приложения или из другого модуля к переменным верхнего уровня модуля, даже если попытаетесь сделать это намеренно.

Чтобы продемонстрировать это, следующий код содержит простой модуль, имеющий переменную верхнего уровня с именем **globalValue**, и функции для установки и возвращения значения. В функции, возвращающей значение, контент объекта **global** выводится методом **console.log**.

```
var globalValue;
exports.setGlobal = function(val) {
   globalValue = val;
};
exports.returnGlobal = function() {
   console.log(global);
   return globalValue;
};
```

Можно ожидать, что при выводе на экран объекта `global` мы увидим значение переменной `globalValue`, как и в тех случаях, когда устанавливаем значение переменной в наших приложениях. Но этого не происходит.

Начните REPL-сеанс и введите вызов функции `require` для подключения нового модуля:

```
> var mod1 = require('./mod1.js');
```

Установите значение, а затем запросите его:

```
> mod1.setGlobal(34);
> var val = mod1.returnGlobal();
```

Метод `console.log` выводит объект `global` до возвращения его глобально определенного значения. Мы можем увидеть в самом низу, что новая переменная содержит ссылку на импортированный модуль, но переменная `val` имеет значение `undefined`, потому что она еще не установлена. Кроме того, в вывод не включена ссылка на собственную верхнеуровневую переменную модуля `globalValue`:

```
mod1: { setGlobal: [Function], returnGlobal: [Function] },
_: undefined,
val: undefined }
```

Если запустить команду еще раз, внешняя переменная приложения будет установлена, но мы по-прежнему не увидим переменную `globalValue`:

```
mod1: { setGlobal: [Function], returnGlobal: [Function] },
_: undefined,
val: 34 }
```

Мы имеем доступ только к тем данным модуля, к которым средства доступа предоставляются самим модулем. Для JavaScript-разработчиков это означает отсутствие неожиданных и опасных конфликтов данных из-за случайно или намеренно совпавших имен глобальных переменных в библиотеках.

Объект process

Каждое Node-приложение является экземпляром Node-объекта `process` и в качестве такового имеет конкретный набор встроенных функций.

Многие методы и свойства объекта `process` предоставляют информацию о приложении и его среде. Метод `process.execPath` возвращает путь выполнения для Node-приложения, метод `process.version` предоставляет версию Node, а метод `process.platform` идентифицирует платформу сервера:

```
console.log(process.execPath);
console.log(process.version);
console.log(process.platform);
```

На моей системе (на момент написания книги) этот код возвращает следующую информацию:

```
/usr/local/bin/node
v0.6.9
linux
```

Объект process также служит оболочкой для стандартных потоков ввода-вывода stdin, stdout и stderr. Потоки stdin и stdout являются асинхронными и, соответственно, доступными по чтению и записи. Поток stderr является синхронным (блокирующим).

Чтобы продемонстрировать, как прочитать данные из потока stdin и записать их в поток stdout, рассмотрим листинг 3.1, где Node-приложение прослушивает данные в потоке stdin и повторяет их в потоке stdout. По умолчанию поток stdin приостановлен, поэтому нам нужно перед отправкой данных вызвать метод resume (возобновить).

Листинг 3.1. Чтение и запись данных с использованием соответственно потоков stdin и stdout

```
process.stdin.resume();

process.stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});
```

Запустите приложение в Node, а затем начните набирать что-нибудь в терминале. При каждом наборе какого-нибудь фрагмента текста и нажатии клавиши ввода все, что набрано, выводится на экран еще раз.

Еще одним полезным методом является метод memoryUsage, который сообщает о том, сколько памяти расходует Node-приложение. Это может пригодиться для настройки производительности или просто для удовлетворения вашего любопытства. Ответ имеет следующую структуру:

```
{ rss: 7450624, heapTotal: 2783520, heapUsed: 1375720 }
```

Свойства heapTotal и heapUsed относятся к расходованию памяти движком V8.

Последний метод, о котором я собираюсь упомянуть, называется process.nextTick. Этому методу назначается функция обратного вызова, которая активизируется в ходе следующего прохода цикла обработки событий в Node.

Метод process.nextTick используется, когда по какой-либо причине нужно приостановить функцию, причем сделать это в асинхронном режиме. Хорошим примером может служить создание новой функции, у которой в качестве параметра имеется функция обратного вызова, и нужно обеспечить реальную асинхронность этой функции. Для демонстрации рассмотрим следующий код:

```
function asynchFunction = function (data, callback) {
   process.nextTick(function() {
      callback(val);
   });
);
```

Если бы мы просто вызвали функцию обратного вызова, действие было бы синхронным. А теперь функция обратного вызова вызывается не сразу, а при следующем проходе цикла обработки событий.

Вместо метода process.nextTick можно использовать метод setTimeout, указав нулевую задержку:

```
setTimeout(function() {
   callback(val);
}, 0);
```

Однако метод `setTimeout` не столь эффективен, как `process.nextTick`. При сравнительном тестировании метод `process.nextTick` вызывается значительно быстрее, чем `setTimeout` с нулевой задержкой. Вы также можете использовать метод `process.nextTick` при запуске приложения, в котором есть функция, выполняющая сложную с вычислительной точки зрения и затратную по времени операцию. Кроме того, можно разбить процесс на этапы и вызывать каждый из них через `process.nextTick`, чтобы позволить другим запросам к Node-приложению обрабатываться, не ожидая окончания затратного по времени процесса.

Разумеется, невозможно разбить на этапы такой процесс, которому требуется непрерывность выполнения, поскольку иначе можно столкнуться с неожиданными результатами.

Объект Buffer

Класс `Buffer` также является глобальным объектом, поддерживающим в Node обработку двоичных данных. Как показано в разделе «Серверы, потоки ввода-вывода и сокеты», потоки ввода-вывода зачастую являются двоичными данными, а не строками. Чтобы преобразовать двоичные данные в строку, кодировка данных для потокового сокета изменяется с помощью функции `setEncoding`.

В качестве демонстрации можно с помощью следующего кода создать новый буфер:

```
var buf = new Buffer(string);
```

Если в буфере хранится строка, можно передать необязательный второй параметр, указывающий на кодировку. Возможны следующие варианты кодировки:

`ascii`

: Семибитный ASCII-код.

`utf8`

: Юникод-символы с многобайтной кодировкой.

`usc2`

: Юникод-символы с двухбайтной кодировкой и прямым порядком следования байтов.

`base64`

: Кодировка Base64.

`hex`

: Кодировка каждого байта в виде двух шестнадцатеричных чисел.

Можно также записать строку в существующий буфер, предоставляя необязательные параметры смещения (`offset`), длины (`length`) и кодировки (`encoding`):

```
buf.write(string); // смещение по умолчанию равно 0, длина по умолчанию
                   // равна buffer.length - offset, кодировка имеет
                   // значение utf8
```

По умолчанию данные, отправляемые между сокетами, передаются в виде буфера (в двоичном формате). Чтобы вместо этого отправить строку, нужно либо непосредственно вызвать для сокета функцию `setEncoding`, либо указать кодировку в функции, ведущей запись в сокет. По умолчанию метод `socket.write` протокола управления передачей (Transmission Control Protocol, TCP) устанавливает в качестве второго параметра значение `utf8`, но сокет, возвращаемый функцией `connectionListener`, осуществляет обратный вызов TCP-функции `createServer`, отправляя данные в виде буфера, а не в виде строки.

Таймерные функции setTimeout, clearTimeout, setInterval и clearInterval

В JavaScript таймерные функции на клиентской стороне являются частью глобального объекта `windows`, а не JavaScript, но поскольку при JavaScript-разработке они используются повсеместно, создатели Node решили включить их в API ядра Node.

Таймерные функции работают в Node точно так же, как в браузере, точнее, в браузере Chrome, поскольку в основе Node лежит JavaScript-движок V8, используемый в Chrome.

В качестве первого параметра Node-функция `setTimeout` получает функцию обратного вызова, а в качестве второго — время задержки (в миллисекундах), после чего может следовать необязательный список аргументов:

```
// таймер для открытия файла и чтения контента в объект HTTP-ответа
function on_OpenAndReadFile(filename, res) {
  console.log('открытие ' + filename);
  // открытие файла и считывание его контента
  fs.readFile(filename, 'utf8', function(err, data) {
     if (err)
         res.write('Could not find or open file for reading\n');
     else {
         res.write(data);
     }
  // ответ готов
  res.end();
}
setTimeout(openAndReadFile, 2000, filename, res);
```

В этом коде функция обратного вызова `on_OpenAndReadFile` открывает и читает файл в HTTP-ответ, когда функция вызывается после истечения примерно 2000 мс.

Как особо подчеркивается в документации по Node, нет никаких гарантий, что функция обратного вызова активизируется точно через *n* миллисекунд (независимо от значения *n*). Это полностью соответствует условиям применения функции setTimeout в браузере — мы не можем полностью контролировать среду, поскольку те или иные факторы могут задержать таймер.

Функция `clearTimeout` сбрасывает параметры, заданные функцией `setTimeout`. Если нужен периодически срабатывающий таймер, можно воспользоваться функцией `setInterval` для вызова функции через каждые *n* миллисекунд — в этом случае *n* является вторым параметром, передаваемым функции. Сбросить заданный интервал срабатывания можно вызовом функции `clearInterval`.

Серверы, потоки ввода-вывода и сокеты

Основная часть API ядра Node предназначена для создания служб прослушивания конкретных видов взаимодействий. В примерах главы 1 для создания HTTP-сервера использовался модуль HTTP. Другие методы позволяют создать TCP-сервер, TLS-сервер и UDP-сокет, или сокет дейтаграмм. Протокол TLS рассматривается в главе 15, а в данном разделе я хочу представить функциональные возможности ядра Node, касающиеся протоколов TCP и UDP. Но сначала кратко рассмотрим терминологию.

Сокет является конечной точкой соединения, а *сетевой сокет* является конечной точкой соединения между приложениями, запущенными на двух различных компьютерах сети. Данные переносятся между сокетами в так называемых *потоках ввода-вывода*. Данные в потоке могут передаваться как двоичные данные в буфере или как строки в кодировке Юникод. Оба типа данных передаются в *пакетах*: фрагментах данных определенного размера. Существуют также пакеты особого вида — это завершающие пакеты (Finish Packet, FIN). FIN-пакет отправляется сокетом, сигнализируя об окончании передачи. Управление соединениями и надежность передачи потока зависит от типа созданного сокета.

TCP-сокеты и TCP-серверы

Основные TCP-сервер и TCP-клиент могут быть созданы с помощью Node-модуля Net. Протокол TCP (Transmission Control Protocol — протокол управления передачей) является базовым для многих интернет-приложений, таких как веб-службы и электронная почта. Он предоставляет средство надежной передачи данных между клиентским и серверным сокетами.

Создание TCP-сервера немного отличается от создания HTTP-сервера в листинге 1.1 (см. главу 1). Мы создаем сервер, передавая функцию обратного вызова. TCP-сервер отличается от HTTP-сервера тем, что в TCP вместо объекта `requestListener` единственным аргументом функции обратного вызова является экземпляр сокета, прослушивающего входящие соединения.

Листинг 3.2 содержит код создания TCP-сервера. После создания серверного сокета он прослушивает два события: получения данных и закрытия соединения клиентом.

Листинг 3.2. Простой TCP-сервер с сокетом, прослушивающим клиентское соединение через порт 8124

```
var net = require('net');

var server = net.createServer(function(conn) {
  console.log('connected');
  conn.on('data', function (data) {
    console.log(data + ' от ' + conn.remoteAddress + ' ' +
      conn.remotePort);
    conn.write('Repeating: ' + data);
  });
  conn.on('close', function() {
      console.log('client closed connection');
  });
}).listen(8124);
console.log('listening on port 8124');
```

У функции `createServer` есть необязательный параметр: `allowHalfOpen`. Установка этого параметра в `true` заставляет сокет не отправлять FIN-пакет после получения FIN-пакета от клиента. Таким образом сокет остается открытым для записи (но не для чтения). Чтобы закрыть сокет, нужно явным образом вызвать метод `end`. По умолчанию параметр `allowHalfOpen` имеет значение `false`.

Обратите внимание, как посредством метода `on` функция обратного вызова назначается двум событиям. Многие Node-объекты, генерирующие события с помощью метода `on`, предоставляют возможность назначения в качестве слушателя событий специальной функции. Этот метод в первом параметре получает имя события, а во втором — функцию прослушивания.

Метод обработки событий *on* позволяет использовать Node-объекты, наследуемые от особого объекта EventEmitter, который рассматривается в данной главе чуть позже.

Создание TCP-клиента, как показано в листинге 3.3, происходит так же просто, как и создание сервера. Кодировка получаемых данных меняется вызовом метода `setEncoding` для клиента. Как уже упоминалось в разделе «Объект Buffer», данные передаются в виде буфера, но мы можем воспользоваться методом `setEncoding` для преобразования этого буфера в utf8-строку. Для передачи данных служит

принадлежащий сокету метод `write`. Он также назначает методы прослушивания двум событиям: `data` (получение данных) и `close` (закрытие соединения со стороны сервера).

Листинг 3.3. Клиентский сокет, предназначенный для отправки данных на TCP-сервер

```
var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// установка соединения с сервером
client.connect ('8124','localhost', function () {
    console.log('connected to server');
    client.write('Who needs a browser to communicate?');
});

// подготовка к вводу данных с терминала
process.stdin.resume();

// отправка данных при их получении на сервер
process.stdin.on('data', function (data) {
   client.write(data);
});

// при получении ответных данных вывод их на консоль
client.on('data',function(data) {
    console.log(data);
});

// при закрытии сервера
client.on('close',function() {
    console.log('connection is closed');
});
```

Данные, передаваемые между двумя сокетами, набираются в терминале и передаются после нажатия клавиши ввода. Клиентское приложение сначала отправляет только что набранную строку, которую TCP-сервер выводит на консоль. Сервер дублирует сообщение обратно клиенту, который, в свою очередь, выводит сообщение на свою консоль. Сервер также выводит для клиента IP-адрес и порт, используя свойства `remoteAddress` и `remotePort` сокета. Следующие строки выводятся на консоль сервером после отправки клиентом нескольких строк (в целях безопасности IP-адреса изменены):

```
Hey, hey, hey, hey-now.
 from #ipaddress 57251
Don't be mean, we don't have to be mean.
 from #ipaddress 57251
Cuz remember, no matter where you go,
```

```
 from #ipaddress 57251
there you are.
 from #ipaddress 57251
```

Соединение между клиентом и сервером поддерживается до тех пор, пока не будет прервано с той или другой стороны с помощью комбинации клавиш Ctrl+C. Тот сокет, который останется открытым, получит событие закрытия `close`; сведения об этом выводятся на консоль. Сервер может обслуживать более одного соединения с более чем одним клиентом, поскольку все необходимые функции являются асинхронными.

Как уже упоминалось, в настоящее время TCP является основным транспортным протоколом для решения большинства задач в Интернете, в том числе для рассматриваемого далее протокола HTTP.

Протокол HTTP

В главе 1 мы уже работали с HTTP-модулем. Мы создавали серверы, используя метод `createServer`, передаваемый функции, которая должна была действовать как слушатель запросов (`requestListener`). Запросы обрабатывались по мере их получения в асинхронном режиме.

В сети протокол TCP является транспортным протоколом, в то время как HTTP (HyperText Transfer Protocol — протокол передачи гипертекста) относится к прикладному уровню. Если покопаться в модулях, входящих в состав Node, можно заметить, что при создании HTTP-сервера наследуется функциональность от основанного на TCP объекта `net.Server`.

Для HTTP-сервера `requestListener` является сокетом, в то время как объект `http.ServerRequest` — потоком чтения, а объект `http.ServerResponse` — потоком записи. HTTP добавляет еще один уровень сложности, поскольку поддерживает *кодировку фрагментированной передачи* (chunked transfer encoding). Кодировка фрагментированной передачи позволяет передавать данные, когда точный размер ответа неизвестен до полной обработки данных. Для обозначения конца запроса отправляется фрагмент данных нулевой длины. Этот тип кодировки применяется при обработке объемных запросов к базам данных, когда результат запроса выводится в HTML-таблицу — в этом случае запись данных может начаться еще до получения оставшейся части запрошенных данных.

Более полные сведения о потоках ввода-вывода можно найти в разделе «Потоки ввода-вывода, каналы и построчное чтение».

Приведенные ранее в данной главе примеры, касающиеся протокола TCP, а также примеры применения протокола HTTP в главе 1 содержали код для работы с сетевыми сокетами. Однако все модули серверов и сокетов могут также подключаться к Unix-сокету, а не к конкретному сетевому порту. В отличие от сетевого

сокета, Unix-, или IPC-сокет, поддерживает взаимодействие между процессами (InterProcess Communication, IPC) в пределах одной и той же системы.

Чтобы продемонстрировать взаимодействие с Unix-сокетом, я продублировал код из листинга 1.3, но вместо привязки к порту привязал новый сервер к Unix-сокету, как показано в листинге 3.4. В приложении используется также функция `readFileSync` — это синхронная версия функции, открывающей файл и читающей его контент.

Листинг 3.4. HTTP-сервер, привязанный к Unix-сокету

```
// создание сервера
// и функции обратного вызова
var http = require('http');
var fs = require('fs');

http.createServer(function (req, res) {

  var query = require('url').parse(req.url).query;
  console.log(query);
  file = require('querystring').parse(query).file;

  // заголовок контента
  res.writeHead(200, {'Content-Type': 'text/plain'});

  // инкремент глобальной переменной, запись в адрес клиента
  for (var i = 0; i<100; i++) {
    res.write(i + '\n');
  }

  // открытие файла и чтение его контента
  var data = fs.readFileSync(file, 'utf8');
  res.write(data);
  res.end();
}).listen('/tmp/node-server-sock');
```

Работа клиента основана на примере кода, который приводится в документации по ядру Node для объекта `http.request` на сайте the Node.js. Объект `http.request` по умолчанию использует объект `http.globalAgent`, который поддерживает объединенные в пул сокеты. Размер этого пула составляет по умолчанию пять сокетов, но его можно перенастроить, изменив значение свойства `agent.maxSockets`.

Клиент принимает от сервера фрагментированные данные, выводя их на консоль. Он также отправляет ответ на сервер, используя для этого две небольшие операции записи, как показано в листинге 3.5.

Листинг 3.5. Подключение к Unix-сокету и вывод полученных данных

```
var http = require('http');
var options = {
   method: 'GET',
   socketPath: '/tmp/node-server-sock',
```

```
    path: "/?file=main.txt"
};
var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('chunk o\' data: ' + chunk);
  });
});
req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});
// запись данных в тело запроса
req.write('data\n');
req.write('data\n');
req.end();
```

Я не использовал с объектом `http.request` асинхронную функцию чтения файла, потому что на момент вызова асинхронной функции соединение уже закрыто и контент файла не возвращается.

Перед тем как завершить данный раздел, касающийся работы с HTTP-модулем в Node, напомню, что базовая функциональность, столь привычная в Apache и других веб-серверах, в HTTP-сервер не встроена. Например, если защитить веб-сайт паролем, Apache-сервер выведет окно с запросом имени пользователя и пароля, а в Node HTTP-сервер этого не сделает. Чтобы добиться в Node аналогичной функциональности, ее нужно специально запрограммировать.

В главе 15 рассматривается SSL-версия HTTP, HTTPS, а также Crypto и TLS/SSL.

UDP-сокеты, или сокеты дейтаграмм

Протокол TCP требует для взаимодействия между двумя конечными точками выделенного соединения. Протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм) этого не требует. Это означает отсутствие гарантии взаимодействия между двумя конечными точками. То есть UDP является менее надежным и устойчивым протоколом по сравнению с TCP. Однако UDP в целом функционирует быстрее, чем TCP, что делает его популярным среди пользователей, работающих в режиме реального времени или применяющих такие технологии, как VoIP (Voice over Internet Protocol — голосовая связь через протокол Интернета), где требования к TCP-соединению могут неблагоприятно влиять на качество сигнала.

Ядро Node поддерживает оба типа сокетов. В двух последних разделах я расскажу о функциональных возможностях TCP, а сейчас настала очередь UDP.

Идентификатором UDP-модуля служит `dgram`:

```
require ('dgram');
```

Для создания UDP-сокета воспользуйтесь методом `createSocket`, передав ему тип сокета (`udp4` или `udp6`). Можно также передать функцию обратного вызова, прослушивающую события. В отличие от TCP-сообщений, UDP-сообщения должны передаваться в виде буферов, а не в виде строк.

В листинге 3.6 содержится код для демонстрации UDP-клиента. В нем доступ к данным осуществляется через объект `process.stdin`, а затем эти данные отправляются в неизменном виде через UDP-сокет. Заметьте, что здесь не нужно устанавливать строковую кодировку, поскольку UDP-сокет принимает только буфер, а данные `process.stdin` *являются* именно буфером. Но нам все же приходится превращать буфер в строку методом `toString`, чтобы получить содержательную строку для вызова метода `console.log`, который действует в качестве повторителя введенных данных.

Листинг 3.6. Дейтаграмма клиента, отправляющего сообщения, набираемые в терминале

```
var dgram = require('dgram');
var client = dgram.createSocket("udp4");
// подготовка к вводу с терминала
process.stdin.resume();
process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
        if (err)
          console.log('error: ' + err);
        else
          console.log('successful');
   });
});
```

UDP-сервер, представленный в листинге 3.7, еще проще, чем клиент. Все, что делает серверное приложение — создает сокет, привязывает его к конкретному порту (8124) и прослушивает событие `message`. При поступлении сообщения приложение выводит его, используя `console.log`, также оно выводит IP-адрес и порт отправителя. Обратите особое внимание на то, что для вывода сообщения указание кодировки не требуется, поскольку происходит автоматическое преобразование буфера в строку.

И хотя привязывать сокет к порту не обязательно, без такой привязки сокет пытался бы прослушивать каждый порт.

Листинг 3.7. Серверный UDP-сокет, привязанный к порту 8124 и прослушивающий этот порт с целью приема сообщений

```
var dgram = require('dgram');
var server = dgram.createSocket("udp4");
server.on ("message", function(msg, rinfo) {
```

```
    console.log("Message: " + msg + " от " + rinfo.address + ":"
                + rinfo.port);
});
server.bind(8124);
```

После отправки или получения сообщения я не вызываю метод `close` ни для клиента, ни для сервера. Тем не менее соединение между клиентом и сервером не поддерживается, поскольку сокеты сохраняют возможность поддерживать взаимодействие путем отправки и получения сообщения.

Потоки ввода-вывода, каналы и построчное чтение

Коммуникационный поток между сокетами, рассматривавшийся в предыдущих разделах, является реализацией основной абстракции интерфейса потоков ввода-вывода. Потоки ввода-вывода могут быть предназначенными для чтения, для записи или для обеих целей, и все они являются экземплярами объекта `EventEmitter` (см. далее раздел «События и объект EventEmitter»).

А в данном разделе важно понять, что все эти коммуникационные потоки, включая `process.stdin` и `process.stdout`, являются реализацией абстракции интерфейса потоков ввода-вывода. Благодаря наличию основного интерфейса всем Node-потокам ввода-вывода присущи следующие функциональные возможности:

- С помощью метода `setEncoding` можно изменять кодировку данных потока.
- Можно проверить, для чего поток предназначен, для чтения, записи или для того и другого.
- Можно перехватывать события потока, например события получения данных или закрытия соединения, и назначать каждому событию функцию обратного вызова.
- Можно приостанавливать и возобновлять поток.
- Можно создать канал передачи данных между потоком чтения и потоком записи.

Последняя возможность еще не рассматривалась. Проще всего продемонстрировать функционирование канала (pipe), открыв REPL-сеанс и введя следующий код:

```
> process.stdin.resume();
> process.stdin.pipe(process.stdout);
```

Далее можно наблюдать, как все, что вводится, тут же выводится на экран.

Если нужно сохранять выходящий поток открытым для продолжающих поступать данных, передайте выходящему потоку ключ `{ end:false }`:

```
process.stdin.pipe(process.stdout, { end : false });
```

Есть еще один дополнительный объект, предоставляющий потокам чтения особую функциональную возможность — `readline`. Модуль `readline` подключается с помощью следующего кода:

```
var readline = require('readline');
```

Модуль `readline` позволяет осуществлять построчное чтение потока. Однако вам следует знать, что при подключении этого модуля Node-программа не завершит свою работу до тех пор, пока вы не закроете интерфейс и поток `stdin`. В документации на сайте Node приводится код запуска и закрытия интерфейса `readline`, этот код я адаптировал для листинга 3.8. Приложение сразу после запуска задает вопрос, а затем выводит ответ. Оно также прослушивает любую «команду», которая в действительности является любой строкой, заканчивающейся символами `\n`. Ели это команда `.leave`, происходит выход из приложения, в противном случае команда просто повторяется, и пользователю выводится приглашение на ввод следующей команды. Прервать работу приложения можно также комбинациями клавиш Ctrl+C или Ctrl+D.

Листинг 3.8. Применение модуля readline для создания простого пользовательского интерфейса, управляемого с помощью команд

```
var readline = require('readline');

// создание нового интерфейса
var interface = readline.createInterface(process.stdin, process.stdout,
                                         null);

// задание вопроса
interface.question(">>What is the meaning of life?  ", function(answer) {
   console.log("About the meaning of life, you said: " + answer);
   interface.setPrompt(">>");
   interface.prompt();
});

// функция для закрытия интерфейса
function closeInterface() {
   console.log('Leaving interface...');
   process.exit();
}
// прослушивание команды .leave
interface.on('line', function(cmd) {
   if (cmd.trim() == '.leave') {
      closeInterface();
      return;
   } else {
      console.log("repeating command: " + cmd);
   }
   interface.setPrompt(">>");
   interface.prompt();
});

interface.on('close', function() {
    closeInterface();
});
```

Вот как выглядит пример сеанса:

```
>>What is the meaning of life? ===
About the meaning of life, you said ===
>>This could be a command
repeating command: This could be a command
>>We could add eval in here and actually run this thing
repeating command: We could add eval in here and actually run this thing
>>And now you know where REPL comes from
repeating command: And now you know where REPL comes from
>>And that using rlwrap replaces this Readline functionality
repeating command: And that using rlwrap replaces this Readline functionality
>>Time to go
repeating command: Time to go
>>.leave
Leaving interface...
```

Все это должно быть уже знакомо. Помните, в главе 2 мы использовали утилиту `rlwrap` для изменения функциональности командной строки в REPL. Чтобы произвести такое изменение, мы применили следующий код:

```
env NODE_NO_READLINE=1 rlwrap node
```

Теперь мы знаем, что именно переключалось этим флагом — он заставлял REPL вместо Node-модуля `readline` использовать для обработки командной строки утилиту `rlwrap`.

После краткого введения в потоковые Node-модули настало время сменить курс и заняться дочерними процессами в Node.

Дочерние процессы

Операционные системы предоставляют доступ к широкому спектру функциональных возможностей, однако большинство из этих возможностей доступно только через командную строку. А чтобы иметь к ним доступ из Node-приложения, требуются *дочерние процессы*.

Node позволяет запустить системную команду в рамках нового дочернего процесса и прослушивать его ввод-вывод. При этом возможна передача аргументов в команду и даже создание канала, передающего результаты выполнения одной команды другой команде. Более подробно эта функциональность рассматривается в следующих нескольких разделах.

Во всех примерах этого раздела, исключая последний, используются Unix-команды. Они работают на платформе Linux, должны работать на Mac, но в командном окне Windows они работать не будут.

Метод child_process.spawn

Для создания дочерних процессов можно воспользоваться четырьмя различными технологиями. Чаще всего используется метод `spawn`. Он запускает команду в новом процессе, передавая ей любое количество аргументов. Следующий код служит для создания дочернего процесса, вызывающего Unix-команду `pwd` с целью вывода контента текущего каталога. Аргументы команде не передаются:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd');

pwd.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

pwd.on('exit', function (code) {
  console.log('child process exited with code ' + code);
});
```

Обратите внимание на события, перехватываемые в потоках `stdout` и `stderr` дочернего процесса. Если ошибки не будет, любой вывод из команды передается потоку `stdout` дочернего процесса, который генерирует для процесса событие `data`. Если произойдет ошибка, как при запуске следующего кода, где команде передается неправильный ключ, она передается в поток `stderr`:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd', ['-g']);
```

Далее поток `stderr` выводит ошибку на консоль:

```
stderr: pwd: invalid option -- 'g'
Try 'pwd --help' for more information.

child process exited with code 1
```

Выход из процесса произошел с кодом 1, что свидетельствует о возникновении ошибки. Код выхода варьируется в зависимости от используемой операционной системы и характера ошибки. При отсутствии ошибки выход из дочернего процесса происходит с кодом 0.

Предыдущий код иллюстрирует направление вывода потокам `stdout` и `stderr` дочернего процесса, а как обстоят дела с потоком `stdin`? В документации к Node, относящейся к дочерним процессам, можно найти пример направления данных потоку `stdin`. Он использовался для эмуляции Unix-канала (`|`), посредством которого результат одной команды тут же направляется на вход другой команды. Я адаптировал пример для своих целей, чтобы продемонстрировать один из моих любимых вариантов использования Unix-канала — просмотр всех подкаталогов,

начиная с локального каталога, при поиске файла, в имени которого имеется конкретное слово (в данном случае — `test`):

```
find . -ls | grep test
```

В листинге 3.9 эта функциональность реализуется в виде дочернего процесса. Обратите внимание, что первая команда, выполняющая поиск, получает два аргумента, вторая команда получает один аргумент — элемент, передаваемый из потока `stdin` посредством пользовательского ввода. Также обратите внимание на то, что в отличие от примера, приведенного в документации к Node, кодировка потока `stdout` дочернего процесса изменяется с помощью функции `setEncoding`. В противном случае при выводе данных они были бы представлены в виде буфера.

Листинг 3.9. Использование дочерних процессов для поиска файлов в подкаталогах по заданному аргументу test

```
var spawn = require('child_process').spawn,
    find = spawn('find',['.','-ls']),
    grep = spawn('grep',['test']);
grep.stdout.setEncoding('utf8');
// направление результатов поиска в адрес grep
find.stdout.on('data', function(data) {
   grep.stdin.write(data);
});
// а теперь запуск grep и вывод результатов
grep.stdout.on('data', function (data) {
  console.log(data);
});
// обработка ошибки для обоих процессов
find.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
// и завершение обработки для обоих процессов
find.on('exit', function (code) {
  if (code !== 0) {
    console.log('find process exited with code ' + code);
  }
  // продолжение обработки и завершение процесса grep
  grep.stdin.end();
});
grep.on('exit', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

При запуске приложения вы получите список всех файлов, в именах которых содержится строка `test`, находящихся в текущем каталоге и любых его подкаталогах.

До сих пор все примеры приложений одинаково работали как в Node 0.8, так и в Node 0.6. Листинг 3.9 является исключением, поскольку в нижележащем прикладном программном интерфейсе произошло изменение.

В Node 0.6 событие выхода `exit` не будет выдаваться, пока существует дочерний процесс и пока не закрыты все STDIO-каналы. В Node 0.8 событие выдается сразу же по завершении дочернего процесса. Это вызовет сбой приложения, поскольку канал ввода-вывода дочернего процесса, относящегося к `grep`, закрывается при попытке обработки своих данных. Чтобы приложение работало в Node 0.8, ему нужно прослушивать не событие `exit`, а событие `close`, относящееся к дочернему процессу `find`:

```
// и выход из обработки для обоих процессов
find.on('close', function (code) {
  if (code !== 0) {
    console.log('find process exited with code ' + code);
  }
  // продолжение обработки и завершение процесса grep
  grep.stdin.end();
});
```

В Node 0.8 событие `close` выдается при выходе из дочернего процесса и при закрытии всех STDIO-каналов.

Методы child_process.exec и child_process.execFile

Кроме порождения дочернего процесса для выполнения команды в оболочке и буферизации результатов можно также использовать методы `child_process.exec` и `child_process.execFile`. Единственным различием между `child_process.exec` и `child_process.execFile` является то, что метод `execFile` запускает не команду, а приложение в файле.

Первым параметром двух методов является либо команда, либо файл и его размещение, вторым параметром — ключ команды, третьим — функция обратного вызова.

Функция обратного вызова получает три аргумента: `error`, `stdout` и `stderr`. Если ошибки не происходит, данные буферизуются в `stdout`.

Пусть исполняемый файл имеет такой контент:

```
#!/usr/local/bin/node
console.log(global);
```

Тогда следующее приложение выводит буферизированные результаты:

```
var execfile = require('child_process').execFile,
    child;
child = execfile('./app.js', function(error, stdout, stderr) {
  if (error == null) {
    console.log('stdout: ' + stdout);
  }
});
```

Метод child_process.fork

Последним методом дочернего процесса является `child_process.fork`. Это вариация метода `spawn` для порожденных Node-процессов.

Обособление процесса, порожденного методом `child_process.fork`, от других процессов вызвано тем, что он фактически является каналом связи, установленным дочерним процессом. Однако здесь следует отметить, что каждый процесс требует полноценного нового экземпляра движка V8, что отнимает как время, так и память.

В документации к Node, касающейся fork, есть несколько хороших примеров использования этого метода.

Запуск приложения дочернего процесса в Windows

Ранее я уже предупреждал, что дочерние процессы, в которых активизируются системные Unix-команды, в Windows не работают, и наоборот. При всей очевидности происходящего, не все еще знают, что в отличие от JavaScript-приложений в браузерах, Node-приложения могут вести себя в разных средах по-разному.

До недавнего времени двоичная установка Node для Windows вообще не обеспечивала доступ к дочерним процессам. В Windows вам также придется запускать нужную команду через интерпретатор командной строки `cmd.exe`.

Запуск команды в Windows иллюстрирует листинг 3.10. Windows-приложение `cmd.exe` служит для создания листинга каталога, который затем выводится на консоль с помощью обработчика события `data`.

Листинг 3.10. Запуск дочернего процесса в Windows

```
var cmd = require('child_process').spawn('cmd', ['/c', 'dir\n']);
cmd.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
});
cmd.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});
cmd.on('exit', function (code) {
    console.log('child process exited with code ' + code);
});
```

Ключ `/c`, переданный `cmd.exe` в качестве первого аргумента, заставляет выполнить команду, а затем завершить процесс. Без этого ключа приложение не работает. А вот ключ `/K` передавать категорически противопоказано, поскольку он заставит `cmd.exe` выполнить приложение и остаться в этом состоянии, не завершив приложение.

Дополнительные примеры дочерних процессов есть в главах 9 и 12.

Разрешение имен доменов и обработка URL-адресов

Модуль DNS (Domain Name System – система доменных имен) обеспечивает разрешение имен доменов, используя библиотеку c-ares, написанную на языке C и поддерживающую асинхронные DNS-запросы. Он используется в Node наряду с несколькими другими модулями и может применяться в приложениях, которым требуется находить домены или IP-адреса.

Для нахождения IP-адреса заданного домена нужно вызвать метод `dns.lookup` и вывести возвращенный IP-адрес:

```
var dns = require('dns');
dns.lookup('burningbird.net',function(err,ip) {
   if (err) throw err;
   console.log(ip);
});
```

Метод `dns.reverse` возвращает массив доменных имен для заданного IP-адреса:

```
dns.reverse('173.255.206.103', function(err,domains) {
domains.forEach(function(domain) {
  console.log(domain);
  });
});
```

Метод `dns.resolve` возвращает массив записей заданного типа, например `A`, `MX`, `NS` и т. д. В следующем коде ведется поиск сервера имен доменов для моего доменного имени burningbird.net:

```
var dns = require('dns');
dns.resolve('burningbird.net', 'NS', function(err,domains) {
domains.forEach(function(domain) {
  console.log(domain);
  });
});
```

Он возвращает следующее:

```
ns1.linode.com
ns3.linode.com
ns5.linode.com
ns4.linode.com
```

В листинге 1.3 (см. главу 1) был использован модуль URL. Этот простой модуль обеспечивает синтаксический разбор URL-адреса и возвращает объект со всеми компонентами URL-адреса. Например:

```
var url = require('url');
var urlObj = url.parse('http://examples.burningbird.net:8124/?file=main');
```

Синтаксический разбор этого URL-адреса позволяет возвратить следующий JavaScript-объект:

```
{ protocol: 'http:',
  slashes: true,
  host: 'examples.burningbird.net:8124',
  port: '8124',
  hostname: 'examples.burningbird.net',
  href: 'http://examples.burningbird.net:8124/?file=main',
  search: '?file=main',
  query: 'file=main',
  pathname: '/',
  path: '/?file=main' }
```

Затем к каждому из компонентов может быть организован отдельный доступ:

```
var qs = urlObj.query; // получение строки запроса
```

Вызов метода `URL.format` обеспечивает обратную операцию:

```
console.log(url.format(urlObj)); // возвращение оригинального URL-адреса
```

Модуль URL часто используется с модулем Query String. Последний является простым вспомогательным модулем, предоставляющим функциональность для синтаксического разбора полученной строки запроса или для подготовки строки с целью ее использования в качестве строки запроса.

Для разбивки строки запроса на пары ключ-значение служит метод `querystring.parse`. Следующий код приводит к получению JavaScript-объекта:

```
var vals = querystring.parse('file=main&file=secondary&type=html");
```

Этот JavaScript-объект позволяет получить простой доступ к отдельным значениям строки запроса:

```
{ file: [ 'main', 'secondary' ], type: 'html' }
```

Поскольку `file` задается в строке запроса дважды, в массив группируются оба значения, к каждому из которых можно получить отдельный доступ:

```
console.log(vals.file[0]); // возвращает main
```

Объект можно также преобразовать в строку запроса, используя метод `querystring.stringify`:

```
var qryString = querystring.stringify(vals)
```

Модуль Utilities и объектное наследование

Модуль Utilities предлагает несколько полезных функций. Этот модуль подключается с помощью следующего кода:

```
var util = require('util');
```

Модуль Utilities можно использовать для тестирования принадлежности объекта к массиву (`util.isArray`) или к регулярному выражению (`util.isRegExp`), а также для форматирования строки (`util.format`). Новое экспериментальное добавление

к модулю предоставляет функциональность извлечения данных из потока чтения (`util.pump`):

```
util.pump(process.stdin, process.stdout);
```

Однако я не стану набирать этот код в REPL, поскольку все, что вы с этого момента наберете, будет по мере набора дублироваться, что вряд ли вам понравится.

Я часто использую метод `util.inspect` для получения строкового представления объекта. Я считаю, что это отличный способ выяснения дополнительных сведений об объекте. Первым аргументом (обязательным) является объект, вторым (необязательным) — указание на вывод неисчислимых свойств, третьим (необязательным) — количество (глубина) рекурсий объекта, и четвертым (также необязательным) — указание на стилизацию вывода в цветах ANSI. Если присвоить глубине значение `null`, рекурсия будет длиться столько, сколько нужно для полного изучения объекта (по умолчанию рекурсия проводится два раза). Исходя из собственного опыта, я бы предостерег вас от использования значения `null` в качестве глубины, поскольку полученный при этом объем выводимых данных может быть слишком большим.

Метод `util.inspect` можно использовать в REPL, но я рекомендую простое приложение, например:

```
var util = require('util');
var jsdom = require('jsdom');
console.log(util.inspect(jsdom, true, null, true));
```

При его запуске создайте канал для направления результата в файл:

```
node inspectjsdom.js > jsdom.txt
```

Теперь можно проводить инспекцию интерфейса объекта в любое время. И опять хочу напомнить, что при указании в качестве глубины рекурсии значения `null` следует ожидать на выходе большого по объему файла.

Модуль Utilities предоставляет ряд других методов, но самым востребованным, скорее всего, окажется метод `util.inherits`. Он принимает два параметра: конструктор и суперконструктор. В результате конструктор наследует всю функциональность от суперконструктора.

Все нюансы, связанные с использованием метода `util.inherits`, иллюстрирует листинг 3.11. Затем следует разъяснение.

Листинг 3.11 и его разъяснение затрагивают некоторую базовую функциональность JavaScript, с которой вы уже можете быть знакомы. Тем не менее важно, чтобы все читатели, завершая чтение данного раздела, имели одинаковое представление о происходящем.

Листинг 3.11. Включение объектного наследования с помощью метода util.inherits

```
var util = require('util');
// определение исходного объекта
```

```
function first() {
  var self = this;
  this.name = 'first';
  this.test = function() {
    console.log(self.name);
   };
}
first.prototype.output = function() {
   console.log(this.name);
}
// наследование из first
function second() {
  second.super_.call(this);
  this.name = 'second';
}
util.inherits(second,first);
var two = new second();
function third(func) {
   this.name = 'third';
   this.callMethod = func;
}
var three = new third(two.test);
// при всех трех вызовах должно быть выведено "second"
two.output();
two.test();
three.callMethod();
```

Приложение создает три объекта с именами `first`, `second` и `third`.

У объекта `first` есть два метода: `test` и `output`. Метод `test` определяется непосредственно в объекте, а метод `output` добавляется позже через объект-прототип. Поводом для применения обеих технологий при определении метода объекта стала необходимость демонстрации важных аспектов наследования с помощью метода `util.inherits` (или, точнее, наследования с помощью JavaScript, инициируемого методом `util.inherits`).

Второй объект содержит следующую строку:

```
second.super_.call(this);
```

Если убрать эту строку из конструктора второго объекта, любой вызов метода `output` в отношении второго объекта будет успешным, а вот вызов метода `test` приведет к ошибке и заставит Node-приложение завершить работу с сообщением о том, что переменная `test` не определена.

Метод `call` устанавливает связь конструкторов двух объектов, гарантируя вызов суперконструктора наряду с конструктором. Суперконструктор является конструктором для наследуемого объекта.

Вызов суперконструктора нужен, поскольку метод `test` не существует, пока не создан объект `first`. Однако для метода `output` метод `call` не требуется, поскольку он определяется напрямую в объекте-прототипе объекта `first`. Когда объект `second` наследует свойства объекта `first`, он также наследует вновь добавленный метод.

Если заглянуть внутрь метода `util.inherits`, можно увидеть фрагмент определения `super_`:

```
exports.inherits = function(ctor, superCtor) {
  ctor.super_ = superCtor;
  ctor.prototype = Object.create(superCtor.prototype, {
    constructor: {
      value: ctor,
      enumerable: false,
      writable: true,
      configurable: true
    }
  });
};
```

При вызове `util.inherits` фрагмент `super_` в качестве свойства присваивается объекту `second`:

```
util.inherits (second, first);
```

Третий объект, имеющийся в приложении, `third`, также имеет свойство `name`. Оно не наследуется ни у `first`, ни у `second`, но ожидает функцию, передаваемую ему при создании. Эта функция присваивается его собственному свойству `callMethod`. Когда код создает экземпляр этого объекта, конструктору передается метод `test` экземпляра объекта `two`:

```
var three = new third(two.test);
```

При вызове `three.callMethod` выводится строка `"second"`, а не строка `"third"`, как могло показаться на первый взгляд. И здесь вступает в силу ссылка `self` в объекте `first`.

В JavaScript `this` представляет собой контекст объекта, который может меняться. Единственным способом сохранить данные для метода объекта является присваивание `this` переменной объекта, в данном случае — `self`, а затем использовать переменную внутри любых функций объекта.

Запуск этого приложения приведет к следующему результату:

```
second
second
second
```

Многое из всего этого вам, скорее всего, знакомо по разработке JavaScript-программ на клиентской стороне, но важно понимать, как модуль Utilities участвует в наследовании. В следующем разделе дается обзор функциональности Node-объекта `EventEmitter`, которые в значительной степени определяются рассмотренным режимом наследования.

События и объект EventEmitter

Если внимательно присмотреться ко многим объектам ядра Node, то там обнаружится объект `EventEmitter`. Как только выясняется, что объект генерирует

событие, которое обрабатывается методом on, значит, здесь не обошлось без объекта EventEmitter. Понимание того, как работает объект EventEmitter и как его использовать, весьма важно в Node-разработки.

Именно объект EventEmitter обеспечивает Node-объектам асинхронную обработку событий. Для демонстрации его базовой функциональности мы запустим небольшое тестовое приложение.

Сначала подключим модуль Events:

```
var events = require('events');
```

Затем создадим экземпляр объекта EventEmitter:

```
var em = new events.EventEmitter();
```

Использование только что созданного экземпляра EventEmitter решает две важные задачи: назначает событию обработчик и генерирует само событие. Обработчик события on активизируется при возникновении конкретного события. Первым параметром метода является имя события, вторым — функция его обработки:

```
em.on('someevent', function(data) { ... });
```

В соответствии с неким критерием событие генерируется для объекта методом emit:

```
if (некий_критерий) {
   en.emit('data');
}
```

В листинге 3.12 создается экземпляр EventEmitter, который генерирует событие timed каждые три секунды. В функции обработки этого события на консоль выводится сообщение с показанием счетчика.

Листинг 3.12. Очень простой тест функционирования объекта EventEmitter

```
var eventEmitter = require('events').EventEmitter;
var counter = 0;
var em = new eventEmitter();
setInterval(function() { em.emit('timed', counter++); }, 3000);
em.on('timed', function(data) {
  console.log('timed ' + data);
});
```

При запуске приложения на консоль выводится сообщение для события timed до тех пор, пока приложение не завершит работу.

Это довольно интересный, но не особо полезный пример. Нам нужна возможность наделять существующие объекты функциональностью объекта EventEmitter, а не использовать экземпляры EventEmitter во всех приложениях.

Для добавления требуемой функциональности к объекту служит метод util.inherits, рассмотренный в предыдущем разделе:

```
util.inherits(someobj, EventEmitter);
```

Благодаря методу util.inherits метод emit можно вызывать внутри методов объекта, а код обработчиков событий — для экземпляров объекта:

```
someobj.prototype.somemethod = function() { this.emit('event'); };
...
someobjinstance.on('event', function() { });
```

Вместо попыток разгадать, как работает `EventEmitter` в абстрактном смысле, давайте перейдем к листингу 3.13, в котором представлен работоспособный пример объекта, наследующего функциональность `EventEmitter`. В приложении создается новый объект `inputChecker`. Конструктор получает два значения: имя человека и имя файла. Он присваивает имя человека переменной объекта, а также создает ссылку на поток записи и принадлежащий методу `createWriteStream` модуля File System (дополнительные сведения о модуле File System см. во врезке).

ПОТОКИ ЧТЕНИЯ И ЗАПИСИ

Имеющийся в Node модуль File System (fs) позволяет открыть файл для чтения и записи, искать конкретные файлы для новых действий и управлять каталогами. Он также предоставляет возможность использовать потоки чтения и записи.

Поток чтения создается методом fs.createReadStream, которому передаются имя и путь к файлу и другие необязательные параметры. Поток записи создается методом fs.createWriteStream, которому также передается имя и путь к файлу.

Вы будете использовать потоки чтения и записи вдобавок к традиционным методам чтения и записи при чтении данных из файла и при записи их в файл на основе событий, которые могут инициировать операции чтения и записи довольно часто. Потоки открываются в фоновом режиме, и операции чтения (и записи) выстраиваются в очередь.

У объекта имеется также метод `check`, который проверяет поступающие данные на наличие определенных команд. Одна команда (`wr:`) генерирует событие записи, еще одна команда (`en:`) — событие завершения. Если команда отсутствует, генерируется событие `echo`. Экземпляр объекта предоставляет обработчики для всех трех событий. Для события записи осуществляется запись в выходной файл, при вводе, не содержащем команды, осуществляется дублирование этого ввода, и для события завершения работы приложения заканчивается с помощью метода `process.exit`.

Весь ввод поступает от стандартного потока ввода (`process.stdin`).

Листинг 3.13. Создание объекта на основе событий, который наследует свойства у объекта EventEmitter

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');
function inputChecker (name, file) {
   this.name = name;
```

```
    this.writeStream = fs.createWriteStream('./' + file + '.txt',
       {'flags' : 'a',
        'encoding' : 'utf8',
        'mode' : 0666});
};
util.inherits(inputChecker,eventEmitter);
inputChecker.prototype.check = function check(input) {
  var command = input.toString().trim().substr(0,3);
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));
  } else if (command == 'en:') {
     this.emit('end');
  } else {
     this.emit('echo',input);
  }
};
// проверка нового объекта и обработка событий
var ic = new inputChecker('Shelley','output');
ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
ic.on('end', function() {
   process.exit();
});
process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

Функции объекта `EventEmitter` выделены в листинге полужирным шрифтом. Обратите внимание на то, что функциональные возможности включают также метод обработки события `process.stdin.on`, поскольку `process.stdin` является одним из многочисленных Node-объектов, который наследует свойства у объекта `EventEmitter`.

Мы не должны выстраивать конструкторы в цепочку от нового объекта к `EventEmitter`, как демонстрировалось в одном из предыдущих примеров, касающихся `util.inherits`, поскольку нужная нам функциональность (`on` и `emit`) относится к методам прототипа, а не к свойствам экземпляра объекта.

Метод `on` на самом деле является сокращением для вызова метода `EventEmitter.addListener`, который принимает точно такие же параметры. Например:

```
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

Этот код является абсолютным эквивалентом следующего кода:

```
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

А с помощью такого кода можно прослушивать только первое событие:

```
ic.once(event, function);
```

Когда количество слушателей события оказывается больше десяти, то по умолчанию вы получаете предупреждение. Для изменения количества слушателей нужно их число передать функции `setMaxListeners`. Для неограниченного количества слушателей используется значение нуль (`0`).

Объект `EventEmitter` требуется многим объектам ядра Node, а также модулям сторонних разработчиков. В главе 4 будет показано, как превратить в модуль код из листинга 3.13.

4 Модульная система Node

Базовая реализация Node является максимально простой. Чтобы не встраивать непосредственно в Node все возможные компоненты, разработчики предлагают получать дополнительную функциональность посредством модулей.

Модульная система Node сделана по образцу и подобию модульной системы CommonJS как средства создания взаимодействующих модулей. В основу положен договор, которого придерживаются разработчики, чтобы их модули могли работать вместе с другими модулями.

В Node реализованы следующие требования, присущие CommonJS:

- Включена поддержка функции `require`, которая принимает идентификатор модуля и возвращает экспортируемый прикладной программный интерфейс.
- Именем модуля является строка символов, которая может включать в себя начальные слэши (для идентификации пути).
- Модуль должен конкретным образом экспортировать все, что должно быть видимо за его пределами.
- Переменные в модуле являются закрытыми.

В нескольких следующих разделах показано, как эти требования поддерживаются в Node.

Загрузка модуля с помощью инструкции require и путей по умолчанию

Node поддерживает простую систему загрузки модулей: между файлом и модулем есть однозначное соответствие.

Для подключения модуля в Node-приложение нужно воспользоваться инструкцией `require`, передав ей строку с идентификатором модуля:

```
var http = require ('http');
```

Можно также подключить конкретный объект, а не все объекты модуля:

```
var spawn = require('child_process').spawn;
```

Можно загружать модули ядра, то есть модули, входящие в состав Node, или модули из папки node_modules, просто предоставив идентификатор модуля, например `http` для модуля HTTP. Модули, не являющиеся частью ядра или не включенные в папку node_modules, должны для индикации пути содержать в своем имени ведущие слэши. Например, в следующей инструкции `require` Node ожидает найти модуль по имени mymodule.js в том же каталоге, где находится Node-приложение:

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

Или же можно использовать полный путь:

```
require ('/home/myname/myapp/mymodule.js');
```

Файлы модулей могут иметь расширение .js, .node или .json. Расширение .node означает, что файл является откомпилированным двоичным файлом, а не текстом, содержащим JavaScript-код.

Модули ядра Node имеют более высокий приоритет, чем внешние модули. При попытке загрузки нестандартного модуля с именем `http` Node загружает ту версию модуля HTTP, которая находится в ядре. Вам нужно будет либо предоставить другой идентификатор модуля, либо указать полный путь к модулю.

Папка node_modules ранее уже упоминалась. Если указать идентификатор модуля, не предоставив пути, имея в виду тот модуль, который не входит в состав ядра, Node сначала ищет папку node_modules, локальную по отношению к приложению, а затем ищет модуль в этой папке. Если модуль найден не будет, Node начнет поиск в родительском каталоге папки node_modules и модуля и т. д.

Пусть модуль имеет имя `mymodule`, а приложение находится в подкаталоге, имеющем такой путь:

```
/home/myname/myprojects/myapp
```

В этом случае Node ищет модуль в следующих местах по очереди:

- /home/myname/myprojects/myapp/node_modules/mymodule.js;
- /home/myname/myprojects/node_modules/mymodule.js;
- /home/myname/node_modules/mymodule.js;
- /node_modules/mymodule.js.

Node может оптимизировать поиск в зависимости от того, где находится файл, выдавший инструкцию `require`. Например, если файл, выдавший инструкцию `require`, сам является модулем в подкаталоге папки node_modules, Node начинает поиск требуемого модуля в самой верхней папке node_modules.

Существует еще пара дополнительных инструкций `require`: `require.resolve` и `require.cache`. Метод `require.resolve` ведет поиск заданного модуля, но вместо его загрузки просто возвращает разрешенное имя файла. Объект `resolve.cache` содержит кэшированную версию всех загруженных модулей. При попытке загрузить модуль еще раз в один и тот же контекст модуль загружается из кэша. Если нужно осуществить новую загрузку принудительно, следует удалить запись из кэша.

Предположим, путь имел следующий вид:

```
var circle = require('./circle.js');
```

В этом случае удалите запись с помощью такого кода:

```
delete require.cache('./circle.js');
```

Это инициирует перезагрузку модуля при следующем вызове инструкции `require`.

Внешние модули и диспетчер пакетов в Node

Как уже упоминалось, большую часть функциональности Node получает от модулей сторонних разработчиков. К ним относятся модули маршрутизации, модули для работы с реляционными или документными системами баз данных, модули шаблонов, тестовые модули и даже модули платежных шлюзов.

Хотя какой-либо формальной системы разработки Node-модулей не существует, поощряется выкладывание разработчиками своих модулей на хостинге GitHub. Кроме того, Node-модули можно найти на следующих ресурсах:

- Реестр npm (http://search.npmjs.org/).
- Вики для Node-модулей (https://github.com/joyent/node/wiki/modules).
- Node-инструментарий (http://toolbox.no.de/).
- Nipster! (http://eirikb.github.com/nipster/).

Как уже отмечалось, модули можно в первом приближении разделить по типам, таким как маршрутизаторы, модули для работы с базами данных, модули шаблонов, платежные шлюзы и т. д.

Чтобы воспользоваться модулем, нужно загрузить его исходный код с сайта GitHub (или оттуда, где он находится), а затем установить этот модуль вручную в среду вашего приложения. У многих модулей имеются базовые инструкции по установке или, как минимум, установочные требования можно выяснить, изучив файлы и каталоги, включенные в модуль. Но есть намного более простой путь установки Node-модуля — использовать диспетчер Node-пакетов (Node Package Manager, npm).

Адрес сайта npm — http://npmjs.org/. Основные инструкции по npm можно найти по адресу http://npmjs.org/doc/README.html. Разработчикам Node-модулей стоит прочитать раздел «Developers» руководства по npm, который можно найти по адресу http://npmjs.org/doc/developers.html. Полезные публикации, объясняющие разницу между локальной и глобальной установкой, можно найти по адресу http://blog.nodejs.org/2011/03/23/npm-1-0-global-vs-local-installation/.

Современные копии включают в себя npm, тем не менее, чтобы убедиться в наличии диспетчера, можно набрать `npm` в командной строке той же среды, которая используется для доступа к Node.

Для просмотра npm-команд воспользуйтесь следующей командой:

```
$ npm help npm
```

Модули могут устанавливаться глобально или локально. Локальная установка предпочтительнее, если ведется работа над проектом, но никто из ваших коллег не нуждается в доступе к этим модулям. Локальная установка, предлагаемая по умолчанию, помещает модуль в текущее место в каталог node_modules:

```
$ npm install modulename
```

Например, для установки модуля Connect, который является весьма популярной связующей структурой, используется следующая команда:

```
$ npm install connect
```

Диспетчер пакетов не только устанавливает модуль Connect, он, как показано на рис. 4.1, находит также модули, от которых зависит работа модуля Connect, и устанавливает эти модули.

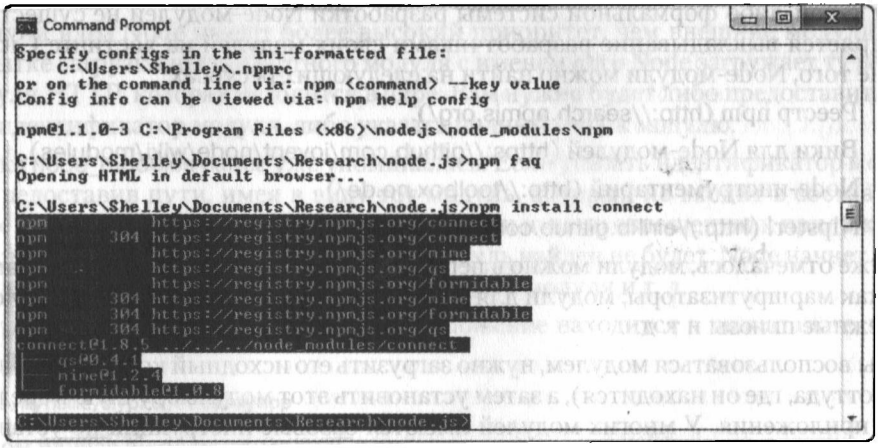


Рис. 4.1. Установка модуля Connect в Windows 7 с помощью npm

После установки модуль можно найти в подкаталоге node_modules своего локального каталога. Все модули, от которых зависит работа данного модуля, устанавливаются в его каталог node_modules.

Если нужна глобальная установка пакета, следует воспользоваться ключом `-g` или `--global`:

```
$ npm -g install connect
```

Эти примеры позволяют установить пакеты, зарегистрированные на сайте npm. Можно также установить модуль, находящийся в папке файловой системы или в tarball-дистрибутиве, имеющемся на локальной системе или извлекаемом посредством URL-адреса:

```
npm install http://somecompany.com/somemodule.tgz
```

Если пакет имеет версии, можно установить конкретную версию:

```
npm install modulename@0.1
```

Диспетчер пакетов может также работать с Git, как показано в приложении к книге.

Можно также установить старого друга, модуль jQuery:

```
npm install jquery
```

Теперь вы можете воспользоваться при разработке своих Node-приложений привычным синтаксисом.

Если модуль больше не нужен, его можно удалить:

```
npm uninstall modulename
```

Следующая команда предписывает npm провести проверку наличия новых модулей и выполнить обновление, если таковые найдутся:

```
npm update
```

Или же можно обновить отдельно взятый модуль:

```
npm update modulename
```

Если нужно только лишь проверить наличие устаревших пакетов, воспользуйтесь следующей командой:

```
npm outdated
```

Эту команду можно также запустить для отдельно взятого модуля.

Вывести список установленных пакетов и тех модулей, от которых зависит их работа, можно с помощью команды `list`, `ls`, `la` или `ll`:

```
npm ls
```

Команды `la` и `ll` предлагают расширенные описания. Следующий текст я получил, запустив команду `npm ll` на своей машине, работающей под управлением Windows 7:

```
C:\Users\Shelley>npm ls ll
npm WARN jsdom >= 0.2.0 Unmet dependency in C:\Users\Shelley\node_modules\html5
C:\Users\Shelley
├── async@0.1.15
├── colors@0.6.0-1
├── commander@0.5.2
├─┬ connect@1.8.5
│ ├── formidable@1.0.8
│ ├── mime@1.2.4
│ └── qs@0.4.1
├─┬ html5@v0.3.5
```

продолжение ⇩

```
| ├── UNMET DEPENDENCY jsdom >= 0.2.0
| ├── opts@1.2.2
| └─┬ tap@0.0.13
|   ├── inherits@1.0.0
|   ├── tap-assert@0.0.10
|   ├── tap-consumer@0.0.1
|   ├── tap-global-harness@0.0.1
|   ├── tap-harness@0.0.3
|   ├── tap-producer@0.0.1
|   ├── tap-results@0.0.2
|   ├─┬ tap-runner@0.0.7
|   | ├── inherits@1.0.0
|   | ├── slide@1.1.3
|   | ├── tap-assert@0.0.10
|   | ├── tap-consumer@0.0.1
|   | ├── tap-producer@0.0.1
|   | ├── tap-results@0.0.2
|   | └── yamlish@0.0.3
|   ├── tap-test@0.0.2
|   └── yamlish@0.0.2
└─┬ optimist@0.3.1
  └── wordwrap@0.0.2
```

Обратите внимание на неудовлетворенную зависимость модуля HTML5. Этот модуль требует старую версию библиотеки JSDOM. Чтобы исправить положение, я установил необходимую версию модуля:

```
npm install jsdom@0.2.0
```

Вы также можете непосредственно установить все зависимости, воспользовавшись ключом `-d`. Например, в каталоге модуля наберите следующую команду:

```
npm install -d
```

Если нужно установить версию модуля, которая еще не была выложена в реестр npm, ее можно установить непосредственно из репозитория Git:

```
npm install https://github.com/visionmedia/express/tarball/master
```

Но хочу предостеречь: я обнаружил, что при установке еще не выпущенной версии модуля и проведении после этого обновления с помощью команды `npm update` используемая вами версия может быть заменена версией из реестра npm.

Чтобы посмотреть, какие модули установлены глобально, воспользуйтесь командой:

```
npm ls -g
```

Получить дополнительные сведения о вашей npm-установке можно с помощью команды `config`. Следующая команда выводит список конфигурационных параметров npm:

```
npm config list
```

А с помощью следующей команды можно получить еще более подробное представление обо всех настройках конфигурации:

```
npm config ls -l
```

Изменить или удалить конфигурационные параметры можно с помощью командной строки:

```
npm config delete keyname
npm config set keyname value
```

Кроме того, конфигурационный файл можно редактировать:

```
$ npm config edit
```

Настоятельно рекомендую оставить настройки конфигурации npm в покое, пока не будете абсолютно уверены в эффективности изменений.

Можно искать модуль, используя те поисковые термины, которые, по вашему мнению, могут вернуть наилучшие варианты:

```
npm search html5 parser
```

При первом поиске npm создает индекс, что может занять несколько минут. Но когда создание индекса завершится, вы получите список всех доступных модулей, соответствующих предоставленному поисковому термину или терминам. Поисковые термины `html5` и `parser` возвращают только два модуля: HTML5 и HTML parser, включающие поддержку SVG и MathML, а также Fabric, объектной модели с поддержкой HTML5 Canvas и парсера SVG-to-Canvas.

Веб-сайт npm предоставляет реестр модулей, который можно просмотреть, и актуальный листинг зависимостей, то есть тех модулей, которые наиболее часто используются другими модулями или Node-приложениями. В следующем разделе будет рассмотрена выборка этих модулей.

Другие npm-команды рассмотрены в данной главе чуть позже, в разделе «Создание собственного пользовательского модуля».

Поиск модулей

Хотя Node.js находится в активном применении всего несколько лет, в рамках этого проекта уже проделан большой объем работы. Если посмотреть на вики-страницу модулей Node.js, можно найти весьма существенное количество модулей. Хорошо, что среди этого количества можно найти массу полезных модулей, реализующих нужную вам функциональность. Но плохо то, что определить, какой из модулей

лучше использовать, довольно трудно, иными словами, сложно понять, какой из модулей «лучший в своем классе».

Такие поисковые инструменты, как Google, могут дать вам ценную идею насчет популярности тех или иных модулей. Например, когда я исследовал модули связующих программ и структур, мне довольно быстро стало очевидно, что весьма популярными являются модули Connect и Express.

Кроме того, при изучении записей в реестре GitHub можно увидеть, насколько активно эти модули поддерживаются и соответствуют состоянию текущей копии Node. Вот еще один пример. Я проверял инструментарий под названием Apricot, предназначенный для анализа HTML и рекомендуемый в документации к Node, но заметил, что он не обновлялся в течение некоторого времени, а когда я попытался воспользоваться этим модулем, обнаружилось, что он не работает с моей копией Node (по крайней мере, не работал на момент написания этой книги).

Во многих модулях предоставляются примеры приложений, краткий тест которых позволяет понять, можно ли использовать модуль в вашей текущей среде.

Как уже упоминалось, сайт документации к Node предлагает список рекомендуемых модулей сторонних разработчиков, начиная с диспетчера пакетов, который ныне включен в установку Node. Однако веб-сайт npm и его реестр модулей позволяют лучше понять, какие модули используются в большинстве приложений.

На странице реестра npm можно вести поиск модулей, но можно также просматривать список модулей, от которых «наиболее зависимы» другие модули, используемые либо в других модулях, либо в Node-приложениях. На момент написания книги наиболее популярными модулями были следующие:

Underscore

Предоставляет полезные JavaScript-функции общего назначения.

Coffee-script

Позволяет использовать язык CoffeeScript, компилируемый в JavaScript.

Request

Упрощенный клиент HTTP-запросов.

Express

Инфраструктура.

Optimist

Предлагает упрощенный синтаксический разбор ключей.

Async

Предоставляет функции и схемы для асинхронного кода.

Connect

Связующее программное обеспечение.

Colors

Добавляет цвета на консоли.

Uglify-js

Парсер и компрессор-форматировщик.

Socket.IO

Позволяет вести обмен данными между клиентом и сервером в реальном времени.

Redis

Клиент Redis.

Jade

Движок шаблонов.

Commander

Модуль для программ командной строки.

Mime

Предлагает поддержку MIME-расширений файлов.

JSDOM

Реализует W3C DOM.

Некоторые из этих модулей рассматриваются в следующих главах, но три из них я хочу рассмотреть уже сейчас, во-первых, потому что они дают возможность лучше разобраться в том, как работает Node, во-вторых, потому что они особенно полезны. Вот эти три модуля:

- Colors;
- Optimist;
- Underscore.

Модуль Colors: чем проще, тем лучше

Colors — один из самых простых модулей. Его можно использовать для предоставления различных цветовых и стилевых эффектов выводу `console.log`, и это все, на что он способен. Тем не менее это неплохой пример эффективного модуля, поскольку он прост в использовании, нацелен на предоставление всего одной услуги и прекрасно с этим справляется.

Тестирование модуля — весомый аргумент для использования REPL. Чтобы проверить модуль Colors, установите его с помощью npm:

```
$ npm install colors
```

Откройте новый REPL-сеанс и подключите библиотеку `colors`:

```
> var colors = require('colors');
```

Поскольку модуль Colors включен в подкаталог `node_modules` текущего места, Node может найти его довольно быстро.

А теперь попробуйте что-нибудь вывести на экран, например следующее:

```
console.log('This Node kicks it!'.rainbow.underline);
```

В результате вы получите красочный подчеркнутый текст своего сообщения. Стиль применяется только к одному сообщению, а для еще одного сообщения придется применить еще один стиль.

Если вы работали с jQuery, то узнаете цепочку, позволяющую получить комбинацию эффектов. В примере используются два эффекта: эффект `underlined` относится к шрифту, а эффект `rainbow` — к цвету шрифта.

Попробуйте применить другие эффекты, на этот раз `zebra` и `bold`:

```
console.log('We be Nodin'.zebra.bold);
```

Вы можете изменить стиль отдельных фрагментов выводимого на консоль сообщения:

```
console.log('rainbow'.rainbow, 'zebra'.zebra);
```

А для чего может пригодиться такой модуль, как Colors? Он позволяет задать форматирование для различных событий, например один цвет использовать для ошибок в одном модуле, другой цвет или эффект — для предупреждений во втором модуле и т. д. Для этого можно взять готовые параметры модуля Colors или создать собственную нестандартную тему:

```
> colors.setTheme({
..... mod1_warn: 'cyan',
..... mod1_error: 'red',
..... mod2_note: 'yellow'
..... });
> console.log("This is a helpful message".mod2_note);
This is a helpful message
> console.log("This is a bad message".mod1_error);
This is a bad message
```

Дополнительные сведения о модуле Colors можно найти по адресу https://github.com/Marak/colors.js.

Модуль Optimist — еще один небольшой и простой модуль

Optimist является еще одним модулем, предназначенным для решения конкретной проблемы: синтаксическом разборе ключей команды. И это все, что он делает, но делает он это очень хорошо.

Например, следующее простое приложение использует модуль Optimist для вывода на консоль ключей командной строки:

```
#!/usr/local/bin/node
var argv = require('optimist').argv;
console.log(argv.o + " " + argv.t);
```

Можно запустить приложение с короткими ключами. Следующая команда приведет к выводу на консоль значений 1 и 2:

```
./app.js -o 1 -t 2
```

Можно также обработать длинные ключи:

```
#!/usr/local/bin/node
var argv = require('optimist').argv;
console.log(argv.one + " " + argv.two);
```

Эти ключи можно проверить со следующей командой, в результате чего будет выведена строка My Name:

```
./app2.js --one="My" --two="Name"
```

Модуль Optimist можно также использовать для обработки булевых и бездефисных ключей.

Дополнительные сведения о модуле Optimist можно найти по адресу https://github.com/substack/node-optimist.

ЗАПУСК NODE-ПРИЛОЖЕНИЯ В АВТОНОМНОМ РЕЖИМЕ

Большинство примеров в этой книге запускаются с использованием синтаксиса node appname.js.

Однако Node-приложение можно запустить в качестве автономного, внеся всего два изменения.

Во-первых, в первой строке приложения должен быть следующий код: #!/usr/local/bin/node. При этом приложение должно размещаться там же, где установлена среда Node.

Во-вторых, нужно изменить права доступа к файлу: chmod a+x appname.js.

А теперь можно запустить приложение с помощью команды ./appname.js.

Модуль Underscore

Установите модуль Underscore с помощью следующей команды:

```
npm install underscore
```

Разработчики считают Underscore библиотекой, связывающей воедино утилиты для Node. Этот модуль предоставляет множество усовершенствованных JavaScript-функций, к которым нас приучили такие библиотеки сторонних разработчиков, как jQuery и Prototype.js.

Этот модуль назван Underscore (подчеркивание), поскольку по традиции доступ к его функциональности осуществляется посредством знака подчеркивания (_), который аналогичен знаку $ в библиотеке jQuery. Вот как это выглядит:

```
var _ = require('underscore');
_.each(['apple','cherry'], function (fruit) { console.log(fruit); });
```

Разумеется, при использовании знака подчеркивания есть проблема, которая заключается в том, что он имеет в REPL специальное значение. Но не стоит волноваться, вместо подчеркивания можно просто задействовать другую переменную, us:

```
var us = require('underscore');
us.each(['apple','cherry'], function(fruit) { console.log(fruit); });
```

Underscore предлагает расширенную функциональность для массивов, коллекций, функций, объектов, цепочек и другие полезные вещи общего назначения. К счастью, для всех этих функциональных возможностей существует превосходная документация, поэтому детали я здесь опущу.

Но все же я упомяну об одной весьма привлекательной возможности — расширении Underscore, контролируемом вашими собственными полезными функциями через функцию mixin. Работу этого и других методов можно накоротке изучить в REPL-сеансе:

```
> var us = require('underscore');
undefined
> us.mixin({
... betterWithNode: function(str) {
..... return str + ' is better with Node';
..... }
... });
> console.log(us.betterWithNode('chocolate'));
chocolate is better with Node
```

В некоторых Node-модулях вы можете встретить термин mixin. Его употребляют, когда свойства одного объекта «примешиваются» (mixed in) к свойствам другого объекта.

Разумеется, было бы вполне логично расширить Underscore за счет модуля, который можно было бы многократно использовать в своих приложениях, что приводит нас к нашей следующей теме — созданию собственных нестандартных модулей.

Создание собственного пользовательского модуля

Точно так же, как это делалось в вашем JavaScript-коде на стороне клиента, вам захочется разбить многократно используемый JavaScript-код на свои библиотеки. Единственным отличием является необходимость в нескольких дополнительных

этапах для превращения JavaScript-библиотеки в модуль для использования совместно с Node.

Предположим, что у вас есть библиотечная JavaScript-функция под названием `concatArray`, которая получает строку и массив строк, а затем объединяет первую строку с каждой строкой в массиве:

```
function concatArray(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
}
```

Вам хочется иметь возможность пользоваться этой и другими функциями в своих Node-приложениях.

Для преобразования вашей JavaScript-библиотеки с целью ее применения в Node нужно экспортировать все свои открытые функции, используя объект `exports`, как показано в следующем примере:

```
exports.concatArray = function(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
};
```

Для использования функции `concatArray` в Node-приложении импортируйте библиотеку с помощью инструкции `require`, присвоив ее имя переменной. После этого вы можете вызывать любую открытую функцию в вашем коде:

```
var newArray = require ('./arrayfunctions.js');
console.log(newArray.concatArray('hello', ['test1','test2']));
```

Все очень просто, если помнить о двух обстоятельствах:

- Для экспорта функции следует использовать объект `exports`.
- Библиотеку следует рассматривать как отдельный импортированный объект, присвоенный переменной для доступа к функциям.

Пакетирование всего каталога

Модуль можно разбить на отдельные JavaScript-файлы, находящиеся в одном каталоге. Node может загрузить содержимое каталога, если этот контент будет упорядочен одним из двумя способов.

Первый способ заключается в предоставлении JSON-файла с именем `package.json` с информацией о каталоге. Структура может содержать другую информацию, но к Node относятся только следующие записи:

```
{ "name" : "mylibrary",
  "main" : "./mymodule/mylibrary.js" }
```

Первое свойство, `name`, является именем модуля. Второе, `main`, обозначает точку входа для модуля.

Второй способ заключается во включении в каталог либо файла index.js, либо файла index.node. Такой файл будет служить точкой входа для основного модуля.

А зачем предоставлять каталог, а не просто отдельный модуль? Наиболее веской причиной является наличие существующих JavaScript-библиотек с предоставлением простого файла-оболочки, который помещает открытые функции в инструкции `exports`. Другая причина может быть в слишком большом размере вашей библиотеки, побудившем вас разбить ее на части, чтобы облегчить задачу внесения изменений.

Независимо от причин нужно иметь в виду, что все экспортированные объекты должны быть в одном основном файле, который загружается средой Node.

Подготовка модуля к публикации

Если нужно, чтобы к вашему пакету имели доступ другие разработчики, можете продвигать его через свой веб-сайт, но тогда будет упущена существенная часть аудитории заинтересованных лиц. Когда модуль будет готов к публикации, имеет смысл добавить его в список модулей на веб-сайте Node.js, а также опубликовать в реестре npm.

Ранее я уже упоминал о файле package.json. Фактически, он основан на рекомендациях системы модулей CommonJS, которые можно найти по адресу http://wiki.commonjs.org/wiki/Packages/1.0#Package_Descriptor_File (хотя можно проверить, не вышла ли более новая версия).

Среди рекомендуемых в файле package.json находятся следующие поля:

`name`

: Имя пакета.

`description`

: Описание пакета.

`version`

: Текущая версия, соответствующая семантическим требованиям к версии.

`keywords`

: Массив поисковых терминов.

`maintainers`

: Массив сведений о тех, кто осуществляет поддержку пакета (сюда включаются имена, адреса электронной почты и веб-сайты).

`contributors`

: Массив сведений о спонсорах пакета (сюда включаются имена, адреса электронной почты и веб-сайты).

`bugs`

: URL-адрес, на который можно отправлять сообщения об ошибках.

`licenses`

Массив лицензий.

`repositories`

Массив сведений о репозиториях, в которых может быть найден пакет.

`dependencies`

Необходимые пакеты и номера их версий.

Хотя остальные поля являются необязательными, и без них в этом файле довольно много полей. К счастью, npm упрощает создание такого файла. Наберите в командной строке следующую команду:

```
npm init
```

В результате начнется перебор всех обязательных полей с выдачей приглашения на ввод информации в каждое из них. По окончании этой работы будет создан файл package.json.

В листинге 3.13 (см. главу 3) я приступил к созданию объекта под названием `inputChecker`, который проверяет входящие данные на наличие команд, после чего обрабатывает команду. В примере демонстрируется, как нужно использовать в программе объект `EventEmitter`. А сейчас мы модифицируем этот простой объект, чтобы им можно было пользоваться в других приложениях и модулях.

Сначала мы создадим подкаталог в node_modules и назовем его inputcheck, после чего поместим в него файл с существующим кодом объекта `inputChecker`. Файл нужно переименовать в index.js. Затем нужно модифицировать код, выделив ту его часть, которая реализует новый объект. Мы сохраним ее для будущего тестового файла. И последней модификацией является добавление объекта `exports`, в результате чего должен получиться код, показанный в листинге 4.1.

Листинг 4.1. Приложение из листинга 3.13, модифицированное под получение объекта модуля

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.inputChecker = inputChecker;

function inputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
};

util.inherits(inputChecker,eventEmitter);
inputChecker.prototype.check = function check(input) {
  var self = this;
```

продолжение ⇩

Листинг 4.1 *(продолжение)*

```
  var command = input.toString().trim().substr(0,3);
  if (command == 'wr:') {
     self.emit('write',input.substr(3,input.length));
  } else if (command == 'en:') {
     self.emit('end');
  } else {
     self.emit('echo',input);
  }
};
```

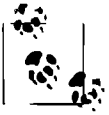
Мы не можем экспортировать функцию объекта напрямую, поскольку `util.inherits` ожидает наличие объекта в файле `inputChecker`. Мы также изменяем имеющийся далее в этом файле прототип объекта `inputChecker`. Мы могли бы изменить ссылки в этом коде для использования `exports.inputChecker`, но лучше этого не делать. Проще присвоить объект в отдельной инструкции.

