# Node.js

## в действии

Майк Кантелон
Марк Хартер
TJ Головайчук
Натан Райлих

**М. Кантелон , М. Хартер, Т. Головайчук, Н. Райлих**

Node.js в действии

2014

Переводчик *А. Сергеев*

Литературный редактор *А. Жданов*

Художники *Л. Адуевская, В. Шимкевич, А. Шляго (Шантурова)*

Корректоры *Н. Викторова, В. Листова*

Верстка *Л. Родионова*

**М. Кантелон , М. Хартер, Т. Головайчук, Н. Райлих**

Node.js в действии. — СПб.: Питер, 2014.

ISBN 978-5-496-01079-5

© ООО Издательство "Питер", 2014

*Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.*

## Вступительное слово

Чтобы написать книгу о Node.js, требуются недюжинные усилия. Эта платформа является относительно новой и еще не устоялась. Ядро Node.js продолжает развиваться, а темпы роста коллекции пользовательских модулей поражают воображение. Сообщество разработчиков пока не определилось со стандартами. Книга, посвященная столь динамичному предмету, может быть успешной только в том случае, если автор понимает суть Node и способен донести до читателя причины успеха этой платформы. Майк Кантелон проявляет невиданную активность в сообществе Node-разработчиков, экспериментирует с возможностями этой платформы, читает лекции. Ему присуще великолепное чутье, позволяющее безошибочно распознать положительные и, что еще важнее, отрицательные качества Node. TJ Головайчук — один из наиболее плодотворных разработчиков модулей для Node.js, а также фантастически популярного веб-фреймворка Express. Натан Раджлич, более известный как TooTallNate, получил статус разработчика ядра Node.js и принимает активное участие в создании и развитии платформы.

Авторы книги делятся с читателем своим богатейшим опытом, начиная с установки Node.js на компьютер и заканчивая такими сложными вопросами, как создание, отладка и развертывание рабочих приложений. Вы поймете, что делает Node столь интересным и многообещающим проектом, познакомитесь с тем, как его понимают авторы, узнаете, в каких направлениях этот проект будет развиваться. Причем здесь важно то, что сложность излагаемого в книге материала растет постепенно, а теория предшествует практическим упражнениям.

Проект Node развивается столь стремительно, что напоминает набирающую скорость ракету, и авторы книги проделали громадную работу, которая позволит вам быть в курсе последних событий. Рассматривайте эту книгу в качестве стартовой площадки, которая позволит вам достигнуть новых высот в освоении Node.js.

*Исаак Шлютер (Isaac Z. Schlueter),*

*автор NPM,*

*руководитель проекта Node.js*

## Предисловие

В начале 2011 года, когда издательство Manning озадачило нас идеей написания книги о Node.js, состояние сообщества разработчиков проекта Node было далеким от нынешнего. Несмотря на то что проект Node уже в те годы начал привлекать внимание, сообщество разработчиков было малочисленным. К тому же к Node относились как к суперсовременной и малоизученной технологии. В 2011 году еще

не было ни одной книги о Node, поэтому мы взялись за этот проект, несмотря на его сложность.

Опираясь на свой опыт разработчиков, мы хотели написать книгу, в которой не только рассматривается использование Node для разработки веб-приложений, но и исследуются альтернативные области применения Node, представляющие интерес. Мы хотели, чтобы разработчики веб-приложений, использующие стандартные технологии, учли предлагаемые Node подходы при разработке серверных приложений.

Процесс написания книги занял более двух лет. На протяжении этих лет технологии неоднократно менялись, и все эти изменения отражены в книге. Помимо технологических изменений выросла численность сообщества разработчиков. К тому же Node используется во многих компаниях.

Если вы являетесь разработчиком веб-приложений и хотите опробовать новый инструмент, смело приступайте к изучению Node. Благодаря этой книге вы сможете быстро освоить новые технологии и даже получите при этом удовольствие.

## Благодарности

Выражаем благодарность сотрудникам издательства Manning за их неоценимую роль в написании этой книги. В первую очередь мы хотели бы поблагодарить Рене Грегуар (Renae Gregoire), которая помогла нам сделать материал понятным, четким и выразительным. Берт Бейтс (Bert Bates) помог нам разработать дизайн книги и представить в графическом виде излагаемые в книге концепции. Марьян Бейс (Marjan Bace) и Майкл Стефенс (Michael Stephens) помогли нам поверить в свои силы и воплотить замыслы в реальность, а также оказывали помощь во время осуществления проекта. Также мы хотели бы выразить благодарность персоналу издательства Manning, с которым нам было очень приятно работать.

Мы показывали рукопись книги на разных этапах ее написания многим людям и благодарны им за отзывы. В число этих людей входят участники программы MEAP, которые отсылали комментарии и замечания на онлайновый форум книги. Приводим список рецензентов книги, которые несколько раз прочитали рукопись и мнение и комментарии которых помогли нам улучшить книгу: Алекс Медьюрел (Alex Madurell), Берт Томас (Bert Thomas), Брэдли Мек (Bradley Meck), Бридж Панд (Braj Panda), Брайан Л. Кулей (Brian L. Cooley), Брайан Дель Веччио (Brian D Vecchio), Брайан Диллард (Brian Dillard), Брайан Эльман (Brian Ehmann), Брайа Фальк (Brian Falk), Дэниел Бретой (Daniel Bretoi), Гэри Эван Парк (Gary Ewan Parl Джереми Мартин ( Jeremy Martin), Джероен Ноувс ( Jeroen Nouws), Джерое Трапперс ( Jeroen Trappers), Кассандра Перч (Kassandra Perch), Кэвин Байстер (Kevi Baister), Майкл Пискателло (Michael Piscatello), Патрик Штегер (Patrick Steger), Пол

Стек (Paul Stack) и Скотт Банаховски (Scott Banachowski).

Выражаем благодарность Валентину Гриттаз (Valentin Crettaz). Благодарим Майкла Левина (Michael Levin) за аккуратное техническое редактирование окончательного варианта рукописи перед самой сдачей ее в производство. И конечно же, мы хотели бы выразить благодарность Исааку Шлютеру (Isaac Schlueter), руководителю проекта Node Project, за вступительное слово к книге.

**Майк Кантелон**

Я хочу выразить благодарность моему другу, Джошуа Полю ( Joshua Paul), которому я обязан своими первыми успехами в программировании. Джошуа познакомил меня с миром проектов с открытым кодом и вдохновил на написание книги. Я благодарен Мальколм, которая придала мне веру в свои силы, а также за ее терпение, проявленное во времена моего вынужденного затворничества при написании книги. Я очень благодарен родителям, которые развили во мне склонность к творчеству и познанию нового, а также заразили меню страстью к 8-разрядным компьютерам моего отрочества. Я также благодарен дедушке и бабушке, которые способствовали зарождению моей любви к программированию, подарив мне компьютер Commodore 64.

В процессе написания книги для меня просто бесценными были профессиональные знания и юмор моих соавторов, TJ Головайчука и Натана Раджлича. Я благодарен им за деловой настрой и организацию работы в команде. Огромную помощь оказал Марк Хартер (Marc Harter), взяв на себя колоссальный труд по редактированию, вычитке и написанию дополнительного текста, объединившего в единое целое материал книги.

**Марк Хартер**

Я хочу поблагодарить Райана Дала (Ryan Dahl), который около четырех лет назад серьезно заинтересовал меня JavaScript-программированием серверов. Спасибо Бену Нурдхуису (Ben Noordhuis) за бесценный ресурс, посвященный внутренней организации Node. Благодарю Берта Бейтса (Bert Bates) за создание рабочего настроя и постоянное желание оказать помощь при написании книги. Спасибо Майку, Нату и TJ за возникшее у меня желание работать по 11 часов в день. Для меня было большой честью быть с ними в одной команде. Особую благодарность выражаю Ханне, моей жене и лучшему другу, которая вдохновила меня на сей труд.

**Натан Раджлич**

Я хотел бы поблагодарить Гильермо Рауча (Guillermo Rauch), который помог мне

найти свое место в сообществе Node-разработчиков. Я хотел бы выразить благодарность Дэвиду Бликштейну (David Blickstein), вдохновившему меня на участие в этом проекте. Я благодарен Райану Далу (Ryan Dahl), зачинателю Node-движения, и Исааку Шлютеру (Isaac Schlueter), возглавляющему проект Node последние два года. Спасибо моей семье, моим друзьям и моей подруге за поддержку бессонными ночами и позитивные эмоции, наполнявшие меня энергией. Также хочу сказать огромное спасибо за поддержку моих компьютерных увлечений. Без этого я никогда бы не стал специалистом в компьютерной сфере.

## Об этой книге

Основное предназначение книги — научить читателя создавать и развертывать Node-приложения (веб-приложения). Важную часть книги занимает рассмотрение среды разработки веб-приложений Express и среды разработки приложений промежуточного уровня Connect, которые широко применяются при создании приложений и поддерживаются сообществом разработчиков. Вы также научитесь разрабатывать автоматизированные тесты и освоите принципы развертывания приложений.

Книга предназначена для опытных разработчиков веб-приложений, которые занимаются созданием чувствительных и масштабируемых Node-приложений.

Поскольку Node-приложения пишутся на языке JavaScript, обязательным условием является знание этого языка. Также рекомендуется знакомство с Windows, OS X и командной строкой Linux.

### Структура книги

Книга состоит из трех частей.

В части I рассматриваются основы Node.js и фундаментальные методики, используемые для разработки приложений на этой платформе. В главе 1 описываются характеристики Node и содержатся примеры кода. Глава 2 проведет вас поэтапно через создание примера приложения. В главе 3 рассматриваются проблемы, возникающие при разработке Node-приложений, предлагаются методики, позволяющие решить эти проблемы, приводятся способы организации кода приложения.

Часть II, которая является самой большой в книге, посвящена разработке веб-приложений. В главе 4 изучаются основы создания веб-приложений на платформе Node, а в главе 5 рассматриваются вопросы сохранения данных Node-приложениями.

Кроме того, в части II мы продолжим знакомство с миром сред разработки веб-

приложений. В главе 6 содержатся начальные сведения о среде Connect, рассматриваются ее преимущества и принципы работы. В главе 7 изучается использование различных компонентов, встроенных в среду Connect и предназначенных для добавления в веб-приложения тех или иных функциональных возможностей. В главе 8 вы познакомитесь со средой Express, а в главе 9 вас ожидают более сложные вопросы, связанные с применением этой среды разработки.

Наряду с основами разработки веб-приложений в части II рассматриваются связанные темы. Глава 10 посвящена различным вопросам применения для Node сред тестирования. В главе 11 рассматриваются вопросы шаблонизации (templating) при разработке веб-приложений в Node, что позволяет отделить представление данных от программной логики.

В части III рассматриваются задачи, которые также способна решать платформа Node (помимо разработки веб-приложений). В главе 12 рассказывается о развертывании Node-приложений на рабочих серверах, о поддержке безотказной работы и максимизации производительности. В главе 13 объясняется, каким образом могут создаваться приложения, не являющиеся HTTP-приложениями, как использовать среду Socket.io для создания приложений реального времени, как применять многочисленные прикладные программные интерфейсы, встроенные в Node. В завершающей главе 14 обсуждаются вопросы функционирования Node-сообщества и публикации Node-приложений с помощью диспетчера Node-пакетов.

## Правила оформления и загрузка примеров кода

Примеры кода, приведенные в книге, оформляются в соответствии со стандартным соглашением по оформлению JavaScript-кода. Для создания отступов в коде вместо символов табуляции применяются пробелы. Существует ограничение на длину строки кода, равное 80 символам. Код, приведенный в листингах, сопровождается комментариями, которые иллюстрируют ключевые концепции.

Каждая инструкция занимает отдельную строку и завершается точкой с запятой. Блоки кода, содержащие несколько инструкций, заключены в фигурные скобки. Левая фигурная скобка находится в первой (открывающей) строке блока. Правая фигурная скобка закрывает блок кода и находится на одном уровне с открывающей скобкой.

Примеры кода, используемые в книге, можно загрузить с веб-сайта www.manning.com/Node.jsinAction.

## Иллюстрация на обложке

Иллюстрация, помещенная на обложку книги и озаглавленная Man about Town («Прожигатель жизни»), была позаимствована из изданного в XIX веке во Франци четырехтомного каталога Сильвена Марешаля (Sylvain Marechal), включающег изображения одежды, характерной для разных регионов Франции. Каждая иллюстрация красиво нарисована и раскрашена от руки. Иллюстрации из каталога Марешаля напоминают о культурных различиях между городами и весями мира, имевшими место почти двести лет назад. Люди, проживавшие в изолированных друг от друга регионах, говорили на разных языках и диалектах. По одежде человека можно было определить, в каком городе, поселке или поселении он проживает.

С тех пор дресс-код сильно изменился, да и различия между разными регионами стали не столь выраженными. В наше время довольно трудно узнать жителей разных континентов, не говоря уже о жителях разных городов или регионов.

Сейчас, когда все компьютерные книги похожи друг на друга, издательство Manning стремится к разнообразию и помещает на обложки книг иллюстрации, показывающие особенности жизни в разных регионах Франции два века назад.

## От издательства

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

Мы будем рады узнать ваше мнение!

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

# Часть 1. Основы Node

Начиная изучать новый язык программирования или среду разработки, вы тут же сталкиваетесь с новыми понятиями, которые требуют перестройки мышления. Среда Node не является исключением из этого правила и для ее использования требуется выработка нового подхода к процессу разработки приложений.

В части I вкратце рассматриваются отличия Node от других платформ, а также излагаются основы работы с ней. Вы познакомитесь с внешним видом приложений, созданных в Node, узнаете, как они устроены, получите сведения, которые позволят вам устранять проблемы, возникающие при разработке приложений в Node. После изучения части I будет заложен фундамент, необходимый для освоения методик создания веб-приложений в Node (см. часть II), а также приложений, не относящихся к категории веб-приложений (см. часть III).

## Глава 1. Добро пожаловать в Node.js

- Знакомство с Node.js
- JavaScript на стороне сервера
- Асинхронная и событийно-ориентированная природа Node
- Типы приложений, разрабатываемых в Node
- Примеры Node-приложений

Попробуем ответить на вопрос о том, что же такое Node.js. Возможно, вы уже слышали о Node. Или уже используете эту платформу. Или хотите больше узнать о Node. Хотя эта платформа появилась в 2009 году, она уже успела стать весьма популярной среди разработчиков. Проект Node является вторым по количеству просмотров на сайте GitHub (https://github.com/joyent/node), имеет много последователей в группе Google (http://groups.google.com/group/nodejs) и в IRC-канале (http://webchat.freenode.net/?channels=node.js). В сети NPM опубликованс более 15 000 модулей, разработанных сообществом Node-разработчиков, также создан диспетчер пакетов (http://npmjs.org). Все это свидетельствует *о серьезном интересе к этой платформе*.

**РАЙАН ДАЛ, ОСНОВАТЕЛЬ NODE**

Райан Дал (Ryan Dahl) создал первую презентацию, посвященную Node. Эта презентация опубликована на веб-сайте JSCONF Berlin 2009 (http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html).

На официальном веб-сайте (http://www.nodejs.org) Node определяется как «платформа, основанная на исполняемой JavaScript-библиотеке Chrome, которая позволяет упростить создание быстрых масштабируемых сетевых приложений. В Node.js используется событийно-управляемая неблокирующая модель ввода-вывода, легковесная и эффективная, которая превосходно подходит для разработки приложений реального времени, обрабатывающих большие объемы данных и выполняемых на распределенных устройствах».

В этой главе рассматриваются следующие темы:

- почему JavaScript используется для разработки серверных приложений;
- как браузер обрабатывает ввод-вывод с помощью JavaScript;
- как Node поддерживает ввод-вывод на сервере;
- что подразумевается под DIRTy-приложениями и почему они хорошо подходят для Node;
- примеры простейших программ, разрабатываемых в Node.

Итак, сначала мы поговорим о JavaScript...

## 1.1. JavaScript

В настоящее время JavaScript является одним из наиболее популярных в мире языков программирования[1]. Если вы когда-либо занимались веб-программированием, то, скорее всего, использовали этот язык. В JavaScript нашла свое воплощение идея «однажды написанное выполняется везде», которая владела умами Java-разработчиков в 1990-е годы.

Во времена революции Ajax, которая произошла в 2005 году, JavaScript прошел путь от «игрушечного» языка до инструмента, применяемого для создания реальных программ. Первые серьезные приложения, написанные на JavaScript, — это карты Google и Gmail. Всем известные Твиттер, Фейсбук и GitHub тоже написаны на JavaScript.

После появления библиотеки Google Chrome (в конце 2008 года) производительность JavaScript многократно возросла. Этому также способствовала конкуренция между разработчиками браузеров Mozilla, Microsoft, Apple, Opera и Google. Благодаря быстродействию современных виртуальных машин, на которых устанавливается JavaScript, стремительно расширяется список веб-приложений, создаваемых на этом языке[2]. Пример серьезного JavaScript-приложения — jslinux[3]. Это реализованный на JavaScript эмулятор PC, позволяющий непосредственно в браузере загрузить ядро Linux, взаимодействовать с терминальным сеансом и компилировать C-программу.

Для работы платформы Node используется виртуальная машина V8, которая задействует Google Chrome для серверного программирования. Благодаря V8 производительность Node «взлетает до небес», поскольку устраняются промежуточные этапы создания исполняемого кода. Вместо генерирования байткода или использования интерпретатора выполняется непосредственная компиляция в собственный машинный код. В связи с тем, что Node применяет JavaScript на стороне сервера, появляются следующие преимущества:

- Разработчики могут создавать веб-приложения на одном языке, благодаря чему снижается потребность в переключении контекста при разработке серверов и клиентов. При этом обеспечивается совместное использование кода клиентом и сервером, например кода проверки данных, вводимых в форму, или кода игровой логики.

- Популярнейший формат обмена данными JSON является собственным форматом JavaScript.

- Язык JavaScript применяется в различных базах данных NoSQL (например, в CouchDB и MongoDB), поэтому подключение к таким базам данных осуществляется в естественной форме. Например, оболочкой и языком запросов для базы данных MongoDB является язык JavaScript; языком проецирования/сведения для базы данных CouchDB также является JavaScript.

- Целью компиляции в Node.js является JavaScript, к тому же в настоящее время существует ряд других языков программирования, компилируемых в JavaScript[4].

- В Node используется единственная виртуальная машина (V8), совместимая со стандартом ECMAScript[5]. Другими словами, вам не придется ожидать, пока во всех браузерах станут доступны все новые средства языка JavaScript, связанные с платформой Node.

Еще несколько лет назад разработчики даже и предположить не могли, что серверные приложения будут создаваться на JavaScript. Притягательность Node для разработчиков объясняется высокой производительностью и некоторыми другими упомянутыми ранее преимуществами. Все эти преимущества обеспечивает не только JavaScript, но и то, *как* этот язык используется в Node. Чтобы понять суть Node, начнем с рассмотрения знакомой вам JavaScript-среды — браузера.

## 1.2. Асинхронный и событийно-ориентированный браузер

В Node реализована асинхронная событийно-управляемая платформа для серверных JavaScript-приложений. JavaScript-код выполняется на сервере точно так же, как в клиентском браузере. Если вы хотите разобраться в том, как работает Node, начните с изучения принципов функционирования браузера. Как браузер, так и Node управляются событиями (используется цикл событий) и не блокируются при выполнении операций ввода-вывода (используется асинхронный ввод-вывод). Рассмотрим пример, который позволит лучше понять суть сказанного.

### ЦИКЛ СОБЫТИЙ И АСИНХРОННЫЙ ВВОД-ВЫВОД

Чтобы получить дополнительные сведения о цикле событий и асинхронном вводе-выводе, обратитесь к соответствующим статьям в Википедии (см. http://en.wikipedia.org/wiki/Event_loop и http://en.wikipedia.org/wiki/Asynchronous_I/O).

Рассмотрим общий фрагмент jQuery-кода, выполняющий Ajax-запрос с помощью XHR (XMLHttp-Request):

```
// Ввод-вывод не блокирует выполнение кода
$.post('/resource.json', function (data) {
  console.log(data);
  });
  // Продолжение выполнения сценария
```

Эта программа выполняет HTTP-запрос файла ресурсов resource.json. После выполнения запроса вызывается анонимная функция («обратный вызов» в данном контексте) с аргументом data, содержащим данные, полученные в результате запроса.

Обратите внимание, что этот код *не* был записан в следующем виде:

```
// Ввод-вывод блокирует выполнение кода
```

```
var data = $.post('/resource.json');
  console.log(data);
```

В рассматриваемом примере предполагается, что ответ на запрос файла ресурсов resource.json сохраняется в переменной data *сразу же после его появления*, а функция console.log *не вызывается до появления этого ответа*. Операция ввода-вывода (Ajax-запрос) до своего завершения «блокирует» выполнение сценария. В силу однопоточной природы браузера, если, например, запрос выполняется в течение 400 мс, на этот период времени блокируются остальные события, происходящие на странице. Вы можете вспомнить из собственного опыта примеры подобных задержек, например приостановку анимации на странице во время заполнения формы или какого-либо другого взаимодействия со страницей.

К счастью, в рассматриваемом случае все не так грустно. Операции ввода-вывода, выполняемые в браузере, происходят за пределами цикла событий (вне области выполнения основного сценария), и если «событие» происходит после завершения[6] ввода-вывода, оно обрабатывается функцией, которая часто называется функцией «обратного вызова» (рис. 1.1).

Операции ввода-вывода обрабатываются асинхронно и не «блокируют» выполнение сценария, допуская реакцию цикла событий на другие взаимодействия или запросы, возникающие на странице. В результате чувствительность браузера повышается, кроме того, становится возможной обработка большого числа интерактивных взаимодействий на странице.

А теперь перейдем к рассмотрению сервера.