Для создания файла package.json я запустил команду `npm init` и ответил на каждое приглашение ввода данных. Получившийся файл показан в листинге 4.2.

Листинг 4.2. Создание файла package.json для модуля inputChecker

```
{
  "author": "Shelley Powers <shelleyp@burningbird.net>
    (http://burningbird.net)",
  "name": "inputcheck",
  "description": "Looks for commands within the string
    and implements the commands",
  "version": "0.0.1",
  "homepage": "http://inputcheck.burningbird.net",
  "repository": {
    "url": "
  },
  "main": "inputcheck.js",
  "engines": {
    "node": "~0.6.10"
  },
  "dependencies": {},
  "devDependencies": {},
  "optionalDependencies": {}
}
```

Команда `npm init` не выдает приглашений на ввод зависимостей (`dependencies`), их нужно добавлять в файл напрямую. Однако модуль `inputChecker` не зависит от каких бы то ни было внешних модулей, поэтому в данном случае мы можем оставить эти поля незаполненными.



Более глубокое рассмотрение файла package.json предстоит в главе 16.

Теперь можно протестировать новый модуль, чтобы убедиться, что он действительно работает как модуль. В листинге 4.3 приведен фрагмент знакомого нам приложения inputChecker, но теперь он выделен в отдельное тестовое приложение, которое тестирует новый объект.

Листинг 4.3. Тестовое приложение InputChecker

```
var inputChecker = require('inputcheck').inputChecker;
// тестирование нового объекта и обработка события
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

Теперь мы можем переместить тестовое приложение в новый подкаталог examples в каталоге модуля, чтобы поместить его в пакет вместе с модулем в качестве примера. Установившаяся практика требует, чтобы мы также предоставили каталог test с одним или несколькими тестовыми приложениями, а также каталог doc с документацией. Для модуля будет достаточно совсем небольшого файла README. И наконец, мы создаем tarball-дистрибутив, упакованный с помощью программы gzip.

Как только будет предоставлено все необходимое, модуль может быть опубликован.

Публикация модуля

Команда разработчиков npm предоставила великолепный источник информации для Node-разработчиков — руководство разработчика (Developer Guide). В нем можно найти все, что нужно знать о публикации модулей.

В руководстве указаны некоторые дополнительные требования к файлу package.json. Вдобавок к уже созданным полям нам нужно также внести в него поле каталогов directories с хэшем папок, например, ранее упомянутых test и doc:

```
"directories" : {
    "doc" : ".",
    "test" : "test",
    "example" : "examples"
}
```

Перед публикацией модуля руководство рекомендует проверить, что модуль может быть должным образом установлен. Чтобы протестировать эту возможность, введите в корневом для модуля каталоге следующую команду:

```
npm install . -g
```

На данный момент нами проведены операции тестирования модуля `inputChecker` и модификации пакета package.json с целью добавления каталогов, а также подтверждена успешная установка пакета.

Далее нам нужно добавить себя в пользователи npm, если это еще не сделано. Для этого служит следующая команда:

```
npm adduser
```

Далее после приглашения нужно ввести имя пользователя, пароль и адрес электронной почты.

Осталось выполнить еще одно последнее действие:

```
npm publish
```

Мы можем предоставить путь к tarball-дистрибутиву или к каталогу. В соответствии с руководством все в каталоге выставляется для открытого использования, если только мы не указываем в файле package.json список `.npmignore` для игнорирования материала. Но лучше все же просто удалить из публикуемого модуля все лишнее.

После публикации и после того, как исходный код будет также выложен на хостинге GitHub (если вы задействуете это хранилище), модуль официально готов для всеобщего использования. Предложите модуль в Твиттере, в Google+, в Фейсбук, на своем веб-сайте и в любых других местах, где, по вашему мнению, люди захотят узнать о модуле. Такой вид продвижения нельзя считать хвастовством — это предложение *совместного использования*.

5 Поток управления, асинхронные паттерны и обработка исключений

Временами, когда речь заходит об асинхронных событиях, функциях обратного вызова и таких новых объектах, как `EventEmitter`, не говоря уже обо всех новшествах в функциональности серверной стороны, с которыми нам приходится иметь дело, Node кажется неким пугающим монстром. Однако если вам приходилось работать с современными JavaScript-библиотеками, то у вас уже должен быть опыт использования основной функциональности, поддерживаемой в Node, по крайней мере, относящейся к разработке кода для асинхронного режима.

Например, если вы применяли таймер в JavaScript, значит, вы использовали асинхронную функцию. Если вы когда-нибудь программировали в Ajax, то использовали асинхронные функции. Даже обычный и давно всем знакомый обработчик события `onclick` является асинхронной функцией, поскольку заранее никогда не известно, когда пользователь решит щелкнуть кнопкой мыши.

Асинхронной функцией является любой метод, не блокирующий управляющий программный поток в ожидании наступления какого-либо события или результата. Что касается обработки события `onclick`, приложение не блокирует всю остальную прикладную работу, ожидая, пока пользователь щелкнет кнопкой мыши, точно так же, как оно не блокирует свою работу, ожидая срабатывания таймера или ответа от сервера на Ajax-вызов.

В этой главе детально рассматривается все, что связано с понятием *асинхронного управления*. В частности, изучаются некоторые асинхронные паттерны и Node-модули, обеспечивающие более тонкое управление ходом выполнения программы в этой новой среде. И поскольку асинхронное управление позволяет ввести в систему обработки ошибок несколько новых и интересных приемов, мы также собираемся более пристально присмотреться к обработке исключений в асинхронной среде Node.

Обязательства? Никаких обязательств, только обратный вызов

Когда технология Node только появилась, для реализации асинхронной функциональности использовалась концепция *обязательств* (promises), предложенная в 1970-х годах. Обязательство — это объект, отражающий результат некоего асинхронного действия. Он также известен как *будущий* (future), *отсроченный* (delay) или *отложенный* (deferred). Концепция обязательств поддерживалась в модели разработки CommonJS.

В ранних реализациях Node объект обязательства генерировал только два события: `success` (успех) и `error` (ошибка). Пользоваться им было довольно просто: если асинхронная операция завершалась успешно, генерировалось событие `success`, в противном случае — событие `error`. Никаких других событий не было, причем объект генерировал либо одно, либо другое событие, но не оба и не более одного одновременно. Листинг 5.1 демонстрирует предварительно реализованное обязательство в функции, которая открывает файл и считывает его контент.

Листинг 5.1. Использование предварительно реализованного Node-объекта promise

```
function test_and_load(filename) {
  var promise = new process.Promise();
  fs.stat(filename).addCallback(function (stat) {

    // фильтрация элементов, не являющихся файлами
    if (!stat.isFile()) { promise.emitSuccess(); return; }

    // в противном случае - чтение файла
    fs.readFile(filename).addCallback(function (data) {
      promise.emitSuccess(data);
    }).addErrback(function (error) {
      promise.emitError(error);
    });

  }).addErrback(function (error) {
    promise.emitError(error);
  });
return promise;
}
```

Каждый объект был способен возвращать объект `promise`. Код обработки успешного результата передавался методу `addCallback` обязательства в виде функции с единственным параметром — данными. Код обработки ошибки передавался методу `addErrback` обязательства в виде функции с единственным параметром — ошибкой:

```
var File = require('file');
var promise = File.read('mydata.txt');
promise.addCallback(function (data) {
  // обработка данных
});
```

```
promise.addErrback(function (err) {
  // обработка ошибки
})
```

Когда бы ни происходило событие, обязательство гарантировало выполнение необходимых действий: либо предоставляя возможность манипулирования результатами, либо обеспечивая обработку ошибки.

Код листинга 5.1 является одним из множества примеров использования асинхронных функций, которые приводятся в документации, находящейся по адресу http://groups.google.com/group/nodejs/browse_thread/thread/8dab9f0a5ad753d5, в рамках дискуссии о том, как в будущем эта концепция могла бы применяться в Node.

От объекта обязательств решено было отказаться в Node версии 0.1.30. Вот что Райан Дал (Ryan Dahl) написал по этому поводу:

> Причина в том, что многие (не исключая меня самого) для операций с файловой системой, которые не влекут за собой обязательного создания объекта, хотели бы использовать низкоуровневый интерфейс. В то же время многие другие хотели бы использовать что-то вроде обязательств, но отличающихся в том или ином отношении. Поэтому вместо обязательств в качестве последнего аргумента мы задействуем функцию обратного вызова, а задачу создания лучших уровней абстракции передаем пользовательским библиотекам.

Место Node-объекта обязательства в качестве последнего аргумента всех асинхронных методов заняли *функции обратного вызова* — именно их мы применяли в предыдущих главах. Первым аргументом такой функции всегда является объект ошибки.

Чтобы продемонстрировать базовую функциональность обратного вызова, в листинге 5.2 представлено полноценное Node-приложение, которое создает объект с одним методом по имени `someMethod`. Этот метод принимает три аргумента, второй из которых должен быть строкой, а третий — функцией обратного вызова. В этом методе, если второй аргумент не передан или не является строкой, объект создает новый объект ошибки, который передается функции обратного вызова. В противном случае функции обратного вызова передается результат вызова метода.

Листинг 5.2. Базовая структура функции обратного вызова, передаваемой в качестве последнего аргумента

```
var obj =function() { };

obj.prototype.doSomething = function(arg1, arg2_) {
  var arg2 = typeof(arg2_) === 'string' ? arg2_ : null;

  var callback_ = arguments[arguments.length - 1];
  callback = (typeof(callback_) == 'function' ? callback_ : null);

  if (!arg2)
```

продолжение

Листинг 5.2 *(продолжение)*

```
    return callback(new Error('second argument missing or not a string'));
  callback(arg1);
}
var test = new obj();

try {
  test.doSomething('test', 3.55, function(err,value) {
    if (err) throw err;
    console.log(value);
  });
} catch(err) {
  console.error(err);
}
```

Ключевые элементы в функциональности обратного вызова выделены в коде полужирным шрифтом.

Первой ключевой функциональностью должна быть гарантия того, что последним аргументом является функция обратного вызова. Мы не можем определить намерений пользователя, но мы можем убедиться в том, что последний аргумент является функцией, и будем это делать. Второй ключевой функциональностью должно быть создание нового Node-объекта `Error` при возникновении ошибки и возвращение его в качестве результата функции обратного вызова. И последней важной функциональностью является обращение к функции обратного вызова с передачей этой функции результата при отсутствии ошибки. Короче говоря, все остальное может как угодно меняться при условии, что поддерживаются три ключевые функциональности:

- гарантируется функция в качестве последнего аргумента;
- создается Node-объект `Error`, который возвращается в случае ошибки;
- при отсутствии ошибки активизируется функция обратного вызова, передающая результат вызова метода.

С имеющимся в листинге 5.1 кодом приложение выводит на консоль следующее сообщение об ошибке (второй аргумент отсутствует или не является строкой):

```
[Error: second argument missing or not a string]
```

Давайте изменим код вызова метода следующим образом:

```
test.doSomething('test','this',function(err,value) {
```

Это приводит к выводу на консоль строки `test`. А теперь сделаем следующее изменение в коде вызова метода:

```
test.doSomething('test',function(err,value) {
```

Это код опять приводит к ошибке, но на этот раз из-за отсутствия второго аргумента.

Если просмотреть код в каталоге lib установленной копии Node, можно увидеть, что паттерн использования обратного вызова в качестве последнего аргумента повторяется повсюду. При всех функциональных изменениях этот паттерн остается одним и тем же.

Этот подход достаточно прост и обеспечивает целостность результатов вызова асинхронных методов. Тем не менее он порождает также присущие только ему проблемы, о которых мы поговорим в следующем разделе.

Последовательная функциональность, вложенные обратные вызовы и обработка исключений

Нередко в JavaScript-приложениях на стороне клиента можно встретить такой код:

```
val1 = callFunctionA();
val2 = callFunctionB(val1);
val3 = callFunctionC(val2);
```

Функции вызываются поочередно, а результаты передаются от предыдущей функции к следующей в данной последовательности. Поскольку все функции являются синхронными, нам не приходится волноваться, что вызовы функций окажутся вне этой последовательности и мы получим какие-либо неожиданные результаты.

В листинге 5.3 показана довольно часто возникающая в подобных последовательных программах ситуация. В приложении используются синхронные версии Node-методов для работы с файловой системой, которые обеспечивают открытие файла и получение содержащихся в нем данных, изменение данных путем замены всех ссылок на «apple» ссылками на «orange» и вывод получившихся строк в новый файл.

Листинг 5.3. Последовательное синхронное приложение

```
var fs = require('fs');

try {
  var data = fs.readFileSync('./apples.txt','utf8');
  console.log(data);
  var adjData = data.replace(/[A|a]pple/g,'orange');

  fs.writeFileSync('./oranges.txt', adjData);
} catch(err) {
  console.error(err);
}
```

Поскольку при возникновении проблем мы не можем гарантировать внутреннюю обработку ошибок в каждой из функций, все вызовы функций заключены в блок try, что позволяет обеспечить элегантную или, по крайней мере, более информативную обработку исключений. Вот на что будет похоже сообщение об ошибке, если приложение не сможет найти файл, информацию из которого нужно прочитать:

```
{ [Error: ENOENT, no such file or directory './apples.txt']
  errno: 34,
  code: 'ENOENT',
  path: './apples.txt',
  syscall: 'open' }
```

Возможно, это не самая полезная информация, но, по крайней мере, она намного лучше следующей альтернативы:

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
        ^
Error: ENOENT, no such file or directory './apples.txt'
    at Object.openSync (fs.js:230:18)
    at Object.readFileSync (fs.js:120:15)
    at Object.<anonymous> (/home/examples/public_html/node/read.js:3:18)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

В этом примере мы рассчитываем получить вполне ожидаемые результаты, поскольку вызов каждой функции осуществляется в заданной последовательности.

Преобразование этого паттерна синхронного последовательного приложения в асинхронную реализацию требует внесения нескольких изменений. Сначала нужно заменить все функции их асинхронными аналогами. Но нам также потребуется учесть тот факт, что каждая функция не блокирует приложение при своем вызове, то есть функции вызываются независимо друг от друга, а значит, мы не можем гарантировать нужную последовательность. Единственный способ обеспечить нужную очередность вызова каждой функции — использовать *вложенные обратные вызовы* (nested callbacks).

В листинге 5.4 представлена асинхронная версия приложения из листинга 5.3. Все вызовы функций файловой системы заменены их асинхронными версиями, а правильная очередность вызовов функций обеспечивается посредством вложенных обратных вызовов.

Листинг 5.4. Приложение из листинга 5.3, переделанное с использованием асинхронных вложенных обратных вызовов

```
var fs = require('fs');

try {
  fs.readFile('./apples2.txt','utf8', function(err,data) {

    if (err) throw err;

    var adjData = data.replace(/[A|a]pple/g,'orange');

    fs.writeFile('./oranges.txt', adjData, function(err) {

      if (err) throw err
    });
  });
} catch(err) {
  console.error(err);
}
```

В листинге 5.4 файл на входе открывается и считывается, а затем, только когда оба действия завершаются, активизируется функция обратного вызова, переданная в качестве последнего аргумента. В этой функции проводится проверка на отсутствие ошибки, то есть проверяется, что значение равно `null`. Если это не так, вбрасывается ошибка для ее последующего перехвата во внешнем блоке обработки исключений.

В некоторых руководствах по стилю программирования вбрасывание ошибки не одобряется, а предлагаются более сложные конструкции, предоставляющие такие объекты и функции обработки ошибок, которые сами принимают все возможные ошибки. Моя же главная задача состоит в том, чтобы ошибки обрабатывались.

Если ошибок не было, данные проходят обработку и вызывается асинхронный метод `writeFile`. Его функция обратного вызова имеет один параметр, объект ошибки. Если этот объект не равен нулю, он вбрасывается для обработки внешним блоком исключений.

В случае ошибки все должно выглядеть следующим образом:

```
/home/examples/public_html/node/read2.js:11
        if (err) throw err;
                 ^
Error: ENOENT, no such file or directory './boogabooga/oranges.txt'
```

Если нужно получить трассировку стека ошибки, можно вывести свойство `stack` Node-объекта ошибки:

```
catch(err) {
  console.log(err.stack);
}
```

Включение следующего последовательного вызова функции добавляет обратным вызовам новый уровень вложенности. В листинге 5.5 мы обращаемся к списку файлов определенного каталога. В каждом из этих файлов мы заменяем общее доменное имя конкретным доменным именем, используя для замены строковый метод `replace`, и записываем результат *обратно* в исходный файл. С помощью открытого потока записи для каждого измененного файла ведется журнал.

Листинг 5.5. Извлечение из каталога списка модифицируемых файлов

```
var fs = require('fs');

var writeStream = fs.createWriteStream('./log.txt',
  {'flags' : 'a',
   'encoding' : 'utf8',
   'mode' : 0666});

try {
  // получение списка файлов
  fs.readdir('./data/', function(err, files) {
```

продолжение ⇩

Листинг 5.5 *(продолжение)*

```
    // для каждого файла
    files.forEach(function(name) {

      // изменение контента
      fs.readFile('./data/' + name,'utf8', function(err,data) {
        if (err) throw err;
        var adjData = data.replace(/somecompany\.com/g,'burningbird.net');

        // запись в файл
        fs.writeFile('./data/' + name, adjData, function(err) {

         if (err) throw err;

         // запись в журнал
         writeStream.write('changed ' + name + '\n', 'utf8', function(err) {

            if(err) throw err;
         });
        });
      });
    });
  });
} catch(err) {
console.error(util.inspect(err));
}
```

Хотя приложение выглядит так, будто оно обрабатывает каждый файл индивидуально, перед тем как переходить к следующему файлу, следует помнить, что каждый из методов в этом приложении является асинхронным. Если запустить приложение несколько раз и проверить файл log.txt, то вы увидите, что файлы обрабатываются с разной (похожей случайной) очередностью. В моем каталоге data имеется пять файлов. Запуск приложения три раза подряд привел к следующему выводу в файл log.txt (пустые строки вставлены, чтобы было понятней):

```
changed data1.txt
changed data3.txt
changed data5.txt
changed data2.txt
changed data4.txt

changed data3.txt
changed data1.txt
changed data5.txt
changed data2.txt
changed data4.txt

changed data1.txt
changed data3.txt
```

```
changed data5.txt
changed data4.txt
changed data2.txt
```

Еще одна проблема возникает, если для какой-либо цели требуется проверить, завершены ли изменения во всех файлах. Метод `forEach` обращается к функциям обратного вызова асинхронно, поэтому он не блокирует программу. Добавим после метода `forEach` такую инструкцию:

```
console.log('all done');
```

Однако на самом деле эта инструкция вовсе не означает, что приложение полностью завершило свою работу, поскольку метод `forEach` не блокирует приложение. Чтобы показать это, укажем инструкцию `console.log` в момент записи в журнал сведений об измененном файле:

```
writeStream.write('changed ' + name + '\n', 'utf8', function(err) {

  if(err) throw err;
  console.log('finished ' + name);
});
```

Кроме того, добавим после вызова метода `forEach` следующий код:

```
console.log('all finished');
```

В результате на консоль будет выведена следующая информация:

```
all done
finished data3.txt
finished data1.txt
finished data5.txt
finished data2.txt
finished data4.txt
```

Чтобы решить проблему, нужно добавить счетчик, значение которого увеличивается с каждым сообщением о записи в журнал, а затем сравнить значение счетчика с длиной массива файлов и при их равенстве вывести сообщение «все сделано»:

```
// перед обращением к каталогу
var counter = 0;
...
      writeStream.write('changed ' + name + '\n', 'utf8', function(err) {
        if(err) throw err;
        console.log('finished ' + name);
        counter++;
        if (counter >= files.length)
          console.log('все сделано');

      });
```

После этого будет получен ожидаемый результат: сообщение «все сделано» появится уже после сообщений об обновлении всех файлов.

Приложение работает весьма неплохо, исключая случаи, когда каталог, к которому идет обращение, наряду с файлами содержит подкаталоги. Если приложению попадется подкаталог, оно выдает следующую ошибку:

```
/home/examples/public_html/node/example5.js:20
            if (err) throw err;
                     ^
Error: EISDIR, illegal operation on a directory
```

В листинге 5.6 возникновение подобной ошибки предотвращается с помощью метода `fs.stats`, для возвращения объекта представляющего данные из Unix-команды `stat`. Этот объект содержит информацию об объекте, включая и то, является он файлом или нет. Разумеется, метод `fs.stats` является еще одним асинхронным методом, требующим еще более глубокого вложения обратных вызовов.

Листинг 5.6. Добавление проверки методом stats каждого объекта каталога, чтобы убедиться в том, что объект является файлом

```
var fs = require('fs');

var writeStream = fs.createWriteStream('./log.txt',
  {'flags' : 'a',
   'encoding' : 'utf8',
   'mode' : 0666});

try {
  // получение списка файлов
  fs.readdir('./data/', function(err, files) {

    // для каждого файла
    files.forEach(function(name) {

      // проверка принадлежности объекта к файлу
      fs.stat('./data/' + name, function(err, stats) {

        if (err) throw err;

        if (stats.isFile())

        // изменение контента
        fs.readFile('./data/' + name,'utf8', function(err,data) {

          if (err) throw err;
          var adjData = data.replace(/somecompany\.com/g,'burningbird.net');

          // запись в файл
          fs.writeFile('./data/' + name, adjData, function(err) {

            if (err) throw err;

            // запись в журнал
            writeStream.write('changed ' + name + '\n', 'utf8',
```

```
            function(err) {

              if(err) throw err;
            });
          });
        });
      });
    });
  });
} catch(err) {
console.error(err);
}
```

Приложение справляется с поставленной задачей и делает это вполне приемлемо, но читать и поддерживать такое приложение не просто. Я слышал, что такого рода вложенные обратные вызовы называют *спагетти обратных вызовов* и даже еще более выразительно — *пирамидой судьбы*, что можно считать вполне удачными определениями.

Вложенные обратные вызовы продолжают сдвигать код к правой границе документа, делая все более проблематичным безошибочный ввод кода для каждой функций обратного вызова. Однако мы не можем разбить вложенность обратных вызовов на части, поскольку методы должны вызываться строго в заданном порядке:

1. Начало просмотра каталога.
2. Фильтрация подкаталогов.
3. Чтение контента каждого файла.
4. Изменение контента.
5. Запись в исходный файл.

Нам нужно найти способ реализации этой серии вызовов методов, но без зависимости от вложенных обратных вызовов. Для этого нужно изучить модули сторонних разработчиков, предлагающие асинхронный поток управления.

В одном из подходов для каждого метода в качестве функции обратного вызова предлагается именованная функция. Это позволяет выровнять пирамиду, что может помочь упростить отладку. Однако данный подход не решает ряд других проблем, в частности проблему определения момента завершения всех процессов. Для этого все равно понадобятся библиотеки сторонних разработчиков.

Асинхронные паттерны и модули потока управления

Приложение из листинга 5.6 является примером асинхронного паттерна, где каждая функция вызывается в нужный момент потока управления и передает свои

результаты следующей функции, причем вся цепочка останавливается только при возникновении ошибки. Существует несколько подобных паттернов, хотя некоторые из них являются вариациями других, к тому же не во всех из них используется одна и та же терминология.

Один из Node-модулей, Async, предлагает названия и поддержку для наиболее широкого перечня асинхронных паттернов потока управления:

`waterfall`

Функции вызываются по очереди, а результаты вызова всех функций передаются в виде массива последней функции обратного вызова (другие названия этого паттерна: `series` и `sequence`).

`series`

Функции вызываются по очереди, а (не обязательно) результаты передаются в виде массива последней функции обратного вызова.

`parallel`

Функции запускаются параллельно, а затем завершаются, результаты передаются последней функции обратного вызова (но результирующий массив в некоторых интерпретациях параллельного паттерна его частью не является).

`whilst`

Осуществляет повторяющийся вызов одной функции, активизирующей последнюю функцию обратного вызова, только если предварительный тест выдает значение `false` или случается ошибка.

`queue`

Вызывает функции в параллельном режиме вплоть до заданного ограничения параллельности, и новые функции выстраиваются в очередь, пока одна из функций не завершит свою работу.

`until`

Осуществляет повторяющийся вызов одной функции, активизирующей последнюю функцию обратного вызова, только если последующий тест выдает значение `false` или случается ошибка.

`auto`

Функции вызываются по требованию, каждая функция получает результаты предыдущих функций обратного вызова.

`iterator`

Каждая функция вызывает следующую функцию, при этом имеется возможность индивидуального доступа к следующему итератору.

`apply`

Функция продленного действия с уже применяемыми аргументами, вступающая во взаимодействие с другими функциями потока управления.

`nextTick`

Вызов функции обратного вызова на следующем проходе цикла обработки событий основан на применении Node-метода `process.nextTick`.

В списке модулей, предоставляемым веб-сайтом Node.js, есть категория «Control Flow/Async Goodies». В этой категории есть модуль Async, предоставляющий перечисленные здесь паттерны потока управления. Хотя не каждый модуль потока управления дает возможность работать со всеми возможными паттернами, большинство модулей поддерживает функциональность наиболее популярных паттернов: `series` (также называемый `sequence` и иногда `waterfall`, как в предыдущем списке, хотя в списке модуля Async паттерн `waterfall` указывается отдельно от `series`) и `parallel`. Кроме того, некоторые модули восстанавливают концепцию объектов обязательств ранних версий Node, в то время как другие реализуют концепцию *волокон* (fibers), призванную эмулировать программные потоки.

В следующих двух разделах описываются два наиболее популярных из активно поддерживаемых модулей потока управления: Step и Async. Каждый из них предлагает собственный уникальный взгляд на организацию асинхронного потока управления, хотя оба предлагают весьма полезную и, наверное, очень важную программную конструкцию — службу.

Модуль Step

Модуль Step является специальным вспомогательным модулем, позволяющим реализовать упрощенный поток управления для последовательного и параллельного выполнения. Его можно установить с помощью диспетчера Node-пакетов (npm):

```
npm install step
```

Модуль Step экспортирует всего один объект. Чтобы использовать объект для последовательного выполнения, нужно поместить ваши асинхронные вызовы внутри функций, которые затем передаются объекту в качестве параметров. В листинге 5.7 модуль Step используется для чтения контента из файла, внесения изменений в этот контент и последующей записи контента обратно в файл.

Листинг 5.7. Использование модуля Step для решения последовательных асинхронных задач

```
var fs = require('fs'),
  Step = require('step');

try {

  Step (
    function readData() {
      fs.readFile('./data/data1.txt', 'utf8', this);
    },
    function modify(err, text) {
      if (err) throw err;
      return text.replace(/somecompany\.com/g,'burningbird.net');
    },
```

продолжение ⇩

Листинг 5.7 *(продолжение)*

```
    function writeData(err, text) {
      if (err) throw err;
      fs.writeFile('./data/data1.txt', text, this);
    }
  );
} catch(err) {
  console.error(err);
}
```

Первая функция в Step-последовательности, `readData`, считывает контент файла в строку, которая затем передается второй функции. Вторая функция модифицирует строку путем замены, и результат передается третьей функции. В третьей функции измененная строка записывается обратно в исходный файл.

Дополнительные сведения можно найти на сайте Step GitHub по адресу https://github.com/creationix/step.

Если углубиться в детали, то первая функция является оболочкой для асинхронного метода `fs.readFile`. Однако вместо того, чтобы передать в качестве последнего параметра функцию обратного вызова, код передает контекст `this`. Когда функция завершает свою работу, ее данные и любая возможная ошибка отправляются следующей функции по имени `modify`. Эта функция не является асинхронной, поскольку она всего лишь заменяет одну подстроку другой. Ей не требуется контекст `this`, она просто в конце функции возвращает результат.

Последняя функция получает только что измененную строку и записывает ее обратно в исходный файл. И опять, поскольку она является асинхронной, то получает `this` вместо функции обратного вызова. Если бы в качестве последнего параметра последней функции мы не указали `this`, любые ошибки, если бы таковые случились, не были бы вброшены и перехвачены во внешнем контуре. Предположим, в следующем модифицированном коде каталога `boogabooga` не существует:

```
function writeFile(err, text) {
  if (err) throw err;
  fs.writeFile('./boogabooga/data/data1.txt');
}
```

В этом случае о том, что запись не удалась, мы никогда так и не узнаем.

Хотя вторая функция не является асинхронной, каждая функция в модуле Step, исключая первую, в целях согласованности требует, чтобы первым параметром был объект ошибки. По умолчанию в синхронной функции он имеет значение `null`.

Листинг 5.7 частично реализует функциональность приложения из листинга 5.6. Мог бы он реализовать оставшуюся функциональность, особенно относящуюся к внесению изменений в несколько файлов? Ответ и да, и нет. Да, такая работа может быть проделана, но только если мы вставим туда чуть более сложный код.

В листинге 5.8 для получения списка файлов заданного каталога я добавил асинхронную функцию `readir`. Массив с именами файлов обрабатывается с помощью

команды `forEach`, как было в листинге 5.6, но в конце вызова функции `readFile` нет ни функции обратного вызова, ни ключевого слова `this`. В модуле Step вызов для создания объекта `group` служит сигналом к сохранению некоего параметра для группового результата; вызов объекта `group` в асинхронной функции `readFile` приводит к тому, что каждая из функций обратного вызова активизируется по очереди, а результаты группируются в массив для следующей функции.

Листинг 5.8. Использование метода group() модуля Step для обработки групповых асинхронных процессов

```
var fs = require('fs'),
  Step = require('step'),
  files,
  _dir = './data/';

try {
  Step (
    function readDir() {
      fs.readdir(_dir, this);
    },
  function readFile(err, results) {
    if (err) throw err;
    files = results;
    var group = this.group();
    results.forEach(function(name) {
      fs.readFile(_dir + name, 'utf8', group());
    });
  },
  function writeAll(err, data) {
    if (err) throw err;
    for (var i = 0; i < files.length; i++) {
       var adjdata = data[i].replace(/somecompany\.com/g,'burningbird.net');
       fs.writeFile(_dir + files[i], adjdata, 'utf8',this);
      }
    }
  );
} catch(err) {
  console.log(err);
}
```

Для сохранения имен файлов результат выполнения функции `readdir` присваивается глобальной переменной `files`. В последней функции модуля Step обычный цикл `for` осуществляет перебор данных для внесения в них изменений, а затем осуществляет перебор содержимого переменной `files` для получения имени файла. И имя файла, и измененные данные используются в последнем асинхронном вызове функции `writeFile`.

Если нужно внести изменения в каждый файл, можно воспользоваться другим подходом — задействовать предлагаемые модулем Step возможности параллельной обработки. В листинге 5.9 функция `readFile` вызывается для нескольких

различных файлов с передачей ей в качестве последнего параметра функции `this.parallel()`. В результате параметр передается следующей функции для каждого вызова `readFile` в первой функции. Вызов функции `parallel` требуется также в функции `writeFile` второй функции, чтобы каждая функция обратного вызова обрабатывалась в нужное время.

Листинг 5.9. Чтение и запись для группы файлов с использованием средств групповой обработки модуля Step

```
var fs = require('fs'),
  Step = require('step'),
  files;

try {

  Step (
    function readFiles() {
      fs.readFile('./data/data1.txt', 'utf8',this.parallel());
      fs.readFile('./data/data2.txt', 'utf8',this.parallel());
      fs.readFile('./data/data3.txt', 'utf8',this.parallel());
    },
    function writeFiles(err, data1, data2, data3) {
      if (err) throw err;
      data1 = data1.replace(/somecompany\.com/g,'burningbird.net');
      data2 = data2.replace(/somecompany\.com/g,'burningbird.net');
      data3 = data3.replace(/somecompany\.com/g,'burningbird.net');
      fs.writeFile('./data/data1.txt', data1, 'utf8', this.parallel());
      fs.writeFile('./data/data2.txt', data2, 'utf8', this.parallel());
      fs.writeFile('./data/data3.txt', data3, 'utf8', this.parallel());
    }
  );
} catch(err) {
  console.log(err);
}
```

Программа работает, хотя выглядит не слишком изящно. Было бы лучше приберечь параллельную функциональность для последовательности асинхронных функций, которые могли бы вызываться параллельно и при наличии данных, обрабатываемых после обратного вызова.

Что касается нашего более раннего приложения, то вместо того чтобы пытаться адаптировать модуль Step для нашего случая, можно воспользоваться модулем Async — другой библиотекой, предлагающей дополнительную гибкость.

Модуль Async

Модуль Async предлагает функциональность управления коллекциями, в том числе собственные вариации конструкций `forEach`, `map` и `filter`. Он также предлагает ряд полезных функций, включая те, которые *работают с памятью*. Но нас в нем интересуют те механизмы, которые относятся к реализации потока управления.

Не путайте друг с другом модули Async и Async.js — это два разных модуля. В этом разделе рассматривается модуль Async, созданный Каоланом Мак-Мэхоном (Caolan McMahon). Его GitHub-сайт можно найти по адресу https://github.com/caolan/async.

Установите Async, используя диспетчер Node-пакетов:

```
npm install async
```

Как уже упоминалось, Async предлагает средства реализации потока управления для различных асинхронных паттернов, включая `serial`, `parallel` и `waterfall`. Как и Step, этот модуль позволяет упростить пирамиду вложенных функций обратного вызова, но при этом он опирается на совершенно другой подход. Он позволяет не вставлять собственный код между каждой функцией и ее обратным вызовом. Вместо этого мы встраиваем функцию обратного вызова в процесс, делая ее частью процесса.

Как мы уже определили, паттерн ранее рассмотренного приложения соответствует реализуемому в Async паттерну `waterfall`, поэтому в качестве примера мы используем метод `async.waterfall`. В листинге 5.10 я задействовал метод `async.waterfall` для открытия файла и чтения его контента методом `fs.readFile`, выполнения синхронной замены в строке и последующей записи строки обратно в файл методом `fs.writeFile`. Обратите особое внимание на функцию обратного вызова, используемую в приложении на каждом шаге.

Листинг 5.10. Использование метода async.waterfall для чтения, модификации и записи контента файла в асинхронном режиме

```
var fs = require('fs'),
  async = require('async');

try {

  async.waterfall([
    function readData(callback) {
      fs.readFile('./data/data1.txt', 'utf8', function(err, data){
        callback(err,data);
      });
    },
    function modify(text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, adjdata);
    },
    function writeData(text, callback) {
      fs.writeFile('./data/data1.txt', text, function(err) {
        callback(err,text);
      });
    }
  ], function (err, result) {
    if (err) throw err;
```

продолжение ⇩

Листинг 5.10 *(продолжение)*

```
    console.log(result);
  });
} catch(err) {
  console.log(err);
}
```

Метод `async.waterfall` получает два параметра: массив задач и необязательную завершающую функцию обратного вызова. Каждая асинхронная функция задачи является элементом массива метода `async.waterfall`, и каждая функция требует в качестве своего последнего параметра функцию обратного вызова. Это та самая функция обратного вызова, которая позволяет построить цепочку результатов вызова асинхронных функций обратного вызова без необходимости организовать физическую вложенность функций. Однако, как можно видеть в коде, каждое обращение к функции обратного вызова обрабатывается точно так же, как в случае вложенных функций обратного вызова, к тому же нам не нужно проверять каждую функцию на наличие ошибки. Функции обратного вызова ищут объект ошибки в первом параметре. Если объект ошибки передается функции обратного вызова, процесс на этом завершается и вызывается завершающая процедура обратного вызова. В этой завершающей процедуре мы можем провести проверку на наличие ошибки и выбросить ошибку на внешний блок обработки исключений (или обработать ее как-то иначе).

Наш вызов метода `fs.readFile` заключен в оболочку из функции `readData`, которая в первую очередь проводит проверку на наличие ошибки. Если ошибка обнаруживается, происходит вбрасывание ошибки и процесс завершается. Если ошибки нет, функция `readData` в качестве своей последней операции обращается к функции обратного вызова. Для модуля Async это является сигналом к вызову следующей функции с передачей любых данных, имеющих отношение к этому вызову. Следующая функция не является асинхронной, поэтому она выполняет свою работу, передавая в качестве объекта ошибки значение `null`, и модифицирует данные. Последняя функция, `writeData`, вызывает асинхронный метод `writeFile`, используя данные, переданные от предыдущей функции обратного вызова, а затем проводит проверку на наличие ошибки в собственной функции обратного вызова.

В листинге 5.10 используются именованные функции, в то время как в документации к Async — безымянные. Те и другие функции работают одинаково хорошо, но именованные функции помогают упростить отладку и обработку ошибок.

Обработка очень похожа на ту, которая была в листинге 5.4, но без вложенности функций (и необходимости проверки на наличие ошибки в каждой функции). Код может показаться более сложным, чем в листинге 5.4, и я не стал бы настаивать на его использовании для столь незначительного уровня вложенности. Однако посмотрите, на что он способен для более глубокого уровня вложенности функций обратного вызова. В листинге 5.11 полностью дублируется функциональность приложения из листинга 5.6, но без вложенности функций обратного вызова и без избытка отступов.

Листинг 5.11. Получение объектов из каталога, проверка файлов, чтение файлового теста, внесение изменений и запись результатов в журнал

```
var fs = require('fs'),
  async = require('async'),
  _dir = './data/';

var writeStream = fs.createWriteStream('./log.txt',
  {'flags' : 'a',
   'encoding' : 'utf8',
   'mode' : 0666});

try {
  async.waterfall([
    function readDir(callback) {
      fs.readdir(_dir, function(err, files) {
        callback(err,files);
      });
    },
    function loopFiles(files, callback) {
      files.forEach(function (name) {
        callback (null, name);
      });
    },
    function checkFile(file, callback) {
      fs.stat(_dir + file, function(err, stats) {
        callback(err, stats, file);
      });
    },
    function readData(stats, file, callback) {
      if (stats.isFile())
      fs.readFile(_dir + file, 'utf8', function(err, data){
        callback(err,file,data);
      });
    },
    function modify(file, text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, file, adjdata);
    },
    function writeData(file, text, callback) {
      fs.writeFile(_dir + file, text, function(err) {
        callback(err,file);
      });
    },
    function logChange(file, callback) {
      writeStream.write('changed ' + file + '\n', 'utf8', function(err) {
        callback(err, file);
      });
    }
  ], function (err, result) {
```

продолжение ⇩

Листинг 5.11 *(продолжение)*

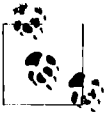
```
    if (err) throw err;
    console.log('modified ' + result);
  });
} catch(err) {
  console.log(err);
}
```

Здесь полностью реализована функциональность приложения из листинга 5.6. Метод `fs.readdir` используется для получения массива объектов каталога. Для доступа к каждому конкретному объекту служит Node-метод `forEach` (а не `Async-forEach`). Метод `fs.stats` предназначен для получения кода состояния каждого объекта. Метод `stats` проверяет принадлежность к файлам, и когда файл найден, он открывается и происходит доступ к его данным. Затем данные модифицируются и передаются для записи обратно в файл с помощью метода `fs.writeFile`. Операция регистрируется в журнале logfile, и параллельно запись выводится на консоль.

Заметьте, что в некоторых функциях обратного вызова данных передается больше. Для большинства функций требуются имя файла и текст, поэтому в последних нескольких методах передаются эти данные. В методах может передаваться любое количество данных, но при этом первым параметром должен быть объект ошибки (или `null` при отсутствии такого объекта), а последним параметром — функция обратного вызова.

Нам уже не нужно проверять наличие ошибки при выполнении задачи в каждой асинхронной функции, поскольку Async тестирует объект ошибки в каждой функции обратного вызова, и при наличии ошибки останавливает обработку и вызывает завершающую функцию обратного вызова. Кроме того, нам не нужно волноваться насчет специальной обработки массива элементов, как это было ранее в этой главе при использовании модуля Step.

Другие Async-методы реализации потока управления, такие как `async.parallel` и `async.serial`, работают в сходном ключе, имея в качестве первого параметра метода массив задач, а в качестве второго аргумента — завершающую функцию обратного вызова. Но, как вы наверно и предполагали, порядок обработки асинхронных задач у них отличается.



Метод async.serial используется в главе 9 с приложением Redis.

Метод `async.parallel` вызывает сразу все асинхронные функции, и когда каждая из них завершает свою работу, вызывает необязательную последнюю функцию обратного вызова. В листинге 5.12 метод `async.parallel` служит для чтения в параллельном режиме контента трех файлов. Однако вместо массива функций в этом примере используется альтернативный подход, поддерживаемый в Async: передача объекта с каждой асинхронной задачей, указанной в виде свойства объекта. Когда все три задачи завершаются, результат выводится на консоль.

Листинг 5.12. Открытие трех файлов в параллельном режиме и чтение их контента

```
var fs = require('fs'),
  async = require('async');

try {

  async.parallel({
    data1 : function (callback) {
      fs.readFile('./data/data1.txt', 'utf8', function(err, data){
        callback(err,data);
      });
    },
    data2 : function (callback) {
      fs.readFile('./data/data2.txt', 'utf8', function(err, data){
        callback(err,data);
      });
    },
    data3 : function readData3(callback) {
      fs.readFile('./data/data3.txt', 'utf8', function(err, data){
        callback(err,data);
      });
    },
  }, function (err, result) {
    if (err) throw err;
    console.log(result);
  });
} catch(err) {
  console.log(err);
}
```

Результаты возвращаются в виде массива объектов, причем каждый результат связан с каждым из свойств. Пусть три файла данных в этом примере имеют следующий контент:

- data1.txt: apples;
- data2.txt: oranges;
- data3.txt: peaches.

В этом случае результат выполнения листинга 5.12 имел бы следующий вид:

```
{ data1: 'apples\n', data2: 'oranges\n', data3: 'peaches\n' }
```

Изучение других методов реализации потока управления модуля Async я решил оставить читателям этой книги в качестве упражнения. Помните только, что при работе с методами реализации потока управления модуля Async достаточно передать функцию обратного вызова каждой асинхронной задаче и вызвать эту функцию, когда работа будет завершена, передав ей объект ошибки (или `null`) и те данные, который вам необходимы.

Node-стиль

Несколько раз в этой главе упоминались специалисты, рекомендующие те или иные ограничения, например использование в Node-приложениях именованных, а не безымянных функций. В совокупности эти ограничения известны как *предпочтительный Node-стиль*, хотя никаких исчерпывающих руководств по этому стилю или определенного общепринятого набора предпочтений не существует. На самом деле по поводу Node-стиля есть несколько разных рекомендаций.

Одним из полезных руководств по Node-стилю является Felix's Node.js Style Guide, его можно найти по адресу http://nodeguide.com/style.html.

Вот некоторые рекомендации с моим мнением относительно каждого из них.

Вместо синхронных используйте асинхронные функции.

Да, это необходимое требование для Node-приложений.

Используйте в коде отступы из двух пробелов.

Извините, но я привык к трем пробелам и буду вставлять эти три пробела и дальше. Я полагаю, что важнее не изменять своим привычкам и не использовать символ табуляции. И вообще я не собираюсь переживать по поводу количества пробелов.

Используйте (не используйте) точки с запятой.

Чрезвычайно спорный вопрос. Я использую точки с запятой, но вам советую полагаться на собственное чутье.

Используйте одинарные кавычки.

Я привык к двойным кавычкам, но уже почти избавился от этой привычки. Как бы то ни было, лучше использовать двойные кавычки, чем отключать одинарную кавычку в строке эскейп-символом.

При определении нескольких переменных используйте (не используйте) только одно ключевое слово var.

В одних приложениях, рассматриваемых в данной книге, ключевое слово `var` используется для каждой переменной, в других нет. В данном случае опять-таки трудно избавиться от старых привычек, но я не думаю, что эта такая уж серьезная проблема, как некоторые ее пытаются выставить.

Константы должны иметь имена в верхнем регистре.

С этой рекомендацией я полностью согласен.

Имена переменных должны быть в смешанном регистре.

С этой рекомендацией я отчасти согласен, но не считаю, что ей нужно неукоснительно следовать.

Используйте оператор строгого равенства (===).

Звучит убедительно, но я повторюсь, что от старых привычек избавляться довольно трудно. Собираясь использовать строгое равенство, я часто указываю обычное равенство (==). Не повторяйте моих ошибок.

Давайте имена своим замыканиям.

И здесь я не на высоте. Это вполне разумный совет, и я стараюсь воплощать его в жизнь, но в большей части моего кода до сих пор встречаются безымянные функции.

Длина строки кода не должна превышать 80 символов.

Это тоже ценный совет.

Фигурные скобки должны открываться на той же строке, где расположен код, в котором они понадобились.

Я следую этому совету неукоснительно.

Самым главным правилом, которое следует запомнить в первую очередь, является правило использовать асинхронные функции при любой возможности. В конце концов, асинхронные функции — это сердце Node.

6 Маршрутизация трафика, служебные файлы и связующее программное обеспечение

После щелчка на ссылке, размещенной на веб-странице, ожидается, что что-то должно произойти — обычно загрузка страницы. Однако фактически перед загрузкой веб-ресурса случается множество событий, часть из которых оказывается вне нашего контроля (например, маршрутизация пакетов), а часть зависит от установленного нами программного обеспечения, способного правильно реагировать на тот или иной контент ссылки.

Разумеется, при использовании таких веб-серверов, как Apache, и такого программного обеспечения, как Drupal, основная часть работы по обслуживанию файла или ресурса происходит «за сценой». Однако при создании в Node собственных серверных приложений в обход традиционной технологии нам приходится принимать в этом более активное участие, чтобы нужные ресурсы доставлялись по назначению в нужное время.

В этой главе основное внимание уделяется технологии, которую Node-разработчики используют для реализации базовой маршрутизации и связующей функциональности, призванных гарантировать быструю и качественную доставку ресурса *А* пользователю *Б*.

Создание простого статического файлового сервера «с нуля»

У нас есть вся необходимая функциональность для создания простого маршрутизатора или для обработки статических файлов, встроенных непосредственно в Node.

Однако *иметь возможность* что-то сделать и суметь сделать это *без проблем* — абсолютно разные вещи.

Решая, что именно необходимо для создания простого, но работоспособного статического файлового сервера, можно прийти к следующему перечню:

1. Создание HTTP-сервера и прослушивание запросов.
2. При поступлении запроса синтаксический разбор URL-адреса запроса для определения местонахождения файла.
3. Проверка наличия файла.
4. Если нужного файла нет, выдача соответствующего ответа.
5. Если нужный файл существует, открытие файла для чтения.
6. Подготовка заголовка ответа.
7. Запись файла для ответа.
8. Ожидание следующего запроса.

Создание HTTP-сервера и чтение файлов требует использования модулей HTTP и File System. Пригодится также модуль Path, поскольку с его помощью перед открытием файла для чтения можно проверить, существует ли он. Кроме того, нам понадобится определить глобальную переменную для базового каталога или воспользоваться предопределенной переменной `__dirname` (более подробно о ней рассказывается далее в этой главе в соответствующей врезке).

К данному моменту в начальной части приложения должен располагаться следующий код:

```
var http = require('http'),
      path = require('path'),
       fs = require('fs'),
    base = '/home/examples/public_html';
```

В создании HTTP-сервера с помощью модуля HTTP нет ничего нового. И приложение может получить запрошенный документ путем непосредственного доступа к свойству `url` объекта HTTP-запроса. Для дополнительной проверки соответствия ответа запросам мы также выводим в `console.log` путь к запрашиваемому файлу. Это в дополнение к сообщению `console.log`, которое выводится при первом запуске сервера:

```
http.createServer(function (req, res) {

  pathname = base + req.url;
  console.log(pathname);

}).listen(8124);

console.log('Server running at 8124/');
```

Перед попыткой открытия файла для чтения и записи HTTP-ответа приложению нужно проверить факт наличия этого файла. Для этого вполне подойдет функция `path.exists`. Если такого файла нет, то в ответ нужно записать краткое сообщение

document not found (документ не найден) и установить значение 404 в качестве кода состояния:

```
path.exists(pathname, function(exists) {
   if (exists) {
      // сюда нужно вставить код обработки запроса
   } else {
        res.writeHead(404);
        res.write('Bad request 404\n');
        res.end();
}
```

А теперь мы подходим к основной части нового приложения. В примерах предыдущих глав мы использовали для чтения из файла метод `fs.readFile`. Но проблема метода `fs.readFile` заключается в том, что ему нужно прочитать весь файл в память, прежде чем он станет доступным.

Документы, обрабатываемые через Интернет, могут быть очень большими. Кроме того, в любой момент времени к документу могут обращаться множество запросов, а функции, подобные `fs.readFile`, просто не подходят для масштабирования.

В Node 0.8 вместо метода path.exists рекомендуется использовать метод fs.exists. В файлы примеров, на которые была ссылка во введении, включены приложения, поддерживаемые в обеих средах.

Вместо использования метода `fs.readFile` приложение создает поток чтения с помощью метода `fs.createReadStream` с параметрами, предлагаемыми по умолчанию. После этого достаточно *по каналу* передать контент файла непосредственно HTTP-объекту ответа. Поскольку по завершении своей работы поток ввода-вывода отправляет сигнал окончания, вызывать метод `end` с потоком чтения не нужно:

```
res.setHeader('Content-Type', 'test/html');

// код состояния 200 найден, значит, ошибок нет
res.statusCode = 200;

// создание потока чтения и направление его в канал
var file = fs.createReadStream(pathname);
file.on("open", function() {
   file.pipe(res);
});
file.on("error", function(err) {
  console.log(err);
});
```

В потоке чтения нас интересуют два события: `open` и `error`. Событие `open` происходит, когда поток готов к работе, а событие `error` — когда возникает проблема. Для события `open` приложение вызывает в функции обратного вызова метод `pipe`.

На данный момент статический файловый сервер выглядит как приложение, показанное в листинге 6.1.

Листинг 6.1. Простой статический файловый веб-сервер

```
var http = require('http'),
    path = require('path'),
    fs   = require('fs'),
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   path.exists(pathname, function(exists) {
      if (!exists) {
        res.writeHead(404);
        res.write('Bad request 404\n');
        res.end();
      } else {
         res.setHeader('Content-Type', 'text/html');

         // код состояния 200 найден, значит, ошибок нет
         res.statusCode = 200;

         // создание потока чтения и направление его в канал
         var file = fs.createReadStream(pathname);
         file.on("open", function() {
            file.pipe(res);
         });
         file.on("error", function(err) {
           console.log(err);
         });
       }
    });
}).listen(8124);

console.log('Server running at 8124/');
```

Я проверил это приложение с простым HTML-файлом, в котором не содержится ничего кроме элемента `img`, и файл загрузился и отобразился без проблем:

```
<!DOCTYPE html>
<head>
   <title>Test</title>
   <meta charset="utf-8" />
</head>
<body>
<img src="./phoenix5a.png" />
</body>
```

Затем я протестировал его с другим имеющимся у меня файлом примера, в котором содержался элемент video из HTML5:

```
<!DOCTYPE html>
<head>
   <title>Video</title>
   <meta charset="utf-8" />
</head>
<body>
   <video id="meadow" controls>
      <source src="videofile.mp4" />
      <source src="videofile.ogv" />
      <source src="videofile.webm" />
   </video>
</body>
```

Хотя при просмотре страницы с помощью браузера Chrome файл открывался и видео воспроизводилось, при тестировании страницы в Internet Explorer 10 элемент video не работал. Причина этого становится понятной при взгляде на консоль:

```
Server running at 8124/
/home/examples/public_html/html5media/chapter1/example2.html
/home/examples/public_html/html5media/chapter1/videofile.mp4
/home/examples/public_html/html5media/chapter1/videofile.ogv
/home/examples/public_html/html5media/chapter1/videofile.webm
```

Хотя браузер IE10 способен воспроизводить видео в формате MP4, он проверяет все три видеоклипа, поскольку тип контента в заголовке ответа для каждого из них указан как text/html. Хотя другие браузеры игнорируют неправильный тип контента и воспроизводят мультимедийный контент должным образом, IE так не делает.

Это приложение является вполне удачным примером того, что серверные приложения нужно проверять во всех ключевых браузерах, хотя, казалось бы, достаточно было протестировать его только в одном браузере, поскольку тестируется функциональность серверной стороны.

В приложение следует внести изменения, чтобы оно проверяло расширение имени каждого файла с последующим возвращением в заголовке ответа соответствующего MIME-типа. Хотя код для этого мы можем создать самостоятельно, я бы предпочел использовать готовый модуль node-mime.

Модуль node-mime можно установить с помощью диспетчера Node-пакетов: npm install mime. Сайт GitHub находится по адресу https://github.com/broofa/node-mime.

Для заданного имени файла (с указанием или без указания пути) модуль node-mime может вернуть правильный MIME-тип, также он может вернуть расширения файлов для заданного типа контента. Модуль node-mime добавляется к числу требуемых модулей с помощью следующего кода:

```
mime = require('mime');
```

Возвращаемый им тип контента используется в заголовке ответа, а также выводится на консоль, чтобы мы могли проконтролировать значение при тестировании приложения:

```
// тип контента
var type = mime.lookup(pathname);
console.log(type);
res.setHeader('Content-Type', type);
```

Теперь при обращении к файлу в элементе `video` в IE10 видеоклип будет воспроизведен.

Однако если мы обратимся не к файлу, а к каталогу, приложение работать не будет. В этом случае на консоль выводится сообщение об ошибке, а страница для пользователя остается пустой:

```
{ [Error: EISDIR, illegal operation on a directory] errno: 28, code: 'EISDIR' }
```

Нам нужно не только проверить существование того ресурса, к которому идет обращение, но и проверить, что собой представляет этот ресурс: файл или каталог. Если обращение происходит к каталогу, мы можем вывести либо его содержимое, либо сообщение об ошибке; выбор тут остается за разработчиком.

В финальной версии минимального статического файлового сервера, представленной в листинге 6.2, для проверки факта существования запрошенного объекта и его принадлежности к файлу используется метод `fs.stats`. Если ресурс отсутствует, в качестве кода состояния возвращается HTTP-код 404. Если ресурс имеется, но является каталогом, в качестве кода состояния возвращается HTTP-код ошибки 403, forbidden (запрещенный ресурс). Во всех случаях запрос обрабатывается правильно.

Листинг 6.2. Финальная версия минимального статического файлового сервера

```
var http = require('http'),
    url =  require('url'),
    fs   = require('fs'),
    mime = require('mime');
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   fs.stat(pathname, function(err, stats) {
      if (err) {
```

продолжение ⇩

Листинг 6.2 *(продолжение)*

```
        res.writeHead(404);
        res.write('Bad request 404\n');
        res.end();
      } else if (stats.isFile()) {
        // тип контента
        var type = mime.lookup(pathname);
        console.log(type);
        res.setHeader('Content-Type', type);

        // код состояния 200 найден, значит, ошибок нет
        res.statusCode = 200;

        // создание потока чтения и направление его в канал
        var file = fs.createReadStream(pathname);
        file.on("open", function() {

          file.pipe(res);
        });
        file.on("error", function(err) {
          console.log(err);
        });
      } else {
        res.writeHead(403);
        res.write('Directory access is forbidden');
        res.end();
      }
    });
}).listen(8124);
console.log('Server running at 8124/');
```

Вот как выглядит вывод на консоль при доступе к единственной веб-странице, содержащей ссылки как на изображение, так и на видеофайл:

```
/home/examples/public_html/html5media/chapter2/example16.html
text/html
/home/examples/public_html/html5media/chapter2/bigbuckposter.jpg
image/jpeg
/home/examples/public_html/html5media/chapter2/butterfly.png
image/png
/home/examples/public_html/favicon.ico
image/x-icon
/home/examples/public_html/html5media/chapter2/videofile.mp4
video/mp4
/home/examples/public_html/html5media/chapter2/videofile.mp4
video/mp4
```

Обратите внимание на соответствующую обработку типов контента. На рис. 6.1 показана веб-страница, содержащая элемент `video` и загруженная в браузер Chrome, и обращение к сети, отображенное в консоли браузера.

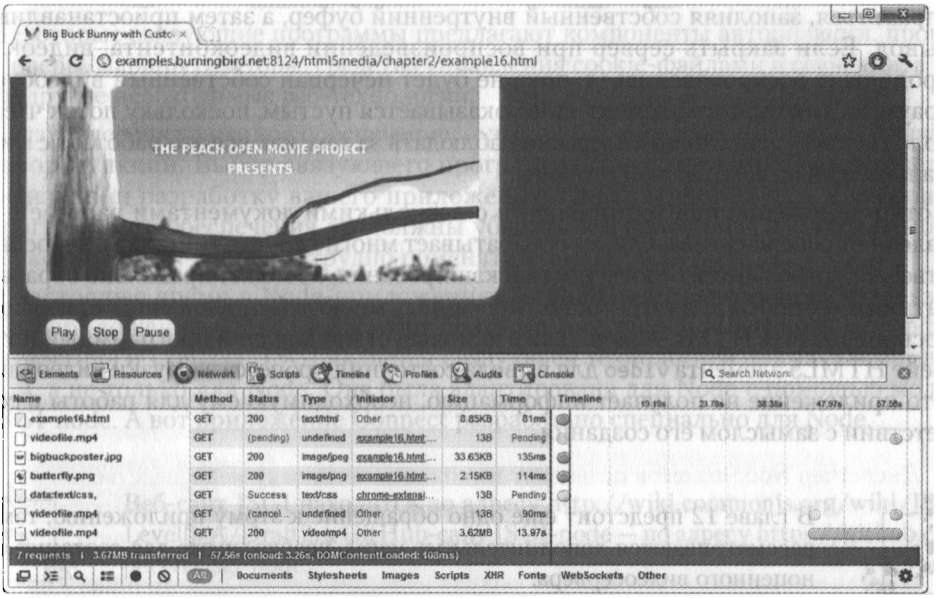


Рис. 6.1. Вид консоли браузера при загрузке веб-страницы, обслуживаемой простым статическим файловым сервером из листинга 6.2

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ __DIRNAME

В некоторых примерах этой книги для веб-документов указываются фиксированные места, например /home/examples/public_html. Вы можете удивиться, почему для указания текущего рабочего каталога Node-приложения я не использую предопределенную переменную __dirname. Хотя в примерах этой главы я обращался к файлам, хранящимся отдельно от моего Node-приложения, вы должны знать о существовании переменной __dirname и о ее применении в Node-разработке, поскольку она позволяет тестировать приложения и затем передавать их в эксплуатацию без изменения значения переменной, указывающей на основное местоположение файлов.

Переменную __dirname нужно использовать следующим образом:

```
var pathname = __dirname + req.url;
```

Обратите внимание на то, что в имени этой переменной используется двойной знак подчеркивания.

Вы получите более полное представление о функционировании потока чтения при загрузке страницы, имеющей элемент `video` и начинающей его воспроизводить. Браузер захватывает вывод потока чтения на той скорости, с которой может

справиться, заполняя собственный внутренний буфер, а затем приостанавливает вывод. Если закрыть сервер при воспроизведении видеоконтента, видеоклип продолжит воспроизводиться, пока не будет исчерпан собственный видеобуфер браузера. После этого элемент `video` оказывается пустым, поскольку поток чтения более недоступен. Очень интересно наблюдать за тем, как все это работает с минимальным участием с нашей стороны.

Хотя приложение при тестировании с несколькими документами работает, оно далеко от совершенства. Оно не обрабатывает многие другие типы веб-запросов, не отвечает требованиям безопасности и кэширования, не может правильно обрабатывать видеозапросы. Одно из протестированных мною веб-приложений, призванных воспроизводить HTML-видео, также использует прикладной программный интерфейс HTML5 элемента `video` для вывода состояния процесса загрузки видеоклипа. Это приложение не получает информацию, необходимую ему для работы в соответствии с замыслом его создания.

В главе 12 предстоит еще одно обращение к этому приложению, там же рассматриваются дополнительные меры, необходимые для создания полноценного видеосервера.

Существует множество мелких нюансов, которые нужно преодолеть на пути создания статического файлового сервера. Другой подход предполагает использование существующего статического файлового сервера. В следующем разделе рассматривается один из таких серверов, входящий в состав связующего программного модуля Connect.

Связующее программное обеспечение

Что такое *связующее программное обеспечение* (middleware)? Хороший вопрос, но он, к сожалению, относится к разряду вопросов, не имеющих точного ответа.

В целом связующим называют программное обеспечение, находящееся между вами, как разработчиком, и той системой, на которой вы работаете. Под *системой* понимается либо операционная система, либо положенная в основу разработки технология, например предлагаемая Node. Точнее говоря, связующее программное обеспечение встраивается в коммуникационную цепочку между вашим приложением и основной системой, то есть связующее программное обеспечение — это скорее не термин, а описательное название.

Например, вместо того чтобы предоставлять всю функциональность, необходимую для обслуживания статических файлов через веб-сервер, для обработки большей их части можно воспользоваться связующим программным обеспечением. Связующее программное обеспечение возьмет все заботы на себя, а вы сможете сконцентрироваться на уникальных для вашего приложения аспектах. Однако связующее программное обеспечение не ограничивается обслуживанием статических файлов.

Некоторые связующие программы предлагают компоненты авторизации, прокси-серверы, маршрутизаторы, средства управления cookie-файлами и сеансами, а также другие необходимые веб-технологии.

Связующее программное обеспечение — это не библиотека утилит и не обычный набор функций. Выбор связующего программного обеспечения предопределяет и дизайн, и разработку вашего приложения. Перед встраиванием связующего программного обеспечения вы должны убедиться в правильности своего выбора, поскольку будет довольно трудно поменять что-то на полпути.

В настоящее время в Node-приложениях используются два основных связующих приложения: JSGI (JavaScript Gateway Interface — шлюзовый интерфейс JavaScript) и Connect. JSGI является связующей технологией для JavaScript в целом, а не только для Node. Она была адаптирована для Node с помощью модуля JSGI-node. А вот приложение Connect разработано специально для Node.

Веб-сайт JSGI находится по адресу http://wiki.commonjs.org/wiki/JSGI/Level0/A/Draft2, а GitHub-сайт JSGI-node — по адресу https://github.com/persvr/jsgi-node.

В этой книге рассматривается только технология Connect, и для этого есть три причины. Во-первых, ее проще использовать. Технология JSGI требует слишком много времени изучения общих принципов ее работы (независимо от использования вместе с Node), а Connect позволяет войти в курс дела сразу. Во-вторых, Connect обеспечивает поддержку связующим программным обеспечением весьма популярной среды Express (см. главу 7). В-третьих, и это, возможно, самая важная причина, похоже, что технология Connect постепенно заняла лидирующие позиции в своей категории. Судя по данным реестра npm, это наиболее востребованное связующее программное обеспечение.

Введение в Connect 2.0 можно найти по адресу http://tjholowaychuk.com/post/18418627138/connect-2-0. Исходный код Connect находится по адресу https://github.com/senchalabs/Connect. (Дополнительные сведения по установке даны далее во врезке «Альфа-модули».)

Основы Connect

Connect можно установить с помощью диспетчера Node-пакетов:

```
npm install connect
```

По сути, Connect является платформой, на которой можно использовать одно или несколько связующих приложений. Хотя документации по Connect довольно мало, работать с этой технологией относительно просто, достаточно познакомиться с несколькими работающими примерами.

АЛЬФА-МОДУЛИ

Когда писался первый черновой вариант этой главы, в реестре npm имелась стабильная версия (1.8.5) Connect, но я хотел рассказать о версии 2.x, которая еще находилась в разработке, поскольку вам, скорее всего, придется столкнуться именно с ней.

Я загрузил исходный код Connect 2.x непосредственно с GitHub и перешел в каталог моей среды разработки node_modules. Затем я перешел в каталог Connect и установил этот модуль с помощью диспетчера Node-пакетов, но без указания имени модуля и с указанием ключа -d для установки тех модулей, от которых зависит его работа:

npm install –d

Диспетчер Node-пакетов позволяет выполнить установку непосредственно из хранилища Git. Можно также задействовать Git напрямую для клонирования версии, а затем для установки использовать только что описанный прием.

Имейте в виду, что при установке модуля непосредственно из источника с последующим обновлением с помощью диспетчера Node-пакетов диспетчер перепишет модуль, выбрав ту его версию, которую он посчитает «самой последней», даже если вы использовали новейшую версию модуля.

В листинге 6.3 с помощью модуля Connect создается простое серверное приложение, использующее две связующие программы[1] в комплекте с Connect: `connect.logger` и `connect.favicon`. Связующее приложение `logger` записывает все запросы в поток ввода-вывода, в данном случае, в предлагаемый по умолчанию поток `STDIO.output`, а `favicon` обслуживает файл favicon.ico. Приложение включает связующее программное обеспечение, применив метод `use` к слушателю Connect-запросов, который затем передается в качестве параметра методу `createServer` HTTP-объекта.

Листинг 6.3. Встраивание связующих программ logger и favicon в приложение на базе модуля Connect

```
var connect = require('connect');
var http = require('http');

var app = connect()
   .use(connect.favicon())
   .use(connect.logger())
   .use(function(req,res) {
      res.end('Hello World\n');
   });

http.createServer(app).listen(8124);
```

[1] В Connect отдельные связующие решения указываются просто как «связующие программы». В этой главе я тоже придерживаюсь данного соглашения.

Вы можете использовать любое количество связующих программ, либо встроенных в модуль Connect, либо предоставленных сторонними разработчиками, для этого достаточно добавить соответствующие инструкции `use`.

Чтобы предварительно не создавать слушатель Connect-запросов, связующую программу модуля Connect можно встроить непосредственно в метод `createServer`, как показано в листинге 6.4.

Листинг 6.4. Непосредственное встраивание в приложение связующей программы модуля Connect

```
var connect = require('connect');
var http = require('http');

http.createServer(connect()
   .use(connect.favicon())
   .use(connect.logger())
   .use(function(req,res) {
      res.end('Hello World\n');
   })).listen(8124);
```

Связующие программы модуля Connect

В комплекте с Connect поставляется как минимум 20 связующих программ. Я не собираюсь в этом разделе рассматривать все эти программы, а расскажу ровно о том их количестве, которое позволит хорошо разобраться в их совместной работе.

Другие примеры связующих программ модуля Connect используются в Express-приложениях, о которых рассказывается в главе 7.

Программа connect.static

Ранее мы уже создавали упрощенный статический файловый сервер «с нуля». Connect предоставляет связующую программу, которая не только реализует функциональность такого сервера, но и предлагает дополнительные возможности. Пользоваться ею чрезвычайно просто, достаточно указать `connect.static` в качестве параметра связующей программы, передав при этом для всех запросов корневой каталог. Следующий фрагмент реализует основную часть всего того, что делалось в листинге 6.2, но в меньшем объеме кода:

```
var connect = require('connect'),
    http = require('http'),
    __dirname = '/home/examples';

http.createServer(connect()
   .use(connect.logger())
   .use(connect.static(_dirname + '/public_html'), {redirect: true})
   ).listen(8124);
```

Связующая программа `connect.static` в качестве первого параметра получает путь к корневому каталогу, а в качестве второго необязательного параметра — объект. Вот возможные значения этого объекта:

`maxAge`

Период хранения данных в кэше браузера в миллисекундах (по умолчанию — 0).

`hidden`

Устанавливается в `true`, чтобы позволить перемещение скрытых файлов (по умолчанию — `false`).

`redirect`

Устанавливается в `true` для перенаправления по замыкающему слэшу `/`, когда в качестве имени пути указан каталог.

Возможности этого маленького приложения, использующего связующий модуль Connect, значительно отличаются от возможностей приложения, которое мы создали ранее «с нуля». Решение на основе Connect обрабатывает кэш браузера, защищает от неправильных URL-адресов и правильнее обрабатывает HTTP-элемент `video` из HTML5, чем не мог похвастать созданный нами сервер. Его единственным недостатком по сравнению с предыдущим сервером является ограниченная обработка ошибок. Тем не менее связующая программа `connect.static` предоставляет браузеру соответствующий ответ и код состояния.

В только что показанном коде и в ранее показанных примерах этого раздела демонстрировалась еще одна связующая программа модуля Connect — `connect.logger`. Ее мы сейчас и рассмотрим.

Программа connect.logger

Связующая программа `logger` записывает все запросы в поток ввода-вывода — по умолчанию это `stdout`. Вы можете изменить поток ввода-вывода, а также другие параметры, включая продолжительность хранения данных в буфере, формат и флаг `immediate`, сигнализирующий о том, нужно ли регистрировать запись немедленно или по ответу.

В дополнение к четырем предопределенным форматам существует еще несколько лексем, с помощью которых можно создавать строку формата:

```
default
    ':remote-addr - - [:date] ":method :url HTTP/:http-version" :status
    :res[content-length] ":referrer" ":user-agent"'
short
    ':remote-addr - :method :url HTTP/:http-version :status :res[content-
    length] - :response-time ms'
tiny
    ':method :url :status :res[content-length] - :response-time ms'
dev
```

Краткий результат, цвет которого соответствует состоянию ответа; используется при разработке.

Формат, предлагаемый по умолчанию, генерирует в журнале записи, похожие на эту:

```
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:22 GMT] "GET /example1.html HTTP/1.1" 304
- "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.56 Safari/535.11"
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:22 GMT] "GET /phoenix5a.png HTTP/1.1" 304
- "http://examples.burningbird.net:8124/example1.html" "Mozilla/5.0 (Windows NT 6.1;
WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11"
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:22 GMT] "GET /favicon.ico HTTP/1.1" 304
- "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.56 Safari/535.11"
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:28 GMT] "GET /html5media/chapter2/
example16.html HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11"
```

Это информация полезна, но слишком многословна. Кроме того, она напоминает формат записи в журнал, используемый по умолчанию в таких серверах, как Apache. Вы можете изменить формат, а также можете направить вывод в файл. В листинге 6.5 программа `connect.logger` направляет запись в файл и в качестве формата устанавливает предопределенный формат `dev`.

Листинг 6.5. Направление записи в файл и изменение формата журнала

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    __dirname = '/home/examples';

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

http.createServer(connect()
   .use(connect.logger({format : 'dev', stream : writeStream }))
   .use(connect.static(__dirname + '/public_html'))
   ).listen(8124);
```

Теперь журнал имеет следующий вид:

```
GET /example1.html 304 4ms
GET /phoenix5a.png 304 1ms
GET /favicon.ico 304 1ms
GET /html5media/chapter2/example16.html 304 2ms
GET /html5media/chapter2/bigbuckposter.jpg 304 1ms
GET /html5media/chapter2/butterfly.png 304 1ms
GET /html5media/chapter2/example1.html 304 1ms
GET /html5media/chapter2/bigbuckposter.png 304 0ms
GET /html5media/chapter2/videofile.mp4 304 0ms
```

Хотя он не настолько информативен, как предыдущий, это весьма удобное средство проверки состояния запроса и времени загрузки.

Программы connect.parseCookie и connect.cookieSession

В созданном нами «с нуля» файловом сервере не предоставлялась функциональность ни для работы с cookie-файлами, ни для обработки состояния сеанса. К счастью для нас, обе эти возможности предоставляются связующими программами модуля Connect.

Скорее всего, одним из первых ваших клиентских JavaScript-приложений было создание cookie-файла HTTP-запроса. Связующая программа `connect.parseCookie` предоставляет возможность обращения к данным cookie-файла на сервере. Она проводит синтаксический разбор заголовка cookie-файла, заполняя файл `req.cookies` парами cookie/данные. В листинге 6.6 показан простой веб-сервер, извлекающий cookie-файл для ключевого значения `username` и записывающий не очень притязательное, но вполне уместное сообщение в поток `stdout`.

Листинг 6.6. Доступ к cookie-файлам HTTP-запроса и вывод сообщения на консоль с их помощью

```
var connect = require('connect')
  , http = require('http');

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.cookieParser())
  .use(function(req, res, next) {
     console.log('tracking ' + req.cookies.username);
     next();
   })
  .use(connect.static('/home/examples/public_html'));

http.createServer(app).listen(8124);
console.log('Server listening on port 8124');
```

К вопросам использования анонимной функции и особенно к назначению функции `next` я перейду в разделе «Создание пользовательских связующих программ для модуля Connect». А здесь мы видим, что связующая программа `connect.cookieParser` перехватывает входящий запрос, извлекает данные cookie-файла из заголовка и сохраняет данные в объекте запроса. Затем анонимная функция обращается к данным `username` из объекта `cookies`, выводя их на консоль.

Чтобы создать cookie-файл HTTP-ответа, мы использовали в паре с программой `connect.parseCookie` программу `connect.cookieSession`, обеспечивающую безопасную сохранность сеанса. Текст в виде строки передается функции `connect.cookieParser`, предоставляя секретный ключ для данных сеанса. Данные добавляются непосредственно в объект сеанса. Для очистки данных сеанса объекту сеанса нужно присвоить значение `null`.

В листинге 6.7 создаются две функции — одна для очистки данных сеанса, другая для вывода сообщения трассировки, — они используются в качестве связующего программного обеспечения для входящих запросов. Они добавляются как связующие программы в дополнение к программам `logger`, `parseCookie`, `cookieSession` и `static`. Пользователю дается приглашение на ввод его имени пользователя

на странице клиента, затем это имя служит для указания cookie-файла запроса. На сервере имя пользователя и количество запрашиваемых пользователем ресурсов в текущем сеансе сохраняются с помощью закодированного cookie-файла ответа.

Листинг 6.7. Использование cookie-файла сеанса для отслеживания доступа к ресурсам

```
var connect = require('connect')
    , http = require('http');

// очистка всех данных сеанса
function clearSession(req, res, next) {
  if ('/clear' == req.url) {
    req.session = null;
    res.statusCode = 302;
    res.setHeader('Location', '/');
    res.end();
  } else {
    next();
  }
}

// отслеживание пользователя
function trackUser(req, res, next) {
  req.session.ct = req.session.ct || 0;
  req.session.username = req.session.username || req.cookies.username;
  console.log(req.cookies.username + ' requested ' +
                      req.session.ct++ + ' resources this session');
  next();
}

// cookie-файлы и сеанс
var app = connect()
  .use(connect.logger('dev'))
  .use(connect.cookieParser('mumble'))
  .use(connect.cookieSession({key : 'tracking'}))
  .use(clearSession)
  .use(trackUser);

// статический сервер
app.use(connect.static('/home/examples/public_html'));
// запуск сервера и прослушивание
http.createServer(app).listen(8124);
console.log('Server listening on port 8124');
```

На рис. 6.2 показана веб-страница, доступ к которой получен через серверное приложение из листинга 6.8. JavaScript-консоль открыта для вывода обоих cookie-файлов. Заметьте, что cookie-файл ответа, в отличие от cookie-файла запроса, закодирован.

Количество документов, к которым получил доступ пользователь, отслеживается либо до тех пор, пока пользователь не обратится к URL-адресу `/clear` (в таком

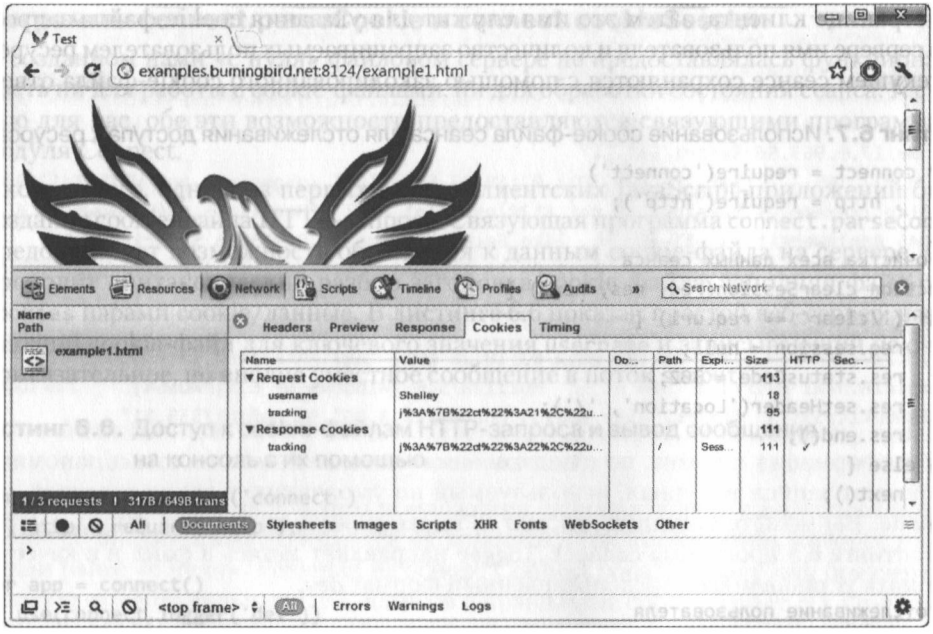


Рис. 6.2. Открытая в Chrome JavaScript-консоль показывает cookie-файлы запроса и ответа

случае объект сеанса устанавливается в `null`), либо пока не будет закрыт браузер, завершив этим сеанс.

В листинге 6.7 также используется пара нестандартных связующих функций. В следующем (и заключительном) разделе, посвященном модулю Connect, мы рассмотрим, как они взаимодействуют с Connect и как создается связующее программное обеспечение сторонних производителей.

Создание пользовательских связующих программ для модуля Connect

В предыдущем разделе мы создали две функции в качестве связующих программ модуля Connect, чтобы обрабатывать входящие сообщения перед их передачей финальному статическому серверу (см. листинг. 6.7). Тремя параметрами, передаваемыми функциям, были HTTP-объекты запроса и ответа, а также функция обратного вызова `next`. Эти три параметра формируют характерную структуру связующей функции модуля Connect.

Чтобы более детально изучить работу связующего программного обеспечения модуля Connect, давайте изучим одну из функций, которые использовались в предыдущем коде, а именно — функцию `connect.favicon`. Это не что иное, как простая функция, которая либо обслуживает применяемый по умолчанию файл `favicon.ico`, либо предлагает нестандартный путь:

```
connect()
.use (connect.favicon('someotherloc/favicon.ico'))
```

Причина, по которой мы рассматриваем именно функцию `connect.favicon`, если не считать ее полезности, заключается в том, что это одна из самых простых связующих программ и поэтому ее проще изучать.

Исходный код `connect.favicon`, особенно по сравнению с другими, показывает, что все экспортируемые связующие программы возвращают некую функцию со следующей минимальной сигнатурой, или профилем:

```
return function(req, res, next)
```

Обратный вызов `next`, передаваемый функции в качестве последнего параметра, происходит, если связующая функция вообще не обрабатывает текущий запрос или не обрабатывает его до конца. Кроме того, обратный вызов `next` происходит, если в связующей функции произошла ошибка, в этом случае в качестве параметра возвращается объект `error`, как показано в листинге 6.8.

Листинг 6.8. Связующая функция favicon модуля Connect

```
module.exports = function favicon(path, options){
  var options = options || {}
    , path = path || __dirname + '/../public/favicon.ico'
    , maxAge = options.maxAge || 86400000;

  return function favicon(req, res, next){
    if ('/favicon.ico' == req.url) {
      if (icon) {
        res.writeHead(200, icon.headers);
        res.end(icon.body);
      } else {
        fs.readFile(path, function(err, buf){
          if (err) return next(err);
          icon = {
            headers: {
                'Content-Type': 'image/x-icon'
              , 'Content-Length': buf.length
              , 'ETag': '"' + utils.md5(buf) + '"'
              , 'Cache-Control': 'public, max-age=' + (maxAge / 1000)
            },
            body: buf
          };
          res.writeHead(200, icon.headers);
          res.end(icon.body);
        });
      }
    } else {
      next();
    }
  };
};
```

Разумеется, обратный вызов `next` — это способ последовательного вызова функций, выстроенных в цепочку. Если при поступлении входящего запроса связующая функция может полностью обработать запрос, например запрос `favicon.ico`, никакая другая связующая функция не вызывается. Именно поэтому вы должны включить в свое приложение перед `connect.logger` связующую функцию `connect.favicon`, позволяющую предотвратить засорение журналов запросами `favicon.ico`:

```
http.createServer(connect()
   .use(connect.favicon('/public_html/favicon.ico'))
   .use(connect.logger())
   .use(connect.static(_dirname + '/public_html'))
   ).listen(8124);
```

Вы уже узнали, как непосредственно в приложении создать собственную нестандартную связующую функцию для модуля Connect и как выглядит связующая функция, входящая в модуль Connect. Однако вы пока не знаете, как создается связующее программное обеспечение сторонних производителей, не предназначенное для непосредственного встраивания в приложение.

Чтобы создать для модуля Connect внешнее связующее программного обеспечение, создайте обычный модуль, но убедитесь, что у него имеются все составные части, необходимые для модуля Connect, а именно: точно определены три параметра (`req`, `res` и `next`), причем `next` вызывается в случае, если запрос обрабатывается не до конца.

В листинге 6.9 для модуля Connect создается связующая функция, которая проверяет, что запрашиваемый файл существует и что это файл, а не каталог. Если запрашивается каталог, она возвращает код состояния 403 и специальное сообщение. Если файл не существует, она возвращает код состояния 404 и другое специальное сообщение. Если ни одного из этих событий не происходит, вызывается функция `next`, которая заставляет связующее программное обеспечение модуля Connect вызвать следующую функцию (в данном случае — `connect.static`).

Листинг 6.9. Создание нестандартного связующего модуля для обработки ошибок

```
var fs = require('fs');

module.exports = function customHandler(path, missingmsg, directorymsg) {
   if (arguments.length < 3) throw new Error('missing parameter
                                              in customHandler');
   return function customHandler(req, res, next) {
      var pathname = path + req.url;
      console.log(pathname);
      fs.stat(pathname, function(err, stats) {
         if (err) {
            res.writeHead(404);
            res.write(missingmsg);
            res.end();
         } else if (!stats.isFile()) {
            res.writeHead(403);
            res.write(directorymsg);
            res.end();
         } else {
```

```
            next();
        }
    });
  }
}
```

При возникновении ошибки она вбрасывается связующим программным обеспечением модуля Connect, но если ошибка возникает в возвращаемой функции, вызывается функция next с объектом ошибки:

```
next(err);
```

В следующем коде показано, как задействовать в приложении это нестандартное связующее программное обеспечение:

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    custom = require('./custom'),
    base = '/home/examples/public_html';

http.createServer(connect()
   .use(connect.favicon(base + '/favicon.ico'))
   .use(connect.logger())
   .use(custom(base + '/public_html', '404 File Not Found',
          '403 Directory Access Forbidden'))
   .use(connect.static(base))
   ).listen(8124);
```

В Connect имеется встроенная функция errorHandler, но она не подходит для той цели, которую мы пытаемся достичь. Она предназначена для форматированного вывода исключения. Вы познакомитесь с ней вместе с Express-приложением в главе 7.

Существует и другое встроенное связующее программное обеспечение, а также значительное количество связующих программ сторонних производителей, пригодных для использования с модулем Connect. Кроме того, Connect формирует связующий уровень для Express-приложения, рассматриваемого в главе 7. Но сначала давайте познакомимся с двумя другими типами служб, необходимыми для многих Node-приложений: маршрутизаторами (routers) и прокси-серверами (proxies).

Маршрутизаторы

Маршрутизаторы получают данные из одного источника и направляют их в другое место. Обычно передается пакет данных, но на уровне приложения это может быть также запрос ресурса. Если вы уже пользовались Drupal или WordPress, то видели маршрутизатор в действии. Предположим, какие-либо варианты перенаправления URL-адресов отсутствуют и остается только обращение ваших читателей к статье (article) по следующему URL-адресу:

```
http://yourplace.org/article/your-title
```

Тогда бы в реальности они использовали следующий URL-адрес:

```
http://yourplace.org/node/174
```

Последний URL-адрес является примером маршрутизатора в действии. URL предоставляет информацию о том, что должно сделать веб-приложение, в данном случае:

- обратиться к базе данных узла (под узлом `node` в данном случае понимается Drupal-узел);
- найти и вывести узел с идентификатором 174.

Другой вариант URL-адреса может иметь следующий вид:

```
http://yourplace.org/user/3
```

Здесь опять осуществляется обращение к базе данных, а также ведется поиск и вывод информации о пользователе (user), имеющем идентификатор 3.

В Node маршрутизатор служит в основном для извлечения нужной информации из URI-идентификатора — обычно в соответствии с неким образцом — и с целью использования этой информации для запуска нужного процесса передает извлеченную информацию этому процессу.

Node-разработчикам доступно несколько маршрутизаторов, включая встроенный в Express, но я собираюсь показать один из наиболее популярных маршрутизаторов — Crossroads.

Основной сайт Crossroads находится по адресу http://millermedeiros.github.com/crossroads.js/.

Модуль маршрутизатора Crossroads можно загрузить с помощью диспетчера Node-пакетов:

```
npm install crossroads
```

Модуль предлагает обширный и хорошо документированный API-интерфейс, но я собираюсь сконцентрироваться только на трех методах:

`addRoute`

Определяет новый эталонный слушатель маршрута.

`parse`

Проводит анализ строки и при нахождении соответствия обеспечивает диспетчеризацию строки по нужному маршруту.

`matched.add`

Отображает обработчик маршрута на соответствующий маршрут.

Определение маршрута осуществляется с помощью регулярного выражения, которое может содержать фигурные скобки ({}), задающие границы именованных переменных, передаваемых обработчику маршрута. Например:

```
{type}/{id}
node/{id}
```

Оба этих паттерна маршрутизации будут соответствовать адресу:

```
http://something.org/node/174
```

Разница в том, что параметр type передается обработчику маршрута при использовании только первого паттерна.

Для обозначения необязательных сегментов служит двоеточие (:):

```
category/:type:/:id:
```

Это выражение будет соответствовать любому из перечисленных:

```
category/
category/tech/
category/history/143
```

Для запуска обработчика маршрута проводится синтаксический разбор запроса:

```
parse(request);
```

Если запрос соответствует каким-нибудь имеющимся функциям обработки маршрута, то вызываются эти обработчики.

В листинге 6.10 создается простое приложение, которое ищет любую заданную категорию, а также дополнительно публикацию и статью публикации. Оно осуществляет вывод данных на консоль, то есть выполняет действие, указанное в запросе.

Листинг 6.10. Использование модуля Crossroads для маршрутизации URL-запроса в соответствии с указанными действиями

```
var crossroads = require('crossroads'),
    http = require('http');

crossroads.addRoute('/category/{type}/:pub:/:id:', function(type,pub,id) {
   if (!id && !pub) {
     console.log('Accessing all entries of category ' + type);
     return;
   } else if (!id) {
     console.log('Accessing all entries of category ' + type +
                 ' and pub ' + pub);
     return;
   } else {
     console.log(' Accessing item ' + id + ' of pub ' + pub +
                 ' of category ' + type);
   }
});
http.createServer(function(req,res) {

  crossroads.parse(req.url);
  res.end('and that\'s all\n');
}).listen(8124);
```

Рассмотрим запросы:

```
http://examples.burningbird.net:8124/category/history
http://examples.burningbird.net:8124/category/history/journal
http://examples.burningbird.net:8124/category/history/journal/174
```

Эти запросы генерируют вывод на консоль следующих сообщений:

```
Accessing all entries of category history
Accessing all entries of category history and pub journal
Accessing item 174 of pub journal of category history
```

Чтобы программа функционировала примерно так же, как Drupal, где комбинируются тип объекта и идентификатор, в листинге 6.11 метод `matched.add` модуля Crossroads служит для отображения обработчика маршрута на конкретный маршрут.

Листинг 6.11. Отображение обработчика маршрута на заданный маршрут

```
var crossroads = require('crossroads'),
    http = require('http');

var typeRoute = crossroads.addRoute('/{type}/{id}');

function onTypeAccess(type,id) {
   console.log('access ' + type + ' ' + id);
};

typeRoute.matched.add(onTypeAccess);

http.createServer(function(req,res) {

  crossroads.parse(req.url);
  res.end('processing');
}).listen(8124);
```

Это приложение найдет соответствие для любого из следующих маршрутов:

```
/node/174
/user/3
```

При маршрутизации обычно происходит обращение к базе данных с целью получения контента для возвращаемой страницы. Маршрутизация может также применяться с модулями связующего программного обеспечения или платформы с целью обработки поступающих запросов, хотя эти приложения могут также предлагать собственные программы маршрутизации. В следующем разделе демонстрируется использование модуля Crossroads с модулем Connect и прокси-сервером.

Прокси-серверы

Прокси-сервер является средством маршрутизации запросов из нескольких разных источников через один сервер. Причинами применения прокси-сервера являются: кэширование, требования безопасности и даже сокрытие инициатора запроса.

Например, общедоступные прокси-серверы использовались для ограничения доступа некоторой категории людей к конкретному веб-контенту, имитируя, что запрос исходит из какого-то места, отличающегося от фактического источника запроса. Такой тип прокси-сервера называется *прямым прокси-сервером* (forward proxy).

Обратный прокси-сервер (reverse proxy) является средством управления отправкой запросов на сервер. Например, у вас могут быть пять серверов, но вы не хотите, чтобы люди имели непосредственный доступ к четырем из них. Тогда вы направляете весь трафик через пятый сервер, который исполняет роль посредника при запросах к другим серверам. Обратные прокси-серверы могут также использоваться для балансирования нагрузки и повышения общей производительности системы путем кэширования запросов по мере их поступления.

Еще один тип прокси-сервера служит для предоставления локальной службы облачной службе. Примером этого типа прокси-сервера может являться reddish-proxy, который предоставляет локальный Redis-экземпляр новой Reddish-службе по адресу https://reddi.sh/.

В Node наиболее популярным прокси-сервером является модуль http-proxy. Этот модуль предлагает все варианты прокси-серверов, которые я только могу себе представить, и некоторые варианты, о которых я не имею представления. Он дает возможность использовать прямые и обратные прокси-серверы, может применяться с WebSockets, поддерживает HTTPS и может быть скрытым. Он используется на популярном вебсайте nodejitsu.com, так что, как утверждают его создатели, он уже закален в боях.

GitHub-страница http-proxy находится по адресу https://github.com/nodejitsu/node-http-proxy.

Установите модуль http-proxy с помощью диспетчера Node-пакетов:

```
npm install http-proxy
```

Самым простым вариантом использования модуля http-proxy является создание автономного прокси-сервера, который слушает входящие запросы на одном порту и перенаправляет их веб-серверу, слушающему другой порт:

```
var http = require('http'),
    httpProxy = require('http-proxy');

httpProxy.createServer(8124, 'localhost').listen(8000);

http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.write('request successfully proxied!' + '\n' +
    JSON.stringify(req.headers, true, 2));
  res.end();
}).listen(8124);
```

Это простое приложение занимается прослушиванием запросов на порту 8000 и перенаправляет их на HTTP-сервер, слушающий порт 8124.

Вывод в браузер после запуска этого приложения на моей системе был следующим:

```
Запрос успешно перенаправлен!
{
  "host": "examples.burningbird.net:8000",
  "connection": "keep-alive",
  "user-agent": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11
  (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11",
  "accept": "text/html,application/xhtml+xml,
    application/xml;q=0.9,*/*;q=0.8",
  "accept-encoding": "gzip,deflate,sdch",
  "accept-language": "en-US,en;q=0.8",
  "accept-charset": "ISO-8859-1,utf-8;q=0.7,*;q=0.3",
  "cookie": "username=Shelley",
  "x-forwarded-for": "99.190.71.234",
  "x-forwarded-port": "54344",
  "x-forwarded-proto": "http"
}
```

Все, что относится к использованию прокси-сервера, выделено полужирным шрифтом. Обратили внимание, что cookie-файлы по-прежнему берутся из ранее приведенного примера?

Модуль http-proxy можно также использовать из командной строки. В каталоге bin есть приложение командной строки, которому передаются порт, цель, файл конфигурации, ключ, показывающий, нужно ли замалчивать вывод регистрационной информации прокси-сервера, или ключ `-h` (для вывода справки). Для прослушивания порта 8000 и перенаправления на порт 8124 на адресе `localhost` воспользуйтесь следующей командой:

```
./node-http-proxy --port 8000 --target localhost:8124
```

Проще не бывает. Если нужно запустить прокси-сервер в фоновом режиме, нужно в конце команды поставить символ амперсанда (`&`).

Чуть позже в этой книге я покажу некоторые возможности модуля http-proxy с WebSockets и HTTPS, а сейчас мы объединим технологии, продемонстрированные в данной главе (статический файловый сервер, связующий модуль Connect, маршрутизатор Crossroads и прокси-сервер на основе модуля http-proxy), создав еще один последний пример, в котором сочетаются все эти компоненты.

В листинге 6.12 я задействовал модуль http-proxy для проверки динамических входящих запросов (URL-адрес запроса начинается с `/node/`). При нахождении соответствия маршрутизатор перенаправляет запрос серверу с маршрутизатором Crossroads для получения в результате анализа запроса соответствующих данных. Если запрос не касается динамического ресурса, прокси-сервер направляет его на статический файловый сервер, который использует несколько функций связующего модуля Connect, включая `logger`, `favicon` и `static`.

Листинг 6.12. Объединение модулей Connect, Crossroads и http-proxy для обработки запросов динамического и статического контента

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    crossroads = require('crossroads'),
  httpProxy = require('http-proxy'),
base = '/home/examples/public_html';

// создание прокси-сервера, слушающего все запросы
httpProxy.createServer(function(req,res,proxy) {

  if (req.url.match(/^\/node\//))
    proxy.proxyRequest(req, res, {
      host: 'localhost',
      port: 8000
    });
  else
    proxy.proxyRequest(req,res, {
      host: 'localhost',
      port: 8124
    });
}).listen(9000);

// добавление маршрута для запросов динамического ресурса
crossroads.addRoute('/node/{id}/', function(id) {
  console.log('accessed node ' + id);
});

// динамический файловый сервер
http.createServer(function(req,res) {
  crossroads.parse(req.url);
  res.end('that\'s all!');
}).listen(8000);

// статический файловый сервер
http.createServer(connect()
  .use(connect.favicon())
  .use(connect.logger('dev'))
  .use(connect.static(base))
).listen(8124);
```

Пусть сервер обрабатывает следующие запросы:

```
/node/345
/example1.html
/node/800
/html5media/chapter2/example14.html
```

Обработка этих URL-адресов приводит к следующим записям на консоли, а также к надлежащему ответу, возвращаемому браузеру:

```
accessed node 345
GET /example1.html 304 3ms
GET /phoenix5a.png 304 1ms
accessed node 800
GET /html5media/chapter2/example14.html 304 1ms
GET /html5media/chapter2/bigbuckposter.jpg 304 1ms
```

Я не говорю, что мы уже на полпути к созданию собственной системы управления контентом (Content Management System, CMS), но у нас уже есть инструментарий, который понадобится на случай ее создания. И потом, зачем создавать собственную систему, когда можно воспользоваться платформами, способными работать с Node? О них рассказывается в следующей главе.

7 Платформа Express

Программная платформа обеспечивает поддержку инфраструктуры, которая позволяет быстрее создавать веб-сайты и приложения. Это своеобразный каркас разработки, предоставляющий многие привычные и повсеместно используемые программные компоненты, в результате разработчик может сконцентрироваться на уникальной для приложения или сайта функциональности. Кроме того, она обеспечивает целостность вашего кода, что позволяет упростить управление этим кодом и его поддержку.

Термины *платформы* (frameworks) и *библиотеки* (libraries) весьма близки, поскольку те и другие предлагают функциональность многократного использования, доступную разработчикам в широком спектре приложений. И хотя отдельные возможности тех и других схожи, отличие состоит в том, что платформы обычно предоставляют также инфраструктуру, способную определить общую конструкцию вашего приложения.

Node.js предлагает весьма известные платформы, включая Connect (см. главу 6), хотя я считаю, что Connect больше относится к связующему программному обеспечению, чем к программной платформе. Благодаря поддержке, возможностям и популярности особое место занимают две Node-платформы — Express и Geddy. Если спросить специалистов о различиях между ними, можно услышать, что платформа Express больше похожа на Sinatra, а Geddy — на Rails. В терминах языка Ruby это означает, что в основе Geddy лежит концепция MVC (Model-View-Controller — модель-представление-контроллер), а Express больше относится к концепции RESTful (подробнее значение этого термина будет раскрыто позже).

На этом поле появился также и новый игрок, Flatiron, который ранее был представлен автономными компонентами, а теперь собран в единый комплексный продукт. Еще одной популярной программной платформой, представленной на связанном с Node инструментальном веб-сайте, является Ember.js, которая ранее была известна как SproutCore 2.0. Это дополнение к платформе CoreJS, которая также основана на концепции MVC.

Когда я размышлял о том, сколько места уделить каждой платформе в этой главе, то понял, что с трудом смогу рассмотреть лишь одну из них, и решил сконцентрировать свои усилия на наиболее популярной сейчас платформе Express.

Сайт Geddy.js находится по адресу http://geddyjs.org/. Flatiron можно найти по адресу http://flatironjs.org/, Github-страница Ember.js находится по адресу https://github.com/emberjs/ember.js, а основной сайт CoreJS — по адресу http://echo.nextapp.com/site/corejs. Github-страница Express находится по адресу https://github.com/visionmedia/express. Документацию по Express можно найти по адресу http://expressjs.com/.

Установка и запуск платформы Express

Express легко устанавливается с помощью диспетчера Node-пакетов:

```
npm install express
```

Чтобы разобраться в Express, лучше всего начать с создания приложения с помощью версии этой платформы для командной строки. Поскольку нет уверенности в том, что именно будет делать создаваемое приложение, лучше запустить его в пустом каталоге, а не в том, который содержит какую-либо важную информацию.

Я назвал свое новое приложение `site`, и сделать это было довольно просто:

```
express site
```

Приложение создает ряд каталогов:

```
  create : site
     create : site/package.json
     create : site/app.js
     create : site/public
     create : site/public/javascripts
     create : site/public/images
     create : site/routes
     create : site/routes/index.js
     create : site/public/stylesheets
     create : site/public/stylesheets/style.css
     create : site/views
     create : site/views/layout.jade
     create : site/views/index.jade
```

Оно также предоставляет полезное сообщение для замены текущего каталога каталогом приложения `site` и для запуска `npm install`:

```
npm install -d
```

После установки нового приложения запустите с помощью команды `node` сгенерированный файл app.js:

```
node app.js
```

Он запустит сервер, использующий порт 3000. При обращении к приложению выводится веб-страница со словами:

```
Express
Welcome to Express
```

Итак, вы создали свое первое Express-приложение. Теперь давайте посмотрим, что нам нужно сделать, чтобы это приложение стало интереснее.

О файле app.js

В листинге 7.1 показан исходный код файла app.js, который мы только что запускали.

Листинг 7.1. Исходный код файла app.js

```
/*
  * Module dependencies.
  */

var express = require('express')
   , routes = require('./routes')
   , http = require('http');

var app = express();

app.configure(function(){
   app.set('views', __dirname + '/views');
   app.set('view engine', 'jade');
   app.use(express.favicon());
   app.use(express.logger('dev'));
   app.use(express.static(__dirname + '/public'));
   app.use(express.bodyParser());
   app.use(express.methodOverride());
   app.use(app.router);
});

app.configure('development', function(){
   app.use(express.errorHandler());
});

app.get('/', routes.index);

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

В самом начале в приложение включаются три модуля: модуль Express, Node-модуль HTTP и только что сгенерированный модуль routes. В файле index.js, который находится в подкаталоге routes, содержится следующий код:

```
/*
  * GET home page.
  */
exports.index = function(req, res){
    res.render('index', { title: 'Express' });
};
```

При ближайшем рассмотрении этого кода видно, что метод `render`, относящийся к входящему в Express объекту ответа, выводит заданное представление с набором параметров — в данном случае это заголовок (`title`) «Express». Все это более подробно рассматривается в разделе «Маршрутизация».

А теперь вернемся к файлу app.js. После включения всех необходимых модулей мы создаем экземпляр объекта `Express`, а затем конфигурируем его с помощью набора параметров и метода `configure` (дополнительные сведения о методе `configure` даны в следующей врезке). Первый необязательный параметр конфигурирования является строкой, показывающей, в какой конкретной среде находится приложение, например в среде разработки (`development`) или в производственной среде (`production`). Если этот параметр опущен, приложение запускается в любой среде. Второй вызов метода `configure` в app.js специфичен только для среды разработки. Метод `configure` можно вызывать, если нужно, для каждой возможной среды, в этом случае выполняется тот метод, который соответствует среде.

НАСТРОЙКА РЕЖИМА РАБОТЫ ПРИЛОЖЕНИЯ

С помощью метода configure в Express-приложениях можно определить связующее программное обеспечение, настройки и параметры для каждого заданного режима. В следующем примере метод применяет заключенные в скобки настройки и параметры ко всем режимам:

```
app.config(function() { ... }
```

В следующем вызове метода configure заключенные в скобки настройки и параметры применяются только в среде разработки:

```
app.config('development', function() { ... }
```

Режим может быть любым из указанных и управляется он переменной окружения NODE_ENV:

```
$ export NODE_ENV=production
```

Или так:

```
$ export NODE_ENV=ourproduction
```

Можно использовать любой термин по своему усмотрению. По умолчанию используется термин development.

Чтобы обеспечить постоянный запуск своего приложения в указанном режиме, добавьте команду экспорта NODE_ENV в файл профиля пользователя.

Вторая функция, используемая для конфигурирования, является безымянной и включает в себя несколько ссылок на связующие программы. Некоторые из них нам уже знакомы (например, метод `use`) по нашей работе со связующим программным обеспечением модуля Connect (см. главу 6); в этом нет ничего неожиданного, поскольку основным автором этих двух приложений является один и тот же человек, Холовайчук (TJ Holowaychuk). А вот два вызова метода `app.set` нам незнакомы.

Метод `app.set` используется для определения различных параметров, например места, где находятся представления приложения:

```
app.set('views', __dirname + '/views');
```

Кроме того, он определяет движок для представлений (в данном случае — Jade):

```
app.set('view engine', 'jade');
```

Далее в app.js следует вызов Express, в который заключены связующие функции `favicon`, `logger` и `static` файлового сервера, что в дополнительном разъяснении не нуждается:

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.static(__dirname + '/public'));
```

Связующее программное обеспечение можно также вызвать в виде методов при создании сервера:

```
var app = express.createServer(
      express.logger(),
      express.bodyParts()
);
```

Какой из подходов выбрать, особой роли не играет.

Следующие три компонента связующего программного обеспечения (платформы), включаемые в создаваемое приложение:

```
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
```

Связующий компонент `bodyParser`, как и все другие связующие компоненты, берется непосредственно из Connect. Express занимается только его реэкспортом.

Функции `logger`, `favicon` и `static` рассматривались в предыдущей главе, а функция `bodyParse` нам еще не знакома. Эта связующая программа выполняет синтаксический разбор тела входящих запросов, преобразуя их в свойства объекта запроса. Функция `methodOverride`, также попавшая в Express из Connect, позволяет Express-приложениям эмулировать REST-возможности с помощью скрытого поля формы по имени `_method`.

Последним компонентом конфигурирования является `app.router`. Этот необязательный связующий компонент содержит все определенные маршруты и осущест-

вляет поиск любого заданного маршрута. Если он опущен, при первом же вызове `app.get`—`app.post` и т. д. в первую очередь будут проложены маршруты.

Полная поддержка REST (Representational State Transfer — передача репрезентативного состояния) означает поддержку HTTP-команд PUT и DELETE, как и команд GET и POST. Более подробно этот вопрос рассматривается в разделе «Маршрутизация и HTTP-команды».

Как и в случае модуля Connect, важную роль играет порядок вызова связующих программ. Связующая функция `favicon` вызывается перед функцией `logger`, потому что мы не хотим, чтобы обращения к favicon.ico засоряли журнал. Связующая функция `static` указывается перед функциями `bodyParser` и `methodOverride`, потому что ни одна из них при работе со статичными страницами не нужна — обработка форм в Express-приложении происходит в динамическом режиме, а не через статичную страницу.

Более подробно вопросы взаимодействия Express и Connect рассматриваются в разделе «Детали партнерства Express и Connect».

Второй вызов `configure`, характерный для режима разработки, добавляет к Express функцию `errorHandler`. Об этой и других технологиях обработки ошибок рассказывается в следующем разделе.

Обработка ошибок

Модуль Express предоставляет собственную технологию обработки ошибок, а также доступ к функции `errorHandler` модуля Connect в качестве инструмента обработки исключений. Этот инструмент позволяет лучше понять, что происходит в случае исключения. Функцию `errorHandler` можно включить в программу точно так же, как и другое связующее программное обеспечение:

```
app.use(express.errorHandler());
```

Используя флаг `dumpExceptions`, исключения можно направить в `stderr`:

```
app.use(express.errorHandler({dumpExceptions : true }));
```

Используя флаг `showStack`, для исключения можно также сгенерировать HTML-код:

```
app.use(express.errorHandler({showStack : true; dumpExceptions : true}));
```

Повторюсь: этот вариант обработки ошибок требуется только разработчикам, конечным пользователям нашего приложения исключения определенно видеть не нужно. Что же касается нас, то нам желательно обеспечить более эффективную

обработку ситуаций, когда страница не найдена или пользователь пытается обратиться к подкаталогу, доступ к которому ограничен.

Один из подходов предполагает добавление пользовательской безымянной функции в качестве последней программы в списке связующего программного обеспечения. Если ни одна из других связующих программ не может обработать запрос, он должен попасть прямо к этой последней функции:

```
app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.static(__dirname  +  '/public'));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
    app.use(function(req,  res,  next){
        res.send('Sorry ' + req.url + ' does not exist');
    });
});
```

В следующей главе мы настроим ответ, воспользовавшись шаблоном, генерирующим красивую страницу 404.

Можно обратиться и к другой форме обработки ошибок, когда вброшенные ошибки перехватываются и соответствующим образом обрабатываются. В документации по Express обработчик ошибок этого типа называется `app.error`, но, похоже, на момент написания этой книги его еще не было. Тем не менее сигнатура функции уже сложилась. В ней фигурируют четыре параметра: ошибка (`error`), запрос (`request`), ответ (`response`) и следующая функция (`next`).

Я добавил вторую связующую функцию обработки ошибок и настроил связующую функцию 404 на вбрасывание ошибки вместо ее непосредственной обработки:

```
app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.static(__dirname + '/public'));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
    app.use(function(req,  res,  next){
        throw new Error(req.url + ' not found');
    });
    app.use(function(err,  req, res, next) {
        console.log(err);
        res.send(err.message);
    });
});
```

Теперь я могу обработать ошибку 404, а также другие ошибки в рамках одной и той же функции. И опять-таки я могу воспользоваться шаблонами, чтобы сгенерировать более привлекательную страницу.

Детали партнерства Express и Connect

На протяжении всей этой главы мы наблюдали в действии активное партнерство Express и Connect. Именно благодаря модулю Connect платформа Express получает основную часть своей базовой функциональности.

Например, для поддержки сеансов можно воспользоваться связующими Connect-программами — функциями `cookieParser`, `cookieSession` и `session`. Нужно только не забыть задействовать Express-версию связующего программного обеспечения:

```
app.use(express.cookieParser('mumble'))
app.use(express.cookieSession({key : 'tracking'}))
```

С помощью связующей функции `staticCache` можно включить режим статического кэширования:

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.staticCache());
app.use(express.static(__dirname + '/public'));
```

По умолчанию кэш обслуживает максимум 128 объектов с максимальным объемом каждого в 256 Кбайт, что составляет в общем около 32 Мбайт. Эти параметры можно перенастроить с помощью свойств `maxObjects` и `maxLength`:

```
app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
```

Улучшить внешний вид списка файлов можно методом `directory`:

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
app.use(express.directory(__dirname + '/public'));
app.use(express.static(__dirname + '/public'));
```

Однако если `express.directory` используется с маршрутизацией, нужно обеспечить, чтобы связующая функция `directory` следовала за связующей функцией `app.router`, иначе она может конфликтовать с маршрутизацией.

Есть одно хорошее правило: помещать `express.directory` после всего остального связующего программного обеспечения, но перед любой обработкой ошибок.

Помимо прочего параметры метода `express.directory` позволяют задать режимы отображения скрытых файлов (по умолчанию — `false`) и значков (по умолчанию — `false`), а также режим фильтрации.

С Express можно также использовать связующий модуль Connect сторонних производителей. Однако при наличии маршрутизации нужно проявлять осмотрительность.

А теперь самое время вернуться к ключевому компоненту Express — маршрутизации.

Маршрутизация

Ключевым компонентом всех Node-платформ, как, впрочем, и многих современных платформ, является маршрутизация. Автономный модуль маршрутизации рассматривался в главе 6, там же было показано, как извлечь из URL-адреса служебный запрос.

В Express управление маршрутизацией осуществляется с помощью HTTP-команд `GET`, `PUT`, `DELETE` и `POST`. Методы называются соответствующим образом, например `app.get` для `GET` и `app.post` для `POST`. В созданном приложении, представленном в листинге 7.1, метод `app.get` служит для доступа к корневому каталогу приложения (`'/'`), а для обработки данных запроса передается слушатель запроса, в данном случае — функция `routes.index`.

Функция `routes.index` очень проста:

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

Она вызывает метод `render` для объекта ресурса. Этот метод получает имя файла, предоставляющего шаблон. Указывать расширение для этого файла нет необходимости, так приложение уже идентифицировало движок вывода представления:

```
app.set('view engine', 'jade');
```

Тем не менее можно также использовать следующий код:

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

Файл шаблона можно найти в еще одном сгенерированном каталоге по имени views. Там находятся два файла: index.jade и layout.jade. Файл index.jade является шаблоном, на который осуществляется ссылка в методе `render` и который имеет следующее содержимое:

```
extends layout

block content
    h1= title
    p Welcome to #{title}
```

Контентом документа является элемент `H1` с заголовком, а также элемент абзаца с приветствием, ссылающимся на значение заголовка. Шаблон layout.jade предоставляет общую разметку документа, включая `title` и `link` в элементе `head`, и тело документа в элементе `body`:

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

Файл layout.jade предоставляет контент для элемента body, определенного в файле index.jade.

Об использовании Jade-шаблонов и CSS-стилей с Express-приложениями рассказывается в главе 8.

Восстановим картину происходящего в этом приложении.

1. Основное Express-приложение использует метод `app.get` для назначения функции прослушивания запроса (`routes.index`) HTTP-запросу `GET`.
2. Функция `routes.index` вызывает функцию `res.render` для визуализации ответа на запрос `GET`.
3. Функция `res.render` вызывает функцию `render` объекта приложения.
4. Функция приложения `render` визуализирует указанное представление с параметрами, в данном случае — с заголовком `title`.
5. Затем визуализированный контент записывается в объект ответа, который отправляется по обратному маршруту на браузер пользователя.

На последнем этапе процесса сгенерированный контент записывается в ответ для отправки обратно браузеру. Более внимательный взгляд на исходный код позволяет увидеть, что метод `render` получает третий аргумент в виде функции обратного вызова, которая вызывается с любой ошибкой и сгенерированным текстом.

Желая лучше изучить сгенерированный контент, я внес изменения в файл `route.index`, добавив в него функцию, перехватив сгенерированный текст и выведя его на консоль. Поскольку я переписал исходные функции, я также отправил сгенерированный текст браузеру `res.write` точно так же, как мы это делали в других приложениях из предыдущих глав, а затем вызвал метод `res.end`, чтобы обозначить завершение:

```
exports.index = function(req, res){
    res.render('index', { title: 'Express' }, function(err, stuff) {
          if (!err) {
                console.log(stuff);
                res.write(stuff);
                res.end();
          }
    });
};
```

Как мы и надеялись, теперь приложение выводит контент на консоль, а также передает в браузер. Тем самым я просто демонстрирую, что даже при использовании неизвестной платформы в основе всего лежат Node и ранее рассмотренная нами функциональность. Разумеется, поскольку это все же платформа, мы знаем, что она должна предложить какой-нибудь более подходящий метод, чем методы `res.write` и `res.end`. Такой метод есть и о нем рассказывается в следующем разделе, в котором пути маршрутизации изучаются более детально.

Путь маршрутизации

Маршрут, или путь маршрутизации, заданный в листинге 7.1, представляет собой просто символ / (прямой слэш), обозначающий корневой адрес. Express внутренне преобразует все маршруты в объект регулярного выражения, поэтому вы можете использовать строки со специальными символами или просто напрямую указывать регулярные выражения в строках, описывающих пути.

Чтобы продемонстрировать эту возможность, я создал в листинге 7.2 простейшее приложение проверки путей маршрутизации, которое слушает три разных маршрута. Если запрос к серверу осуществляется по одному из этих маршрутов, параметры этого запроса возвращаются отправителю методом `send` Express-объекта ответа.

Листинг 7.2. Простое приложение для проверки паттернов разных путей маршрутизации

```
var express = require('express')
    , http = require('http');

var app = express();

app.configure(function(){
});

app.get(/^\/node?(?:\/(\d+)(?:\.\.(\d+))?)?/,  function(req,  res){
      console.log(req.params);
      res.send(req.params);
});

app.get('/content/*',function(req,res)  {
      res.send(req.params);
});

app.get("/products/:id/:operation?",  function(req,res)  {
      console.log(req);
      res.send(req.params.operation  +  '  '  +  req.params.id);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

Мы не выполняем маршрутизацию к представлениям и не обрабатываем статический контент, поэтому нам не нужно предоставлять в методе `app.configure` никакого связующего программного обеспечения. Однако нам нужно вызывать метод `app.configure`, если мы хотим справиться с обработкой запросов, не соответствующих ни одному из маршрутов. В приложении также используется среда разработки (`development`), которая предлагается по умолчанию.

При первом вызове метода `app.get` для указания пути используется регулярное выражение. Это регулярное выражение позаимствовано из руководства Express

Guide, с его помощью прослушивается любой запрос к узлу. Если в запросе также предоставляется уникальный идентификатор или диапазон идентификаторов, все это сохраняется в принадлежащем объекту массиве `params`, который отправляется назад в качестве ответа. Рассмотрим следующие запросы:

```
node
nodes
  /node/566
/node/1..10
/node/50..100/something
```

Эти запросы возвращают следующие значения массива `params`:

```
[null,  null]
[null,  null]
["566",  null]
["1",  "10"]
["50",  "100"]
```

С помощью регулярного выражения ведется поиск отдельного идентификатора или диапазона идентификаторов, заданного в виде двух значений и знака диапазона между ними (`..`). Все, что не относится к идентификатору или диапазону идентификаторов, игнорируется. Если идентификатор или диапазон идентификаторов не предоставлен, параметры имеют значение `null`.

Код, предназначенный для обработки запроса, не использует для отправки параметров обратно отправителю запроса методы `write` и `end` исходного HTTP-объекта ответа, вместо этого в нем используется Express-метод `send`. Этот метод устанавливает нужные заголовки для ответа (определяемые типом отправляемых данных), а затем передает контент, вызывая исходный HTTP-метод `end`.

Далее в `app.get` используется строка для определения паттерна маршрута. В данном случае мы ищем любой элемент контента. Данный паттерн будет соответствовать всему, что начинается с фрагмента `/content/`. Рассмотрим следующие запросы:

```
/content/156
/content/this_is_a_story
/content/apples/oranges
```

Эти запросы приведут к таким значениям массива `params`:

```
["156"]
["this_is_a_story"]
["apples/oranges"]
```

Символ звездочки (`*`) заставляет относиться ко всему либерально, поэтому возвращается все, что следует после фрагмента `content/`.

Последний вызов метода `app.get` ищет запрос продукта (`products`). Если задан идентификатор продукта, он может быть доступен напрямую через `params.id`. Если задана операция, к ней можно получить непосредственный доступ через `params.operation`. Разрешена любая комбинация двух значений, но при этом необходим как минимум один идентификатор или одна операция.

Рассмотрим следующие URL-адреса:

```
/products/laptopJK3444445/edit
/products/fordfocus/add
/products/add
/products/tablet89/delete
/products/
```

Эти адреса приводят к таким возвращаемым значениям:

```
edit laptopJK3444445
add fordfocus
undefined add
delete tablet89
Cannot GET /products/
```

Приложение выводит объект запроса на консоль. При запуске приложения я направляю вывод в файл output.txt, чтобы иметь возможность детально изучить объект запроса:

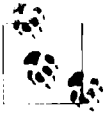
```
node  app.js > output.txt
```

Конечно, объект запроса является сокетом, и мы узнаем многое об этом объекте из нашей предыдущей работы, исследующей Node-объект HTTP-запроса. Нас главным образом интересует объект маршрута, добавляемый через Express. Следующие данные представляет собой вывод для объекта маршрута одного из запросов:

```
route:
  {  path: '/products/:id/:operation?',
     method: 'get',
     callbacks:  [ [Function] ],
     keys: [ [Object], [Object] ],
     regexp: /^\/products\/(?:([^\/]+?))(?:\/([^\/]+?))?\/?$/i,
     params: [ id: 'laptopJK3444445', operation: 'edit' ] },
```

Обратите внимание на сгенерированный объект регулярного выражения, который преобразует использованный мною дополнительный индикатор (:) в строке, описывающей путь, в нечто значимое для исходного JavaScript-движка (мне это как раз на руку, поскольку я не очень силен в регулярных выражениях).

Теперь, когда мы понимаем, как работают пути маршрутизации, давайте присмотримся к HTTP-командам.



Любой запрос, не соответствующий трем заданным паттернам маршрутов, просто генерирует ответ 404: Cannot GET /whatever (не могу получить запрошенное /whatever).

Маршрутизация и HTTP-команды

В предыдущих примерах для обработки входящих запросов мы использовали `app.get`. Этот метод, основанный на HTTP-методе `GET`, служит для получения данных,

но не для обработки поступающих данных, как и не для редактирования или удалении существующих данных. Для создания приложения, которое не только извлекает данные, но и работает с данными, нужно воспользоваться другими командами. Иными словами, нужно разработать приложение, работающее по технологии *RESTful*.

Как уже отмечалось, REST означает передачу репрезентативного состояния, а понятие RESTful служит для описания любого веб-приложения, в котором применяются принципы HTTP и REST, а именно: URL-адреса структурированы по образцу каталогов, состояние не сохраняется, данные преобразованы в один из типов медиаданных для Интернета (например, JSON), используются HTTP-методы (GET, POST, DELETE и PUT).

Пусть наше приложение управляет таким не самым завидным продуктом, как виджет. Чтобы создать новый виджет, нам нужно создать веб-страницу, предоставляющую форму, которая получает информацию о новом виджете. Мы можем сгенерировать эту форму с помощью приложения, и я продемонстрирую такой подход в главе 8, а пока воспользуемся статической веб-страницей, показанной в листинге 7.3.

Листинг 7.3. Образец HTML-формы для передачи Express-приложению данных виджета

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"  />
  <title>Widgets</title>
</head>
<body>
<form  method="POST" action="/widgets/add"
enctype="application/x-www-form-urlencoded">

  <p>Widget name: <input type="text" name="widgetname" id="widgetname"
size="25"  required/></p>

  <p>Widget Price: <input type="text"
pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
name="widgetprice" id="widgetprice" size="25" required/></p>

  <p>Widget Description: <br /><textarea name="widgetdesc" id="widgetdesc"
cols="20" rows="5">Описание  отсутствует</textarea>
  <p>

  <input type="submit" name="submit" id="submit" value="Submit"/>
  <input type="reset" name="reset" id="reset" value="Reset"/>
  </p>
  </form>
</body>
```

Для проверки допустимости данных на странице используются новые HTML5-атрибуты `required` и `pattern`. Разумеется, они работают только с браузерами, который поддерживают HTML5, но на данном этапе будем считать, что вы используете новый браузер, совместимый с HTML5.

В форме виджета требуется ввести имя виджета, цену (с ней в атрибуте `pattern` связано регулярное выражение для проверки структуры данных) и описание. Проверка приемлемости данных в браузере должна обеспечить получение нами трех значений и формат цены, соответствующий долларам США.

В Express-приложении мы собираемся сохранять виджеты в памяти, поскольку нам в данный момент нужно сосредоточиться на технологии Express. При отправке приложению каждого нового виджета он с помощью метода `app.post` добавляется в массив виджетов. Доступ к каждому виджету можно получить методом `app.get` по сгенерированному приложением идентификатору. Все приложение целиком представлено в листинге 7.4.

Листинг 7.4. Express-приложение для добавления и отображения виджетов

```
var express = require('express')
  , http = require('http')
  , app = express();

app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.static(__dirname + '/public'));
    app.use(express.bodyParser());
    app.use(app.router);
});

app.configure('development', function(){
    app.use(express.errorHandler());
});

// хранилище данных в памяти
var widgets = [
    { id : 1,
      name : 'My Special Widget',
      price : 100.00,
      descr : 'A widget beyond price'
    }
]

// добавление виджета
app.post('/widgets/add', function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
      { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice),
```

продолжение ⇩

Листинг 7.4 *(продолжение)*

```
            descr  :  req.body.widgetdesc  };
    console.log('added ' + widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});

// вывод виджета
app.get('/widgets/:id', function(req, res) {
      var indx = parseInt(req.params.id) - 1;
      if (!widgets[indx])
            res.send('There is no widget with id of ' + req.params.id);
      else
            res.send(widgets[indx]);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

Первый виджет уже введен в массив виджетов, поэтому у нас есть данные на случай, если потребуется сразу запросить виджет без его предварительного добавления. Обратите внимание на проверку условий в методе `app.get` для отправки ответа на запрос несуществующего или удаленного виджета.

Запуск приложения (example4.js в каталоге examples) и обращение к приложению с помощью символа / или /index.html (или /example3.html в каталоге examples) приводит к предоставлению статической HTML-страницы с формой. Отправка формы приводит к созданию страницы, выводящей сообщение о добавлении виджета, а также его идентификатора. Затем мы можем использовать идентификатор для вывода виджета — на самом деле, дампа экземпляра объекта виджета:

```
http://whateverdomain.com:3000/widgets/2
```

Запрос срабатывает, но... с приложением возникает проблема.

Во-первых, заполняя текстовые поля виджета, можно ошибиться. Хотя в поле цены не удастся ввести данные в невалютном формате, можно указать неверную цену. Можно также легко допустить опечатку в поле имени или в поле описания. Нам нужна возможность обновить виджет, а также возможность удалить виджет, когда он нам больше уже не нужен.

Приложению требуется обеспечить поддержку еще двух RESTful-команд: `PUT` и `DELETE`. Команда `PUT` служит для обновления виджета, а команда `DELETE` — для его удаления.

Чтобы обновить виджет, требуется форма с полями, заранее заполненными данными виджета и возможностью их редактирования. Чтобы удалить виджет, требуется форма, в которой подтверждается, что мы действительно хотим удалить виджет. В приложении эти формы генерируются в динамическом режиме с помощью шаблона, но сейчас, поскольку мы сосредоточены на HTTP-командах, я создал статические веб-страницы, обеспечивающие редактирование и последующее удаление созданного виджета 1.

Форма для обновления виджета 1 показана в следующем коде. Если не считать информацию для виджета 1, в ней всего лишь одно отличие от формы для добавления нового виджета: появилось скрытое поле с именем `_method` (оно выделено полужирным шрифтом):

```
<form  method="POST"  action="/widgets/1/update"
enctype="application/x-www-form-urlencoded">

  <p>Имя виджета: <input type="text" name="widgetname"
  id="widgetname" size="25" value="My Special Widget" required/></p>

  <p>Цена виджета: <input type="text"
  pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
  name="widgetprice" id="widgetprice" size="25" value="100.00" required/></p>

  <p>Widget Description: <br />
  <textarea name="widgetdesc" id="widgetdesc" cols="20"
  rows="5">A widget beyond price</textarea>
  <p>

  <input type="hidden" value="put" name="_method"  />

  <input type="submit" name="submit" id="submit" value="Submit"/>
  <input type="reset" name="reset" id="reset" value="Reset"/>
  </p>
  </form>
```

Поскольку команды PUT и DELETE не поддерживаются в атрибуте `method` формы, нам приходится добавлять их, используя скрытое поле с конкретным именем `_method` и присваивая ему значение либо `put` для команды PUT, либо `delete` для команды DELETE.

Форма удаления виджета имеет весьма простой вид: она содержит скрытое поле `_method` и кнопку, подтверждающую необходимость удаления виджета 1:

```
<p>Are you sure you want to delete Widget 1?</p>
<form method="POST" action="/widgets/1/delete"
  enctype="application/x-www-form-urlencoded">

  <input type="hidden" value="delete" name="_method" />

  <p>
  <input type="submit" name="submit" id="submit" value="Delete Widget 1"/>
  </p>
</form>
```

Чтобы гарантировать правильную обработку HTTP-команд, нужно в вызов метода `app.configure` следом за функцией `express.bodyParser` добавить еще одну связующую функцию, `express.methodOverride`, призванную преобразовать HTTP-метод в то, что указывается в скрытом поле в качестве значения:

```
app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.static(__dirname + '/public'));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
});
```

Далее нужно добавить функциональность обработки этих двух новых команд. Запрос на обновление заменяет содержимое объекта виджета новым контентом, а запрос на удаление удаляет запись в массиве, относящуюся к виджету, специально оставляя значение null, поскольку мы не хотим переупорядочивать массив из-за удаления виджета.

Чтобы придать нашему приложению завершенный вид, нужно также добавить страницу индекса для доступа к виджетам без какого-либо идентификатора или операции. На странице индекса просто выводится список всех виджетов, хранящихся на данный момент в памяти.

В листинге 7.5 представлено полноценное приложение для управления виджетами со всей новой функциональностью (выделена полужирным шрифтом).

Листинг 7.5. Модифицированное приложение для управления виджетами, поддерживающее редактирование и удаление виджетов, а также вывод списка всех виджетов

```
var express = require('express')
    , http = require('http')
    , app = express();

app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.static(__dirname + '/public'));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
});

app.configure('development', function(){
    app.use(express.errorHandler());
});

// хранилище данных в памяти
var widgets = [
    { id : 1,
        name : 'My Special Widget',
        price : 100.00,
        descr : 'A widget beyond price'
    }
]
```

```
// индекс /widgets/
app.get('/widgets', function(req, res) {
      res.send(widgets);
});

// вывод конкретного виджета
app.get('/widgets/:id', function(req, res) {
      var indx = parseInt(req.params.id) - 1;
      if (!widgets[indx])
            res.send('There is no widget with id of ' + req.params.id);
      else
            res.send(widgets[indx]);
});

//  добавление виджета
app.post('/widgets/add', function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length]  =
      { id : indx,
        name   :  req.body.widgetname,
        price  :  parseFloat(req.body.widgetprice),
        descr  :  req.body.widgetdesc  };
    console.log(widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});

// удаление виджета
app.del('/widgets/:id/delete', function(req,res) {
      var indx = req.params.id - 1;
      delete  widgets[indx];
      console.log('deleted ' + req.params.id);
      res.send('deleted ' + req.params.id);
});

// обновление/редактирование виджета
app.put('/widgets/:id/update', function(req,res) {
    var indx = parseInt(req.params.id) - 1;
    widgets[indx]  =
      {  id     :  indx,
         name   :  req.body.widgetname,
         price  :  parseFloat(req.body.widgetprice),
         descr  :  req.body.widgetdesc  };
    console.log(widgets[indx]);
    res.send ('Updated ' + req.params.id);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

После запуска приложения я добавил новый виджет, вывел список виджетов, обновил цену виджета 1 , удалил виджет, а затем снова вывел список всех виджетов. Все эти действия привели к следующим сообщениям на консоли, выведенным с помощью метода `console.log`:

```
Express server listening on port 3000
{  id:  2,
    name:  'This is my Baby',
    price:  4.55,
    descr:  'Baby widget'  }
POST  /widgets/add 200 4ms
GET  /widgets 200 2ms
GET  /edit.html 304 2ms
{  id:  0,
    name: 'My Special Widget',
    price: 200,
    descr: 'A widget beyond price' }
PUT  /widgets/1/update 200 2ms
GET  /del.html  304  2ms
deleted  1
DELETE  /widgets/1/delete 200 3ms
GET  /widgets 200 2ms
```

Обратите внимание на то, что HTTP-команды PUT и DELETE выделены полужирным шрифтом. При повторном выводе списка виджетов были возвращены следующие значения:

```
[
    null,
    {
        "id":  2,
        "name":  "This is my Baby",
        "price":  4.55,
        "descr":  "baby widget"
    }
]
```

Теперь у нас есть Express-приложение, поддерживающее соглашение RESTful. Тем не менее одна проблема еще осталась.

Если бы наше приложение управляло только одним объектом, возможно, нас бы вполне устраивало, что вся функциональность сосредоточена в единственном файле. Однако большинство приложений управляет более чем одним объектом, и функциональность всех этих приложений не столь проста, как в нашем небольшом примере. То есть нам нужно преобразовать наше Express-приложение, поддерживающее соглашение RESTful, в Express-приложение, поддерживающее соглашение RESTful и архитектуру MVC.

Курс на MVC

Реализация всей функциональности вашего приложения в единственном файле может быть приемлемой для очень простых приложений, но большинству приложений требуется лучшая организация. Уже упомянутая программная архитектура MVC (модель-представление-контроллер) является весьма популярной, и нам бы хотелось иметь ее преимущества в нашем Express-приложении. Требуемые для этого усилия не столь значительны, как это могло бы показаться, поскольку у нас уже есть нужная функциональность — Ruby on Rails.

Среда Ruby on Rails стала основой большей части того, что составляет природу Node, предоставив фундамент, который можно использовать, чтобы встроить поддержку MVC в наше Express-приложение. В Express уже встроена поддержка маршрутов (что является основой Rails), поэтому полдела уже сделано. Теперь нам нужно сделать следующий шаг — разделить модель, представление и контроллер. Для такого компонента, как контроллер, нам понадобится набор действий, определенных для каждого обслуживаемого объекта.

Rails поддерживает несколько разных действий, в которых маршрут (команда и путь) отображается на действие с данными. В основе отображения лежит понятие CRUD (create, read, update, delete — создание, чтение, обновление, удаление), описывающее четыре фундаментальные для надежного хранения функции. На веб-сайте Rails имеется руководство, где представлена замечательная таблица, показывающая те варианты отображения, которые необходимо реализовать в приложении. Я провел экстраполяцию этой таблицы, создав свою таблицу, в которой показаны варианты отображения для работы с виджетами (табл. 7.1).

Таблица 7.1. Варианты отображения для работы с виджетами

HTTP-команда	Путь	Действие	Назначение
GET	/widgets	index	Вывод виджетов
GET	/widgets/new	new	Возвращение HTML-формы для создания нового виджета
POST	/widgets	create	Создание нового виджета
GET	/widgets/:id	show	Вывод заданного виджета
GET	/widgets/:id/edit	edit	Возвращение HTML-формы для редактирования заданного виджета
PUT	/widgets/:id	update	Обновление заданного виджета
DELETE	/widgets/:id	destroy	Удаление заданного виджета

Большая часть функциональности у нас уже готова, нам осталось просто привести все в порядок.

Следует напомнить, что при внесении изменений, направленных на поддержку технологии MVC, у вас могут возникнуть проблемы с существующим связующим программным обеспечением. Например, связующая программа directory, обеспечивающая красивый вид содержимого каталога, конфликтует с действием create, поскольку они работают на одном и том же маршруте. Каким же видится решение этой проблемы? В вызове метода configure поместите связующую функцию express.directory после функции app router.

Сначала мы создадим подкаталог controllers и новый файл в нем по имени widgets.js. Затем мы скопируем все наши вызовы методов `apt.get` и `apt.put` в этот новый файл.

После этого нам нужно преобразовать вызовы методов в подходящий для MVC формат. Это означает, что нужно превратить вызов каждого метода маршрутизации в функцию и затем ее экспортировать. Например, рассмотрим функцию создания нового виджета:

```
// добавление виджета
app.post('/widgets/add', function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
      { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice)};
    console.log(widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});
```

Эта функция превращается в функцию `widgets.create`:

```
// добавление виджета
exports.create = function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
      { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice)},
    console.log(widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
};
```

Каждая функция по-прежнему получает объект запроса и объект ресурса. Единственное отличие состоит в том, что маршрут непосредственно на функцию больше не отображается.

В листинге 7.6 показан новый файл widgets.js, помещаемый в каталог controllers. Два из имеющих в нем методов, `new` и `edit`, пока оставим пустыми (оставим их заполнение до главы 8). Мы по-прежнему используем хранилище данных в памяти, кроме того, я упростил объект виджета, удалив поле описания, чтобы приложение было проще тестировать.

Листинг 7.6. Контроллер виджетов

```
var widgets = [
      { id : 1,
          name : "The Great Widget",
          price : 1000.00
      }
]

// индексированный список виджетов в каталоге /widgets/
exports.index = function(req, res) {
      res.send(widgets);
};

// вывод формы для нового виджета
exports.new = function(req, res) {
      res.send('displaying new widget form');
};

// добавление виджета
exports.create = function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
      { id : indx,
          name : req.body.widgetname,
          price : parseFloat(req.body.widgetprice) };
    console.log(widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
};

// вывод виджета
exports.show = function(req, res) {
      var indx = parseInt(req.params.id) - 1;
      if (!widgets[indx])
            res.send('There is no widget with id of ' + req.params.id);
      else
            res.send(widgets[indx]);
};

// удаление виджета
exports.destroy = function(req, res) {
      var indx = req.params.id - 1;
      delete widgets[indx];

      console.log('deleted ' + req.params.id);
      res.send('deleted ' + req.params.id);
};

// вывод формы редактирования
exports.edit = function(req, res) {
```

продолжение ⇩

Листинг 7.6 *(продолжение)*

```
    res.send('displaying edit form');
};

// обновление виджета
exports.update = function(req, res) {
    var indx = parseInt(req.params.id) - 1;
    widgets[indx] =
      { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice)}
    console.log(widgets[indx]);
    res.send ('Updated ' + req.params.id);
};
```

Обратите внимание, что методы `edit` и `new` относятся к методу `GET`, поскольку их единственная задача — служить формой. Есть соответствующие методы создания (`create`) и обновления (`update`), которые реально изменяют данные: первый из них служит в качестве метода `POST`, а второй — метода `PUT`.

Чтобы отобразить маршруты на новые функции, я создал второй модуль, `maproutecontroller`, с одной экспортируемой функцией `mapRoute`. У нее есть два параметра — это Express-объект `app`, а также `prefix`, представляющий отображенный объект контроллера (в данном случае — `widgets`). Функция `mapRoute` использует `prefix` для доступа к объекту контроллера `widgets`, а затем отображает методы, известные ему в этом объекте (поскольку объект является контроллером и имеет фиксированный набор требуемых методов), на соответствующий маршрут. Код этого нового модуля показан в листинге 7.7.

Листинг 7.7. Функция для отображения маршрутов на методы объекта контроллера

```
exports.mapRoute = function(app, prefix) {

    prefix = '/' + prefix;

    var prefixObj = require('./controllers/' + prefix);

    // индекс
    app.get(prefix, prefixObj.index);

    // добавление
    app.get(prefix + '/new', prefixObj.new);

    // вывод
    app.get(prefix + '/:id', prefixObj.show);

    // создание
    app.post(prefix + '/create', prefixObj.create);

    // редактирование
    app.get(prefix + '/:id/edit', prefixObj.edit);
```

```
    // обновление
    app.put(prefix + '/:id', prefixObj.update);

    // удаление
    app.del(prefix + '/:id', prefixObj.destroy);
};
```

Метод `mapRoute` является весьма простой функцией, он должен быть узнаваем, когда вы сравниваете маршруты с теми, которые представлены в табл. 7.1.

И наконец, мы завершаем работу над главным приложением, в котором все части собраны воедино. К счастью, теперь оно стало намного чище, поскольку у нас уже нет всех вызовов методов маршрутизации. Для того чтобы справиться с возможным разрастанием количества объектов, для хранения префиксного имени каждого из них я воспользовался массивом. При добавлении нового объекта в массив добавляется новый префикс.

Express поставляется с MVC-приложением, которое находится в каталоге examples. В нем используется процедура, которая обращается к каталогу controllers и выводит префиксные имена из найденных там имен файлов. При таком подходе нам не нужно изменять файл приложения для добавления нового объекта.

В листинге 7.8 показано готовое приложение. Я еще добавил в исходном виде функцию `routes.index` за исключением того, что изменил значение заголовка в файле routes/index.js с «Express» на «Widget Factory».

Листинг 7.8. Приложение, использующее для работы с виджетами новую инфраструктуру MVC

```
var express = require('express')
  , routes = require('./routes')
  , map = require('./maproutecontroller')
  , http = require('http')
  , app = express();

app.configure(function(){
    app.use(express.favicon());
    app.use(express.logger('dev'));
    app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
    app.use(express.static(__dirname + '/public'));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
    app.use(express.directory(__dirname + '/public'));
    app.use(function(req, res, next){
        throw new Error(req.url + ' not found');
    });
    app.use(function(err, req, res, next) {
        console.log(err);
        res.send(err.message);
    });
});
```

продолжение ↩

Листинг 7.8 *(продолжение)*

```
app.configure('development', function(){
    app.use(express.errorHandler());
});

app.get('/', routes.index);
var prefixes = ['widgets'];

// отображение маршрута на контроллер
prefixes.forEach(function(prefix) {
    map.mapRoute(app, prefix);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

Программа стала чище и проще, она стала расширяемой. У нас по-прежнему нет той части MVC, которая относится к *представлению*, но она появится в следующей главе.

Тестирование Express-приложения с помощью cURL

Вместо тестирования в браузере мы протестируем приложение с помощью cURL. Эта Unix-утилита особенно полезна при тестировании приложений, поддерживающих соглашение RESTful, без необходимости создания всех форм.

Для тестирования индексной страницы виджетов воспользуйтесь следующей cURL-командой (запуск приложения с моего учебного сайта и использование порта 3000, а вам следует настроить команду в соответствии с вашими параметрами):

```
curl  --request GET http://examples.burningbird.net:3000/widgets
```

Все, что следует за ключом `request`, определяет метод (в данном случае — `GET`), далее следует URL-адрес запроса. Вам должен вернуться дамп всех виджетов, находящихся к этому времени в хранилище данных.

Для тестирования нового виджета сначала нужно выдать запрос на новый объект:

```
curl  --request GET http://examples.burningbird.net:3000/widgets/new
```

Возвращаемое сообщение касается извлечения формы нового виджета. Затем утилита добавляет новый виджет, передавая данные для виджета в cURL-запросе и изменяя метод на `POST`:

```
curl  --request POST http://examples.burningbird.net:3000/widgets/create
  --data  'widgetname=Smallwidget&widgetprice=10.00'
```

Еще раз выполните тест индексной страницы, чтобы убедиться в выводе этого виджета на экран:

```
  curl  --request GET http://examples.burningbird.net:3000/widgets
```

В результате вы должны увидеть следующее:

```
[
    {
        "id": 1,
        "name": "The Great Widget",
        "price": 1000
    },
    {
        "id": 2,
        "name": "Smallwidget",
        "price": 10
    }
```

Затем обновите новый виджет, установив для него значение цены, равное 75.00. Теперь будет использоваться метод PUT:

```
curl --request PUT http://examples.burningbird.net:3000/widgets/2
--data 'widgetname=Smallwidget&widgetprice=75.00'
```

После проверки факта изменения данных продолжите тестирование и удалите новую запись, изменив HTTP-метод на DELETE:

```
curl --request DELETE http://examples.burningbird.net:3000/widgets/2
```

Теперь, когда у нас есть MVC-компонент контроллера, нам нужно добавить компоненты представлений, о которых рассказывается в главе 8. Но перед тем как перейти к этой главе, прочтите несколько заключительных советов.

ЗА ПРЕДЕЛАМИ EXPRESS

Хотя Express и является платформой, это очень простая платформа. Если нужно заняться чем-то вроде создания системы управления контентом, придется проделать большой объем работы.

Существуют приложения сторонних разработчиков, построенные на базе Express и предлагающие оба типа функциональности. Одно из них, Calipso, представляет собой полноценную систему управления контентом (Content Management System, CMS), построенную на базе Express и использующую в качестве надежного хранилища MongoDB.

Express-Resource является низкоуровневой платформой, которая предлагает для Express упрощенную MVC-функциональность, избавляя вас от необходимости создавать собственную.

Tower.js — еще одна веб-платформа, предлагающая высокоуровневую абстракцию и смоделированная по образцу Ruby on Rails с полной поддержкой MVC. RailwayJS — это тоже MVC-платформа, построенная на базе Express и смоделированная по образцу Ruby on Rails.

Следующей высокоуровневой платформой является Strata, но в ней используется другой подход, нежели в Tower.js и RailwayJS. Вместо модели Rails она следует установленным модулям WSGI (Python) и Rack (Ruby). Это низкоуровневая абстракция, с которой будет проще работать тому, кто не умеет программировать в Ruby on Rails.

---

8 Express, системы шаблонов и CSS

Такие платформы, как Express, предлагают массу полезной функциональности, но кое-чего они все же не обеспечивают — не предоставляют способа отделить данные от представления. Чтобы генерировать HTML-разметку для обработки результатов запросов или обновлений, можно использовать JavaScript-код, но это может быстро наскучить, особенно если придется генерировать разметку для каждой части страницы, включая боковые панели, заголовки и нижние колонтитулы. Естественно, для этого можно задействовать функции, но и эта работа может оказаться совершенно неподъемной.

К счастью для нас, параллельно с разработкой платформ шло также создание систем шаблонов, что в полной мере относится к Node и Express. В главе 7 для создания страницы индекса нам уже пришлось задействовать Jade — систему шаблонов, устанавливаемую по умолчанию вместе с Express. В Express поддерживаются также другие совместимые системы шаблонов, включая систему EJS (Embedded JavaScript — внедряемый JavaScript-код), пользующуюся особой популярностью. В Jade и EJS реализованы совершенно разные подходы, но оба они дают вполне приемлемые результаты.

Кроме того, хотя CSS-файлы для своего веб-сайта или приложения можно создавать вручную, удобнее использовать *CSS-движок*, который позволит упростить этот аспект вашей работы. Вместо добавления многочисленных фигурных скобок и точек с запятой вам достаточно задать упрощенную структуру, которую к тому же будет проще поддерживать. Одним из таких CSS-движков, неплохо зарекомендовавших себя при работе с Express и другими Node-приложениями, является Stylus.

В этой главе основное внимание уделяется системе шаблонов Jade, поскольку она по умолчанию устанавливается вместе с Express. Однако я собираюсь также кратко рассказать о EJS, чтобы вы могли познакомиться с двумя разными системами шаблонов и узнать, как они работают. Кроме того, я расскажу, как управлять CSS-

стилями с помощью движка Stylus, добиваясь привлекательного внешнего вида страниц.

Внедряемый JavaScript-код

Внедряемый JavaScript-код (Embedded JavaScript, EJS) — вполне подходящее название для этой системы шаблонов, поскольку оно наилучшим образом описывает суть ее работы: JavaScript-код внедряется в HTML-разметку, управляя слиянием данных и структуры документа. Это очень простая система шаблонов, построенная на базе системы ERB (Embedded Ruby — внедряемый Ruby-код).

GitHub-страница EJS находится по адресу https://github.com/visionmedia/ejs.

Базовый синтаксис

Если вам уже приходилось работать с системами управления контентом (Content Management Systems, CMS), вы быстро освоите EJS на базовом уровне. Вот как выглядит EJS-шаблон:

```
<% if (names.length) { %>
  <ul>
    <% names.forEach(function(name){ %>
      <li><%= name %></li>
    <% }) %>
  </ul>
<% } %>
```

В этом коде EJS-инструкции внедряются непосредственно в HTML-разметку, в данном случае предоставляя данные для отдельных позиций неупорядоченного списка. Пары, составленные из угловых скобок и знаков процента (<%, %>), задают границы EJS-инструкций: проверка условия позволяет убедиться, что получен массив, затем происходит обработка массива в JavaScript с выводом его отдельных значений.

В основе системы EJS лежит ERB, поэтому для описания ее формата мы часто будем использовать «erb-подобный» синтаксис.

Сами значения выводятся с помощью знака равенства (=), который можно считать сокращенным написанием команды «вывести это значение в этом месте»:

```
<%= name %>
```

При выводе значение экранируется. Чтобы вывести неэкранированное значение, используется символ дефиса (-):

```
<%- name %>
```

Если по каким-то причинам вам не хочется использовать стандартные открывающие и закрывающие EJS-теги (<%, %>), с помощью методов open и close EJS-объекта можно определить собственные теги:

```
ejs.open('<<');
ejs.close('>>');
```

Далее эти нестандартные теги можно применять вместо предлагаемых по умолчанию:

```
<h1><<=title >></h1>
```

Однако если у вас нет на то особо веских причин, я бы все же рекомендовал вам ограничиться тегами, предлагаемыми по умолчанию.

Хотя EJS-инструкции перемешиваются с HTML-разметкой, они остаются JavaScript-кодом, поэтому при использовании метода forEach, принадлежащего объекту массива, нужно указывать открывающую и закрывающую фигурные скобки, как, впрочем, и нужный формат.

В готовом продукте HTML-разметка визуализируется посредством вызова EJS-функции, которая либо возвращает JavaScript-функцию, генерирующую результат, либо сама генерирует окончательный результат. Как только мы установим версию EJS для Node, я вам все это покажу.

Использование EJS совместно с Node

Устанавливаемый модуль является специальной версией EJS, приспособленной для работы с Node. Это совсем не та версия, которую можно установить, перейдя на сайт EJS и непосредственно загрузив EJS. Версия EJS для Node может использоваться с JavaScript на стороне клиента, но я собираюсь сконцентрироваться на использовании EJS с Node-приложениями.

Установите систему шаблонов с помощью диспетчера Node-пакетов:

```
npm install ejs
```

После того как система EJS установлена, с ней можно непосредственно работать в Node-приложении, причем такие платформы, как Express, для этого не требуются. Для демонстрации давайте визуализируем HTML-разметку из заданного файла шаблона:

```
<html>
<head>
<title><%= title %></title>
</head>
<body>
<% if (names.length) { %>
  <ul>
    <% names.forEach(function(name){ %>
```

```
      <li><%= name %></li>
    <% }) %>
  </ul>
<% } %>
</body>
```

Вызовите метод `renderFile` EJS-объекта напрямую. Это приведет к открытию шаблона и использованию данных, предоставленных в виде параметра для генерации HTML-разметки.

В листинге 8.1 используется стандартный HTTP-сервер, поставляемый вместе с Node для прослушивания запросов через порт 8124. При получении запроса приложение вызывает EJS-метод `renderFile`, передавая ему путь к файлу шаблона, а также массив имен `names` и заголовок документа `title`. Последний параметр представляет собой функцию обратного вызова, которая возвращает либо ошибку (во вполне читабельном виде), либо сгенерированную итоговую HTML-разметку. В этом примере при отсутствии ошибок результат возвращается в объекте ответа. В случае же ошибки в качестве результата отправляется сообщение об ошибке, и объект ошибки выводится на консоль.

Листинг 8.1. Генерирование HTML-разметки из данных и EJS-шаблона

```
var   http = require('http')
    , ejs = require('ejs')
;
// создание http-сервера
http.createServer(function (req, res) {
  res.writeHead(200, {'content-type': 'text/html'});
  // визуализируемые данные
  var names = ['Joe', 'Mary', 'Sue', 'Mark'];
  var title = 'Testing EJS';
  // вывод данных или ошибки
  ejs.renderFile(__dirname + '/views/test.ejs',
                 {title : 'testing', names : names},
                   function(err, result) {
      if (!err) {
         res.end(result);
      } else {
         res.end('An error occurred accessing page');
         console.log(err);
      }
  });
}).listen(8124);
console.log('Server running on 8124/');
```

Одним из методов визуализации является метод `render`, который в качестве строки получает EJS-шаблон, а затем возвращает отформатированную HTML-разметку:

```
var str = fs.readFileSync(__dirname + '/views/test.ejs', 'utf8');

var html = ejs.render(str, {names : names, title: title });
res.end(html);
```

Третий метод визуализации, который я пока не хотел бы демонстрировать, называется `compile`. Он получает строку EJS-шаблона и возвращает JavaScript-функцию, которую можно в любой момент вызывать для визуализации HTML-разметки. Этот метод можно также использовать, чтобы настроить EJS для Node в клиентских приложениях.

Использование метода compile демонстрируется в разделе «Создание таблицы высших достижений в игре» главы 9.

Использование фильтров EJS для Node

Помимо поддержки визуализирующих EJS-шаблонов, EJS для Node предлагает набор предопределенных фильтров, которые могут еще больше упростить генерацию HTML-разметки. Один из фильтров, `first`, извлекает первое значение из предоставленного массива значений. Другой фильтр, `downcase`, получает результат работы фильтра `first` и переводит текстовые символы в нижний регистр:

```
var names = ['Joe Brown', 'Mary Smith', 'Tom Thumb', 'Cinder Ella'];
var str = '<p><%=: users | first | downcase %></p>';
var html = ejs.render(str, {users : names });
```

Получается следующий результат:

```
<p>joe brown</p>
```

Фильтры могут выстраиваться в цепочку, когда результат работы одного фильтра передается по каналу другому фильтру. Использование фильтра инициируется символом двоеточия (:), следующего за символом равенства (=), за которыми идет объект данных. В следующем примере применения фильтров берется набор объектов `people` (люди), который отображается на новый объект, содержащий только имена людей, затем эти имена сортируются, после чего выводится объединенная строка имен:

```
var people = [
   {name : 'Joe Brown', age : 32},
   {name : 'Mary Smith', age : 54},
   {name : 'Tom Thumb', age : 21},
   {name : 'Cinder Ella', age : 16}];
var str = "<p><%=: people | map:'name' | sort | join %></p>";
var html = ejs.render(str, {people : people });
```

Результат применения этой комбинации фильтров выглядит следующим образом:

```
Cinder Ella, Joe Brown, Mary Smith, Tom Thumb
```

В версии EJS для Node фильтры не документированы, и используя их вперемешку, нужно быть осмотрительным, поскольку некоторые фильтры требуют строку, а не массив объектов. В табл. 8.1 представлен список фильтров с кратким описанием их назначения и типов данных, с которыми они работают.

Таблица 8.1. Фильтры, доступные в версии EJS для Node

Фильтр	Тип данных	Назначение
first	Принимает и возвращает массив	Возвращает первый элемент массива
last	Принимает и возвращает массив	Возвращает последний элемент массива
capitalize	Принимает и возвращает строку	Переводит первый символ строки в верхний регистр
downcase	Принимает и возвращает строку	Переводит все символы строки в нижний регистр
upcase	Принимает и возвращает строку	Переводит все символы строки в верхний регистр
sort	Принимает и возвращает массив	Применяет к массиву метод Array.sort
sort_by:'свойство'	Принимает массив и имя свойства, возвращает массив	Создает нестандартную функцию сортировки массива объектов по свойству
size	Принимает массив, возвращает число	Возвращает результат вызова метода Array.length
plus:n	Принимает два числа или две строки, возвращает число	Возвращает a + b
minus:n	Принимает два числа или две строки, возвращает число	Возвращает b – a
times:n	Принимает два числа или две строки, возвращает число	Возвращает a * b
divided_by:n	Принимает два числа или две строки, возвращает число	Возвращает a / b
join:'значение'	Принимает массив, возвращает строку	Применяет метод Array.join с заданным значением или с символом запятой (,), предлагаемым по умолчанию
truncate:n	Принимает строку и длину, возвращает строку	Применяет метод String.substr
truncate_words:n	Принимает строку и длину слов, возвращает строку	Применяет метод String.split, а затем метод String.splice

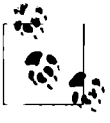
продолжение

Таблица 8.1 *(продолжение)*

Фильтр	Тип данных	Назначение
replace:шаблон, подстановка	Принимает строку, шаблон и подстановку, возвращает строку	Применяет метод String.replace
prepend:значение	Принимает строку и строковое значение, возвращает строку	Добавляет значение в начало строки
append:значение	Принимает строку и строковое значение, возвращает строку	Добавляет значение к строке
map:'свойство'	Принимает массив и имя свойства, возвращает массив	Создает новый массив, состоящий из свойств заданного объекта, используя для этого метод Array.map
reverse	Принимает массив или строку	Если передан массив, применяет к нему метод Array.reverse; если передана строка, разбивает ее на слова, меняет порядок слов на обратный и заново объединяет их в строку
get	Принимает объект и имя свойства	Возвращает свойство заданного объекта
json	Принимает объект	Преобразует объект в JSON-строку

Использование EJS совместно с Express

Система шаблонов предоставляет недостающую часть, необходимую для заполнения *представления* в прикладной архитектуре MVC (Model-View-Controller – модель-представление-контроллер), о которой рассказывалось в главе 7.



О добавлении той части архитектуры MVC, которая относится к модели, рассказывается в главе 10.

С системой шаблонов мы познакомились в листинге 7.1 (см. главу 7). В том примере применялась система шаблонов Jade, но мы можем легко переделать этот пример для использования EJS. Но насколько легко? Листинг 8.2 является точной копией листинга 7.1, только в нем вместо Jade задействована система шаблонов EJS. Изменение коснулось всего лишь одной строки, которая выделена полужирным шрифтом.

Листинг 8.2. Использование EJS в качестве системы шаблонов для приложения

```
var express = require('express')
  , routes = require('./routes')
  , http = require('http');
var app = express();
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});
app.configure('development', function(){
  app.use(express.errorHandler());
});
app.get('/', routes.index);
http.createServer(app).listen(3000);
console.log("Express server listening on port 3000");
```

Маршрут к index.js вообще не требует изменений, поскольку в нем нет ничего, что было бы специфическим для той или иной системы шаблонов; в нем используется метод `render` Express-объекта ресурса, который не зависит от системы шаблонов (конечно, при условии, что она совместима с Express):

```
exports.index = function(req, res){
  res.render('index', { title: 'Express' }, function(err, stuff) {
     if (!err) {
        console.log(stuff);
        res.write(stuff);
        res.end();
     }
  });
};
```

В каталоге views находится файл index.ejs (обратите внимание на расширение), использующий аннотацию версии EJS для Node, а не аннотацию системы Jade, с которой мы познакомились в главе 7:

```
<html>
<head>
<title><%= title %></title>
</head>
<body>
<h1><%= title %></title>
<p>Welcome to <%= title %></p>
</body>
```

Этот пример демонстрирует всю прелесть работы с приложением, в котором модель, контроллер и представление не зависят друг от друга. Вы можете поменять технологию ввода и вывода информации, например выбрать другую систему шаблонов, и это не окажет никакого влияния ни на прикладную логику, ни на доступ к данным.

Напомню, что происходит с этим приложением.

1. Главное Express-приложение использует метод `app.get`, чтобы назначить функцию прослушивания запросов (`routes.index`) HTTP-запросу `GET`.
2. Функция `routes.index` вызывает метод `res.render` для визуализации ответа в запросе `GET`.
3. Функция `res.render` вызывает функцию `render` объекта приложения.
4. Функция `render` объекта приложения визуализирует заданное представление с соответствующими параметрами, в данном случае — с заголовком `title`.
5. Визуализированный контент записывается в объект ответа и возвращается обратно в браузер пользователя.

В главе 7 основное внимание уделялось не представлению, а разным аспектам маршрутизации приложения. Теперь мы возьмем приложение, созданное в листингах с 7.6 по 7.8, и добавим к нему функциональность представлений. Однако сначала нам нужно провести небольшую реструктуризацию среды, чтобы обеспечить приложению возможность роста.

Реструктуризация среды для нескольких объектов

Хотя приложение — это фабрика виджетов (widget factory), объектами, которые мы собираемся поддерживать в системе, должны быть не только виджеты. Нам нужно реструктурировать среду, чтобы к ней можно было добавлять любые объекты, которые нам потребуются.

Сейчас среда имеет следующий вид:

```
/application directory
   /routes - home directory controller
   /controllers - object controllers
   /public - static files
   /views - template files
```

Каталоги routes и controllers можно не трогать, а вот каталоги views и public нужно изменить, чтобы в них можно было размещать различные объекты. Чтобы не размещать все представления виджетов непосредственно в каталоге views, мы добавим для них новый подкаталог представлений с соответствующим именем widgets:

```
/application directory
   / views
      /widgets
```

Чтобы не размещать все статические файлы виджетов непосредственно в каталоге public, мы также создадим для них подкаталог widgets:

```
/application directory
   /public
      /widgets
```

Теперь мы можем добавлять новые объекты в новые каталоги и для каждого сможем использовать имена файлов new.html и edit.ejs, не волнуясь насчет опасности перезаписать существующие файлы.

Обратите внимание, что в этой структуре предполагается наличие у нас статических файлов для нашего приложения. На следующем этапе нужно решить, как интегрировать статические файлы в новую динамическую среду.

Маршруты к статическим файлам

Первым требующим переделки компонентом приложения является код добавления нового виджета. Он состоит из двух частей: вывод формы для получения информации о новом виджете и сохранения нового виджета в существующем хранилище данных виджетов.

Для формы можно создать EJS-шаблон, хотя в нем не будет никаких динамических компонентов или, по крайней мере, в соответствии с предназначением страницы их не будет на данный момент. Однако нет никакого смысла обрабатывать через систему шаблонов то, что не нуждается в ее возможностях.

Мы также могли бы просто изменить способ обращения к форме — вместо /widgets/new обращаться к ней посредством /widgets/new.html. Но это приводит к непоследовательности в маршрутизации приложения. Кроме того, если позже к странице формы будут добавлены динамические компоненты, нам придется менять ссылки на новую форму.

Лучше всего было бы обрабатывать маршрутизацию запросов и обслуживать статическую страницу так, как будто она является динамической, но при этом не пользоваться системой шаблонов.

В Express-объекте ресурса есть метод `redirect`, которым можно воспользоваться для перенаправления запроса к файлу new.html, но по завершении перенаправления в адресной строке браузера будет отображаться файл new.html. Кроме того, при этом возвращается код состояния 302, который нам не нужен. Вместо этого мы воспользуемся имеющимся в объекте ресурса методом `sendfile`. Этот метод принимает в качестве параметров путь к файлу, возможные варианты настройки и в качестве необязательного параметра функцию обратного вызова с единственным параметром `error`. Контроллер виджетов использует только первый параметр (путь к файлу):

```
__dirname + "/../public/widgets/widget.html"
```

Мы воспользовались относительным индикатором `..`, поскольку каталог public находится за пределами родительского каталога, относящегося к каталогу controllers. Однако мы не можем использовать этот путь в методе `sendfile`, поскольку он станет причиной ошибки запрещения доступа 403. Вместо него мы воспользуемся методом `normalize` модуля `path` для преобразования используемых в пути относительных индикаторов в эквивалентный абсолютный путь:

```
// вывод формы нового виджета
exports.new = function(req, res) {
    var filePath = require('path').normalize(__dirname +
                                                "/../public/widgets/new.html");
    res.sendfile(filePath);
};
```

Переживать о HTML-странице с формой не стоит, это простая форма, показанная в листинге 8.3. Но мы вернули в нее поле `description`, чтобы сделать данные немного интереснее.

Листинг 8.3. HTML-разметка формы нового виджета

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Widgets</title>
</head>
<body>
<h1>Add Widget:</h1>

<form method="POST" action="/widgets/create"
enctype="application/x-www-form-urlencoded">

  <p>Widget name: <input type="text" name="widgetname"
  id="widgetname" size="25" required/></p>

  <p>Widget Price: <input type="text"
  pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
  name="widgetprice" id="widgetprice" size="25" required/></p>

  <p>Widget Description: <br />
  <textarea name="widgetdesc" id="widgetdesc" cols="20"
  rows="5"></textarea>
  <p>

  <input type="submit" name="submit" id="submit" value="Submit"/>
  <input type="reset" name="reset" id="reset" value="Reset"/>
  </p>
  </form>
</body>
```

Разумеется, форма отправляется методом `POST`.