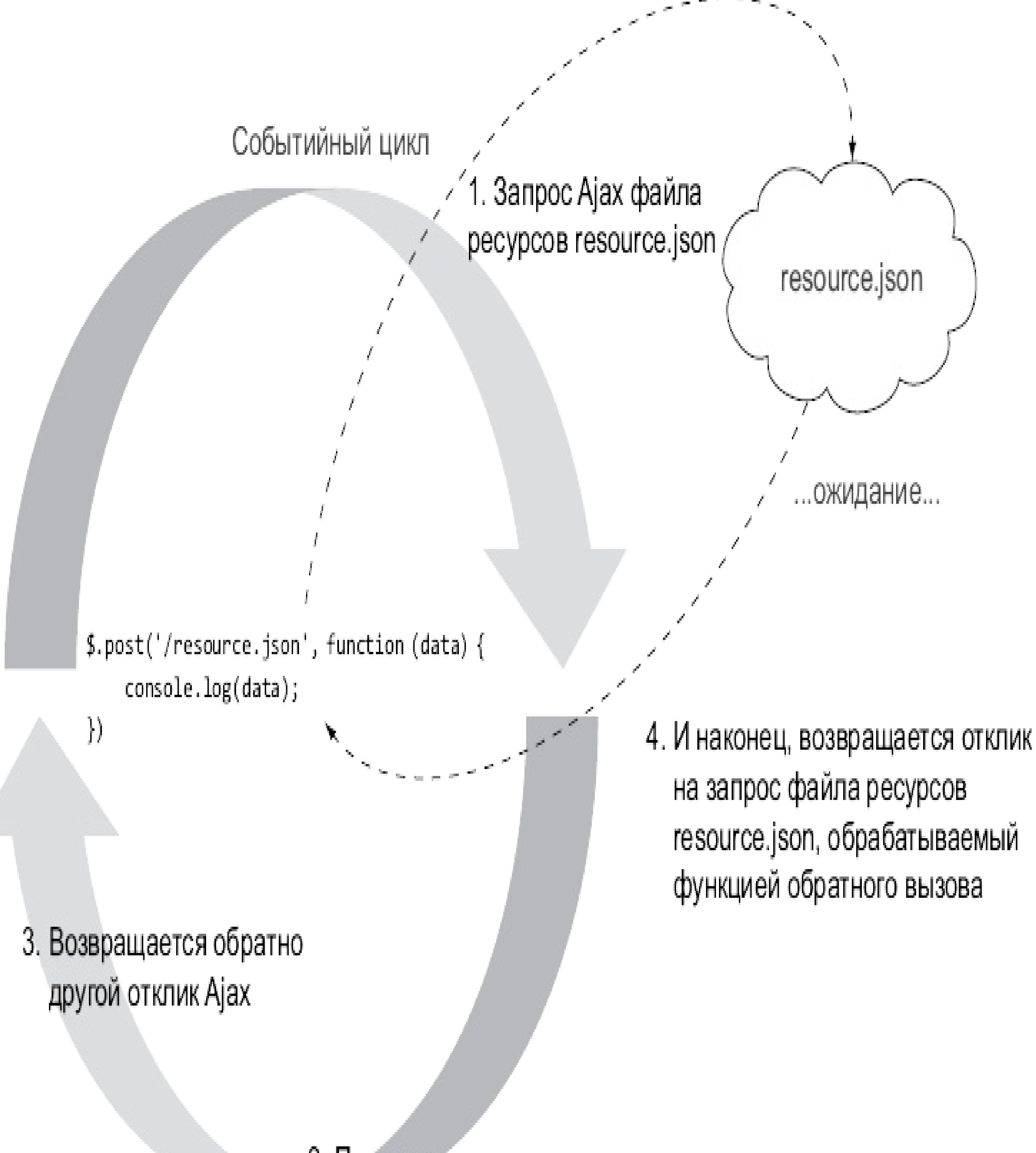



**Рис. 1.1.** Пример неблокирующего ввода-вывода в браузере

### 1.3. Асинхронный и событийно-ориентированный сервер

По большому счету, вы уже знакомы с обычной моделью ввода-вывода, применяемой в серверном программировании, например с примером «блокирующего» jQuery-кода, рассмотренным в разделе 1.2. Обратите внимание на аналогичный пример PHP-кода:

```
// Выполнение прекращается до завершения DB-запроса
$result = mysql_query('SELECT * FROM myTable');
  print_r($result);
```

Этот код реализует аналогичные операции ввода-вывода, а сам процесс блокируется до тех пор, пока все данные не вернутся обратно. В силу своей простоты эта модель подходит для многих приложений. Не сразу бросается в глаза, что процесс имеет состояние, или память, и ничего не делает до завершения ввода-вывода. Период бездействия может варьироваться от 10 мс до нескольких минут в зависимости от величины задержки, определяемой операцией ввода-вывода. Задержка может обусловливаться следующими причинами:

- Диск занят выполнением служебной операции, при этом чтение-запись данных приостанавливается.
- Скорость выполнения запроса к базе данных падает из-за увеличения нагрузки.
- В силу ряда причин скорость доступа к ресурсу, находящемуся на веб-сайте sitexyz.com, замедляется.

Если программа блокирует ввод-вывод, что же будет делать сервер в случае необходимости обработки других запросов? Обычно в подобных случаях применяется многопоточный подход. Для каждого соединения выделяется отдельный программный поток, в результате чего появляется пул потоков, выделенный для соединений. Программные потоки можно трактовать как вычислительные области, в каждой из которых процессор выполняет одно задание. Во многих случаях программный поток находится внутри процесса и поддерживает собственную рабочую память. Каждый поток обрабатывает одно либо больше соединений с сервером. И хотя на первый взгляд кажется естественным делегировать серверу обработку потоков путем создания соответствующих приложений, задача по управлению потоками в приложении может оказаться сложной. Если для обработки множества отдельных соединений с сервером требуется большое число потоков, это увеличивает потребление ресурсов операционной системы. Для поддержки программных потоков требуются быстродействующий процессор и дополнительная оперативная память.

1 Количество запросов, обрабатываемых за одну секунду.

2 Число открытых подключений клиент/сервер.

3 Программы, в которых используется асинхронный и управляемый событиями подход, такие как NGINX, могут обрабатывать больше подключений, установленных между сервером и клиентом.

4 Подобные программы также более отзывчивы. В рассматриваемом примере установлены 3500 подключений, и каждый запрос NGINX обрабатывается примерно в три раза быстрее.

**Рис. 1.2.** Сравнительная оценка быстродействия Apache- и NGINX-серверов, опубликованная на сайте WebFaction

Обратите внимание на графики производительности, которые доступны на веб-

сайте http://mng.bz/eaZT (рис. 1.2). На этих графиках сравнивается быстродействие NGINX- и Apache-серверов. NGINX-сервер (http://nginx.com/) — это HTTP-сервер, подобный Apache. Отличие от Apache заключается в том, что вместо многопоточного подхода с применением блокирующего ввода-вывода используется цикл событий с асинхронным вводом-выводом (как в браузере или в Node). В силу подобных особенностей архитектуры NGINX-сервер может обрабатывать больше запросов и поддерживает большее число подключенных клиентов. К тому же это решение является более чувствительным[7].

В Node операции ввода-вывода практически всегда выполняются за пределами основного цикла событий, поэтому в любых ситуациях сервер остается производительным и чувствительным подобно NGINX. В результате снижается вероятность блокирования процесса операциями ввода-вывода, поскольку связанные с вводом-выводом задержки не приводят к отказу сервера или к потреблению ресурсов даже при наличии блокировки. Благодаря этому исключается необходимость в создании серверов с избыточной вычислительной мощностью, требуемой для выполнения ресурсоемких операций[8].

Благодаря использованию асинхронной и событийно-управляемой модели, а также широкой распространенности языка JavaScript вы сможете получить доступ в мир разработки высокопроизводительных приложений реального времени.

## 1.4. DIRTy-приложения

Название «DIRTy» произошло от аббревиатуры DIRT (data-intensive real-time), относящейся к приложениям *реального времени*, *обрабатывающим большие объемы данных*. Так, в частности, называют приложения, разрабатываемые в Node. Поскольку в Node поддерживается неблокирующий ввод-вывод, эта платформа хорошо подходит для перемешанных или доверенных данных между каналами. Эта платформа позволяет серверу поддерживать определенное количество открытых соединений в процессе обработки множества запросов, занимая при этом небольшой объем памяти. Создаваемые на основе этой платформы приложения будут чувствительными подобно браузеру.

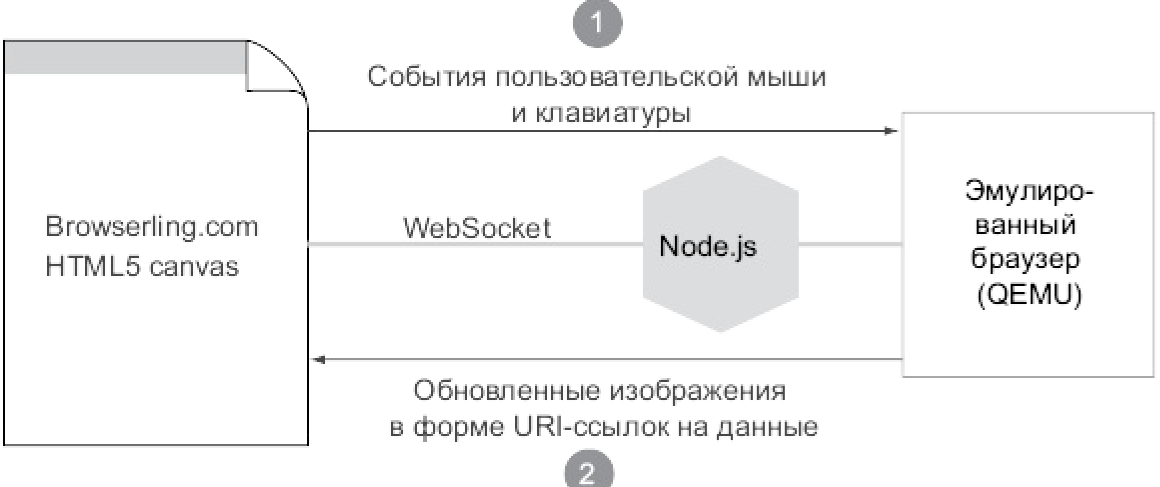
Веб-приложения реального времени появились относительно недавно. За небольшой строк они стали широко используемыми, многие современные веб-приложения выводят данные практически непрерывно. Появились приложения, предназначенные для совместной работы, отслеживания движения общественного транспорта в режиме реального времени и реализации сетевых игр. Независимо от того, были ли расширены возможности существующих приложений компонентами реального времени либо созданы полностью новые типы приложений, Интернет превратился в более чувствительную и дружественную среду. Веб-приложения

реального времени нуждаются в платформе, способной реагировать практически мгновенно в ответ на запросы одновременно подключившихся пользователей. В качестве этой среды может использоваться платформа Node, которая подходит не только для разработки веб-приложений, но и для создания других приложений с интенсивным вводом-выводом данных.

Хороший пример созданного в Node DIRTy-приложения — Browserling (browserling.com). Это приложение позволяет в окне одного браузера открывать и использовать другие браузеры (рис. 1.3). Оно чрезвычайно полезно для разработчиков внешних веб-интерфейсов, поскольку дает возможность отказаться от установки многочисленных браузеров или операционных систем исключительно для тестирования. С помощью Browserling можно задействовать созданный на платформе Node проект StackVM, который управляет виртуальными машинами (Virtual Machine, VM), разрабатываемыми с помощью эмулятора QEMU (Quick Emulator — быстрый эмулятор). QEMU эмулирует центральный процессор и периферийные устройства, требуемые для выполнения браузера.

**Рис. 1.3.** Browserling: интерактивное межбраузерное тестирование с помощью Node.js

Приложение Browserling включает для тестовых браузеров виртуальные машины, транслирующие эмулируемому браузеру данные, вводимые с помощью клавиатуры и мыши в окне пользовательского браузера. В свою очередь эмулируемый браузер создает потоки данных для перерисовываемых в окне эмулируемого браузера областей и их перерисовки на холсте пользовательского браузера (рис. 1.4).




1 В браузере события, порожденные пользовательской клавиатурой и мышью, передаются с помощью WebSocket в режиме реального времени платформе Node.js, которая в свою очередь передает их эмулятору.

2 Создаются потоки данных на основе закрашенных участков в окне эмулированного браузера, с которыми взаимодействует пользователь. Эти потоки передаются обратно с помощью Node и WebSocket и формируют изображение в окне пользовательского браузера.

**Рис. 1.4.** Последовательность операций в приложении Browserling

Приложение Browserling поддерживает также дополнительный проект Testling (testling.com), созданный на основе Node и позволяющий параллельно тестировать из командной строки несколько браузеров.

Приложения Browserling и Testling — это хорошие примеры DIRTy-приложений Инфраструктура, применяемая для создания подобных масштабируемых сетевых приложений, поможет вам при написании вашего первого приложения в Node. Давайте посмотрим, какие инструменты предлагает API-библиотека Node для разработки DIRTy-приложений.

## 1.5. Инструменты разработки DIRTy-приложений

Платформа Node была создана на основе асинхронной событийно-управляемой модели. Язык JavaScript никогда не включал стандартные библиотеки ввода-вывода, которые обычно используются в серверных языках. Ввод-вывод в JavaScript всегда задавался с помощью среды «хоста». Наиболее распространенная хост-среда для JavaScript, используемая большинством разработчиков, — браузер, который является событийно-управляемым и асинхронным.

Платформа Node поддерживает совместимость между браузером и сервером путем повторной реализации таких общих хост-объектов, как:

- API таймера (например, setTimeout);

• API консоли (например, console.log).

Платформа Node также включает базовый набор модулей для многих типов сетей и файлового ввода-вывода. В этот набор включены модули для HTTP, TLS HTTPS, filesystem (POSIX), Datagram (UDP) и NET (TCP). Этот базовый набор (яд умышленно сделан компактным, низкоуровневым и несложным по структуре; он включает лишь «строительные блоки» для приложений, реализующих ввод-вывод. Модули от независимых производителей, созданные на основе этих блоков, являются более абстрактными и позволяют решать задачи общего характера.

**ПЛАТФОРМА ИЛИ среда?**

Node — это платформа, а не среда разработки JavaScript-приложений. Распространенной ошибкой является отождествление Node со средой Rails или Django for JavaScript, так как на самом деле Node является гораздо более низкоуровневым инструментом.

Если же вы интересуетесь средами разработки веб-приложений, прочитайте соответствующий раздел книги, в котором рассматривается одна из популярных сред Node-разработки под названием Express.

После теоретического вступления рассмотрим код, создаваемый с помощью Node. В следующих разделах представлено несколько простых примеров:

- простой пример реализации асинхронного ввода-вывода;
- веб-сервер Hello World;
- пример реализации потоков данных.

Сначала мы рассмотрим пример реализации асинхронного ввода-вывода.

### 1.5.1. Простой пример реализации асинхронного ввода-вывода

В разделе 1.2 рассматривался пример Ajax-запроса, созданного с помощью jQuery:

```
$.post('/resource.json', function (data) {
  console.log(data);
```

```
});
```

Теперь мы создадим подобный код с помощью Node, но в этом случае файл ресурсов resource.json будет загружаться с диска с помощью модуля filesystem (fs). Обратите внимание на сходство между этим и предыдущим примером кода, созданным с помощью jQuery:

```
var fs = require('fs');
fs.readFile('./resource.json', function (er, data) {
  console.log(data);
})
```

При выполнении этой программы с диска загружается файл ресурсов resource.json. После считывания всех данных вызывается анонимная функция (функция обратного вызова), включающая аргументы er (коды ошибок) и data (файл данных).

Цикл событий, работающий в фоновом режиме, может реализовать ряд других операций, требуемых для подготовки данных. Все рассматриваемые ранее преимущества, связанные с событийно-управляемым и асинхронным выполнением, реализуются автоматически. Различия, имеющие место в данном случае, связаны с тем, что вместо создания Ajax-запроса с помощью jQuery из браузера осуществляется доступ к объекту filesystem в Node, реализующему доступ к файлу resource.json. Выполнение последнего действия иллюстрирует рис. 1.5.

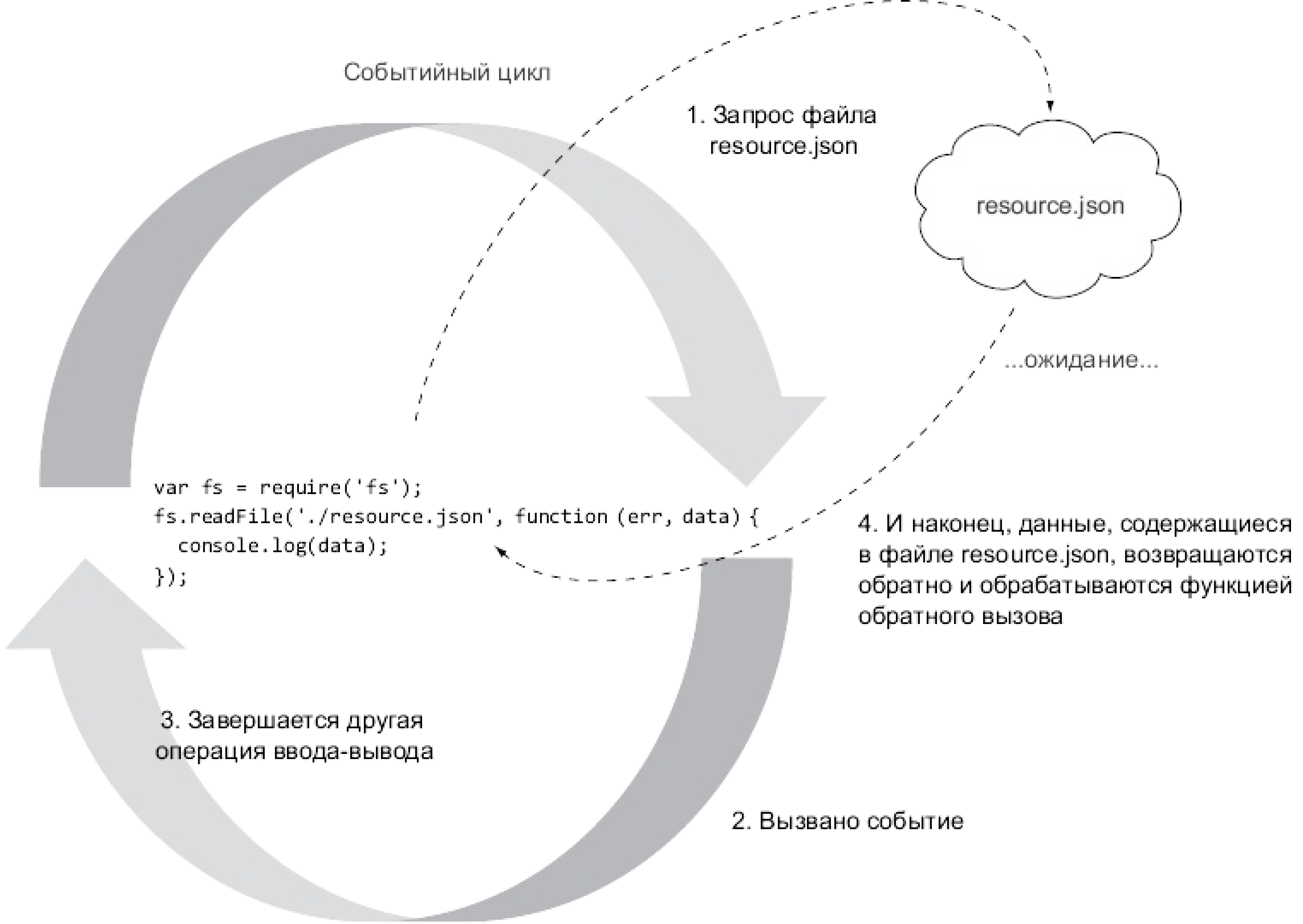



**Рис. 1.5.** Пример неблокирующего ввода-вывода в Node

### 1.5.2. HTTP-сервер Hello World

Обычно Node применяется для программирования серверов. В Node очень просто создавать различные типов серверов. Если вы ранее занимались программированием серверов, то знаете, что сервер является хостом для приложения (например, для PHP-приложения хостом является HTTP-сервер Apache). В Node между сервером и приложением *нет никакой разницы*.

Вот пример HTTP-сервера, который в ответ на любой запрос выводит сообщение «Hello World»:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

В ответ на запрос вызывается функция обратного вызова function (req, res) и на экране появляется ответ «Hello World». Эта модель событий напоминает прослушивание события onclick в браузере. Поскольку щелчок мышью может произойти в любой момент, следует воспользоваться функцией, реализующей нужную для этого случая логику. В Node имеется такая функция, которая откликается на поступивший в любой момент запрос.

А теперь рассмотрим другой способ создания того же самого сервера, который позволит сделать запрос событий более явным:

```
var http = require('http');
var server = http.createServer();
// Настройка слушателя событий для запроса
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
})
server.listen(3000);
console.log('Server running at http://localhost:3000/');
```

### 1.5.3. Реализация потоков данных

Платформа Node подходит для программирования потоков данных и организации их передачи. Потоки данных можно представлять как массивы, но вместо распределения данных в области (в массиве) выполняется распределение данных *во времени*. Благодаря передаче данных фрагмент за фрагментом разработчик получает возможность обрабатывать данные по мере их накопления, а не ждать, пока будут переданы все данные, а потом выполнять какие-либо действия. Обратите внимание на организацию передачи потоков данных для файла ресурсов resource.json:

```
var stream = fs.createReadStream('./resource.json')
// Событие Data вызывается после появления нового фрагмента данных
stream.on('data', function (chunk) {
  console.log(chunk)
})
stream.on('end', function () {
  console.log('finished')
})
```

Событие data вызывается после появления нового фрагмента данных, а событие

end — после загрузки всех фрагментов кода. Фрагменты данных могут иметь разные размеры (в зависимости от типа данных). Благодаря чтению потока данных на низком уровне обеспечивается более эффективная обработка данных, чем ожидание передачи всех данных в буфер памяти.

В Node также поддерживаются записываемые потоки данных, позволяющие записывать фрагменты данных. Один из подобных потоков — объект ответа (res), генерируемый в ответ на запрос к HTTP-серверу.

Считываемые и записываемые потоки могут соединяться, образуя каналы, как в случае использования оператора | в языке написания сценариев оболочки. При этом обеспечивается эффективный способ записи только что считанных данных без ожидания считывания и сохранения всего ресурса.

Рассмотрим пример HTTP-сервера, взятый из предыдущего раздела. В данном случае он будет передавать клиенту поток данных изображения:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'image/png'});
  // Передача по каналам из считываемого потока в записываемый
  fs.createReadStream('./image.png').pipe(res);
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

При выполнении этого примера кода данные считываются из файла (fs.createReadStream) и отсылаются (.pipe) клиенту (res) по мере считывания. Цикл событий может обрабатывать другие события, пока данные передаются в потоке.

Node поддерживает описанный подход (когда механизм DIRTy предлагается по умолчанию) на различных платформах, включая разные версии UNIX и Windows. Базовая библиотека асинхронного ввода-вывода (libuv) была создана для того, чтобы поддерживать универсальную методику обработки независимо от родительской операционной системы. В результате обеспечивается упрощенный перенос программ на другие устройства, на которых они могут выполняться.

## 1.6. Резюме

Как и любая другая технология, Node не является «панацеей от всех бед». Эта платформа поможет вам решить определенные проблемы и откроет новые возможности. Одна из интересных особенностей платформы Node заключается в том, что она открывает доступ ко всем аспектам веб-разработки. Одни разработчики приходят к Node, будучи программистами клиентских JavaScript-

приложений; другие занимаются программированием серверов, третьи являются системными программистами. Кем бы вы ни были, мы надеемся, вы по достоинству оцените Node.

Особенности платформы Node:

- она построена на языке JavaScript.
- она асинхронная и событийно-управляемая;
- она предназначена для создания приложений реального времени, обрабатывающих большие объемы данных (DIRTy-приложений).

В главе 2 мы создадим простое веб-приложение, которое проиллюстрирует принципы работы Node.

## Глава 2. Создание приложения для многокомнатного чата

- Знакомство с компонентами Node
- Пример приложения реального времени, использующего Node
- Взаимодействие между сервером и клиентом

В главе 1 рассматривались различия между разработкой асинхронных приложений с помощью Node и созданием обычных синхронных приложений. В этой главе мы взглянем на Node с практической стороны, а также создадим небольшое событийно-управляемое приложение для чата. Не беспокойтесь, если не сможете понять некоторые детали разработки приложения. Ваша задача — познакомиться с принципами разработки приложений в Node и получить представление о возможностях, которыми вы будете обладать по завершении чтения этой книги.

Материал главы предполагает наличие у вас опыта разработки веб-приложений, базовых знаний HTTP и знакомства с jQuery. По мере чтения главы вас ожидает следующее:

- обзор приложения для чата и рассмотрение принципов его работы;
- знакомство с технологическими требованиями и первоначальная настройка приложения;

- обслуживание файлов приложения: HTML-, CSS- и JavaScript-файлов на стороне клиента;

- обработка сообщений чата с помощью библиотеки Socket.IO;

- применение языка JavaScript на стороне клиента для разработки пользовательского интерфейса приложения.

Начнем с обзора приложения. Вам предстоит познакомиться с внешним видом приложения. Также вы узнаете о том, как будет себя вести готовое приложение.

## 2.1. Знакомство с приложением

Созданное в этой главе приложение обеспечит пользователям возможность общения друг с другом в Интернете путем ввода сообщений в простую форму (рис. 2.1). Введенное сообщение смогут просмотреть все пользователи, находящиеся в соответствующей комнате чата.

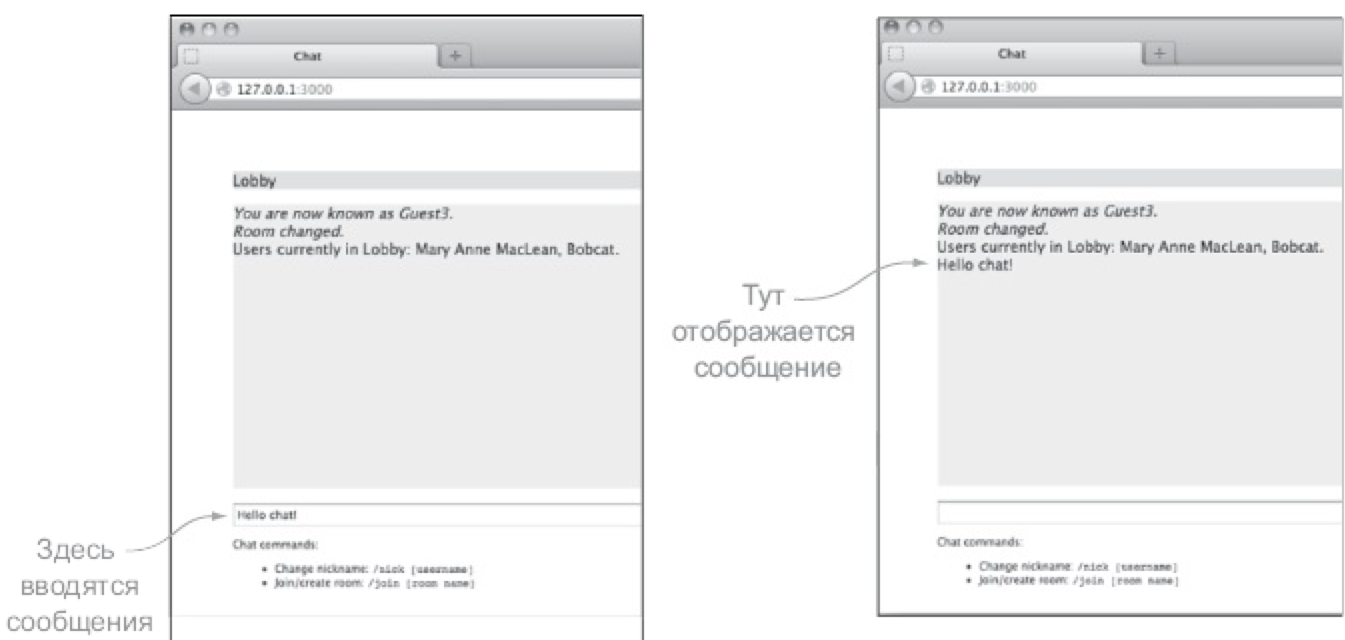



**Рис. 2.1.** Ввод сообщения в окне чата

Сразу же после запуска приложения пользователю автоматически присваивается гостевое имя (Guest). Это имя можно изменить путем ввода соответствующей команды (рис. 2.2). Команды чата предваряются символом прямого слеша (/).

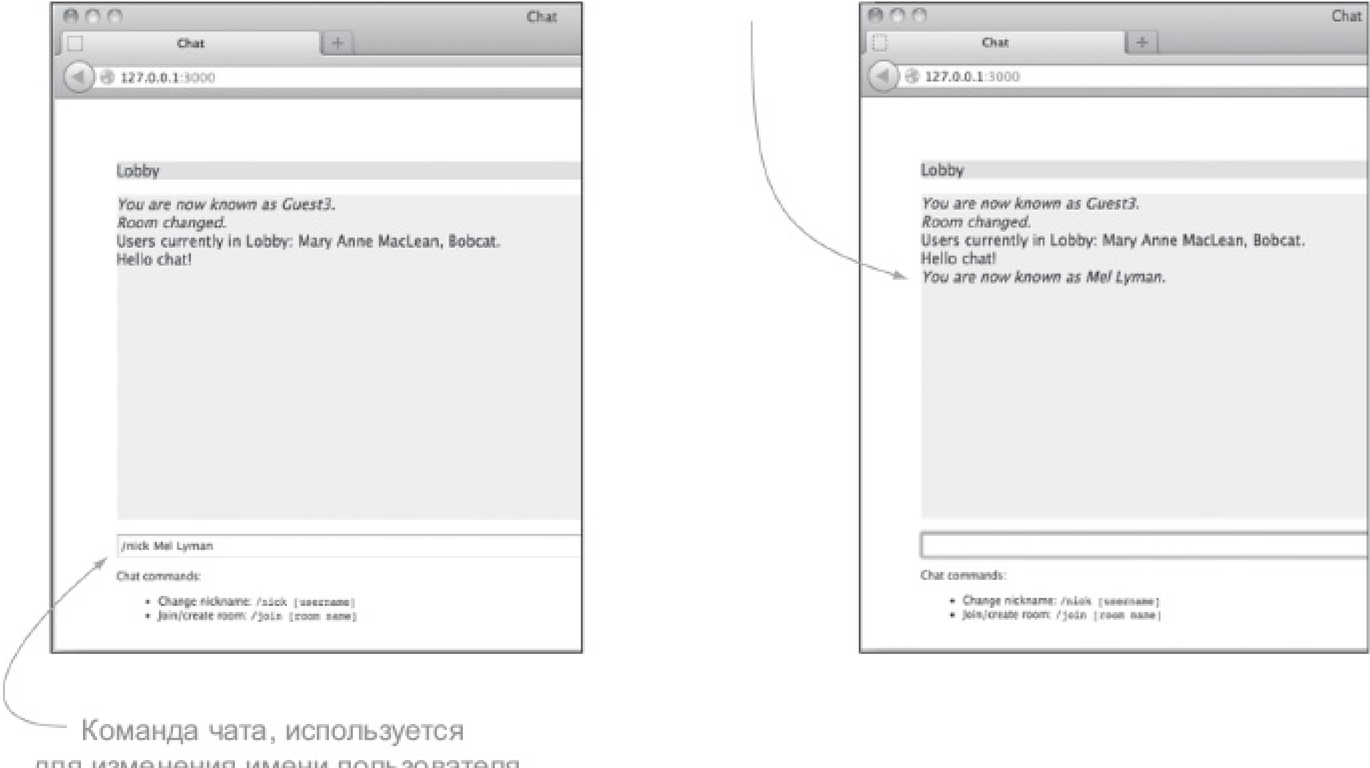



**Рис. 2.2.** Изменение имени пользователя чата

Пользователь также может ввести команду, чтобы создать новую комнату чата (либо присоединиться к существующей комнате), как показано на рис. 2.3. После присоединения к комнате чата (входа в комнату чата) или создания новой комнаты соответствующее имя отображается на горизонтальной панели, находящейся в верхней части окна приложения чата. Комната чата также включает список доступных комнат, который отображается в правой части области сообщений чата.

Текущая комната

Chat

Lobby

You are now known as Guest3.
Room changed.
Users currently in Lobby: Mary Anne MacLean, Bobcat.
Hello chat!
You are now known as Mel Lyman.

Lobby

Список комнат, созданных всеми пользователями

/join Lounge

Send

Chat commands:

Команда чата, используемая для создания комнаты чата или присоединения к существующей комнате чата

**Рис. 2.3.** Изменение комнат чата

После создания новой комнаты система попросит подтвердить необходимость изменения (рис. 2.4).

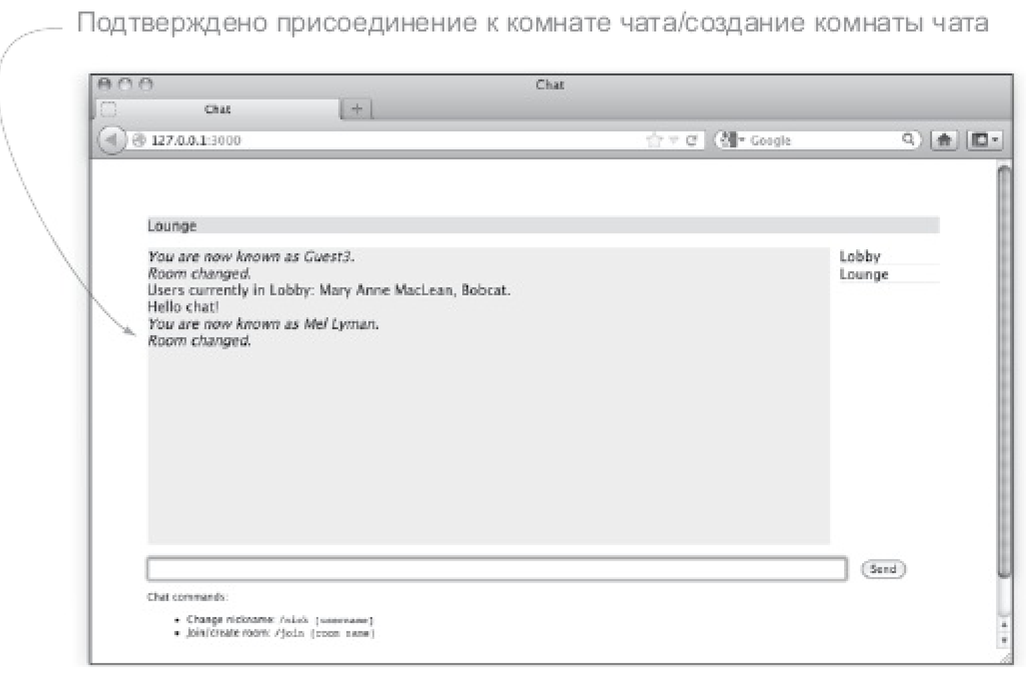



**Рис. 2.4.** Результат создания новой комнаты чата/присоединения к комнате чата

Несмотря на ограниченную функциональность приложения, оно прекрасно иллюстрирует важные компоненты и фундаментальные методики, применяемые при создании приложений реального времени. На примере этого приложения

демонстрируется, каким образом Node может одновременно обслуживать обычные HTTP-данные (например, статические файлы) и данные, получаемые в режиме реального времени (сообщения чата). Также демонстрируется структура приложений в Node и способы управления зависимостями.

А теперь рассмотрим технологии, требуемые для реализации приложения.

### 2.2. Требования к приложению и начальная настройка

Создаваемое в этой главе приложение для чата должно соответствовать следующим требованиям:

- обслуживать статические файлы (такие, как HTML-, CSS- и JavaScript-файлы на стороне клиента);
- обрабатывать сообщения чата на сервере;
- обрабатывать связанные с чатом сообщения в веб-браузере пользователя.

Чтобы обслуживать статические файлы, нужно обратиться к встроенному Node-модулю http. При этом недостаточно просто отправлять содержимое файла, следует также указать тип отсылаемого файла. Для выполнения этой операции в HTTP-заголовке Content-Type нужно указать корректный MIME-тип файла. Чтобы просмотреть доступные MIME-типы, воспользуйтесь модулем mime от независимого производителя.

**MIME-ТИПЫ**

Mime-типы подробно рассмотрены в статье Википедии по адресу http://ru.wikipedia.org/wiki/MIME.

Для обработки сообщений чата можно опрашивать сервер путем передачи Ajax-запросов. Но если нужно сделать приложение более чувствительным, лучше этого не делать, поскольку Ajax в качестве механизма передачи данных использует протокол HTTP, который не предназначен для поддержки обмена данными в режиме реального времени. Если сообщения отсылаются по протоколу HTTP, создается новое TCP/IP-соединение. Открытие и закрытие соединений требует времени, к тому же увеличивается размер передаваемых данных, поскольку каждый запрос содержит HTTP-заголовок. Поэтому вместо решения на базе HTTP

данном приложении используется реализация протокола WebSocket (http://ru.wikipedia.org/wiki/WebSocket) — легковесного протокола двусторонней связи, поддерживающего обмен данными в режиме реального времени.

Поскольку поддержка протокола WebSocket гарантируется лишь в браузерах, совместимых с HTML5, в приложении используется популярная библиотека Socket.IO (http://socket.io/), которая включает дополнительную функциональность, компенсирующую отсутствие непосредственной поддержки WebSocket в браузере. В частности, включен режим поддержки технологии Flash. Реализация дополнительной функциональности в Socket.IO не требует добавления соответствующего кода или какой-либо настройки. Библиотека Socket.IO подробно рассматривается в главе 13.

Прежде чем заняться предварительной настройкой файловой структуры и зависимостей приложения, рассмотрим, каким образом в Node можно одновременно обрабатывать HTTP- и WebSocket-запросы, — именно благодаря наличию подобной возможности Node применяют для создания приложений реального времени.

### 2.2.1. Обслуживание HTTP- и WebSocket-запросов

Хотя мы постараемся не применять в приложении технологию Ajax для отправки/получения сообщений чата, в нем по-прежне му сохранится поддержка протокола HTTP, требуемого для соответствующей настройки, выполняемой в пользовательском браузере с помощью HTML-, CSS- и JavaScript-файлов на стороне клиента.

В Node допускается одновременное обслуживание HTTP- и WebSocket-запросов с помощью единственного порта TCP/IP (рис. 2.5). В пакет Node включен модуль, реализующий обслуживание HTTP-запросов. Также имеются модули от независимых производителей (например, Express), которые расширяют функциональность Node, облегчая обслуживание веб-запросов. В главе 8 подробно рассматривается использование модуля Express для создания веб-приложений. В этой главе даны лишь начальные сведения о работе с модулем Express.

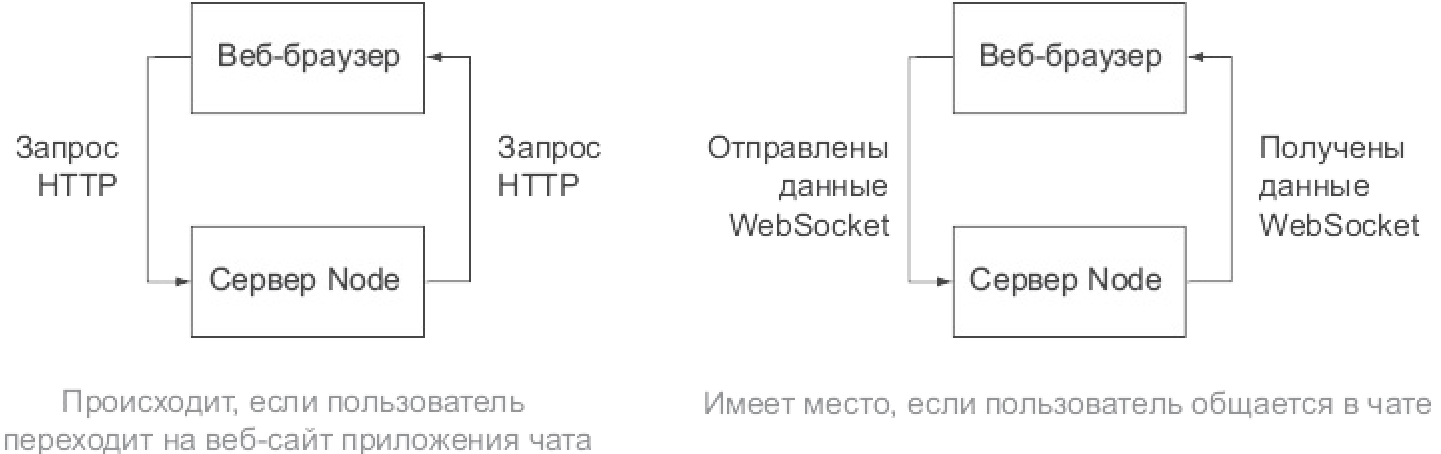



**Рис. 2.5.** Обработка HTTP- и WebSocket-запросов в рамках одного приложения

Теперь, познакомившись с теорией, можно приступать к разработке приложения.

**УСТАНОВКа Node**

Если вы до сих пор не установили Node, обратитесь к приложению А за инструкциями.

### 2.2.2. Создание файловой структуры приложения

Прежде чем приступать к разработке учебного приложения, создадим папку проекта. В этой папке будет находиться основной файл приложения. Также нужно создать подпапку lib, в которой будет размещаться некоторая серверная логика (код), и вложенную папку public, предназначенную для файлов, имеющих отношение к клиентской части. В папке public должны быть также созданы вложенные папки javascripts и stylesheets.

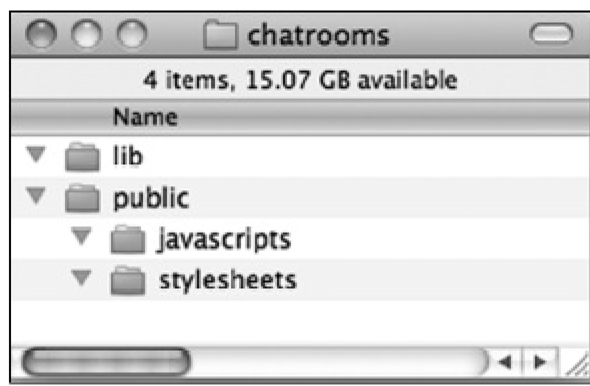



**Рис. 2.6.** Структура папок, выбранная для приложения

Структура папок должна выглядеть так, как показано на рис. 2.6. Обратите внимание, что подобная структура папок характерна именно для данной главы.

Однако поскольку в Node придерживаться подобной структуры при разработке приложений не обязательно, создавайте собственные папки и размещайте их так, как вам больше нравится.

Завершив создание структуры папок, можно приступать к определению зависимостей приложения.

В рассматриваемом контексте *зависимость приложения* — это модуль, который нужно установить, чтобы обеспечить необходимую для приложения функциональность. Предположим, например, что разрабатывается приложение, которое нуждается в доступе к данным, хранящимся в базе данных MySQL. Поскольку в Node отсутствует модуль, обеспечивающий доступ к MySQL, нужно установить соответствующий модуль от независимого производителя. Именно этот модуль и рассматривается как зависимость.

### 2.2.3. Спецификация зависимостей

Несмотря на то что в Node можно создавать приложения без формальной спецификации зависимостей, все же лучше найти время, чтобы выполнить эту операцию. В результате другим пользователям будет проще работать с вашим приложением. Если же вы планируете выполнять приложение на нескольких разных платформах, спецификация зависимостей облегчает выполнение первоначальной настройки.

Для спецификации зависимостей приложения используется файл package.json, находящийся в корневой папке приложения. Файл package.json включает JSON-выражение, создаваемое в соответствии со стандартом дескрипторов пакета CommonJS (http://wiki.commonjs.org/wiki/Packages/1.0) и описывающее приложение. В файле package.json можно задать множество параметров, наиболее важными из которых являются имя приложения, версия приложения, описание функциональных возможностей приложения и зависимости приложения.

В листинге 2.1 представлен файл дескрипторов пакета, описывающий функциональность и зависимости нашего учебного приложения. Сохраните этот файл под названием package.json в корневой папке учебного приложения.

**Листинг 2.1.** Файл дескрипторов пакета

```
{
  "name": "chatrooms",
  // Название пакета
  "version": "0.0.1",
  "description": "Minimalist multiroom chat server",
  // Зависимости пакета
```

```
  "dependencies": {
    "socket.io": "~0.9.6",
    "mime": "~1.2.7"
  }
}
```

Если содержимое файла выглядит немного непонятным, не волнуйтесь, структура файла package.json вкратце рассматривается в следующей главе, а в главе 14 она описывается более подробно.

### 2.2.4. Установка зависимостей

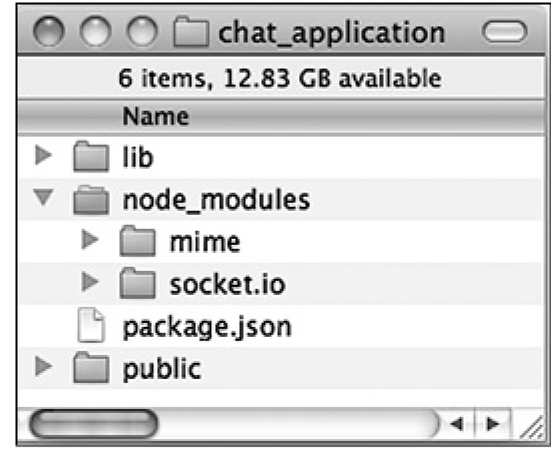



**Рис. 2.7.** С помощью команды npm устанавливаются зависимости и создается папка node_modules

После определения файла package.json установка зависимостей приложения не составляет особого труда. При этом используется поставляемая с Node утилита Node Package Manager (npm;https://github.com/isaacs/npm). Эта утилита предлагает пользователям разнообразную функциональность, позволяя легко устанавливать Node-модули от независимых производителей и выполнять глобальную публикацию Node-модулей, созданных вами. Еще одно преимущество заключается в том, что вы можете считывать зависимости из файлов package.json и устанавливать их с помощью единственной команды.