Теперь, когда приложение может вывести форму нового виджета, нам нужно изменить контроллер виджетов, чтобы обработать передачу формы.

Существует также модуль расширения для Express по имени express-rewrite, который предоставляет возможность перезаписи URL-адресов. Его можно установить с помощью диспетчера Node-пакетов:

```
npm install express-rewrite
```

Обработка нового объекта передачи

Перед добавлением кода поддержки нового шаблона нужно внести изменения в главный файл приложения, встроив туда механизм использования системы EJS. Я не буду целиком повторять файл app.js из листинга 7.8 (см. главу 7), поскольку изменения касаются только вызова метода конфигурирования для подключения движка EJS-шаблонов и каталога views:

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.directory(__dirname + '/public'));
  app.use(function(req, res, next){
    throw new Error(req.url + ' not found');
  });
  app.use(function(err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});
```

Теперь мы готовы преобразовать контроллер виджетов для использования шаблонов, начиная с кода добавления нового виджета.

Фактическая обработка данных в контроллере виджетов для новых виджетов не изменяется. Мы по-прежнему извлекаем данные из тела запроса и добавляем их в хранилище виджетов в памяти. Однако теперь, когда у нас есть доступ к системе шаблонов, мы попробуем изменить характер реакции на успешное добавление нового виджета.

Я создал новый EJS-шаблон с именем added.ejs, показанный в листинге 8.4. Все что он делает — это предоставляет список свойств виджета и сообщение, состоящее из заголовка, отправленного с объектом виджета.

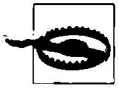
Листинг 8.4. Шаблон представления для подтверждения факта добавления виджета

```
<head>
<title><%= title %></title>
</head>
<body>
<h1><%= title %> | <%= widget.name %></h1>
<ul>
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: <%= widget.price.toFixed(2) %></li>
<li>Desc: <%= widget.desc %></li>
</ul>
</body>
```

Этот код мало отличается от кода обработки обновления, показанного в главе 7, за исключением того факта, что теперь мы выводим представление, а не возвращаем пользователю сообщение (измененный фрагмент выделен полужирным шрифтом)

```
// добавление виджета
exports.create = function(req, res) {
  // создание id виджета
  var indx = widgets.length + 1;
  // добавление виджета
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc : req.body.widgetdesc };
  // вывод на консоль и подтверждение добавления, отправляемое пользователю
  console.log(widgets[indx-1]);
  res.render('widgets/added', {title: 'Widget Added', widget :
                               widgets[indx-1]});
};
```

Представлению отправляются два параметра: заголовок страницы и объект виджета. Простой по виду, но вполне информативный результат показан на рис. 8.1.



Код обработки нового виджета не проверяет ни правильность вводимых данных, ни права доступа, ни попытки взлома с помощью SQL-инъекций. Вопросы проверки данных, безопасности и авторизации рассматриваются в главе 15.

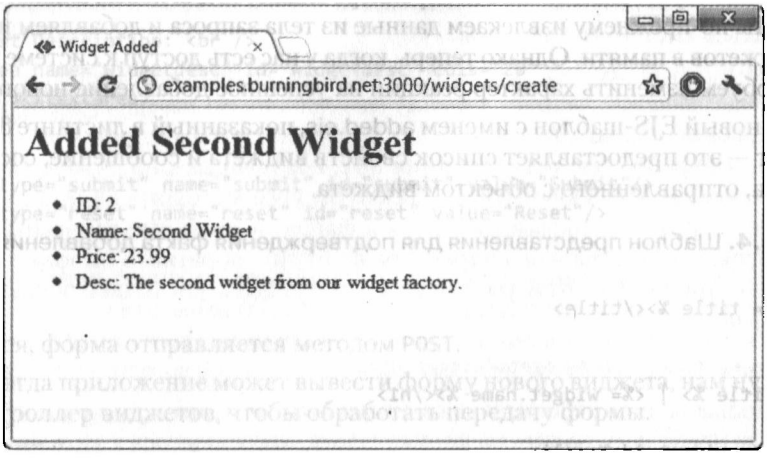


Рис. 8.1. Подтверждение добавления виджета

Следующие два процесса преобразования в шаблоны — обновление (`update`) и удаление (`deletion`) — требуют указывать виджет, в отношении которого должно быть

выполнено действие. Кроме того, нам нужно преобразовать индекс вывода для всех виджетов. Для всех трех процессов мы собираемся использовать представление, позволяющее построить как страницу индекса для виджетов, так и список выбора виджетов, чем сейчас и займемся.

Работа с индексами виджетов и создание списка выбора

Список выбора (picklist) — это список, в котором пользователь может сделать выбор. Для приложения, работающего с виджетами, список выбора мог бы быть обычным или раскрывающимся, встроенным в страницу обновления или добавления, использующим технологию Ajax и сценарий на стороне клиента, но мы собираемся встроить эту функциональность в индексную страницу приложения.

На данный момент на индексной странице приложения для виджетов имеется только дамп данных из хранилища данных виджетов. Он достаточно информативен, но непритязателен по виду и практически бесполезен. Чтобы его улучшить, мы собираемся добавить представление для построчного вывода всех имеющихся виджетов в таблице, столбцами которой послужат их свойства. Мы также собираемся добавить еще два столбца: один со ссылкой для редактирования виджета, а другой со ссылкой для его удаления. Тем самым мы закроем имеющийся в приложении пробел: предоставим способ редактирования или удаления виджета без необходимости запоминать или знать его идентификатор.

В листинге 8.5 показан контент шаблона для нашего нового представления, файл которого называется index.ejs. Поскольку файл находится в подкаталоге widgets, нам не стоит беспокоиться, что такое же имя имеет файл index.ejs на более высоком уровне иерархии.

Листинг 8.5. Страница индекса приложения для виджетов со ссылками на редактирование и удаление отдельных виджетов

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title><%= title %></title>
</head>
<body>
<% if (widgets.length) { %>
  <table>
  <caption>Widgets</caption>
  <tr><th>ID</th><th>Name</th><th>Price</th><th>Description</th></tr>
    <% widgets.forEach(function(widget){ %>
       <tr>
       <td><%= widget.id %></td>
       <td><%= widget.name %></td>
       <td>$<%= widget.price.toFixed(2) %></td>
       <td><%= widget.desc %></td>
       <td><a href="/widgets/<%= widget.id %>/edit">Edit</a></td>
       <td><a href="/widgets/<%= widget.id %>">Delete</a></td>
```

продолжение ⇩

Листинг 8.5 *(продолжение)*

```
        </tr>
      <% }) %>
    </table>
<% } %>
</body>
```

Код контроллера для запуска этого представления очень простой — это вызов метода `render` для визуализации представления с отправкой через него в качестве данных всего массива виджетов:

```
// индексированный список виджетов в каталоге /widgets/
exports.index = function(req, res) {
  res.render('widgets/index', {title : 'Widgets', widgets : widgets});
};
```

Если в листинге 8.5 у объекта есть свойство `length` (если это массив), перебираются его объекты элементов и их свойства выводятся в качестве табличных данных в дополнение к ссылкам на редактирование и удаление объекта. На рис. 8.2 показана таблица после добавления к нашему хранилищу в памяти нескольких виджетов.

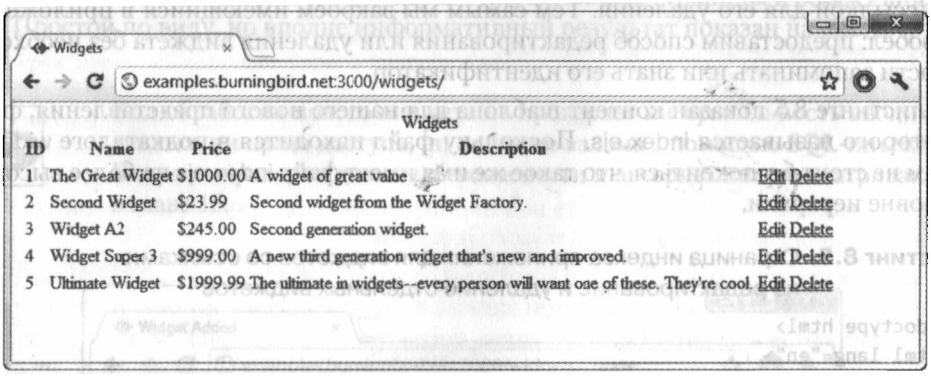


Рис. 8.2. Таблица после добавления нескольких виджетов

Ссылка (маршрут) на удаление фактически такая же, как и ссылка (маршрут) на показ виджета: /widgets/:id. Мы добавим на страницу показа виджета скрытую по большей части форму с кнопкой для удаления виджета, если он больше не нужен. Это позволит встроить в приложение необходимый для удаления триггер без необходимости добавлять новый маршрут. Кроме того, так мы обеспечим новый уровень защиты, дающий пользователю возможность убедиться, что он удаляет именно тот виджет, который хочет.

Вместо встраивания запроса на удаление в страницу показа виджета можно просто создать еще один маршрут, например /widgets/:id/delete, и из ссылки на странице индекса сгенерировать страницу «Вы уверены?», из которой затем инициировать удаление.

Показ отдельного объекта и подтверждение удаления объекта

Показать отдельный виджет так же просто, как и предоставить место для всех его свойств, внедренное в ту HTML-разметку, которую вы хотите использовать. В нашем приложении для свойств виджета я использовал неупорядоченный список (unordered list, `ul`).

Поскольку мы также размещаем на странице триггер для удаления объекта, в нижней части я добавил форму с кнопкой удаления виджета. В этой форме находится скрытое поле `_method`, генерирующее HTTP-команду `DELETE`, которая приводит к вызову метода удаления `destroy`, используемого в приложении. Весь шаблон показан в листинге 8.6.

Листинг 8.6. Представление для вывода свойств виджета и формы для его удаления

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title><%= title %></title>
</head>
<body>
<h1><%= widget.name %></h1>
<ul>
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: $<%= widget.price.toFixed(2) %></li>
<li>Description: <%= widget.desc %></li>
</ul>
<form method="POST" action="/widgets/<%= widget.id %>"
enctype="application/x-www-form-urlencoded">
  <input type="hidden" value="delete" name="_method" />
  <input type="submit" name="submit" id="submit" value="Delete Widget"/>
</form>
</body>
```

Код контроллера следует немного изменить либо для метода `show`, либо для метода `destroy`. На данный момент я остановился на методе `destroy`. Он лишь удаляет объект из хранилища в памяти и отправляет обратно сообщение об этом:

```
exports.destroy = function(req, res) {
  var indx = req.params.id - 1;
  delete widgets[indx];

  console.log('deleted ' + req.params.id);
  res.send('deleted ' + req.params.id);
};
```

Метод `show` требует небольших изменений — нужно просто заменить передачу сообщения `send` вызовом метода `render` для визуализации нового представления:

```
// показ виджета
exports.show = function(req, res) {
  var indx = parseInt(req.params.id) - 1;
  if (!widgets[indx])
    res.send('There is no widget with id of ' + req.params.id);
  else
    res.render('widgets/show', {title : 'Show Widget', widget :
                                widgets[indx]});
};
```

На рис. 8.3 показано, как выглядит страница показа виджета с кнопкой удаления виджета в нижней части.

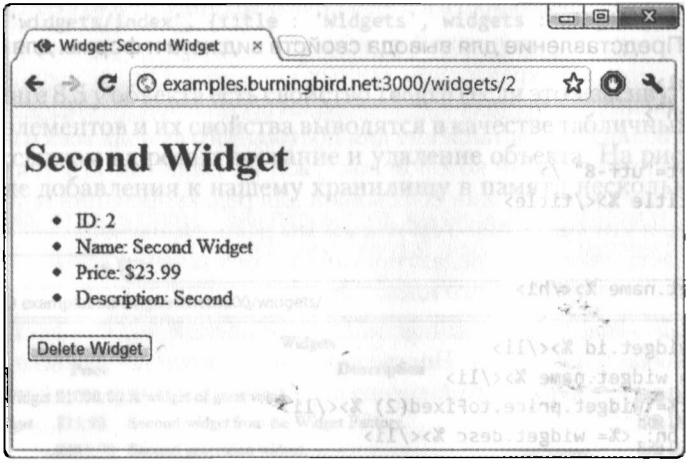


Рис. 8.3. Страница показа виджета с кнопкой удаления

Теперь вы уже поняли, как просто включать в приложение представления. Самое лучшее в этой системе то, что вы можете вносить изменения в шаблоны представлений, не останавливая работу приложения: оно просто использует измененный шаблон при следующем обращении к представлению.

Осталось добавить одно представление для обновления виджета, и мы завершим преобразование нашего приложения для использования системы EJS.

Предоставление формы обновления и обработка запроса PUT

Форма для редактирования виджета точно такая же, как и форма для добавления виджета, за исключением добавления еще одного поля `_method`. Кроме того, форма заранее заполняется данными для редактируемого виджета, поэтому нам нужно встроить теги шаблонов и соответствующие значения.

В листинге 8.7 представлен контент файла шаблона edit.ejs. Обратите внимание на теги шаблона со значениями полей в элементах ввода. Также обратите внимание на добавление поля `_method`.

Листинг 8.7. Файл шаблона для представления с заранее введенными данными

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title><%= title %></title>
</head>
<body>
<h1>Edit <%= widget.name %></h1>
<form method="POST" action="/widgets/<%= widget.id %>"
enctype="application/x-www-form-urlencoded">

  <p>Widget name: <input type="text" name="widgetname"
  id="widgetname" size="25" value="<%=widget.name %>" required /></p>

  <p>Widget Price: <input type="text"
  pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
  name="widgetprice" id="widgetprice" size="25" value="<%= widget.price %>"
                                                        required/></p>
  <p>Widget Description: <br />
  <textarea name="widgetdesc" id="widgetdesc" cols="20"
  rows="5"><%= widget.desc %></textarea>
  <p>
  <input type="hidden" value="put" name="_method" />
  <input type="submit" name="submit" id="submit" value="Submit"/>
  <input type="reset" name="reset" id="reset" value="Reset"/>
  <p>

  <input type="hidden" value="put" name="_method" />

  <input type="submit" name="submit" id="submit" value="Отправить"/>
  <input type="reset" name="reset" id="reset" value="Сбросить"/>
  </p>
  </form>
</body>
```

На рис. 8.4 показана страница с загруженным виджетом. Вам остается только отредактировать значения полей, а затем щелкнуть на кнопке отправки для принятия изменений.

В контроллер нужно внести такие же, как и раньше, незначительные изменения. Доступ к представлению редактирования осуществляется методом `res.render`, а объект виджета передается в виде данных:

```
// вывод формы редактирования
exports.edit = function(req, res) {
  var indx = parseInt(req.params.id) - 1;
  res.render('widgets/edit', {title : 'Edit Widget',
                              widget : widgets[indx]});
};
```

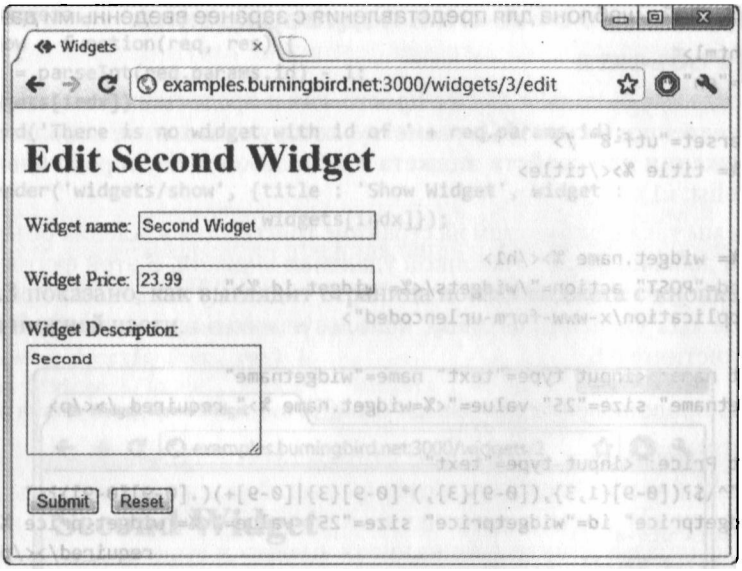


Рис. 8.4. Представление для редактирования виджета

Код для обработки обновления очень похож на код, использовавшийся в главе 7, за исключением того, что вместо отправки сообщения об обновлении объекта используется представление. Однако мы не создаем новое представление. Вместо этого мы задействуем представление widgets/added.ejs, которое использовалось ранее. Поскольку оба представления всего лишь выводят свойства объекта и могут получать заголовок, передаваемый в виде данных, мы можем легко изменить назначение представления простой заменой заголовка:

```
// обновление виджета
exports.update = function(req, res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id : indx + 1,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc : req.body.widgetdesc}
  console.log(widgets[indx]);
  res.render('widgets/added', {title: 'Widget Edited',
                               widget : widgets[indx]})
};
```

Здесь опять используемое представление не влияет на выводимый маршрут (URL-адрес), поэтому повторному использованию представления ничто не препятствует. По мере усложнения приложения возможность многократного использования представления может помочь сэкономить массу сил.

Во всех последних разделах, связанных с преобразованием контроллера для использование шаблонов, мы могли видеть фрагменты кода контроллера. В листинге 8.8

представлена полная копия измененного файла, которую можно сравнить с листингом 7.6 (см. главу 7), чтобы понять, как просто встроить представление в код и сколько сил благодаря этому можно сэкономить.

Листинг 8.8. Контроллер приложения для виджетов с реализованными представлениями

```
var widgets = [
   { id : 1,
     name : "The Great Widget",
     price : 1000.00,
     desc: "A widget of great value"
   }
]
// индексный список виджетов в каталоге /widgets/
exports.index = function(req, res) {
   res.render('widgets/index', {title : 'Widgets', widgets : widgets});
};
// вывод формы для нового виджета
exports.new = function(req, res) {
   var filePath = require('path').normalize(__dirname + "/../public/widgets/new.html");
    res.sendfile(filePath);
};
// добавление виджета
exports.create = function(req, res) {
  // генерирование id виджета
  var indx = widgets.length + 1;
  // добавление виджета
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc : req.body.widgetdesc };
  // вывод на консоль и подтверждение добавления в адрес пользователя
  console.log(widgets[indx-1]);
  res.render('widgets/added', {title: 'Widget Added',
                               widget : widgets[indx-1]});
};
// показ виджета
exports.show = function(req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
      res.send('There is no widget with id of ' + req.params.id);
   else
      res.render('widgets/show', {title : 'Show Widget',
                                  widget : widgets[indx]});
};
// удаление виджета
exports.destroy = function(req, res) {
    var indx = req.params.id - 1;
```

продолжение ⇩

Листинг 8.8 *(продолжение)*

```
    delete widgets[indx];

    console.log('deleted ' + req.params.id);
    res.send('deleted ' + req.params.id);
};
// вывод формы редактирования
exports.edit = function(req, res) {
    var indx = parseInt(req.params.id) - 1;
    res.render('widgets/edit', {title : 'Edit Widget',
                                widget : widgets[indx]});
};
// обновление виджета
exports.update = function(req, res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id : indx + 1,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc : req.body.widgetdesc}
  console.log(widgets[indx]);
  res.render('widgets/added', {title: 'Widget Edited',
                               widget : widgets[indx]})
};
```

Хотя код контроллера в данной главе показан в последний раз, мы собираемся внести в приложение глубокие изменения — заменить систему шаблонов.

Система шаблонов Jade

Система шаблонов Jade устанавливается вместе с платформой Express по умолчанию. Она сильно отличается от EJS: вместо вставки тегов шаблона непосредственно в HTML-разметку используется упрощенная версия HTML-разметки.

Веб-сайт Jade находится по адресу http://jade-lang.com/.

Краткий курс Jade-синтаксиса

В системе шаблонов Jade HTML-элементы задаются по именам, но угловые скобки при этом не используются, а вложенность обозначается отступами. Например:

```
<html>
<head>
<title>This is the title</title>
```

```
</head>
<body>
<p>Say hi World</p>
</body>
</html>
```

Вместо этой разметки мы получаем следующее:

```
html
   head
      title This is the title
   body
      p Say hi World
```

Контент как элемента заголовка, так и элемента абзаца просто указывается после имени элемента. Закрывающие теги отсутствуют, они подразумеваются, а отступы инициируют вложенность. Далее показан еще один пример, в котором также используются как имя класса и идентификатор, так и дополнительная вложенность:

```
      html
   head
      title This is the title
   body
      div.content
         div#title
            p nested data
```

Этот код генерирует следующую разметку:

```
<html>
<head>
<title>Это заголовок</title>
</head>
<body>
<div class="content">
<div id="title">
<p>nested data</p>
</div>
</div>
</body>
</html>
```

При большом объеме контента, например текста абзаца, для объединения текста можно использовать вертикальную черту (|):

```
p
  | some text
  | more text
  | and even more
```

Это все превращается в следующий код:

```
<p>some text more text and even more </p>
```

Еще один подход предусматривает завершение элемента абзаца точкой (.), показывающей, что блок содержит только текст, и позволяющей нам опустить вертикальную черту:

```
p.
  some text
  more text
  and even more
```

Если нужно, можно включить HTML-разметку в виде текста, тогда в сгенерированном исходном коде она не будет трактоваться как HTML-разметка:

```
body.
  <h1>A header</h1>
  <p>A paragraph</p>
```

Обычно элементы формы содержат атрибуты; они вставляются в Jade-шаблон в круглых скобках, включая установку их значений (если таковые имеются). Атрибуты требуют только разделения пробелами, но я каждый из них указываю на отдельной строке, чтобы шаблон лучше читался.

Показанный далее код является Jade-шаблоном:

```
html
   head
     title This is it
   body
      form(method="POST"
           action="/widgets"
           enctype="application/x-www-form-urlencoded")
         input(type="text"
               name="widgetname"
               id="widgetname"
               size="25")
         input(type="text"
               name="widgetprice"
               id="widgetprice"
               size="25")
         input(type="submit"
               name="submit"
               id="submit"
               value="Submit")
```

Это шаблон генерирует следующую HTML-разметку:

```
<html>
<head>
<title>This is it</title>
</head>
<body>
<form method="POST" action="/widgets"
```

```
 enctype="application/x-www-form-urlencoded">
<input type="text" name="widgetname" id="widgetname" size="25"/>
<input type="text" name="widgetprice" id="widgetprice" size="25"/>
<input type="submit" name="submit" id="submit" value="Submit"/>
</form>
</body>
</html>
```

Использование инструкций block и extends для сборки шаблонов представлений из блоков

Итак, мы собираемся преобразовать наше приложение для виджетов, чтобы в нем вместо EJS использовалась система Jade. Все изменения, которые нужно внести в код приложения, касаются только файла app.js, где заменяется движок системы шаблонов:

```
app.set('view engine', 'jade');
```

Больше никаких изменений в приложение вносить не нужно. Вообще никаких. На этом все. Точка.

Все шаблоны используют один и тот же макет страницы. Этот макет довольно прост; в нем нет ни боковых панелей, ни нижних колонтитулов, в нем никак не используются CSS-стили. Благодаря простой природе страниц в предыдущих примерах мы не возражали против дублирования одного и того же макета в каждом представлении. Однако если заняться добавлением дополнительных компонентов к общей структуре страниц, таких как боковые панели, заголовок и нижний колонтитул, необходимость обрабатывать одну и ту же информацию в каждом файле макета может превратиться в обузу.

Поэтому первое, что мы собираемся сделать в ходе наших преобразований, это создать шаблон макета, который будет использоваться всеми шаблонами.

Обработка представления в Express 3.x полностью изменилась, включая способ реализации частичных представлений и использование макетов. Jade-шаблоны, описываемые в этом разделе, можно использовать с Express 2.x, если добавить следующий вызов метода конфигурации:

```
app.set('view options', {layout: false});
```

В листинге 8.9 содержится полноценный файл layout.jade. В нем используется инструкция `doctype`, подключающая HTML5, добавляется элемент `head` с элементами `title` и `meta`, добавляется элемент `body`, а затем следует ссылка на инструкцию `block` с именем `content`. Здесь показано, как включать блоки контента, определенные в других файлах шаблонов.

Листинг 8.9. Простой шаблон макета, созданный в Jade

```
doctype 5
html(lang="en")
  head
    title #{title}
    meta(charset="utf-8")
  body
    block content
```

Обратите внимание на использование для заголовка символа решетки и фигурных скобок (`#{}`). Именно так в Jade вставляются данные, переданные шаблону. Идентификатор по сравнению с EJS не изменился, изменился только синтаксис.

Чтобы воспользоваться новым шаблоном макета, мы начинаем каждый из шаблонов контента следующей строкой:

```
extends layout
```

Использование инструкции `extends` позволяет движку шаблонов узнать, где именно искать шаблон макета для представления страницы, а использование инструкции `block` сообщает движку шаблона, куда помещать сгенерированный контент.

Для имени блока указывать слово `content` не обязательно, кроме того, можно задать более одного блока. В дополнение к этому вы можете включать другие файлы шаблонов, если хотите продолжать дробить шаблон макета на части. Я изменил файл layout.jade, чтобы включить заголовок, а не размечать его непосредственно в файле макета:

```
doctype 5
html(lang="en")
  include header
  body
    block content
```

Затем я определил контент заголовка в файле header.jade, поместив туда следующий код:

```
head
  title #{title}
  meta(charset="utf-8")
```

В коде новых файлов layout.jade и header.jade нужно отметить два обстоятельства.

Во-первых, директива `include` имеет относительный характер. Давайте разобьем представление на следующую структуру подкаталогов:

```
/views
   /widgets
       layout.jade
      /standard
         header.jade
```

В этом случае для включения шаблона заголовка в файл макета придется использовать следующий код:

```
include standard/header
```

Файл не обязательно должен относиться к Jade, каждый из файлов может относиться к HTML-разметке, тогда нужно будет указывать расширение файла:

```
include standard/header.html
```

Во-вторых, отступы в файле header.jade не нужны. Отступы будут взяты из родительского файла и их не нужно дублировать во включаемом файле шаблона. Если это сделать, при обработке шаблона возникнет ошибка.

После определения шаблона макета настало время преобразовать EJS-представления в Jade-представления.

Также настало время подумать о замене статического файла формы добавления виджета динамическим, что позволило бы воспользоваться преимуществами нового шаблона макета.

Преобразование представлений виджет-приложения в Jade-шаблоны

Первым представлением, в котором мы перейдем с EJS на Jade, будет шаблон added.ejs, обеспечивающий отклик на успешное добавление нового виджета. Файл шаблона мы назовем added.jade (чтобы работать с существующим кодом контроллера имя должно остаться прежним, хотя расширение будет другим), и в нем, как показано в листинге 8.10, будет использоваться только что определенный файл layout.jade.

Листинг 8.10. Страница подтверждения о добавлении виджета, преобразованная для системы шаблонов Jade

```
extends layout
block content
  h1 #{title} | #{widget.name}
  ul
    li id: #{widget.id}
    li Name: #{widget.name}
    li Price: $#{widget.price.toFixed()}
    li Desc: #{widget.desc}
```

Обратите внимание, что мы по-прежнему можем использовать метод `toFixed` для форматирования поля цены.

Блок называется `content`, следовательно, он отвечает ожиданиям, касающимся имени блока в файле layout.jade. Упрощенная HTML-разметка для заголовка `h1` и неупорядоченного списка объединена с данными, передаваемыми контроллером, в данном случае — с объектом виджета.

Запуск приложения для виджетов и добавление нового виджета приведут к созданию такой же HTML-разметки, которая генерировалась системой EJS: она будет состоять из заголовка и списка свойств виджета для только что добавленного экземпляра, и все это без какого бы то ни было изменения кода контроллера.

Преобразование главного представления виджетов

Следующий преобразуемый шаблон — индексный шаблон, который выводит список всех виджетов в таблице с дополнительными столбцами, предназначенными для редактирования или удаления виджета. В этот шаблон мы собираемся внести небольшие изменения. Мы отделим генерирование строки таблицы для каждого виджета от генерирования всей таблицы.

Сначала мы создадим шаблон row.jade. Предполагается, что данные представлены объектом по имени `widget` с доступными в этом объекте свойствами:

```
tr
  td #{widget.id}
  td #{widget.name}
  td $#{widget.price.toFixed(2)}
  td #{widget.desc}
  td
    a(href='/widgets/#{widget.id}/edit') Edit
  td
    a(href='/widgets/#{widget.id}') Delete
```

Каждая ссылка *должна* находиться на отдельной строке, в противном случае будет потеряна информация о вложенности, которая формируется с помощью отступов.

Главный файл index.jade, который ссылается на только что созданный строковый шаблон, показан в листинге 8.11. В этом шаблоне представлены две новые Jade-конструкции: проверка условия и итерация. Условие используется для проверки свойства длины `length` объекта `widgets`, которая убеждает нас, что мы имеем дело с массивом. В конструкции итерации используется сокращенная форма метода `Array.forEach`, где происходит перебор элементов массива и каждый экземпляр присваивается новой переменной `widget`.

Листинг 8.11. Индексный шаблон для создания таблицы виджетов

```
extends layout

block content
  table
    caption Widgets
      if widgets.length
        tr
          th ID
          th Name
          th Price
          th Description
          th
          th
        each widget in widgets
          include row
```

Здесь в целом тратится значительно меньше усилий, чем при ручном вводе всех этих угловых скобок, особенно с заголовками таблицы (`th`). Результаты,

получаемые при использовании Jade-шаблона, идентичны тем, что были получены для EJS-шаблона: HTML-разметка таблицы с виджетами в каждой строке, а также возможность изменить или удалить каждый виджет.

Преобразование форм редактирования и удаления

Следующие две переделки относятся к формам.

Сначала мы преобразуем шаблон редактирования для системы Jade. Единственной по-настоящему непростой частью преобразования является обработка всевозможных атрибутов. Хотя их можно разделить пробелами, я полагаю, что полезнее разместить каждый из них на отдельной строке. Тогда вы сможете видеть, что все атрибуты включены правильно, и легко сможете проверить их значения. В листинге 8.12 представлен довольно длинный шаблон для формы редактирования виджета.

Листинг 8.12. Jade-шаблон для формы редактирования виджета

```
extends layout
block content
  h1 Edit #{widget.name}
  form(method="POST"
       action="/widgets/#{widget.id}"
       enctype="application/x-www-form-urlencoded")
    p Widget Name:
       input(type="text"
             name="widgetname"
             id="widgetname"
             size="25"
             value="#{widget.name}"
             required)
    p Widget Price:
       input(type="text"
             name="widgetprice"
             id="widgetprice"
             size="25"
             value="#{widget.price}"
             pattern="="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9]
                                                    [0-9])?$" required)
    p Widget Description:
       br
       textarea(name="widgetdesc"
                id="widgetdesc"
                cols="20"
                rows="5") #{widget.desc}
    p
       input(type="hidden"
             name="_method"
             id="_method"
             value="put")
```

продолжение

Листинг 8.12 *(продолжение)*

```
        input(type="submit"
              name="submit"
              id="submit"
              value="Submit")
        input(type="reset"
              name="reset"
              id="reset"
              value="reset")
```

Когда мы переделывали страницу показа виджетов, я отмечал, что верхняя часть страницы в основном повторяет то, что выводится в шаблоне added.jade листинга 8.10 — это неупорядоченный список со всеми свойствами виджета. Есть еще одна возможность упростить конструкцию!

Я создал новый шаблон widget.jade, который просто выводит информацию о виджете в виде неупорядоченного списка:

```
ul
  li id: #{widget.id}
  li Name: #{widget.name}
  li Price: $#{widget.price.toFixed(2)}
  li Desc: #{widget.desc}
```

Затем я изменил файл added.jade из листинга 8.10, чтобы воспользоваться этим новым шаблоном:

```
extends layout
block content
  h1 #{title} | #{widget.name}
  include widget
```

Как видно в листинге 8.13, в новом шаблоне показа виджетов используется новый шаблон widget.jade.

Листинг 8.13. Новый Jade-шаблон показа виджетов

```
extends layout
block content
  h1 #{widget.name}
  include widget
  form(method="POST"
       action="/widgets/#{widget.id}"
       enctype="application/x-www-form-urlencoded")
       input(type="hidden"
             name="_method"
             id="_method"
             value="delete")
       input(type="submit"
             name="submit"
             id="submit"
             value="Delete Widget")
```

Как видите, разбиение шаблонов на блоки делает каждый шаблон намного понятнее и проще в поддержке.

С новым модульным шаблоном мы можем теперь показать и удалить конкретный виджет... и это приводит нас к довольно неожиданному нюансу, который отличает Jade-шаблон от EJS-шаблона.

При удалении виджетов в приложении они удаляются *на своем месте*. Это означает, что элемент массива, по сути, получает значение `null`, чтобы положение виджета в массиве относительно его идентификатора оставалось прежним. Такое «сохранение на месте» в EJS не вызывает никаких проблем, когда мы добавляем или удаляем виджеты и показываем их на странице индекса, но при использовании Jade возникает проблема: мы получаем ошибку отсутствия свойств, поскольку в Jade не поддерживается фильтрация элементов массива со значением `null`, выполняемая в EJS.

Эту ошибку довольно легко исправить. Как показано в листинге 8.11, нужно просто добавить к Jade-разметке еще одну проверку условий в файле index.jade, чтобы гарантировать наличие объекта виджета (что его элемент не имеет значение `null`):

```
extends layout
block content
  table
    caption Widgets
      if widgets.length
        tr
          th ID
          th Name
          th Price
          th Description
          th
          th
        each widget in widgets
          if widget
            include row
```

Теперь, когда все шаблоны представлений переделаны для использования в системе Jade, работа над приложением завершена. (По крайней мере, до тех пор, пока в главе 10 мы не добавим порцию данных.)

Тем не менее, хотя приложение уже готово, оно не отличается особой привлекательностью. Разумеется, не так уж сложно добавить в заголовок таблицу стилей, изменив вид всех элементов, но предварительно мы кратко рассмотрим другой подход — использование модуля Stylus.

Подключение модуля Stylus к приложению для упрощения CSS-стилей

Включить таблицу стилей в любой файл шаблонов совсем не трудно. Мы может включить такую таблицу в файл Jade-шаблона header.jade следующим образом:

```
head
  title #{title}
  meta(charset="utf-8")
  link(type="text/css"
       rel="stylesheet"
       href="/stylesheets/main.css"
       media="all")
```

Данная таблица стилей будет доступна всем представлениям нашего приложения, поскольку все они используют шаблон макета, в котором находится этот заголовок.

Теперь можно вполне определенно заметить ценность преобразования статического файла new.html в шаблон представления: внесение изменения в заголовок не оказывает на этот файл никакого влияния, его нужно редактировать вручную.

Однако, если вам полюбился синтаксис Jade, можете задействовать вариант этого синтаксиса для CSS, подключив к вашему приложению модуль Stylus.

Для использования Stylus сначала с помощью диспетчера Node-пакетов нужно установить соответствующий модуль:

```
npm install stylus
```

Stylus — это совсем не то же самое, что Jade. Этот движок не создает динамические CSS-представления, а генерирует статические таблицы стилей из Stylus-шаблона при первом обращении к этому шаблону или при каждом изменении шаблона.

Дополнительные сведения о Stylus можно найти по адресу `http://learnboost.github.com/stylus/docs/js.html`.

Для встраивания в приложение для виджетов функциональности движка Stylus нужно включить соответствующий модуль в раздел `require` главного файла приложения (app.js). Кроме того, нужно включить связующее программное обеспечение Stylus наряду с другими в вызов метода `configure`, передав ему параметр с источником Stylus-шаблонов и приемником, куда нужно будет помещать скомпилированные таблицы стилей. Измененный файл app.js показан в листинге 8.14 (изменения выделены полужирным шрифтом).

Листинг 8.14. Добавление поддержки Stylus-шаблонов к приложению для виджетов

```
var express = require('express')
  , routes = require('./routes')
  , map = require('./maproutecontroller')
  , http = require('http')
```

```
  , stylus = require('stylus')
  , app = express();
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
  app.use(stylus.middleware({
      src: __dirname + '/views'
    , dest: __dirname + '/public'
  }));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.directory(__dirname + '/public'));
  app.use(function(req, res, next){
    throw new Error(req.url + ' not found');
  });
  app.use(function(err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});
app.configure('development', function(){
  app.use(express.errorHandler());
});
app.get('/', routes.index);
var prefixes = ['widgets'];
// отображение маршрута на контроллер
prefixes.forEach(function(prefix) {
  map.mapRoute(app, prefix);
});
http.createServer(app).listen(3000);
console.log("Express server listening on port 3000");
```

При первом обращении к приложению после внесения этого изменения вы можете заметить небольшую задержку. Причина в том, что модуль Stylus генерирует таблицу стилей, что происходит при каждом перезапуске, связанном с добавлением или изменением шаблона таблицы стилей. Но после создания таблицы стилей модуль создает копию, которая и используется для работы, не подвергаясь перекомпиляции при каждом обращении к странице.

Поле внесения изменений в шаблон таблиц стилей вам понадобится перезапустить Express-приложение.

Все Stylus-шаблоны таблиц стилей имеют расширение .styl. Исходным каталогом является views, но ожидается, что шаблоны таблиц стилей будут находиться в подкаталоге stylesheets каталога views. При создании статических таблиц стилей они помещаются в подкаталог stylesheets, который находится в целевом каталоге (в данном случае — /public).

После Jade синтаксис модуля Stylus покажется вам очень знакомым. Здесь тоже каждый стилизуемый элемент выводится в списке за установленными в таблице стилей отступами. Синтаксис исключает необходимость в фигурных скобках, двоеточиях и точках с запятой.

Например, чтобы изменить цвет фона веб-страницы на желтый, а цвет текста — на красный, используется следующий Stylus-шаблон:

```
body
   background-color yellow
   color red
```

Если нужно, чтобы элементы имели одинаковые параметры, перечислите их на одной строке через запятую, точно так же, как это делается в CSS:

```
p, tr
   background-color yellow
   color red
```

Или же можно перечислить элементы на отдельных строках:

```
p
tr
   background-color yellow
   color red
```

Если требуется псевдокласс, например `:hover` или `:visited`, воспользуйтесь следующим синтаксисом:

```
textarea
input
  background-color #fff
  &:hover
    background-color cyan
```

Амперсанд (`&`) представляет родительский селектор. Все вместе это можно показать в таком Stylus-шаблоне:

```
p, tr
   background-color yellow
   color red
textarea
input
  background-color #fff
  &:hover
    background-color cyan
```

Этот код приводит к созданию следующего статического CSS-файла:

```
p,
tr {
  background-color: #ff0;
  color: #f00;
}
textarea,
input {
  background-color: #fff;
}
textarea:hover,
input:hover {
  background-color: #0ff;
}
```

О Stylus можно писать еще довольно долго, но я предлагаю вам заняться этим самостоятельно. На веб-сайте Stylus предоставлен хороший набор документов по синтаксису. Однако перед тем, как перейти к следующей главе, мы с помощью Stylus создадим таблицу стилей, призванную улучшить внешний вид нашего приложения для виджетов.

Точнее говоря, мы добавим рамку и отступы для элементов HTML-таблицы в индексной таблице списка виджетов. Мы также собираемся изменить шрифт заголовков и убрать маркеры неупорядоченного списка. Это небольшие изменения, но они станут началом в совершенствовании внешнего вида приложения для виджетов.

Новый шаблон таблицы стилей показан в листинге 8.15. Он не отличается большим размером и в нем не используются CSS-стили. У него весьма простое содержимое, но оно все же улучшает внешний вид приложения.

Листинг 8.15. Stylus-шаблон приложения для виджетов

```
body
  margin 50px
table
  width 90%
  border-collapse collapse
table, td, th, caption
  border 1px solid black
td
  padding 20px
caption
  font-size larger
  background-color yellow
  padding 10px
h1
  font 1.5em Georgia, serif
ul
  list-style-type none
form
  margin 20px
  padding 20px
```

На рис. 8.5 показана индексная страница после добавления нескольких виджетов. В ней нет ничего особенного, но с новой таблицей стилей данные читаются намного лучше.

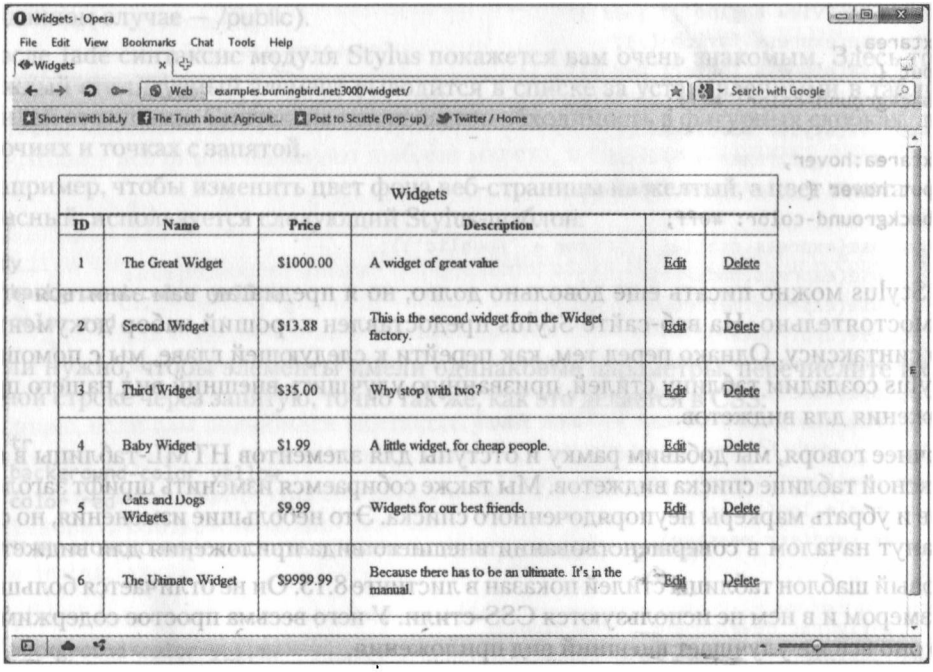


Рис. 8.5. Индексная страница виджет-приложения со стилевым оформлением, созданным с помощью Stylus

9 Получение структурированных данных в Node и Redis

Что касается данных, то есть реляционные базы данных и все остальное, известное как *не SQL*. В категории «не SQL» тип структурированных данных основан на парах ключ-значение, которые обычно хранятся в оперативной памяти для максимально быстрого доступа к ним. К трем наиболее популярным хранилищам подобного относятся Memcached, Cassandra и Redis. К счастью для Node-разработчиков, в Node поддерживаются все три хранилища.

Memcached используется главным образом как средство кэширования данных запросов для быстрого доступа к ним в памяти. Это хранилище также хорошо подходит для распределенных вычислений, но имеет ограниченную поддержку более сложных данных. Оно может быть полезно для приложений, работающих с множеством запросов, но хуже подходит для приложений, выполняющих большое количество операций записи и чтения данных. В то же время, для последнего типа приложений великолепно подходит хранилище данных Redis. Кроме того, Redis может быть надежным хранилищем, и оно значительно гибче хранилища Memcached, особенно в плане поддержания различных типов данных. Однако в отличие от Memcached хранилище Redis работает только на одной машине.

Те же факторы учитываются при сравнении Redis и Cassandra. Как и в Memcached, в Cassandra реализована поддержка кластеров. Однако так же как и Memcached, это хранилище имеет ограниченную поддержку структур данных. Этого вполне достаточно для специализированных запросов, для которых не подходит Redis. Зато Redis отличается простотой в использовании, это не сложное хранилище и к тому же оно обычно работает быстрее, чем Cassandra. По этим и по другим причинам хранилище Redis получило больше приверженцев среди Node-разработчиков, именно поэтому в данной главе при рассмотрении хранилищ данных в оперативной памяти в формате ключ-значение я отдал предпочтение этому хранилищу, а не Memcached и Cassandra.

Я собираюсь отказаться от стиля изложения предыдущих глав, больше подходящего для учебного пособия, и продемонстрировать возможности Node и Redis, реализовав три варианта использования этих двух технологий:

- Создание таблицы высших достижений в игре.
- Создание очереди сообщений.
- Отслеживание статистики веб-страницы.

В этих приложениях будут также использоваться модули и технологии, рассмотренные в предыдущих главах, например система шаблонов Jade (см. главу 8), модуль Async (см. главу 5) и платформа Express (см. главы 7 и 8).

Сайт Redis находится по адресу http://redis.io/. О Memcached можно узнать на сайте http://memcached.org/, а о Apache Cassandra — на сайте http://cassandra.apache.org/.

Начало работы с Node и Redis

Redis поддерживается несколькими модулями, включая модуль Redback, который предоставляет высокоуровневый интерфейс, но в данной главе мы сконцентрируемся только на модуле node_redis, или просто redis (именно этого названия мы будем придерживаться), созданным Мэттом Рэнни (Matt Ranney). Мне нравится redis, потому что он предлагает простой и элегантный интерфейс для непосредственного доступа к Redis-командам, поэтому вы можете воспользоваться своими знаниями об этом хранилище данных, практически не прибегая к дополнительным источникам информации.

GitHub-страница redis находится по адресу https://github.com/mranney/node_redis.

Модуль redis можно установить с помощью диспетчера Node-пакетов:

```
npm install redis
```

Я также рекомендую воспользоваться библиотекой hiredis, поскольку ее код является неблокирующим, что дает более высокую производительность. Установите эту библиотеку, воспользовавшись следующей командой:

```
npm install hiredis redis
```

Для использования redis в Node-приложениях этот модуль нужно сначала подключить:

```
var redis = require('redis');
```

Затем нужно создать Redis-клиента. Для этого используется метод `createClient`:

```
var client = redis.createClient();
```

Метод `createClient` может получать три дополнительных параметра: порт, хост и ключи (см. далее), по умолчанию в качестве хоста устанавливается значение 127.0.0.1, а в качестве порта — 6379. Для порта берется одно из тех значений, которые предлагаются по умолчанию для Redis-сервера, поэтому, если Redis-сервер находится на той же самой машине, что и Node-приложение, то значения, предлагаемые по умолчанию, нас вполне устроят.

Третьим параметром является объект, поддерживающий следующие ключи:

`parser`

: Парсер ответа Redis-протокола; по умолчанию имеет значение `hiredis`. Можно также воспользоваться парсером `javascript`.

`return_buffers`

: По умолчанию имеет значение `false`. Если устанавливается значение `true`, все ответы отправляются в виде Node-объектов буфера, а не в виде строк.

`detect_buffers`

: По умолчанию имеет значение `false`. Если устанавливается значение `true`, ответы отправляются в виде объектов буфера, если какой-либо ввод в исходных командах был представлен в виде буферов.

`socket_nodelay`

: По умолчанию имеет значение `true`; определяет, стоит ли вызывать `setNoDelay` для TCP-потока.

`no_ready_check`

: По умолчанию имеет значение `false`. Если устанавливается значение `true`, запрет на «проверку готовности» (ready check) отправляется на сервер для проверки его готовности к получению дополнительных команд.

Пока в работе Node и Redis вас все устраивает, используйте установки, предлагаемые по умолчанию.

Имея клиентское подключение к хранилищу данных Redis, вы можете отправлять команды на сервер, пока не будет вызван метод `client.quit`, который закрывает подключение к Redis-серверу. Для принудительного завершения работы можно вместо него воспользоваться методом `client.end`, но он завершает работу, не дожидаясь синтаксического разбора всех ответов. Поэтому метод `client.end` подходит в ситуации, когда ваше приложение зависло или вы решили начать все сначала.

Выдача Redis-команд через клиентское подключение является интуитивно понятным процессом. Все команды доступны в виде методов объекта клиента, а аргументы командам передаются в виде параметров. Поскольку это Node, последний параметр является функцией обратного вызова, которая возвращает ошибку и все, что является данными или ответом на Redis-команду.

В следующем коде метод `client.hset` используется для задания свойства хэша:

```
client.hset("hashid", "propname", "propvalue", function(err, reply) {
   // какие-либо действия с ошибкой или откликом
});
```

Команда hset устанавливает значение, поэтому нет никаких возвращаемых данных, есть только подтверждение от Redis. Если вызывается метод, который выдает несколько значений, например client.hvals, вторым параметром в функции обратного вызова будет массив: либо массив отдельных строк, либо массив объектов:

```
client.hvals(obj.member, function (err, replies) {
   if (err) {
      return console.error("error response - " + err);
   }

   console.log(replies.length + " replies:");
   replies.forEach(function (reply, i) {
     console.log("    " + i + ": " + reply);
   });
});
```

Поскольку в Node функции обратного вызова используются повсеместно, а в Redis так много команд, которые просто отвечают подтверждением об успешности выполнения, модуль redis предоставляет метод redis.print, передаваемый в качестве последнего параметра:

```
client.set("somekey", "somevalue", redis.print);
```

Метод redis.print выводит на консоль либо ошибку, либо ответ и возвращает управление.

Теперь, разобравшись с работой модуля redis, пора испытать его в реальных приложениях.

Создание таблицы высших достижений в игре

Одним из возможных применений Redis является создание таблицы высших достижений (рекордов) в игре. Такая таблица хранит показатели цифровых игр на компьютерах и портативных устройствах вроде смартфонов и планшетов. Одной из широко используемых таблиц является OpenFeint, которая позволяет игроку создать онлайновый профиль, а затем сохранять свои показатели в различных играх. Игроки могут соревноваться с друзьями или самостоятельно добиваться максимальных результатов в любой выбранной игре.

В такого типа приложениях решение вопроса о хранении данных может быть гибридным. Профили могут храниться в реляционном хранилище данных, а сами показатели — в таких хранилищах данных, как Redis. Данные, необходимые для показателей, довольно просты, к ним происходят частые обращения и они часто меняются многочисленными пользователями. Один из разработчиков игры оценил количество соперников в 10 000, а количество запросов в минуту в пиковые моменты игры — в 200 000. Тем не менее система обработки запросов не должна быть излишне сложной, поскольку данные не отличаются сложностью, да и транзакционные правила здесь реально не нужны. Честно говоря, применять здесь реляционную базу данных или базу данных документов — это перебор. Поэтому наиболее подходящей будет хранилище пар ключ-значение вроде Redis.

Наиболее подходящими структурами данных для такого типа приложений в Redis являются *хэш* и *отсортированный набор*. Хэш — это идеальный вариант, поскольку информация о каждом показателе занимает больше, чем одно или два поля. Обычно сохраняются идентификатор участника, возможно, имя игрока (чтобы ограничить частоту обращений к реляционному хранилищу или хранилищу документов), возможно, название игры, если система предоставляет таблицы высших достижений более чем для одной игры, последняя дата игры, сам показатель и любая другая сопутствующая информация.

Отсортированный набор является наиболее удобной структурой данных для отслеживания только показателей и имен пользователей, обеспечивая быстрый доступ к лучшим 10 или 100 показателям.

Для создания приложения, обновляющего базу данных Redis, я преобразовал клиент-серверное TCP-приложение, созданное в главе 3, для отправки данных от TCP-клиента на сервер с последующим обновлением данных в хранилище Redis. В хранении данных с помощью TCP-сокета, а не с помощью HTTP или каких-нибудь других средств, предназначенных для игровых приложений, нет ничего необычного.

TCP-клиент принимает все, что набирается в командной строке, и отправляет это на сервер. Код абсолютно такой же, как и в листинге 3.3, поэтому повторять его я не буду. В отличие от предыдущих проверок, при запуске TCP-клиента вместо отправки данных через простые текстовые сообщения я отправляю JSON-представление информации, хранящейся в базе данных Redis. Пример имеет следующий вид:

```
{"member" : 400, "first_name" : "Ada", "last_name" : "Lovelace", "score" : 53455,
"date" : "10/10/1840"}
```

Сервер изменен для преобразования строки данных, получаемой в JavaScript-объекте, и обращения к отдельным игрокам для сохранения их в хэше. Идентификатор игрока и его показатель также добавляются к отсортированному списку. В листинге 9.1 показано измененное приложение TCP-сервера.

Листинг 9.1. TCP-сервер, обновляющий хранилище данных Redis

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // создание Redis-клиента
   var client = redis.createClient();

   client.on('error', function(err) {
     console.log('Error ' + err);
   });

   // пятая база данных, являющаяся базой данных показателей игры
   client.select(5);
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
```

продолжение ⇩

Листинг 9.1 *(продолжение)*

```
        conn.remotePort);
      try {
         var obj = JSON.parse(data);

         // добавление или перезапись показателя
         client.hset(obj.member, "first_name", obj.first_name, redis.print);
         client.hset(obj.member, "last_name", obj.last_name, redis.print);
         client.hset(obj.member, "score", obj.score, redis.print);
         client.hset(obj.member, "date", obj.date, redis.print);

         // добавление показателя для игры Zowie!
         client.zadd("Zowie!", parseInt(obj.score), obj.member);
      } catch(err) {
         console.log(err);
      }
   });
   conn.on('close', function() {
        console.log('клиент закрыл подключение');
        client.quit();
   });

}).listen(8124);

console.log('listening on port 8124');
```

Redis-соединение устанавливается при создании сервера и закрывается при отключении сервера. Другой подход заключается в создании статического клиентского соединения, существующего между запросами, но у него имеются недостатки. Дополнительные сведения о том, когда создавать Redis-клиента, можно найти далее в соответствующей врезке. В Redis преобразование объектов и надежное сохранение данных реализовано посредством обработки исключений — это позволяет оградить сервер от падения в случае опечаток при вводе.

Как уже отмечалось, на данном этапе обновляются два разных хранилища данных: информация об индивидуальном счете (включая имя, счет и дату) хранится в хэше, а идентификатор игрока и счет — в отсортированном наборе данных. Идентификатор игрока используется в хэше в качестве ключа, в то время как счет игры — в качестве показателя для идентификатора игрока в отсортированном наборе. Важным для работы приложения является идентификатор игрока, который фигурирует в обоих хранилищах данных.

Следующей частью приложения является механизм вывода обладателей пяти наивысших показателей в нашей вымышленной игре (она называется Zowie!). В отсортированном наборе вы можете извлечь диапазон данных по показателям, используя Redis-команду `zrange`. Однако эта функция возвращает диапазон значений, отсортированный от самого низкого до самого высокого, что является результатом, обратным желаемому. А нам нужно, чтобы были возвращены первые пять наивысших результатов, отсортированных так, чтобы первым был наивысший из них. Для решения этой задачи используется Redis-команда `zrevrange`.

Теперь для вывода пятерки самых лучших игроков мы создадим HTTP-сервер, который вернет результаты в виде простого табличного списка. Чтобы обеспечить относительно приглядный внешний вид, мы задействуем систему шаблонов Jade, причем без Express, поскольку для нашего игрового приложения платформа Express не нужна.

Чтобы использовать Jade без Express, нужно прочитать исходный файл шаблона, а затем вызвать метод `compile`, передав ему строку файла шаблона и параметры. Единственным параметром является `filename` (имя файла), поскольку в файле шаблона я использую директиву `include`, для которой требуется имя файла. Фактически, я задаю имя файла шаблона и его местоположение, но вам нужно использовать любое имя файла, которое возвращает размещение каталога относительно файлов, включенных в Jade-шаблон.

Что касается самого шаблона, в листинге 9.2 показано содержимое Jade-файла. Обратите внимание на то, что я использую директиву `include` для непосредственной вставки CSS в файл. Поскольку в этом приложении статический файловый сервер я не применяю, приложение не сможет обработать CSS-файл, если я просто вставлю ссылку на него. Также обратите внимание на наличие канала (`|`) с открывающими и закрывающими тегами `style`, которые относятся к HTML-, а не к Jade-синтаксису. Причина в том, что Jade не обрабатывает включенный файл, если он указан внутри тега `style`.

Листинг 9.2. Файл Jade-шаблона для вывода пяти наивысших результатов

```
doctype 5
html(lang="en")
  head
    title Zowie! Top Scores
    meta(charset="utf-8")
    | <style type="text/css">
    include main.css
    | </style>
  body
    table
      caption Zowie! Top Scorers!
        tr
          th Score
          th Name
          th Date
          if scores.length
            each score in scores
              if score
                tr
                  td #{score.score}
                  td #{score.first_name} #{score.last_name}
                  td #{score.date}
```

Для визуализации шаблона приложение читает файл шаблона (в синхронном режиме, поскольку это происходит только единожды при первом запуске приложения), а затем использует его для компилирования функции шаблона:

```
var layout = require('fs').readFileSync(__dirname + '/score.jade', 'utf8');
var fn = jade.compile(layout, {filename: __dirname + '/score.jade'});
```

Далее Jade-функция компилирования может использоваться в любой момент, когда потребуется визуализировать HTML-разметку из шаблона, с передачей ей тех данных, которые ожидаются шаблоном:

```
var str = fn({scores : result});
res.end(str);
```

Смысл всего этого станет понятен, когда мы увидим готовое серверное приложение, а теперь давайте вернемся к Redis-части приложения.

В приложении наивысших результатов используются два Redis-вызова: `zrevrange` служит для получения диапазона показателей, а `hgetall` — для получения всех хэш-полей каждого игрока, упомянутого в наивысших показателях. И здесь все немного усложняется.

В реляционной базе данных объединение результатов из нескольких таблиц не составляет никакого труда, но в отношении доступа к данным в таком хранилище данных, как Redis, все обстоит иначе. Задача может быть решена, но так как мы имеем дело с Node-приложением, у нас возникают дополнительные сложности в придании каждому Redis-вызову асинхронного характера.

Здесь нам пригодится библиотека Async, которая рассматривалась в главе 5, где были продемонстрированы два Async-метода: `waterfall` и `parallel`. Одним из не продемонстрированных там методов является `series`, который идеально подходит для нашего случая. Redis-функции нужно вызывать по очереди, чтобы данные возвращались по очереди, но каждому промежуточному шагу не требуются данные предыдущих шагов. Функция `parallel` библиотеки Async позволяет инициировать все вызовы одновременно, что нам вполне подходит, но результаты каждого вызова возвращаются в случайном порядке, что не гарантирует возвращение наивысшего показателя первым. Функция `waterfall` нам не нужна, поскольку опять же каждый шаг не нуждается в данных предыдущего шага. Async-функция `series` гарантирует, что все вызовы Redis-метода `hgetall` делаются последовательно и данные возвращаются последовательно, но при этом берется в расчет, что каждый функциональный шаг не зависит от других.

Итак, теперь у нас есть способ вызова Redis-команд по очереди с гарантией, что данные будут возвращены в нужной последовательности, но код для решения этой задачи довольно грубый: нам приходится добавлять отдельный шаг в Async-методе `series` для каждого вызова Redis-метода `hgetall` и возвращать результат, как только будет возвращен каждый показатель. Когда требуется получить 5 значений, проблем не возникает, но что если понадобится вернуть 10 или 100 значений? Необходимость вручную вводить код для каждого Redis-вызова в Async-метод `series` будет становиться все более утомительной, что может привести к ошибкам в коде и к сложностям в его поддержании.

Приложение наивысших результатов обеспечивает последовательный перебор значений массива, возвращенных вызовом Redis-метода `zrevrange`, передавая каждое значение в качестве параметра функции по имени `makeCallbackFunc`. Эта вспомогательная функция занята только лишь возвращением функции обратного вызова,

которая инициирует вызов Redis-метода `hgetall`, используя параметр для получения данных конкретного игрока, а затем обращается к функции обратного вызова в последней строке своего обратного вызова — это требование Async, чтобы была возможность выстраивания результатов в цепочку. Функция обратного вызова, возвращенная из `makeCallbackFunc`, помещается в массив, и это именно тот массив, который отправляется в качестве параметра Async-методу `series`. Кроме того, поскольку модуль `redis` возвращает результат вызова метода `hgetall` в виде объекта, а Async-метод `series`, завершая свою работу, вставляет каждый объект в массив, то когда вся эта функциональность оказывается реализованной, мы можем просто взять конечный результат и передать его в движок шаблонов, чтобы сгенерировать текст для возвращения на сервер.

В листинге 9.3 показан весь код серверного приложения наивысших результатов. Хотя, казалось бы, проделан довольно большой объем работы, благодаря элегантности и удобству модулей Redis и Async кода получилось совсем не много.

Листинг 9.3. Служба предоставления наивысших результатов игры

```
var http = require('http');
var async = require('async');
var redis = require('redis');
var jade = require('jade');

// настройка Jade-шаблона
var layout = require('fs').readFileSync(__dirname + '/score.jade', 'utf8');
var fn = jade.compile(layout, {filename: __dirname + '/score.jade'});
// запуск Redis-клиента
var client = redis.createClient();

// выбор пятой базы данных
client.select(5);

// вспомогательная функция
function makeCallbackFunc(member) {
   return function(callback) {
      client.hgetall(member, function(err, obj) {
         callback(err,obj);
      });
   };
}
http.createServer(function(req,res) {

   // первый фильтр из запроса значка
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }
```

продолжение ⇩

Листинг 9.3 *(продолжение)*

```
    // получение показателей, выстраивание в обратном порядке
    // только первых пяти результатов
    client.zrevrange('Zowie!',0,4, function(err,result) {
       var scores;
       if (err) {
          console.log(err);
          res.end('Top scores not currently available, please check back');
          return;
       }

       // создание массива функций обратного вызова для вызова Async.series
       var callFunctions = new Array();

       // обработка результатов с помощью makeCallbackFunc с помещением
       // только что возвращенных результатов в массив
       for (var i = 0; i < result.length; i++) {
          callFunctions.push(makeCallbackFunc(result[i]));
       }

       // использование Async-метода series для обработки
       // каждого обратного вызова по очереди и возвращения
       // конечного результата в виде массива объектов
       async.series(
          callFunctions,
          function (err, result) {
             if (err) {
                console.log(err);
                res.end('Scores not available');
                return;
             }

             // передача массива объектов движку шаблонов
             var str = fn({scores : result});
             res.end(str);
        });
    });

}).listen(3000);

console.log('Server running on 3000/');
```

Перед созданием HTTP-сервера мы задаем функцию Jade-шаблона, а также устанавливаем функционирующего клиента для хранилища данных Redis. Когда к серверу делается новый запрос, мы отфильтровываем все запросы для файла favicon.ico (вызывать Redis для запроса favicon.ico не нужно), а затем обращаемся к пятерке наивысших результатов методом `zrevrange`. В том случае, если у приложения имеются результаты, оно использует Async-метод `series` для поочередной и последовательной обработки Redis-хэша запросов, чтобы получить назад упорядоченные результаты. Полученный в результате массив передается движку Jade-шаблонов.

На рис. 9.1 показано приложение после добавления к нему нескольких результатов для разных игроков.

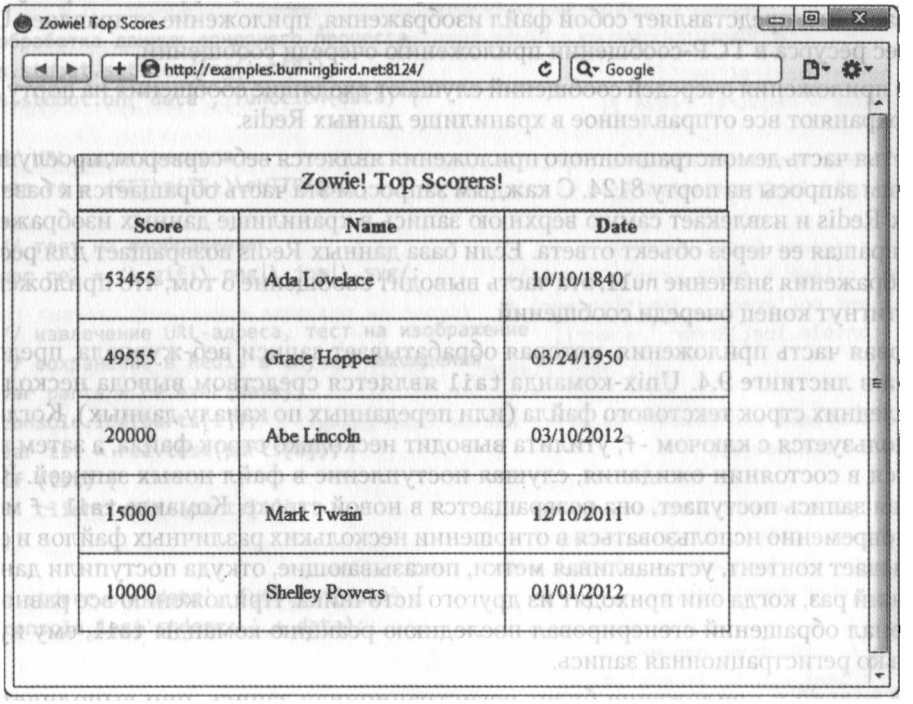


Рис. 9.1. Наивысшие показатели в игре Zowie!

Создание очереди сообщений

Очередь сообщений является приложением, которое получает на входе некую информацию, которая затем сохраняется в очереди. Сообщения сохраняются до тех пор, пока не будут извлечены получателем сообщения, в этом случае они удаляются из очереди и отправляются получателю (либо по одному, либо связкой). Взаимодействие имеет асинхронную природу, поскольку приложение, сохраняющее сообщения, не требует соединения с получателем, а получатель не требует соединения с приложением, сохраняющим сообщения.

Для такого типа приложений Redis является идеальным средством хранения. По мере того как сообщения поступают в приложение, которое их сохраняет, они помещаются в конец очереди сообщений. Когда сообщения извлекаются приложением, которое их получает, они извлекаются с верхушки очереди сообщений.

Чтобы продемонстрировать очередь сообщений, я создал Node-приложение для обращения к регистрационным веб-файлам нескольких различных поддоменов. Для доступа к последним записям различных регистрационных файлов в приложении используется дочерний Node-процесс и Unix-команда `tail -f`. Для этих

регистрационных записей приложение применяет два объекта регулярных выражений: одно для извлечения ресурса, к которому было обращение, второе для проверки, был ли этот ресурс файлом изображения. Если ресурс, к которому было обращение, представляет собой файл изображения, приложение отправляет URL-адрес ресурса в TCP-сообщении приложению очереди сообщений.

Все приложения очередей сообщений слушают входящие сообщения на порту 3000 и сохраняют все отправленное в хранилище данных Redis.

Третья часть демонстрационного приложения является веб-сервером, прослушивающим запросы на порту 8124. С каждым запросом эта часть обращается к базе данных Redis и извлекает самую верхнюю запись в хранилище данных изображений, возвращая ее через объект ответа. Если база данных Redis возвращает для ресурса изображения значение `null`, эта часть выводит сообщение о том, что приложением достигнут конец очереди сообщений.

Первая часть приложения, которая обрабатывает записи веб-журнала, представлена в листинге 9.4. Unix-команда `tail` является средством вывода нескольких последних строк текстового файла (или переданных по каналу данных). Когда она используется с ключом `-f`, утилита выводит несколько строк файла, а затем находится в состоянии ожидания, слушая поступление в файл новых записей. Когда такая запись поступает, она возвращается в новой строке. Команда `tail -f` может одновременно использоваться в отношении нескольких различных файлов и обрабатывает контент, устанавливая метки, показывающие, откуда поступили данные, всякий раз, когда они приходят из другого источника. Приложению все равно, чей журнал обращений сгенерировал последнюю реакцию команды `tail`, ему нужна только регистрационная запись.

Как только у приложения будет регистрационная запись, оно выполняет ряд сравнений данных с регулярными выражениями, находя обращения к ресурсам изображений (к файлам с расширением .jpg, .gif, .svg или .png). Если соответствие шаблону обнаруживается, приложение отправляет URL-адрес ресурса приложению очереди сообщений (TCP-серверу).

Листинг 9.4. Node-приложение, обрабатывающее записи веб-журнала и отправляющее запросы к ресурсам изображений в очереди сообщений

```
var spawn = require('child_process').spawn;
var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// соединение с к TCP-сервером
client.connect ('3000','examples.burningbird.net', function() {
    console.log('connected to server');
});

// запуск дочернего процесса
var logs = spawn('tail', ['-f',
        '/home/main/logs/access.log',
        '/home/tech/logs/access.log',
```

```
        '/home/shelleypowers/logs/access.log',
        '/home/green/logs/access.log',
        '/home/puppies/logs/access.log']);

// обработка данных дочернего процесса
logs.stdout.setEncoding('utf8');
logs.stdout.on('data', function(data) {

   // URL-адрес ресурса
   var re = /GET\s(\S+)\sHTTP/g;

   // тест на изображение
   var re2 = /\.gif|\.png|\.jpg|\.svg/;

   // извлечение URL-адреса, тест на изображение
   // сохранение в Redis в случае нахождения
   var parts = re.exec(data);
   console.log(parts[1]);
   var tst = re2.test(parts[1]);
   if (tst) {
      client.write(parts[1]);
   }
});
logs.stderr.on('data', function(data) {
   console.log('stderr: ' + data);
});

logs.on('exit', function(code) {
   console.log('child process exited with code ' + code);
   client.end();
});
```

Обычные консольные регистрационные записи для этого приложения заданы в следующем блоке кода; записи, представляющие интерес (обращение к файлу изображения), выделены полужирным шрифтом:

```
/robots.txt
/weblog
/writings/fiction?page=10
/images/kite.jpg
/node/145
/culture/book-reviews/silkworm
/feed/atom/
/images/visitmologo.jpg
/images/canvas.png
/sites/default/files/paws.png
/feeds/atom.xml
```

В листинге 9.5 содержится код для очереди сообщений. Это простое приложение, запускающее TCP-сервер и слушающее входящие сообщения. При получении сообщения из него извлекаются данные, которые сохраняются в базе данных Redis.

В приложении используется Redis-команда rpush для помещения данных в конец списка изображений (выделена полужирным шрифтом).

Листинг 9.5. Очередь сообщений, получающая входящие сообщения и помещающая их в Redis-список

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // создание Redis-клиента
   var client = redis.createClient();
   client.on('error', function(err) {
     console.log('Error ' + err);
   });

   // очередью изображений является шестая база данных
   client.select(6);

   // прослушивание входящих данных
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);

      // сохранение данных
      client.rpush('images',data);
   });

}).listen(3000);
server.on('close', function(err) {
   client.quit();
});

console.log('listening on port 3000');
```

Консольные записи приложения очереди сообщений обычно выглядят следующим образом:

```
listening on port 3000
connected
/images/venus.png from 173.255.206.103 39519
/images/kite.jpg from 173.255.206.103 39519
/images/visitmologo.jpg from 173.255.206.103 39519
/images/canvas.png from 173.255.206.103 39519
/sites/default/files/paws.png from 173.255.206.103 39519
```

Последней частью демонстрационного приложения очереди сообщений является HTTP-сервер, который слушает запросы, поступающие на порт 8124 (листинг 9.6). По мере получения HTTP-сервером каждого запроса он обращается к базе данных Redis, извлекает следующую запись из списка изображений и выводит запись

в ответе. Если записей в списке больше нет (то есть Redis возвращает в качестве ответа null), выводится сообщение о том, что очередь сообщений пуста.

Листинг 9.6. HTTP-сервер, извлекающий сообщения из Redis-списка и возвращающий их пользователю

```
var redis = require("redis"),
    http = require('http');

var messageServer = http.createServer();

// прослушивание входящих запросов
messageServer.on('request', function (req, res) {

   // сначало фильтрация запросов на значок
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }

  // создание Redis-клиента
  var client = redis.createClient();

  client.on('error', function (err) {
    console.log('Error ' + err);
  });

  // установка на шестую базу данных
  client.select(6);

   client.lpop('images', function(err, reply) {
      if(err) {
        return console.error('error response ' + err);
      }

      // если это данные
      if (reply) {
         res.write(reply + '\n');
      } else {
         res.write('End of queue\n');
      }
      res.end();
   });
   client.quit();
});

messageServer.listen(8124);

console.log('listening on 8124');
```

При обращении к приложению HTTP-сервера с помощью веб-браузера возвращается URL-адрес ресурса изображения каждого запроса (браузер обновляет изображение) до тех пор, пока очередь изображений не опустеет.