Перейдите в корневую папку учебного примера и введите следующую команду:

```
npm install
```

Если теперь просмотреть папку учебного примера, вы увидите только что созданную папку node_modules, в которой указаны зависимости приложения (рис. 2.7).

После создания структуры папок и настройки зависимостей можно приступать к разработке логики приложения.

## 2.3. Обслуживание приложением HTML-, CSS- и JavaScript-файлов на стороне клиента

Как упоминалось ранее, клиентское приложение должно уметь делать три базовые вещи:

- обслуживать статические файлы в пользовательском веб-браузере;
- обрабатывать сообщения чата на стороне сервера;
- обрабатывать сообщения чата в пользовательском веб-браузере.

Логику приложения можно реализовать на базе нескольких файлов, некоторые из них выполняются на сервере, а другие — на клиенте (рис. 2.8). JavaScript-файлы, выполняемые на стороне клиента, должны использоваться как статические ресурсы, а не вызываться на выполнение платформой Node.

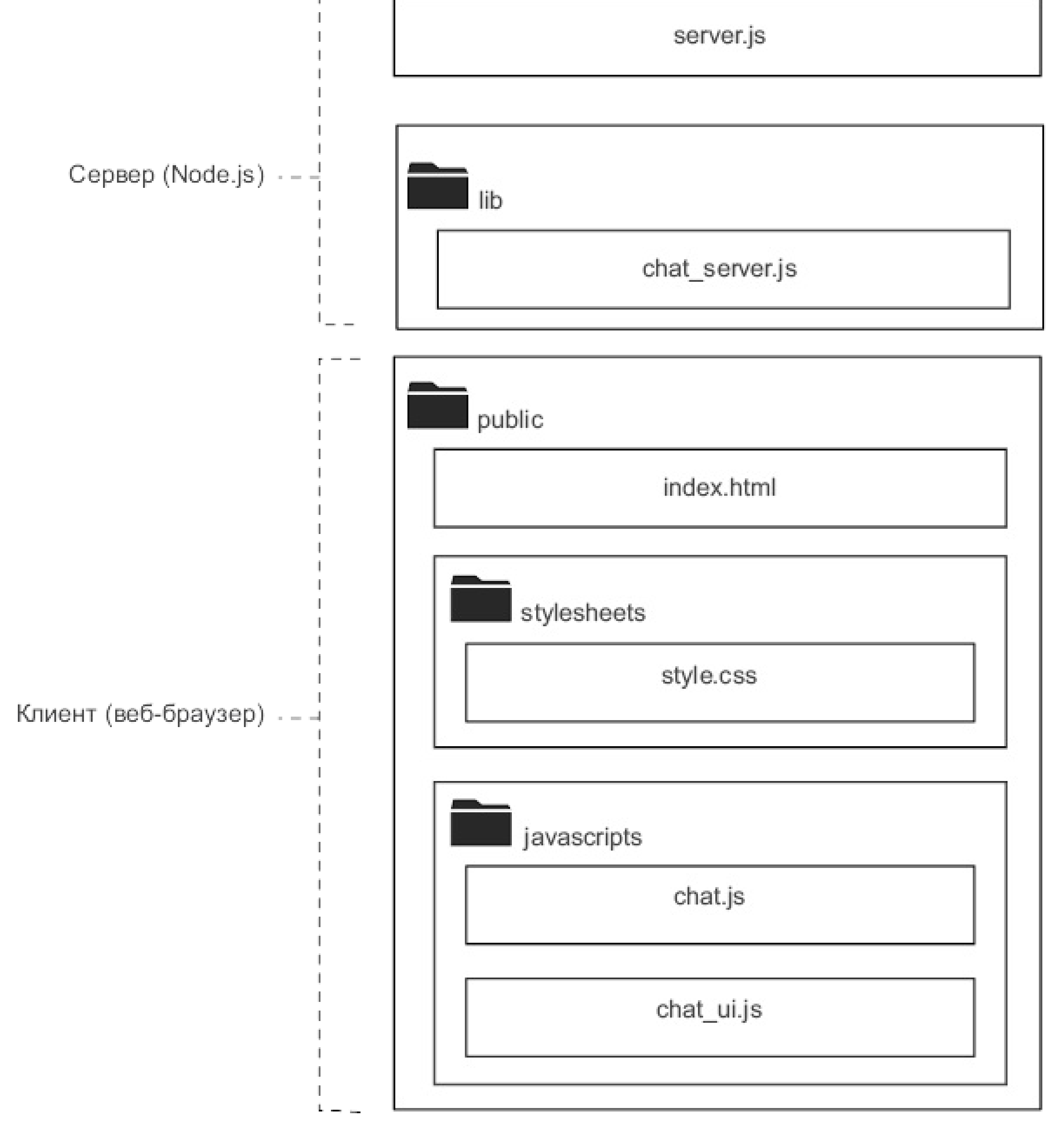


**Рис. 2.8.** В приложении реализована клиентская и серверная JavaScript-логика

В следующем разделе будет рассмотрен код, требуемый для использования статических файлов. Затем будет добавлен статический код HTML и CSS.

### 2.3.1. Создание базового статического файлового сервера

Чтобы создать статический файловый сервер, нужно воспользоваться встроенной в Node функциональностью, а также надстройкой mime от независимого производителя (для определения MIME-типа файла).

Чтобы приступить к формированию основного файла приложения, в корневой папке проекта создайте файл server.js, а затем поместите в этот файл объявления переменных из листинга 2.2. С помощью этих объявлений обеспечивается доступ к Node-функциональности, связанной с HTTP, к средствам взаимодействия с файловой системой, к функциональности для работы с файловыми путями, а также к средствам определения MIME-типа файла. Переменная cache служит для кэширования данных файла.

**Листинг 2.2.** Объявления переменных

```
// Встроенный модуль http поддерживает
// функциональность HTTP-сервера и HTTP-клиента
var http = require('http');
// Встроенный модуль fs поддерживает функциональность файловой систем
var fs = require('fs');
// Встроенный модуль path поддерживает функциональность, связанную
// с путями файловой системы
var path = require('path');
// Дополнительный модуль mime поддерживает порождение MIME-типов
// на основе расширения имен файлов
var mime = require('mime');
// Объект cache реализует хранение содержимого кэшированных файлов
var cache = {};
```

#### *Отправка данных файла и ответов на ошибки*

А теперь нужно добавить три вспомогательные функции, выполняющие обслуживание HTTP-файлов. Первая функция отправляет код ошибки 404, если запрашиваемый файл отсутствует. Добавьте код этой функции в файл server.js:

```
function send404(response) {
```

```
  response.writeHead(404, {'Content-Type': 'text/plain'});
  response.write('Error 404: resource not found.');
  response.end();
}
```

Вторая вспомогательная функция обслуживает данные файла. Сначала она записывает в файл соответствующие HTTP-заголовки, а затем отсылает содержимое файла. В файл server.js добавьте следующий код:

```
function sendFile(response, filePath, fileContents) {
  response.writeHead(
    200,
    {"content-type": mime.lookup(path.basename(filePath))}
  );
  response.end(fileContents);
}
```

Доступ к оперативной памяти (RAM) выполняется намного быстрее, чем к файловой системе. Поэтому Node-приложения обычно кэшируют часто используемые данные в оперативной памяти. Приложение для чата кэширует статические файлы в оперативной памяти, считывая их с диска при первом обращении. Следующая вспомогательная функция определяет, кэширован файл или нет. Если да, файл обслуживается. Если нет, файл сначала считывается с диска, а затем обслуживается. Если файл не существует, в качестве отклика возвращается HTTP-ошибка 404. Добавьте код этой вспомогательной функции в файл server.js.

**Листинг 2.3.** Использование статических файлов

```
function serveStatic(response, cache, absPath) {
  // Проверка факта кэширования файла в памяти
  if (cache[absPath]) {
    // Обслуживание файла, находящегося в памяти
    sendFile(response, absPath, cache[absPath]);
  } else {
    // Проверка факта существования файла
    fs.exists(absPath, function(exists) {
      if (exists) {
        // Считывание файла с диска
        fs.readFile(absPath, function(err, data) {
          if (err) {
            send404(response);
```

```
      } else {
        cache[absPath] = data;
        // Обслуживание файла, считанного с диска
        sendFile(response, absPath, data);
      }
    });
  } else {
    // Отсылка HTTP-ответа 404
    send404(response);
  }
 });
}
}
```

***Создание HTTP-сервера***

Для HTTP-сервера анонимная функция задается аргументом create-Server и выступает в качестве функции обратного вызова, определяющей способ обработки каждого HTTP-запроса. Функция обратного вызова принимает два аргумента request и response. После выполнения обратного вызова HTTP-сервер присваивает этим аргументам объекты, которые позволяют соответственно проработать детали запроса и отправить обратно ответ. Модуль http подробно рассматривается в главе 4.

Чтобы создать HTTP-сервер, добавим в файл server.js код, содержащийся в листинге 2.4.

**Листинг 2.4.** Код создания HTTP-сервера

```
 // Создание HTTP-сервера с помощью анонимной функции,
 // определяющей его поведение при выполнении запросов
var server = http.createServer(function(request, response) {
 var filePath = false;
 if (request.url == '/') {
   // Определение HTML-файла, обслуживаемого по умолчанию
   filePath = 'public/index.html';
 } else {
   // Преобразование URL-адреса в относительный путь к файлу
   filePath = 'public' + request.url;
```

```
  }
  var absPath = './' + filePath;
  // Обслуживание статического файла
  serveStatic(response, cache, absPath);
});
```

### *Запуск HTTP-сервера*

В предыдущем разделе мы написали код HTTP-сервера, осталось добавить логику которая запускает этот сервер на выполнение. Следующие строки кода запускают сервер, прослушивающий TCP/IP-порт под номером 3000. Выбор этого порта произошел случайно, на самом деле можно выбрать произвольный неиспользуемый порт, номер которого превышает 1024. Можно также задействовать порты, номера которых меньше 1024, если вы работаете на платформе Windows. Если же применяется Linux или OS X, то для использования портов с номерами, меньшими 1024, нужно запускать приложение от имени учетной записи привилегированного пользователя (например, root).

```
server.listen(3000, function() {
  console.log("Server listening on port 3000.");
});
```

Если вы хотите увидеть результат выполнения приложения, запустите сервер, введя в командной строке следующий код:

```
node server.js
```

После запуска сервера на выполнение в адресной строке веб-браузера введите ссылку http://127.0.0.1:3000. В результате будет вызвана соответствующая вспомогательная функция и на экране появится сообщение Error 404: resource not found (Ошибка 404: ресурс не найден). Появление этой ошибки объясняется отсутствием статических файлов. Чтобы остановить выполняющийся сервер, в командной строке нажмите комбинацию клавиш Ctrl+C.

А теперь добавим статические файлы, которые расширят функциональность приложения для чата.

## 2.3.2. Добавление HTML- и CSS-файлов

Начнем с добавления статического файла, содержащего базовый HTML-код. Создайте в общей папке файл index.html и поместите в этот файл код из листинга 2.5. HTML-код будет подключать CSS-файл, настраивать HTML-элементы div задающие вывод контента приложения, и загружать несколько клиентских

JavaScript-файлов, поддерживающих на стороне сервера функциональность Socket.IO, jQuery (для облегчения выполнения DOM-операций), а также пару специфичных для приложения файлов, обеспечивающих функциональность чата.

**Листинг 2.5.** HTML-код приложения

```
<!doctype html>
<html lang='en'>

<head>
  <title>Chat</title>
  <link rel='stylesheet' href='/stylesheets/style.css'></link>
</head>

<body>
<div id='content'>
  // Элемент div, определяющий вывод названия текущей комнаты
  <div id='room'></div>

  // Элемент div, задающий вывод списка свободных комнат
  <div id='room-list'></div>
  // Элемент div, определяющий вывод сообщений чата
  <div id='messages'></div>

  <form id='send-form'>
    // Элемент input формы, в котором пользователь вводит
    // команды и сообщения
    <input id='send-message' />
    <input id='send-button' type='submit' value='Send'/>

    <div id='help'>
    Chat commands:
    <ul>
      <li>Change nickname: <code>/nick [username]</code></li>
      <li>Join/create room: <code>/join [room name]</code></li>
    </ul>
  </div>
```

```
</form>
</div>

<script src='/socket.io/socket.io.js' type='text/javascript'></script>
<script src='http://code.jquery.com/jquery-1.8.0.min.js'
  type='text/javascript'></script>
<script src='/javascripts/chat.js' type='text/javascript'></script>
<script src='/javascripts/chat_ui.js' type='text/javascript'></script>
</body>
</html>
```

Следующий файл служит для определения CSS-стилей, используемых в приложении. В папке public/stylesheets создайте файл style.css и поместите в него код из листинга 2.6.

**Листинг 2.6.** CSS-стили, используемые в приложении

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}

a {
  color: #00B7FF;
}

// Ширина окна приложения составляет 800 пикселов, применено
// центрирование по горизонтали
#content {
  width: 800px;
  margin-left: auto;
  margin-right: auto;
}

// CSS-правила для области, в которой выводится название текущей комнаты
#room {
  background-color: #ddd;
```

```
  margin-bottom: 1em;
}

// Область вывода сообщений имеет 690 пикселов в ширину
// и 300 пикселов в высоту
#messages {
  width: 690px;
  height: 300px;
  // Обеспечивает прокрутку выводимых сообщений
  // после заполнения соответствующей области
  overflow: auto;
  background-color: #eee;
  margin-bottom: 1em;
  margin-right: 10px;
}
```

После ввода HTML- и CSS-кода запустите приложение и просмотрите н результат в окне веб-браузера. Выполняющееся приложение должно выглядеть так, как показано на рис. 2.9.

**Рис. 2.9.** Окно создаваемого приложения для чата

Приложение пока не функционирует в полном объеме. На данном этапе используются статические файлы и создана базовая визуальная компоновка. В следующем разделе мы продвинемся немного дальше и определим, каким образом управляются сообщения чата на сервере.

## 2.4. Обработка сообщений чата с помощью Socket.IO

Ранее были сформулированы три вещи, которые должно уметь делать разрабатываемое в этой главе приложение. На данный момент оно умеет делать первую — обслуживать статические файлы. В этом разделе показано, как добиться второй вещи — научить приложение поддерживать обмен сообщениями между браузером и сервером. В современных браузерах такой обмен выполняется с помощью WebSocket. (Просмотрите страницу http://socket.io/#browser-support, чтобы узнать о поддерживаемых библиотекой Socket.IO браузерах.)

С помощью библиотеки Socket.IO обеспечивается уровень абстракции между WebSocket и другими механизмами передачи данных для Node и JavaScript с клиентской стороны. Благодаря Socket.IO реализуется «прозрачный» переход к альтернативам WebSocket, если WebSocket не поддерживается текущим веб-

браузером. При этом используется та же API-библиотека. В данном разделе рассматриваются следующие темы:

- знакомство с Socket.IO и определение Socket.IO-функциональности, требуемой для сервера;
- добавление кода для Socket.IO-сервера;
- добавление кода обработки различных событий приложения.

В библиотеке Socket.IO поддерживаются виртуальные *каналы*, поэтому вместо передачи сообщений каждому подключившемуся пользователю можно передавать сообщения только тем пользователям, которые подписались на определенный канал. В результате, как вы вскоре увидите, реализация комнат чата в приложении становится довольно простой.

**ГЕНЕРАТОРЫ СОБЫТИЙ**

Генератор событий связан с концептуальным ресурсом некоторого рода и может передавать сообщения этому ресурсу и принимать сообщения от него. В качестве ресурса может выступать соединение с удаленным сервером или нечто более абстрактное, например игровой персонаж. В проекте Johnny-Five (https://github.com/rwldrn/johnny-five), в котором Node используется в приложениях для роботов, генераторы событий служат для управления микроконтроллерами Arduino.

Библиотека Socket.IO — это хороший пример полезности генераторов событий. Вообще говоря, *генераторы событий* — это удобный шаблон проектирования, позволяющий организовать асинхронную логику. В этой главе рассматриваются лишь отдельные примеры кода генераторов событий, а более подробное знакомство с ними вас ожидает в следующей главе.

Начнем мы с функциональности сервера и зададим логику соединения, а затем определим функциональность, которую нужно обеспечить на стороне сервера.

### 2.4.1. Настройка сервера Socket.IO

Начнем с добавления двух следующих строк в файл server.js. Первая строка выполняет загрузку функциональности из создаваемого Node-модуля,

поддерживающего возможности чата на базе серверной библиотеки Socket.IO. Этот модуль будет определен позднее. Вторая строка кода запускает Socket.IO-сервер, предоставляя ему уже определенный HTTP-сервер, чтобы использовать один и тот же TCP/IP-порт:

```
var chatServer = require('./lib/chat_server');
chatServer.listen(server);
```

Теперь нужно создать новый файл chat_server.js, который будет находиться в папке lib. Начните создавать этот файл с добавления следующих объявлений переменных. С помощью этих объявлений обеспечивается использование библиотеки Socket.IO, а также инициализируются переменные, задающие состояние чата:

```
var socketio = require('socket.io');
var io;
var guestNumber = 1;
var nickNames = {};
var namesUsed = [];
var currentRoom = {};
```

### *Создание логики установки соединения*

А теперь добавим код из листинга 2.7, определяющий функцию прослушивания сервера чата. Эта функция вызывается в файле server.js. Она запускает Socket.IO-сервер, ограничивает многословие при записи на консоль Socket.IO-данных и задает способ обработки каждого входящего соединения.

Код, выполняющий обработку соединений, вызывает несколько вспомогательных функций, которые определяются в файле chat_server.js.

**Листинг 2.7.** Запуск SocketIO-сервера

```
exports.listen = function(server) {
  // Запуск Socket.IO-сервера, чтобы выполняться вместе
  // с существующим HTTP-сервером
  io = socketio.listen(server);
  io.set('log level', 1);
  // Определение способа обработки каждого пользовательского соединени
  io.sockets.on('connection', function (socket) {
    guestNumber = assignGuestName(socket, guestNumber,
    // Присваивание подключившемуся пользователю имени guest
```

```
    nickNames, namesUsed);
  // Помещение подключившегося пользователя в комнату Lobby
  joinRoom(socket, 'Lobby');
  // Обработка пользовательских сообщений, попыток изменения имени
  // и попыток создания/изменения комнат
  handleMessageBroadcasting(socket, nickNames);
  handleNameChangeAttempts(socket, nickNames, namesUsed);
  handleRoomJoining(socket);
  // Вывод списка занятых комнат по запросу пользователя
  socket.on('rooms', function() {
    socket.emit('rooms', io.sockets.manager.rooms);
  });
  // Определение логики очистки, выполняемой после выхода
  // пользователя из чата
  handleClientDisconnection(socket, nickNames, namesUsed);
 });
};
```

После создания кода, используемого для установки соединения, нужно добавить отдельные вспомогательные функции, реализующие различные потребности приложения.

### 2.4.2. Обработка сценариев и событий приложения

В приложении для чата обрабатываются следующие типы сценариев и событий:

- присваивание гостевого имени;
- запросы на выбор другой комнаты;
- запросы на изменение имени;
- отправка сообщений чата;
- создание комнаты;
- выход пользователя из чата.

Чтобы выполнять обработку перечисленных сценариев и событий, нужно

добавить несколько вспомогательных функций.

### *Присваивание гостевого имени*

Первая вспомогательная функция называется assignGuestName и служит для присваивания имен новым пользователям. Пользователь, который первый раз подключается к серверу чата, помещается в комнату Lobby. При этом вызывается функция assignGuestName, которая присваивает пользователю имя, отличающееся от имен других пользователей.

Каждое гостевое имя состоит из слова Guest (гость), за которым следует число, увеличивающееся на единицу после подключения каждого нового пользователя. Гостевое имя хранится в переменной nickNames и служит в качестве ссылки, связанной с внутренним идентификатором сокета (Socket ID). Гостевое имя также хранится в переменной namesUsed как использованное имя. Чтобы реализовать только что описанные возможности, в файл lib/chat_server.js добавьте код из листинга 2.8.

**Листинг 2.8.** Присваивание гостевого имени

```
function assignGuestName(socket, guestNumber, nickNames, namesUsed) {
  // Создание нового гостевого имени
  var name = 'Guest' + guestNumber;
  // Связывание гостевого имени с идентификатором клиентского подключе
  nickNames[socket.id] = name;
  // Сообщение пользователю его гостевого имени
  socket.emit('nameResult', {
success: true,
    name: name
  });
  // Обратите внимание, что гостевое имя уже используется
  namesUsed.push(name);
  // Для генерирования гостевых имен применяется счетчик с приращением
  return guestNumber + 1;
}
```

### *Присоединение к существующей комнате чата*

Вторая вспомогательная функция, добавляемая в файл chat_server.js, — это функция joinRoom. В ней реализуется логика присоединения пользователя к

комнате чата (листинг 2.9).

**Листинг 2.9.** Код входа пользователя в комнату чата

```
function joinRoom(socket, room) {
  // Вход пользователя в комнату чата
  socket.join(room);
  // Обнаружение пользователя в данной комнате
  currentRoom[socket.id] = room;
  // Оповещение пользователя о том, что он находится в новой комнате
  socket.emit('joinResult', {room: room});
  // Оповещение других пользователей о появлении нового
  // пользователя в комнате чата
  socket.broadcast.to(room).emit('message', {
    text: nickNames[socket.id] + ' has joined ' + room + '.'
  });

// Идентификация других пользователей, находящихся в той же
// комнате, что и пользователь
var usersInRoom = io.sockets.clients(room);
// Если другие пользователи присутствуют в данной
// комнате чата, просуммировать их
if (usersInRoom.length > 1) {
  var usersInRoomSummary = 'Users currently in ' + room + ': ';
  for (var index in usersInRoom) {
    var userSocketId = usersInRoom[index].id;
    if (userSocketId != socket.id) {
      if (index > 0) {
        usersInRoomSummary += ', ';
      }
        usersInRoomSummary += nickNames[userSocketId];
      }
    }
    usersInRoomSummary += '.';
    // Вывод отчета о других пользователях, находящихся в комнате
    socket.emit('message', {text: usersInRoomSummary});
```

```
    }
  }
```

Чтобы войти в комнату чата с помощью библиотеки Socket.IO, нужно вызвать метод join объекта socket. После этого приложение сообщает соответствующую информацию пользователю и другим пользователям, находящимся в той же комнате. Пользователю, присоединившемуся к комнате чата, сообщается о других пользователях, находящихся в этой же комнате, а другие пользователи узнают о новом посетителе комнаты.

### *Обработка запросов об изменении имени*

Если каждому пользователю чата присвоить гостевое имя, будет довольно трудно разобраться в том, кто есть кто. Поэтому в приложении чата можно изменять имя, присвоенное по умолчанию. Как показано на рис. 2.10, в процессе изменения имени веб-браузер пользователя выполняет запрос с помощью библиотеки Socket.IO, а затем принимает ответ, свидетельствующий об успехе или неудаче запроса.

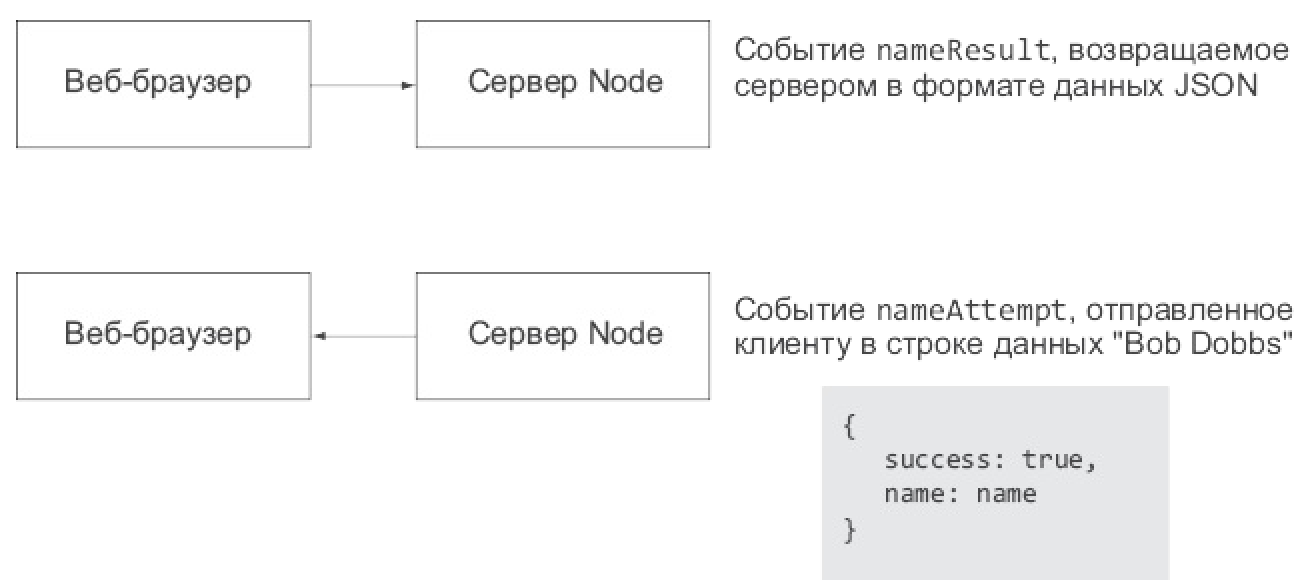



**Рис. 2.10.** Запрос об изменении имени и ответ

Чтобы определить функцию, которая обрабатывает запросы пользователей на изменение имени, добавьте код из листинга 2.10 в файл lib/chat_server.js. В приложении для чата пользователю не разрешается выбирать имена, начинающиеся с префикса Guest, или задействовать уже выбранные имена.

**Листинг 2.10.** Код запроса об изменении имени

```
function handleNameChangeAttempts(socket, nickNames, namesUsed) {
  // Добавление слушателя событий nameAttempt
  socket.on('nameAttempt', function(name) {
```

```
// Не допускаются имена, начинающиеся с Guest
if (name.indexOf('Guest') == 0) {
  socket.emit('nameResult', {
    success: false,
    message: 'Names cannot begin with "Guest".'
  });
} else {
  // Если имя не используется, выберите его
  if (namesUsed.indexOf(name) == -1) {
    var previousName = nickNames[socket.id];
    var previousNameIndex = namesUsed.indexOf(previousName);
    namesUsed.push(name);
    nickNames[socket.id] = name;
    // Удаление ранее выбранного имени, которое
    // освобождается для других клиентов
    delete namesUsed[previousNameIndex];
    {
      success: true,
      name: name
    }
    socket.emit('nameResult', {
      success: true,
      name: name
    });
    socket.broadcast.to(currentRoom[socket.id]).emit('message', {
      text: previousName + ' is now known as ' + name + '.'
    });
  } else {
    socket.emit('nameResult', {
     // Если имя зарегистрировано, отправка клиенту
     // сообщения об ошибке
     success: false,
     message: 'That name is already in use.'
    });
```

```
      }
    }
  });
}
```

### *Передача сообщений чата*

Теперь, когда присваивание имен (ников) пользователям чата реализовано, нужно добавить функцию, которая выполняет обработку рассылаемых сообщений. Соответствующий базовый процесс иллюстрирует рис. 2.11. Пользователь генерирует событие путем выбора комнаты чата, в которую будет передано сообщение, и ввода текста сообщения. Сервер пересылает сообщение остальным пользователям, находящимся в этой комнате.

Добавьте следующий код в файл lib/chat_server.js. Для рассылки сообщения используется функция broadcast из библиотеки Socket.IO:

```
function handleMessageBroadcasting(socket) {
  socket.on('message', function (message) {
    socket.broadcast.to(message.room).emit('message', {
      text: nickNames[socket.id] + ': ' + message.text
    });
  });
}
```



**Рис. 2.11.** Передача сообщения в чате

### *Создание комнат чата*

На этом этапе нужно добавить в приложение функциональность, которая даст пользователю возможность входить в существующую комнату, а при отсутствии нужной комнаты — создавать ее. На рис. 2.12 показан процесс обмена сообщениями между пользователем и сервером.

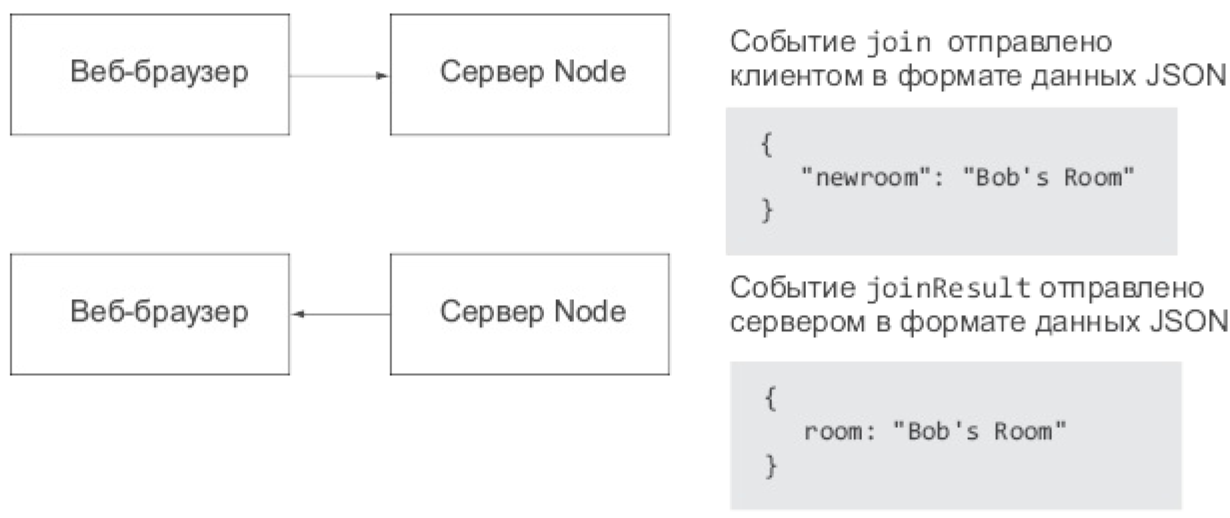



**Рис. 2.12.** Переход в другую комнату чата

Чтобы получить возможность выбора (или создания) другой комнаты чата, в файл lib/chat_server.js добавьте следующий код. Обратите внимание на использование метода leave библиотеки Socket.IO:

```
function handleRoomJoining(socket) {
  socket.on('join', function(room) {
    socket.leave(currentRoom[socket.id]);
    joinRoom(socket, room.newRoom);
  });
}
```

### *Обработка выхода пользователя из чата*

Теперь осталось добавить следующий код в файл lib/chat_server.js, чтобы удалить имя пользователя из переменных nickNames и namesUsed после его выхода из чата:

```
function handleClientDisconnection(socket) {
  socket.on('disconnect', function() {
    var nameIndex = namesUsed.indexOf(nickNames[socket.id]);
    delete namesUsed[nameIndex];
    delete nickNames[socket.id];
```

```
  });
}
```

Теперь, после завершения создания серверных компонентов, осталось добавить клиентский код.

## 2.5. Применение JavaScript для разработки на стороне клиента пользовательского интерфейса приложения

Завершив создание серверного кода, основанного на использовании библиотеки Socket.IO и выполняющего управление сообщениями, переданными из браузера, нужно добавить на сторону клиента JavaScript-код, предназначенный для обмена данными с сервером. С помощью этого кода реализуется следующая функциональность:

- отправка серверу пользовательских сообщений и запросов об изменении имени пользователя/комнаты чата;

- вывод сообщений других пользователей и списка доступных комнат.

Начнем с реализации первой части функциональности.

### 2.5.1. Ретрансляция серверу сообщений и запросов об изменении имени пользователя/комнаты

Первый фрагмент создаваемого на стороне клиента JavaScript-кода представляет собой JavaScript-прототип, который обрабатывает команды чата, отсылает сообщения и запросы относительно изменения комнаты и имени пользователя чата.

В папке public/javascripts создайте файл chat.js и поместите в него следующий код. Этот код представляет собой JavaScript-эквивалент «класса», который после создания экземпляра принимает единственный аргумент — Socket.IO-сокет:

```
var Chat = function(socket) {
  this.socket = socket;
};
```

Затем добавьте следующую функцию, которая будет отсылать сообщения чата:

```
Chat.prototype.sendMessage = function(room, text) {
  var message = {
    room: room,
```

```
    text: text
  };
  this.socket.emit('message', message);
 };
```

Следующая функция изменяет комнаты чата:

```
Chat.prototype.changeRoom = function(room) {
  this.socket.emit('join', {
    newRoom: room
  });
};
```

И наконец, осталось добавить функцию, предназначенную для обработки команд чата (листинг 2.11). Эта функция распознает следующие команды чата:

- join — присоединение к комнате или создание комнаты;
- nick — изменение имени пользователя (ника) чата.

**Листинг 2.11.** Обработка команд чата

```
Chat.prototype.processCommand = function(command) {
  var words = command.split(' ');
  var command = words[0]
        // Команда синтаксического разбора, начиная с первого слова
          .substring(1, words[0].length)
          .toLowerCase();
  var message = false;

  switch(command) {
    case 'join':
      words.shift();
      var room = words.join(' ');
      // Обработка изменения/создания комнаты чата
      this.changeRoom(room);
      break;

    case 'nick':
```

```
      words.shift();
      var name = words.join(' ');
      // Обработка попыток изменения имени пользователя чата
      this.socket.emit('nameAttempt', name);
      break;

    default:
      // Возврат сообщения об ошибке, если команда не распознается
      message = 'Unrecognized command.';
      break;
    }
  return message;
};
```

### 2.5.2. Вывод в пользовательском интерфейсе сообщений и доступных комнат

А теперь приступим к разработке кода, реализующего непосредственное взаимодействие с помощью jQuery с пользовательским интерфейсом на базе браузера. И начнем с реализации функции вывода текстовых данных.

В веб-приложениях используется два типа текстовых данных, которые отличаются по уровню безопасности. Во-первых, это *надежные* текстовые данные, которые состоят из текста, предоставляемого приложением, во-вторых, это *ненадежные* текстовые данные, представляющие собой текст, создаваемый пользователями приложения. Текстовые данные, созданные пользователями, считаются опасными, поскольку злоумышленники могут намеренно между тегами <script> вводить в них JavaScript-код. Просмотр подобного текста другими пользователями может привести к неприятным последствиям, например к переходу на другую веб-страницу. Подобный метод взлома веб-приложения называется атакой межсайтового скриптинга (Cross-Site Scripting, XSS).

В приложении для чата используются две вспомогательные функции вывода текстовых данных. Первая функция должна выводить на экран ненадежные текстовые данные, вторая — надежные.

Функция divEscapedContentElement служит для вывода ненадежного текста. Эта функция очищает текст, преобразуя специальные символы в HTML-сущности (рис. 2.13). В результате браузер показывает эти сущности так, как они выглядят, а не пытается интерпретировать их как части HTML-тега.

```
<script>alert('XSS attack!');</script>
```

Сообщение очищается с помощью функции divEscapedContentElement и помещается между тегами <div>

```
<div>&It;script&gt;alert('XSS attack!');&Lt:/script&gt;<div>
```

**Рис. 2.13.** Преобразование специальных символов ненадежного текста в HTML-сущности

Функция divSystemContentElement призвана отображать надежный контент, создаваемый системой, а не другими пользователями.

В папку public/javascripts скопируйте файл chat_ui.js и поместите в этот файл код следующих двух вспомогательных функций:

```
function divEscapedContentElement(message) {
  return $('<div></div>').text(message);
}
function divSystemContentElement(message) {
  return $('<div></div>').html('<i>' + message + '</i>');
}
```

Очередная функция, код которой нужно добавить в файл chat_ui.js, обрабатывает вводимые пользователем данные (листинг 2.12). Если вводимые пользователем данные начинаются символом слеша (/), они трактуются как команды чата. Если же символ слеша отсутствует, текст отсылается серверу в виде сообщения чата, транслируемого другим пользователям, а также выводится в комнате чата, в которой находится пользователь.

**Листинг 2.12.** Обработка исходных данных, вводимых пользователем

```
function processUserInput(chatApp, socket) {
  var message = $('#send-message').val();
  var systemMessage;
  // Начинающиеся со слеша данные, вводимые пользователем,
  // трактуются как команды
  if (message.charAt(0) == '/') {
    systemMessage = chatApp.processCommand(message);
    if (systemMessage) {
      $('#messages').append(divSystemContentElement(systemMessage));
```

```
    }
  } else {
    // Трансляция вводимых пользователем данных другим пользователям
    chatApp.sendMessage($('#room').text(), message);
    $('#messages').append(divEscapedContentElement(message));
    $('#messages').scrollTop($('#messages').prop('scrollHeight'));
  }
  $('#send-message').val('');
}
```

Теперь, когда определены требуемые вспомогательные функции, нужно добавить код из листинга 2.13. Этот код, который вызывается после полной загрузки веб-страницы пользовательским браузером, выполняет инициализацию на стороне клиента системы обработки событий библиотеки Socket.IO.

**Листинг 2.13.** Логика инициализации приложения на стороне клиента

```
var socket = io.connect();

$(document).ready(function() {
  var chatApp = new Chat(socket);

// Вывод результатов попытки изменения имени
socket.on('nameResult', function(result) {
  var message;

  if (result.success) {
    message = 'You are now known as ' + result.name + '.';
  } else {
    message = result.message;
  }
  $('#messages').append(divSystemContentElement(message));
});

// Вывод результатов изменения комнаты
socket.on('joinResult', function(result) {
  $('#room').text(result.room);
  $('#messages').append(divSystemContentElement('Room changed.'));
```

```
});

// Вывод полученных сообщений
socket.on('message', function (message) {
  var newElement = $('<div></div>').text(message.text);
  $('#messages').append(newElement);
});

// Вывод списка доступных комнат
socket.on('rooms', function(rooms) {
  $('#room-list').empty();
  for(var room in rooms) {
    room = room.substring(1, room.length);
    if (room != '') {
      $('#room-list').append(divEscapedContentElement(room));
    }
  }

  // Разрешено щелкнуть на имени комнаты, чтобы изменить ее
  $('#room-list div').click(function() {
    chatApp.processCommand('/join ' + $(this).text());
    $('#send-message').focus();
  });
});

  // Запрос списка поочередно доступных комнат чата
  setInterval(function() {
    socket.emit('rooms');
  }, 1000);
  $('#send-message').focus();
  // Отправка сообщений чата с помощью формы
  $('#send-form').submit(function() {
    processUserInput(chatApp, socket);
    return false;
```

```
  });
});
```

Чтобы завершить разработку приложения, осталось добавить из листинга 2.14 в файл public/stylesheets/style.css финальный CSS-код стилизации.

**Листинг 2.14.** Финальный код, добавляемый в файл style.css

```
#room-list {
  float: right;
  width: 100px;
  height: 300px;
  overflow: auto;
}

#room-list div {
  border-bottom: 1px solid #eee;
}

#room-list div:hover {
  background-color: #ddd;
}

#send-message {
  width: 700px;
  margin-bottom: 1em;
  margin-right: 1em;
}

#help {
  font: 10px "Lucida Grande", Helvetica, Arial, sans-serif;
}
```

После добавления этого кода попробуйте выполнить приложение (с помощью команды node server.js). Результат показан на рис. 2.14.

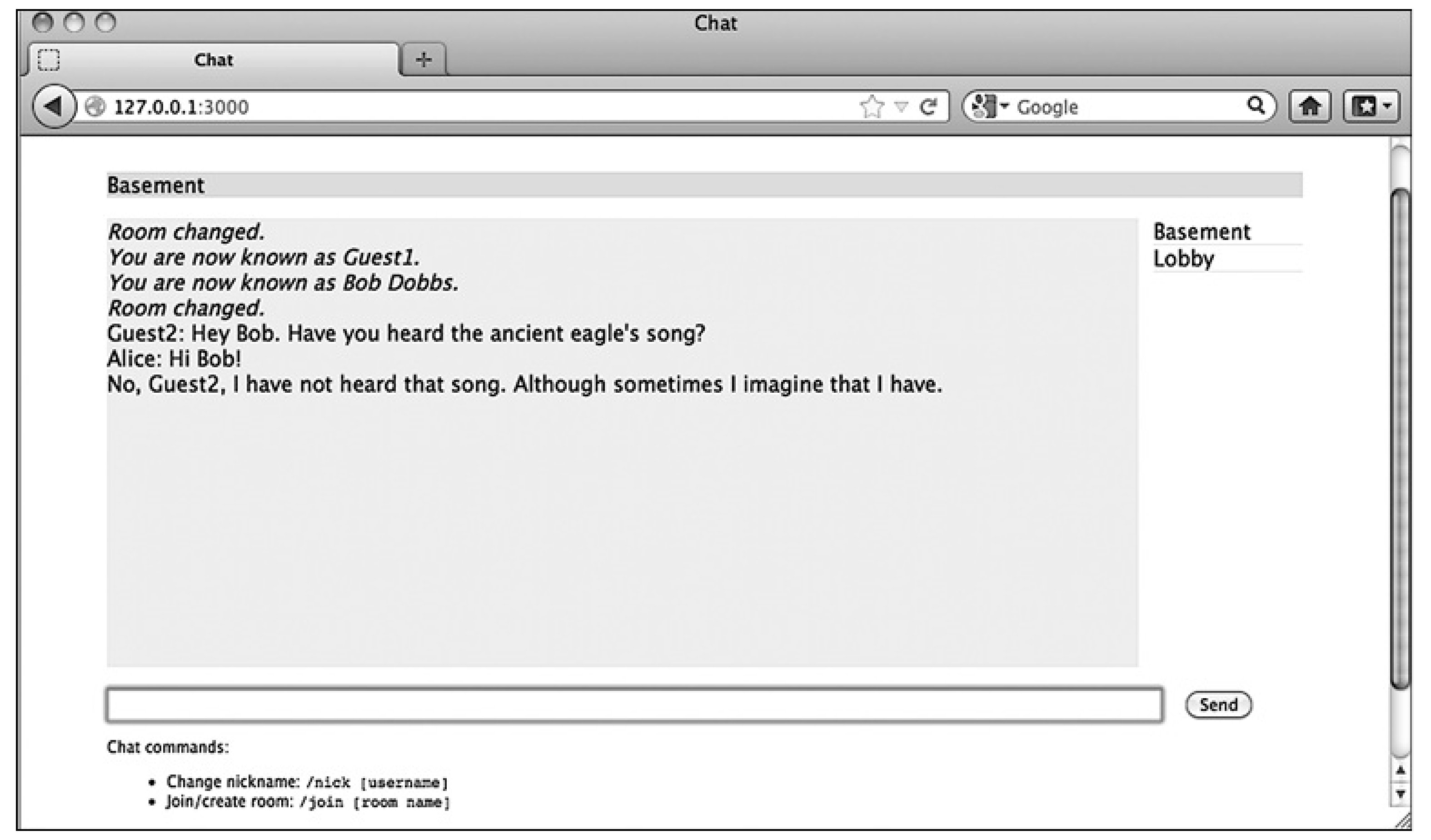



**Рис. 2.14.** Результат выполнения готового приложения для чата

### 2.6. Резюме

Итак, мы только что завершили разработку небольшого веб-приложения реального времени с помощью Node.js!

В этой главе вы должны были получить представление о способах создания подобных приложений и о том, как выглядит их код. Если некоторые аспекты разработки, затронутые в этой главе, остались непонятными, не волнуйтесь. В следующих главах методики и технологии, использованные при разработке этого веб-приложения, изучаются более подробно.

Но прежде чем вплотную заняться разработкой на платформе Node, следует научиться справляться с проблемами, возникающими при создании асинхронных приложений. В следующей главе описываются важные методики и трюки, которые позволят вам сэкономить немало времени и сил.

## Глава 3. Основы программирования в Node

- Размещение кода в модулях

- Соглашения по созданию кода

- Обработка одиночных событий посредством обратных вызовов
- Обработка повторяющихся событий с помощью генераторов событий
- Реализация последовательного и параллельного выполнения операций
- Инструменты управления порядком выполнения операций

Платформа Node, в отличие от других платформ с открытым исходным кодом, проста в установке, не требует большого объема памяти и не занимает много места на жестком диске. Для нее не нужны сложные интегрированные среды разработки или системы компоновки. Тем не менее кое-какие базовые знания серьезно помогут вам на начальном этапе. В этой главе рассматриваются две проблемы, с которыми сталкиваются начинающие Node-разработчики:

- как организовать программный код;
- как писать асинхронные программы.

Проблема организации кода знакома большинству опытных программистов. Концептуально код организуется в виде классов и функций. Файлы, содержащие классы и функции, хранятся в древовидной структуре папок файловой системы. В конечном счете код размещается в приложениях и библиотеках. В Node используется система модулей, представляющая собой мощный механизм организации кода, который подробно рассмотрен в этой главе.

Асинхронное программирование требует времени на освоение. От программиста, в частности, требуется изменить парадигму мышления, чтобы понять принципы выполнения программного кода. В рамках синхронного программирования каждая строка кода выполняется после того, как выполнены все предыдущие строки. В асинхронном программировании ситуация совершенно иная, и создаваемый вами код на первых порах будет напоминать вам машину Руба Гольдберга (Rube Goldberg), которая простые операции выполняет чрезвычайно сложным образом. Поэтому перед разработкой большого проекта стоит посвятить время изучению методик, позволяющих аккуратно контролировать поведение приложения.

В этой главе вам предстоит освоить множество важных приемов асинхронного программирования, позволяющих держать выполнение приложения под контролем. Вы узнаете:

- как реагировать на одиночные события;

- как обрабатывать повторяющиеся события;
- как контролировать асинхронную логику.

Знакомство с основами программирования в Node начинается с изучения методик распределения кода по *модулям*, облегчающим организацию и пакетирование кода для его многократного использования.

### 3.1. Организация и многократное использование программного кода в Node

В процессе разработки приложения (на платформе Node или на какой-либо другой платформе) рано или поздно наступает момент, когда код становится слишком громоздким для его размещения в единственном файле. В этом случае можно воспользоваться традиционным подходом, когда большой файл в соответствиис программной логикой разбивается на несколько меньших по размеру файлов (рис. 3.1).

**Рис. 3.1.** Разбираться в коде проще, если хранить его не в одном большом файле, а в отдельных файлах, распределенных по разным папкам

В некоторых языках программирования, таких как PHP и Ruby, включение в программу другого файла с кодом (так называемого включаемого файла) может привести к тому, что программная логика включаемого файла меняет глобальные области видимости. В этом случае произвольные переменные, созданные во включаемом файле, и функции, объявленные в том же файле, отчасти переопределяют переменные и функции, созданные и объявленные в приложении.

Приложение, созданное в PHP, может содержать следующий код:

```
function uppercase_trim($text) {
  return trim(strtoupper($text));
```

```
}
include('string_handlers.php');
```

Если в файле string_handlers.php предпринимается попытка определения функции uppercase_trim, на экране появится следующее сообщение об ошибке:

```
Fatal error: Cannot redeclare uppercase_trim()
```

В PHP эта проблема решается с помощью *пространств имен*, в Ruby в подобном случае применяются *модули*. В то же время в Node не существует простого способа решения потенциальной проблемы, возникающей вследствие «загрязнения» глобального пространства имен.

**ПРОСТРАНСТВА ИМЕН в PHP, МОДУЛИ в RUBY**

Пространства имен в PHP рассматриваются в руководстве, доступном на веб-сайте по адресу [http://php.net/manual/en/language.namespaces.php](http://php.net/manual/en/language.namespaces.php). Ruby-модули описаны в документации по Ruby, находящейся на веб-сайте по адресу [www.ruby-doc.org/core-1.9.3/Module.html](www.ruby-doc.org/core-1.9.3/Module.html).

Хотя в Node многократно используемый код тоже объединяется в модулях, этот код не меняет глобальной области видимости. Например, предположим, что вы с помощью PHP разрабатываете систему управления контентом (Content Management System, CMS) с открытым кодом. При этом планируется использование API-библиотеки от независимого производителя, в которой пространства имен не предусмотрены. В этой библиотеке может содержаться класс с тем же самым именем, что и класс в вашем приложении. Эта ситуация будет приводить к неправильной работе приложения, пока вы не измените имя класса в приложении или в библиотеке. Однако изменение имени класса в приложении может привести к проблемам у других разработчиков, которые на основе вашей системы управления контентом разрабатывают собственные проекты. Если же вы решите поменять имя класса в библиотеке, вам придется повторять эту операцию при каждом обновлении библиотеки в дереве исходного кода приложения. Чтобы решить проблему, вызванную коллизией имен, требуется комплексный подход.

В отличие от PHP при использовании Node-модулей во включаемом файле можно выбрать те функции и переменные, которые должны экспонироваться для приложения. Если модуль возвращает несколько функций или переменных, присвойте соответствующие значения свойствам объекта exports. Если же модуль возвращает единственную функцию или переменную, достаточно присвоить нужное значение свойству объекта module.exports. Иллюстрация приведена на рис.

3.2.

**Рис. 3.2.** Путем присваивания значений свойству объекта module.exports или свойствам объекта exports можно выбрать в модуле переменные и функции, которые должны использоваться в приложении

Если вы не все поняли, не волнуйтесь. Излагаемые в этом разделе концепции далее иллюстрируются на других примерах.

Благодаря устранению «загрязнения» глобальной области видимости система модулей в Node исключает конфликт между именами и облегчает многократное использование кода. Модули публикуются в хранилище диспетчера Node-пакетов (Node Package Manager, npm), которое представляет собой коллекцию готовых к применению Node-модулей. Эти модули могут использоваться сообществом разработчиков Node-приложений без риска переопределения переменных и функций одного модуля другим модулем. Дополнительные сведения о публикации модулей в npm-хранилище приведены в главе 14.

Чтобы успешно распределять программную логику по отдельным модулям, нужно знать:

- как создавать модули;
- в каком месте файловой системы хранятся модули;
- что принимать во внимание при создании и использовании модулей.

Итак, приступим к изучению системы модулей в Node, создав первый простой модуль.

### 3.1.1. Создание модулей

Модуль представляет собой либо единственный файл, либо одну папку, в которой находится один или несколько файлов (рис. 3.3). Если модуль представляет собой папку, в ней находится файл index.js, который обычно анализируется в первую очередь. (Этот режим работы можно изменить, как описано в разделе 3.1.4.)

**Рис. 3.3.** В Node модули могут создаваться либо как файлы (пример 1), либо как папки (пример 2)

Для получения типичного модуля следует создать файл, в котором определить свойства объекта exports в качестве данных произвольного типа, таких как строки, объекты и функции.

Чтобы продемонстрировать, как создать обычный модуль, добавим функции конвертации валют в файл currency.js. Код, находящийся в этом файле, приведен в листинге 3.1. Этот код содержит две функции, выполняющие конвертацию канадских долларов в доллары США и обратно.

**Листинг 3.1.** Определение модуля в Node

```
var canadianDollar = 0.91;
function roundTwoDecimals(amount) {
  return Math.round(amount * 100) / 100;
}
// Функция canadianToUS задана в модуле exports, поэтому она
// может использоваться в коде, в котором этот модуль объявлен
// как загружаемый по требованию
exports.canadianToUS = function(canadian) {
  return roundTwoDecimals(canadian * canadianDollar);
}
// Функция USToCanadian также определена в модуле exports
exports.USToCanadian = function(us) {
  return roundTwoDecimals(us / canadianDollar);
}
```

Обратите внимание, что установлены значения только двух свойств объекта exports. Это означает, что только две функции, canadianToUS и USToCanadian, могут быть доступны в приложении, использующем этот модуль. Переменная canadianDollar является закрытой, то есть хотя она влияет на программную логику функций canadianToUS и USToCanadian, приложение не может иметь к ней

непосредственного доступа.

Чтобы задействовать новый модуль, воспользуйтесь Node-функцией require, аргументом которой является путь к включаемому модулю. Чтобы найти модуль и загрузить содержимое файла, Node выполняет синхронный поиск.

### ФУНКЦИЯ REQUIRE И СИНХРОННЫЙ ВВОД-ВЫВОД

Функция require — эта одна из нескольких функций синхронного ввода-вывода, доступных в Node. Поскольку модули используются часто и обычно помещаются в начале файла, функция require благодаря своей синхронной природе позволяет писать понятый, организованный и читабельный код.

Избегайте применять функцию require в компонентах приложения, осуществляющих интенсивные операции ввода-вывода. При асинхронном вызове блокируется выполнение произвольных операций в Node до тех пор, пока асинхронный вызов не будет завершен. Например, если выполняется HTTP-сервер, то при использовании в каждом исходящем запросе функции require быстродействие этого сервера сильно замедлится. Поэтому функция require и другие синхронные операции обычно применяют только в том случае, если приложение изначально было загружено в память.

В следующем листинге показан код, находящийся в файле test-currency.js. Модуль currency.js объявлен как загружаемый по требованию.

**Листинг 3.2.** Объявление модуля как загружаемого по требованию

```
// В пути используются символы ./, которые указывают на то,
// что модуль находится в той же папке, что и сценарий приложения
var currency = require('./currency');

console.log('50 Canadian dollars equals this amount of US dollars:');
// Использование функции canadianToUS из модуля currency
console.log(currency.canadianToUS(50));

console.log('30 US dollars equals this amount of Canadian dollars:');
// Использование функции USToCanadian из модуля currency
console.log(currency.USToCanadian(30));
```

Если перед именем модуля, загружаемого по требованию, указываются символы ./, это означает, что соответствующий файл модуля (currency.js) должен находиться в папке currency_app, в которую помещен сценарий приложения test-currency.js (рис. 3.4). Предполагается, что файлы модулей, загружаемых по требованию, имеют расширение .js, поэтому расширение можно явно не указывать.

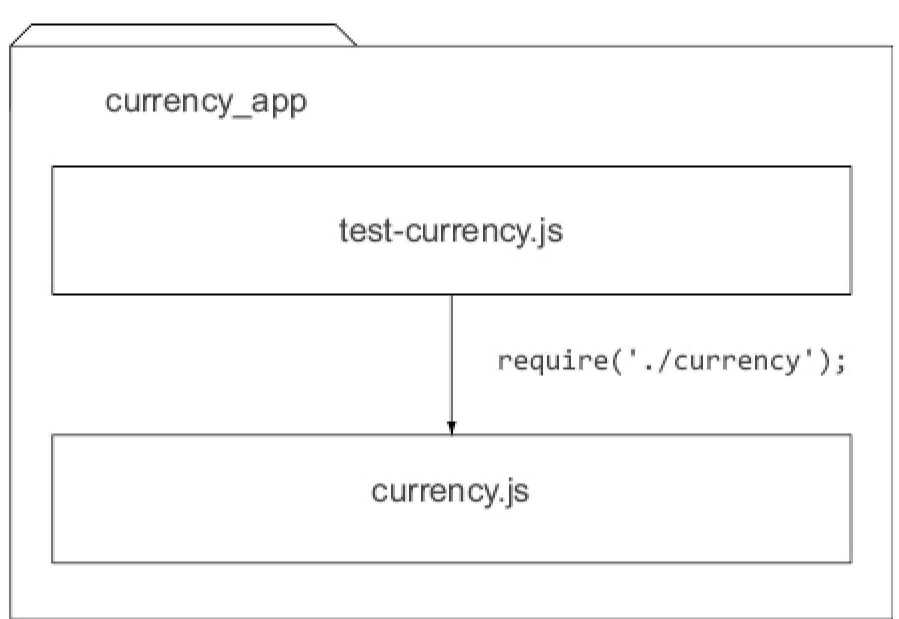



**Рис. 3.4.** Если символы ./ поместить перед именем модуля, загружаемого по требованию, Node будет искать этот модуль в той же папке, в которой находится файл исполняемой программы

Когда Node найдет и проанализирует модуль, функция require сможет вернуть содержимое объекта exports, определенного в модуле. После этого обе возвращаемые модулем функции можно использовать для конвертации валют.

Если нужно поместить модуль во вложенную папку, например в lib, измените строку, в которой находится функция require, следующим образом:

```
var currency = require('./lib/currency');
```

Заполнение переменными и функциями объекта exports, относящегося к модулю, дает вам простой способ распределения многократно используемого кода по разным файлам.

### 3.1.2. Создание узкоспециализированного модуля с помощью объекта module.exports

После заполнения объекта exports переменными и функциями его можно использовать для решения большинства задач, тем не менее остается ряд проблем, которых невозможно избежать в рамках предлагаемой методики.

Например, рассмотренный ранее модуль конвертера валют можно переработать таким образом, чтобы вместо объекта, содержащего функции, возвращать единственную функцию-конструктор Currency. Объектно-ориентированная реализация модуля может выглядеть следующим образом:

```
var Currency = require('./currency');
var canadianDollar = 0.91;

var currency = new Currency(canadianDollar);
console.log(currency.canadianToUS(50));
```

Код получится более элегантным, если возвращать функцию из модуля, а не из объекта. При этом нужно учитывать, что из модуля можно вернуть только одну сущность.

На первый взгляд создается впечатление, что для создания модуля, который возвращает единственную переменную или функцию, нужно присвоить возвращаемую сущность объекту exports. Но на самом деле это не так, поскольку объект exports не может переназначаться любому другому объекту, функции или переменной. В коде модуля, приведенном в листинге 3.3, делается попытка присвоить объекту exports функцию.

**Листинг 3.3.** Этот модуль не работает так, как ожидалось

```
var Currency = function(canadianDollar) {
  this.canadianDollar = canadianDollar;
}

Currency.prototype.roundTwoDecimals = function(amount) {
  return Math.round(amount * 100) / 100;
}

Currency.prototype.canadianToUS = function(canadian) {
  return this.roundTwoDecimals(canadian * this.canadianDollar);
}

Currency.prototype.USToCanadian = function(us) {
  return this.roundTwoDecimals(us / this.canadianDollar);
}
// Неверно: Node не ожидает переназначения объекта exports
exports = Currency;
```

Чтобы добиться корректной работы кода модуля, вместо exports нужно указать module.exports. С помощью объекта module.exports можно экспортировать единственную переменную, функцию или объект. Если создан модуль, в котором присваиваются значения свойствам обоих объектов (exports и module.exports), то

возвращается module.exports, а exports игнорируется.

**ЧТО ЖЕ на самом деле ЭКСПОРТИРУЕТСЯ**

В конечном счете в приложение экспортируется объект module.exports, а exports просто исполняет роль глобальной ссылки на module.exports. В свою очередь, module.exports определяется как пустой объект, которому могут присваиваться свойства. Например, exports.myFunc — это просто сокращенная запись для module.exports.myFunc.

Это означает, что если что-то присвоить объекту exports, связь между module.exports и exports будет разорвана. Поскольку реально экспортируется именно module.exports, объект exports перестает функционировать, как ожидалось, поскольку он больше не связан с объектом module.exports. Если же вы хотите использовать эту связь, ее можно восстановить следующей инструкцией:

```
module.exports = exports = Currency;
```

В зависимости от текущих задач используйте либо exports, либо module.exports, чтобы распределить код по разным модулям и тем самым избежать проблем, порождаемых постоянно растущими размерами сценариев приложения.

### 3.1.3. Многократное использование модулей с помощью папки node_modules

Модули, хранящиеся в файловой системе и загружаемые по требованию какого-либо приложения, могут быть полезны для организации кода этого приложения, но они не помогут, если код должен многократно или совместно использоваться другими приложениями. Node предлагает уникальный механизм многократного использования кода, который позволяет по требованию загружать модули, даже не зная их местоположения в файловой системе. Этот механизм реализован на основе папок node_modules.

В рассмотренном ранее примере модуля использовалась конструкция из require и ./currency. Если в этой конструкции опустить символы ./, Node будет искать модуль в соответствии с правилами, проиллюстрированными на рис. 3.5.

С помощью переменной окружения NODE_PATH определяются альтернативны местоположения для Node-модулей. Переменной NODE_PATH може присваиваться список папок, разделенных точками с запятой (в Windows) или двоеточиями (в другой операционной системе).

### 3.1.4. Возможные проблемы

Хотя система модулей в Node довольно проста, следует обратить внимание на два момента.

Во-первых, если в качестве модуля используется папка, анализируемый файл должен называться index.js, пока не задан другой режим с помощью файла package.json, находящегося в папке модуля. Чтобы вместо файла index.js использовать альтернативный файл, в файл package.json должны быть включены данные в формате JSON ( JavaScript Object Notation — нотация JavaScript-объектов определяющие объект с помощью ключа main. Этот ключ описывает путь к главному файлу в папке модуля. На рис. 3.6 показана блок-схема, иллюстрирующая эти правила.

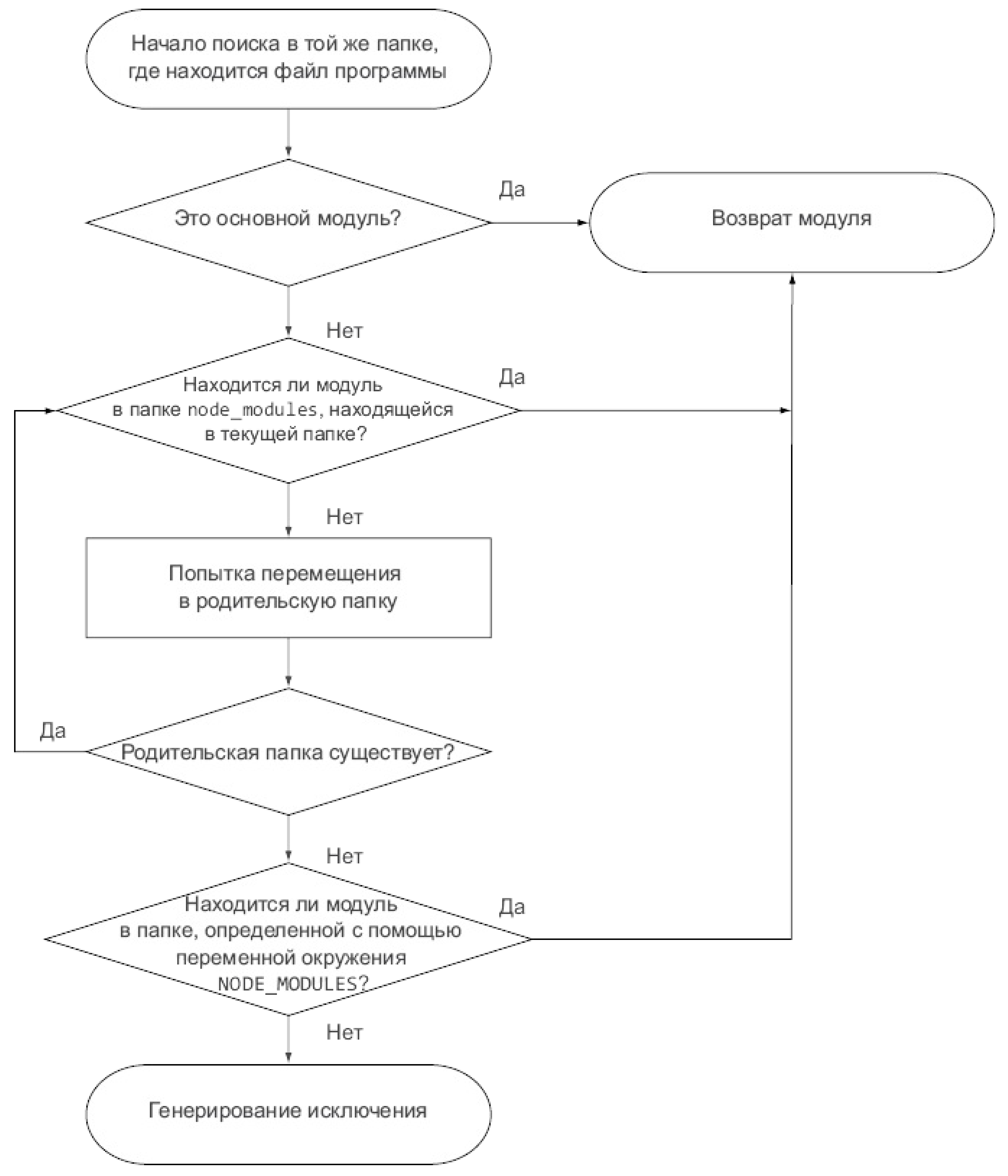



**Рис. 3.5.** Этапы поиска модуля

Вот пример файла package.json, который присваивает файлу currency.js статус главного файла:

```
{
  "main": "./currency.js"
}
```

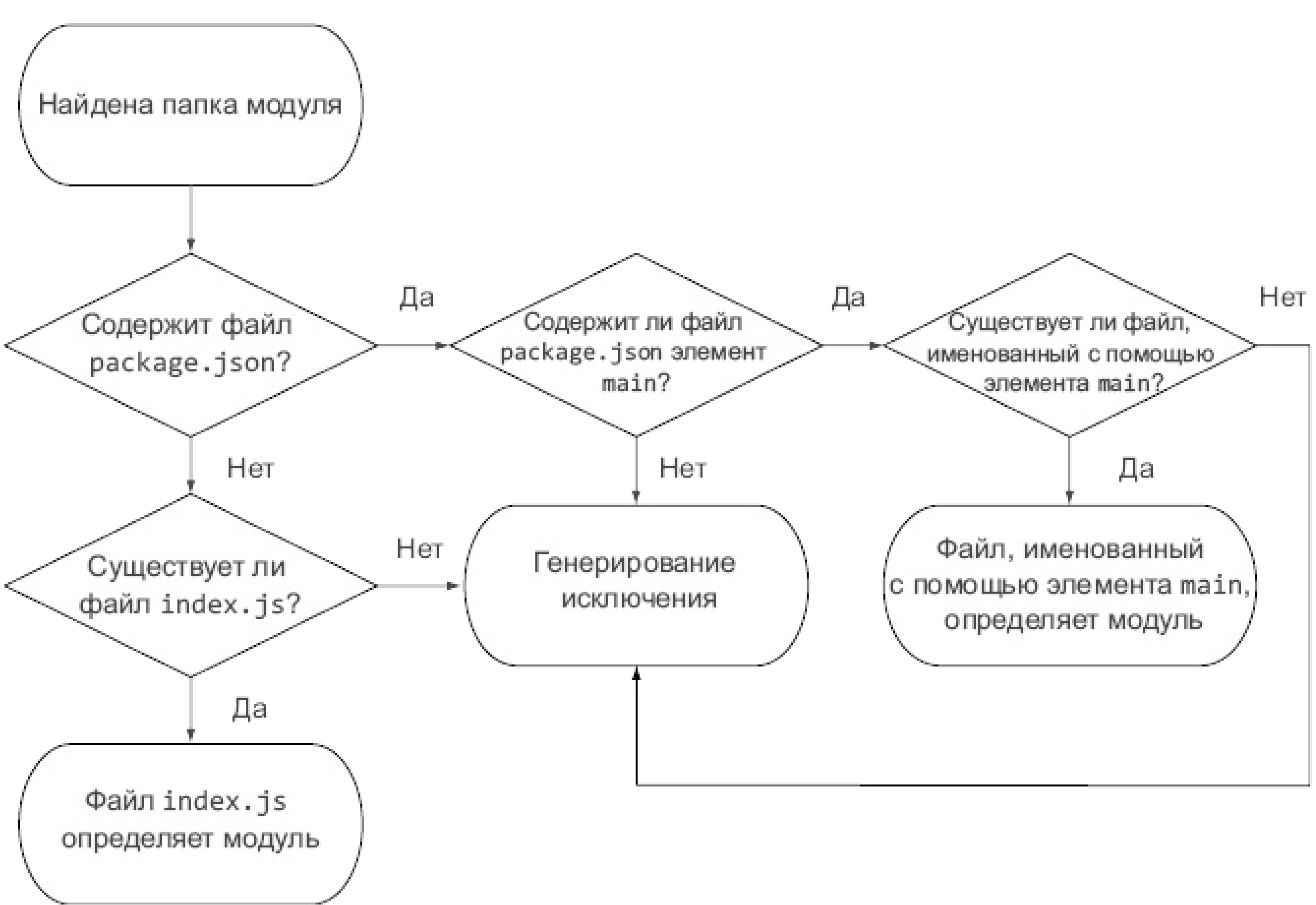



**Рис. 3.6.** Файл package.json, помещенный в папку модуля, позволяет определить модуль без использования файла index.js для

Во-вторых, следует учитывать способность Node кэшировать модули как объекты. Если двум файлам в приложении требуется загрузить один и тот же модуль, первый файл сохранит возвращаемые данные в памяти приложения, в результате второму файлу не нужно будет получать доступ к исходным файлам модуля и заниматься их анализом. Фактически второй файл получает возможность даже менять кэшированные данные. Благодаря такому режиму один модуль может менять поведение другого модуля, освобождая разработчика от необходимости создавать новую версию второго модуля.

Наилучший способ освоить систему модулей в Node — попробовать поработать с ними, применить описанные здесь методики и убедиться в том, что поведение модулей совпадает с ожидаемым.

Теперь, после знакомства с базовыми принципами использования модулей, мы перейдем к изучению методик асинхронного программирования.

## 3.2. Методики асинхронного программирования

Если вы создавали клиентские веб-приложения, в которых события интерфейса (например, щелчки мыши) вызывали выполнение определенного кода, то вы, по сути, занимались асинхронным программированием. Асинхронные серверные веб-

приложения выполняются аналогичным образом — происходящие события вызывают выполнение соответствующего кода. В Node применяются две модели управления логикой событий — обратные вызовы и слушатели событий.

В общем случае с помощью *обратных вызовов* (callbacks) реализуется логика одиночных откликов. Например, при создании запроса к базе данных можно использовать обратный вызов для обработки результатов выполнения запроса. Этот обратный вызов может вывести на экран результаты выполнения запроса, провести на их основе вычисления или выполнить другой обратный вызов, используя их в качестве аргумента.

*Слушатели событий* (event listeners) фактически являются обратными вызовами, связанными с некой абстрактной сущностью, в качестве которой в данном случае выступает *событие* (event). К примеру, щелчок мыши — это событие, которое вы могли бы обрабатывать в браузере. Или, например, когда в Node выполняется HTTP-запрос, HTTP-сервер генерирует событие request. Можн прослушивать это событие и написать код, реализующий отклик на него. В следующем примере кода в ответ на сгенерированное событие запроса (request) вызывается функция handleRequest:

```
server.on('request', handleRequest)
```

Экземпляр HTTP-сервера в Node представляет собой пример *генератора событий* (event emitter). Генератор событий — это класс EventEmitter, который может наследоваться и применяться для инициирования и обработки событий. В Node значительная часть базовой функциональности наследуется от класса EventEmitter. Вы также можете разрабатывать собственную функциональность.

Теперь, когда вы знаете, что логика событий в Node реализована в двух формах, давайте посмотрим, как она работает, выяснив:

- как обрабатывать одиночные события посредством обратных вызовов;
- как реагировать на повторяющиеся события с помощью слушателей событий;
- какие проблемы присущи асинхронному программированию.

Сначала мы рассмотрим один из наиболее распространенных механизмов обработки асинхронного кода — обратные вызовы.

### 3.2.1. Обработка одиночных событий посредством обратных вызовов

При *обратном вызове* функция обратного вызова передается в качестве аргумента асинхронной функции, которая определяет, что должно произойти после

завершения асинхронной операции. Разработчики Node-приложений используют обратные вызовы чаще, чем генераторы событий, к тому же обратные вызовы проще в применении.

Чтобы продемонстрировать, как использовать обратные вызовы при разработке приложений, создадим простой HTTP-сервер, выполняющий следующие действия:

- извлечение в асинхронном режиме заголовков последних постов, хранящихся в JSON-файле;
- извлечение в асинхронном режиме HTML-шаблона;
- формирование HTML-страницы, содержащей заголовки;
- передача HTML-страницы пользователю.

Примерный результат работы этого сервера представлен на рис. 3.7.

**Рис. 3.7.** HTML-ответ, полученный от веб-сервера, который извлекает заголовки сообщений из JSON-файла и возвращает результаты в виде веб-страницы

JSON-файл (titles.json), содержимое которого показано в листинге 3.4, отформатирован в виде массива строк, состоящего из заголовков постов.

**Листинг 3.4.** Список заголовков постов

```
[
  "Kazakhstan is a huge country... what goes on there?",
  "This weather is making me craaazy",
  "My neighbor sort of howls at night"
]
```

HTML-файл шаблона (template.html) содержит некую базовую конструкцию, в которую вставляются заголовки постов блога (листинг 3.5).

**Листинг 3.5.** Базовый HTML-шаблон для визуализации заголовков

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

```
  <body>
    <h1>Latest Posts</h1>
    // Символ % заменяется данными заголовка
    <ul><li>%</li></ul>
  </body>
</html>
```

В листинге 3.6 представлен код, который извлекает содержимое JSON-файла и визуализирует веб-страницу (blog_recent.js). Функции обратного вызова выделены полужирным шрифтом.

**Листинг 3.6.** Пример использования обратных вызовов в простом приложении

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

// Создание HTTP-сервера и определение кода отклика
// с помощью функции обратного вызова
http.createServer(function(req, res) {
  if (req.url == '/') {
    // Считывание JSON-файла и определение способа обработки
    // его содержимого с помощью обратного вызова
    fs.readFile('./titles.json', function(err, data) {
      // Если происходит ошибка, она регистрируется, а клиенту
      // возвращается сообщение "Server Error"
      if (err) {
        console.error(err);
        res.end('Server Error');
      }
      else {
        // Синтаксический разбор данных, полученных из текста JSON-файла
        var titles = JSON.parse(data.toString());

        // Загрузка HTML-шаблона и выполнение обратного
        // вызова после завершения загрузки
        fs.readFile('./template.html', function(err, data) {
          if (err) {
            console.error(err);
```

```
            res.end('Server Error');
          }
          else {
            var tmpl = data.toString();
            // Сборка HTML-страницы, показывающей заголовки блога
            var html = tmpl.replace('%', titles.join('</li><li>'));
            res.writeHead(200, {'Content-Type': 'text/html'});
            // Передача HTML-страницы пользователю
            res.end(html);
          }
        });
      }
    });
  }
}).listen(8000, "127.0.0.1");
```

В этом примере используются три уровня вложенности обратных вызовов:

```
http.createServer(function(req, res) { ...
  fs.readFile('./titles.json', function (err, data) { ...
    fs.readFile('./template.html', function (err, data) { ...
```

Наличие трех уровней вложенности обратных вызовов не означает, что код плох, тем не менее в общем случае чем больше уровней вложенности вы используете, тем более фрагментарным выглядит код и тем сложнее его переработка и тестирование. Поэтому стремитесь ограничивать количество уровней вложенности обратных вызовов. Преобразовать код можно с помощью именованных функций, реализующих отдельные уровни вложенности обратных вызовов. В результате увеличится количество строк, но зато код будет проще поддерживать, тестировать и перерабатывать. Код в листинге 3.7 эквивалентен коду, приведенному ранее в листинге 3.6.

**Листинг 3.7.** Пример уменьшения количества уровней вложенности обратных вызовов с помощью промежуточных функций

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

// Клиентский запрос сначала поступает сюда
var server = http.createServer(function (req, res) {
  // Управление передается getTitles
```

```
  getTitles(res);
}).listen(8000, "127.0.0.1");

// getTitles извлекает заголовки и передает управление getTemplate
function getTitles(res) {
  fs.readFile('./titles.json', function (err, data) {
    if (err) {
      hadError(err, res);
    }
    else {
      getTemplate(JSON.parse(data.toString()), res);
    }
  })
}

// getTemplate загружает файл шаблона и передает управление formatHtml
function getTemplate(titles, res) {
  fs.readFile('./template.html', function (err, data) {
    if (err) {
      hadError(err, res);
    }
    else {
      formatHtml(titles, data.toString(), res);
    }
  })
}

// formatHtml получает заголовки и шаблон, а затем передает ответ клиенту
function formatHtml(titles, tmpl, res) {
  var html = tmpl.replace('%', titles.join('</li><li>'));
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(html);
}
```

```
// Если происходит ошибка, hadError выводит ошибку
// на консоль и передает клиенту сообщение "Server Error"
function hadError(err, res) {
  console.error(err);
  res.end('Server Error');
}
```

Можно также уменьшить количество вложений, источником которых являются блоки if/else. Для этого используется другая распространенная идиома Node-программирования — ранний возврат из функции. Листинг 3.8 функционально аналогичен двум предыдущим. Отличие заключается в раннем возврате из функций, что позволяет избежать чрезмерной вложенности блоков if/else. При этом также явно определяются условия завершения выполнения функции.

**Листинг 3.8.** Пример уменьшения количества вложений за счет раннего возврата из функций

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

var server = http.createServer(function (req, res)
  getTitles(res);
}).listen(8000, "127.0.0.1");

function getTitles(res) {
  fs.readFile('./titles.json', function (err, data) {
    // Вместо создания ветви else выполняется возврат,
    // поскольку из-за ошибки функция завершает выполнение
    if (err) return hadError(err, res)
    getTemplate(JSON.parse(data.toString()), res)
  })
}

function getTemplate(titles, res) {
  fs.readFile('./template.html', function (err, data) {
    if (err) return hadError(err, res)
    formatHtml(titles, data.toString(), res)
  })
```

```
}

function formatHtml(titles, tmpl, res) {
  var html = tmpl.replace('%', titles.join('</li><li>'));
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(html);
}

function hadError(err, res) {
  console.error(err)
  res.end('Server Error')
}
```

Теперь, когда мы научились выполнять обратные вызовы для обработки одиночных событий при решении таких задач, как определение откликов в рамках чтения файлов и запросов веб-сервера, поговорим об обработке событий с помощью генераторов событий.

**принятое в node Соглашение относительно обратных вызовов**

В большинстве встроенных Node-модулей используются обратные вызовы с двумя аргументами. Первый аргумент служит для передачи ошибки (в случае ее возникновения), второй — результатов. Аргумент, передающий ошибку, часто обозначается как er или err.

Вот типичный пример использования этого соглашения:

```
var fs = require('fs');
fs.readFile('./titles.json', function(er, data) {
  if (er) throw er;
  // Что-то делаем с данными при отсутствии ошибки
});
```

### 3.2.2. Обработка повторяющихся событий с помощью генераторов событий

Генераторы событий инициируют события и имеют средства их обработки. В Node некоторые важные API-компоненты, такие как HTTP-серверы, TCP-серверы

потоки данных, реализованы в виде генераторов событий. Вы также можете создавать собственные генераторы событий.

Как упоминалось ранее, события обрабатываются с помощью слушателей. *Слушатель* связывает событие с функцией обратного вызова, которая выполняется при возникновении этого события. Например, TCP-сокет в Node поддерживает событие data, которое возникает, когда сокету становятся доступны новые данные:

```
socket.on('data', handleData);
```

Давайте поговорим об использовании событий data для создания эхо-сервера.

### *Пример генератора событий*

Эхо-сервер — это простой пример приложения, в котором могут возникать повторяющиеся события. Он отсылает переданные ему данные обратно, создавая эффект эха (рис. 3.8).

```
2. Shell
Last login: Sun Nov 27 15:08:28 on ttys000
Mike-Cantelons-MacBook|~$ telnet 127.0.0.1 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Line one
Line one
Line two
Line two
```

**Рис. 3.8.** Эхо-сервер дублирует переданные ему данные

В листинге 3.9 представлен код реализации эхо-сервера. После подключения к серверу клиента создается сокет. Этот сокет представляет собой генератор событий, к которому можно добавить слушатель. При этом используется метод on, реагирующий на событие data. Событие data возникает всякий раз, когда сокету становятся доступны новые данные.

**Листинг 3.9.** Использование метода on для реакции на события

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

var server = net.createServer(function(socket) {
  // Событие data обрабатывается при считывании новых данных
  socket.on('data', function(data) {
    // Данные записываются (возвращаются в виде эха) обратно клиенту
    socket.write(data);
  });
```

```
});

server.listen(8888);
```

Чтобы запустить эхо-сервер, введите такую команду:

```
node echo_server.js
```

Для подключения к запущенному эхо-серверу воспользуйтесь следующей командой:

```
telnet 127.0.0.1 8888
```

Данные, отправленные серверу в рамках установленного соединения telnet-сеанса, возвращаются обратно telnet-сеансу.

**telnet для Windows**

Если вы используете операционную систему Microsoft Windows, приложение telnet может быть по умолчанию не установлено, поэтому устанавливать его придется отдельно. Инструкции по установке telnet для разных версий Windows можно найти в сети TechNet (http://mng.bz/egzr).

### *Отклик на одиночное событие*

Можно определять слушателей, которые будут многократно реагировать на события, как показано в предыдущем примере. Можно также задавать слушателей, реагирующих на единственное событие. В коде, приведенном в листинге 3.10, используется метод once. Этот метод изменяет предыдущий пример с эхо-сервером таким образом, чтобы отправлять пользователю обратно только первый фрагмент данных.

**Листинг 3.10.** Демонстрация метода once для отклика на одиночное событие

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

var server = net.createServer(function(socket) {
  // Событие data будет обработано только один раз
  socket.once ('data', function(data) {
    socket.write(data);
  });
});
```

```
server.listen(8888);
```

### *Система публикации/подписки на базе генераторов событий*

В предыдущем примере использовалась встроенная в Node API-библиотека которая задействовала генераторы событий. Однако благодаря встроенному в Node модулю events можно создавать собственные генераторы событий.

В следующем примере кода определяется генератор событий channel с единственным слушателем, который отвечает всем подключившимся к каналу. Обратите внимание на использование метода on, чтобы добавить слушатель к генератору событий (либо в качестве альтернативы можно задействовать более длинную форму метода addListener):

```
var EventEmitter = require('events').EventEmitter;
var channel = new EventEmitter();
channel.on('join', function() {
  console.log("Welcome!");
});
```

Обратный вызов join никогда не произойдет, поскольку в данном примере не случается ни одного события. Добавить строку кода, которая будет вызывать событие, можно с помощью функции emit:

```
channel.emit('join');
```

**именование событий**

В качестве имен событий используются произвольные строковые значения, такие как data или join, в том числе длинные строки, например: some crazy long event name. Существует лишь одно зарезервированное для специального события имя error — его мы рассмотрим позже.

В главе 2 рассматривалось создание приложения для чата, функции публикации и подписки в котором поддерживались с помощью модуля Socket.io. Давайте создадим код, реализующий логику публикации/подписки.

Если запустить на выполнение сценарий, код которого приведен в листинге 3.11, мы получим простой сервер для чата. Канал сервера для чата реализован в виде генератора событий, который откликается на события join, генерируемые клиентами. Как только клиент присоединяется к каналу, код слушателя событий

join, в свою очередь, создает в канале дополнительный слушатель, ориентированный на клиента. Этот слушатель предназначается для обработки события broadcast и записи любого сообщения, транслируемого сокету клиента. Названия событий join и broadcast выбраны произвольным образом. При желании вы можете выбрать для них другие названия.

**Листинг 3.11.** Простая система публикации/подписки, созданная на основе генер

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

var channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};

// Добавление слушателя для события join, сохраняющего клиентский объе
// пользователя, что позволяет приложению отсылать данные
// обратно пользователю
channel.on('join', function(id, client) {
  this.clients[id] = client;
  this.subscriptions[id] = function(senderId, message) {
    // Игнорируем данные, непосредственно транслируемые пользователем
    if (id != senderId) {
      this.clients[id].write(message);
    }
  }
  // Добавление слушателя события broadcast для текущего пользователя
  this.on('broadcast', this.subscriptions[id]);
});

var server = net.createServer(function (client) {
  var id = client.remoteAddress + ':' + client.remotePort;
  client.on('connect', function() {
    // Генерируем событие join после подключения пользователя
    // к серверу, указывая идентификатор пользователя
    // и клиентский объект
    channel.emit('join', id, client);
```

```
  });
  client.on('data', function(data) {
    data = data.toString();
    // Генерируем событие broadcast для канала, задавая
    // идентификатор пользователя и сообщение, когда
    // какой-либо пользователь отсылает данные
    channel.emit('broadcast', id, data);
  });
});
server.listen(8888);
```

После запуска сервера для чата откройте новое окно командной строки и введите следующую команду для входа в чат:

```
telnet 127.0.0.1 8888
```

Если открыто несколько окон командной строки, то вы увидите, как символы, введенные в одном окне, будут повторяться в других окнах.

Проблема, связанная с этим сервером для чата, заключается в том, что когда пользователь разрывает соединение и покидает комнату чата, слушатель остается активным, продолжая попытки записать данные на отключившегося клиента. Подобная ситуация ведет к ошибке. Чтобы устранить проблему, нужно добавить слушатель к генератору событий канала (листинг 3.12), а также добавить к слушателю события close сервера логику генерирования события leave канала. Событие leave фактически удаляет слушатель событий broadcast, изначально добавленный к серверу.

**Листинг 3.12.** Создание слушателя, выполняющего очистку после отключения клиента

```
...
// Создание слушателя для события leave
channel.on('leave', function(id) {
  channel.removeListener('broadcast', this.subscriptions[id]);
  // Удаление слушателя события broadcast для указанного клиента
  channel.emit('broadcast', id, id + " has left the chat.\n");
});

var server = net.createServer(function (client) {
...
  client.on('close', function() {
```

```
    // Генерирование события leave после отключения клиента
    channel.emit('leave', id);
  });
});
server.listen(8888);
```

**обработка ошибок**

При создании генераторов событий применяется соглашение, суть которого заключается в том, что вместо непосредственного вызова ошибки можно генерировать событие error. В результате появляется возможность определить логику отклика на нестандартное событие путем присваивания одному или нескольким слушателям этого типа события.

Следующий код демонстрирует порядок обработки слушателем событий типа error путем регистрации в консоли:

```
var events = require('events');
var myEmitter = new events.EventEmitter();
myEmitter.on('error', function(err) {
  console.log('ERROR: ' + err.message);
});
myEmitter.emit('error', new Error('Something is wrong.'));
```

Если не задан ни один слушатель для события этого типа, то в результате генерирования события типа error генератор событий выведет на экран трассу стека (список программных инструкций, которые вызывались до момента возникновения ошибки) и остановит выполнение программы. Трасса стека будет указывать на ошибку, тип которой определяется с помощью второго аргумента вызова emit. Подобное поведение присуще только событиям типа error. Если генерируются другие типы событий, для которых не существует слушателей, ничего не происходит.

Если событие типа error генерируется без объекта error, поддерживаемого в качестве второго аргумента, при трассировке стека выводится сообщение об ошибке (Uncaught, unspecified ‘error’ event), после чего выполнение приложения прекращается. В этом случае можно воспользоваться неприветствуемым методом

обработки ошибки, суть которого заключается в определении собственного отклика путем создания глобального обработчика с помощью следующего кода:

```
process.on('uncaughtException', function(err){
  console.error(err.stack);
  process.exit(1);
});
```

Имеются и другие альтернативы, например домены (http://nodejs.org/api/domain.html), но их реализация пока находится на стадии эксперимента.

Если вы хотите приостановить чат в силу каких-либо причин, но не желаете завершать работу сервера, воспользуйтесь методом removeAllListeners генератора событий для удаления всех слушателей выбранного типа. В следующем примере кода показано, как использовать этот подход в нашем примере сервера для чата:

```
channel.on('shutdown', function() {
  channel.emit('broadcast', '', "Chat has shut down.\n");
  channel.removeAllListeners('broadcast');
});
```

Затем можно добавить поддержку команды чата, которая вызовет его завершение. Для этого измените слушателя события data, как показано в следующем примере кода:

```
client.on('data', function(data) {
  data = data.toString();
  if (data == "shutdown\r\n") {
    channel.emit('shutdown');
  }
  channel.emit('broadcast', id, data);
});
```

Теперь если любой пользователь чата введет команду shutdown, это приведет к выходу из чата всех остальных пользователей.

Если вы хотите дать пользователям возможность входить в чат с учетом подключенных в данный момент пользователей, можете воспользоваться следующим методом listeners, возвращающим массив слушателей события данного

типа:

```
channel.on('join', function(id, client) {
  var welcome = "Welcome!\n"
        + 'Guests online: ' + this.listeners('broadcast').length;
  client.write(welcome + "\n");
  ...
```

Чтобы увеличить количество слушателей, которыми может располагать генератор событий, и избежать появления предупреждений, выводимых Node при наличии более десяти слушателей, воспользуйтесь методом setMaxListeners method. Используя генератор событий канала в качестве примера, можно увеличить количество разрешенных слушателей с помощью следующего кода:

```
channel.setMaxListeners(50);
```

### *Создание наблюдателя файлов, в котором расширены возможности генератора событий*

Чтобы использовать в своем коде поведение генератора событий, нужно создать новый JavaScript-класс, наследуемый от генератора событий. Например, можно создать класс Watcher, который будет обрабатывать файлы, находящиеся в выбранной папке файловой системы. На основе этого класса можно создать утилиту, которая будет просматривать в файловой системе заданную папку, преобразовывать в нижний регистр имена помещенных в эту папку файлов, а затем копировать эти файлы в другую папку.

Чтобы расширить возможности генератора событий, достаточно выполнить три шага.

1. Создать конструктор класса.

2. Наследовать поведение генератора событий.

3. Расширить это поведение.

В следующем коде показано, как создать конструктор класса Watcher. Этот конструктор в качестве аргументов получает нужную папку, а также папку для сохранения измененных файлов:

```
function Watcher(watchDir, processedDir) {
  this.watchDir = watchDir;
  this.processedDir = processedDir;
```

```
}
```

Далее нужно добавить логику наследования поведения генератора событий:

```
var events = require('events')
  , util = require('util');
util.inherits(Watcher, events.EventEmitter);
```

Обратите внимание на использование функции inherits, которая является частью встроенного в Node модуля util. С помощью функции inherits обеспечивается простой способ наследования поведения другого объекта.

Инструкция inherits, используемая в предыдущем примере кода, эквивалентна следующему JavaScript-коду:

```
Watcher.prototype = new events.EventEmitter();
```

После создания объекта Watcher нужно расширить возможности методов, наследуемых от класса EventEmitter. Это делается в двух новых методах, как показано в листинге 3.13.

**Листинг 3.13.** Расширение возможностей генератора событий

```
var fs = require('fs')
  , watchDir = './watch'
  , processedDir = './done';

// Расширение возможностей класса EventEmitter путем добавления метода
// обработки файлов
Watcher.prototype.watch = function() {
  // Сохранение ссылки на объект Watcher, используемой при обратном
  // вызове readdir
  var watcher = this;
  fs.readdir(this.watchDir, function(err, files) {
    if (err) throw err;
    for(var index in files) {
      // Обработка каждого файла в заданной папке
      watcher.emit('process', files[index]);
    }
  })
}