КОГДА СОЗДАВАТЬ REDIS-КЛИЕНТА

В примерах этой главы я иногда создаю Redis-клиента и сохраняю его на время работы приложения, в других случаях я создаю Redis-клиента и освобождаю его, как только завершается выполнение Redis-команды. Что же лучше, создавать постоянное Redis-соединение или создавать соединение и немедленно его разрывать?

Хороший вопрос.

Чтобы проверить влияние двух различных подходов, я создал TCP-сервер, который прослушивал запросы и сохранял простой хэш в базе данных Redis. Затем я создал еще одно приложение в качестве TCP-клиента, призванное лишь отправлять объект в TCP-сообщении на сервер.

Для запуска нескольких параллельных итераций клиента и тестирования продолжительности каждого запуска я воспользовался приложением ApacheBench. Я запустил первый пакет с Redis-клиентом, подключенным на время существования сервера, и запустил второй пакет, где соединение с клиентом устанавливалось для каждого запроса и тут же разрывалось.

Я ожидал, что приложение с постоянным клиентским соединением будет работать быстрее, и оказался прав. Примерно на полпути тестирования приложения с постоянным соединением его работа существенно замедлялась на короткий период времени, а затем относительно быстрыми темпами восстанавливалась.

Скорее всего, выстроенные в очередь запросы к базе данных Redis в конечном счете блокировали Node-приложение, по крайней мере, временно, до тех пор пока очередь не освобождалась. При открытии и закрытии соединения при каждом запросе в такую ситуацию я не попадал, потому что дополнительные издержки, необходимые для этого процесса, замедляли работу приложения, но не до такой степени, чтобы при параллельной работе пользователей достигались пиковые значения.

Дополнительные сведения об этом и других тестах, проводимых с помощью ApacheBench и других инструментов определения производительности и проведения отладки, имеются в главах 14 и 16.

Добавление к Express-приложению связующего модуля Stats

Создатель Redis изначально намеревался использовать эту технологию для создания статистического приложения. Это идеальный вариант применения Redis: простое хранилище данных, быстрые и часты записи, предоставление возможности для наращивания активности.

В этом разделе мы собираемся использовать Redis для добавления статистики к виджет-приложению, разработка которого начиналась в предыдущих главах. Статистика ограничивается двумя коллекциями: набором всех IP-адресов, с которых происходили обращения к страницам виджет-приложения, и количеством обращений к различным ресурсам. Для реализации этой функциональности мы воспользуемся Redis-*набором* и возможностью приращения значений цифровых строк. В нашем приложении также используется `multi` — средство управления Redis-транзакциями, позволяющее одновременно получать две отдельные коллекции данных.

Первым шагом приложения является добавление нового связующего программного обеспечения, записывающего информацию о доступе к базе данных Redis. Эта связующая программа использует Redis-набор и метод `sadd` для добавления каждого IP-адреса, поскольку набор гарантирует, что существующее значение не будет записано дважды. Мы собираем набор IP-адресов посетителей, но не отслеживаем время, когда посетитель обращается к ресурсу. В функции также используется одна из Redis-функций приращения значения, но не `incr`, которая дает приращение строкового значения, а `hincrby`, поскольку URL-адрес ресурса и связанный с ним счетчик обращений хранятся в виде хэша.

В листинге 9.7 показан код связующего программного обеспечения, которое находится в файле stats.js. Вторая база данных Redis используется для приложения, IP-адреса хранятся в наборе, идентифицируемом по `ip`, а хэш URL/счетчик хранится в хэше, идентифицируемом по `myurls`.

Листинг 9.7. Связующая программа Redis-stats

```
var redis = require('redis');
module.exports = function getStats() {

   return function getStats(req, res, next) {
      // создание Redis-клиента
      var client = redis.createClient();

      client.on('error', function (err) {
         console.log('Error ' + err);
      });

      // Установка на вторую базу данных
      client.select(2);

      // добавление IP к набору
      client.sadd('ip',req.socket.remoteAddress);

      // приращение значения счетчика обращений к ресурсам
      client.hincrby('myurls',req.url, 1);

     client.quit();
      next();
   }
}
```

Интерфейс статистики доступен в домене верхнего уровня, поэтому мы добавим код для маршрутизатора к файлу index.js в подкаталоге routes.

Сначала нам нужно добавить маршрут к файлу основного приложения сразу же после маршрута для индекса верхнего уровня:

```
app.get('/', routes.index);

app.get('/stats',routes.stats);
```

В коде контроллера статистического приложения управление Redis-транзакциями обеспечивается вызовом функции `multi`. Доступны два набора данных: набор уникальных IP-адресов, возвращаемый методом `smembers`, и хэш URL/счетчик, возвращаемый методом `hgetall`. Как показано в листинге 9.8, обе функции вызываются последовательно при вызове метода `exec`, и оба набора возвращаемых данных добавляются в виде элементов массива в методе обратного вызова, принадлежащего функции `exec`. После извлечения данных они передаются в вызов метода `render` новому представлению с именем `stats`. Новые функциональные возможности в файле index.js выделены полужирным шрифтом.

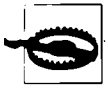
Листинг 9.8. Файл индекса маршрутов с новым кодом контроллера для статистического приложения

```
var redis = require('redis');

// главная страница
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};

// статистика
exports.stats = function(req, res){

   var client = redis.createClient();

   client.select(2);

   // Redis-транзакция для сбора данных
   client.multi()
   .smembers('ip')
   .hgetall('myurls')
   .exec(function(err, results) {
        var ips = results[0];
        var urls = results[1];
        res.render('stats',{ title: 'Stats', ips : ips, urls : urls});
        client.quit();
   });
};
```

Я уже упоминал, что `multi` и `exec` являются Redis-командами управления транзакциями. Это совсем не тот тип управления транзакциями, с которым вы, возможно,

сталкивались в реляционных базах данных. Команда `multi` всего лишь объединяет группу Redis-команд, которые затем обрабатываются последовательно, когда задана команда `exec`. Такой тип функциональности полезен в мире Node, поскольку позволяет получить несколько коллекций данных, возвращаемых одновременно, что исключает необходимость применения вложенных функций обратного вызова или каких-либо средств вроде Step или Async для одновременного получения всех данных.

Нужно сказать, что глядя как Redis-команды выстраиваются в цепочку, не стоит делать скоропалительный вывод о том, что данные из одной команды затем оказываются доступны следующей команде, как при выстраивании в цепочку JavaScript-функций. Каждая Redis-команда обрабатывается изолированно, и данные просто добавляются в результат в виде массива элементов и возвращаются все одновременно.



Во время транзакции блокирования данных, которой можно ожидать в отношении транзакции в реляционной базе данных, не возникает. Поэтому любые изменения в базе данных Redis в ходе обработки запроса могут влиять на результат.

Последней частью приложения является представление, созданное в виде Jade-шаблона. Шаблон предельно прост: IP-адреса выводятся в неупорядоченном списке, а статистика URL/счетчик — в таблице. Для последовательного перебора IP-массива используется Jade-конструкция `for...in`, а для обращения к именам свойств и значениям того объекта, который возвращается Redis-методом `hgetall`, — конструкция `each...in`. Шаблон показан в листинге 9.9.

Листинг 9.9. Jade-шаблон для статистического приложения

```
extends layout

block content
  h1= title

  h2 Visitor IP Addresses
  ul
    for ip in ips
      li=ip

  table
    caption Page Visits
    each val, key in urls
      tr
        td #{key}
        td #{val}
```

Страница статистики после доступа к нескольким страницам ресурсов приложения для виджетов с двух различных IP-адресов показана на рис. 9.2.

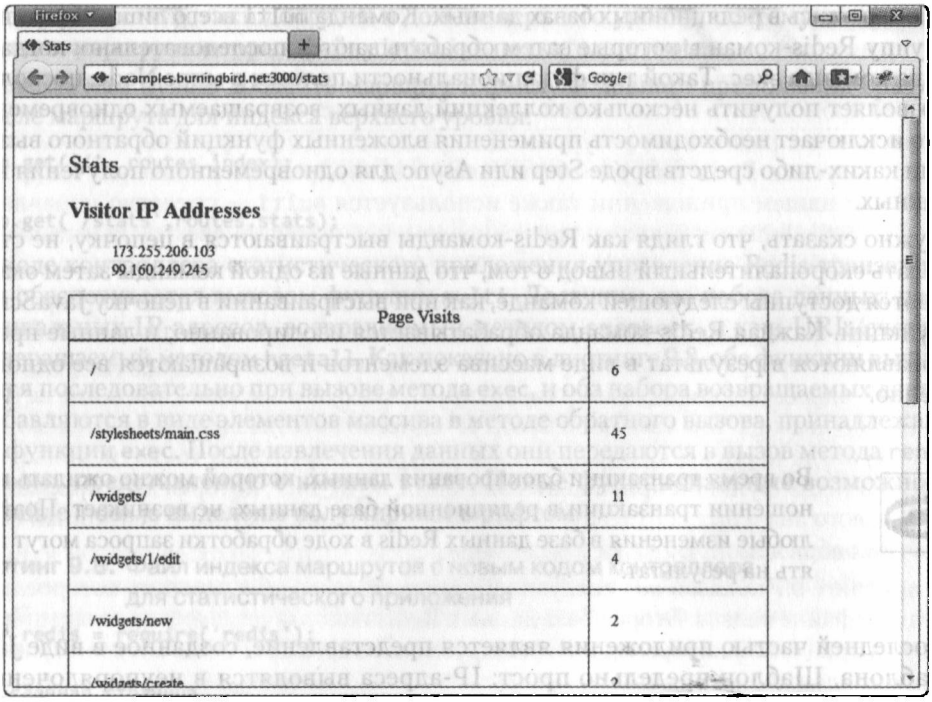


Рис. 9.2. Страница статистики, которую можно создать с помощью Redis

При использовании метода `hincrby` не обязательно сначала создавать хэш. Если ключ хэша отсутствует, Redis создает его автоматически и устанавливает для него нулевое значение, перед тем как это значение увеличить.

Сбой случается, только если поле уже существует, и значение в поле не является цифровой строкой (то есть не может быть увеличено).

Другим подходом к увеличению показателя счетчика каждого ресурса является использование Redis-строк и назначение ключом URL-адреса ресурса:

```
client.incr(url);
```

Однако этот подход означает, что мы должны были бы сначала получить все ключи (URL-адреса), а затем — счетчики для каждого URL-адреса. Мы не всегда можем выполнить все эти условия с помощью функции `multi`, а из-за асинхронной природы доступа к данным, нам, в конечном итоге, пришлось бы для объединения всех данных вместе использовать вложенные функции обратного вызова или какой-нибудь другой подход.

При наличии встроенной функциональности, доступной через Radis-хэш и метод `hincrby`, все эти дополнительные усилия абсолютно ни к чему.

10 Node и MongoDB: данные в формате документов

В главе 9 мы рассмотрели популярную структуру баз данных Redis, относящуюся к категории «не SQL» (пары ключ-значение), а в этой главе рассмотрим еще одну, MongoDB, являющуюся хранилищем данных, представленных в формате документов.

От реляционных систем баз данных, например от MySQL, MongoDB отличается тем, что структурированные данные в ней хранятся в виде документов, а не в виде более традиционных таблиц. Эти документы кодируются в формате BSON, который является двоичной формой JSON, чем, наверное, и объясняется популярность данного хранилища среди JavaScript-разработчиков. Вместо строки таблицы вы получаете BSON-документ, а вместо таблицы — коллекцию.

MongoDB не является единственной базой, в которой данные хранятся в формате документов. Другими популярными хранилищами данных этого типа являются CouchDB от Apache, SimpleDB от Amazon, RavenDB и даже древняя база данных Lotus Notes. Node в той или иной степени поддерживает большинство современных хранилищ данных, представленных в формате документов, но в наибольшей мере это относится к MongoDB и CouchDB. Я решил ограничиться рассмотрением MongoDB по тем же причинам, по которым выбрал Express среди других платформ: я считаю, что человеку проще усвоить Node-примеры, поскольку они не заставляют его слишком отвлекаться на второстепенные технологии, не имеющие отношение к Node (в данном случае это касается хранилищ данных). Благодаря MongoDB мы можем запрашивать данные напрямую, а в случае с CouchDB пришлось бы работать с представлениями. Это более высокий уровень абстракции, требующий больше времени на изучение. Я считаю, что с MongoDB дела пойдут быстрее, чем с CouchDB.

Существует несколько модулей, ориентированных на MongoDB, но я собираюсь сконцентрировать внимание на двух из них: MongoDB Native Node.js Driver — это

драйвер, написанный на JavaScript; а Mongoose — это инструмент объектного моделирования, обеспечивающий поддержку ORM (Object-Relational Mapping объектно-реляционное отображение).

Хотя в этой главе я не буду особо вдаваться в детали работы MongoDB, вам потребуется проявить свои способности и выполнять примеры, даже если раньше работать с системами баз данных вам не приходилось. Дополнительные сведения о MongoDB, включая справку по установке, можно найти по адресу http://www.mongodb.org/.

MongoDB Native Node.js Driver

Модуль MongoDB Native Node.js Driver является «родным» MongoDB-драйвером для Node. Получение MongoDB-инструкций с помощью этого драйвера несколько отличается от получения точно таких же инструкций в клиентском интерфейсе MongoDB.

GitHub-страница модуля node-mongodb-native находится по адресу https://github.com/mongodb/node-mongodb-native, а документация — по адресу http://mongodb.github.com/node-mongodb-native/.

После установки MongoDB (следуя инструкциям, имеющимся на веб-сайте MongoDB) и запуска базы данных установите с помощью диспетчера Node-пакетов модуль MongoDB Native Node.js Driver:

```
npm install mongodb
```

Перед тем как проверить работу любого из примеров в следующих нескольких разделах, убедитесь, что драйвер MongoDB установлен локально и запущен.

Если вы уже используете MongoDB, перед тем как запускать примеры из этой главы, позаботьтесь о создании резервной копии своих данных.

Начало работы с MongoDB

Чтобы воспользоваться драйвером MongoDB, сначала нужно затребовать модуль:

```
var mongodb = require('mongodb');
```

Чтобы установить соединение с MongoDB, воспользуйтесь конструктором объекта `mongodb.Server`:

```
var server = new mongodb.Server('localhost',:27017, {auto_reconnect: true});
```

Весь обмен данными с MongoDB осуществляется через TCP. Конструктор сервера принимает в качестве первых двух параметров хост и порт, в данном случае — используемый по умолчанию хост `localhost` и порт 27017. Третий параметр является набором вариантов настройки. Показанный в коде вариант настройки `auto_reconnect`, равный `true`, означает, что драйвер пытается восстановить разорванное соединение. Еще одним вариантом настройки является `poolSize`, с его помощью определяется, сколько TCP-соединений может обслуживаться параллельно.

В MongoDB для каждого соединения запускается новый программный поток, поэтому создатели базы данных рекомендуют разработчикам использовать пул соединений.

После установки соединения с MongoDB вы можете создать базу данных или подключиться к существующей базе данных. Для создания базы данных воспользуйтесь конструктором объекта `mongodb.Db`:

```
var db = new mongdb.Db('mydb', server);
```

Первым параметром является имя базы данных, вторым — соединение с MongoDB, третьим — объект с набором вариантов настройки. Варианты настройки, предлагаемые по умолчанию, вполне подходят для тех задач, которые мы собираемся решать в этой главе, другие описаны в документации по драйверу MongoDB, поэтому я не стану их здесь повторять.

Если вам еще не приходилось работать с MongoDB, то вы можете заметить, что коду не нужно предоставлять для аутентификации имя пользователя и пароль. По умолчанию MongoDB запускается без аутентификации. Раз система аутентификация отключена, драйвер должен запускаться в надежном окружении. Это означает, что MongoDB поддерживает соединение только с надежными хостами, обычно только с адресами локального хоста.

Определение, создание и удаление MongoDB-коллекции

Хотя в принципе MongoDB-коллекции являются эквивалентом таблиц реляционных баз данных, ничего в них такие таблицы не напоминает.

При определении MongoDB-коллекции вы можете указать, чтобы объект коллекции был создан прямо сейчас или только после добавления первой строки. Следующий код иллюстрирует разницу между двумя вариантами: первая инструкция на самом деле не создает никакой коллекции, в то время как вторая создает:

```
db.collection('mycollection', function(err, collection{});
db.createCollection('mycollection', function(err, collection{});
```

Обоим методам можно передать необязательный второй параметр, `{safe : true}`, который при использовании этого параметра с `db.collection` заставляет драйвер выдавать ошибку, если коллекция не существует, а при его использовании с `db.createCollection` выдавать ошибку, если коллекция уже существует:

```
db.collection('mycollection', {safe : true}, function (err, collection{});
db.createCollection('mycollection', {safe : true}, function(err, collection{});
```

Если вы применяете метод `db.createCollection` к существующей коллекции, вы просто получаете доступ к этой коллекции — драйвер не станет ее переписывать. Оба метода возвращают в функции обратного вызова объект коллекции, который вы затем можете использовать для добавления, изменения или извлечения данных в формате документов.

Если нужно полностью удалить коллекцию, воспользуйтесь методом `db.dropCollection`:

```
db.dropCollection('mycollection', function(err, result){});
```

Обратите внимание, что все эти методы являются асинхронными и зависят от вложенных функций обратного вызова, если вы хотите обрабатывать команды последовательно. Более полно это показано в следующем разделе, в котором мы добавим данные к коллекции.

Добавление данных к коллекции

Прежде чем заниматься механикой добавления данных к коллекции и рассматривать полностью работоспособные примеры, я хочу посвятить немного времени рассмотрению типов данных. Точнее, я хочу повторить то, что рассказывается в документации по драйверу MongoDB насчет типов данных, поскольку использование JavaScript привело к некоторым интересным соглашениям между драйвером и базой данных MongoDB.

В табл. 10.1 представлены типы данных, поддерживаемые MongoDB, и их JavaScript-эквиваленты. Следует заметить, что большинство преобразований весьма точны и не являются потенциальными источниками неожиданных эффектов. Однако для реализации некоторых из них пришлось сделать несколько допущений, о которых вы должны знать. Кроме того, некоторые типы данных, поддерживаемые драйвером MongoDB Native Node.js Driver, не имеют эквивалентов в MongoDB. Драйвер преобразует предоставляемые нами данные в те, что способна воспринять база данных MongoDB.

Таблица 10.1. Отображение типов данных драйвера Node.js MongoDB на JavaScript-данные

Тип MongoDB-данных	Тип JavaScript-данных	Примечания и примеры
JSON-массив	Array [1,2,3]	[1,2,3]
string	String	Кодировка utf8
boolean	Boolean	Значение true или false
integer	Number	MongoDB поддерживает 32- и 64-разрядные числа; JavaScript поддерживает 64-разрядные числа с плавающей точкой. Драйвер MongoDB пытается поместить значение в 32 разряда, а если это не получается, переходит к 64 разрядам, а если и это не получается, переходит к классу Long

Тип MongoDB-данных	Тип JavaScript-данных	Примечания и примеры
integer	Класс Long	Класс Long предоставляет полную поддержку 64-разрядных целых чисел
float	Number	
float	Класс Double	Специальный класс, представляющий значение с плавающей точкой
Date	Date	
Regular expression	RegExp	
null	null	
Object	Object	
Object id	Класс ObjectID	Специальный класс, в котором хранятся идентификаторы MongoDB-документов
Binary data	Класс Binary	Класс для хранения двоичных данных
	Класс Code	Класс для хранения JavaScript-функции и область видимости для метода, в которой он должен выполняться
	Класс DbRef	Класс для хранения ссылок на другой документ
	Класс Symbol	Определяет символ (специфичен не для JavaScript, а для языков, использующих символы)

Как только у вас будет ссылка на коллекцию, в нее можно будет добавить документ. Данные имеют структуру JSON, поэтому вы можете создать JSON-объект, а затем вставить его непосредственно в коллекцию.

Чтобы продемонстрировать весь рассмотренный на данный момент код и, кроме того, добавить данные к коллекции, в листинге 10.1 создается первая MongoDB-коллекция (с именем `Widgets`), а затем в нее добавляется два документа. На тот случай, если вам захочется запустить пример несколько раз, он сначала удаляет коллекцию документов, используя метод `remove`. Этому методу передаются три дополнительных параметра:

- Селектор документа (документов), если опущен, удаляются все документы.
- Необязательный индикатор безопасного режима:

```
safe {true | {w:n, wtimeout:n} | {fsync:true}, default:false}
```

- Функция обратного вызова (требуется в случае установки индикатора безопасного режима в `true`).

В примере приложение реализует безопасное удаление, передавая `null` для первого параметра (что обеспечивает удаление всех документов) и предоставляет функцию

обратного вызова. После удаления документа приложение вставляет два новых документа, используя при второй вставке безопасный режим. После второй вставки приложение выводит результат на консоль.

Метод вставки `insert` также получает три параметра: вставляемый документ (документы), параметр вариантов настройки и функцию обратного вызова. Вы можете вставить сразу несколько документов, поместив их в массив. Для метода `insert` используются следующие варианты настройки:

Безопасный режим

`safe {true | {w:n, wtimeout:n} | {fsync:true}, default:false}`

`keepGoing`

Установка в `true` заставляет приложение продолжать выполнение, когда вставка одного из документов вызвала ошибку.

`serializeFunctions`

Сериализация функций в документе.

Поскольку вызовы методов осуществляются в асинхронном режиме, нет никаких гарантий, что первый документ будет вставлен перед вторым. Однако в случае приложения для виджетов это проблемой не является, по крайней мере, в данном примере. Однако далее в главе мы более внимательно рассмотрим ряд сложностей, касающихся асинхронной работы с приложениями для баз данных.

Листинг 10.1. Создание или открытие базы данных с удалением всех документов и добавлением двух новых документов

```
var mongodb = require('mongodb');

var server = new mongodb.Server('localhost', 27017, {auto_reconnect: true});
var db = new mongodb.Db('exampleDb', server);

// открытие соединения с базой данных
db.open(function(err, db) {
  if(!err) {

    // доступ к коллекции виджетов или создание этой коллекции
    db.collection('widgets', function(err, collection) {

      // удаление всех виджет-документов
      collection.remove(null,{safe : true}, function(err, result) {
        if (!err) {
          console.log('result of remove ' + result);

          // создание двух записей
          var widget1 = {title : 'First Great widget',
                         desc : 'greatest widget of all',
                         price : 14.99};
          var widget2 = {title : 'Second Great widget',
                         desc : 'second greatest widget of all',
                         price : 29.99};
```

```
            collection.insert(widget1);

            collection.insert(widget2, {safe : true}, function(err, result) {
               if(err) {
                  console.log(err);
               } else {
                  console.log(result);

                  // закрытие базы данных
                  db.close();
               }
            });
         }
      });
   });
  }
});
```

Вывод на консоль после второй вставки может выглядеть так:

```
[  { title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99,
    _id: 4fc108e2f6b7a3e252000002 } ]
```

Для каждого документа MongoDB создает уникальный системный идентификатор. Хотя после этого доступ к документу можно получить с помощью этого идентификатора, вам лучше сделать для каждого документа более осмысленный идентификатор, позволяющий легко определить контекст использования.

Как уже упоминалось, мы можем одновременно вставить сразу несколько документов, предоставив методу не отдельный документ, а массив документов. В следующем коде показано, как обе записи о виджетах могут быть вставлены одной командой. Код также включает в себя идентификатор приложения с полем `id`:

```
// создание двух записей
var widget1 = {id: 1, title : 'First Great widget',
  desc : 'greatest widget of all',
  price : 14.99};
var widget2 = {id: 2, title : 'Second Great widget',
  desc : 'second greatest widget of all',
  price : 29.99};

collection.insert([widget1,widget2], {safe : true}, function(err, result) {
  if(err) {
     console.log(err);
  } else {
     console.log(result);

     // закрытие базы данных
     db.close();
  }
});
```

При пакетной вставке документов вам нужно установить параметр `keepGoing` в `true`, чтобы вставка документов продолжалась, даже если одна из вставок завершится неудачей. По умолчанию если при вставке происходит ошибка, приложение останавливает свою работу.

Запрос данных

В MongoDB Native Node.js Driver имеется четыре метода поиска данных:

`find`

Возвращает курсор со всеми документами, возвращенными запросом.

`findOne`

Возвращает указатель на первый документ, соответствующий запросу.

`findAndRemove`

Находит, а затем удаляет документ.

`findAndModify`

Находит документ, а затем выполняет некое действие (например, `remove` или `upsert`).

В данном разделе рассматриваются только методы `collection.find` и `collection.findOne`, а остальные два я приберегу для следующего раздела.

Оба метода, `collection.find` и `collection.findOne`, поддерживают три аргумента: запрос, варианты настройки и функцию обратного вызова. Объект вариантов настройки и функция обратного вызова являются необязательными. В табл. 10.2 представлен перечень возможных вариантов настройки для использования в обоих методах.

Таблица 10.2. Варианты настройки методов поиска

Настройка	Значение по умолчанию	Описание
limit	Число, по умолчанию равное нулю	Ограничивает количество возвращаемых документов (0 означает отсутствие ограничений)
sort	Массив индексов	Набор для сортировки документов, возвращаемых запросом
fields	Объект	Поля для возвращения запросом. Используйте имя свойства и значение 1 для включения или 0 для исключения; то есть {'свойство' : 1} или {'свойство' : 0}, но не оба вместе
skip	Число, по умолчанию равное нулю	Приводит к пропуску n документов (полезно для разбиения на страницы)

Настройка	Значение по умолчанию	Описание
hint	Объект	Предписывает базе данных использование конкретных индексов, {'_id' : 1}
explain	Булево значение, по умолчанию равное false	Приводит к расширению запроса вместо возвращения данных
snapshot	Булево значение, по умолчанию равное false	Приводит к выдаче краткой характеристики запроса (в MongoDB должно быть включен режим ведения журнала)
timeout	Булево значение, по умолчанию равное false	Задает время действия курсора
tailable	Булево значение, по умолчанию равное false	Включает режим установки курсора на конец коллекции (только на закрытых коллекциях, допускающих продолжение поиска или поиск с извлечением, соответствует Unix-команде tail)
batchSize	Число, по умолчанию равное нулю	Приводит к установке параметра batchSize для метода getMoreCommand при последовательном переборе результатов
returnKey	Булево значение, по умолчанию равное false	Приводит к возвращению только индексного ключа
maxScan	Число	Ограничивает количество сканируемых элементов
min	Число	Устанавливает границы индексов
max	Число	Устанавливает границы индексов
showDiskLoc	Булево значение, по умолчанию равное false	Показывает местонахождение диска с результатами
comment	Строка	Приводит к добавлению комментария к запросу для регистрационных записей профайлера
raw	Булево значение, по умолчанию равное false	Приводит к возвращению BSON-результатов в виде строкового буфера документов
read	Булево значение, по умолчанию равное false	Направляет запрос на вспомогательный сервер

Варианты настройки позволяют повысить гибкость запросов, хотя большинству запросов обычно достаточно лишь нескольких. Кое-какие варианты настройки

описаны в примерах, остальные я рекомендую проверить самостоятельно на своей копии MongoDB.

Самый простой запрос на получение всех документов коллекции заключается в вызове метода `find` без параметров. Результаты сразу же преобразуются в массив методом `toArray` и передаются в функцию обратного вызова, которая принимает ошибку и массив документов. В листинге 10.2 показано приложение, реализующее эту функциональность.

Листинг 10.2. Вставка четырех документов с последующим их извлечением с помощью метода find

```
var mongodb = require('mongodb');

var server = new mongodb.Server('localhost', 27017, {auto_reconnect: true});
var db = new mongodb.Db('exampleDb', server);

// открытие соединения с базой данных
db.open(function(err, db) {
  if(!err) {

    // доступ к коллекции виджетов или создание этой коллекции
    db.collection('widgets', function(err, collection) {

      // удаление всех виджет-документов
      collection.remove(null,{safe : true}, function(err, result) {
        if (!err) {

          // создание четырех записей
          var widget1 = {id: 1, title : 'First Great widget',
                         desc : 'greatest widget of all',
                         price : 14.99, type: 'A'};
          var widget2 = {id: 2, title : 'Second Great widget',
                         desc : 'second greatest widget of all',
                         price : 29.99, type: 'A'};
          var widget3 = {id: 3, title: 'third widget',
                         desc: 'third widget',
                         price : 45.00, type: 'B'};
          var widget4 = {id: 4, title: 'fourth widget',
                         desc: 'fourth widget',
                         price: 60.00, type: 'B'};

          collection.insert([widget1,widget2,widget3,widget4],
                         {safe : true}, function(err, result) {

            if(err) {
              console.log(err);
            } else {

              // возвращение всех документов
```

```
                collection.find().toArray(function(err, docs) {
                  console.log(docs);

                  // закрытие базы данных
                  db.close();
                });
              }
            });
          }
        });
      });
  }
});
```

Результаты, выведенные на консоль, показывают все четыре добавленных документа с идентификаторами, созданными системой:

```
[ { id: 1,
    title: 'First Great widget',
    desc: 'greatest widget of all',
    price: 14.99,
    type: 'A',
    _id: 4fc109ab0481b9f652000001 },
  { id: 2,
    title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99,
    type: 'A',
    _id: 4fc109ab0481b9f652000002 },
  { id: 3,
    title: 'third widget',
    desc: 'third widget',
    price: 45,
    type: 'B',
    _id: 4fc109ab0481b9f652000003 },
  { id: 4,
    title: 'fourth widget',
    desc: 'fourth widget',
    price: 60,
    type: 'B',
    _id: 4fc109ab0481b9f652000004 } ]
```

Чтобы не возвращать все документы целиком, можно использовать селектор. В следующем коде мы запрашиваем все документы, имеющие тип A, и возвращаем все поля, кроме поля type:

```
// возвращение всех документов, имеющих тип A
collection.find({type:'A'},{fields:{type:0}}).toArray(function(err, docs) {
   if(err) {
     console.log(err);
   } else {
```

продолжение

```
    console.log(docs);

    // закрытие базы данных
    db.close();
  }
});
```

Результаты запроса будут следующими:

```
[ { id: 1,
    title: 'First Great widget',
    desc: 'greatest widget of all',
    price: 14.99,
    _id: 4f7ba035c4d2204c49000001 },
  { id: 2,
    title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99,
    _id: 4f7ba035c4d2204c49000002 } ]
```

Мы также можем обратиться только к одному документу, используя для этого метод `findOne`. Результат этого запроса не нужно преобразовывать в массив, поэтому к нему можно обратиться напрямую. В следующем коде запрашивается документ с идентификатором 1 и возвращается только заголовок документа:

```
// возвращение только одного документа
collection.findOne({id:1},{fields:{title:1}}, function(err, doc) {
  if (err) {
     console.log(err);
  } else {
      console.log(doc);

      // закрытие базы данных
      db.close();
  }
});
```

Результат этого запроса будет иметь следующий вид:

```
{ title: 'First Great widget', _id: 4f7ba0fcbfede06649000001 }
```

С результатами запроса всегда возвращается созданный системой идентификатор.

Даже если я изменю запрос на возвращение всех документов, имеющих тип A (а их всего два), метод `collection.findOne` возвратит только один документ. Изменение параметра `limit` в объекте вариантов настройки ситуацию не изменит: метод при успешном выполнении запроса всегда возвращает только один документ.

Обновления, обновления со вставкой, поиск и удаление

MongoDB Native Node.js Driver поддерживает несколько методов, которые либо модифицируют, либо удаляют существующий документ, либо делают то и другое в одном вызове:

`update`

Либо обновляет документ, либо обновляет со вставкой (добавляет документ, если такого документа еще не существует).

`remove`

Удаляет документ.

`findAndModify`

Находит и изменяет либо удаляет документ (возвращая модифицированный или удаленный документ).

`findAndRemove`

Находит и удаляет документ (возвращая удаленный документ).

Основное различие между методами `update`/`remove` и `findAndModify`/`findAndRemove` состоит в том, что последний набор методов возвращает обрабатываемый документ.

Функциональность этих методов не слишком отличается от функциональности методов вставки. Вам нужно открыть соединение с базой данных, получить ссылку на интересующую вас коллекцию, а затем выполнить операции.

Предположим, в MongoDB содержится следующий документ:

```
{ id : 4,
   title: 'fourth widget',
   desc: 'fourth widget'.
   price: 60.00,
   type: 'B'}
```

Чтобы изменить заголовок (`title`) этого документа, можно воспользоваться методом `update`, как показано в листинге 10.3. Можно предоставить все поля, и MongoDB заменит документ, но лучше воспользоваться одним из модификаторов, доступных в MongoDB, например `$set`. Модификатор `$set` заставляет базу данных модифицировать только те поля, которые переданы модификатору в виде свойств.

Листинг 10.3. Обновление MongoDB-документа

```
var mongodb = require('mongodb');

var server = new mongodb.Server('localhost', 27017, {auto_reconnect: true});
var db = new mongodb.Db('exampleDb', server);

// открытие соединения с базой данных
db.open(function(err, db) {
  if(!err) {

    // доступ к коллекции виджетов или создание этой коллекции
    db.collection('widgets',function(err, collection) {
```

продолжение

Листинг 10.3 *(продолжение)*

```
        // обновление
        collection.update({id:4},
          {$set : {title: 'Super Bad Widget'}},
            {safe: true}, function(err, result) {
           if (err) {
              console.log(err);
           } else {
              console.log(result);

              // запрос на обновленнную запись
              collection.findOne({id:4}, function(err, doc) {
                 if(!err) {
                    console.log(doc);

                    // закрытие базы данных
                    db.close();
                 }
               });
           }
          });
      });
   }
});
```

Теперь в итоговом документе выводятся обновленные поля.

Модификатор $set можно использовать с несколькими полями.

Есть и другие модификаторы, обеспечивающие интересные атомарные обновления данных:

`$inc`

Приращение значения поля на конкретную величину.

`$set`

Устанавливает поле, как было показано в предыдущем примере.

`$unset`

Удаление поля

`$push`

Добавление значения к массиву, если поле является массивом (с преобразованием поля в массив, если поле им не является).

`$pushAll`

Добавление нескольких значений к массиву.

`$addToSet`

Добавление к массиву, только если поле является массивом.

`$pull`

Удаление значения из массива.

`$pullAll`

Удаление из массива нескольких значений.

`$rename`

Переименование поля.

`$bit`

Выполнение поразрядной операции.

А почему же вместо использования модификатора просто не удалить документ и не вставить новый? Потому что хотя мы должны предоставить все определенные пользователем поля, созданный системой идентификатор мы предоставлять не должны. Это значение при обновлении остается неизменным. Если созданный системой идентификатор хранится в виде поля в другом документе, например в родительском, удаление документа оставит «сиротой» ссылку на исходный документ в родительском документе.

Хотя в этой главе концепция деревьев (сложных структур данных родители-дети) не рассматривается, документацию по ним можно найти на вебсайте MongoDB.

Более важным обстоятельством является то, что модификаторы обеспечивают выполнение операции на месте, давая некоторые гарантии того, что обновление, внесенное одним пользователем, не отменит обновление, сделанное другим пользователем.

Хотя в примере мы их не использовали, метод `update` имеет четыре варианта настройки:

- `safe` — безопасное обновление.
- `upsert` — булево значение, устанавливается в `true`, если вставка должна осуществляться при отсутствующем документе (по умолчанию имеет значение `false`).
- `multi` — булево значение, устанавливается в `true`, если должны быть обновлены все документы, соответствующие критерию выбора.
- `serializeFunction` — сериализация функций в документе.

Если вы не уверены в том, что документ уже присутствует в базе данных, установите параметр `upsert` в `true`.

В листинге 10.3 осуществляется поиск модифицированной записи, чтобы изменения вступили в силу. В данном случае было бы лучше воспользоваться методом

findAndModify. Параметры этого метода похожи на параметры метода update с добавлением в качестве второго параметра массива сортировки. Если возвращается несколько документов, обновление выполняется в порядке сортировки:

```
// обновление
collection.findAndModify({id:4}, [[ti]],
  {$set : {title: 'Super Widget', desc: 'A really great widget'}},
    {new: true}, function(err, doc) {
   if (err) {
     console.log(err);
   } else {
     console.log(doc);DB
   }
   db.close();
});
```

Метод findAndModify с параметром remove можно использовать для удаления документа. При этом в функции обратного вызова никакого документа не возвращается. Для удаления документа можно также воспользоваться методами remove и findAndRemove. В предыдущих примерах для удаления всех документов перед вставкой вызывался метод remove без селектора. Для удаления конкретного документа нужно предоставить селектор:

```
collection.remove({id:4},
    {safe: true}, function(err, result) {
  if (err) {
    console.log(err);
  } else {
    console.log(result);
  }
```

В качестве результата возвращается количество удаленных документов (в данном случае — 1). Чтобы увидеть удаляемый документ, следует воспользоваться функцией findAndRemove:

```
collection.findAndRemove({id:3}, [['id',1]],
    function(err, doc) {
  if (err) {
    console.log(err);
  } else {
    console.log(doc);
  }
```

Я рассмотрел основные CRUD-операции, которые могут выполняться из Node-приложений с Native-драйвером, но существующие возможности гораздо шире и включают в себя работу с закрытыми коллекциями, индексами, другими MongoDB-модификаторами, а также разбиение (sharding) данных, то есть распределение данных между машинами, и многое другое. Все это вместе с хорошими примерами можно найти в документации к Native-драйверу.

Ряд сложностей, возникающих при доступе к данным в асинхронной среде, более полно рассматриваются в следующей врезке.

СЛОЖНОСТИ АСИНХРОННОГО ДОСТУПА К ДАННЫМ

Одним из тонких мест асинхронной обработки и доступа к данным является уровень вложенности, позволяющий гарантировать, что одна операция завершится до того, как начнется другая. В нескольких последних разделах у вас была возможность посмотреть, насколько быстро осуществляются обратные вызовы для некоторых простых операций: доступе к MongoDB, получении ссылки на коллекцию, выполнении некой операции и проверки, что она действительно выполнена.

В документации по MongoDB Native Node.js Driver содержатся примеры, в которых разработчики использовали таймер, гарантирующий завершение предыдущий функции перед выполнением следующей. Данный подход вам вряд ли подойдет. Чтобы избежать проблем со слишком глубокой вложенностью функций обратного вызова, можно задействовать либо именованные функции, либо один из асинхронных модулей, например Step или Async.

Но лучше всего обеспечить минимально необходимую функциональность каждого метода, обновляющего базу данных MongoDB. Если у вас будут затруднения с отказом от чрезмерного использования вложенных функций обратного вызова, а приложение будет трудно переделать для работы с такими модулями, как Async, то, скорее всего, вы слишком сильно увлеклись сложностью. В таком случае вам нужно рассмотреть возможность разбиения сложных функций, выполняющих многочисленные обращения к базе данных, на более управляемые фрагменты.

При асинхронном программировании простота приветствуется.

Реализация виджет-модели с помощью Mongoose

Хотя драйвер MongoDB Native Node.js Driver обеспечивает привязку к базе данных MongoDB, он не предоставляет высокоуровневую абстракцию. Для этого нужно использовать какой-либо механизм ODM (Object-Document Mapping — отображение объекта на документ), например Mongoose.

Веб-сайт Mongoose можно найти по адресу http://mongoosejs.com/.

Чтобы воспользоваться Mongoose, установите этот модуль с помощью диспетчера Node-пакетов:

```
npm install mongoose
```

Вместо использования команд для непосредственного обращения к базе данных MongoDB, вы определяете объекты с помощью Mongoose-объекта `Schema`, а затем синхронизируете его с базой данных с помощью Mongoose-объекта `model`:

```
var Widget = new Schema({
  sn : {type: String, require: true, trim: true, unique: true},
  name : {type: String, required: true, trim: true},
  desc : String,
  price : Number
});

var widget = mongoose.model('Widget', Widget);
```

При определении объекта мы предоставляем информацию, управляющую всем, что происходит впоследствии с полем документа. В только что предоставленном коде мы определили объект `Widget` с четырьмя явно указанными полями, три из которых имеют тип `String`, а одно `Number`. Поля `sn` и `name` оба являются обязательными и усеченными, а поле `sn` должно быть в базе данных документов уникальным.

К этому времени коллекция еще не создана и она не будет создана до тех пор, пока не будет создан как минимум один документ. А после создания ей присваивается имя `widgets` — это имя виджет-объекта в нижнем регистре и во множественном числе.

Когда нужен доступ к коллекции, совершается точно такой же вызов метода `mongoose.model`.

Этот код является первым шагом в добавлении завершающего компонента к реализации системы виджетов формата Model-View-Controller (MVC), которая начиналась в предыдущих главах. В следующих двух разделах мы завершим преобразование от хранилища данных в памяти к MongoDB. Но сначала нам нужно немного переделать наше приложение для виджетов.

Переделка фабрики виджетов

Переделка (refactoring) подразумевает реструктуризацию существующего кода, таким образом, чтобы влияние на пользовательский интерфейс было минимальным или нулевым. Поскольку мы переделываем приложение для виджетов, чтобы оно могло работать с базой данных MongoDB, теперь самое время узнать, какие еще изменения требуются.

На данное время структура файловой системы приложения для виджетов имеет следующий вид:

```
/каталог приложения
   /routes - главный каталог контроллеров
   /controllers - объекты контроллеров
   /public - статические файлы
      /widgets
   /views - файлы шаблонов
      /widgets
```

Подкаталог routes предоставляет функциональность верхнего уровня (не относящуюся к бизнес-объекту). Имя невыразительное, поэтому я переименовал его в main (главный). Это потребовало небольших изменений в файле app.js:

```
// верхний уровень
app.get('/', main.index);
app.get('/stats', main.stats);
```

Затем я добавил новый подкаталог с именем models. В этом каталоге хранятся определения модели MongoDB, точно так же, как код контроллеров находится в подкаталоге controllers. Структура каталогов теперь имеет следующий вид:

```
/каталог приложения
   /main - главный каталог контроллеров
   /controllers - объекты контроллеров
   /public - статические файлы
      /widgets
   /views - файлы шаблонов
      /widgets
```

Другое изменение в приложении касается структуры данных. Сейчас первичным ключом приложения является поле ID, созданное системой, но доступное пользователю через систему маршрутизации. Для вывода виджета нужно воспользоваться следующим URL-адресом:

```
http://localhost:3000/widgets/1
```

Этот подход не отличается оригинальностью. При отсутствии модуля перенаправления URL-адресов аналогичным образом действует популярная CMS-система Drupal для доступа к Drupal-узлам (статьям) и пользователям:

```
http://burningbird.net/node/78
```

Проблема в том, что MongoDB создает для каждого объекта идентификатор, формат которого неудобен для маршрутизации. Но есть обходной маневр, требующий создания третьей коллекции, предназначенной для идентификатора, но этот подход не отличается элегантностью, поскольку идет вразрез с положенной в основу MongoDB структуре и не всегда работает с Mongoose.

Уникальным является поле заголовка виджета, но у него есть пробелы и символы, мешающие его использованию для маршрутизации в качестве URL-адреса. Вместо него мы определим новое поле по имени sn, которое будет содержать новый регистрационный номер продукта. При создании нового виджет-объекта пользователь назначает продукту желаемый регистрационный номер, который потом служит для обращения к виджету. Если, к примеру, регистрационным номером виджета является 1A1A, к нему можно обратиться следующим образом:

```
http://localhost:3000/widgets/1A1A
```

С точки зрения приложения новая структура данных имеет следующий вид:

```
sn: string
title: string
desc: string
price: number
```

Это изменение требует небольшой модификации пользовательского интерфейса, но она стоит потраченного времени. Также должны быть изменены и Jade-шаблоны, но эти изменения минимальны — в основном они касаются замены ссылок на `id` ссылками на `sn` и добавления к каждой форме поля регистрационного номера.

Чтобы заново не приводить весь код после внесения минимальных изменений, я создал примеры для этой книги (http://oreilly.com/catalog/9781449323073). Там в подкаталоге chap12 вы найдете все новые файлы приложения для виджетов.

Более существенные изменения требуется внести в код контроллера в файле widget.js. Изменения, вносимые в этот и другие файлы, связаны с добавлением серверной части MongoDB, но об этом рассказывается в следующем разделе.

Добавление серверной части MongoDB

Первое изменение касается соединения с базой данных MongoDB. Оно добавляется в главный файл app.js, а это означает, что соединение сохраняется на протяжении всего времени жизни приложения.

В первую очередь в файл включается модуль Mongoose:

```
var mongoose = require('mongoose');
```

Затем осуществляется подключение к базе данных:

```
// MongoDB
mongoose.connect('mongodb://127.0.0.1/WidgetDB');
mongoose.connection.on('open', function() {
   console.log('Connected to Mongoose');
});
```

Обратите внимание на URI для MongoDB. Конкретная база данных передается в качестве последней части URI.

Этим изменением и вышеупомянутым изменением, касающимся преобразования routes в main, и ограничиваются все изменения, которые необходимо было внести в файл app.js.

Следующее изменение касается файла maproutecontroller.js. Маршруты, в которых есть ссылка на `id`, должны быть изменены с указанием теперешней ссылки на `sn`. Измененные маршруты показаны в следующем блоке кода:

```
// показ
app.get(prefix + '/:sn', prefixObj.show);

// редактирование
app.get(prefix + '/:sn/edit', prefixObj.edit);

// обновление
app.put(prefix + '/:sn', prefixObj.update);
```

```
// удаление
app.del(prefix + '/:sn', prefixObj.destroy);
```

Если эти изменения не внести, код контроллера будет ожидать в качестве параметра `sn`, а вместо него получит `id`.

Следующий код является дополнением, а не изменением. В подкаталоге models создается новый файл с именем **widgets.js**. Именно в нем определяется модель приложения виджетов. Чтобы модель была доступна за пределами файла, она, как показано в листинге 10.4, экспортируется.

Листинг 10.4. Определение новой модели приложения для виджетов

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema
    ,ObjectId = Schema.ObjectId;

// создание новой модели
var Widget = new Schema({
  sn : {type: String, require: true, trim: true, unique: true},
  name : {type: String, required: true, trim: true},
  desc : String,
  price : Number
});

module.exports = mongoose.model('Widget', Widget);
```

Последнее изменение касается кода контроллера приложения для виджетов. Мы отказались от хранилища данных в памяти в пользу MongoDB, используя для этого модель Mongoose. Хотя с точки зрения функционирования это изменение является весьма существенным, объем изменений кода не так уж велик, всего лишь несколько правок, как и при изменении `id` на `sn`. Окончательный вариант кода контроллера приложения виджетов показан в листинге 10.5.

Листинг 10.5. Измененный код контроллера приложения для виджетов

```
var Widget = require('../models/widget.js');

// индексный список виджетов в каталоге /widgets/
exports.index = function(req, res) {
   Widget.find({}, function(err, docs) {
      console.log(docs);
      res.render('widgets/index', {title : 'Widgets', widgets : docs});
   });
};

// вывод формы нового виджета
exports.new = function(req, res) {
    console.log(req.url);
```

продолжение ⇩

Листинг 10.5 *(продолжение)*

```
    var filePath = require('path').normalize(__dirname +
                                    "/../public/widgets/new.html");
    res.sendfile(filePath);
};

// добавление виджета
exports.create = function(req, res) {

   var widget = {
     sn : req.body.widgetsn,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc: req.body.widgetdesc};

   var widgetObj = new Widget(widget);

   widgetObj.save(function(err, data) {
      if (err) {
         res.send(err);
     } else {
       console.log(data);
       res.render('widgets/added', {title: 'Widget Added', widget: widget});
     }
   });
};

// вывод виджета
exports.show = function(req, res) {
   var sn = req.params.sn;
   Widget.findOne({sn : sn}, function(err, doc) {
      if (err)
         res.send('There is no widget with sn of ' + sn);
   else
      res.render('widgets/show', {title : 'Show Widget', widget : doc});
   });
};

// удаление виджета
exports.destroy = function(req, res) {
   var sn = req.params.sn;
   Widget.remove({sn : sn}, function(err) {
      if (err) {
        res.send('There is no widget with sn of ' + sn);
      } else {
         console.log('deleted ' + sn);
         res.send('deleted ' + sn);
      }
   });
};
```

```
// вывод формы редактирования
exports.edit = function(req, res) {
   var sn = req.params.sn;
   Widget.findOne({sn : sn}, function(err, doc) {
      console.log(doc);
      if(err)
         res.send('There is no widget with sn of ' + sn);
      else
        res.render('widgets/edit', {title : 'Edit Widget', widget : doc});
   });
};

// обновление виджета
exports.update = function(req, res) {
  var sn = req.params.sn;
  var widget = {
     sn : req.body.widgetsn,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     desc : req.body.widgetdesc};

  Widget.update({sn : sn}, widget, function(err) {
     if (err)
       res.send('Problem occured with update' + err)
     else
        res.render('widgets/added', {title: ''Widget Edited',
                                     widget : widget})
  });
};
```

Теперь данные приложения для виджетов постоянно находятся в базе данных, а не исчезают при каждом выходе из приложения. И все приложение настроено таким образом, что мы можем добавить поддержку новых категорий данных с минимальным влиянием на неизменные компоненты приложения.

Приложение для виджетов в примерах данной главы создано в ходе работы с предыдущей главой. Это означает, что для корректной работы приложения вдобавок к MongoDB вам нужно запустить еще и Redis-сервер.

11 Node и привязки к реляционным базам данных

Для традиционных веб-приложений самым популярным средством хранения данных являются реляционные базы данных. В отличие от них Node-приложения в гораздо большей степени ориентированы не на реляционные базы данных, а на такие инструменты, как Redis и MongoDB, возможно, благодаря привлекательности этих инструментов или из-за интереса к нетрадиционным направлениям разработки.

Существует ряд модулей для реляционных баз данных, которые можно использовать в Node-приложениях, но они могут оказаться не настолько совершенными, как те, к которым вы привыкли в таких языках, как PHP и Python. На мой взгляд, Node-модули для реляционных баз данных еще не готовы для их применения в конечных продуктах.

Тем не менее у модулей, поддерживающих реляционные базы данных, есть одна положительная сторона: они предельно просты в использовании. В этой главе я собираюсь показать два различных подхода, позволяющих интегрировать реляционную базу данных MySQL в Node-приложение. В первом подходе используется модуль mysql (node-mysql) — популярный MySQL-клиент, написанный JavaScript, а во втором написанный на C++ модуль db-mysql, являющийся частью новой инициативы node-db по созданию общей платформы для управления базами данных из Node-приложений.

Ни один из упомянутых модулей на данный момент транзакций не поддерживает, хотя технология mysql-series позволяет добавить данную функциональность к модулю node-mysql. Вам будет предоставлена краткая демонстрация этого, кроме того, вы познакомитесь с Sequelize — библиотекой ORM (Object-Relational Mapping — объектно-реляционное отображение), работающей с MySQL.

Существует множество реляционных баз данных, включая SQL Server, Oracle и SQLite, однако я остановился на MySQL, поскольку эту базу данных можно устанавливать в средах Windows и Unix, она бесплатна для некоммерческого использования и часто применяется с такими приложениями, которые знакомы

большинству из нас. К тому же эта реляционная база данных в наибольшей степени поддерживается в Node.

Тестовая база данных, используемая в данной главе, называется nodetest2 и в ней содержится одна таблица со следующей структурой:

```
id - int(11), primary key, not null, autoincrement
title - varchar(255), unique key, not null
text - text, nulls allowed
created - datetime, nulls allowed
```

Начало работы с db-mysql

Node-модуль db-mysql, являющийся «родным» для Node, требует установки на вашей системе библиотек MySQL-клиента. Инструкции по их установке и настройке можно найти по адресу http://nodejsdb.org/db-mysql/.

После настройки вашей среды db-mysql можно установить с помощью диспетчера Node-пакетов:

```
npm install db-mysql
```

Модуль db-mysql предоставляет два класса для взаимодействия с базой данных MySQL. Первый из них, класс `database`, служит для соединения (`connect`) с базой данных и отсоединения (`disconnect`) от нее, а также для отправки запросов. Класс `query` — это то, что возвращает метод `query` базы данных. Класс `query` можно использовать для создания запроса либо через выстроенные в цепочку методы, представляющие каждый компонент запроса, либо непосредственно с посредством строки запроса; модуль db-mysql является весьма гибким средством.

Результаты, включая любую ошибку, передаются в последней функции обратного вызова любого метода. Для выстраивания действий в цепочку вы можете задействовать вложенные обратные вызовы или обрабатывать событие `EventEmitter`, чтобы контролировать как ошибки, так и результаты выполнения команд базы данных.

При установке соединения с базой данных MySQL вы можете передать ряд параметров, влияющих на созданную базу данных. Вам придется предоставить, как минимум, имя хоста, порт или сокет, а также имя пользователя, пароль и имя базы данных:

```
var db = new mysql.Database({
   hostname: 'localhost',
   user: 'username',
   password: 'userpass',
   database: 'databasenm'
});
```

Параметры подробно описаны в документации по db-mysql, а также в документации по MySQL.

Использование строки запроса или выстроенных в цепочку методов

Чтобы показать гибкость db-mysql, приложение в листинге 11.1 подключается к базе данных и дважды запускает один и тот же запрос: сначала используя выстроенные в цепочку методы класса query, а затем строку запроса. В первом запросе результат обрабатывается во вложенной функции обратного вызова, а во втором прослушиваются события успешного выполнения и ошибки, после чего дается соответствующий ответ. В обоих случаях результат возвращается в объекте **rows**, который возвращает массив объектов, представляющих каждую строку данных.

Листинг 11.1. Демонстрация гибкости db-mysql в двух стилях запроса

```
var mysql = require('db-mysql');

// определение подключения к базе данных
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'userpass',
    database: 'databasenm'
});

// подключение
db.connect();

db.on('error', function(error) {
  console.log("CONNECTION ERROR: " + error);
});

// база данных подключена
db.on('ready', function(server) {

  // запрос, использующий выстроенные в цепочку методы
  // и вложенную функцию обратного вызова
  this.query()
      .select('*')
      .from('nodetest2')
      .where('id = 1')
      .execute(function(error, rows, columns) {
        if (error) {
            return console.log('ERROR: ' + error);
        }
        console.log(rows);
        console.log(columns);
  });

  // запрос, использующий непосредственную строку запроса и событие
  var qry = this.query();
```

```
  qry.execute('select * from nodetest2 where id = 1');

  qry.on('success', function(rows, columns) {
    console.log(rows);    // вывод возвращенных строк
    console.log(columns); // вывод возвращенных столбцов
  });
  qry.on('error', function(error) {
    console.log('ERROR: ' + error);
  });
});
```

Сразу после подключения базы объект `database` выдает событие `ready` или `error`, если с подключением возникли проблемы. Объект `server`, который передается в качестве параметра функции обратного вызова, инициируемой по событию `ready`, содержит следующие свойства:

`hostname`

Имя хоста базы данных.

`user`

Имя пользователя для подключения с базой данных.

`database`

Имя базы данных, к которой осуществляется подключение.

`version`

Версия программного обеспечения сервера.

В первом запросе примера используются выстроенные в цепочку методы класса `query`, формирующие каждый компонент запроса. Для SQL-запроса доступны следующие методы:

`select`

Критерий выбора, относящийся к запросу, например список имен столбцов, символ звездочки (`*`) для выбора всех столбцов или строка выбора.

`from`

Массив имен таблиц или строка, используемая в инструкции `from`.

`join`

Команда объединения, состоящая из объекта `options`, в котором находится тип объединения, объединяемая таблица, псевдоним таблицы (если таковой имеется), условия объединения (если таковые имеются) и указание на то, нужно ли экранировать имена таблиц и их псевдонимы (по умолчанию имеет значение `true`).

`where`

Условная инструкция, в которой могут содержаться местозаполнители и другие выстроенные в цепочку методы, отражающие условия `and` (и) и `or` (или).

`order`

Добавляет команду `ORDER BY`.

`limit`

Добавляет команду `LIMIT`.

`add`

Добавляет обобщенную команду, например `UNION`.

Выстроенные в цепочку методы предлагают более нейтральный в отношении баз данных подход для выполнения аналогичных SQL-инструкций. На данный момент драйверы базы данных Node.js поддерживают MySQL (db-mysql) и Drizzle (db-drizzle). Выстроенные в цепочку методы справляются со всеми вариациями между этими двумя базами данных. Кроме того, выстроенные в цепочку методы автоматически обрабатывают любое экранирование данных в SQL-инструкции, необходимое для ее безопасного использования. В случае же непосредственного запроса для правильного экранирования SQL вам придется задействовать метод `query.escape`.

Объект `query` при успешном выполнении запроса выдает событие `success`, в противном случае — событие `error`. Кроме того, он выдает событие `each` для каждой строки, возвращенной запросом. Если событие `success` выдается для запроса, возвращающего строки, функция обратного вызова получает как объект `rows`, так и объект `columns`. Каждая строка является массивом, в котором каждый элемент содержит объект, состоящий из пар имя-значение столбца. Объект `columns` представляет столбцы, являющиеся частью результата, и каждый объект столбца содержит имя (`name`) и тип (`type`) столбца. Пусть тестовая таблица в примере имеет столбцы `id`, `title`, `text`, тогда объект `rows` мог бы выглядеть так:

```
{ id: 1,
    title: 'this is a nice title',
    text: 'this is a nice text',
    created: Mon, 16 Aug 2010 09:00:23 GMT }
```

А объект `columns` мог бы иметь следующий вид:

```
[ { name: 'id', type: 2 },
  { name: 'title', type: 0 },
  { name: 'text', type: 1 },
  { name: 'created', type: 6 } ]
```

Если событие `success` относится к запросу на обновление, удаление или вставку, функция обратного вызова, инициируемая событием `success`, получает в качестве параметра объект `result`. Более подробно этот объект рассматривается в следующем разделе.

Хотя каждый из запросов обрабатывается по-разному, все они должны быть реализованы в функции обратного вызова, инициируемой событием `success` базы данных. Поскольку db-mysql относится к Node-функциональности, методы являются асинхронными. Если попытаться выполнить один из запросов вне функции

обратного вызова соединения с базой данных, ничего не получится, поскольку вне функции обратного вызова соединение с базой данных установить невозможно.

Обновление базы данных с помощью непосредственных запросов

Как уже упоминалось, модуль db-mysql предоставляет два различных способа обновления данных в реляционной базе данных: непосредственный запрос или выстроенные в цепочку методы. Сначала мы рассмотрим использование непосредственного запроса.

В непосредственном запросе можно задействовать те же конструкции языка SQL, что и в MySQL-клиенте:

```
qry.execute('update nodetest2 set title =
  "This is a better title" where id = 1');
```

Можно также указывать местозаполнители:

```
qry.execute('update nodetest2 set title = ? where id = ?',
  ["This was a better title", 1]);
```

Местозаполнители могут использоваться либо со строкой непосредственного запроса, либо с выстроенными в цепочку методами. Местозаполнители позволяют заблаговременно создать строку запроса, а затем просто передать необходимые ей значения. Местозаполнители представлены в строке знаками вопроса (?), и каждое значение передается методу в виде элемента массива во втором параметре.

Результат выполненной в отношении базы данных операции отражается в параметре, возвращаемом в функции обратного вызова для события `success`. В листинге 11.2 в тестовую базу данных вставляется новая строка. Обратите внимание на использование MySQL-функции `NOW` для установки текущих даты и времени для создаваемого поля. При использовании MySQL-функции ее нужно помещать непосредственно в строку запроса, местозаполнитель здесь применять нельзя.

Листинг 11.2. Использование местозаполнителей в строке запроса

```
var mysql = require('db-mysql');

// определение подключения к базе данных
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'userpass',
    database: 'databasenm'
});

// подключение
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
```

продолжение ⇩

Листинг 11.2 *(продолжение)*

```
});

// база данных подключена
db.on('ready', function(server) {

  // запрос, использующий непосредственную строку запроса и событие
  var qry = this.query();
  qry.execute(
    'insert into nodetest2 (title, text, created) values(?,?,NOW())',
    ['Third Entry','Third entry in series']);

  qry.on('success', function(result) {
    console.log(result);
  });

  qry.on('error', function(error) {
    console.log('ERROR: ' + error);
  });
});
```

Если операция проходит успешно, в качестве параметра функции обратного вызова возвращается следующий результат:

```
{ id: 3, affected: 1, warning: 0 }
```

Здесь `id` — это сгенерированный идентификатор строки таблицы; свойство `affected` показывает количество измененных строк (1), а свойство `warning` — количество предупреждений, сгенерированных запросом для строк (в данном случае — 0).

Обновление и удаление строк таблицы базы данных обрабатываются схожим образом: либо с помощью точно такого же синтаксиса, как и в MySQL-клиенте, либо с помощью местозаполнителей. В листинге 11.3 к тестовой базе данных добавляется новая запись, обновляется поле `title` (заголовок), а затем та же самая запись удаляется. Как видите, для каждого запроса я создал свой объект `query`. Хотя один и тот же запрос можно запустить несколько раз, у каждого запроса имеются собственные аргументы, включая количество аргументов, ожидаемое при каждом выполнении запроса. В запросе на вставку я использовал четыре заменяемых значения, но если я попытаюсь в запросе на обновление задействовать только два из них, произойдет ошибка. В приложении вместо перехвата событий используются вложенные функции обратного вызова.

Листинг 11.3. Вставка, обновление и удаление записи с использованием вложенных функций обратного вызова

```
var mysql = require('db-mysql');

// определение подключения к базе данных
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
```

```
    password: 'password',
    database: 'databasenm'
});

// подключение
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
});

// база данных подключена
db.on('ready', function(server) {

  // запрос, использующий непосредственную строку запроса
  // и вложенные функции обратного вызова
  var qry = this.query();

  qry.execute(
    'insert into nodetest2 (title, text,created) values(?,?,NOW())',
    ['Fourth Entry','Fourth entry in series'], function(err,result) {
      if (err) {
        console.log(err);
      } else {
         console.log(result);

         var qry2 = db.query();
         qry2.execute('update nodetest2 set title = ? where id = ?',
                   ['Better title',4], function(err,result) {
            if(err) {
               console.log(err);
            } else {
               console.log(result);
               var qry3 = db.query();
               qry3.execute('delete from nodetest2 where id = ?',[4],
                     function(err, result) {
                  if(err) {
                     console.log(err);
                  } else {
                     console.log(result);
                  }
               });
            }
         });
      }
    });
});
```

Как можно заметить, в этом примере отсутствует возможность отката ранее выполненных SQL-инструкций в случае ошибки в одной из них. На данный момент

управление транзакциями в db-mysql отсутствует. Чтобы обеспечить целостность базы данных, реализовать управление транзакциями в вашем приложении следует самостоятельно. Это можно сделать, реализовав проверку на наличие ошибки после выполнения каждой SQL-инструкции с последующим откатом предыдущей успешной операции (или нескольких операций) в случае ошибки. Эту ситуацию нельзя признать идеальной, поэтому вам следует постоянно проявлять осторожность при выполнении операций.

Нечто вроде поддержки транзакций реализовано в модуле mysql-queues, краткое описание которого есть в этой главе далее.

Обновление базы данных с помощью выстроенных в цепочку методов

Методы модуля db-mysql для вставки, обновления и удаления записи называются соответственно `insert`, `update` и `delete`. Методы `update` и `delete` могут воспользоваться методом `where`, который, в свою очередь, задействует условные выстраиваемые в цепочку методы `and` и `or`. Метод `update` может также использовать другой выстраиваемый в цепочку метод `set` для установки значений SQL-инструкции.

В листинге 11.4 воспроизводится функциональность листинга 11.3, но для вставки и обновления применяются выстроенные в цепочку методы. Для удаления выстроенные в цепочку методы не используются, поскольку на момент написания данной книги метод удаления работал неправильно.

Листинг 11.4. Использование выстроенных в цепочку методов для вставки новой записи с ее последующим обновлением

```
var mysql = require('db-mysql');

// определение подключения к базе данных
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'password',
    database: 'databasenm'
});

// подключение
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
});

// база данных подключена
```

```
db.on('ready', function(server) {

  // запрос, использующий выстроенные в цепочку методы
  // и вложенные функции обратного вызова
  var qry = this.query();
  qry.insert('nodetest2',['title','text','created'],
            ['Fourth Entry', 'Fourth entry in series', 'NOW()'])
     .execute(function(err,result) {
       if (err) {
        console.log(err);
       } else {
         console.log(result);

         var qry2 = db.query();
         qry2.update('nodetest2')
             .set({title: 'Better title'})
             .where('id = ?',[4])
             .execute(function(err, result) {
            if(err) {
               console.log(err);
            } else {
               console.log(result);
            }
         });
      }
    });
});
```

Я не являюсь страстным приверженцем выстроенных в цепочку методов, хотя, думаю, что они удобны при передаче данных из приложения или для такого приложения, которое может поддерживать несколько баз данных.

Собственный JavaScript-доступ к MySQL с помощью модуля node-mysql

В отличие от db-mysql для node-mysql устанавливать специализированное программное обеспечение MySQL-клиента не нужно. Следует лишь установить модуль, и все будет готово к работе:

```
npm install mysql
```

Собственный драйвер весьма прост в использовании. Вы создаете клиентское соединение с базой данных MySQL, выбираете базу данных и пользуетесь выбранным клиентом для всех операций с базой данных посредством метода `query`. В качестве последнего параметра методу `query` может быть передана функция обратного вызова, а также предоставлена информация, имеющая отношение к последней операции. Если функция обратного вызова не используется, для определения момента завершения обработки можно слушать события.

Выполнение основных CRUD-операций с помощью node-mysql

Как уже отмечалось, прикладной программный интерфейс модуля node-mysql предельно прост: создается клиент, осуществляется настройка на базу данных и в виде клиентских запросов отправляются SQL-инструкции. Функции обратного вызова применять не обязательно, а поддержка событий минимальна. При использовании функций обратного вызова их параметрами обычно являются ошибка и результат, хотя в случае отправки запроса `SELECT` функция обратного вызова также имеет параметр `fields` (поля).

В листинге 11.5 показано, как использовать модуль node-mysql для соединения с базой данных виджетов, создания новой записи, ее обновления и удаления. При всей своей простоте этот пример иллюстрирует всю функциональность, поддерживаемую модулем node-mysql.

Листинг 11.5. Демонстрация CRUD-операций, выполняемых с помощью node-mysql

```
var mysql = require('mysql');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });

client.query('USE databasenm');

// создание
client.query('INSERT INTO nodetest2 ' +
   'SET title = ?, text = ?, created = NOW()',
   ['A seventh item', 'This is a seventh item'], function(err, result) {
   if (err) {
     console.log(err);
   } else {
     var id = result.insertId;
     console.log(result.insertId);

     // обновление
     client.query('UPDATE nodetest2 SET ' +
         'title = ? WHERE ID = ?', ['New title', id], function (err, result) {
        if (err) {
          console.log(err);
        } else {
          console.log(result.affectedRows);

          // удаление
          client.query('DELETE FROM nodetest2 WHERE id = ?',
            [id], function(err, result) {
            if(err) {
               console.log(err);
            } else {
```

```
            console.log(result.affectedRows);

            // использование вместо вложенной функции обратного вызова
            // именованной функции
            getData();
          }
        });
      }
    });
  }
});

// извлечение данных
function getData() {
  client.query(
      'SELECT * FROM nodetest2 ORDER BY id', function(err, result,fields) {
    if(err) {
      console.log(err);
    } else {
      console.log(result);
      console.log(fields);
    }
    client.end();
  });
}
```

Результат запроса не противоречит нашим ожиданиям: это массив объектов, каждый из которых представляет одну строку таблицы. Следующие данные являются примером вывода первой возвращенной строки:

```
[ { id: 1,
    title: 'This was a better title',
    text: 'this is a nice text',
    created: Mon, 16 Aug 2010 15:00:23 GMT },
 ... ]
```

Параметр `fields` также отвечает нашим ожиданиям, хотя формат может отличаться от формата, выдаваемого другими модулями. Вместо массива объектов возвращается объект, где каждое поле таблицы является свойством объекта, а его значение — объектом, представляющим информацию об этом поле. Я не буду дублировать весь вывод, а следующие данные возвращаются для первого поля с именем `id`:

```
{ id:
   { length: 53,
     received: 53,
     number: 2,
     type: 4,
     catalog: 'def',
     db: 'nodetest2',
     table: 'nodetest2',
     originalTable: 'nodetest2',
```

продолжение ⇩

```
    name: 'id',
    originalName: 'id',
    charsetNumber: 63,
    fieldLength: 11,
    fieldType: 3,
    flags: 16899,
    decimals: 0 }, ...
```

Модуль не поддерживает объединение нескольких SQL-инструкций и не поддерживает транзакции. Единственным способом несколько приблизиться к поддержке транзакции является использование рассматриваемого далее модуля mysql-queues.

Поддержка MySQL-транзакций с помощью mysql-queues

Модуль mysql-queues представляет собой оболочку для node-mysql и обеспечивает поддержку как нескольких запросов к базе данных, так и транзакций. Порядок его использования может показаться немного странным, особенно из-за того, что он поддерживает асинхронный режим без каких-либо внешних проявлений.

Обычно убедиться в завершении асинхронной функции позволяют вложенные функции обратного вызова, именованные функции или такие модули, как Async. В то же время в листинге 11.6 модуль mysql-queues реализует управляющую логику, гарантируя посредством *очереди* завершение выстроенных в очередь SQL-инструкций перед обработкой последней инструкции `SELECT`. SQL-инструкции выполняются по порядку: вставка, обновление и, наконец, последнее извлечение.

Листинг 11.6. Реализация управляющей логики SQL-инструкций с помощью очереди

```
var mysql = require('mysql');
var queues = require('mysql-queues');

// подключение к базе данных
var client = mysql.createClient({
   user: 'username',
   password: 'password'
   });

client.query('USE databasenm');

// связывание очередей с запросом
// использование отладки
queues(client, true);

// создание очереди
q = client.createQueue();

// вставка
q.query('INSERT INTO nodetest2 (title, text, created) ' +
          'values(?,?,NOW())',
          ['Title for 8', 'Text for 8']);
```

```
// обновление
q.query('UPDATE nodetest2 SET title = ? WHERE title = ?',
            ['New Title for 8','Title for 8']);

q.execute();

// выборки не произойдет, пока не завершатся предыдущие запросы
client.query(
  'SELECT * FROM nodetest2 ORDER BY ID', function(err, result, fields) {
    if (err) {
      console.log(err);
    } else {

      // будут показаны все записи, включая самые новые
      console.log(result);
      client.end();
    }
});
```

Если добиться поддержки транзакций, нужно инициировать не очередь, а транзакцию. Для этого достаточно вызвать метод `rollback` при возникновении ошибки, а метод `commit` при завершении транзакции. Кроме того, при вызове для транзакции метода `execute` любые запросы, следующие за вызовом метода, выстраиваются в очередь до завершения транзакции. В листинге 11.7 представлено то же приложение, что и в листинге 11.6, но теперь использующее транзакцию.

Листинг 11.7. Использование транзакции для ужесточения контроля над SQL-обновлениями

```
var mysql = require('mysql');
var queues = require('mysql-queues');

// подключение к базе данных
var client = mysql.createClient({
   user: 'username',
   password: 'password'
   });

client.query('USE databasenm');

// связывание очередей с запросом
// использование отладки
queues(client, true);

// создание транзакции
var trans = client.startTransaction();

// вставка
trans.query('INSERT INTO nodetest2 (title, text, created) ' +
         'values(?,?,NOW())',
```

продолжение ⇩

Листинг 11.7 *(продолжение)*

```
          ['Title for 8', 'Text for 8'], function(err,info) {
    if (err) {
       trans.rollback();
    } else {
      console.log(info);

      // обновление
      trans.query('UPDATE nodetest2 SET title = ? WHERE title = ?',
            ['Better Title for 8','Title for 8'], function(err,info) {
        if(err) {
           trans.rollback();
        } else {
           console.log(info);
           trans.commit();
        }
       });
     }
});
trans.execute();

// выборки не произойдет, пока не завершатся предыдущие запросы
client.query(
  'SELECT * FROM nodetest2 ORDER BY ID', function(err, result, fields) {
    if (err) {
      console.log(err);
    } else {

      // будут показаны все записи, включая самые новые
      console.log(result);
      client.end();
    }
});
```

Модуль mysql-queues добавляет к модулю node-mysql два важных компонента:

- Поддержка нескольких запросов без необходимости использования вложенных обратных вызовов.
- Поддержка транзакций.

Если вы собираетесь задействовать модуль node-mysql, я настоятельно рекомендую вам воспользоваться также модулем mysql-queues.

Поддержка ORM с помощью Sequelize

Хотя модули из предыдущего раздела обеспечивают привязку к базе данных MySQL, высокоуровневая абстракция им недоступна. Именно такую абстракцию в отношении ORM обеспечивает модуль Sequelize, хотя в настоящее время он не поддерживает транзакции.

Определение модели

Чтобы использовать модуль Sequelize, нужно определить модель, представляющую собой отображение таблицы (или таблиц) базы данных на JavaScript-объекты. В наших предыдущих примерах мы работали с простой таблицей `nodetest2` со следующей структурой:

```
id - int(11), primary key, not null
title - varchar(255), unique key, not null
text - text, nulls allowed,
created - datetime, nulls allowed
```

Модель для этой таблицы создается с помощью соответствующей базы данных и флагов для каждого поля:

```
// определение модели
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   created : Sequelize.DATE
   });
```

Поддерживаемые типы данных и варианты их отображения:

- `Sequelize.STRING` => `VARCHAR(255)`;
- `Sequelize.TEXT` => `TEXT`;
- `Sequelize.INTEGER` => `INTEGER`;
- `Sequelize.DATE` => `DATETIME`;
- `Sequelize.FLOAT` => `FLOAT`;
- `Sequelize.BOOLEAN` => `TINYINT(1)`.

Для дальнейшего уточнения значений полей можно использовать следующие параметры:

`type`

: Тип данных поля.

`allowNull`

: Значение параметра `false` допускает наличие значений `null`; по умолчанию параметр имеет значение `true`.

`unique`

: Значение `true` не позволяет иметь повторяющиеся значения; по умолчанию параметр имеет значение `false`.

`primaryKey`

: Значение `true` задает первичный ключ.

`autoIncrement`

: Значение `true` приводит к автоматическому приращению значения поля.

Вероятнее всего, как приложение, так и база данных у вас новые, поэтому для создания таблицы базы данных после определения модели нужно синхронизировать эту модель с базой данных:

```
// синхронизация
Nodetest2.sync().error(function(err) {
   console.log(err);
});
```

После синхронизации и проверки таблицы в базе данных вы обнаружите, что таблица и модель отличаются друг от друга из-за тех изменений, которые модуль Sequelize внес в таблицу. Во-первых, таблица теперь называется `nodetest2s`, а во-вторых, в ней имеется два новых поля:

```
id - int(11), primary key, autoincrement
title - varchar(255), unique key, nulls not allowed
text - text, nulls allowed
created - datetime, nulls allowed
createdAt - datetime, nulls not allowed
updatedAt - datetime, nulls not allowed
```

Эти изменения вносятся модулем Sequelize, и способов воспрепятствовать этому не существует. Чтобы база данных соответствовала вашим ожиданиям, ее придется соответствующим образом скорректировать. Для начала можно отказаться от столбца `created`, поскольку он вам больше не нужен. С помощью модуля Sequelize это делается путем удаления поля из класса и повторной синхронизации:

```
// определение модели
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });

// синхронизация
Nodetest2.sync().error(function(err) {
   console.log(err);
});
```

Теперь у вас есть JavaScript-объект, представляющий модель, которая также отображается на таблицу реляционной базы данных. Далее вам понадобится добавить к этой таблице некоторые данные.

Использование CRUD-операций в ORM-стиле

Различия между привязкой базы данных MySQL и ORM остаются. В случае ORM вы не вставляете строку в базу данных, а создаете новый экземпляр объекта и сохраняете этот экземпляр. То же самое относится к обновлению: оно выполняется не средствами SQL, а либо путем непосредственного изменения значения свойства, либо вызовом метода `updateAttributes` и передачей ему объекта с измененными

свойствами. Кроме того, вы не удаляете строку из базы данных, а получаете доступ к экземпляру объекта и затем удаляете этот экземпляр.

Чтобы показать, как это все вместе работает, в листинге 11.8 создается модель, далее модель синхронизируется с базой данных (если таблицы в базе нет, она создается), затем создается новый экземпляр, который сохраняется. После создания нового экземпляра он обновляется дважды. Все объекты извлекаются, и перед тем, как только что добавленный экземпляр объекта удаляется, контент выводится на экран.

Листинг 11.8. Выполнение CRUD-операций с использованием модуля Sequelize

```
var Sequelize = require('sequelize');

var sequelize = new Sequelize('databasenm',
                'username', 'password',
                { logging: false});

// определение модели
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });

// синхронизация
Nodetest2.sync().error(function(err) {
   console.log(err);
});
var test = Nodetest2.build(
   { title: 'New object',
     text: 'Newest object in the data store'});

// сохранение записи
test.save().success(function() {

  // первое обновление
  Nodetest2.find({where : {title: 'New object'}}).success(function(test) {
     test.title = 'New object title';
     test.save().error(function(err) {
       console.log(err);
     });
     test.save().success(function() {

        // второе обновление
        Nodetest2.find(
            {where : {title: 'New object title'}}).success(function(test) {
           test.updateAttributes(
           {title: 'An even better title'}).success(function() {});
           test.save().success(function() {
```

продолжение ⇩

Листинг 11.8 *(продолжение)*

```
            // поиск всего
            Nodetest2.findAll().success(function(tests) {
               console.log(tests);

              // поиск и удаление нового объекта
              Nodetest2.find({ where: {title: 'An even better title'}}).
                 success(function(test) {
                 test.destroy().on('success', function(info) {
                   console.log(info);
                   });
              });
            });
          });
        })
      });
  });
});
```

При выводе результатов метода `findAll` количество получаемых обратно данных может вызвать удивление. Конечно, доступ к свойствам можно получить напрямую из возвращенного значения, сначала через элемент массива, а затем через значение:

```
tests[0].id; // возвращает идентификатор
```

Однако другие данные, связанные с этим новым объектом, демонстрируют, что вы находитесь уже совсем в другом мире, не имеющем никакого отношения к миру привязок к реляционным базам данных. Вот как выглядит пример одного из возвращенных объектов:

```
[ { attributes: [ 'id', 'title', 'text', 'createdAt', 'updatedAt' ],
    validators: {},
    __factory:
     { options: [Object],
       name: 'nodetest2',
       tableName: 'nodetest2s',
       rawAttributes: [Object],
       daoFactoryManager: [Object],
       associations: {},
       validate: {},
       autoIncrementField: 'id' },
    __options:
     { underscored: false,
       hasPrimaryKeys: false,
       timestamps: true,
       paranoid: false,
       instanceMethods: {},
       classMethods: {},
       validate: {},
       freezeTableName: false,
       id: 'INTEGER NOT NULL auto_increment PRIMARY KEY',
```

```
    title: 'VARCHAR(255) NOT NULL UNIQUE',
    text: 'TEXT',
    createdAt: 'DATETIME NOT NULL',
    updatedAt: 'DATETIME NOT NULL' },
  id: 14,
  title: 'A second object',
  text: 'second',
  createdAt: Sun, 08 Apr 2012 20:58:54 GMT,
  updatedAt: Sun, 08 Apr 2012 20:58:54 GMT,
  isNewRecord: false },...
```

Упрощенный способ добавления нескольких объектов

Асинхронная природа модуля Sequelize становится абсолютно очевидной при анализе листинга 11.8. Как правило, вопрос о вложенных функциях обратного вызова не вызывает проблем, поскольку выполнять слишком много операций в строке вам не понадобится, исключая случай добавления сразу нескольких новых экземпляров объекта. Только в этом случае вложенные функции обратного вызова могут вызвать проблемы.

К счастью, модуль Sequelize предоставляет простой способ выстраивания запросов в цепочку, чтобы вы смогли создать множество новых экземпляров объектов и одновременно их сохранить. Для этого в модуле Sequelize есть объект `chainer`, в который вы можете одно за другим добавлять задания `EventEmitter` (например, запросы), но они не выполнятся до тех пор, пока не будет вызван метод `run`. Затем результаты всех операций возвращаются либо как успешные, либо как ошибочные.

В листинге 11.9 представлен вспомогательный объект `chainer`, в который добавляются три новых экземпляра объекта. Затем запускается метод `findAll` для базы данных, в которой эти экземпляры были успешно сохранены.

Листинг 11.9. Использование объекта chainer для упрощенного добавления нескольких экземпляров объектов

```
var Sequelize = require('sequelize');

var sequelize = new Sequelize('databasenm',
                'username', 'password',
                { logging: false});

// определение модели
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });

// синхронизация
Nodetest2.sync().error(function(err) {
   console.log(err);
});
```

продолжение ⇩

Листинг 11.9 *(продолжение)*

```
var chainer = new Sequelize.Utils.QueryChainer;
chainer.add(Nodetest2.create({title: 'A second object',text: 'second'}))
       .add(Nodetest2.create({title: 'A third object', text: 'third'}));

chainer.run()
      .error(function(errors) {
         console.log(errors);
       })
      .success(function() {
         Nodetest2.findAll().success(function(tests) {
           console.log(tests);
         });
       });
```

Этот код намного проще и в нем гораздо легче разобраться. К тому же данный подход упрощает работу с пользовательским интерфейсом или с MVC-приложением.

На веб-сайте с документацией по модулю Sequelize можно найти намного больше информации, включая сведения о том, как работать с ассоциированными объектами (представляющими отношения между таблицами).

Решение проблем перехода от привязок к реляционным базам данных к модели ORM

При работе с ORM нужно иметь в виду, что в этой модели делаются определенные допущения относительно структуры данных. Одно из них заключается в том, что если объект модели, к примеру, называется `Widget`, то таблица базы данных должна называться `widgets`. Еще одно допущение касается того, что в таблице содержится информация относительно времени добавления или обновления строки. Однако оба допущения могут не поддерживаться существующей системой баз данных, преобразованной от использования прямой привязки к базе данных к использованию модели ORM.

Одна из реальных проблем модуля Sequelize касается того, что он всегда переводит имена таблиц во множественное число. То есть при определении модели для таблицы модуль будет пытаться преобразовать имя модели в имя таблицы во множественном числе. Даже когда вы сами предоставите имя таблицы, модуль Sequelize попытается преобразовать его во множественное число. Это не станет проблемой, если у вас нет таблицы базы данных, поскольку вызов `sync` создает таблицу автоматически. Проблема возникнет при наличии готовой реляционной базы данных, и эта проблема настолько серьезна, что я настоятельно рекомендую вам использовать модуль Sequelize только с абсолютно новыми приложениями.

12 Графика и HTML5-видео

Node предлагает множество возможностей для работы с несколькими графическими приложениями и библиотеками. Поскольку речь идет о серверной технологии, ваши приложения могут использовать любое графическое программное обеспечение, предназначенное для работы на стороне сервера, например ImageMagick или GD. Однако поскольку в основе Node лежит тот же самый JavaScript-движок, который функционирует в браузере Chrome, вы также можете работать с клиентскими графическими приложениями, например Canvas и WebGL.

Кроме того, в Node в определенной степени реализована поддержка аудио- и видеофайлов с помощью новых медиасредств технологии HTML5, предлагаемой всеми современными браузерами. Хотя средства для непосредственной работы с видео и аудио ограничены, мы можем обрабатывать файлы обоих типов, как это было показано в предыдущих главах. Мы также можем воспользоваться такими серверными технологиями, как FFmpeg.

Ни одна из глав, посвященных веб-графике, не может быть полноценной без хотя бы одного упоминания о PDF. На радость тем, кто пользуется PDF-документами на веб-сайтах, у нас есть очень хороший Node-модуль, генерирующий PDF-документы, а также различные полезные PDF-инструменты и PDF-библиотеки, устанавливаемые на стороне сервера.

Я не собираюсь излишне вдаваться в детали всех форм графической или мультимедийной реализации и средствами управления из среды Node. Во-первых, я не знаком со всеми этими формами, а во-вторых, некоторые виды поддержки слишком примитивны либо технологии могут расходовать слишком много ресурсов. Вместо этого я сконцентрирую ваше внимание на более стабильных технологиях, которые имеет смысл применять в Node-приложениях. Сюда можно отнести основные манипуляции с фотографиями с помощью ImageMagick, HTML5-видео, создание и использование PDF-документов, создание и вывод изображений с помощью Canvas.

Создание и использование PDF-документов

Операционные системы, версии HTML-разметки и технологии разработки могут приходить и уходить, но одно несомненно будет оставаться постоянным — PDF. Независимо от типа создаваемого приложения или службы, весьма вероятно, что вам придется предоставлять документацию в формате PDF. Как бы сказал доктор Who, *PDF — это круто*.

Для использования формата PDF в Node-приложении у нас есть два варианта. Один их них заключается в доступе с помощью дочернего процесса Node к соответствующему инструментарию операционной системы, такому как PDF Toolkit или wkhtmltopdf, непосредственно под управлением Linux. Другой вариант заключается в применении такого популярного модуля, как PDFKit. Или же всегда можно воспользоваться тем и другим.

Доступ к PDF-инструментарию путем создания дочернего процесса

Раз в мире Windows есть утилиты командной строки для работы с PDF-документами, значит, похожие утилиты есть для Linux и OS X. Но, к счастью, есть два инструмента, PDF Toolkit и wkhtmltopdf, которые могут устанавливаться и использоваться во всех трех средах.

Получение экранных снимков с помощью wkhtmltopdf

Утилита wkhtmltopdf позволяет конвертировать HTML в PDF-файл с помощью WebKit-движка визуализации. В частности, это удобное средство получения снимков веб-сайта, графики и всего остального. Хотя некоторые сайты предоставляют возможность генерировать PDF-контент, зачастую при этом удаляется вся графика. Утилита wkhtmltopdf сохраняет внешний вид страницы.

Есть устанавливаемые версии этой утилиты для OS X и Windows, также можно загрузить исходный код для создания утилиты в среде Unix. Если приложение запускается на вашем сервере, то сначала нужно выполнить настройку.

Для работы с wkhtmltopdf на моей системе (Ubuntu) мне пришлось установить библиотеки поддержки:

```
apt-get install openssl build-essential xorg libssl-dev
```

Затем мне пришлось установить инструментарий (xvfb), позволяющий wkhtmltopdf автоматически запускаться на виртуальном X-сервере (и не зависеть от X Windows):

```
apt-get install xvfb
```

Далее я создал сценарий, который назвал wkhtmltopdf.sh, чтобы заключить wkhtmltopdf в оболочку xvfb. Он содержит одну строку кода:

```
xvfb-run -a -s "-screen 0 640x480x16" wkhtmltopdf $*
```

После этого я переместил сценарий оболочки в каталог /usr/bin и изменил права доступа с помощью команды chmod a+x. После всех этих действий появилась возможность обращаться к wkhtmltopdf из моих Node-приложений.

Существует большое количество вариантов настройки утилиты wkhtmltopdf, но я собираюсь просто показать, как вызывать ее из Node-приложения. В следующей командной строке берется URL-адрес удаленной веб-страницы, а затем генерируется PDF-документ в соответствии с вариантами настройки, предлагаемыми по умолчанию (при этом используется версия сценария оболочки):

```
wkhtmltopdf.sh http://remoteweb.com/page1.html page1.pdf
```

Для реализации этого действия в Node нам нужен дочерний процесс. С целью расширяемости приложение также должно получать URL-адрес на входе и имя выходного файла. Весь код приложения показан в листинге 12.1.

Листинг 12.1. Простое Node-приложение, служащее оболочкой для wkhtmltopdf

```
var spawn = require('child_process').spawn;

// аргументы командной строки
var url = process.argv[2];
var output = process.argv[3];

if (url && output) {
   var wkhtmltopdf = spawn('wkhtmltopdf.sh', [url, output]);

  wkhtmltopdf.stdout.setEncoding('utf8');
   wkhtmltopdf.stdout.on('data', function (data) {
       console.log(data);
   });

   wkhtmltopdf.stderr.on('data', function (data) {
       console.log('stderr: ' + data);
   });

   wkhtmltopdf.on('exit', function (code) {
      console.log('child process exited with code ' + code);
   });
} else {
   console.log('You need to provide a URL and output file name');
}
```

Как правило, вам не придется использовать утилиту wkhtmltopdf в Node-приложении саму по себе, но она будет удобным дополнением к любому веб-сайту или приложению, в котором желательно предоставить способ создания стабильной PDF-версии веб-страницы.

Доступ к данным о PDF-файле с помощью PDF Toolkit

PDF Toolkit, или pdftk, предлагает функциональность для разбиения PDF-документа на части (*разбивка*) или объединения нескольких документов в один. Это

средство можно также использовать для заполнения PDF-файла, применения водяных знаков, вращения PDF-документа, применения или удаления сжатия, распаковки PDF-потока ввода-вывода с целью редактирования. Существуют пакеты установки как на Mac, так и на Windows, а также простые и понятные инструкции по установке на многих версиях Unix.

Утилита PDF Toolkit доступна через дочерний Node-процесс. В качестве примера следующий код создает дочерний процесс, который инициирует создание комментария `dump_data` утилиты PDF Toolkit с информацией о PDF-документе, например сколько страниц содержится в документе:

```
var   spawn = require('child_process').spawn;

var pdftk = spawn(
  'pdftk', [__dirname + '/pdfs/datasheet-node.pdf', 'dump_data']);

pdftk.stdout.on('data', function (data) {

  // преобразование результатов в объект
  var array = data.toString().split('\n');
  var obj = {};
  array.forEach(function(line) {
    var tmp = line.split(':');
    obj[tmp[0]] = tmp[1];
  });

  // вывод количества страниц
  console.log(obj['NumberOfPages']);
});

pdftk.stderr.on('data', function (data) {
   console.log('stderr: ' + data);
});

pdftk.on('exit', function (code) {
   console.log('child process exited with code ' + code);
});
```

Возвращаемые результаты имеют примерно следующий вид:

```
stdout: InfoKey: Creator
InfoValue: PrintServer150&#0;
InfoKey: Title
InfoValue: &#0;
InfoKey: Producer
InfoValue: Corel PDF Engine Version 15.0.0.431
InfoKey: ModDate
InfoValue: D:201109142231522
InfoKey: CreationDate
InfoValue: D:201109142231522
PdfID0: 7fbe73224e44cb152328ed693290b51a
```

```
PdfID1: 7fbe73224e44cb152328ed693290b51a
NumberOfPages: 3
```

Формат результатов можно легко преобразовать в объект, чтобы облегчить доступ к отдельным свойствам.

PDF Toolkit является достаточно тонким инструментом, и вам нужно проявлять осмотрительность, задерживая ответ на веб-запрос в ожидании завершения его работы. Чтобы показать, как обращаться к PDF Toolkit из веб-приложения и как решать вопрос ожидаемого времени задержки, причиной которого может стать графическое приложение, требующее сложных вычислений, мы создадим простой PDF-загрузчик.

Создание PDF-загрузчика и решение проблем задержки

Способность PDF Toolkit разбивать PDF-документ на части или объединять несколько PDF-документов в один может пригодиться на веб-сайте, который позволяет пользователям выкладывать и загружать PDF-документы, предоставляя затем индивидуальный доступ к каждой PDF-странице. Вспомним Google Docs, или такой веб-сайт, как Scribd, который разрешает совместное использование PDF-документов.

Компонентами приложения такого типа являются:

- Форма для выбора выкладываемого PDF-документа.
- Веб-служба, получающая PDF-документ, а затем инициирующая PDF-обработку.
- Дочерний процесс, в котором запускается PDF Toolkit для разбиения PDF-документа на отдельные страницы.
- Ответ пользователю, предоставляющий ссылку на выложенный документ и доступ к отдельным страницам.

Компонент, разбивающий PDF-документ, должен сначала создать место для страниц, а затем определить, как они будут названы, и только затем приступать к разбиению. Это потребует доступа к Node-модулю File System с целью создания каталога для файлов разбиваемого документа. Поскольку обработка больших файлов может занять довольно много времени и ответа, посылаемого по окончании работы PDF Toolkit, ждать не стоит, приложение отправляет пользователю сообщение по электронной почте с URL-адресами только что выложенных файлов. Это требует использования модуля Emailjs, который в предыдущих главах еще не упоминался. Модуль Emailjs предлагает базовую функциональность электронной почты.

Модуль Emailjs можно установить с помощью диспетчера Node-пакетов:

```
npm install emailjs
```

Форма для выкладывания PDF-документов проста по конструкции и не требует пространных объяснений. В ней используется поле для входного файла, а также поле для имени и электронного адреса выкладывающего этот файл человека, средства задания метода `POST` и действия, выполняемого веб-службой. Поскольку мы выкладываем файл, поле `enctype` должно иметь значение `multipart/form-data`. Окончательный вариант страницы формы показан в листинге 12.2.

Листинг 12.2. Форма, позволяющая выложить PDF-файл на сервер

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title>Upload PDF</title>
 <script>
   window.onload=function() {
      document.getElementById('upload').onsubmit=function() {
         document.getElementById('submit').disabled=true;
      };
   }
 </script>
</head>
<body>
<form id="upload" method="POST" action="http://localhost:8124"
enctype="multipart/form-data">
 <p><label for="username">User Name:</label>
   <input id="username" name="username" type="text" size="20" required /></p>
 <p><label for="email">Email:</label>
   <input id="email" name="email" type="text" size="20" required /></p>
 <p><label for="pdffile">PDF File:</label>
   <input type="file" name="pdffile" id="pdffile" required /></p>
 <p>
 <p>
 <input type="submit" name="submit" id="submit" value="Submit"/>
 </p>
</form>
</body>
```

У нас есть возможность укрепить свои навыки JavaScript-разработки на стороне клиента, заблокировав кнопку отправки после отправки формы. В форме используется HTML5-атрибут `required` (обязательное поле), гарантирующий предоставление нужных данных.

Приложение веб-службы, обрабатывающее оба запроса, на форму и на выкладывание PDF-документа, использует связующий модуль Connect, но на этот раз без платформы Express.

В этой службе статический связующий модуль Connect служит для обработки статических файлов, а связующий модуль `directory` — для улучшения внешнего вида вывода каталога при обращении к нему. Кроме того, функциональность процесса должна обеспечивать синтаксический разбор PDF-файла и формы выкладывания файла. В приложении используется Connect-метод `bodyParser`, способный обрабатывать любой тип отправленных данных:

```
connect()
        .use(connect.bodyParser({uploadDir: __dirname + '/pdfs'}))
        .use(connect.static(__dirname + '/public'))
        .use(connect.directory(__dirname + '/public'))
        .listen(8124);
```

Затем данные становятся доступны нестандартной связующей программе upload, которая обрабатывает как данные, так и PDF-документ, привлекая нестандартный модуль к обработке PDF-файла. Связующий модуль bodyParser делает имя пользователя и адрес электронной почты доступными объекту request.body и выкладывает файл в объект request.files. Если файл загружен, то он загружен как объект по имени pdffile, поскольку это имя указывается в поле выкладываемого файла. Нужна еще дополнительная проверка типа файла, чтобы убедиться, что загруженный файл имеет формат PDF.

В листинге 12.3 показан весь код приложения PDF-службы.

Листинг 12.3. Приложение веб-службы по выкладыванию PDF-файлов

```
var connect = require('connect');
var pdfprocess = require('./pdfprocess');

// если используется метод POST
// загрузка файла, запуск PDF-разбиения, ответ с подтверждением
function upload(req, res, next){
  if ('POST' != req.method) return next();

  res.setHeader('Content-Type', 'text/html');
  if (req.files.pdffile && req.files.pdffile.type === 'application/pdf') {
  res.write('<p>Thanks ' + req.body.username +
      ' for uploading ' + req.files.pdffile.name + '</p>');
  res.end("<p>You'll receive an email with file links when processed.</p>");

     // обработка полученного материала
     pdfprocess.processFile(req.body.username, req.body.email,
                            req.files.pdffile.path, req.files.pdffile.name);
   } else {
     res.end('The file you uploaded was not a PDF');
   }
}
// в следующем порядке:
// статические файлы
// выкладывание файла с помощью метода POST
// в противном случае, вывод содержимого каталога
connect()
         .use(connect.bodyParser({uploadDir: __dirname + '/pdfs'}))
         .use(connect.static(__dirname + '/public'))
         .use(upload)
         .use(connect.directory(__dirname + '/public'))
         .listen(8124);

console.log('Server started on port 8124');
```

Используемый в приложении нестандартный модуль pdfprocess обрабатывает PDF-файл и при этом выполняет следующие действия:

1. Создает в общем подкаталоге pdfs каталог для пользователя, если таковой не существует.

2. Использует значение даты-времени создания файла для создания уникального имени для текущего загружаемого PDF-файла.
3. Использует значение даты-времени вместе с именем PDF-файла для создания нового подкаталога для PDF-файлов в подкаталоге пользователя.
4. Перемещает PDF-файл из временного каталога загрузки в этот новый каталог и изменяет исходное имя PDF-файла.
5. Выполняет с помощью PDF Toolkit операцию разбиения этого файла, и все отдельные PDF-файлы помещает в каталог pdfs.
6. Отправляет пользователю электронное сообщение с URL-адресом для доступа к новому каталогу, содержащему исходный выложенный PDF-файл и отдельные PDF-страницы.

Работу с файловой системой поддерживает Node-модуль File System, с электронной почтой — модуль Emailjs, а PDF Toolkit управляется в дочернем процессе. Этот дочерний процесс не возвращает никаких данных, поэтому требуется только отслеживать события выхода из дочернего процесса и ошибки. Код для этой завершающей части приложения представлен в листинге 12.4.

Листинг 12.4. Модуль обработки PDF-файла и отправки пользователю электронного сообщения с указанием того места, где находятся обработанные файлы

```
var fs = require('fs');
var spawn = require('child_process').spawn;
var emailjs = require('emailjs');

module.exports.processFile = function(username, email, path, filename) {

    // сначала создание пользовательского каталога, если его еще нет
    fs.mkdir(__dirname + '/public/users/' + username, function(err) {

        // затем создание каталога для файлов, если его еще нет
        var dt = Date.now();

        // url для отправляемого позже сообщения
        var url = 'http://examples.burningbird.net:8124/users/' +
                username + '/' + dt + filename;

        // каталог для файла
        var dir = __dirname + '/public/users/' + username + '/' +
                dt + filename;

        fs.mkdir(dir, function(err) {
          if (err)
            return console.log(err);

          // теперь переименование файла для нового места
          var newfile = dir + '/' + filename;

          fs.rename(path, newfile, function(err) {
             if (err)
                return console.log(err);
```

```
            // разбиение pdf
            var pdftk = spawn('pdftk', [newfile, 'burst', 'output',
                              dir + '/page_%02d.pdf' ]);

            pdftk.on('exit', function (code) {
               console.log('child process ended with ' + code);
               if (code != 0)
                  return;

               console.log('sending email');
               // отправка сообщения по электронной почте

               var server = emailjs.server.connect({
                   user : 'gmail.account.name',
                   password : 'gmail.account.passwod',
                   host : 'smtp.gmail.com',
                   port : 587,
                   tls : true
               });

               var headers = {
                  text : 'You can find your split PDF at ' + url,
                  from : 'youremail',
                  to : email,
                  subject: 'split pdf'
               };

               var message = emailjs.message.create(headers);

               message.attach({data:"<p>You can find your split PDF at " +
                              "<a href='" + url + "'>" + url + "</a></p>",
                              alternative: true});
               server.send(message, function(err, message) {
                   console.log(err || message);
               });
               pdftk.kill();
            });

            pdftk.stderr.on('data', function (data) {
              console.log('stderr: ' + data);
            });
         });
       });
     });
};
```

Реальный дочерний процесс, вызывающий утилиту PDF Toolkit, выделен в коде полужирным шрифтом. Используется следующий синтаксис командной строки:

```
pdftk filename.pdf burst output /home/location/page_%02d.pdf
```

Сначала задается имя файла, затем операция, а после нее инструкция вывода данных. Операцией, как уже упоминалось, является разбиение на части, когда из PDF-документа извлекаются отдельные страницы. Инструкция вывода данных предписывает PDF Toolkit поместить страницы только что разбитого PDF-документа в указанный каталог и предоставляет форматирование для имен файлов страниц — файл первой страницы будет называться page_01.pdf, второй — page_02.pdf и т. д. Я мог бы воспользоваться методом process.chdir для изменения каталога процесса, но на самом деле в этом не было никакой необходимости, поскольку я мог заставить утилиту PDF Toolkit поместить файлы в указанный каталог.

При отправке электронного сообщения применяется SMTP-сервер Gmail, который поддерживает на порту 587 протокол TLS (Transport Layer Security — безопасность транспортного уровня) с заданными в Gmail именем пользователя и паролем. Естественно, вы могли бы воспользоваться собственным SMTP-сервером. Сообщение отправляется как простым текстом, так и в формате HTML (для тех, чья программа чтения электронных сообщений способна обрабатывать формат HTML).

Конечным результатом приложения является ссылка, отправленная пользователю и ведущая его к каталогу, где он найдет выложенными PDF-файл и страницы, на который этот файл был разбит. Входящий в Connect связующий модуль directory обеспечивает достойный внешний вид контента каталога. На рис. 12.1 показаны результаты выкладывания одного очень большого PDF-файла о глобальном потеплении.

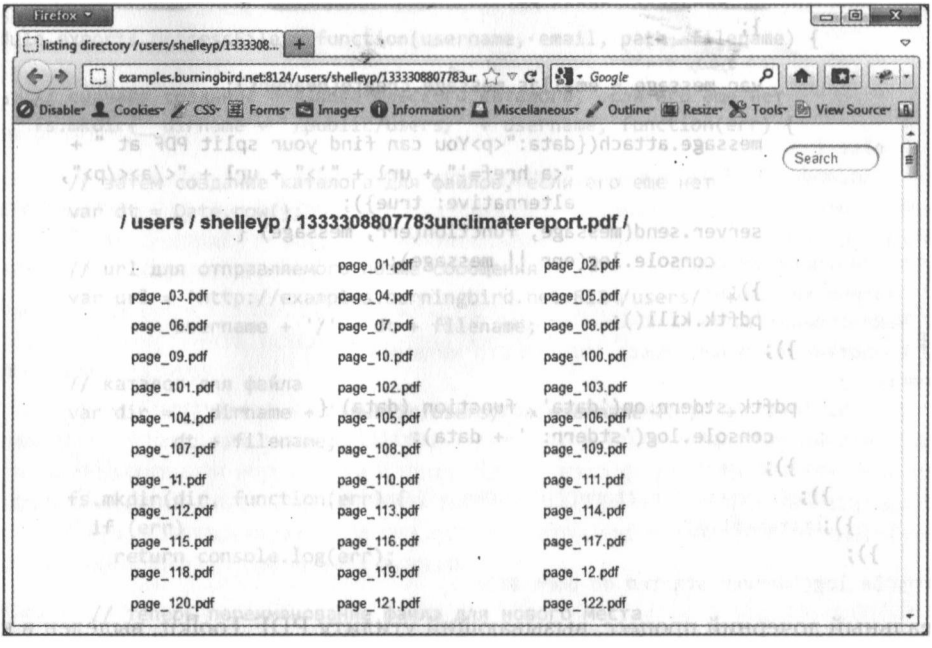


Рис. 12.1. Конечный результат разбиения с помощью утилиты PDF Toolkit большого выложенного PDF-файла

При выборе данного подхода — оповещения пользователя в электронном сообщении — пользователю не приходится ожидать окончания обработки PDF-файла (Node-служба также не зависает в ожидании).

Разумеется, пользователю все же приходится тратить время на выкладывание PDF-файла — здесь мы не касаемся вопросов, связанных с загрузкой файлов большого объема.

Создание PDF-файлов с помощью PDFKit

Если использование дочернего процесса и инструментальные средства командной строки вам не подходят или если вам требуется возможность не только создать PDF-файл, но и манипулировать существующими PDF-файлами, для этих целей существуют специальные Node-модули, основным из которых является PDFKit.

Модуль PDFKit написан на языке CoffeeScript, но для использования модуля знать CoffeeScript вам не нужно, поскольку его прикладной программный интерфейс открыт для JavaScript. Модуль предлагает функциональность для создания PDF-документа, добавления к нему страниц, объединения текста и графики, вставки изображений. Вскоре модуль должен быть дополнен и такими функциональными возможностями, как создание в PDF-файлах оглавлений, градиентов, таблиц и других полезных элементов.

Установите PDFKit с помощью диспетчера Node-пакетов:

```
npm install pdfkit
```

Запустите свое приложение, начав с создания нового PDF-документа:

```
var doc = new PDFDocument();
```

Затем можно добавить шрифт, новую веб-страницу и графику, сделав все это с помощью открытого прикладного программного интерфейса. Чтобы упростить разработку, все API-методы можно выстроить в цепочку.

Чтобы показать, как использовать модуль из JavaScript, я преобразовал CoffeeScript-код одного из примеров, предлагаемых разработчиком модуля, в JavaScript. С самого начала после создания PDF-документа к нему добавляется шрифт TrueType, размер шрифта устанавливается в 25 пикселов, а текст рисуется по координатам (x,y) со значениями 100, 100:

```
doc.font('fonts/GoodDog-webfont.ttf')
   .fontSize(25)
   .text('Some text with an embedded font!', 100, 100);
```

Затем приложение добавляет новую PDF-страницу, опять меняя размер шрифта на 25 пикселов и рисуя текст по координатам со значениями 100, 100:

```
doc.addPage()
   .fontSize(25)
   .text('Here is some vector graphics...', 100, 100);
```

Система координат документа сохраняется, а для рисования красного треугольника используются функции векторной графики:

```
doc.save()
   .moveTo(100, 150)
   .lineTo(100, 250)
   .lineTo(200, 250)
   .fill("#FF3300");
```

В следующем фрагменте кода масштаб координатной системы меняется на 0,6, меняется также начало координат, рисуется траектория в виде звезды, которая заливается красным цветом, затем для документа восстанавливаются исходная система координат и масштаб:

```
doc.scale(0.6)
   .translate(470, -380)
   .path('M 250,75 L 323,301 131,161 369,161 177,301 z')
   .fill('red', 'even-odd')
   .restore();
```

Если вам уже приходилось работать с системами векторной графики, например с Canvas, то многое здесь будет вам знакомо. Если такого опыта нет, возможно, вам сначала захочется ознакомиться с примерами использования Canvas, приводимыми далее в этой книге, а потом вернуться к данному примеру.

Далее добавляется еще одна страница, цвет заливки меняется на синий, а к странице добавляется ссылка. После этого документ записывается в файл output.pdf:

```
doc.addPage()
   .fillColor("blue")
   .text('Here is a link!', 100, 100)
   .underline(100, 100, 160, 27, {color: "#0000FF"})
   .link(100, 100, 160, 27, 'http://google.com/');

doc.write('output.pdf');
```

Создание PDF-документа вручную — слишком трудоемкий процесс. Однако мы можем без особого труда запрограммировать API модуля PDFKit на получение контента из хранилища данных и создание PDF-документа «на лету». Можно также воспользоваться PDFKit для создания по запросу PDF-документа с контентом веб-страницы или предоставить надежный снимок данных.

Однако при этом нужно знать, что многие из доступных в модуле методов не работают в асинхронном режиме. Скорее всего, при создании PDF-документа вся остальная работа окажется заблокированной, поэтому планировать ее нужно в соответствии с обстоятельствами.

Организация доступа к ImageMagick из дочернего процесса

ImageMagick является эффективным графическим инструментом командной строки, доступным в средах Mac, Windows и Unix. Он может использоваться для

кадрирования или изменения размеров изображения, доступа к метаданным, создания анимации из последовательности изображений и добавления любого количества спецэффектов. Это средство также является ресурсоемким и в зависимости от размера изображения и характера работы с ним может тратить на выполнение операции заметное время.

Рассмотрим Node-модули ImageMagick. Одним из первых является модуль imagemagick, предлагающий оболочку для функциональности ImageMagick. Однако он уже некоторое время не обновлялся. Следующим модулем является gm, предоставляющий набор предопределенных функций, работающих с ImageMagick в фоновом режиме. Тем не менее, как вы можете убедиться, работать с ImageMagick напрямую ничуть не сложнее. Для непосредственной работы с этим модулем из Node-приложения нужна лишь установленная копия ImageMagick и дочерний Node-процесс.

ImageMagick предлагает несколько разных инструментов для выполнения разных функций:

`animate`

: Анимация последовательности с помощью X-сервера.

`compare`

: Предоставление математического и визуального пояснения разницы между изображением и реконструкцией изображения после модификации.

`composite`

: Наложение двух изображений.

`conjure`

: Выполнение сценариев, написанных на языке Magick Scripting Language (MSL).

`convert`

: Преобразование изображения с использованием любого количества таких трансформаций, как кадрирование, изменение размеров или добавление эффектов.

`display`

: Вывод изображения на X-сервер.

`identify`

: Описание формата и других характеристик файла изображения или нескольких файлов изображений.

`import`

: Создание снимка любого видимого окна на X-сервере и сохранение его в файле.

`mogrify`

: Изменение изображения на месте (изменение размеров, кадрирование, размывание и т. д.) и сохранение эффектов в существующем изображении.

`montage`

Создание составного изображения из нескольких изображений.

`stream`

Потоки ввода-вывода для попиксельной отправки изображения в хранилище.

Некоторые инструменты зависят от X-сервера, поэтому применять их в Node-приложении вряд ли целесообразно. Однако такие инструменты, как `convert`, `mogrify`, `montage`, `identify` и `stream`, могут найти в Node-приложении интересное применение. В этом и в следующем разделах мы уделим внимание только одному средству — `convert`.

Тем не менее следует знать, что все, что в этом разделе касается `convert`, применимо также и к `mogrify`, за исключением того, что `mogrify` переписывает существующий файл.

Инструмент `convert` является «рабочей лошадкой» ImageMagick. С его помощью можно выполнять некоторые очень впечатляющие графические преобразования, а затем сохранить результаты в отдельном файле. Можно выполнить адаптивное размытие, повысить резкость изображения, дать изображению текстовый комментарий, расположить его на фоне другого изображения, кадрировать изображение, изменить его размеры и даже заменить каждый пиксел в изображении его цветовым дополнением. Труднее сказать, чего нельзя сделать с изображением. Разумеется, не каждая операция может вам подойти, особенно если решающую роль играет время ее выполнения. Некоторые из преобразований проходят быстро, другие могут занять довольно продолжительное время.

Чтобы показать, как использовать `convert` в Node-приложении, рассмотрим небольшое, самодостаточное приложение, показанное в листинге 12.5. Здесь в командной строке указывается имя файла изображения, которое масштабируется таким образом, чтобы поместиться в пространстве не более 150 пикселов по ширине. Это изображение также преобразуется в формат PNG независимо от типа исходного формата.

Версия этого процесса в формате командной строки имеет следующий вид:

```
convert photo.jpg -resize '150' photo.jpg.png
```

Нам нужно захватить четыре аргумента команды, передав их в массив для дочернего процесса: исходную фотографию, ключ `-resize`, значение ключа `-resize` и имя файла нового изображения.

Листинг 12.5. Node-приложение с дочерним процессом, предназначенным для масштабирования изображения с помощью ImageMagick-инструмента convert

```
var spawn = require('child_process').spawn;

// получение фотографии
var photo = process.argv[2];

// массив преобразований
```

```
var opts = [
photo,
'-resize',
'150',
photo + ".png"];

// преобразование
var im = spawn('convert', opts);

im.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

im.on('exit', function (code) {
   if (code === 0)
       console.log(
         'photo has been converted and is accessible at '+ photo + '.png');
});
```

ImageMagick-инструмент `convert` обрабатывает изображение без каких-либо видимых признаков, поэтому дочернему процессу заниматься обработкой событий не нужно. Нас интересуют только такие события, как ошибка и выход по завершении обработки изображения.

Сложности с использованием такого приложения, как ImageMagick, могут возникнуть в случае более ресурсоемкого процесса. Одним из наиболее популярных эффектов, применяемых с помощью ImageMagick к изображениям, является *эффект Polaroid*, при котором происходит небольшое вращение изображения вокруг его центра и добавление рамки и тени, чтобы изображение выглядело как фотография, сделанная камерой Polaroid. Сейчас этот эффект настолько популярен, что для него есть предопределенные варианты настройки, но прежде чем приступать к настройке, нам нужно воспользоваться примерно следующей командой (взята из примеров использования ImageMagick):

```
convert thumbnail.gif \
          -bordercolor white  -border 6 \
          -bordercolor grey60 -border 1 \
          -background  none   -rotate 6 \
          -background  black  \( +clone -shadow 60x4+4+4 \) +swap \
          -background  none   -flatten \
          polaroid.png
```

Здесь довольно много аргументов, причем в таком формате, который мог вам раньше и не встречаться. Как же их преобразовать в массив аргументов дочернего процесса?

Да очень просто.

Все, что выглядит в командной строке как единый аргумент (`\(+clone -shadow 60x4+4+4\)`), для дочернего Node-процесса таковым не является. В листинге 12.6 показан вариант преобразования из листинга 12.5, за исключением того, что теперь

вместо масштабирования изображения ему придается эффект Polaroid. Обратите особое внимание на строку, выделенную полужирным шрифтом.

Листинг 12.6. Применение эффекта Polaroid к фотографии из Node-приложения с помощью ImageMagick

```
var spawn = require('child_process').spawn;

// получение фотографии
var photo = process.argv[2];

// массив преобразований
var opts = [
photo,
"-bordercolor", "snow",
"-border", "6",
"-background","grey60",
"-background", "none",
"-rotate", "6",
"-background", "black",
"(", "+clone", "-shadow", "60x4+4+4", ")",
"+swap",
"-background", "none",
"-flatten",
photo + ".png"];

var im = spawn('convert', opts);
```

Как показано в выделенном фрагменте кода, то что фигурировало в командной строке в виде одного аргумента, для дочернего процесса превратилось в пять аргументов. Конечный результат запуска приложения показан на рис. 12.2.

Маловероятно, что вы будете использовать Node-приложение с дочерним процессом ImageMagick непосредственно из командной строки. В конце концов, можно ведь просто запустить ImageMagick напрямую. Однако вы можете использовать комбинацию из дочернего процесса и непосредственного запуска ImageMagick для выполнения нескольких различных преобразований одного и того же изображения или предоставлять такие услуги с веб-сайта (например, разрешать посетителю изменять размер фотографии для аватара или добавлять комментарии к выложенным изображениям на общем сайте ресурсов).

Ключевым в создании веб-приложений с ImageMagick является то же самое, что и в приложениях для вывода PDF-документов, описываемых в предыдущих разделах главы: если процесс может привести к замедлению работы (особенно при большом числе параллельных обращений), вам следует рассмотреть возможность предоставления такой функциональности, которая позволяет отдельным пользователям выкладывать файл изображения, а затем давать ссылку на завершенный проект (либо на сайте, либо в электронном сообщении), чтобы не блокировать работу в ожидании.

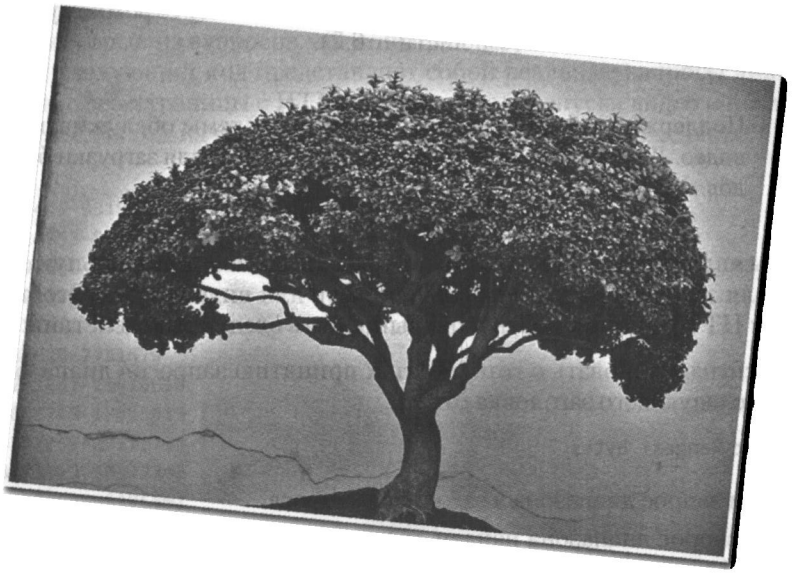


Рис. 12.2. Результат запуска Node-приложения для применения к фотографии эффекта Polaroid

Мы можем адаптировать код из листингов 12.3 и 12.4 для применения эффекта Polaroid к любому выложенному изображению. В частности, мы можем переделать код листинга 12.3 в модуль, который может применяться по тому же образцу: файловый процесс создает новый подкаталог для загруженного файла, инициируется обработка, получившиеся в результате обработки файлы помещаются в тот же каталог.

Корректное обслуживание HTML5-видео с помощью HTTP-сервера

В главе 6 мы создали простой HTTP-сервер, обслуживающий статические файлы, предоставляющий некий базовый каталог и обрабатывавший ошибку 404. Одна из веб-страниц, которые мы тестировали с сервером, содержала внедренное HTML5-видео. У веб-страницы имелась также специальная панель инструментов, позволяющая пользователю щелкать в любом месте шкалы времени, чтобы запустить видео с промежуточной позиции.

Приложение с HTML5-видео работает со статическим веб-сервером модуля Connect, но не с нашим «самодельным» веб-сервером. Причина в том, что наш сервер не обрабатывает *HTTP-диапазоны*. Такие HTTP-серверы как Apache и IIS, имеют поддержку диапазонов, то же касается модуля Connect, а у нашего статического сервера ее нет.

В этом разделе мы добавим поддержку диапазонов к тому скромному веб-серверу, который был создан в главе 6 (см. листинг 6.2).

Поддержка диапазонов выходит за рамки темы обслуживания HTML5-видео. Диапазоны могут также использоваться для загрузки больших файлов.

Диапазоны являются HTTP-заголовком, предоставляющим начальную и конечную позиции для загружаемого ресурса, например для видеофайла. Чтобы добавить поддержку HTTP-диапазонов, нужно выполнить следующие действия:

1. Просигнализировать о готовности к принятию запросов диапазона с помощью следующего заголовка ответа:

   ```
   Accept-Ranges: bytes
   ```

2. Найти запрос диапазона в заголовке запроса.
3. Если запрос диапазона найден, выделить из него значения начальной и конечной позиций.
4. Проверить, что значения начальной и конечной позиций являются числами, затем проверить, что они не превышают размера ресурса, к которому осуществляется доступ.
5. Если конечное значение не предоставлено, установить его равным размеру ресурса, а если не предоставлено начальное значение, установить его в ноль (0).
6. Создать заголовок ответа `Content-Range`, содержащий значения начальной и конечной позиций, а также размера ресурса.
7. Создать заголовок ответа `Content-Length` со значением, вычисляемым путем вычитания начального значения из конечного.
8. Изменить код состояния с 200 на 206 (частичный контент).
9. Передать объект, содержащий начальное и конечное значения, методу `createReadStream`.

Когда веб-клиент обращается к ресурсу на веб-сервере, веб-сервер может просигнализировать клиенту о поддержке диапазонов и предоставить блок диапазона со следующим заголовком:

```
Accept-Ranges: bytes
```

Первым изменением, которое необходимо внести в наш веб-сервер, является добавление нового заголовка:

```
res.setHeader('Accept-Ranges','bytes');
```

Далее клиент будет отправлять запросы диапазона со следующим форматом:

```
bytes=startnum-endnum
```

Здесь значения `startnum` и `endnum` являются начальным и конечным числами диапазона. Некоторые из запросов могут отправляться в процессе воспроизведения. Например, следующий код представляет собой реальные запросы диапазона, отправленные с веб-страницы с HTML5-видео после запуска видео и последовавшими за этим щелчками на шкале времени в процессе воспроизведения:

```
bytes=0-
bytes=7751445-53195861
bytes=18414853-53195861
bytes=15596601-18415615
bytes=29172188-53195861
bytes=39327650-53195861
bytes=4987620-7751679
bytes=17251881-18415615
bytes=17845749-18415615
bytes=24307069-29172735
bytes=33073712-39327743
bytes=52468462-53195861
bytes=35020844-39327743
bytes=42247622-52468735
```

Следующим дополнением нашего веб-сервера станет проверка факта отправки запроса диапазона, и если этот факт подтверждается, извлечение из диапазона начального и конечного значений. Код для проверки запроса диапазона имеет следующий вид:

```
if (req.headers.range) {...}
```

Для извлечения начального и конечного значений я создал функцию **processRange**, которая разбивает строку по символам дефисов (-), а затем извлекает числа из двух возвращенных строк. В функции также проводится повторная проверка, чтобы убедиться, что начальное значение имеется и является числом, которое не выходит за переделы размера файла (в противном случае возвращается код состояния 416, Requested Range Not Satisfiable — невозможно обеспечить запрошенный диапазон). В ней также проводится проверка того, что конечное значение является числом, а если оно не предоставлено, то устанавливается равным размеру файла. Функция возвращает объект, содержащий как начальное, так и конечное значения:

```
function processRange(res,ranges,len) {

  var start, end;

  // извлечение диапазона от начала до конца
  var rangearray = ranges.split('-');

  start =  parseInt(rangearray[0].substr(6));
  end = parseInt(rangearray[1]);

  if (isNaN(start)) start = 0;
  if (isNaN(end)) end = len -1;
```

продолжение ⇩

```
  // начальное значение выходит за пределы размера файла
  if (start > len - 1) {
      res.setHeader('Content-Range', 'bytes */' + len);
      res.writeHead(416);
      res.end();
  }

  // конечное значение не может выходить за пределы размера файла
  if (end > len - 1)
     end = len - 1;
  return {start:start, end:end};
}
```

Следующим этапом станет подготовка заголовка ответа `Content-Range`, в котором в следующем формате предоставляются начальное и конечное значения диапазона, а также размер ресурса:

```
Content-Range bytes 44040192-44062881/44062882
```

Подготавливается также ответ о размере контента (`Content-Length`), вычисляемом путем вычитания начального значения из конечного. Кроме того, HTTP-код состояния устанавливается в 206, что означает Partial Content — частичный контент.

И наконец, начальное и конечное значения отправляются также в качестве параметров вызову метода `createReadStream`. Тем самым гарантируется, что поток ввода-вывода корректно изменяет позицию для потоковой передачи данных.

В листинге 12.7 все эти фрагменты кода собраны вместе в модифицированном минимальном веб-сервере, который теперь может обслуживать диапазоны HTML5-видео (или других ресурсов).

Листинг 12.7. Минимальный веб-сервер с поддержкой диапазонов

```
var http = require('http'),
    url =  require('url'),
    fs   = require('fs'),
    mime = require('mime');

function processRange(res,ranges,len) {

  var start, end;

  // извлечение диапазона от начала до конца
  var rangearray = ranges.split('-');

  start =  parseInt(rangearray[0].substr(6));
  end = parseInt(rangearray[1]);

  if (isNaN(start)) start = 0;
  if (isNaN(end)) end = len -1;

  // начальное значение выходит за пределы размера файла
```

```
  if (start > len - 1) {
      res.setHeader('Content-Range', 'bytes */' + len);
      res.writeHead(416);
      res.end();
  }

  // конечное значение не может выходить за пределы размера файла
  if (end > len - 1)
     end = len - 1;

  return {start:start, end:end};
}
http.createServer(function (req, res) {

   pathname = __dirname + '/public' + req.url;

   fs.stat(pathname, function(err, stats) {
      if (err) {
        res.writeHead(404);
        res.write('Bad request 404\n');
        res.end();
      } else if (stats.isFile()) {

         var opt={};

         // предположение об отсутствии диапазона
         res.statusCode = 200;

         var len = stats.size;

         // у нас есть запрос диапазона
         if (req.headers.range) {
            opt = processRange(res,req.headers.range,len);

            // настройка размера
            len = opt.end - opt.start + 1;

            // изменение кода состояния на частичный диапазон
            res.statusCode = 206;

            // установка заголовка
            var ctstr = 'bytes ' + opt.start + '-' +
                        opt.end + '/' + stats.size;
            res.setHeader('Content-Range', ctstr);
         }

         console.log('len ' + len);
         res.setHeader('Content-Length', len);
```

продолжение ⇩

Листинг 12.7 *(продолжение)*

```
        // тип контента
        var type = mime.lookup(pathname);
        res.setHeader('Content-Type', type);
        res.setHeader('Accept-Ranges','bytes');

        // создание потока чтения и направление его в канал
        var file = fs.createReadStream(pathname,opt);
        file.on("open", function() {

           file.pipe(res);
        });
        file.on("error", function(err) {
          console.log(err);
        });

      } else {
       res.writeHead(403);
       res.write('Directory access is forbidden');
       res.end();
      }
   });
}).listen(8124);
console.log('Server running at 8124/');
```

Модификация минимального веб-сервера показала, что работа с HTTP и другой сетевой функциональностью не обязательно сложна, скорее она может быть утомительной. Ключевым подходом в таких случаях должно быть разделение каждой задачи на отдельные подзадачи с последующим добавлением кода для последовательного решения каждой подзадачи (проводя тестирование после каждого добавления).

Теперь у нас есть веб-страница, позволяющая пользователю получать корректный результат при щелчке на любом месте шкалы времени.

Создание и передача Canvas-контента

Элемент `canvas` (холст) приобрел популярность у разработчиков игр, графических дизайнеров и специалистов по статистике, поскольку позволяет создавать на веб-страницах клиентов динамическую и интерактивную графику. В среде Node элемент `canvas` поддерживается с помощью модулей, например модуля node-canvas (или просто canvas), который рассматривается в данном разделе. Модуль node-canvas основан на применении межплатформенной библиотеки векторной графики Cairo, которая уже давно пользуется популярностью у разработчиков.

Для использования node-canvas нужно установить этот модуль с помощью диспетчера Node-пакетов:

```
npm install canvas
```

Модуль node-canvas делает доступной всю стандартную Canvas-функциональность на странице клиента. Вы создаете Canvas-объект, затем контекст, рисуете в контексте все, что нужно, а затем либо выводите результат на экран либо сохраняете его в файле формата JPEG или PNG.

Имейте в виду, что определенная Canvas-функциональность, например работа с изображением, требует использования версии Cairo с номером больше чем 1.10.

На сервере доступны также два дополнительных метода, которых нет на клиентской стороне. Они позволяют передать Canvas-объект в файл (формата PNG или JPEG), надежно сохраняя результат для последующего доступа (или использования на веб-странице). Можно также преобразовать Canvas-объект в URI-идентификатор данных и включить элемент `img` в сгенерированную веб-страницу или прочитать изображение из внешнего источника (например, из файла или из базы данных Redis) и использовать его напрямую в объекте `Canvas`.

Перейдем непосредственно к демонстрации модуля node-canvas и рассмотрим листинг 12.8, в котором создается Canvas-рисунок, который затем направляется в PNG-файл для последующего доступа. В примере используется графическая фигура вращения из примера Mozilla Developer Network и к ней добавляется рамка и тень. После завершения изображение направляется в PNG-файл для последующего доступа. Основная часть функциональности может быть использована на стороне клиента, а также в Node-приложении. Единственным специфичным для Node компонентом, фигурирующем в самом конце, является код надежного сохранения графики в файле.

Листинг 12.8. Создание графики с помощью модуля node-canvas и надежное сохранение результата в PNG-файле

```
var Canvas = require('canvas');
var fs = require('fs');

  // новый canvas-объект и контекст
  var canvas = new Canvas(350,350);
  var ctx = canvas.getContext('2d');

  // создание заполненного прямоугольника с тенью
  // сохранение контекста для последующего восстановления
  ctx.save();
  ctx.shadowOffsetX = 10;
  ctx.shadowOffsetY = 10;
  ctx.shadowBlur = 5;
  ctx.shadowColor='rgba(0,0,0,0.4)';

  ctx.fillStyle = '#fff';
  ctx.fillRect(30,30,300,300);
```

продолжение ⇩

Листинг 12.8 *(продолжение)*

```
// сделано с тенью
ctx.restore();
ctx.strokeRect(30,30,300,300);

// MDN-пример: красивая графика, вставленная со смещением
// в ранее созданный прямоугольник
ctx.translate(125,125);
for (i=1;i<6;i++){
  ctx.save();
  ctx.fillStyle = 'rgb('+(51*i)+','+(255-51*i)+',255)';
  for (j=0;j<i*6;j++){
  ctx.rotate(Math.PI*2/(i*6));
  ctx.beginPath();
  ctx.arc(0,i*12.5,5,0,Math.PI*2,true);
  ctx.fill();
  }
  ctx.restore();
}
// направление в PNG-файл
var out = fs.createWriteStream(__dirname + '/shadow.png');
var stream = canvas.createPNGStream();

stream.on('data', function(chunk){
  out.write(chunk);
});

stream.on('end', function(){
  console.log('saved png');
});
```

После запуска Node-приложения обратитесь к файлу shadow.png из своего любимого браузера. Созданное изображение показано на стр. 12.3.

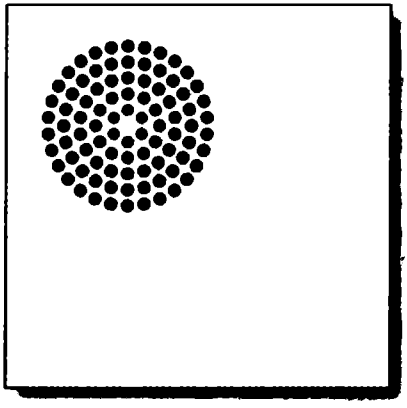


Рис. 12.3. Изображение, созданное с помощью модуля node-canvas

Скорее, вы разместите элемент `canvas` на веб-странице, чем используете Canvas-объект в Node-приложении. Один из примеров применения модуля node-canvas — динамические часы, требующие постоянного HTTP-обновления. Если нужно активизировать часы у клиента, то элемент `canvas` следует задействовать на стороне клиента.

На сервере применение элемента `canvas` имеет смысл при реализации графического представления серверной активности, например запроса к базе данных, к данным в базе данных Redis, к файлу журнала или к другим данным, постоянно находящимся на сервере. Создавая графику на сервере, вы не только можете сохранить ее для нескольких обращений, но и ограничить объем данных, перетекающих к клиенту, поскольку сможете обрабатывать графику на сервере, вместо того чтобы отправлять данные клиенту, а затем создавать графику.

Использование элемента `canvas` в Node-приложении также имеет смысл, если вы создаете компоненты игры, которые должны реагировать на действия пользователя, в особенности если графику нужно надежно сохранять для последующего доступа.

13 Веб-сокеты и Socket.IO

В этой главе мы будем работать как в клиентской, так и в серверной среде, потому что обе они необходимы, когда речь заходит о веб-сокетах и Socket.IO.

Веб-сокеты являются относительно новой веб-технологией, поддерживающей двунаправленный обмен данными в реальном масштабе времени непосредственно от клиента к серверному приложению и обратно. Обмен данными посредством сокетов осуществляется по протоколу управления передачей (Transmission Control Protocol, TCP). Библиотеки Socket.IO обеспечивают поддержку, необходимую для реализации этой технологии. Модуль для использования в вашем Node-приложении предоставляет не только библиотека Socket.IO, но и JavaScript-библиотека на стороне клиента, подключающая клиентскую конечную точку канала связи. В качестве дополнительного бонуса Socket.IO работает также в качестве связующего программного обеспечения платформы Express.

В этой главе я более полно познакомлю вас с веб-сокетами, продемонстрировав, как работает модуль Socket.IO на клиентской и на серверной сторонах.

Веб-сокеты

Однако перед тем как перейти к Socket.IO, я хочу предложить вам краткий обзор веб-сокетов. А для этого мне понадобится объяснить понятие *двунаправленной полнодуплексной связи* (bidirectional full-duplex communication).

Термин *полный дуплекс* описывает любую форму передачи данных, обеспечивающую обмен данными в обоих направлениях. Термин *двунаправленная* в отношении связи означает, что обе конечные точки передачи могут вести обмен данными, в отличие от *однонаправленной* связи, когда одна конечная точка передачи данных является отправителем, а все остальные — получателями. Веб-сокеты позволяют веб-клиенту, например браузеру, начать двунаправленное полнодуплексное взаимодействие с серверным приложением. И для этого не нужен протокол HTTP, который вводит лишние издержки в процесс обмена данными.

Веб-сокеты (WebSockets) стандартизованы консорциумом W3C (World Wide Web Consortium) как часть спецификации под общим названием WebSockets API. Сначала не обошлось без шероховатостей, потому что не успели некоторые наиболее торопливые разработчики браузеров приступить к реализации веб-сокетов в 2009 году, как серьезные проблемы безопасности заставили их либо полностью отказаться от реализации веб-сокетов, либо оставить возможность их подключения в качестве дополнительного параметра.

Для решения проблем безопасности протокол веб-сокетов был модернизирован, и новый протокол поддерживают браузеры Firefox, Chrome и Internet Explorer. Что касается Safari и Opera, то к настоящему времени они поддерживают только более старые версии этой технологии, но веб-сокеты можно подключить при конфигурировании. Большая часть мобильных браузеров также имеет только ограниченную поддержку веб-сокетов или поддерживает их более старую спецификацию.

Socket.IO решает проблему неоднородной поддержки веб-сокетов, предлагая для организации двунаправленного обмена данными несколько различных механизмов. Чтобы их использовать, нужно придерживаться следующего порядка:

- WebSockets;
- Adobe Flash Socket;
- Ajax long polling;
- Ajax multipart streaming;
- Forever iFrame for IE;
- JSONP Polling.

Главное, что нужно усвоить из этого списка, — Socket.IO поддерживает двунаправленный обмен данными в большинстве, если не во всех используемых в настоящий момент браузерах, развернутых как на настольных, так и на мобильных устройствах.

Хотя технически веб-сокеты — это не тот механизм, который используется всеми браузерами в приложениях, описываемых в данной главе, при упоминании обмена данными я оставлю название «веб-сокеты». Это короче, чем набирать фразу «двунаправленный полнодуплексный обмен данными».

Знакомство с модулем Socket.IO

Перед тем как перейти к программной реализации приложения веб-сокетов, нужно установить модуль Socket.IO на вашем сервере. Для установки модуля и поддерживающей JavaScript-библиотеки можно воспользоваться диспетчером Node-пакетов:

```
npm install socket.io
```

Socket.IO-приложение требует два разных компонента: серверное и клиентское приложения. В примерах данного раздела серверным приложением будет Node-приложение, а клиентским — JavaScript-блок в HTML-разметке веб-страницы.

Оба приложения являются адаптацией кода из примеров, предоставленных на веб-сайте Socket.IO.

Простой пример обмена данными

Клиент-серверное приложение, описываемое в данном разделе, инициирует взаимодействие между клиентом и сервером, отправляет в обоих направлениях текстовую строку, которая затем публикуется на веб-странице. Клиент всегда посылает копию последней строки на сервер, который вносит в нее изменения и отправляет обратно клиенту.

Клиентское приложение создает новое веб-сокетное соединение, используя клиентскую библиотеку Socket.IO, и прослушивает любые события с пометкой `news`. При получении события приложение извлекает текст, переданный с событием, и выводит его на веб-странице. Оно также отправляет копию текста назад на сервер посредством события `echo`. В листинге 13.1 полностью показан код клиентской веб-страницы.

Листинг 13.1. Клиентская HTML-страница в Socket.IO-приложении

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>bi-directional communication</title>
  <script src="/socket.io/socket.io.js"></script>
  <script>
     var socket = io.connect('http://localhost:8124');
     socket.on('news', function (data) {
        var html = '<p>' + data.news + '</p>';
        document.getElementById("output").innerHTML=html;
        socket.emit('echo', { back: data.news });
    });
</script>
</head>
<body>
<div id="output"></div>
</body>
</html>
```

Серверное приложение использует HTTP для прослушивания входящих запросов и обслуживает только один файл — клиентский HTML-файл. При установке нового веб-сокетного соединения это приложение для событий с пометкой `news` отправляет клиенту сообщение с текстом `Counting...` (идет подсчет).

Когда сервер получает событие `echo`, приложение извлекает текст, отправленный с этим событием, и добавляет к нему значение счетчика. В приложении создается счетчик, значение которого повышается при каждой передаче события `echo`. Когда значение счетчика достигает 50, сервер больше не передает данные назад клиенту. Весь код серверного приложения представлен в листинге 13.2.

Листинг 13.2. Серверная часть в Socket.IO-приложении

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs')

var counter;

app.listen(8124);

function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.html');
    }
    counter = 1;
    res.writeHead(200);
    res.end(data);
  });
}
io.sockets.on('connection', function (socket) {
  socket.emit('news', { news: 'world' });
  socket.on('echo', function (data) {
    if (counter <= 50) {
       counter++;
       data.back+=counter;
       socket.emit('news', {news: data.back});
    }
  });
});
```

После загрузки клиентским приложением текста на сервер вы можете наблюдать обновление показаний счетчика, пока его значение не достигнет конечного значения. Веб-страница перезагрузки не требует, и для выполнения приложения пользователю не нужно делать ничего особенного. Приложение одинаково ведет себя во всех современных браузерах, хотя технология, которая составляет основу реализации эффекта, в разных браузерах отличается.

Оба события, `news` и `echo`, являются нестандартными. Из событий сокетов Socket.IO поддерживает только событие `connection`, которое возникает на начальном этапе установки соединения, а также следующие события на серверном сокете:

`message`

Возникает при получении сообщения, отправленного методом `socket.send`.

`disconnect`

Возникает, когда либо клиент, либо сервер разрывает соединение.

На клиентском сокете поддерживаются следующие события:

`connect`
: Возникает, когда устанавливается сокетное соединение.

`connecting`
: Возникает, когда предпринимается попытка установки сокетного соединения.

`disconnect`
: Возникает, когда сокет разрывает соединение.

`connect_failed`
: Возникает, когда соединение установить не удается.

`error`
: Возникает при возникновении ошибки.

`message`
: Возникает при получении сообщения, отправленного методом `socket.send`.

`reconnect_failed`
: Возникает, когда Socket.IO не может восстановить разорванное соединение.

`reconnect`
: Возникает после повторной установки разорванного соединения.

`reconnecting`
: Возникает при попытке восстановить разорванное соединение.

Если вместо метода `emit` вы хотите задействовать веб-сокеты, можете использовать метод `send` и событие `message`. Например, на сервере приложение может методом `send` отправить сообщение клиенту и затем ждать ответ, слушая событие `message`:

```
io.sockets.on('connection', function (socket) {
   socket.send("All the news that's fit to print");
   socket.on('message', function(msg) {
      console.log(msg);
   });
});
```

На стороне клиента приложение также может слушать событие `message` и использовать метод `send` для обратной связи:

```
socket.on('message', function (data) {
   var html = '<p>' + data + '</p>';
   document.getElementById("output").innerHTML=html;
   socket.send('OK, got the data');
 });
```

В этом примере метод `send` в «ручном» режиме подтверждает получение сообщения. Если нужно получить автоматическое подтверждение получения сообщения

клиентом, можно передать методу `emit` в качестве последнего аргумента функцию обратного вызова:

```
io.sockets.on('connection', function (socket) {
   socket.emit('news', { news: "All the news that's fit to print" },
                       function(data) {
                          console.log(data);
                       });
});
```

Затем на стороне клиента мы можем передать сообщение обратно, используя функцию обратного вызова:

```
socket.on('news', function (data, fn) {
    var html = '<p>' + data.news + '</p>';
    document.getElementById("output").innerHTML=html;
    fn('Got it! Thanks!');
});
```

Сокет, переданный в виде параметра обработчику события `connection`, является частью уникального соединения между сервером и клиентом и существует до тех пор, пока существует соединение. Если соединение разрывается, Socket.IO пытается восстановить соединение.

Веб-сокеты в асинхронном мире

Приложение работает... до определенного момента. То, что в нем игнорируется асинхронная природа Node, обусловливает его падение. Счетчик, используемый в приложении, является глобальным для приложения. Если к приложению в определенный момент времени осуществляется только одно клиентское обращение, оно работает хорошо. Но если два пользователя обратятся к приложению в одно и то же время, будут получены довольно странные результаты: один браузер может получить меньшее значение, другой большее, но ожидаемого результата, скорее всего, ни один из них не получит. Добавьте еще параллельных обращений, и результаты станут еще хуже.

Нам нужен такой механизм закрепления данных, чтобы они надежно сохранялись в самом сокете независимо от событий. К счастью, у нас есть такой механизм; нужно просто добавить данные непосредственно к объекту сокета, который создается для каждого нового соединения. В листинге 13.3 показан переработанный код из листинга 13.2, где счетчик не находится в подвешенном состоянии в виде глобальной переменной, а непосредственно связан с объектом сокета. Измененный код выделен полужирным шрифтом.

Листинг 13.3. Модифицированный код сервера, в котором данные надежно сохраняются в отдельных сокетах

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs')

app.listen(8124);
```

продолжение ⇩

Листинг 13.3 *(продолжение)*

```
function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.html');
    }
    res.writeHead(200);
    res.end(data);
  });
}
io.sockets.on('connection', function (socket) {
   socket.counter = 1;
   socket.emit('news', { news: 'Counting...' });

   socket.on('echo', function (data) {
      if (socket.counter <= 50) {
         data.back+=socket.counter;
         socket.counter++;
         socket.emit('news', {news: data.back});
      }
   });
});
```

Теперь вы можете иметь несколько конкурирующих пользователей и у каждого из них будет абсолютно одинаковый обмен данными. Объект сокета существует до тех пор, пока не закрыто сокетное соединение без возможности восстановления.

Разные браузеры ведут себя по-разному. В зависимости от браузера показания счетчика могут расти быстрее или медленнее — это зависит от нижележащего механизма, используемого для управления соединением.

Код на стороне клиента

Чтобы технология Socket.IO работала, клиентская сторона приложения должна иметь доступ к JavaScript-библиотеке модуля Socket.IO на клиентской стороне. Эта библиотека подключается к странице с помощью следующего элемента `script`:

```
<script src="/socket.io/socket.io.js"></script>
```

Может возникнуть вопрос, нужно ли помещать этот код на верхний уровень вашего веб-сервера? Ответ — нет, не нужно.

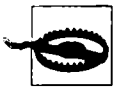
В серверном приложении при создании HTTP-разметки для веб-сервера он передается событию `listen` модуля Socket.IO:

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
```

После этого Socket.IO перехватывает запросы, отправленные на веб-сервер, и слушает запросы:

```
/socket.io/socket.io.js
```

Чтобы определить, что возвращено в ответе, Socket.IO делает довольно хитрый ход. Если клиент поддерживает веб-сокеты, то возвращаемым JavaScript-файлом будет именно тот файл, который использует веб-сокеты для реализации клиентского соединения. Если клиент не поддерживает веб-сокеты, но поддерживает технологию Forever iFrame (IE9), он возвращает именно тот клиентский JavaScript-код, который ему нужен и т. д.



Не изменяйте относительный URL-адрес для Socket.IO-приложения; если вы это сделаете, приложение работать не будет.

Настройка Socket.IO

Socket.IO поставляется с параметрами, установленными по умолчанию, и обычно в них не нужно вносить изменения. В примерах предыдущего раздела я не изменил ни одного параметра, заданного по умолчанию. Но если мне понадобится это сделать, можно будет воспользоваться Socket.IO-методом `configure`, который работает так же, как в Express и Connect. Можно задать и другие параметры, настроив среду работы приложения.

На вики-странице Socket.IO, расположенной по адресу https://github.com/learnboost/socket.io/wiki/, перечислены все варианты настройки, и я не хочу повторять здесь этот весьма пространный список. Вместо этого я продемонстрирую два из них, которые вам может быть захочется изменить.

Устанавливая значение параметра `transports`, можно изменить допустимые транспортные механизмы. По умолчанию допустимыми транспортными механизмами в порядке приоритета являются:

- websocket;
- htmlfile;
- xhr-polling;
- jsonp-polling.

Еще одним вариантом транспортного механизма является Flash Socket, который по умолчанию отключен. Если к листингу 13.3 добавить следующий код, то при доступе к приложению с помощью браузеров Opera и IE приложение задействует Flash Socket (а не Ajax long polling и не Forever iFrame соответственно):

```
io.configure('development', function() {
   io.set('transports', [
          'websocket',
          'flashsocket',
          'htmlfile',
```

продолжение ⇩

```
        'xhr-polling',
        'jsonp-polling']);
});
```

Можно также задать разные варианты конфигурации для разных сред, например для рабочей (`production`) среды и для среды разработки (`development`):

```
io.configure('production', function() {
   io.set('transports', [
        'websocket',
        'jsonp-polling']);
});
io.configure('development', function() {
   io.set('transports', [
        'websocket',
        'flashsocket',
        'htmlfile',
        'xhr-polling',
        'jsonp-polling']);
});
```

Еще один параметр управляет детализацией информации в журнале (содержимое журнала в виде отладочных инструкций выводится на консоль сервера). Чтобы отключить вывод в журнал, параметр `log level` можно установить в 1:

```
io.configure('development', function() {
   io.set('log level', 1);
});
```

Для настройки некоторых параметров, например параметра `store`, который определяет место хранения данных клиента, просто вызвать метод `configuration` недостаточно.

Однако другие предлагаемые по умолчанию параметры (исключая `log level` и `transports`) для обучения работе с Socket.IO вам вполне подойдут.

Чат: «Hello, World» для веб-сокетов

У каждой технологии есть собственная версия приложения «Hello, World» — первого приложения, которое обычно создается при ее изучении. Для веб-сокетов и Socket.IO этим приложением, похоже, должен быть клиент чата. На GitHub-сайте Socket.IO предоставляется клиент чата (а также система групповых дискуссий IRC, или клиент Internet relay chat), а при поиске с аргументом «Socket.IO and chat» можно получить список с рядом интересных примеров.

В этом разделе я демонстрирую код для очень простого клиента чата. В нем нет никаких дополнительных компонентов, используется только Socket.IO (и никаких других библиотек ни на клиентской, ни на серверной сторонах), но при этом он совершенно определенно показывает, насколько модуль Socket.IO позволяет упростить приложение, реализовать которое без него было бы очень трудно.

В приложении используется два новых метода для обслуживания взаимодействия. В предыдущих примерах для передачи данных между клиентом и сервером в приложениях применялся либо метод `send`, либо метод `emit`. Такой вариант обмена данными ограничен сокетом и видим только пользователю, получающему сообщение, независимо от того, сколько человек соединено с сервером.

Для рассылки данных каждому, кто соединен с сервером, можно воспользоваться методом `emit` объекта платформы Socket.IO:

```
io.sockets.emit();
```

Теперь сообщение получит любой, имеющий сокетное соединение с сервером.

Можно также передать сообщение каждому, но указать конкретных лиц. Для этого нужно вызвать метод `broadcast.emit` для сокета того человека, которому вы не хотите показывать сообщение:

```
socket.broadcast.emit();
```

В простом приложении для чата при подключении нового клиента клиентское приложение выдает приглашение на ввод имени, а затем сообщает всем другим подключенным клиентам, что этот клиент вошел в пространство чата. Кроме того, клиентское приложение предоставляет текстовое поле и кнопку для отправки сообщений, а также место, куда выводятся новые сообщения от всех участников. Код клиентского приложения показан в листинге 13.4.

Листинг 13.4. Клиентское приложение для чата

```
  <!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>bi-directional communication</title>
  <script src="/socket.io/socket.io.js"></script>
  <script>
     var socket = io.connect('http://localhost:8124');
     socket.on('connect', function() {
        socket.emit('addme', prompt('Who are you?'));
     });
     socket.on('chat',function(username, data) {
       var p = document.createElement('p');
       p.innerHTML = username + ':  ' + data;
       document.getElementById('output').appendChild(p);
     });
     window.addEventListener('load',function() {
        document.getElementById('sendtext').addEventListener('click',
            function() {
               var text = document.getElementById('data').value;
               socket.emit('sendchat', text);
            }, false);
      }, false);
</script>
</head>
```

продолжение ↛

Листинг 13.4 *(продолжение)*

```
<body>
<div id="output"></div>
<div id="send">
  <input type="text" id="data" size="100" /><br />
  <input type="button" id="sendtext" value="Send Text" />
</div>
</body>
</html>
```

Помимо обычной JavaScript-функциональности для перехвата события щелчка на кнопке и выдачи приглашения на ввод имени пользователя, функциональность этого кода мало чем отличается от рассмотренной в предыдущих примерах.

На сервере имя нового пользователя присоединяется к сокету в качестве данных. Сервер уведомляет пользователя напрямую, а затем распространяет его имя среди других участников чата. При получении сервером любого нового сообщения чата он присоединяет имя пользователя к сообщению, чтобы каждый мог видеть, кто его послал. И наконец, когда клиент отключается от чата, всем подключенным пользователям рассылается еще одно сообщение о том, что этот клиент уже не является участником чата. Полный код серверного приложения показан в листинге 13.5.

Листинг 13.5. Серверное приложение для чата

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs');

app.listen(8124);

function handler (req, res) {
  fs.readFile(__dirname + '/chat.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading chat.html');
    }
    res.writeHead(200);
    res.end(data);
  });
}

io.sockets.on('connection', function (socket) {

   socket.on('addme',function(username) {
      socket.username = username;
      socket.emit('chat', 'SERVER', 'You have connected');
      socket.broadcast.emit('chat', 'SERVER', username + ' is on deck');
   });

   socket.on('sendchat', function(data) {
```

```
    io.sockets.emit('chat', socket.username, data);
  });

  socket.on('disconnect', function() {
    io.sockets.emit('chat', 'SERVER', socket.username + ' has left the
                                                        building');
  });
});
```

На рис. 13.1 показаны результаты запуска приложения при тестировании в четырех различных браузерах (Chrome, Firefox, Opera и IE).

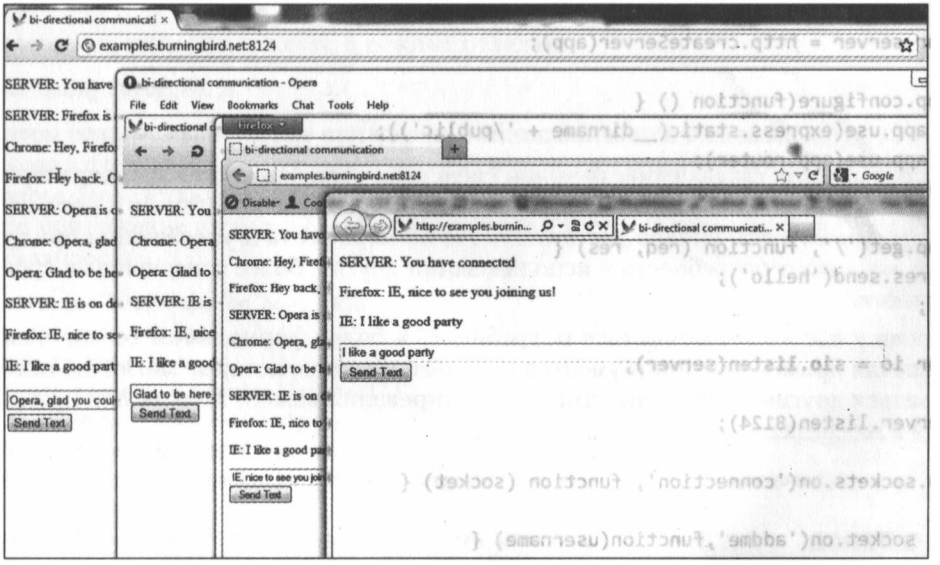


Рис. 13.1. Тестирование в нескольких различных браузерах приложения для чата, созданного с применением технологии Socket.IO

К приложению может быть добавлен список подключенных к чату людей, чтобы новые участники при входе в чат видели, кто в нем участвует. Это дополнение может быть реализовано с помощью глобального массива, поскольку он, в отличие от имени пользователя, доступен всем клиентам. Доработку кода я оставляю вам в качестве самостоятельного упражнения.

Использование Socket.IO с Express

Во всех предыдущих приложениях в качестве веб-сервера использовался Node-модуль HTTP. Однако Socket.IO-приложение легко можно встроить в Express (или Socket.IO встроить в Express-приложение). Главное, помнить, что модуль Socket.IO должен иметь возможность прослушивать запросы до того, как они начнут обрабатываться в Express.

В листинге 13.6 представлена переработанная версия серверного компонента приложения для чата из последнего раздела. Здесь для обработки всех запросов к веб-службе используется Express. Строка кода, необходимая для интеграции Socket.IO и Express, выделена полужирным шрифтом. Фактические компоненты обмена данными по сравнению с листингом 13.5 не изменились.

Листинг 13.6. Перенос в Express сервера для чата

```
var express = require('express'),
    sio = require('socket.io'),
    http = require('http'),
    app = express();

var server = http.createServer(app);

app.configure(function () {
  app.use(express.static(__dirname + '/public'));
  app.use(app.router);
});

app.get('/', function (req, res) {
  res.send('hello');
});

var io = sio.listen(server);

server.listen(8124);

io.sockets.on('connection', function (socket) {

   socket.on('addme',function(username) {
      socket.username = username;
      socket.emit('chat', 'SERVER', 'You have connected');
      socket.broadcast.emit('chat', 'SERVER', username + ' is on deck');
   });

   socket.on('sendchat', function(data) {
      io.sockets.emit('chat', socket.username, data);
   });

   socket.on('disconnect', function() {
      io.sockets.emit('chat', 'SERVER', socket.username + ' has left the
                                                        building');
   });
});
```

Express-приложение передает данные HTTP-серверу, HTTP-сервер, в свою очередь, передает данные модулю Socket.IO. Все три модуля работают вместе, чтобы все запросы, будь они к веб-службе или к чату, обрабатывались правильно.

Хотя для клиента чата используется статическая страница, встроить в нее шаблон не трудно. Единственное, что нужно сделать, — обеспечить целостность блока сценария с кодом клиентского приложения и не забыть включить ссылку на библиотеку Socket.IO.

14 Тестирование и отладка Node-приложений

В предыдущих главах единственным средством отладки, используемым в примерах, был вывод информации на консоль. При разработке небольших и несложных приложений этого вполне достаточно. Но по мере роста и усложнения приложений возникает потребность в использовании других, более совершенных средств отладки.

Также у вас может появиться потребность в более формальном тестировании, включая применение инструментов разработки тестов, которые могли бы использоваться другими специалистами для тестирования вашего модуля или приложения в их средах.

Отладка

Откровенно говоря, для меня файл console.log всегда останется наиболее предпочтительным средством отладки, но по мере разрастания и усложнения вашего приложения этот файл становится все более и более бесполезным. Как только приложение перестает быть простым, приходится задействовать более сложные средства отладки. В следующих разделах рассмотрены некоторые из возможных вариантов.

Отладчик Node.js

Движок V8 поставляется со встроенным отладчиком, который можно использовать с Node-приложениями, к тому же Node предоставляет клиента, упрощающего его применение. Начнем с добавления отладочных инструкций в код в том месте, где мы хотим установить контрольную точку:

```
// создание прокси-сервера, слушающего все запросы
httpProxy.createServer(function(req,res,proxy) {
```

```
    debugger;
    if (req.url.match(/^\/node\//))
       proxy.proxyRequest(req, res, {
         host: 'localhost',
         port: 8000
       });
    else
       proxy.proxyRequest(req,res, {
         host: 'localhost',
         port: 8124
       });
}).listen(9000);
```

Теперь запустим приложение в режиме отладки:

```
node debug debugger.js
```

В этом режиме выполнение приложения останавливается в начале файла. Для перехода к следующей контрольной точке нужно ввести команду cont или ее аббревиатуру c. Это заставляет отладчик остановиться на первой контрольной точке, после чего приложение входит в состояние ожидания ввода от пользователя (например, веб-запроса):

```
< debugger listening on port 5858
connecting... ok
break in app2.js:1
  1 var connect = require('connect'),
  2     http = require('http'),
  3     fs = require('fs'),
debug> cont (--> note it is just waiting at this point for a web request)
break in app2.js:11
  9 httpProxy.createServer(function(req,res,proxy) {
 10
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
debug>
```

В этой ситуации у вас есть несколько вариантов действий. Вы можете выполнять код в пошаговом режиме, используя команду next (n) для перехода к следующей инструкции, команду step (s) для перехода к следующей инструкции с заходом в код функций или команду out (o) для перехода к следующей инструкции с выходом из кода функций. В следующем коде отладчик останавливается в контрольной точке, и следующие несколько строк кода выполняются в пошаговом режиме с помощью команды next до строки 13, в которой находится вызов функции. С этого места используется команда step для пошагового выполнения строк кода внутри функции. Затем можно последовательно выполнить строки кода функции с помощью команды next и вернуться к основному коду приложения, используя команду out:

```
debug> cont
break in app2.js:11
  9 httpProxy.createServer(function(req,res,proxy) {
 10
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
debug> next
break in app2.js:12
 10
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
 14         host: 'localhost',
debug> next
break in app2.js:13
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
 14         host: 'localhost',
 15         port: 8000
debug> step
break in /home/examples/public_html/node/node_modules/http-proxy/lib/
node-http-proxy/routing-proxy.js:144
 142 //
 143 RoutingProxy.prototype.proxyRequest = function (req, res, options) {
 144   options = options || {};
 145
 146   //
debug> next
break in /home/examples/public_html/node/node_modules/http-proxy/lib/
node-http-proxy/routing-proxy.js:152
 150   // arguments are supplied to `proxyRequest`
 151   //
 152   if (this.proxyTable && !options.host) {
 153     location = this.proxyTable.getProxyLocation(req);
 154
debug> out
break in app2.js:22
 20         port: 8124
 21       });
 22 }).listen(9000);
 23
 24 // add route for request for dynamic resource
```

Вы также можете установить новую контрольную точку либо на текущей строке с помощью команды `setBreakpoint` (`sb`), либо на первой строке именованной функции или файла сценария:

```
break in app2.js:22
 20         port: 8124
 21       });
 22 }).listen(9000);
 23
 24 // add route for request for dynamic resource
debug> sb()
 17    else
 18       proxy.proxyRequest(req,res, {
 19         host: 'localhost',
 20         port: 8124
 21       });
*22 }).listen(9000);
 23
 24 // add route for request for dynamic resource
 25 crossroads.addRoute('/node/{id}/', function(id) {
 26    debugger;
 27 });
```

Снимается контрольная точка командой `clearBreakpoint` (`cb`).

В дополнение к использованию REPL для просмотра переменных, вы можете добавить выражение к списку просмотра и вывести список текущих просмотров:

```
break in app2.js:11
  9 httpProxy.createServer(function(req,res,proxy) {
 10
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
debug> repl
Press Ctrl + C to leave debug repl
> req.url
'/node/174'
debug>
```

Команда `backtrace` пригодится для *обратной трассировки* (вывода списка текущих активных вызовов функций) выполняемого фрагмента кода:

```
debug> backtrace
#0 app2.js:22:1
#1 exports.createServer.handler node-http-proxy.js:174:39
```

Когда вам понадобится просмотреть перечень доступных команд, наберите команду `help`:

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o), backtrace (bt),
setBreakpoint (sb), clearBreakpoint (cb), watch, unwatch, watchers, repl,
restart, kill, list, scripts, breakpoints, version
```

Встроенный отладчик — средство полезное, но иногда возникает потребность в чем-то большем. У вас есть и другие варианты, включая обращение к отладчику движка V8 напрямую с помощью ключа командной строки `--debug`:

```
node --debug app.js
```

В результате будет инициировано TCP-соединение с отладчиком, и вы войдете в приглашение на ввод команд отладчика V8. Это интересный вариант, но он требует, чтобы вы хорошо разбирались в работе отладчика V8 (и в его командах).

Другим вариантом может стать отладка с помощью браузера WebKit — для этого служит такое приложение, как рассматриваемый далее Node-инспектор.

Отладка на стороне клиента с помощью Node-инспектора

Чтобы начать отладку, нужно настроить Node-инспектор, но дополнительные усилия тратятся не зря.

В первую очередь установить Node-инспектор с помощью диспетчера Node-пакетов:

```
npm install -g node-inspector
```

Чтобы воспользоваться функциональными возможностями Node-инспектора, нужно запустить приложение, используя ключ отладки движка V8:

```
node --debug app.js
```

Затем нужно запустить Node-инспектор либо в фоновом, либо в приоритетном режиме:

```
node-inspector
```

При запуске приложения вы должны получить следующее сообщение:

```
node-inspector
   info  - socket.io started
visit http://0.0.0.0:8080/debug?port=5858 to start debugging
```

Используя браузер на базе WebKit (Safari или Chrome), обратитесь к отлаживаемой странице. Свой пример я запускаю на своем сервере, поэтому использую следующий URL-адрес:

```
http://examples.burningbird.net:8080/debug?port=5858
```

В браузере на стороне клиента открывается отладчик (часть инструментария разработчика) и выполнение останавливается на первой контрольной точке. Далее вы можете использовать инструменты, которые вам уже, возможно, знакомы по разработке JavaScript-приложений на стороне клиента, например можно перейти через несколько строк кода и проверить свойства объекта, как показано на рис. 14.1.

Node-инспектор безусловно является наиболее предпочтительным решением для отладки серверного приложения. Можно также воспользоваться командной строкой, но возможность видеть сразу весь код и применять знакомый набор инструментов с лихвой компенсирует незначительные усилия, потраченные на установку Node-инспектора.

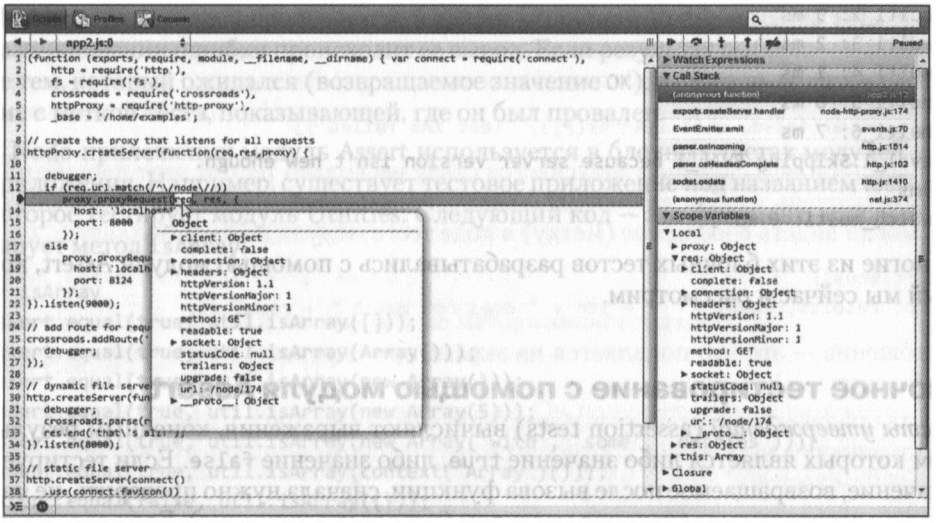


Рис. 14.1. Запуск Node-инспектора в браузере Chrome для Node-приложения, запущенного на удаленном сервере

Если со временем вы решите разместить свое Node-приложение в облачной службе, эта служба, как правило, предлагает собственный инструментарий разработки, включая отладчики.

Блочное тестирование

При *блочном тестировании* отдельные компоненты приложения изолируются от других. Многие тесты, предоставленные в каталоге tests, Node-модулей являются блочными. Все тесты в подкаталоге test установленной копии Node также являются блочными.

Тестовые сценарии для модулей можно запустить с помощью диспетчера Node-пакетов. Наберите в подкаталоге module следующую команду:

```
npm test
```

Эта команда запустит для модуля тестовый сценарий, если таковой имеется. Когда в подкаталоге module я запускал тестовый сценарий для модуля node-redis (рассмотренного в главе 9), на консоль выводились успешные результаты тестирования, например показанный здесь фрагмент вывода:

```
Connected to 127.0.0.1:6379, Redis server version 2.4.11
Using reply parser hiredis
- flushdb: 1 ms
- multi_1: 3 ms
```

продолжение ⇩

```
- multi_2: 9 ms
- multi_3: 2 ms
- multi_4: 1 ms
- multi_5: 0 ms
- multi_6: 7 ms
- eval_1:Skipping EVAL_1 because server version isn't new enough.
 0 ms
- watch_multi: 0 ms
```

Многие из этих блочных тестов разрабатывались с помощью модуля Assert, который мы сейчас и рассмотрим.

Блочное тестирование с помощью модуля Assert

Тесты утверждений (assertion tests) вычисляют выражения, конечным результатом которых является либо значение `true`, либо значение `false`. Если тестируется значение, возвращаемое после вызова функции, сначала нужно проверить, не является ли возвращаемое значение массивом (первое утверждение). Если содержимое массива должно быть определенной длины, выполняется условный тест на длину (второе утверждение) и т. д. Существует только один встроенный в Node модуль, ориентированный на эту форму тестирования — это Assert.

Модуль Assert включается в приложение с помощью следующего кода:

```
var assert = require('assert');
```

Чтобы понять, как использовать модуль Assert, давайте посмотрим, как с ним работают существующие модули. Следующий тест, находящийся в сценарии test.js, можно найти в установленной копии модуля node-redis:

```
var name = "FLUSHDB";
client.select(test_db_num, require_string("OK", name));
```

В тесте имеется функция `require_string`, которая возвращает некую функцию, использующую методы `assert.equal` и `assert.stringEqual` модуля Assert:

```
function require_string(str, label) {
  return function (err, results) {
    assert.strictEqual(
      null, err, "result sent back unexpected error: " + err);
    assert.equal(
      str, results, label + " " + str + "does not match " + results);
    return true;
  };
}
```

Первый тест, `assert.stringEqual`, не проходит, если объект `err`, возвращенный в Redis-тесте, не равен `null`. Второй тест, `assert.equal`, не проходит, если результаты не равны ожидаемому строковому значению. Выполнение функции дойдет до инструкции `return true` только в том случае, если оба теста будут пройдены успешно (то есть ни один из тестов не провалится).

На самом деле, здесь проверяется, была ли успешной Redis-команда `select`. При возникновении ошибки происходит ее вывод. Если результат выборки оказывается не тем, который ожидался (возвращаемое значение `OK`), в ответ выводится сообщение с меткой теста, показывающей, где он был провален.

В Node-приложении модуль Assert используется в блочных тестах модулей этого приложения. Например, существует тестовое приложение под названием test-util.js, которое тестирует модуль Utilities. Следующий код — это фрагмент, который тестирует метод `isArray`:

```
// isArray
assert.equal(true, util.isArray([]));
assert.equal(true, util.isArray(Array()));
assert.equal(true, util.isArray(new Array()));
assert.equal(true, util.isArray(new Array(5)));
assert.equal(true, util.isArray(new Array('with', 'some', 'entries')));
assert.equal(true, util.isArray(context('Array')()));
assert.equal(false, util.isArray({}));
assert.equal(false, util.isArray({ push: function() {} }));
assert.equal(false, util.isArray(/regexp/));
assert.equal(false, util.isArray(new Error));
assert.equal(false, util.isArray(Object.create(Array.prototype)));
```

У методов `assert.equal` и `assert.strictEqual` имеются два обязательных параметра: ожидаемый ответ и выражение, вычисляемое для ответа. В предыдущем Redis-тесте метод `assert.strictEqual` ожидает результата `null` для аргумента `err`. Если это ожидание не оправдается, тест считается проваленным. Если в методе `assert.equal` теста `isArray` в исходном коде Node выражение вычисляется в `true` и ожидаемый ответ имеет значение `true`, метод `assert.equal` завершается успешно, но при этом ничего не выводится, то есть результатом является *молчание*.

Однако если ответ, в который вычисляется выражение, отличается от ожидаемого, метод `assert.equal` отвечает исключением. Внесем изменения в первую инструкцию теста `isArray` в исходном коде Node:

```
assert.equal(false, util.isArray([]));
```

Тогда результат будет следующим:

```
node.js:201
        throw e; // ошибка process.nextTick или сообщение
              ^  // 'error' на первом проходе
AssertionError: false == true
    at Object.<anonymous>
        (/home/examples/public_html/node/chap14/testassert.js:5:8)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

У методов `assert.equal` и `assert.strictEqual` есть также третий необязательный параметр — это сообщение, выводимое вместо того, что выводится по умолчанию в случае провала теста:

```
assert.equal(false, util.isArray([]), 'Test 1Ab failed');
```

Когда в тестовом сценарии запускается сразу несколько тестов, этот параметр может быть очень удобен для определения, какой из тестов оказался проваленным. Вы могли видеть сообщение (метку) в коде теста модуля node-redis:

```
assert.equal(
  str, results, label + " " + str + " does not match " + results);
```

Сообщение — это что появляется на экране при перехвате исключения и выводе сообщения.

Все следующие методы модуля Assert получают те же три параметра, но тестируемое значение и выражение соотносятся друг с другом по-разному (что понятно по названиям тестов):

`assert.equal`

> Тест считается проваленным, если результаты выражения и заданного значения не равны друг другу.

`assert.strictEqual`

> Тест считается проваленным, если результаты выражения и заданного значения не имеют строгого соответствия друг другу.

`assert.notEqual`

> Тест считается проваленным, если результаты выражения и заданного значения равны друг другу.

`assert.notStrictEqual`

> Тест считается проваленным, если результаты выражения и заданного значения строго соответствуют друг другу.

`assert.deepEqual`

> Тест считается проваленным, если результаты выражения и заданного значения не равны друг другу.

`assert.notDeepEqual`

> Тест считается проваленным, если результаты выражения и заданного значения равны друг другу.

Последние два метода, `assert.deepEqual` и `assert.notDeepEqual`, работают со сложными объектами, например с массивами или объектами. Следующий вызов метода `assert.deepEqual` оказывается успешным:

```
assert.deepEqual([1,2,3],[1,2,3]);
```

Однако вызов метода `assert.equal` с теми же параметрами оказывается проваленным.

Остальные методы модуля Assert получают другие параметры. Вызов `assert` в качестве метода, передача ему значения и сообщения эквивалентно вызову метода `assert.isEqual`, с передачей ему в качестве первого параметра значения `true`, выражения и сообщения. Например:

```
var val = 3;
assert(val == 3, 'Equal');
```

Этот эквивалентен следующему:

```
assert.equal(true, val == 3, 'Equal');
```

Еще одним вариантом точно такого же метода является `assert.ok`:

```
assert.ok(val == 3, 'Equal');
```

Метод `assert.fail` вбрасывает исключение. Он получает четыре параметра: значение, выражение, сообщение и оператор, который используется для разделения значения и выражения в сообщении при вбрасывании исключения. Например:

```
try {
  var val = 3;
  assert.fail(3, 4, 'Fails Not Equal', '==');
} catch(e) {
  console.log(e);
}
```

Этот фрагмент кода выдает на консоль следующее сообщение:

```
{ name: 'AssertionError',
  message: 'Fails Not Equal',
  actual: 3,
  expected: 4,
  operator: '==' }
```

Функция `assert.ifError` получает значение и вбрасывает исключение, только если значение разрешается во что-нибудь иное, кроме `false`. Как утверждается в документации по Node, она хорошо подходит для тестирования объекта `error` в качестве первого аргумента функции обратного вызова:

```
assert.ifError(err); // исключение вбрасывается только при значении true
```

Последними методами являются `assert.throws` и `assert.doesNotThrow`. Первый из них ожидает, что будет вброшено исключение, второй нет. Оба метода в качестве первого обязательного параметра получают блок кода, а в качестве необязательных второго и третьего параметров — ошибку и сообщение. Объектом ошибки может быть конструктор, регулярное выражение или функция проверки данных. В следующем фрагменте кода выводится сообщение об ошибке, поскольку регулярное выражение для ошибки, использованное в качестве второго параметра, не соответствует сообщению об ошибке:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
```

продолжение ⇩

```
  /something/
  )
} catch(e) {
  console.log(e.message);
}
```

Модуль Assert позволяет разрабатывать довольно сложные тесты. Однако одним из основных его ограничений является необходимость создавать для тестов множество оболочек, чтобы при провале одного теста не оказался проваленным весь сценарий тестирования. Для решения данного вопроса нам пригодятся платформы блочного тестирования более высокого уровня, например рассматриваемая следующей платформа Nodeunit.

Блочное тестирование с помощью модуля Nodeunit

Nodeunit позволяет составить сценарий для нескольких тестов. Будучи включенным в сценарий, каждый тест проводится последовательно, а результаты вводятся в отчет в согласованной форме. Для использования модуля Nodeunit его нужно глобально установить с помощью диспетчера Node-пакетов:

```
npm install nodeunit -g
```

Nodeunit позволяет запустить последовательность тестов, ничего не заключая в блоки `try-catch`. Этот инструмент поддерживает все тесты модуля Assert, а также предлагает два собственных метода для управления тестами. Тесты имеют вид вариантов тестирования, каждый из которых экспортируется как метод объекта в сценарий тестирования. Каждый вариант тестирования получает контрольный объект, который обычно называется `test`. Первый вызов метода в варианте тестирования относится к методу `expect` объекта `test` и предназначен для Nodeunit-сообщения о том, сколько тестов ожидается в этом варианте тестирования. Последний вызов метода в варианте тестирования относится к методу `done` объекта `test` и предназначен для Nodeunit-сообщения о том, что вариант тестирования завершен. Все, что находится между ними, фактически составляет блочный тест:

```
module.exports = {
   'Test 1' : function(test) {
      test.expect(3); // три теста
      ... // тесты
      test.done();
   },
   'Test 2' : function (test) {
      test.expect(1); // только один тест
      ... // тест
      test.done();
   }
};
```

Для запуска тестов нужно набрать команду `nodeunit`, указав следом имя сценария тестирования:

```
nodeunit thetest.js
```

В листинге 14.1 показан небольшой, но полноценный сценарий тестирования, содержащий шесть утверждений (тестов). Он состоит из двух тестовых блоков с метками `Test 1` и `Test 2`. Первый тестовый блок объединяет четыре отдельных теста, второй — два. Количество тестов, запускаемых в блоке, отражается в вызове метода `expect`.

Листинг 14.1. Сценарий Nodeunit-тестирования с двумя тестовыми блоками, содержащими в целом шесть тестов

```
var util = require('util');

module.exports = {
    'Test 1' : function(test) {
        test.expect(4);
        test.equal(true, util.isArray([]));
        test.equal(true, util.isArray(new Array(3)));
        test.equal(true, util.isArray([1,2,3]));
        test.notEqual(true, (1 > 2));
        test.done();
    },
    'Test 2' : function(test) {
        test.expect(2);
        test.deepEqual([1,2,3], [1,2,3]);
        test.ok('str' === 'str', 'equal');
        test.done();
    }
};
```

Результат запуска сценария Nodeunit-тестирования, показанного в листинге 14.1, имеет следующий вид:

```
example1.js
✓ Test 1
✓ Test 2
OK: 6 assertions (3ms)
```

Символы перед метками тестов означают успешное или неудачное выполнение теста: галочкой обозначается успех, а крестиком — неудача. Ни один из тестов в этом сценарии не завершился неудачей, поэтому в выводе отсутствуют ошибки сценария или трассы стека.

Для любителей CoffeeScript новейшая версия Nodeunit поддерживает приложения Coffee-Script.

Другие платформы тестирования

Помимо платформы Nodeunit, рассмотренной в предыдущем разделе, Node-разработчикам доступны и несколько других платформ тестирования. Некоторые

из них проще в использовании, чем другие, и у каждой есть свои сильные и слабые стороны. Далее я коротко рассмотрю три платформы: Mocha, Jasmine и Vows.

Mocha

Установите модуль Mocha с помощью диспетчера Node-пакетов:

```
npm install mocha -g
```

Mocha считается преемником другой популярной платформы тестирования — Espresso.

Mocha работает как в браузерах, так и в Node-приложениях. Этот инструмент позволяет проводить тестирование в асинхронном режиме с помощью функции done, хотя при тестировании в синхронном режиме вызов функции может быть опущен. Mocha может использоваться с любой библиотекой утверждений (assertion library).

Следующий код является примером Mocha-теста, использующего библиотеку утверждений should.js:

```
should = require('should')
describe('MyTest', function() {
  describe('First', function() {
    it('sample test', function() {
      "Hello".should.equal("Hello");
    });
  });
});
```

Перед запуском теста нужно установить библиотеку should.js:

```
npm install should
```

Затем нужно запустить тест с помощью следующей командной строки:

```
mocha testcase.js
The test should succeed:
  ✓ 1 test complete (2ms)
```

Jasmine

Jasmine является платформой разработки через реализацию поведения (Behavior-Driven Development, BDD) и может использоваться с различными технологиями, включая Node с модулем jasmine-node. Модуль jasmine-node устанавливается с помощью диспетчера Node-пакетов:

```
npm install jasmine-node -g
```

Обратите внимание на имя модуля: jasmine-node. Оно отличается от формата node-имя_модуля (или в сокращенной форме имя_модуля), который встречался в этой книге до сих пор.

В GitHub-репозитории jasmine-node имеются примеры, которые находятся в подкаталоге specs. Как и большинство других платформ тестирования, Node-модуль Jasmine получает функцию done в качестве функции обратного вызова, что позволяет проводить тестирование в асинхронном режиме.

Для использования jasmine-node существуют некоторые требования, предъявляемые к окружению. Во-первых, тест должен быть в подкаталоге specs. Модуль jasmine-node является приложением командной строки, поэтому вы можете указать корневой каталог, но он ожидает, что тесты будут находиться в подкаталоге specs.

Во-вторых, тестам нужно давать имена в определенном формате. Если тест написан на JavaScript, имя файла теста должно оканчиваться на .spec.js, если тест написан на CoffeeScript, имя файла должно оканчиваться на .spec.coffee. В каталоге specs можно использовать подкаталоги. При запуске модуля jasmine-node он инициирует запуск всех тестов во всех подкаталогах.

Чтобы продемонстрировать работу модуля, я создал простой сценарий тестирования, использующий модуль Zombie (рассматриваемый далее) для создания запроса к веб-серверу и обращения к контенту страницы. Я назвал файл tst.spec.js и поместил его в каталог specs своей среды разработки:

```
var zombie = require('zombie');

describe('jasmine-node', function(){

    it("should respond with Hello, World!", function(done) {
      zombie.visit("http://examples.burningbird.net:8124",
                                          function(error, browser, status){
        expect(browser.text()).toEqual("Hello, World!\n");
        done();
      });
    });
});
```

Веб-сервер взят из главы 1, его действия ограничиваются возвращением сообщения «Hello, World!». Обратите внимание на использование символа разделителя строк — если его не будет, тест окажется проваленным.

Я запустил тест с помощью следующей командной строки:

```
jasmine-node --test-dir /home/examples/public_html/node
```

В результате был получен следующий результат:

```
Finished in 0.133 seconds
1 test, 1 assertion, 0 failures
```

То есть тест прошел успешно.

В Jasmine используется метод path.existsSync, вместо которого в Node 0.8 рекомендуется применять метод js.existsSync. Надеемся, что исправление не заставит себя долго ждать.

Для CoffeeScript-сценария мне пришлось бы добавить параметр `--coffee`:

```
jasmine-node --test-dir /home/examples/public_html/node --coffee
```

Vows

Еще одной BDD-платформой тестирования является Vows, причем у нее есть одно преимущество над другими платформами — более полная документация. Тестирование строится на тестовых наборах, которые, в свою очередь, состоят из пакетов последовательно выполняемых тестов. Каждый пакет состоит из одного или нескольких контекстов, выполняемых параллельно, а каждый из контекстов содержит тему (topic), в которой мы наконец-то добираемся до исполняемого кода. Тест внутри кода известен как *обет* (vow). То, что составляет гордость Vows и отличает эту платформу от других, заключается в четком разделении того, что должно тестироваться (тема) и собственно тестом (обет).

Учитывая, что знакомые вроде бы слова используются здесь несколько необычно, давайте посмотрим на простой пример, чтобы лучше понять, как работает Vows-тест. Но сначала Vows нужно установить:

```
npm install vows
```

Для работы с Vows я воспользовался простым модулем круга, созданным ранее в этой книге, а теперь отредактированным для задания точности:

```
var PI = Math.PI;

exports.area = function (r) {
  return (PI * r * r).toFixed(4);
};

exports.circumference = function (r) {
  return (2 * PI * r).toFixed(4);
};
```

Мне понадобилось изменить точность результата, потому что я собирался проводить тест утверждения на равенство в отношении результатов Vows-приложения.

В тестовом Vows-приложении объект круга является темой, а методы вычисления площади (`area`) и длины окружности (`circumference`) — обетами. Оба они инкапсулированы в качестве Vows-контекста. Набором является все тестовое приложение, а пакетом — экземпляр теста (круг и два метода). Весь тест представлен в листинге 14.2.

Листинг 14.2. Тестовое Vows-приложение с одним пакетом, одним контекстом, одной темой и двумя обетами

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');
```

```
var suite = vows.describe('Test Circle');

suite.addBatch({
    'An instance of Circle': {
        topic: circle,
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic.circumference(3.0), 18.8496);
        },
        'should be able to calculate area': function(topic) {
            assert.equal (topic.area(3.0), 28.2743);
        }
    }
}).run();
```

Запуск приложения вместе с Node приводит к запуску теста, потому что в конец метода `addBatch` добавлен метод `run`:

```
node example2.js
```

В результате должны быть проведены два успешных теста:

```
·· ✓ OK " 2 honored (0.003s)
```

Темой всегда является асинхронная функция или некое значение. Вместо использования объекта `circle` в качестве темы я мог бы напрямую указать в качестве тем методы объекта, используя замыкания функций:

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
    'Testing Circle Circumference': {
        topic: function() { return circle.circumference;},
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic(3.0), 18.8496);
        },
    },
    'Testing Circle Area': {
        topic: function() { return circle.area;},
        'should be able to calculate area': function(topic) {
            assert.equal (topic(3.0), 28.2743);
        }
    }
}).run();
```

В этой версии примера каждый контекст является объектом, заданным в заголовке: `Testing Circle Circumference` (тестирование вычисления длины окружности)

и `Testing Circle Area` (тестирование вычисления площади круга). Внутри каждого контекста имеется одна тема и один обет.

Можно объединять множество пакетов с несколькими контекстами в каждом, которые, в свою очередь, могут иметь несколько тем и несколько тестов.

Приемочное тестирование

Приемочное тестирование (acceptance testing) отличается от блочного тестирования тем, что главной целью первого является определение факта соответствия приложения требованиям пользователя. Блочное тестирование гарантирует *надежность* приложения, а приемочное тестирование — его *пригодность*.

Приемочное тестирование может быть выполнено с помощью предопределенных сценариев, разработанных пользователями и реализованных в согласованных параметрах. Приемочное тестирование также может быть автоматизировано с помощью сценариев, но таких, которые реализуются инструментальными средствами, а не людьми. Эти средства не полностью удовлетворяют всем аспектам приемочного тестирования, поскольку не позволяют выработать субъективную точку зрения («Эта форма веб-страницы неудобна в использовании») и не могут точно заострить внимание на тех трудно обнаруживаемых просчетах, которые пользователи, похоже, всегда находят. Тем не менее они могут гарантировать, что программа соответствует предъявляемым к ней требованиям.

Selenium-тестирование с помощью модуля Soda

Если при тестировании вы хотите добиться максимального уровня сложности и использовать настоящие браузеры, а не эмуляторы и готовы платить за подписку на службу тестирования, то вам, возможно, захочется выбрать Selenium, Sauce Labs и Node-модуль Soda.

Проект Selenium появился из-за желания автоматизировать инструменты тестирования. Этот инструмент состоит из библиотеки кода, системы удаленного управления (Remote Control, RC) и интегрированной среды разработки (Integrated Development Environment, IDE). Selenium IDE является дополнительным программным модулем Firefox, а Selenium RC — Java-файлом с расширением .jar. Первая версия Selenium (Selenium 1) была построена на JavaScript, что стало одной из проблем: проект Selenium разделил с JavaScript все ограничения этого языка. Еще одной попыткой предоставить автоматизированный набор для тестирования стал проект WebDriver, появившийся в попытках преодолеть ограничения, присущие Selenium. В настоящее время полным ходом идет работа над проектом Selenium 2 (Selenium WebDriver), призванным объединить проекты Selenium 1 и WebDriver.

Хост для тестирования с помощью Selenium 1 предоставляет компания Sauce Labs. При этом поддерживается тестирование с различными веб-браузерами и в различных средах, например с Opera под управлением Linux или с IE9 под управлением Windows 7. Однако есть два серьезных ограничения: не поддерживается Mac OS X и отсутствует среда для тестирования мобильных устройств. Тем не менее данный

способ тестирования приложений с несколькими версиями браузеров, например IE, без Selenium было бы трудно (если вообще возможно) провести при наличии всего одной машины.

Sauce Labs предоставляет различные планы подписки, включая основной и бесплатный пробный план. Основной план допускает наличие двух параллельных пользователей и предлагает 200 минут в месяц по запросу и 45 минут в месяц на исследование — этого более чем достаточно, чтобы разработчик мог во всем разобраться. Сайт ориентирован на Ruby-разработчиков, но содержит также и Node-модуль Soda, которым можно воспользоваться.

Soda предоставляет Node-оболочку для Selenium-тестирования. Вот как выглядит пример, включенный в документацию модуля:

```
var soda = require('soda');

var browser = soda.createClient({
    host: 'localhost'
  , port: 4444
  , url: 'http://www.google.com'
  , browser: 'firefox'
});

browser.on('command', function(cmd, args){
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', '));
});

browser
  .chain
  .session()
  .open('/')
  .type('q', 'Hello World')
  .end(function(err){
    browser.testComplete(function() {
      console.log('done');
      if(err) throw err;
    });
  });
```

Фактически, код воспринимается на интуитивном уровне. Сначала вы создаете объект браузера, указывая браузер, имя хоста, порт и веб-сайт, к которому будет отправлен запрос. Начинается новый сеанс браузера, загружается веб-страница (`'/'`) и в поле ввода набирается фраза с заданным идентификатором `q`. Когда работа завершится, с помощью `console.log` на консоль выводится сообщение `done` и вбрасывается любая ошибка, если таковая случается.

Для запуска Soda-приложения нужно установить Java-интерпретатор. Затем нужно скопировать в систему Java-файл Selenium RC (с расширением .jar) и запустить его на выполнение:

```
java -jar selenium.jar
```

Приложение рассчитывает, что браузер Firefox уже установлен, поскольку именно этот браузер указан в приложении. Так как на моем компьютере, работающем под управлением Linux, его не было, я все делал на ноутбуке под управлением Windows и смог без труда запустить приложение. Все получилось довольно впечатляюще, вот только сбило с толку появление и внезапное исчезновение окна, когда приложение Selenium RC занималось своими делами.

Другим подходом может быть использование службы Sauce Labs в качестве удаленной среды тестирования с указанием браузера для заданного теста. Сначала нужно будет создать учетную запись, а затем найти свое имя пользователя учетной записи и ключ к прикладному программному интерфейсу (API). Имя пользователя выводится в верхней панели инструментов, а ключ можно найти на вкладке Account (Учетная запись) после щелчка на ссылке View my API Key (Показать мой API-ключ). В этом окне можно также отследить оставшееся количество минут по запросу (OnDemand) и на исследование (Scout) — для тестирования созданного приложения используются минуты режима OnDemand.

Чтобы попробовать провести удаленное тестирование, я разработал простой тест для формы входа в приложение, которую мы создадим в главе 15. В форме входа имеются два текстовых поля и две кнопки. Значениями текстовых полей являются имя пользователя и пароль, а одна из кнопок называется Submit (Отправить). Сценарий тестирования ведет проверку на отказ, а не на успех, поэтому события в нем должны развиваться в следующем порядке:

1. Обращение к веб-приложению (http://examples.burningbird.net:3000).
2. Открытие формы входа (/login).
3. Набор в поле имени пользователя `Sally`.
4. Набор в поле пароля `badpassword`.
5. Страница должна вывести на экран сообщение Invalid Password (Неправильный пароль).

Все эти действия представлены в коде листинга 14.3.

Листинг 14.3. Вариант тестирования для формы входа в приложение с неправильным паролем

```
var soda = require('soda');

var browser = soda.createSauceClient({
    'url': 'http://examples.burningbird.net:3000/'
  , 'username': 'your username'
  , 'access-key': 'your access key'
  , 'os': 'Linux'
  , 'browser': 'firefox'
  , 'browser-version': '3.'
  , 'max-duration': 300 // 5 минут
});

// Протоколирование команд по мере их выполнения
```

```
browser.on('command', function(cmd, args){
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', '));
});

browser
  .chain
  .session()
  .setTimeout(8000)
  .open('/login')
  .waitForPageToLoad(5000)
  .type('username', 'Sally')
  .type('password', 'badpassword')
  .clickAndWait('//input[@value="Submit"]')
  .assertTextPresent('Invalid password')
  .end(function(err){
    browser.setContext('sauce:job-info={"passed": ' + (err === null) + '}',
                                                    function(){
      browser.testComplete(function(){
        console.log(browser.jobUrl);
        console.log(browser.videoUrl);
        console.log(browser.logUrl);
        if (err) throw err;
      });
    });
  });
```

В тестовом приложении создается объект браузера с заданным браузером, версией браузера и операционной системой, в данном случае — Firefox 3.x под управлением Linux. Обратите внимание также на другой клиент браузера — это `soda.createSauceClient`, а не `soda.createClient`. В объекте браузера я ограничил время тестирования пятью минутами, а обращение к сайту производится по адресу http://examples.burningbird.net:3000. Где взять имя пользователя и API-ключ, мы уже обсуждали.

При выполнении каждой команды она протоколируется. Чтобы проверять ответы и находить отказы и отклонения, нам нужен журнал:

```
// Протоколирование команд по мере их выполнения
browser.on('command', function(cmd, args){
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', '));
});
```

Все остальное является самим тестом. Обычно тесты должны быть вложенными функциями обратного вызова (поскольку это асинхронная среда), но Soda предоставляет *цепочечный получатель* (chain getter), существенно упрощающий добавление заданий. Самое первое задание предназначено для запуска нового сеанса, затем следует код каждого пункта сценария тестирования. В конце приложение выводит URL-адреса для задания, журнала и видео тестирования.

Данные, выведенные после запуска приложения, имеют следующий вид:

```
 setTimeout: 8000
 open: /login
 waitForPageToLoad: 5000
 type: username, Sally
 type: password, badpassword
 clickAndWait: //input[@value="Submit"]
 assertTextPresent: Invalid password
 setContext: sauce:job-info={"passed": true}
 testComplete:
https://saucelabs.com/jobs/d709199180674dc68ec6338f8b86f5d6
https://saucelabs.com/rest/shelleyjust/jobs/d709199180674dc68ec6338f8b86f5d6/
results/video.flv
https://saucelabs.com/rest/shelleyjust/jobs/d709199180674dc68ec6338f8b86f5d6/
results/selenium-server.log
```

Можно получить непосредственный доступ к результатам или, как показано на рис. 14.2, войти в службу Sauce Labs и посмотреть результаты всех своих тестов.

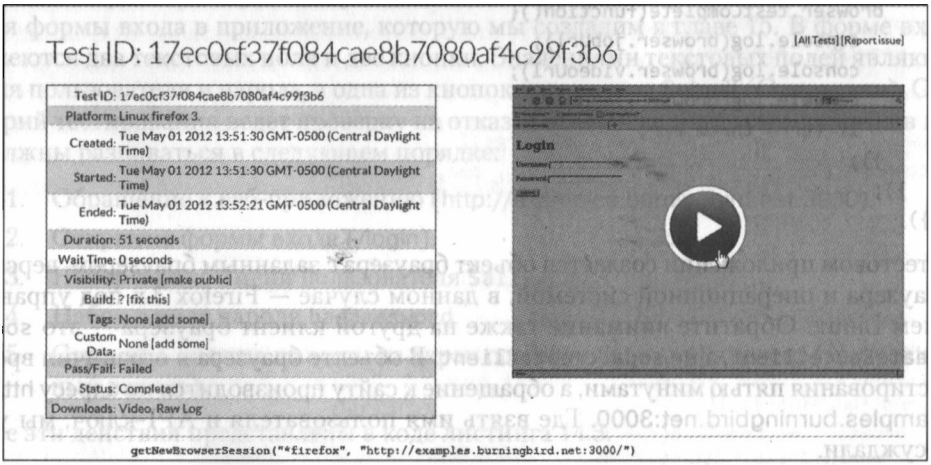


Рис. 14.2. Результаты запуска Soda-теста на ядре Sauce Labs Selenium

Как уже упоминалось, Soda является оболочкой Selenium, поэтому в модуле имеется небольшая документация по Selenium-командам. Вам нужно найти их на веб-сайте Selenium и экстраполировать для работы с Soda.

Веб-сайт Selenium находится по адресу http://seleniumhq.org/.

Эмуляция браузера с помощью Tobi и Zombie

Вместо конкретного браузера при приемочном тестировании можно использовать Node-модуль, эмулирующий браузер. Для этого служат модули Tobi и Zombie. Главным преимуществом этих модулей является то, что вы можете запускать

приложения в среде, не имеющей установленного браузера. В этом разделе я кратко познакомлю вас с порядком применения Zombie для приемочного испытания.

Сначала Zombie нужно установить с помощью диспетчера Node-пакетов:

```
npm install zombie
```

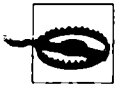
Модуль Zombie похож на Soda тем, что сначала вы создаете браузер, а потом запускаете тесты, эмулирующие действия пользователя в браузере. Он даже поддерживает выстроенные в цепочку методы, позволяющие обойти проблемы, связанные с вложенными функциями обратного вызова.

Я переделал под Zombie вариант тестирования формы входа в листинге 14.3, но на этот раз в тесте используется правильный пароль, к тому же тестируется удачное, а не неудачное завершение (пользователь перенаправляется на страницу /admin). Код этого приемочного теста показан в листинге 14.4.

Листинг 14.4. Тестирование формы входа с помощью Zombie

```
var Browser = require('zombie');
var assert = require('assert');

var browser = new Browser();

browser.visit('http://examples.burningbird.net:3000/login', function() {
   browser.
   fill('username', 'Sally').
   fill('password', 'apple').
   pressButton('Submit', function() {
      assert.equal(browser.location.pathname, '/admin');
  });
});
```

Результатом тестирования является молчание, если утверждение в конце оказывается верным. В этом случае браузеру указывается место /admin, где находится страница, которая должна открыться, если вход оказывается корректным, сигнализируя об успешном прохождении теста.



Несколько примеров зависят от популярного Node-модуля jsdom. Но и у этого модуля есть некоторые проблемы из-за нестабильной сборки Node 0.7.10, но надеюсь, что в Node 0.8.x проблема будет решена.

Тестирование производительности: сравнительные и нагрузочные тесты

Надежное приложение, отвечающее всем требованиям пользователя, но имеющее неважную производительность, обречено на недолгую жизнь. Нам нужна возможность *тестирования производительности* наших Node-приложений, особенно

если повысить производительность можно путем настройки. Мы не можем просто настроить приложение и передать его в производство, возложив решение проблем производительности на пользователей.

Тестирование производительности состоит из *сравнительного* (benchmark testing) и *нагрузочного тестирования* (load testing). При *сравнительном тестировании* запускается несколько версий приложения, а затем определяется лучшая. Это довольно эффективное средство, применяемое при настройке приложения с целью повышения его эффективности и масштабируемости. Создается стандартный тест, который запускается для различных версий, а затем проводится анализ результатов.

В то же время *нагрузочное тестирование* приложения, как правило, проводится в экстремальных условиях. Это попытка определить момент, когда ваше приложение начнет сбоить или сильно замедляться из-за чрезмерного запроса ресурсов или слишком большого количества одновременно обращающихся к нему пользователей. Обычно пытаются довести приложение до отказа, который и является признаком успешного нагрузочного тестирования.

Из существующих средств, поддерживающих оба вида тестирования производительности, одним из наиболее популярных является ApacheBench. Популярность этого инструмента обусловлена доступностью на любом сервере, на котором установлена платформа Apache (а она установлена почти повсеместно). Кроме того, это средство тестирования отличается простотой в использовании и эффективностью при небольшом объеме кода. Когда я пытался понять, что лучше, установить статическое соединение с базой данных для его многократного использования или каждый раз заново устанавливать и разрывать соединение, то выяснял это с помощью ApacheBench-тестов.

ApacheBench работает с веб-приложениями, следовательно, вы предоставляете не имя приложения, а его URL-адрес. Если предпочтение отдается Node-решению или приложению, которое само может запускать приложения (а не только посылать запросы на веб-сайты), есть еще один модуль, запускаемый из командной строки — Nodeload. Это комбинированное средство может взаимодействовать с модулем stats, графически представлять результаты и обеспечивать мониторинг в реальном времени. Он также поддерживает распределенное нагрузочное тестирование.

В следующих двух разделах тестовые приложения работают с Redis, поэтому если вы еще не читали главу 9, то можете сделать это сейчас.

Сравнительное тестирование с помощью ApacheBench

Модуль ApacheBench обычно называют сокращенно ab, и с этого момента и далее я буду использовать именно это имя. Модуль ab является инструментом командной строки, который позволяет указать количество запусков приложения и количество одновременно работающих с ним приложений. Если нужно эмулировать одновременную работу 20 пользователей, обращающихся к приложению в общей сложности 100 раз, нужно воспользоваться следующей командной строкой:

```
ab -n 100 -c 20 http://somewebsite.com/
```

Завершающий слэш нужен обязательно, поскольку ab ожидает полного URL-адреса, включая путь.

Модуль ab предоставляет на выходе довольно богатую информацию. В качестве примера посмотрите на выходные данные одного теста (из этих данных исключен идентификатор используемого средства):

```
Concurrency Level:      10
Time taken for tests:   20.769 seconds
Complete requests:      15000
Failed requests:        0
Write errors:           0
Total transferred:      915000 bytes
HTML transferred:       345000 bytes
Requests per second:    722.22 [#/sec] (mean)
Time per request:       13.846 [ms] (mean)
Time per request:       1.385 [ms] (mean, across all concurrent requests)
Transfer rate:          43.02 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       4
Processing:     1   14  15.7     12     283
Waiting:        1   14  15.7     12     283
Total:          1   14  15.7     12     283

Percentage of the requests served within a certain time (ms)
  50%     12
  66%     14
  75%     15
  80%     16
  90%     18
  95%     20
  98%     24
  99%     40
 100%    283 (longest request)
```

Тест запускался 15 000 раз при одновременной работе 10 пользователей.

Строки, представляющие наибольший интерес (выделенные полужирным шрифтом), относятся ко времени прохождения каждого теста и к совокупному распределению в конце теста (основанному на процентных показателях). Согласно этому выводу, среднее время на запрос (первое значение с меткой `average time per request`) составило 13,846 миллисекунды. Это продолжительность ожидания ответа среднему пользователю. Вторая строка связана с пропускной способностью и, наверное, не так полезна как первая.

Совокупное распределение дает неплохой взгляд на процентные показатели запросов, обработанных в течение определенного периода времени. В нем опять показано то, что мы можем ожидать в отношении среднего пользователя: время ответа между

12 и 283 миллисекундами при том, что подавляющее большинство ответов обработано за 20 миллисекунд и менее.

Последнее интересующее нас значение относится к количеству запросов в секунду (`requests per second`), в данном случае оно составляет 722,22. Это значение позволяет в какой-то мере выстроить прогноз, насколько хорошо приложение будет масштабироваться, поскольку оно дает нам представление о максимальном количестве запросов в секунду, то есть о верхней границе обращений к приложению. Тест нужно запускать в разное время и с разной дополнительной нагрузкой, особенно если тестируется система, обслуживающая других пользователей.

Тестируемое приложение состоит из веб-сервера, слушающего запросы. Каждый запрос инициирует обращение к хранилищу данных Redis. Приложение создает постоянное соединение с хранилищем данных на время всего жизненного цикла Node-приложения. Код приложения показан в листинге 14.5.

Листинг 14.5. Простое приложение для доступа к хранилищу данных Redis, используемое для тестирования постоянного соединения с Redis

```
var redis = require("redis"),
    http = require('http');

// создание Redis-клиента
var client = redis.createClient();

client.on('error', function (err) {
    console.log('ERROR ' + err);
});

// установка на первую базу данных
client.select(1);

var scoreServer = http.createServer();

// прослушивание входящих запросов
scoreServer.on('request', function (req, res) {

  console.time('test');
  req.addListener("end", function() {

     var obj = {
          member : 2366,
          game : 'debiggame',
          first_name : 'Sally',
          last_name : 'Smith',
          email : 'sally@smith.com',
          score : 50000 };

     // добавление или перезапись показателя
     client.hset(obj.member, "game", obj.game, redis.print);
     client.hset(obj.member, "first_name", obj.first_name, redis.print);
```

```
      client.hset(obj.member, "last_name", obj.last_name, redis.print);
      client.hset(obj.member, "email", obj.email, redis.print);
      client.hset(obj.member, "score", obj.score, redis.print);

      client.hvals(obj.member, function (err, replies) {
         if (err) {
            return console.error("error response - " + err);
         }

         console.log(replies.length + " replies:");
         replies.forEach(function (reply, i) {
           console.log("    " + i + ": " + reply);
         });
      });

      res.end(obj.member + ' set score of ' + obj.score);
      console.timeEnd('test');
   });
});

scoreServer.listen(8124);

// HTTP-сервер закрывается, клиентское соединение разрывается
scoreServer.on('close', function() {
   client.quit();
});

console.log('listening on 8124');
```

Мне было интересно, как изменится производительность, если я изменю один параметр в приложении: перейду от поддержки постоянного подключения к Redis к установлению соединения при обращении к веб-службе с разрывом соединения сразу после обработки запроса. В результате получилась вторая версия приложения, код которой показан в листинге 14.6. Изменения, внесенные в код первой версии, выделены полужирным шрифтом.

Листинг 14.6. Измененное приложение с непостоянным подключением к Redis

```
var redis = require("redis"),
    http = require('http');

var scoreServer = http.createServer();

// прослушивание входящих запросов
scoreServer.on('request', function (req, res) {

  console.time('test');

  // создание Redis-клиента
  var client = redis.createClient();
```

продолжение ⇗

Листинг 14.6 *(продолжение)*

```
  client.on('error', function (err) {
    console.log('ERROR ' + err);
  });
  // установка на первую базу данных
  client.select(1);

  req.addListener("end", function() {

     var obj = {
         member : 2366,
         game : 'debiggame',
         first_name : 'Sally',
         last_name : 'Smith',
         email : 'sally@smith.com',
         score : 50000 };

     // добавление или перезапись показателя
     client.hset(obj.member, "game", obj.game, redis.print);
     client.hset(obj.member, "first_name", obj.first_name, redis.print);
     client.hset(obj.member, "last_name", obj.last_name, redis.print);
     client.hset(obj.member, "email", obj.email, redis.print);
     client.hset(obj.member, "score", obj.score, redis.print);

     client.hvals(obj.member, function (err, replies) {
        if (err) {
           return console.error("error response - " + err);
        }

        console.log(replies.length + " replies:");
        replies.forEach(function (reply, i) {
          console.log("    " + i + ": " + reply);
        });
     });

     res.end(obj.member + ' set score of ' + obj.score);
     client.quit();
     console.timeEnd('test');
   });
});

scoreServer.listen(8124);

console.log('listening on 8124');
```

Я запустил ab-тест для второго приложения и получил следующие результаты:

```
Requests per second:    515.40 [#/sec] (mean)
Time per request:       19.402 [ms] (mean)
...
Percentage of the requests served within a certain time (ms)
```

```
  50%     18
  66%     20
  75%     21
  80%     22
  90%     24
  95%     27
  98%     33
  99%     40
 100%    341 (longest request)
```

Тест довольно убедительно показал, что при наличии постоянного соединения производительность выше. Это стало еще нагляднее при проведении второго теста.

Когда я запустил тест 100 000 раз при одновременном обслуживании 1000 пользователей, Node-приложение, поддерживающее постоянное соединение с Redis, успешно завершило тест, в то время как другие варианты дали сбой. Хранилище Redis стало узким местом для слишком большого количества одновременно обслуживаемых пользователей, и оно начало отклонять подключения. До того как произошел сбой приложения, прошло лишь 67 985 тестов.

Проведение нагрузочного тестирования с помощью Nodeload

Комбинированное средство Nodeload предоставляет инструментарий командной строки, поддерживающий те же типы тестирования, что и ab, но с добавлением красивой графики при выводе результатов. Доступен также модуль, который может использоваться для разработки собственных приложений для тестирования.

Есть еще одно приложение, также известное под именем Nodeload, но отвечающее за создание и доставку Git-репозиториев в виде .zip-файлов. Чтобы гарантировать установку именно того модуля Nodeload, который вам нужен, воспользуйтесь следующей командой:

npm install nodeload -g

При глобальной установке Nodeload доступ к версии (nl.js), работающей в командной строке прикладного модуля, можно получить из любого места. Используемые аргументы командной строки подобны тем, что использовались с ab:

```
nl.js -c 10 -n 10000 -i 2 http://examples.burningbird.net:8124
```

Приложение обращается к веб-сайту 10 000 раз, эмулируя одновременную работу 10 пользователей. Ключ `-i` изменяет частоту выдачи статистики и составления отчета (каждые 2 секунды, вместо 10 секунд по умолчанию). Полный набор ключей выглядит следующим образом:

```
-n --number
```

Количество выдаваемых запросов.

`-c --concurrency`

Количество одновременно обслуживаемых пользователей.

`-t --time-limit`

Ограничение теста по времени.

`-m --method`

Используемый HTTP-метод.

`-d --data`

Данные для отправки вместе с запросом `PUT` или `POST`.

`-r --request-generator`

Путь к модулю для функции `getRequest` (если предоставлена нестандартная функция).

`-q --quiet`

Подавление вывода хода теста на экране.

`-h --help`

Получение справки.

Средство Nodeload вызывает интерес тем, что позволяет в ходе тестирования получать «живую» графику. При обращении к порту 8000 сервера, на котором проводится тест (http://localhost:8000 или через домен), вы можете видеть графические результаты по мере их получения. Снимок экрана в ходе выполнения одного из тестов показан на рис. 14.3.

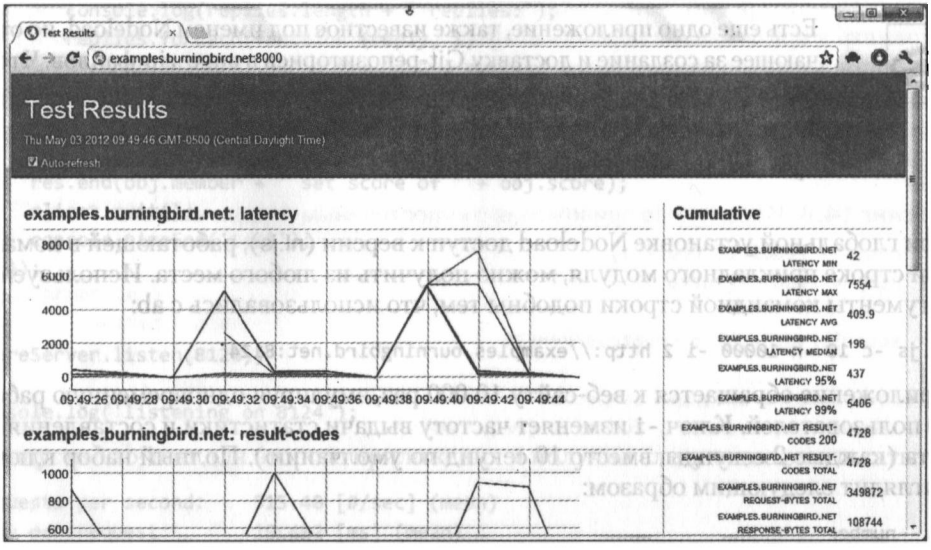
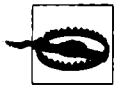


Рис. 14.3. Графика, сопровождающая тест, выполняемый с помощью Nodeload

Графический файл сохраняется для последующего доступа, так же как и файл журнала с результатами теста. В конце теста выдаются итоговые результаты, которые по своей природе очень похожи на результаты работы модуля ab. В качестве примера можно привести следующие результаты:

```
Server:                         examples.burningbird.net:8124
HTTP Method:                    GET
Document Path:                  /
Concurrency Level:              100
Number of requests:             10000
Body bytes transferred:         969977
Elapsed time (s):               19.59
Requests per second:            510.41
Mean time per request (ms):     192.74
Time per request standard deviation:    47.75

Percentages of requests served within a certain time (ms)
  Min: 23
  Avg: 192.7
  50%: 191
  95%: 261
  99%: 372
  Max: 452
```

Если вы хотите предложить собственный нестандартный тест, для разработки тестирующего приложения можете воспользоваться модулем Nodeload. Этот модуль обеспечивает мониторинг в реальном масштабе времени, графические инструменты, статистику, а также возможность распределенного тестирования.



В настоящее время в Nodeload используется метод http.createClient, вместо которого в Node 0.8.x предлагается метод http.request.

Обновление кода с помощью Nodemon

Перед тем как завершить эту главу, я хочу представить вам модуль Nodemon. С технической точки зрения он не имеет никакого отношения ни к тестированию, ни к отладке, но может быть полезен при разработке.

Сначала его нужно установить с помощью диспетчера Node-пакетов:

```
npm install nodemon
```

Nodemon является оболочкой для вашего приложения. Вместо Node используйте для запуска приложения Nodemon:

```
nodemon app.js
```

Nodemon, ничем себя не проявляя, отслеживает каталог (и все содержащиеся в нем каталоги), из которого запущено приложение, реагируя на изменения в файлах. Если найдено изменение, он перезапускает приложение, чтобы в нем учитывались последние изменения.

Приложению можно передать параметры:

```
nodemon app.js param1 param2
```

Можно также воспользоваться модулем CoffeeScript:

```
nodemon someapp.coffee
```

Если нужно, чтобы модуль Nodemon отслеживал не текущий, а какой-нибудь другой каталог, следуют воспользоваться ключом `--watch`:

```
nodemon --watch dir1 --watch libs app.js
```

Описания других ключей имеются в модуле, его можно найти по адресу https://github.com/remy/nodemon/.

Об использовании Nodemon с Forever рассказывается в главе 16, где этот модуль перезапускает приложение, если оно по какой-то причине закрылось.

15 Стражи ворот

Для безопасности веб-приложений мало сделать недоступным сервер приложений. Безопасность должна быть комплексной и даже несколько устрашающей. К счастью, когда речь заходит о Node-приложениях, оказывается, что большинство компонентов, необходимых для обеспечения безопасности, уже создано. Их только нужно подключить в нужное место и в нужное время.

В этой главе рассматриваются четыре основных компонента безопасности: шифрование, аутентификация и авторизация, защита от атак и использование «песочниц»:

Шифрование

Обеспечивает безопасность передаваемых по Интернету данных даже при их перехвате на маршруте следования. Единственным получателем, способным расшифровать данные, является система, имеющая нужные учетные сведения (как правило, ключ). Шифрование применяется также к данным, которые должны храниться в режиме конфиденциальности.

Аутентификация и авторизация

Подразумевает необходимость регистрации для доступа к защищенным областям приложения. Подобная регистрация гарантирует не только то, что данная персона имеет доступ к данному разделу (авторизация), но и то, что это именно та персона, за которую она себя выдает (аутентификация).

Защита от атак

Гарантирует, что при передаче данных через форму не будет предпринята попытка дополнить текст тем, что способно осуществить атаку на используемый вами сервер или базу данных.

Использование «песочниц»

Ограждает сценарий таким образом, чтобы у него не было доступа к системным ресурсам и он работал только в ограниченном контексте.

Шифрование данных

Мы передаем через Интернет большие объемы данных. Большая часть из них не представляет особой важности, включая новые твиты, записи на веб-страницах, посты в блоге. Однако многие данные носят закрытый характер, к ним можно отнести данные кредитных карт, конфиденциальные сообщения электронной почты или информацию, необходимую для входа на наши серверы. Единственным способом обеспечения закрытости при передаче подобных данных и их защиты от взлома в процессе передачи является шифрование.

Настройка TSL/SSL

Безопасная связь высокой стойкости между клиентом и сервером осуществляется по протоколу SSL (Secure Sockets Layer — слой защищенных сокетов) и его улучшенной версии — протоколу TLS (Transport Layer Security — безопасность транспортного уровня). TSL/SSL предоставляет исходное шифрование для протокола HTTPS, который рассматривается в следующем разделе. Однако перед использованием HTTPS нужно задать ряд параметров окружения.

Обмен данными по протоколу TSL/SSL требует *квитирования*, то есть подтверждения факта установления связи между клиентом и сервером. В ходе квитирования клиент (как правило, браузер) дает знать серверу, какой тип функций безопасности он поддерживает. Сервер выбирает функцию, а затем отправляет через SSL *сертификат*, который включает в себя открытый ключ. Клиент подтверждает сертификат, генерирует случайное число, используя ключ сервера, и отправляет его обратно на сервер. Затем сервер использует закрытый ключ для расшифровки числа, которое, в свою очередь, служит для организации безопасного обмена данными.

Чтобы все это работало, вам нужно генерировать как открытый, так и закрытый ключи, кроме того, нужен сертификат. Для производственной системы сертификат должен быть подписан *доверенной организацией*, например вашим регистратором доменов, но для целей разработки подойдет сертификат *с вашей собственной подписью*. Это приведет к появлению в браузере строгих предупреждений, но поскольку сайт разработки не посещается пользователями, это не станет проблемой.

Инструментом для создания нужных файлов является пакет OpenSSL. Если используется Linux, этот пакет уже должен быть установлен. Существует двоичный код для установки под Windows, а Apple придерживается собственной библиотеки Crypto. В этом разделе рассматривается только вариант настройки среды под Linux.

Сначала нужно набрать в командной строке следующую команду:

```
openssl genrsa -des3 -out site.key 1024
```

Эта команда приведет к созданию закрытого ключа, зашифрованного по алгоритму Triple-DES, и сохранению его в формате PEM (Privacy-Enhanced Mail — почта с улучшенной защитой), что позволяет читать его в кодировке ASCII.

У вас будет запрошен пароль, который понадобится для решения следующей задачи — создания запроса на подпись сертификата (Certificate-Signing Request, CSR).

При генерировании CSR у вас будет запрошен только что созданный пароль. Также будут заданы различные вопросы, включая код страны (например, US для United States), название штата или области, название города, название компании и организации, адрес электронной почты. Наиболее важным будет вопрос стандартного имени (common name). Этот вопрос касается имени хоста сайта, например burningbird.net или yourcompany.com. Вам следует предоставить имя хоста, на котором обслуживается приложение. В моем примере я создал сертификат для хоста examples.burningbird.net.

```
openssl req -new -key site.key -out site.csr
```

Для закрытого ключа нужна *кодовая фраза* (passphrase). Проблема в том, что при каждом запуске сервера нужно будет предоставлять эту фразу, что для запущенной в эксплуатацию системы может превратиться в проблему. Поэтому следующим действием вы удалите кодовую фразу из ключа. Но сначала нужно переименовать ключ:

```
mv site.key site.key.org
```

Затем наберите следующую команду:

```
openssl rsa -in site.key.org -out site.key
```

Когда кодовая фраза удалена, чтобы обеспечить безопасность сервера, нужно чтобы файл можно было прочитать только с привилегиями администратора.

Следующей задачей является создание сертификата с самостоятельно поставленной подписью. Следующая команда создает сертификат, сохраняющей свое действие только на 365 дней:

```
openssl x509 -req -days 365 -in site.csr -signkey site.key -out final.crt
```

Теперь у вас есть все компоненты, необходимые для использования TLS/SSL и HTTPS.

Использование протокола HTTPS

Всегда лучше, чтобы веб-страницы, запрашивающие регистрационные данные пользователя или информацию о кредитной карте, работали по протоколу HTTPS, в противном случае от контактов с сайтом нужно отказаться. HTTPS является защищенным вариантом протокола HTTP, он объединен с SSL, гарантируя, что мы имеем дело именно с тем сайтом, с которым хотели, что данные шифруются при передаче и поступают неповрежденными и без какого-либо вмешательства.

Добавление поддержки протокола HTTPS напоминает HTTP, но с включением дополнительных объектов, предоставляющих открытый ключ шифрования и подписанный сертификат. Кроме того, у HTTPS-сервера другой исходный порт: HTTP обслуживается по умолчанию через порт 80, а HTTPS — через порт 443.

В листинге 15.1 представлен очень простой HTTPS-сервер. Его задача — всего лишь отправка на браузер варианта нашего традиционного сообщения «Hello, World».

Листинг 15.1. Создание простейшего HTTPS-сервера

```
var   fs = require("fs"),
      https = require("https");

var privateKey = fs.readFileSync('site.key').toString();
var certificate = fs.readFileSync('final.crt').toString();

var options = {
   key: privateKey,
   cert: certificate
};

https.createServer(options, function(req,res) {
   res.writeHead(200);
   res.end("Hello Secure World\n");
}).listen(443);
```

Открытый ключ и сертификат не шифруются, а их содержимое считывается в синхронном режиме. Данные, присоединенные к объекту `options`, передаются в качестве первого параметра методу `https.createServer` method. Функция обратного вызова для того же самого метода нами уже использовалась, ей в качестве параметров передаются серверный запрос и объект ответа.

Обращение к странице показывает, что происходит, когда используется самостоятельно подписанный сертификат. Результат показан на рис. 15.1. Теперь должно быть понятно, почему собственноручно подписанный сертификат может использоваться только в процессе тестирования.

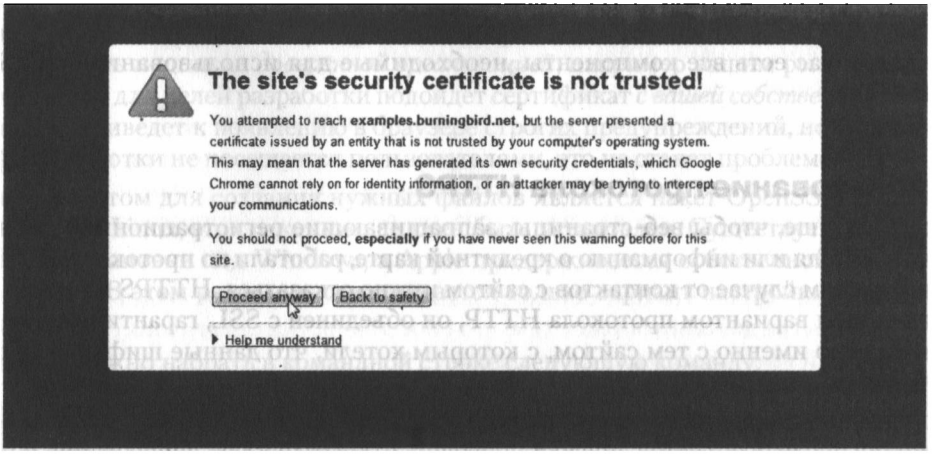


Рис. 15.1. Ситуация, возникающая в браузере Chrome с самостоятельно подписанным сертификатом при обращении к веб-сайту по протоколу HTTPS

В своей адресной строке браузер еще одним способом сигнализирует о том, что сертификат сайта не может пользоваться доверием. Вместо символа закрытого навесного замка, показывающего, что доступ к сайту осуществляется через HTTPS, в ней появляется замок, перечеркнутый крестиком красного цвета и свидетельствующий, что сертификату доверять нельзя. При щелчке на значке раскрывается информационное окно с более подробной информацией о сертификате (рис. 15.2).

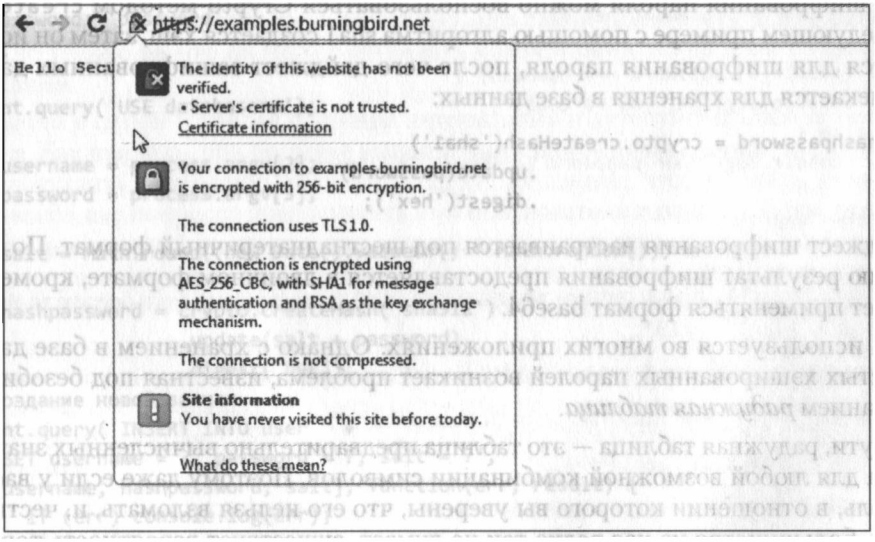


Рис. 15.2. Дополнительная информация о сертификате, появляющаяся при щелчке на значке замка

Шифрованная связь — не единственная область применения шифрования в веб-приложениях. Шифрование можно также применять при сохранении пользовательских паролей и других важных данных.

Безопасное хранение паролей

Node предоставляет модуль шифрования — Crypto. В документации к модулю написано следующее:

> Модуль Crypto требует, чтобы на нижележащей платформе был реализован доступ к OpenSSL. Он предлагает способ инкапсуляции защищаемых свидетельств о полномочиях, используемых как часть безопасной HTTPS-сети или HTTP-соединения.
>
> Им также предлагается набор оболочек для OpenSSL-методов `hash`, `hmac`, `cipher`, `decipher`, `sign` и `verify`.

Нас в этом модуле интересует поддержка OpenSSL-хэша.

Одна из самых распространенных задач, которые должно решать веб-приложение, является также одной из наиболее уязвимых — речь идет о регистрационных

данных пользователя, включая пароль. При хранении в виде простого текста такой важной информации, как имя пользователя и пароль, хватит, пожалуй, пяти минут, чтобы взломать сайт, получить регистрационную информацию и сделать с ней все, что угодно.

К счастью, с Node-модулем Crypto хранить пароль в виде простого текста вам не нужно.

Для шифрования пароля можно воспользоваться Crypto-методом `createHash`. В следующем примере с помощью алгоритма sha1 создается хэш, затем он используется для шифрования пароля, после чего дайджест зашифрованных данных извлекается для хранения в базе данных:

```
var hashpassword = crypto.createHash('sha1')
                                .update(password)
                                .digest('hex');
```

Дайджест шифрования настраивается под шестнадцатеричный формат. По умолчанию результат шифрования предоставляется в двоичном формате, кроме того, может применяться формат base64.

Хэш используется во многих приложениях. Однако с хранением в базе данных простых хэшированных паролей возникает проблема, известная под безобидным названием *радужная таблица*.

По сути, радужная таблица — это таблица предварительно вычисленных значений хэша для любой возможной комбинации символов. Поэтому даже если у вас есть пароль, в отношении которого вы уверены, что его нельзя взломать, и, честно говоря, большинство из нас редко так не думает, существует вероятность того, что нужная последовательность символов имеется где-то в радужной таблице, что существенно упрощает взлом вашего пароля.

Обойти проблему радужной таблицы помогает *соль* — это естественно, не кристаллический порошок, который мы добавляем в пищу, а уникальное сгенерированное значение, объединяемое с паролем перед шифрованием. Это может быть единое значение, используемое со всеми паролями и хранящееся в безопасном месте на сервере, но лучше генерировать уникальную соль для каждого пароля, а затем сохранять ее вместе с паролем. Правда, соль также может быть украдена вместе с паролем, но она все же потребует от злоумышленника, пытающегося взломать пароль, создания радужной таблицы только для одного пароля, что серьезно усложнит взлом любого отдельно взятого пароля.

В листинге 15.2 показано простое приложение, которое в качестве аргументов командной строки принимает имя пользователя и пароль, шифрует пароль, а затем сохраняет оба элемента в качестве данных нового пользователя в таблице базы данных MySQL. Таблица создается с помощью следующей SQL-инструкции:

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT, PRIMARY KEY(userid),
username VARCHAR(400) NOT NULL, password VARCHAR(400) NOT NULL);
```

Соль состоит из значения даты, умноженного на случайное число с округлением. Она объединяется с паролем перед шифрованием получившейся в результате строки. Все пользовательские данные затем вставляются в MySQL-таблицу `user`.

Листинг 15.2. Использование Crypto-метода createHash и соли для шифрования пароля

```
var mysql = require('mysql'),
    crypto = require('crypto');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });

client.query('USE databasenm');

var username = process.argv[2];
var password = process.argv[3];

var salt = Math.round((new Date().valueOf() * Math.random())) + '';

var hashpassword = crypto.createHash('sha512')
                   .update(salt + password)
                   .digest('hex');
// создание новой записи
client.query('INSERT INTO user ' +
   'SET username = ?, password = ?, salt = ?',
   [username, hashpassword, salt], function(err, result) {
      if (err) console.log(err);
      client.end();
});
```

Приложение для тестирования имени пользователя и пароля показано в листинге 15.3. Это приложение на основе имени пользователя запрашивает у базы данных пароль и соль. Приложение опять использует соль для шифрования пароля. Как только пароль зашифрован, он сравнивается с паролем, хранящимся в базе данных. Если два варианта пароля не совпадают, пользователь проверку не проходит. Если они совпадают, пользователь входит в систему.

Листинг 15.3. Проверка имени пользователя и зашифрованного пароля

```
var mysql = require('mysql'),
    crypto = require('crypto');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });

client.query('USE databasenm');

var username = process.argv[2];
var password = process.argv[3];
```

продолжение ⇩

Листинг 15.3 *(продолжение)*

```
client.query('SELECT password, salt FROM user WHERE username = ?',
   [username], function(err, result, fields) {
   if (err) return console.log(err);

   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password)
                 .digest('hex');

   if (result[0].password === newhash) {
      console.log("OK, you're cool.");
   } else {
      console.log("Your password is wrong. Try again.");
   }
   client.end();
});
```

Пробуя запустить приложение, мы сначала передаем ему имя пользователя `Michael` с паролем `applef*rk13*`:

```
node password.js Michael apple*frk13*
```

Затем мы проверяем такие же имя пользователя и пароль:

```
node check.js Michael apple*frk13*
```

И получаем вполне ожидаемый результат (OK, все верно):

```
OK, you're cool.
```

Пробуем еще раз, но уже с другим паролем:

```
node check.js Michael badstuff
```

Теперь мы также получаем вполне ожидаемый результат (введен неверный пароль, повторите попытку):

```
Your password is wrong. Try again.
```

Разумеется, мы не ждем, что наши пользователи будут входить в приложение через командную строку. Кроме того, мы не всегда используем локальные парольные системы для аутентификации людей. Процедура аутентификации рассматривается в следующем разделе.

Аутентификация и авторизация с помощью модуля Passport

Являетесь ли вы тем лицом, за которое себя выдаете? Есть ли у вас полномочия для проведения данного действия? Может ли это действие причинить вред? Ответы на эти вопросы возлагаются на два разных технических компонента: аутентификации и авторизации.

Аутентификация призвана гарантировать, что вы тот, за кого себя выдаете. Когда в Твиттере к учетной записи прикрепляется флажок верификации, он свидетельствует о том, что отмеченная флажком персона является подлинной. В то же время *авторизация* призвана гарантировать, что вы получите доступ только к тому, к чему вам нужно. Например, из одиннадцати пользователей сайта Drupal только у половины может быть право публиковать комментарии, у пяти остальных — право публиковать статьи и оставлять комментарии, и только у одного — абсолютно все права. Сайту может быть все равно, кто такой пользователь `Big Daddy`, ему важно только то, что он может оставлять комментарии, но не может удалять публикации.

Обычно в одной и той же функции авторизация и аутентификация не объединяются. Как правило, при попытке выполнения какого-либо действия вас заставляют предоставить какие-либо аутентификационные данные, чтобы понять, кто вы. Скорее всего, вас попросят предоставить имя пользователя и пароль. Затем, после подтверждения вашей личности, приложение продолжит ограничивать ваши действия: лицо, идентифицированное по вашему имени пользователя, может получать доступ только к конкретным страницам или выполнять только конкретные операции.

Иногда аутентификация осуществляется с участием третьей стороны. Примером может послужить использование OpenID. Вместо того чтобы заставлять ваших пользователей создавать на вашем сайте имя пользователя и пароль, вы проводите их аутентификацию с помощью OpenID, а затем предоставляете им доступ к приложению.

Бывает, что и аутентификация, и авторизация осуществляются на сайте третьей стороны. Например, если приложение хочет получить доступ к учетной записи Твиттера или Фейсбука, либо для публикации сообщения, либо для получения информации, пользователи должны пройти аутентификацию на этих сайтах, а затем ваше приложение должно быть авторизовано для получения доступа. Эта авторизация осуществляется посредством другой стратегии, которая называется OAuth.

Функциональность всех этих сценариев может быть реализована с помощью модуля Passport и одной или нескольких Passport-стратегий.

Passport — не единственный модуль, поддерживающий аутентификацию и авторизацию, но я считаю его самым простым в использовании.

Стратегии авторизации и аутентификации: OAuth, OpenID, верификация имени пользователя и пароля

Давайте более пристально присмотримся к нашим трем стратегиям авторизации и аутентификации.

При обращении к административному разделу такой системы управления контентом (Content Management System, CMS), как Drupal, или такого интерактивного сайта, как Amazon, выполняется проверка учетных данных (credential verification). Вы предоставляете имя пользователя и пароль, и они проходят проверку на сайте

перед тем, как вам будет предоставлен доступ. Это по-прежнему наиболее распространенная стратегия авторизации и аутентификации. И в основном она не утратила свою эффективность.

Ранее в данной главе я уже показывал, как защитить пароль в базе данных. Даже при несанкционированном доступе к системе пользователя похитители данных не получат доступ к вашему паролю в виде простого текста. Разумеется, они могут взломать ваш пароль, но если для него используется комбинация букв, символов и цифр, это действие в некоторой степени потеряет смысл, поскольку взлом пароля потребует много времени и мощности центрального процессора.

Протокол OAuth обеспечивает доступ к данным, например к чей-нибудь учетной записи на Твиттере, без предоставления непосредственного доступа к паролю учетной записи. Этот вариант авторизации доступа к данным не требует хранить персональные учетные данные в различных местах, что повышает вероятность несанкционированного доступа к ним. Кроме того, пользователь получает более жесткий контроль, потому что он в большинстве случаев в любое время может отменить авторизацию из своей главной учетной записи.

Практически, OAuth служит только для авторизации доступа к данным. В отличие от него основным назначением OpenID является аутентификация, хотя авторизация действительно идет с ней «рука об руку».

Протокол OpenID распространен не так широко, как OAuth, преимущественно в системах комментариев и регистрации пользователей на различных сайтах средств массовой информации. Одна из проблем систем комментариев заключается в том, что отдельные личности могут выдать себя за других, а способа проверить, что они те, за кого себя выдают, нет. С помощью OpenID человек может подписаться на систему комментариев или зарегистрироваться как пользователь, и OpenID гарантирует, что этот человек прошел аутентификацию, по крайней мере, внутри этой системы.

Кроме того, OpenID поддерживает регистрацию других мест расположения без необходимости создавать другие имя пользователя и пароль для каждого из них. Вы просто предоставляете свой вариант OpenID, он проверяется, и необходимая системе информация берется у провайдера OpenID.

Ни одна из этих трех стратегий не исключает использования двух других. Многие приложения поддерживают все три системы: локальную систему проверки полномочий для решения административных задач, систему OAuth для совместного использования данных или публикации информации на таких сайтах, как Фейсбук и Твиттер, и систему OpenID, позволяющую пользователям регистрироваться и оставлять комментарии.

Существует несколько модулей, поддерживающих все формы аутентификации и авторизации, но я собираюсь сконцентрироваться на одном из них — на модуле Passport. Этот модуль является связующим программным обеспечением, которое работает как с Connect, так и с Express, обеспечивая и аутентификацию, и авторизацию. Его можно установить с помощью диспетчера Node-пакетов:

```
npm install passport
```

Реализуемые модулем Passport стратегии не зависят от среды. У всех стратегий модуля Passport имеются одни и те же основные требования:

- Стратегия должна быть установлена.
- Стратегия должна быть настроена в приложении.
- В качестве части конфигурации в стратегию включается функция обратного вызова, которая используется для проверки учетных данных пользователя.
- Все стратегии требуют дополнительной работы в зависимости от полномочий, выявленных при их проверке: Фейсбук и Твиттер требуют учетной записи и ключа учетной записи, а локальная стратегия требует базу данных с именами пользователей и паролями.
- Все стратегии требуют локального хранилища данных, в котором имя пользователя, обличенного полномочиями, отображается на имя пользователя приложения.
- Функциональность, предлагаемая модулем Passport, служит для надежного хранения сеанса работы пользователя.

В этой главе рассмотрены две стратегии, реализуемые с помощью модуля Passport: локальная аутентификация и авторизация, а также аутентификация посредством Твиттера с помощью OAuth.

Локальная Passport-стратегия

Модуль локальной Passport-стратегии (passport-local) можно загрузить с помощью диспетчера Node-пакетов:

```
npm install passport-local
```

Passport является связующим модулем и должен быть обозначен в Express-приложении как экземпляр связующего модуля. Подключения модулей passport и passport-local выполняются следующим образом:

```
var express = require('express');
var passport = require('passport');
var localStrategy = require('passport-local').Strategy;
```

Теперь нужно инициализировать связующий модуль Passport:

```
var app = express();

app.configure(function(){
   ...
   app.use(passport.initialize());
   app.use(passport.session());
   ...
});
```

Далее нужно настроить локальную стратегию. Формат конфигурирования локальной стратегии такой же, как у всех остальных стратегий: новый экземпляр

стратегии передается модулю Passport методом use, как это происходило с модулем Express:

```
passport.use(new localStrategy( function (user, password, done) { ... }
```

Модуль passport-local ожидает, что имя пользователя и пароль будут переданы веб-приложению через выведенную форму, а это значения, содержащиеся в полях username и password. Если нужно задействовать два других имени поля, передайте их в виде параметров при создании нового экземпляра стратегии:

```
var options =
 { usernameField : 'appuser',
    passwordField : 'userpass'
  };
passport.use(new localStrategy(options, function(user, password, done) {...}
```

Функция обратного вызова, переданная конструкции стратегии, вызывается после извлечения имени пользователя и пароля из тела запроса. Функция выполняет аутентификацию, возвращая при этом:

- ошибку, если она имела место;
- сообщение о том, что пользователь не прошел аутентификацию, если он ее действительно не прошел;
- объект пользователя, если пользователь прошел аутентификацию.

При попытке пользователя зайти на защищенный сайт Passport отправляет запрос, чтобы определить, авторизован ли он. В следующем коде, когда пользователь пытается получить доступ к конфиденциальной административной странице, вызывается функция ensureAuthenticated для проверки факта его авторизации:

```
app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});
```

Функция ensureAuthenticated проверяет результат вызова метода req.isAuthenticated, который модуль Passport добавил в качестве расширения объекта запроса. Если ответ имеет значение false, пользователь перенаправляется на страницу регистрации:

```
function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/login')
}
```

Чтобы надежно сохранить параметры входа для сеанса работы, Passport предоставляет два метода: serializeUser и deserializeUser. Нам нужно обеспечить данную функциональность в функции обратного вызова, передаваемой этим двум методам. По сути, passport.serializeUser проводит сериализацию пользовательского идентификатора, а passport.deserializeUser задействует этот идентификатор для поиска пользователя в применяемом хранилище данных и возвращает объект со всей пользовательской информацией:

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  ...
});
```

Сериализация сеанса для Passport не требуется. Если вы не хотите выполнять сериализацию данных пользователя, не включайте связующий метод `passport.session`:

```
app.use(passport.session());
```

Если вы решили провести сериализацию данных пользователя (а иначе вы бы получили очень раздраженного пользователя, постоянно получающего запросы на регистрацию), нужно обеспечить, чтобы связующий модуль Passport был включен после связующего программного обеспечения Express-сеанса:

```
app.use(express.cookieParser('keyboard cat'));
app.use(express.session());
app.use(passport.initialize());
app.use(passport.session());
```

Если не обеспечить нужный порядок, пользователь не сможет пройти аутентификацию.

Последняя группа функциональных средств касается случаев, когда пользователь не проходит проверку. Если в ходе аутентификации имя пользователя в хранилище данных не обнаружится, появится сообщение об ошибке. Если имя пользователя будет найдено, но пароль окажется не тем, который хранится, появится сообщение об ошибке. Наша задача — передать эти сообщения об ошибках обратно пользователю.

Для выстраивания очереди сообщений, предназначенных для отправки назад пользователю, модуль Passport использует имеющийся в Express 2.x метод `req.flash`. В предыдущих главах метод `req.flash` не рассматривался, поскольку в Express 3.x его функциональность не была рекомендована к применению. Однако чтобы обеспечить работу модуля Passport с Express версий 2.x и 3.x, разработчики модуля Passport создали новый модуль, connect-flash, который снова поддерживает эту функциональность.

Модуль connect-flash может быть установлен с помощью диспетчера Node-пакетов:

```
npm install connect-flash
```

Чтобы включить модуль connect-flash в приложении, требуется следующая инструкция:

```
var flash = require('connect-flash');
```

А для интеграции с Express в качестве связующего программного обеспечения служит команда:

```
app.use(flash());
```

Теперь в маршруте регистрации POST, если пользователь не прошел аутентификацию, он перенаправляется на форму регистрации и получает уведомление об ошибке:

```
app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login',
                                   failureFlash: true }),
  function(req, res) {
    res.redirect('/admin');
});
```

Сообщение (или сообщения) об ошибке, сгенерированное в ходе аутентификации, может быть передано движку представлений методом `req.flash`, когда выводится форма регистрации:

```
app.get('/login', function(req, res){
  var username = req.user ? req.user.username : '';
  res.render('login', { title: 'authenticate', username: username,
                message: req.flash('error') });
});
```

Затем движок представлений в дополнение к элементам формы регистрации для входа в приложение может вывести сообщение об ошибке, как показано в следующем Jade-шаблоне:

```
extends layout

block content
  h1 Login
  if message
    p= message
  form(method="POST"
       action="/login"
       enctype="application/x-www-form-urlencoded")
    p Username:
        input(type="text"
              name="username"
              id="username"
              size="25"
              value="#{username}"
              required)
    p Password:
        input(type="password"
              name="password"
              id="password"
              size="25"
              required)
      input(type="submit"
            name="submit"
            id="submit"
            value="Submit")
```

```
input(type="reset"
      name="reset"
      id="reset"
      value="reset")
```

Чтобы продемонстрировать все это, я включил выполняющее аутентификацию приложение командной строки из листинга 15.3 в Express-приложение, в котором аутентификация обеспечивается модулем Passport (листинг 15.4). Единственным из маршрутов, поддерживаемых приложением, является маршрут регистрации (login) для вывода формы регистрации, проведения аутентификации и доступа к конфиденциальной административной странице и индексной странице верхнего уровня.

MySQL-код из листинга 15.3 включен непосредственно в процедуру аутентификации (хотя обычно он выделяется во внешнее приложение). Дополнительный MySQL-код доступа служит для поиска пользовательской информации по заданному идентификатору, когда данные пользователя восстановлены из последовательной формы представления.

Листинг 15.4. Объединение хэша пароля, MySQL-таблицы user и механизма аутентификации с помощью Passport в одно Express-приложение

```
// модули
var express = require('express')
  , flash = require('connect-flash')
  , passport = require('passport')
  , LocalStrategy = require('passport-local').Strategy
  , http = require('http');

var mysql = require('mysql')
  , crypto = require('crypto');

// проверка аутентификации пользователя

function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/login')
}

// сериализация пользовательских данных для сеанса
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

// поиск пользователя в базе данных MySQL
passport.deserializeUser(function(id, done) {

   var client = mysql.createClient({
      user : 'username',
      password: 'password'
   });
```

продолжение

Листинг 15.4 *(продолжение)*

```
    client.query('USE databasenm');

    client.query('SELECT username, password FROM user WHERE userid = ?',
              [id], function(err, result, fields) {
        var user = {
            id : id,
            username : result[0].username,
            password : result[0].password};
        done(err, user);
        client.end();
  });
});

// настройка локальной стратегии аутентификации
// на пользовательскую запись в MySQL
passport.use(new LocalStrategy(
  function(username, password, done) {
     var client = mysql.createClient({
        user : 'username',
        password: 'password'
     });

     client.query('USE nodetest2');

     client.query(
        'SELECT userid, password, salt FROM user WHERE username = ?',
        [username], function(err, result, fields) {

         // ошибка базы данных
         if (err) {
            return done(err);

         // имя пользователя не найдено
         } else if (result.length == 0) {
            return done(null, false, {message: ' Unknown user ' +
                                                    username});
         // проверка пароля
         } else {
            var newhash = crypto.createHash('sha512')
                          .update(result[0].salt + password)
                          .digest('hex');

            // если пароль соответствует
            if (result[0].password === newhash) {
               var user = {id : result[0].userid,
                           username : username,
                           password : newhash };
               return done(null, user);
            // или если пароль не соответствует
```

```
            } else {
               return done(null, false, {message: 'Invalid password'});
            }
         }
         client.end();
    });
}));

var app = express();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('keyboard cat'));
  app.use(express.session());
  app.use(passport.initialize());
  app.use(passport.session());
  app.use(flash());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.get('/', function(req, res){
  res.render('index', { title: 'authenticate', user: req.user });
});

app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});

app.get('/login', function(req, res){
  var username = req.user ? req.user.username : '';
  res.render('login', { title: 'authenticate', username: username,
                message: req.flash('error') });
});

app.post('/login',
  passport.authenticate(
     'local', { failureRedirect: '/login', failureFlash: true }),
      function(req, res) {
    res.redirect('/admin');
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

Хотя листинг 15.4 оказался длиннее тех примеров, которые я обычно предпочитаю включать в книгу, вставка заглушек в код примера не дала бы вам реального представления о том, как Passport-компонент работает вместе с ранее рассмотренным компонентом хэширования пароля.

Давайте присмотримся к методу аутентификации. После того как приложение запросило запись пользователя по заданному имени пользователя, оно запускает функцию обратного вызова с ошибкой обращения к базе данных, если произошла ошибка. Если ошибки не было, но имени пользователя не найдено, приложение запускает функцию обратного вызова с именем пользователя, установленным в `false`, чтобы оповестить клиента о том, что такое имя пользователя не было найдено, и предоставляет соответствующее сообщение. Если пользователь найден, но пароль не тот, происходит следующее: для пользователя возвращается значение `false` и генерируется сообщение.

Только при отсутствии ошибки обращения к базе данных, присутствии пользователя в таблице пользователей и соответствии пароля создается объект пользователя, который возвращается через функцию обратного вызова:

```
// ошибка обращения к базе данных
if (err) {
   return done(err);
// имя пользователя на найдено
} else if (result.length == 0) {
   return done(
     null, false, {message: ' Unknown user ' + username});
// проверка пароля
} else {
   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password)
                 .digest('hex');

   // если пароль соответствует
   if (result[0].password === newhash) {
      var user = {id : result[0].userid,
                  username : username,
                  password : newhash };
      return done(null, user);

   // или если пароль не соответствует
   } else {
      return done(null, false, {message: 'Invalid password'});
   }
}
```

Затем этот объект пользователя проходит сериализацию для сеанса, и пользователь получает доступ к административной странице. Пока сеанс не завершится, пользователь будет получать беспрепятственный доступ к административной странице.

Passport-стратегия Твиттера (OAuth)

Вместо сохранения имен пользователей и паролей на локальной машине и реализации аутентификации собственными силами можно воспользоваться сторонней службой, например Твиттером. Это также является способом более тесной интеграции сайта с Твиттером (или с Фейсбуком, или с Google+, или с другим сторонним сайтом).

Passport-аутентификация, использующая Твиттер, поддерживается модулем passport-twitter, который может быть установлен с помощью диспетчера Node-пакетов:

```
npm install passport-twitter
```

Чтобы с помощью OAuth аутентифицировать пользователя через Твиттер, нужно настроить учетную запись разработчика на Твиттере и получить ключ и секрет подписчика. Они используются в приложении для формирования части OAuth-запроса.

При наличии ключа и секрета подписчика они используются вместе с URL-адресом обратного вызова для создания Passport-стратегии Твиттера:

```
passport.use(new TwitterStrategy(
       { consumerKey: TWITTER_CONSUMER_KEY,
         consumerSecret: TWITTER_CONSUMER_SECRET,
         callbackURL:
             "http://examples.burningbird.net:3000/auth/twitter/callback"},
       function(token, tokenSecret,profile,done) {
          findUser(profile.id, function(err,user) {
             console.log(user);
             if (err) return done(err);
             if (user) return done(null, user);
             createUser(profile, token, tokenSecret, function(err, user) {
               return done(err,user);
             });
          })
       })
);
```

Хотя аутентификацию реализует Твиттер, все же вам, скорее всего, понадобится как-то сохранять информацию о пользователе. Во фрагменте кода Passport-стратегии Твиттера обратите внимание на то, что функция обратного вызова передала список из нескольких параметров: `token`, `tokenSecret`, `profile`, а затем последнюю функцию обратного вызова. Твиттер при ответе на запрос аутентификации предоставляет параметры `token` и `tokenSecret`. Далее значения `token` и `tokenSecret` могут использоваться для взаимодействия с отдельной учетной записью Твиттера, например для переиздания последних записей в Твиттере, помещения учетной записи Твиттера в учетную запись пользователя или изучения информации о его списках и читателях. Прикладной программный интерфейс Твиттера предоставляет всю информацию, которую видит сам пользователь при непосредственном взаимодействии с Твиттером.

Однако нас здесь интересует объект профиля. В нем содержится большое количество информации о человеке: его экранное имя в Твиттере, полное имя, описание, место проживания, изображение для аватара, количество последователей, количество тех, чьим последователем является пользователь, количество твиттов и т. д. Именно эти данные нам нужны, чтобы хранить некую значимую информацию о пользователе в нашей локальной базе данных. Пароль мы не сохраняем, поскольку OAuth не предоставляет индивидуальную аутентификационную информацию. Вместо этого мы просто сохраняем информацию, которая может понадобиться в веб-приложениях для ориентации наших сайтов на конкретного человека.

При первой аутентификации пользователя приложение ищет в локальной базе данных его Твиттер-идентификатор. Если идентификатор найден, возвращается объект с информацией, которая хранится о человеке на локальной машине. Если не найден, для человека создается новая запись в базе данных. Для этого процесса служат две функции: `findUser` и `createUser`. Функция `findUser` также вызывается при десериализации пользователя из сеанса:

```
passport.deserializeUser(function(id, done) {
   findUser(id, function(err, user) {
      done(err,user);
   });
});
```

Страницы регистрации больше нет, поскольку форму регистрации предоставляет Твиттер. В приложении регистрация реализуется только посредством ссылки на аутентификацию через Твиттер:

```
extends layout
block content
   h1= title
   p
      a(href='/auth/twitter') Login with Twitter
```

Если человек не зарегистрирован на Твиттере, для него открывается страница регистрации, показанная на рис. 15.3.

Как только пользователь зарегистрируется, веб-страница перенаправляет его в приложение, которое затем выводит для пользователя административную страницу. Но теперь страница персонифицирована данными, извлеченными непосредственно из Твиттера, включая отображаемое имя человека и его аватар:

```
extends layout
block content
  h1 #{title} Administration
  p Welcome to #{user.name}
  p
     img(src='#{user.img}',alt='avatar')
```

Это те данные, которые сохраняются при первой аутентификации человека. Если заглянуть на страницу настройки учетной записи Твиттера и щелкнуть на ссылке Apps (Приложения), можно увидеть приложение среди списка других приложений, как показано на рис. 15.4.

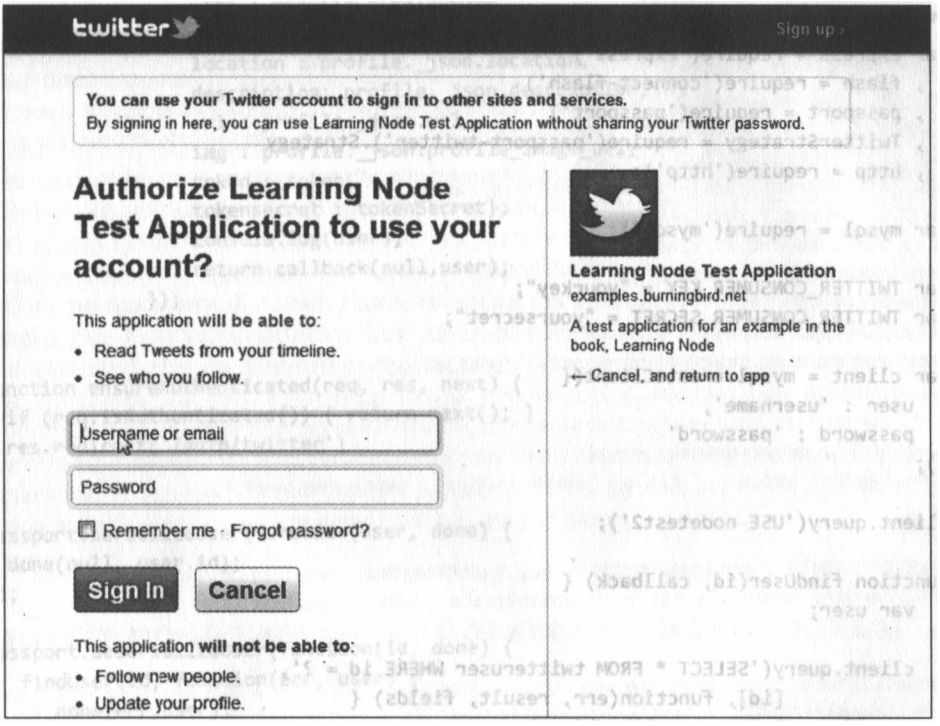


Рис. 15.3. Страница регистрации и авторизации в Твиттере для Node-приложения

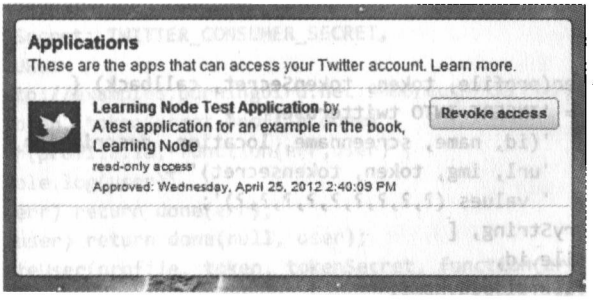


Рис. 15.4. Запись о нашем Node-приложении в окне Твиттера, предназначенном для установки приложений (Apps Settings)

Полный код приложения для аутентификации пользователя через Твиттер и сохранения его данных в базе данных MySQL показан в листинге 15.5. Разумеется, вы также можете сохранить данные в MongoDB или даже в Redis, если вам нужно надежно сохранить свои Redis-данные. Теперь модуль Crypto уже не нужен, поскольку мы больше не сохраняем пароли, что является существенным преимуществом аутентификации через стороннюю службу.

Листинг 15.5. Полная аутентификация пользователя через Твиттер

```
var express = require('express')
  , flash = require('connect-flash')
  , passport = require('passport')
  , TwitterStrategy = require('passport-twitter').Strategy
  , http = require('http');

var mysql = require('mysql');

var TWITTER_CONSUMER_KEY = "yourkey";
var TWITTER_CONSUMER_SECRET = "yoursecret";

var client = mysql.createClient({
   user : 'username',
   password : 'password'
});

client.query('USE nodetest2');

function findUser(id, callback) {
   var user;

   client.query('SELECT * FROM twitteruser WHERE id = ?',
              [id], function(err, result, fields) {
       if (err) return callback(err);
       user = result[0];
       console.log(user);
       return callback(null,user);
  });
};

function createUser(profile, token, tokenSecret, callback) {
   var qryString = 'INSERT INTO twitteruser ' +
                   '(id, name, screenname, location, description,' +
                   'url, img, token, tokensecret)' +
                   ' values (?,?,?,?,?,?,?,?,?)';
   client.query(qryString, [
             profile.id,
             profile.displayName,
             profile.username,
             profile._json.location,
             profile._json.description,
             profile._json.url,
             profile._json.profile_image_url,
             token,
             tokenSecret], function(err, result) {
                if (err) return callback(err);
                var user = {
                  id : profile.id,
```

```
                name : profile.displayName,
                screenname : profile.screen_name,
                location : profile._json.location,
                description: profile._json.description,
                url : profile._json.url,
                img : profile._json.profile_image_url,
                token : token,
                tokensecret : tokenSecret};
                console.log(user);
                return callback(null,user);
            });
};

function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/auth/twitter')
}

passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
   findUser(id, function(err, user) {
      done(err,user);
   });
});

passport.use(new TwitterStrategy(
       { consumerKey: TWITTER_CONSUMER_KEY,
         consumerSecret: TWITTER_CONSUMER_SECRET,
         callbackURL:
              "http://examples.burningbird.net:3000/auth/twitter/callback"},
       function(token, tokenSecret,profile,done) {
          findUser(profile.id, function(err,user) {
             console.log(user);
             if (err) return done(err);
             if (user) return done(null, user);
             createUser(profile, token, tokenSecret, function(err, user) {
               return done(err,user);
             });
          })
       })
);

var app = express();

app.configure(function(){
  app.set('views', __dirname + '/views');
```

продолжение ⇩

Листинг 15.5 *(продолжение)*

```
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('keyboard cat'));
  app.use(express.session());
  app.use(passport.initialize());
  app.use(passport.session());
  app.use(flash());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.get('/', function(req, res){
  res.render('index', { title: 'authenticate', user: req.user });
});

app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});

app.get('/auth', function(req,res) {
   res.render('auth', { title: 'authenticate' });
});

app.get('/auth/twitter',
  passport.authenticate('twitter'),
  function(req, res){
  });

app.get('/auth/twitter/callback',
  passport.authenticate('twitter', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/admin');
  });

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

В отношении других OAuth-служб можно предпринять те же шаги, что и для создания Passport-стратегии Твиттера. В качестве примера можно задействовать точно такой же код, который использовался в приложения для Твиттера, чтобы аутентифицировать пользователя через Фейсбук. Разница лишь в том, что вам нужно предоставить ключ и секрет для Фейсбука, а не для Твиттера.

Из-за схожести в коде и обработке данных сегодня многие приложения допускают прохождение аутентификации в самых разных OAuth-службах.

Модуль Passport пытается так переформатировать данные, возвращаемые различными службами, чтобы функциональность обработки профиля не требовала существенного изменения. Тем не менее вам потребуется изучить профиль, возвращаемый каждой службой, чтобы определить, что систематически ею предоставляется, и решить, что нужно, а что не нужно сохранять.

Еще есть проблема отзыва у пользователя в службе прав доступа к приложению. Разумеется, это касается веб-приложения только в том случае, если человек решил проходить аутентификацию для доступа к тому же приложению в другой службе, в таком случае сохраняется его новая информация, и приложение продолжает работу по накатанной схеме. Единственным негативным последствием является теперь уже ненужная запись в базе данных, содержащая прежнюю информацию для аутентификации данного человека, что касается изменения приложения с целью создания для него идентификатора, связанного с конкретным применением, и обновления записи при изменении серверов аутентификации, то это не потребует большого объема дополнительной работы. Пусть это будет вашим домашним заданием. А теперь настало время посмотреть на другие аспекты безопасности приложения и заняться приведением в порядок данных формы.

Защита приложений и противодействие атакам

Раз вы себя относите к JavaScript-разработчику, то вам уже немного известно об опасности получения пользовательского ввода и непосредственной передачи этого ввода в вызов инструкции `eval`. Как веб-разработчик, вы также знаете об опасностях получения текста от пользователей и добавления его непосредственно в предложение `where` SQL-инструкции.

У Node-приложений имеются те же уязвимости, которые присущи JavaScript-приложениям на стороне клиента, а также дополнительные уязвимости, связанные с серверными приложениями, использующими системы баз данных, особенно системы реляционных баз данных.

Чтобы обеспечить безопасность приложения, нужно, как говорилось в предыдущем разделе, предоставить надежные системы аутентификации и авторизации. Но столь же важно защитить приложение от инъекционных атак и других попыток использования открытых мест вашей системы для доступа к важным и конфиденциальным данным.

Раньше форма регистрации получала текст непосредственно от пользователя и вставляла его в SQL-запрос. Это было далеко не самым мудрым решением, потому что человек мог ввести текст, способный нанести вред базе SQL-данных. Например, пусть текст формирует данные в предложении `WHERE`, и он добавляется непосредственно к строке предложения `WHERE`:

```
var whereString = "WHERE name = " + name;
```

Кроме того, пусть строковое значение `name` имеет следующий контент:

```
'johnsmith; drop table users'
```

В результате у вас могут быть проблемы.

То же самое случается при обработке текста или JSON-строки, получаемой от пользователя, или исходного кода в JavaScript-инструкции `eval`; входная строка может принести больше вреда, чем пользы.

Обе разновидности уязвимостей требуют, чтобы мы очистили вводимые данные перед их использованием в условиях, когда они могут нанести вред. Обе они также требуют применения средств и технологий, обеспечивающих максимальную безопасность приложений.

Откажитесь от функции eval

Независимо от того, относятся ваши JavaScript-приложения к Node или нет, для них есть одно простое правило: откажитесь от использования функции `eval`. Функция `eval` является наименее защищенным и наиболее либеральным JavaScript-компонентом, поэтому нужно относиться к ее применению со страхом и трепетом.

В большинстве случаев функция `eval` вам не потребуется. Один из примеров, где она могла бы пригодиться, связан с конвертацией JSON-строки в объект. Но есть весьма простой способ защиты от инъекционных JavaScript-атак при конвертации строки в объект: использовать для обработки входной JSON-строки не функцию `eval`, а метод `JSON.parse`. Функция `eval` не распознает то, что включено в текст, а вот `JSON.parse` проверяет, что JSON-строка это *только* JSON-строка:

```
var someObj = JSON.parse(jsonString);
```

Поскольку Node использует движок V8, мы знаем, что у нас есть доступ к JSON-объекту, поэтому насчет всех этих межбраузерных ухищрениях нам волноваться не стоит.

Используйте флажки, переключатели и раскрывающиеся списки

Согласно второму простому правилу разработки веб-приложений нужно стремиться минимизировать возможности ввода в веб-форму произвольного текста. Вместо открытых текстовых полей следует предлагать пользователю раскрывающиеся списки, флажки и переключатели. Тем самым гарантируется не только безопасность данных, но также их целостность и надежность.

Много лет назад я занимался очисткой таблицы базы данных, в которую данные поступали из клиентской формы. В форму все данные вводились (авиационными инженерами) в виде открытого текста. Одно из полей предназначалось для сохранения идентификаторов запчастей, если они были приемлемыми. Размытость понятия «приемлемости» запчасти и стала причиной краха приложения.

Инженеры решили использовать это поле для «заметок и прочих сведений», поскольку в форме не было четко обозначено его предназначение. Я находил в нем не только идентификаторы запчастей, но и напоминания о необходимости заказать обед. Хотя чтение было увлекательным, информация зачастую была бесполезна

для компании. И провести очистку было очень сложно, поскольку номера запчастей от различных поставщиков не были в достаточной степени похожи друг на друга, чтобы можно было применить регулярные выражения.

Это пример *непреднамеренного* причинения вреда. С примером *преднамеренного* причинения вреда мы познакомились в предыдущем разделе, где SQL-инструкция на удаление таблицы базы данных была прикреплена к входному имени пользователя.

Если нужно, чтобы пользователь вводил в поля произвольный текст, например имя пользователя при его входе в систему, нужно очищать данные перед их использованием в инструкции обновления данных или в запросе.

Очищайте и санируйте данные с помощью модуля node-validator

Если без текстовых полей ввода не обойтись, перед использованием введенных данных их следует очистить. Модуль node-mysql предоставляет метод `client.escape`, который экранирует введенный текст и защищает от потенциальных инъекционных SQL-атак. Можно также отключить потенциально разрушительную функциональность. В главе 10 при рассмотрении MongoDB я упоминал о том, как установить флаг для обязательной сериализации JavaScript-функций при их хранении.

Нужно также применять средства проверки, которые не только гарантируют безопасность входных данных, но и обеспечивают их целостность. Одним из таких значимых средств проверки является модуль node-validator.

Установите node-validator с помощью диспетчера Node-пакетов:

```
npm install node-validator
```

Модуль экспортирует два объекта, `check` и `sanitize`:

```
var check = require('validator').check,
    sanitize = require('validator').sanitize;
```

С их помощью можно проверить, что входные данные соответствуют определенному формату, например, что вводимый текст является адресом электронной почты:

```
try {
   check(email).isEmail();
} catch (err) {
   console.log(err.message); // Неверный электронный адрес
}
```

Если данные не проходят проверку, node-validator вбрасывает ошибку. Если требуется более подробное сообщение об ошибке, вы можете предоставить его в качестве второго необязательного параметра метода `check`:

```
try {
   check(email, "Please enter a proper email").isEmail();
} catch (err) {
   console.log(err.message); // Введите, пожалуйста, правильный адрес
}
```

Фильтр `sanitize` гарантирует санацию строки в соответствии с используемым методом:

```
var newstr = sanitize(str).xss(); // предупреждение XSS-атаки
```

В листинге 15.6 для проверки и санации трех разных строк используются оба объекта.

Листинг 15.6. Проверка методов модуля node-validator

```
var check = require('validator').check,
    sanitize = require('validator').sanitize;

var email = 'shelleyp@burningbird.net';
var email2 = 'this is a test';

var str = '<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>';
try {
   check(email).isEmail();
   check(email2).isEmail();
} catch (err) {
   console.log(err.message);
}

var newstr = sanitize(str).xss();
console.log(newstr);
```

Результат запуска этого приложения:

```
Invalid email
[removed][removed]
```

У модуля node-validator есть поддержка на платформе Express — это связующий Express-модуль express-validator. Чтобы включить его в ваше Express-приложение, нужны следующие инструкции:

```
var expressValidator = require('express-validator');
...
app.use(expressValidator);
```

В результате непосредственно для объекта запроса вы сможете использовать методы `check`, `sanitize` и др.:

```
app.get('/somepage', function (req, rest) {
   ...
   req.check('zip', 'Please enter zip code').isInt(6);
   req.sanitize('newdata').xss();
   ...
});
```

Код из песочницы

Для безопасного помещения JavaScript-кода в песочницу служит Node-модуль vm. Он предоставляет доступ к новой виртуальной машине движка V8, в которой можно запустить JavaScript-код, передаваемый в качестве параметра.

Под «помещением в песочницу» обычно понимается изоляция кода от всего, что может быть использовано для нанесения вреда.

Есть два варианта использования модуля vm. В первом случае метод `vm.createScript` вызывается со сценарием, переданным методу в качестве параметра. Модуль vm компилирует его и возвращает объект сценария, представляющий сценарий:

```
var vm = require('vm');
var script_obj = vm.createScript(js_text);
```

Затем сценарий можно запустить в отдельном контексте, передав ему в качестве дополнительного объекта любые данные, которые ему могут понадобиться:

```
script_obj.runInNewContext(sandbox);
```

В листинге 15.7 представлен небольшой, но полноценный пример применения модуля vm для компиляции JavaScript-инструкции с использованием двух свойств объекта песочницы и созданием третьего.

Листинг 15.7. Простой пример использования Node-модуля vm с целью создания песочницы для сценария

```
var vm = require('vm');
var util = require('util');

var obj = { name: 'Shelley', domain: 'burningbird.net'};

// компиляция сценария
var script_obj = vm.createScript("var str = 'My name is ' + name + ' at ' +
                                                          domain", 'test.vm');
// запуск в новом контексте
script_obj.runInNewContext(obj);

// обследование объекта песочницы
console.log(util.inspect(obj));
```

При запуске приложения возвращаются следующие данные:

```
{ name: 'Shelley',
  domain: 'burningbird.net',
  str: 'My name is Shelley at burningbird.net' }
```

Объект, передаваемый в новый контекст, является точкой взаимодействия между вызывающим приложением и сценарием из песочницы. У сценария нет никакого

другого доступа к родительскому контексту. Если попробовать использовать такой глобальный объект, как консоль в вашем JavaScript-коде из песочницы, вы получите ошибку.

Чтобы продемонстрировать это, воспользуемся листингом 15.8, который является измененной версией листинга 15.7 для загрузки сценария из файла и его запуска. Загружаемый сценарий повторяет сценарий предыдущего примера, но с добавлением запроса `console.log`:

```
var str = 'My name is ' + name + ' from ' + domain;
 console.log(str);
```

Метод `vm.createScript` не может прочитать файл напрямую. Второй (дополнительный) параметр является не файлом, а именем, используемым в качестве метки в трассе стека, и предназначен он только для отладочных целей. Для чтения содержимого файла сценария нужен метод `readFile` объекта файловой системы.

Листинг 15.8. Изменения в коде для использования vm с целью помещения в песочницу сценария, считанного из файла

```
var vm = require('vm');
var util = require('util');
var fs = require('fs');

fs.readFile('suspicious.js', 'utf8', function(err, data) {
  if (err) return console.log(err);

  try {

     console.log(data);
     var obj = { name: 'Shelley', domain: 'burningbird.net'};

     // компилирование сценария
     var script_obj = vm.createScript(data, 'test.vm');

     // запуск в новом контексте
     script_obj.runInNewContext(obj);

     // проверка объекта песочницы
     console.log(util.inspect(obj));
  } catch(e) {
     console.log(e);
  }
});
```

При запуске приложения будет возвращена следующая информация:

```
[SyntaxError: Unexpected token :]
```

Произошла ошибка, которая и должна была произойти, потому что в виртуальной машине нет объекта консоли, это виртуальная машина движка V8, а не виртуальная машина Node. Мы уже видели, как можно реализовать любой процесс с помощью

дочерних процессов в Node-приложении. Разумеется, мы не хотим, чтобы такая мощная возможность была доступна коду из песочницы.

Мы можем запустить сценарий в контексте движка V8, следовательно, у него будет доступ к глобальному объекту. В листинге 15.9 представлена переделанная версия приложения из листинга 15.8, но на этот раз в нем используется метод `runInContext` с переданным методу объектом контекста. В объект контекста вносится объект, имеющий параметры, ожидаемые сценарием.

Однако вывод на экран результатов исследования объекта после выполнения сценария показывает, что только что определенное свойство `str` в нем больше не присутствует. Нам нужно исследовать контекст, чтобы увидеть, существует ли объект как в текущем контексте, так и в контексте песочницы.

Листинг 15.9. Запуск кода в контексте с объектом контекста, переданным vm

```
var vm = require('vm');
var util = require('util');
var fs = require('fs');

fs.readFile('suspicious.js', 'utf8', function(err, data) {
  if (err) return console.log(err);
  try {

     var obj = { name: 'Shelley', domain: 'burningbird.net' };

     // компиляция сценария
     var script_obj = vm.createScript(data, 'test.vm');

     // создание контекста
     var ctx = vm.createContext(obj);

     // запуск в новом контексте
     script_obj.runInContext(ctx);

     // обследование объекта
     console.log(util.inspect(obj));

     // обследование контекста
     console.log(util.inspect(ctx));

  } catch(e) {
     console.log(e);
  }
});
```

В примерах используется заранее скомпилированный блок сценария — это удобно, если вы собираетесь запускать сценарий несколько раз. Однако если вы хотите запустить его только один раз, можно обратиться к обоим методам `runInContext` и `runInThisContext` непосредственно с виртуальной машины. Разница в том, что вам придется передать сценарий в качестве первого параметра:

```
var obj = { name: 'Shelley', domain: 'burningbird.net' };

// создание контекста
var ctx = vm.createContext(obj);

// запуск в новом контексте
vm.runInContext(data,ctx,'test.vm');

// обследование контекста
console.log(util.inspect(ctx));
```

И опять в предоставленном контексте сценарий из песочницы имеет доступ к определенному через `createContext` глобальному объекту, в который внесены любые данные, необходимые коду из песочницы. А все данные, полученные в результате выполнения сценария, могут быть извлечены из этого контекста после запуска сценария.

16 Масштабирование и развертывание Node-приложений

В один прекрасный момент вам захочется перевести свое Node-приложения из состояния разработки и тестирования в состояние полноценной эксплуатации. В зависимости от того, чем занято ваше приложение и какие службы предоставляет (или в каких службах нуждается), процесс такого перевода может быть простым или очень сложным.

Я собираюсь вкратце рассмотреть возможные комбинации и проблемы, связанные с развертыванием Node-приложения при переводе его в режим эксплуатации. Некоторые приложения требуют минимальных усилий с вашей стороны, например установка модуля Forever гарантирует хорошую и бесперебойную работу Node-приложения. В то же время другие приложения, например развертываемые на облачном сервере, могут потребовать существенных затрат времени и перспективного планирования.

Развертывание вашего Node-приложения на вашем сервере

Перевод приложения из стадии разработки в эксплуатацию не отличается особой сложностью, но вы должны подготовиться к этому и убедиться, что ваше приложение доведено до состояния, когда достигнута его максимальная производительность и все потенциальные потери времени сведены к минимуму.

Рассмотрим условия развертывания Node-приложения:

- Ваше приложение должно быть хорошо протестировано как пользователями, так и разработчиками.
- Вам нужно обеспечить безопасное развертывание приложения и слаженность внесения в него изменений и исправлений.

- Ваше приложение должно отвечать условиям безопасности.
- При сбоях по той или иной причине ваше приложение должно перезапускаться.
- Вам может понадобиться интегрировать свои Node-приложения с другими серверами, например с Apache.
- Вы должны отслеживать производительность своего приложения и быть готовым перенастроить параметры приложения, если производительность начнет падать.
- Вам нужно расходовать ресурсы вашего сервера с максимальной эффективностью.

В главе 14 рассказывалось о блочном и приемочном тестировании, а также о тестировании производительности. В главе 15 рассматривались вопросы безопасности. Здесь мы поговорим о реализации других компонентов развертывания Node-приложения при переводе его в режим эксплуатации на вашем собственном сервере.

Запись в файл package.json

У каждого Node-модуля есть файл package.json, в котором содержится информация о модуле, а также сведения о зависимостях этого модуля от другого кода. Когда в главе 4 рассматривались модули, я упомянул о файле package.json. Теперь я хочу исследовать этот файл более детально, особенно в плане возможности его применения для развертывания приложения.

Как следует из его имени, файл package.json должен иметь формат JSON. Вы можете инициировать процесс package.json путем запуска команды `npm init` и ответа на вопросы. При запуске команды `npm init` в главе 4 я не предоставлял никаких сведений о зависимостях, что требуется для большинства Node-приложений.

Приложение для виджетов, которое создавалось нами на протяжении нескольких глав этой книги, может послужить примером, хотя и довольно скромным, на котором можно исследовать процедуру развертывания. Как же должен выглядеть файл package.json для этого приложения?

Я не буду рассматривать все возможные значения данных в package.json, ограничившись только теми, которые имеют значение для Node-приложения.

Для начала нам нужно предоставить базовую информацию о приложении, включая его имя, версию и основного автора:

```
{
  "name": "WidgetFactory",
  "preferGlobal": "false",
  "version": "1.0.0",
  "author":
    "Shelley Powers <shelley.just@gmail.com> (http://burningbird.net)",
  "description": "World's best Widget Factory",
```

Учтите, что в значении свойства `name` не должно быть никаких пробельных символов.

Хотя проще использовать неразделенный формат, значения, касающиеся авторства, могут быть разбиты на части:

```
"author": { "name": "Shelley Powers",
                  "email": "shelley.just@gmail.com",
                  "url": "http://burningbird.net"},
```

Если к созданию приложения причастны другие люди, вы можете перечислить их в массиве с ключевым словом `contributors`, идентифицируя каждого человека точно так же, как автора.

Если у фабрики виджетов есть двоичное приложение, его можно указать с помощью свойства `bin`.

Пример такого применения свойства `bin` можно найти в файле package.json приложения Nodeload (см. главу 14):

```
"bin": {
      "nodeload.js": "./nodeload.js",
      "nl.js": "./nl.js"
            },
```

Эти параметры показывают, что при глобальной установке модуля я могу запустить Nodeload, просто набрав команду `nl.js`.

У приложения для виджетов нет инструмента командной строки. У него также нет никаких сценариев. Ключевое слово `scripts` идентифицирует любые сценарии, запускаемые в течение жизненного цикла пакета. Есть события, которые могут произойти в течение жизненного цикла, включая `preinstall`, `install`, `publish`, `start`, `test`, `update` и т. д., и с каждым из них может быть запущен сценарий.

В Node-приложении или в каталоге модуля можно запустить следующую npm-команду:

```
npm test
```

В результате будет запущен сценарий test.js:

```
 "scripts": {
  "test": "node ./test.js"
},
```

В сценарий приложения для виджетов можно включить любой сценарий блочного теста, вдобавок к любому другому сценарию, необходимому для установки (например, к сценарию настройки окружения для приложения). Хотя у фабрики виджетов пока нет пускового сценария, он должен быть у вашего приложения, особенно если оно предназначено для развертывания в облачной службе (см. далее в этой главе).

Если для некоторых значений сценарий не предоставить, диспетчер Node-пакетов предложит параметры, используемые по умолчанию. Если у приложения в корневом каталоге пакета есть файл server.js, пусковой сценарий по умолчанию запускает приложение с Node:

```
node server.js
```

Свойство `repository` предоставляет информацию об инструменте управления исходным кодом приложения, а свойство `url` — о месте расположения исходного кода, если он опубликован для доступа через сеть:

```
"repository": {
    "type": "git",
    "url": "https://github.com/yourname/yourapp.git"
  },
```

Свойство `repository` не является необходимым, пока вы не опубликуете исходный код своего приложения (хотя вы можете ограничить доступ к исходному коду определенной группой людей). Один из положительных моментов предоставления этой информации состоит в том, что пользователи получают возможность получить доступ к вашей документации с помощью npm-команды `docs`:

```
npm docs packagename
```

На моей системе Ubuntu я сначала настроил браузер на Lynx:

```
npm config set browser lynx
```

Затем я открыл `docs` для модуля аутентификации Passport, рассмотренного в главе 15:

```
npm docs passport
```

Установка значения свойства `repository` помогает диспетчеру Node-пакетов найти документацию.

Одним из наиболее важных определений в файле `package.json` является версия Node, в которой может быть запущено ваше приложение. Эта версия указывается с помощью свойства `engine`. В случае с фабрикой виджетов она была протестирована в стабильных выпусках 0.6.x и 0.8.2, что означает возможность ее работы также и с будущими версиями (выше 0.8). Питая подобные надежды, я задал следующий параметр движка:

```
  "engines": {
    "node": ">= 0.6.0 < 0.9.0"
  },
```

У приложения для виджетов есть несколько зависимостей как для среды эксплуатации, так и для среды разработки. Они перечисляются отдельно, первая в `devDependencies`, последняя в `dependencies`. Зависимость от каждого модуля указывается как свойство, а необходимая версия — как его значение:

```
  "dependencies": {
    "express": "3.0",
    "jade": "*",
    "stylus": "*",
    "redis": "*",
    "mongoose": "*"
  },
  "devDependencies": {
    "nodeunit": "*"
  }
```

Если есть какая-либо зависимость от операционной системы или центрального процессора, мы можем указать это так:

```
"cpu" : ["x64", "ia32"],
"os": ["darwin","linux"]
```

Есть также несколько значений публикации, включая значение свойства `private`, гарантирующих, что приложение не будет опубликовано случайно:

```
"private": true,
```

А свойство `publishConfig` служит для установки параметров конфигурирования диспетчера Node-пакетов.

К тому времени, когда все будет закончено, файл package.json, относящийся к фабрике виджетов, должен выглядеть так, как показано в листинге 16.1.

Листинг 16.1. Файл package.json для приложения фабрики виджетов

```
{
  "name": "WidgetFactory",
  "version": "1.0.0",
  "author":
    "Shelley Powers <shelley.just@gmail.com> (http://burningbird.net)",
  "description": "World's best Widget Factory",
  "engines": {
    "node": ">= 0.6.0"
  },
  "dependencies": {
    "express": "3.0",
    "jade": "*",
    "stylus": "*",
    "redis": "*",
    "mongoose": "*"
  },
  "devDependencies": {
    "nodeunit": "*"
  },
  "private": true
}
```

Файл package.json можно протестировать путем копирования кода фабрики виджетов в новое место расположения с последующим выполнением команды `npm install -d`, чтобы посмотреть, установлены ли все модули, от которых зависит работа приложения, и запущено ли приложение.

Обеспечение жизнеспособности приложения с помощью модуля Forever

Вы сделали со своим приложением все самое лучшее, что только могли. Основательно его протестировали и ввели обработку ошибок, добившись изящного решения проблем. Тем не менее могут возникать те или иные незапланированные сбои, способные

вызвать падение вашего приложения. В таком случае вам нужно обеспечить повторный запуск приложения, даже если вы не планировали его перезапускать.

Именно для этого предназначен модуль Forever, гарантирующий, что ваше приложение будет перезапущено после падения. Кроме того, он позволяет запустить ваше приложение в фоновом режиме вне рамок текущего терминального сеанса. Модуль Forever может запускаться из командной строки или в качестве части приложения. Если вы используете его из командной строки, вам понадобится установить его глобально:

```
npm install forever -g
```

Вместо непосредственного запуска приложения с Node, запустите его с Forever:

```
forever start -a -l forever.log -o out.log -e err.log httpserver.js
```

Эта команда запускает сценарий httpserver.js и указывает имена для Forever-журналов вывода данных и ошибок. Она также заставляет приложение добавлять записи в журналы, если файлы этих журналов уже существуют.

Если со сценарием произойдет нечто такое, что вызовет сбой, Forever перезапустит сценарий. Forever также гарантирует, что Node-приложение продолжит свое выполнение, даже если вы закроете окно своего терминала, использованное для запуска приложения.

Forever предлагает как варианты настройки, так и действия. Значение `start` в только что показанной командной строке является примером действия. Перечень всех доступных действий выглядит следующим образом:

`start`

: Запускает сценарий.

`stop`

: Останавливает выполнение сценария.

`stopall`

: Останавливает выполнение всех сценариев.

`restart`

: Перезапускает сценарий.

`restartall`

: Перезапускает все сценарии, запущенные с помощью Forever.

`cleanlogs`

: Удаляет все записи в журналах.

`logs`

: Выводит список всех журналов для всех Forever-процессов.

`list`

: Выводит список всех выполняемых сценариев.

`config`

Выводит перечень пользовательских конфигураций.

`set <ключ> <значение>`

Устанавливает для конфигурации пару ключ-значение.

`clear <ключ>`

Очищает значение ключа конфигурации.

`logs <сценарий|индекс>`

Закрывает журналы для сценария или индекса.

`columns add <столбец>`

Добавляет столбец к списку вывода Forever.

`columns rm <столбец>`

Удаляет столбец для списка вывода Forever.

`columns set <столбцы>`

Устанавливает все столбцы для списка вывода Forever.

Следующие данные являются списком вывода после запуска httpserver.js в качестве фонового Forever-процесса:

```
info:  Forever processes running
data:    uid    command  script         forever  pid   logfile          uptime
data:  [0] ZRYB node     httpserver.js  2854     2855
                                    /home/examples/.forever/forever.log
 0:0:9:38.72
```

Существует также большое количество вариантов настройки, включая только что показанные для файлов журналов, а также для запуска сценария (`-s` или `--silent`), для включения подробного вывода данных Forever (`-v` или `--verbose`), для установки исходного каталога сценария (`--sourceDir`) и т. д. Все эти варианты настройки можно увидеть после набора и запуска следующей команды:

```
forever --help
```

Forever можно встроить непосредственно в код, как показано в документации по этому приложению:

```
var forever = require('forever');

  var child = new (forever.Monitor)('your-filename.js', {
    max: 3,
    silent: true,
    options: []
  });

  child.on('exit', this.callback);
  child.start();
```

Кроме того, Forever можно использовать с Nodemon (см. главу 14) не только для перезапуска приложения в случае неожиданного сбоя, но и для обновления приложения при обновлении исходного кода. Нужно просто заключить Nodemon в Forever-оболочку и указать ключ `-exitcrash`, чтобы обеспечить при сбое приложения аккуратный выход из Nodemon с передачей управления Forever:

```
forever nodemon --exitcrash httpserver.js
```

Если случится сбой приложения, Forever запустит модуль Nodemon, который, в свою очередь, запустит Node-сценарий. При этом гарантируется обновление запущенного сценария при изменении исходного кода, а также то, что из-за неожиданного сбоя приложение не останется надолго вне сети.

Если нужно, чтобы приложение запускалось при перезагрузке вашей системы, его нужно установить как фоновое. Среди примеров, проставляемых с модулем Forever, есть пример, помеченный как initd-example. Этот пример является основой сценария, запускающего ваше приложение с Forever при перезагрузке системы. В этот сценарий нужно внести изменения, соответствующие вашему окружению, а также переместить его в каталог /etc/init.d. Как только это будет сделано, ваше приложение сможет перезапускаться без вашего вмешательства даже после перезагрузки системы.

Совместное использование Node и Apache

Все примеры в этой книге не запускаются через порт 80, предлагаемый по умолчанию для веб-служб. Некоторые примеры мы запускали через порт 3000, другие — через порт 8124. В моей системе пришлось использовать другой порт, потому что порт 80 выделен под веб-запросы для Apache. Однако вряд ли кому-то захочется при обращении к веб-сайту указывать порт. То есть нам нужно обеспечить сосуществование Node-приложений с другими веб-службами, будь то Apache, Nginx или другой веб-сервер.

Если система запускает Apache, а у вас нет возможности изменить используемый Apache порт, вы с помощью файла .htaccess можете переписать веб-запросы для Node, перенаправив их на нужный порт без ведома пользователя:

```
<IfModule mod_rewrite.c>

    RewriteEngine on

    # Перенаправление всего подкаталога:
    RewriteRule ^node/(.+) http://examples.burningbird.net:8124/$1 [P]

</IfModule>
```

При наличии соответствующих полномочий вы можете также создать поддомен конкретно для вашего Node-приложения и заставить Apache стать прокси-сервером для всех запросов к Node-приложению.

Такой подход используется в других средах этого типа, например при совместном запуске Apache и Tomcat:

```
<VirtualHost someipaddress:80>
    ServerAdmin admin@server.com
```

```
    ServerName examples.burningbird.net
    ServerAlias www.examples.burningbird.net

    ProxyRequests off

    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    <Location />
        ProxyPass http://localhost:8124/
        ProxyPassReverse http://localhost:8124/
    </Location>
</VirtualHost>
```

Если вы не ожидаете частых обращений к своему Node-приложению, этот механизм будет работать, причем с более чем приемлемой производительностью. Однако проблема обоих подходов состоит в том, что все запросы направляются через Apache-сервер, который для обработки каждого из них создает новый процесс. Весь смысл Node состоит в том, чтобы избегать таких больших накладных расходов. Если ожидается, что Node-приложение будет активно использоваться, то другим подходом является изменение Apache-файла ports.conf, а также изменение порта, прослушиваемого Apache:

```
Listen 80
```

Можно выбрать другой порт по вашему усмотрению, например 78:

```
Listen 78
```

Затем нужно воспользоваться прокси-сервером, таким как Node-модуль http-proxy, для прослушивания запросов и направления их с помощью прокси-сервера на соответствующий порт. В качестве примера, если Apache обрабатывает все запросы к подкаталогу public, а Node — все запросы к node, можно создать отдельный прокси-сервер, который получает входящие запросы и соответствующим образом выполняет их маршрутизацию:

```
var httpProxy = require('http-proxy');

var options = {

   router: {
      'burningbird.net/public_html' : '127.0.0.1:78',
      'burningbird.net/node' : '127.0.0.1:3000'
   }
};

var proxyServer = httpProxy.createServer(options);
proxyServer.listen(80);
```

Пользователю вся эта закулисная магия с портами не видна. Модуль http-proxy также работает с запросами веб-сокетов и с HTTPS.

А зачем вообще продолжать использовать Apache? Для того чтобы Drupal-приложения и другие подобные приложения, работающие с файлами .htaccess, контролировали доступ к своему контенту. К тому же несколько поддоменов на моем сайте применяют файл .htpasswd для парольной защиты контента. У всех этих примеров Apache-конструкций аналогов среди серверных Node-приложений не существует.

История использования Apache складывалась довольно долго. Проигнорировать ее в пользу Node-приложений сложнее, чем просто создать статический сервер с помощью Express.

Повышение производительности

В зависимости от ваших системных ресурсов существуют дополнительные приемы повышения производительности вашего Node-приложения. Большинство из них не столь очевидны, и все они выходят за рамки темы данной книги.

Если у вас многоядерная система и есть желание испробовать экспериментальную технологию, можно воспользоваться Node-кластеризацией. В документации по Node.js есть примеры кластеризации, в соответствии с которой каждый процесс порождается на другом центральном процессоре, хотя все процессы слушают входящие запросы на одном и том же порту.

В какой-нибудь из будущих версий Node мы сможем автоматически пользоваться многоядерной средой, просто передавая параметр `--balance` при запуске приложения.

Можно также пользоваться преимуществом распределенной компьютерной архитектуры с помощью такого модуля, как hook.io.

Существует множество трюков и технологий повышения производительности Node-приложений, но большинство из них требует большого объема работы. Вместо этого можно разместить свое приложение в облачной службе и пользоваться теми мерами повышения производительности, которые предлагаются хостом. Эта возможность рассматривается в следующем разделе.

Развертывание в облачной службе

Набирающим популярность вариантом запуска приложения является не размещение его на собственных серверах, а развертывание в облачной службе. Среди многочисленных причин для этого можно выделить следующие:

- Улучшенная безопасность (это примерно то же самое, что нанять собственную команду специалистов по безопасности).
- 24-часовой мониторинг (позволяющий вам спать спокойно).
- Незамедлительное масштабирование (если ваше приложение окажется вдруг на пике популярности, ваш сервер не рухнет).
- Стоимость (зачастую дешевле разместить приложение в облачной службе, чем на собственном сервере).
- Средства развертывания (облачные службы предоставляют инструментарий, который может упростить развертывание Node-приложений).

- Присутствие на гребне прогресса (единственная причина в этом перечне, *не являющаяся* веским аргументом в пользу развертывания вашего Node-приложения в облачной службе).

Разумеется, есть и негативные стороны. Одной из них являются необходимые ограничения на действия, которые можно производить с вашим приложением. Например, если ваше приложение требует использования такого средства, как ImageMagick, то во многих облачных службах оно не установлено, причем часто вам не разрешат его устанавливать. Кроме того, если в основе вашего приложения лежит Node версии 6.x (или 8.x, или какой-нибудь другой версии), облачная служба может оказаться настроенной на другую версию (например, 4.x).

Могут также возникнуть сложности для настройки вашего приложения в облаке. Некоторые облачные службы предоставляют такой инструментарий, что развертывание в основном сводится к вводу в текстовое поле информации о месте размещения и щелчку на кнопке. В то же время другие облачные службы могут потребовать большого объема подготовительной работы, для выполнения которой может иметься, а может и не иметься достаточно подробной документации.

В последнем разделе я хочу дать краткое представление о некоторых наиболее часто используемых облачных службах, предлагающих хостинг для Node-приложений, и коснуться тех аспектов, которые отличают одни службы от других.

Развертывание на Windows Azure с помощью Cloud9 IDE

Если в основе вашей среды лежит Windows и ранее вы уже использовали функциональность Windows (например, разрабатывали приложения с применением технологии .NET), тогда вас определенно заинтересует хостинг Node-приложения в Windows Azure. Чтобы упростить отправку Node-приложения в Azure, можно воспользоваться интегрированной средой разработки (Integrated Development Environment, IDE) Cloud9.

Cloud9 — это основанная на веб-технологиях интегрированная среда разработки, которая, помимо всего прочего, служит интерфейсом для вашей учетной записи в GitHub. При открытии приложения вам предоставляется интерфейс управления проектом, показанный на рис. 16.1.

На странице управления проектом щелчок на проекте откроет его на отдельной странице, где можно будет выбрать для редактирования любой из файлов проекта, как показано на рис. 16.2. Непосредственно из IDE можно создать клон существующего проекта в GitHub.

Вы можете добавлять и редактировать файлы, а затем запускать приложение непосредственно в среде Cloud9. Кроме того, Cloud9 поддерживает отладку.

Для начала работы с приложением среда Cloud9 предоставляется бесплатно, но для развертывания приложения придется подписаться на дополнительную услугу. Хотя основное внимание сконцентрировано на HTML- и Node-приложениях, поддерживаются несколько языков. Также поддерживается несколько репозиториев, включая GitHub, Bitbucket, Mercurial, Git и FTP-серверы.

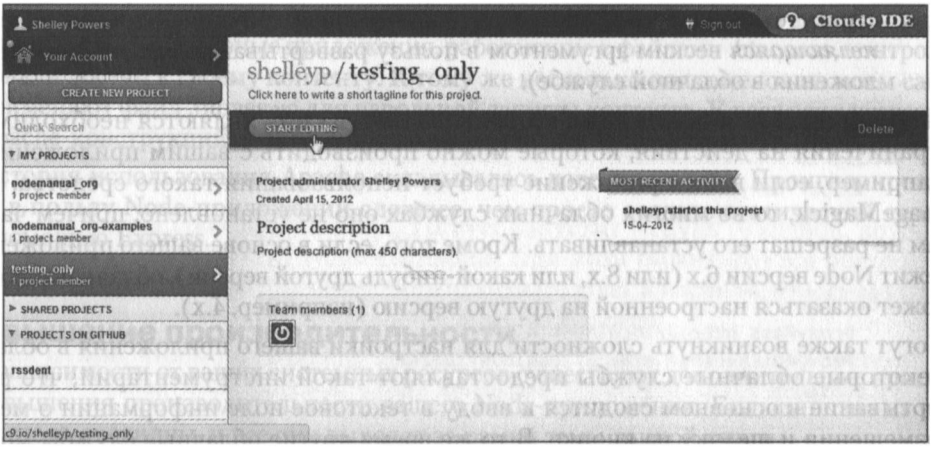


Рис. 16.1. Страница управления проектами в среде Cloud9

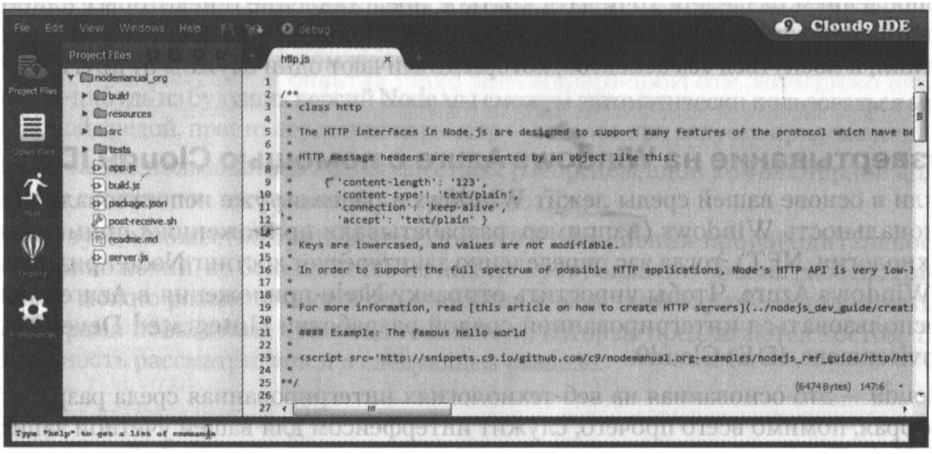


Рис. 16.2. Страница редактирования проекта в среде Cloud9

Интерфейс среды Cloud9 упрощает перевод приложения в Azure (и в другие службы, чуть позже мы к этому еще вернемся). Если у вас есть учетная запись в Azure, перевод Node-приложения в Azure осуществляется простым щелчком на кнопке Deploy (Развернуть) и предоставлением информации в открывающихся диалоговых окнах. Но сначала вы должны познакомиться с Azure. Перед принятием на себя каких-либо обязательств можно воспользоваться предоставляемым службой 90-дневным ознакомительным периодом.

Стоимость Azure зависит от количества экземпляров компьютеров, объема экземпляра базы данных SQL Server, объема хранилища для блока данных больших двоичных объектов (Binary Large Object Block, BLOB) и ширины полосы пропускания. Служба предлагает также неплохой набор документов, включая хорошее руководство по созданию Express-приложения на Azure.

Чуть ранее я упоминал о том, что среду Cloud9 можно развернуть на нескольких облаках. На данный момент поддерживаются три такие службы:

- Windows Azure;
- Heroku;
- Joyent.

Далее мы познакомимся со службами Joyent Development SmartMachine и Heroku.

Joyent Development SmartMachine

Службы Joyent SmartMachine — это виртуальные машины, работающие под управлением либо Linux, либо Windows. Эти машины поставляются готовыми и оптимизированными для запуска Node-приложений. Компания Joyent предлагает также средство Node.js Development SmartMachine, которое позволяет Node-разработчикам без каких-либо дополнительных расходов разместить приложение в облачной службе. Если вы готовы перейти к эксплуатации, то можете обновить эксплуатационную среду.

Joyent предоставляет подробные инструкции для начала работы с Node.js Development SmartMachine. Для этого требуется проделать следующие шаги:

1. Создать учетную запись в облачной службе Joyent.
2. Создать ключ безопасной оболочки (Secure Shell, SSH), если таковой еще не создан.
3. Обновить файл ~/.ssh/config, чтобы в нем был отражен уникальный номер порта для вашей машины.
4. Выполнить развертывание приложения на SmartMachine с помощью Git.
5. Обеспечить приложение файлом package.json и идентифицировать пусковой сценарий.

Здесь мы опять сталкиваемся с тем, что служба Node.js Development SmartMachine предназначена только для целей разработки.

Итак, что же нам предлагает Joyent Development SmartMachine? Для начала — никаких затрат. Это весьма разумный ход, дающий разработчикам возможность испытать облачный хостинг без особых расходов.

Кроме того, Joyent позволяет упростить Git-развертывание одновременно на нескольких машинах, а также поддерживает диспетчер Node-пакетов для управления зависимостями приложения.

Heroku

Мне нравятся облачные службы, которые не заставляют вас ничего платить, чтобы их опробовать, а учетная запись в Heroku бесплатна и оформляется очень быстро. Если вы решили воспользоваться службой для своей производственной системы, то она настраивается точно так же, как и Azure. К тому же облачный сервер предлагает очень хорошую документацию и инструментарий, которые можно установить в своей среде разработки, чтобы упростить развертывание приложения на Heroku (если вы не используете среду Cloud9).

Облачная служба поставляется с предварительно упакованными дополнениями, которые можно добавить к своей учетной записи, включая поддержку для одного из моих любимых хранилищ данных, Redis. Концепция дополнений в Heroku предусматривает очень хорошую управляемость, причем многие дополнения на период ознакомления с ними также бесплатны.

Как уже упоминалось, документация в Heroku одна из лучших среди облачных служб, а средства разработки действительно упрощают развертывание. Вы создаете приложение, создаете файл package.json с описанием всех зависимостей, объявляете тип процесса посредством простого профиля (в котором обычно содержится что-нибудь вроде `web: node app.js`), а затем запускаете приложение с помощью одного из средств, проставляемых как часть инструментария Heroku.

Для развертывания подтвердите принадлежность приложения к Git, а затем разверните приложение с помощью Git. Все предельно просто.

Amazon EC2

Служба Amazon Elastic Compute Cloud, или EC2, сейчас уже имеет свою историю, позволяющую считать ее довольно привлекательным вариантом. Она также не выдвигает слишком много требований к Node-разработчику, подбирающему хост для приложения в этой облачной службе.

Настройка Amazon EC2 немного отличается от настройки более традиционной виртуальной частной сети (Virtual Private Network, VPN). Вы определяете предпочтительную операционную систему, дополняете ее необходимым для запуска Node программным обеспечением, развертываете приложение с помощью Git, а затем используете такой инструмент, как Forever, чтобы гарантировать надежную работу приложения.

Служба Amazon EC2 имеет свой веб-сайт, который поможет упростить настройку экземпляра. Эта служба не предлагается бесплатно, как Joyent, но плата вполне разумна, всего два цента за один час опробования службы.

Если ваше приложение использует базу данных MongoDB, на веб-сайте MongoDB можно найти очень подробные инструкции по настройке Amazon EC2.

Nodejitsu

В настоящее время имеется только бета-версия службы Nodejitsu, в которой предлагаются пробные учетные записи. Подобно многим другим превосходным облачным службам, эта служба позволяет опробовать службу бесплатно.

Как и Heroku, Nodejitsu предоставляет инструментальное средство под названием jitsu, позволяющее упростить развертывание. Этот инструмент устанавливается с помощью диспетчера Node-пакетов. После установки jitsu войдите в Nodejitsu и проведите развертывание, просто набрав следующую команду:

```
jitsu deploy
```

Этот инструмент берет все необходимое из файла package.json, задает несколько малозначимых вопросов, и все проходит хорошо.

Nodejitsu предлагает собственную интегрированную среду разработки на базе веб-технологии, хотя у меня еще не было шанса ею воспользоваться. Похоже, она намного проще, чем среда Cloud9.

Приложение. Node, Git и GitHub

Git является системой управления версиями, подобной CVS (Concurrent Versioning System) или Subversion. Git отличается от других более традиционных систем управления версиями механизмом обслуживания источника по мере внесения в него изменений. Такие системы управления версиями, как CVS, сохраняют изменения в виде различий от базового файла, а в Git сохраняются снимки кода на определенный момент времени. Если файл не изменился, Git просто ссылается на предыдущий снимок.

Чтобы приступить к работе Git, сначала нужно установить эту систему. Существуют двоичные файлы для Windows и Mac OS X, а также исходный код для различных разновидностей Unix. Установка на мой Linux-сервер (Ubuntu 10.04) потребовала ввода всего лишь одной команды:

```
sudo apt-get install git
```

Весь остальной код, от которого зависит работа этой системы, загружается и устанавливается автоматически.

С этого момента предполагается, что для ввода команд используется терминал на основе Unix. При работе под Windows для Git имеется графический интерфейс. Для его установки на своей системе нужно следовать документации, поставляемой вместе с интерфейсом, но основные процедуры во всех средах одинаковы.

После установки Git эту систему следует настроить. Git нужно предоставить имя пользователя (как правило, имя и фамилию) и адрес электронной почты. Из них формируются два компонента *доверенного автора*, которые будут использоваться для обозначения изменений от вашего имени:

```
git config --global user.name "ваше имя"
git config --global user.email "ваш электронный адрес"
```

Поскольку вы собираетесь работать с GitHub, предлагающей хостинг для большинства (если не для всех) Node-модулей, вам также нужно настроить учетную запись GitHub. В GitHub модно указывать произвольное имя пользователя, которое не обязательно должно совпадать с именем пользователя, только что указанным для Git. Нужно также сгенерировать SSH-ключ, чтобы предоставить его GitHub, для чего следуйте указаниям справочной документации по GitHub.

Большинство учебных пособий по Git начинаются с создания простого репозитория (или *repo*, если использовать общепринятую терминологию) для вашей работы. Поскольку нас главным образом интересует Git с Node, мы начнем с клонирования существующего репозитория, а не с создания нового. Но перед тем как можно будет клонировать источник, сначала нужно *разветвить* (получить рабочий снимок) репозиторий на веб-сайте GitHub, щелкнув на кнопке Your Fork (Разветвить), которая, как показано на рис. П.1, находится в верхней правой части главной страницы репозитория GitHub.

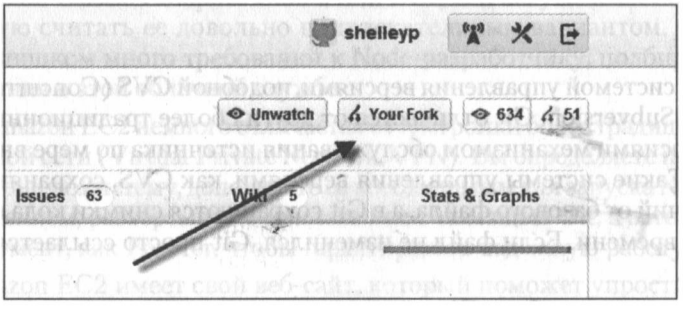


Рис. П.1. Разветвление существующего Node-модуля в GitHub

Затем вы сможете получить доступ к созданному в результате разветвления репозиторию в своем профиле. Вы также сможете получить доступ к URL-адресу Git на веб-странице только что созданного репозитория. Например, когда я инициировал разветвление модуля node-canvas (см. главу 12), URL-адрес был git@github.com:shelleyp/node-canvas.git. Команда для клонирования разветвленного репозитория (в формате `git clone` *URL-адрес*) выглядит следующим образом:

```
git clone git@github.com:shelleyp/node-canvas.git
```

Клонирование можно также провести через HTTP, хотя специалисты GitHub это делать не рекомендуют. Однако такой подход вполне приемлем, чтобы получить предназначенную только для чтения копию исходного репозитория. В такой копии можно найти примеры и другие материалы, которые могут не устанавливаться при установке модуля с помощью диспетчера Node-пакетов (или если вам нужен доступ к копии модуля в той его части, которая еще не передана в диспетчер Node-пакетов).

Получайте по URL-адресу доступ только для чтения через HTTP с веб-страницы каждого репозитория, например следующая команда относится к репозиторию модуля node-canvas:

```
git clone https://github.com/имя_пользователя/node-canvas.git
```

Можно также установить модуль, указав URL-адрес Git:

```
npm install git://github.com/username/node-whatever.git
```

Теперь у вас есть копия репозитория node-canvas (или того репозитория, к которому нужен доступ). Если хотите, можете внести изменения в любой исходный файл. Добавление новых или измененных файлов осуществляется с помощью команды `git add`, а фиксация этих изменений — команды `git commit` (используйте ключ `-m` для предоставления краткого комментария о внесенных изменениях):

```
git add somefile.js
git commit -m 'комментарий о сути изменения'
```

Если нужно посмотреть, установлен ли файл и готов ли он к фиксации изменений, можно набрать команду:

```
git status
```

Если нужно отправить изменения, чтобы они были включены в качестве части исходного репозитория, следует выдать запрос на перемещение. Для этого откройте репозиторий, полученный в результате разветвления, и найдите кнопку Pull Request (Запрос на перемещение) в вашем браузере, как показано на рис. П.2.

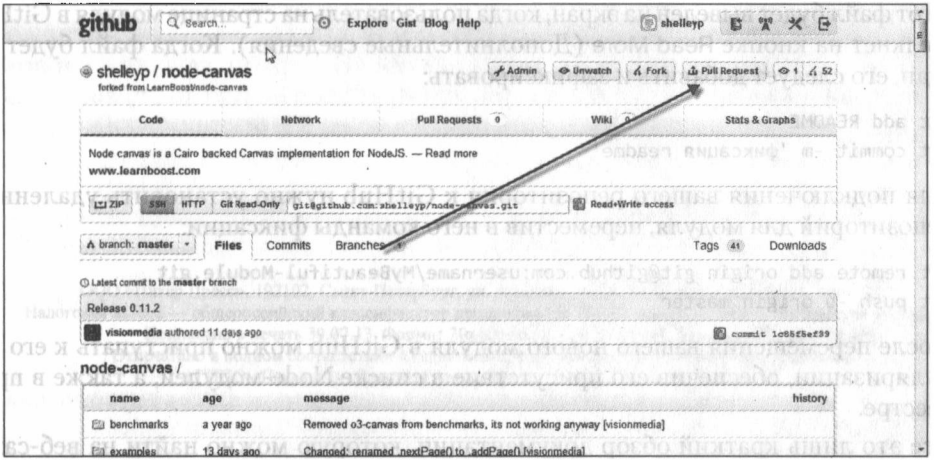


Рис. П.2. Для инициализации запроса на перемещение щелкните на кнопке Pull Request на странице GitHub

Щелчок на кнопке Pull Request (Запрос на перемещение) приведет к открытию панели предварительного просмотра, где можно ввести свое имя и описание изменения, а также предварительно просмотреть то, что вы собираетесь зафиксировать. Здесь также можно изменить диапазон фиксации и целевой репозиторий.

Если вас все устраивает, отправляйте запрос. При этом будет сделана запись в очередь запроса на перемещение для владельца репозитория, чтобы он мог выполнить слияние. Владелец репозитория может просмотреть изменение, обсудить изменение, и если он примет решение, удовлетворить запрос, извлечь его и выполнить слияние, сделать исправление и внести его, выполнить слияние в автоматическом режиме.

В GitHub имеется документация по слиянию при внесении изменений, а также по другим аспектам использования Git с сайтом хостинга.

Если вы создали собственный Node-модуль и хотите поделиться им с другими, то вам понадобится создать репозиторий. Это также можно сделать через GitHub, щелкнув на кнопке New Repository (Новый репозиторий) на вашей главной странице в GitHub и предоставив информацию о модуле, включая сведения о том, закрытый он или открытый.

Пустой репозиторий инициализируется с помощью команды `git init`:

```
mkdir ~/mybeautiful-module
cd ~/mybeautiful-module
git init
```

В своем любимом текстовом редакторе снабдите репозиторий файлом README. Этот файл будет выведен на экран, когда пользователь на странице модуля в GitHub щелкнет на кнопке Read More (Дополнительные сведения). Когда файл будет создан, его следует добавить и зафиксировать:

```
git add README
git commit -m 'фиксация readme'
```

Для подключения вашего репозитория к GitHub нужно установить удаленный репозиторий для модуля, переместив в него команды фиксации:

```
git remote add origin git@github.com:username/MyBeautiful-Module.git
git push -u origin master
```

После перемещения вашего нового модуля в GitHub можно приступать к его популяризации, обеспечив его присутствие в списке Node-модулей, а также в npm-реестре.

Все это лишь краткий обзор документации, которую можно найти на веб-сайте GitHub под ссылкой Help (Помощь).

Шелли Пауэрс

Изучаем Node.js

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Кривцов*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Жданов*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Е. Волошина*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 30.07.13. Формат 70x100/16. Усл. п. л. 30,960. Тираж 1000. Заказ № 554.
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГПИО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

---

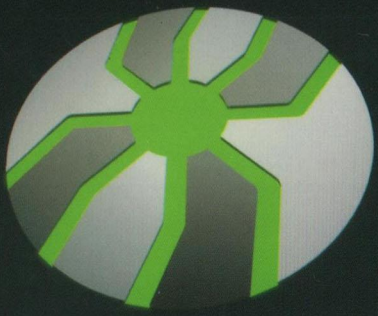
Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

---

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

Node.js является серверной технологией, которая основана на разработанном компанией Google JavaScript-движке V8. Это прекрасно масштабируемая система, поддерживающая не программные потоки или отдельные процессы, а асинхронный ввод-вывод, управляемый событиями. Она идеально подходит для веб-приложений, которые не выполняют сложных вычислений, но к которым происходят частые обращения. По целям использования Node сходен с фреймворками Twisted на языке Python и EventMachine на Ruby. В отличие от большинства программ JavaScript этот фреймворк исполняется не в браузере клиента, а на стороне сервера.

С помощью этого практического руководства вы сможете быстро овладеть основами Node. Книга понравится всем, кто интересуется новыми технологиями, например веб-сокетами или платформами создания приложений. Эти темы раскрываются в ходе рассказа о том, как использовать Node в реальных приложениях.

Тема: JavaScript | **Уровень пользователя:** опытный

O'REILLY®

ПИТЕР®

Заказ книг:
197198, Санкт-Петербург, а/я 127
тел.: (812) 703-73-74, postbook@piter.com
61093, Харьков-93, а/я 9130
тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com
www.piter.com — вся информация о книгах и веб-магазин

ISBN: 978-5-496-00356-8