// Расширение класса EventEmitter путем добавления метода start,
```

```
// инициирующего просмотр
Watcher.prototype.start = function() {
  var watcher = this;
  fs.watchFile(watchDir, function() {
    watcher.watch();
  });
}
```

Метод watch циклически просматривает папку, обрабатывая все найденные файлы. Метод start инициирует просмотр папки. В процессе мониторинга используется Node-функция fs.watchFile, и если в заданной папке что-либо обнаруживается, вызывается метод watch, выполняется циклический просмотр папки и для каждого найденного файла генерируется событие process.

После определения класса Watcher можно воспользоваться им, создав объект Watcher с помощью следующего кода:

```
var watcher = new Watcher(watchDir, processedDir);
```

В только что созданном объекте Watcher можно использовать метод on, унаследованный от класса генератора событий, чтобы создать логику обработки каждого файла, как показано в следующем примере кода:

```
watcher.on('process', function process(file) {
  var watchFile = this.watchDir + '/' + file;
  var processedFile = this.processedDir + '/' + file.toLowerCase();

  fs.rename(watchFile, processedFile, function(err) {
    if (err) throw err;
  });
});
```

Теперь, после создания всего необходимого кода, инициировать мониторинг папки можно с помощью такой команды:

```
watcher.start();
```

После помещения кода Watcher в сценарий и создания папок для мониторинга и обработки вы получаете возможность запускать этот сценарий в Node, помещать файлы в папку просмотра и наблюдать за тем, как переименованные с применением символов нижнего регистра файлы появляются в целевой папке. В данном случае генератор событий выступает в роли полезного класса, на основе которого создаются новые классы.

Освоив применение обратных вызовов для создания асинхронного кода,

обрабатывающего одиночные события, и генераторов событий для создания кода, управляющего повторяющимися событиями, вы сделали еще один шаг на пути к овладению искусством управления поведением Node-приложений. Также имейте в виду, что в одиночный обратный вызов или в слушатель генератора событий можно включить код, выполняющий дополнительные асинхронные операции. Однако если важен порядок выполнения этих операций, вы можете столкнуться с новой проблемой: как контролировать точное время запуска каждой операции в группе выполняющихся асинхронных операций.

Однако прежде чем мы займемся контролем времени запуска заданий (об этом рассказывается в разделе 3.3), рассмотрим некоторые проблемы, характерные для асинхронного программирования.

### 3.2.3. Проблемы асинхронного программирования

При создании асинхронных приложений следует уделять пристальное внимание порядку выполнения в них операций, а также отслеживать их состояние: условия в цикле событий, переменные приложений и любые другие ресурсы, которые могут изменить программную логику.

Например, на платформе Node в цикле событий нужно отслеживать код, который не завершил обработку данных. Пока асинхронный код продолжает выполняться, Node-процесс не завершается. Непрерывно выполняющийся Node-процесс — это желательное поведение для таких компонентов, как веб-сервер, но не слишком желательное для продолжительных процессов, которые должны завершаться через определенные промежутки времени, например для инструментов командной строки. В цикле событий отслеживаются любые соединения с базой данных, и до тех пор, пока они не закрыты, они не дадут Node завершить работу.

Кроме того, при неаккуратном программировании непредусмотренным образом могут меняться переменные приложений. В листинге 3.14 представлен пример неприятных последствий, к которым может привести выполнение асинхронного кода. Если этот пример кода выполнять в синхронном режиме, вы вправе ожидать на выходе строку "The color is blue". Однако поскольку код является асинхронным, значение переменной color изменится до выполнения файла console.log, и в результате мы получим строку "The color is green".

**Листинг 3.14.** Как поведение области видимости приводит к ошибкам

```
function asyncFunction(callback) {
  setTimeout(callback, 200);
}
```

```
var color = 'blue';

asyncFunction(function() {
  // Эта инструкция выполняется последней (на 200 мс позже)
  console.log('The color is ' + color);
});
color = 'green';
```

Чтобы «заморозить» значение переменной color, нужно изменить код, включив в него JavaScript-замыкание. В примере из листинга 3.15 вызов функции asyncFunction заключается в анонимную функцию, имеющую аргумент color. Затем вызывается анонимная функция, которая отсылает текущее значение переменной color. Поскольку аргумент color относится к анонимной функции, он становится локальным для ее области видимости, и если значение переменной color изменяется вне анонимной функции, локальная версия этой переменной не затрагивается.

**Листинг 3.15.** Использование анонимной функции для сохранения значения глобальной переменной

```
function asyncFunction(callback) {
  setTimeout(callback, 200);
}
var color = 'blue';

(function(color) {
  asyncFunction(function() {
    console.log('The color is ' + color);
  })
})(color);

color = 'green';
```

Это лишь один из многих трюков JavaScript-программирования, применяемых при разработке Node-приложений.

**замыкания**

Дополнительные сведения о замыканиях можно найти на сайте https://developer.mozilla.org/ru-ru/docs/JavaScript/Guide/Closures.

После знакомства с использованием замыканий для управления состоянием приложения обратимся к диспетчеризации асинхронного кода, позволяющей контролировать порядок выполнения операций в приложении.

## 3.3. Порядок выполнения асинхронного кода

В процессе выполнения асинхронной программы некоторые операции могут происходить в любое время, которое никак не связано с выполнением остальной части программы, и это не приводит к каким-либо проблемам. Однако существуют и такие операции, которые должны выполняться только после или только перед выполнением других операций.

В сообществе разработчиков Node-приложений концепция разбиения асинхронных операций на поочередно выполняемые группы называется *управлением порядком выполнения* (flow control). Как показано на рис. 3.9, существует две разновидности такого выполнения: *последовательное* (serial) и *параллельное* (parallel).

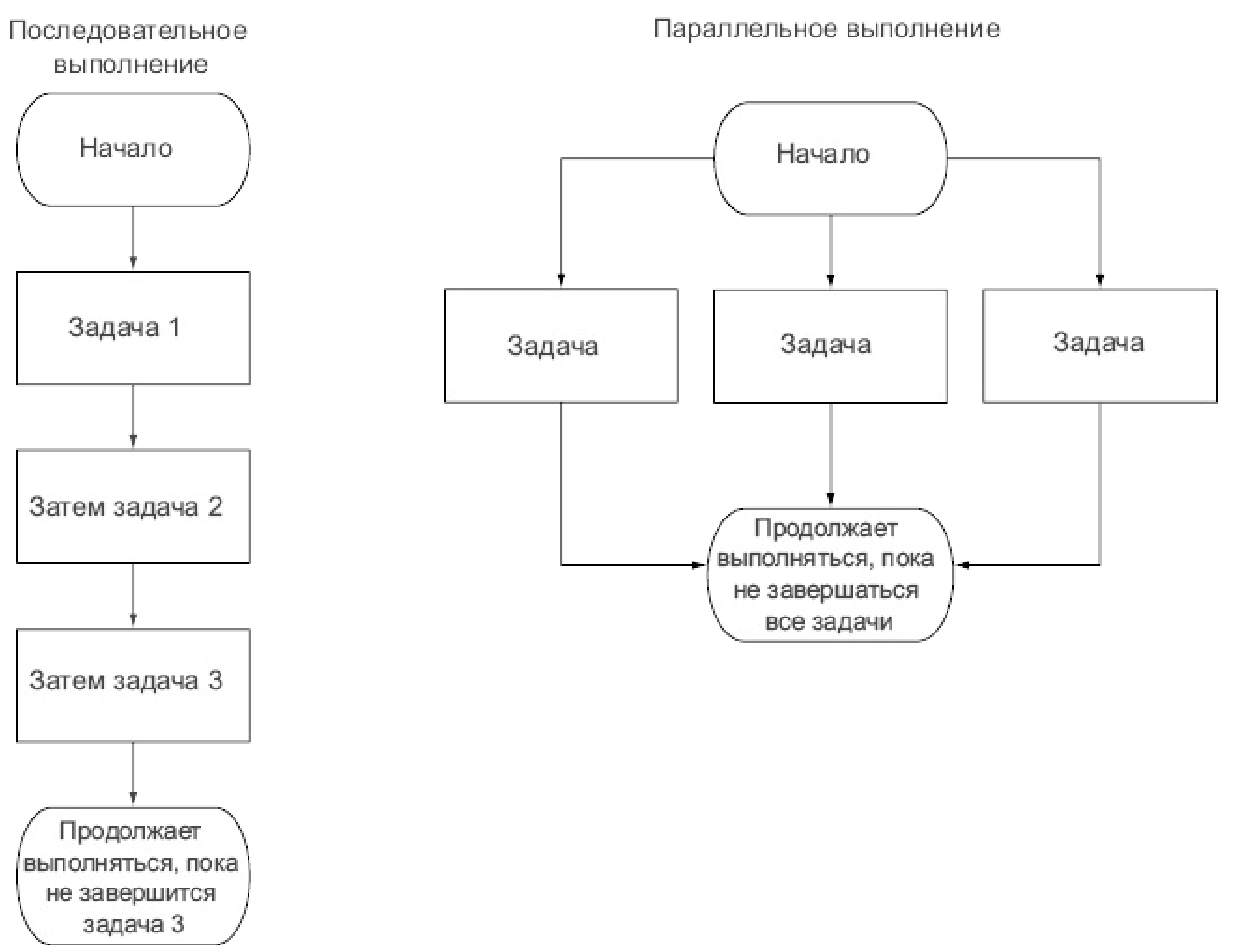



**Рис. 3.9.** Последовательное выполнение асинхронных операций концептуально соответствует синхронной логике: операции выполняются поочередно, одна за другой. Параллельные операции могут выполняться одновременно

Операции, выполняемые одна за другой, называются *последовательными*. В

качестве простого примера можно привести операции создания папки и последующего сохранения в ней файлов. Невозможно сохранить файл в папке, которая еще не создана.

Операции, которые не обязательно выполнять поочередно, называются *параллельными*. Начало и завершение каждой из этих операций никак не связаны с другими, но все они должны быть завершены перед выполнением последующего фрагмента кода. В качестве примера подобных параллельных операций можно привести процесс загрузки нескольких файлов, которые затем пакуются в zip-архив. Файлы могут загружаться одновременно, но перед созданием архива все загрузки должны быть завершены.

Чтобы управлять последовательным и параллельным выполнением операций, нужно вести своего рода «программную бухгалтерию». Если вы реализуете последовательный порядок, нужно отслеживать, какая операция выполняется в данный момент, или поддерживать очередь операций, ожидающих выполнения. Если же реализуется параллельный порядок, нужно отслеживать, сколько операций выполнено полностью.

Подобную «бухгалтерию» обеспечивают инструменты управления порядком выполнения операций, позволяя легко группировать последовательные или параллельные асинхронные операции. Хотя в настоящее время сообществом разработчиков создано немало надстроек, реализующих подобную асинхронную логику, рекомендуется реализовать такое управление самостоятельно. Это позволит вам лучше разобраться в его принципах и методиках устранения проблем асинхронного программирования.

В данном разделе показано:

- когда требуется управлять последовательным выполнением операций;
- как реализовать управление последовательным выполнением операций;
- как реализовать управление параллельным выполнением операций;
- как использовать модули управления порядком выполнения операций от сторонних производителей.

И начнем мы с ответа на вопрос о том, когда и как в асинхронном программировании требуется управлять последовательным выполнением операций.

### 3.3.1. Когда требуется управлять последовательным выполнением операций

Для поочередного выполнения асинхронных операций можно использовать асинхронные вызовы, но если количество таких операций достаточно велико, их сначала нужно упорядочить. Если этого не сделать, код получится некрасивым, с большим количеством вложенных обратных вызовов.

Следующий код представляет собой пример поочередного выполнения операций с помощью обратных вызовов. В этом примере объект setTimeout применяется для моделирования операций, требующих определенного времени на выполнение. Например, первая операция выполняется одну секунду, выполнение следующей занимает полсекунды, последняя требует для выполнения одной десятой секунды. Обратите внимание, что setTimeout обеспечивает лишь имитацию. В реальном коде могут происходить операции чтения файлов, создания HTTP-отчетов и т.п. Несмотря на небольшой размер этого фрагмента кода, он довольно сложен, к тому же не существует простого способа программно добавить еще одну операцию.

```
setTimeout(function() {
  console.log('I execute first.');
  setTimeout(function() {
    console.log('I execute next.');
    setTimeout(function() {
      console.log('I execute last.');
    }, 100);
  }, 500);
}, 1000);
```

В качестве альтернативы можно воспользоваться каким-нибудь инструментом управления порядком выполнения, например Nimble. Этот инструмент прост в работе, к тому же его код занимает очень мало места (в сжатом виде около 837 байт). Чтобы установить Nimble, воспользуйтесь следующей командой:

```
npm install nimble
```

Теперь используйте код из листинга 3.16, чтобы заново реализовать предыдущий фрагмент, обеспечив последовательное выполнение операций.

**Листинг 3.16.** Управление последовательным выполнением операций с помощью надстройки, созданной сообществом разработчиков Node-приложений

```
var flow = require('nimble');

// Массив поочередно вызываемых Nimble-функций
flow.series([
  function (callback) {
```

```
        setTimeout(function() {
          console.log('I execute first.');
          callback();
        }, 1000);
      },
      function (callback) {
        setTimeout(function() {
          console.log('I execute next.');
          callback();
        }, 500);
      },
      function (callback) {
        setTimeout(function() {
          console.log('I execute last.');
          callback();
        }, 100);
      }
    ]);
```

Хотя в этой реализации, в которой происходит управление порядком выполнения операций, больше строк кода, этот код в общем случае проще для чтения и поддержки. Вполне возможно, что вам не придется постоянно управлять порядком выполнения операций, но если это потребуется, используйте инструмент Nimble — он позволит избежать вложенности обратных вызовов и тем самым сделает код более понятным.

Теперь, когда вы познакомились с примером управления последовательным выполнением операций с помощью специализированного инструмента, давайте попробуем реализовать такое управление самостоятельно.

### 3.3.2. Управление последовательным выполнением операций

Чтобы обеспечить поочередное выполнение нескольких асинхронных операций, нужно поместить эти операции в массив в порядке их выполнения. Этот массив функционирует как очередь: как только завершается выполнение одной операции, из массива извлекается следующая (рис. 3.10).

**Рис. 3.10.** Иллюстрация управления последовательным выполнением операций

Каждая операция существует в массиве в виде функции. Как только выполнение операции завершается, вызывается функция обработки, которая выводит на экран статус ошибки и результаты. В данной реализации функция обработки прекращает выполняться в случае ошибки. Если ошибки нет, обработчик извлекает следующую операцию из очереди и выполняет ее.

Чтобы продемонстрировать реализацию управления последовательным выполнением операций, мы разработаем простое приложение, которое будет выводить на экран название и URL-адрес одной статьи из выбранного случайным образом RSS-канала. Список доступных RSS-каналов указывается в текстовом файле. Результат выполнения этого приложения мог бы выглядеть примерно так:

**Of Course ML Has Monads!**

http://lambda-the-ultimate.org/node/4306

В рассматриваемом примере используются два вспомогательных модуля из npm-хранилища. Сначала откройте окно командной строки и с целью создания папки для данного примера и установки вспомогательных модулей введите следующие команды:

**mkdir random_story**

cd random_story

npm install request

npm install htmlparser

Модуль request представляет собой упрощенный HTTP-клиент, который можно использовать для выборки RSS-данных. Модуль htmlparser обладает набором функций, с помощью которых исходные RSS-данные преобразуются в JavaScript-структуры данных.

Затем в новой папке создадим файл random_story.js, код которого представлен в листинге 3.17.

**Листинг 3.17.** Реализация управления последовательным выполнением операций в простом приложении

```
var fs = require('fs');
var request = require('request');
var htmlparser = require('htmlparser');
var configFilename = './rss_feeds.txt';

// Операция 1. Проверка существования файла, содержащего список
// URL-адресов RSS-каналов
function checkForRSSFile () {
  fs.exists(configFilename, function(exists) {
    if (!exists)
      // Ранний возврат при ошибке
      return next(new Error('Missing RSS file: ' + configFilename));

    next(null, configFilename);
  });
}

// Операция 2. Чтение и синтаксический разбор файла,
// содержащего URL-адреса RSS-каналов
function readRSSFile (configFilename) {
  fs.readFile(configFilename, function(err, feedList) {
    if (err) return next(err);

    // Преобразование списка URL-адресов RSS-каналов в строку
    // и помещение ее в массив URL-адресов RSS-каналов
    feedList = feedList
            .toString()
            .replace(/^\s+|\s+$/g, '')
            .split("\n");
    // Случайный выбор URL-адреса RSS-канала из массива URL-адресов
    var random = Math.floor(Math.random()*feedList.length);
```

```
    next(null, feedList[random]);
  });
}

// Операция 3. Выполнение HTTP-запроса и получение данных для
// выбранного RSS-канала
function downloadRSSFeed (feedUrl) {
  request({uri: feedUrl}, function(err, res, body) {
    if (err) return next(err);
    if (res.statusCode != 200)
      return next(new Error('Abnormal response status code'))

    next(null, body);
  });
}

// Операция 4. Синтаксический разбор RSS-данных в массив элементов
function parseRSSFeed (rss) {
  var handler = new htmlparser.RssHandler();
  var parser = new htmlparser.Parser(handler);
  parser.parseComplete(rss);

  if (!handler.dom.items.length)
    return next(new Error('No RSS items found'));

  var item = handler.dom.items.shift();
  // Вывод заголовка и URL-адреса первого элемента
  // RSS-канала (при его наличии)
  console.log(item.title);
  console.log(item.link);
}

// Добавление каждой операции в массив в порядке выполнения
var tasks = [ checkForRSSFile,
```

```
        readRSSFile,
        downloadRSSFeed,
        parseRSSFeed ];
// Следующая вызванная функция выполняет каждую операцию
function next(err, result) {
  // Если при выполнении операции возникает ошибка,
  // генерируется исключение
  if (err) throw err;

  // Из массива операций выбирается следующая операция
  var currentTask = tasks.shift();

  if (currentTask) {
    // Выполнение текущей операции
    currentTask(result);
  }
}

// Начало последовательного выполнения операций
next();
```

Прежде чем приступить к тестированию приложения, создайте файл rss_feeds.txt, находящийся в той же папке, что и сценарий приложения. Поместите URL-адреса RSS-каналов в текстовый файл, чтобы каждый адрес располагался на отдельной строке. После завершения создания файла откройте командную строку и введите следующие команды, чтобы изменить папку приложения и вызвать сценарий на выполнение:

```
cd random_story
node random_story.js
```

Как показано в этой реализации, за счет управления последовательным выполнением операций можно при необходимости использовать несколько обратных вызовов без их вложения друг в друга.

Теперь, когда вы знаете, как управлять последовательным выполнением операций, давайте посмотрим, как выполнять асинхронные операции в параллельном режиме.

### 3.3.3. Управление параллельным выполнением операций

Чтобы выполнить несколько асинхронных операций параллельно, сначала нужно поместить эти операции в массив, но в данном случае порядок их размещения в массиве роли не играет. В рамках каждой операции должна вызываться функция обработки, увеличивающую количество выполненных операций на единицу. Как только все операции будут выполнены, функция обработки должна перейти к выполнению оставшейся части кода.

В качестве примера реализации управления параллельным выполнением операций создадим простое приложение, которое будет считывать содержимое нескольких текстовых файлов и показывать частоту использования слов в них. Считывание содержимого текстовых файлов осуществляется с помощью асинхронной функции readFile, которая позволяет выполнять эти операции в параллельном режиме. Принцип работы этой функции иллюстрирует рис. 3.11.

**Рис. 3.11.** Реализация управления параллельным выполнением операций для подсчета частоты использования слов в нескольких файлах

Результат выполнения приложения будет выглядеть следующим образом, хотя в

вашем случае он может быть длиннее:

```
would: 2
wrench: 3
writeable: 1
you: 24
```

Откройте окно командной строки и введите следующие команды, чтобы создать две папки. В одной папке будет находиться файл примера, во второй — анализируемые текстовые файлы:

```
mkdir word_count
cd word_count
mkdir text
```

Затем создайте в папке word_count файл word_count.js с кодом, представленным в листинге 3.18.

**Листинг 3.18.** Управление параллельным выполнением операций, реализованное в простом приложении

```
var fs = require('fs');
var completedTasks = 0;
var tasks = [];
var wordCounts = {};
var filesDir = './text';

function checkIfComplete() {
  completedTasks++;
  if (completedTasks == tasks.length) {
    // После выполнения всех операций вывод всех слов,
    // встречающихся в файлах, с указанием количества вхождений
    for (var index in wordCounts) {
      console.log(index +': ' + wordCounts[index]);
    }
  }
}

function countWordsInText(text) {
  var words = text
    .toString()
```

```
    .toLowerCase()
    .split(/\W+/)
    .sort();
  // Подсчет количества вхождений слова в тексте
  for (var index in words) {
    var word = words[index];
    if (word) {
      wordCounts[word] =
        (wordCounts[word]) ? wordCounts[word] + 1 : 1;
    }
  }
}

// Получение списка текстовых файлов в папке
fs.readdir(filesDir, function(err, files) {
  if (err) throw err;
  for(var index in files) {
    // Определение операции обработки каждого файла. В рамках каждой
    // операции происходит вызов функции, считывающей файл в асинхронн
    // режиме, и подсчет количества вхождений слова
    var task = (function(file) {
      return function() {
        fs.readFile(file, function(err, text) {
          if (err) throw err;
          countWordsInText(text);
          checkIfComplete();
        });
      }
    })(filesDir + '/' + files[index]);
    // Добавление каждой операции в массив функций, вызываемых
    // в параллельном режиме
    tasks.push(task);
  }
  // Начало выполнения каждой операции в параллельном режиме
```

```
  for(var task in tasks) {
    tasks[task]();
  }
});
```

Прежде чем тестировать приложение, создайте несколько текстовых файлов в ранее созданной папке. Завершив создание файлов, откройте окно командной строки и введите следующие команды, чтобы изменить папку приложения и вызвать на выполнение сценарий:

```
cd word_count
node word_count.js
```

Теперь, когда вы познакомились с принципами управления параллельным и последовательным выполнением операций, рассмотрим созданные сообществом разработчиков инструменты, позволяющие использовать преимущества такого управления и не требующие от вас приложение дополнительных усилий на его реализацию.

### 3.3.4. Инструменты от сообщества Node-разработчиков

Многие надстройки, созданные сообществом разработчиков Node-приложений, предлагают удобные инструменты управления порядком выполнения операций. Среди подобных надстроек можно отметить Nimble, Step и Seq. И хотя каждая из них заслуживает отдельного рассмотрения, мы остановимся на надстройке Nimble и используем ее в следующем примере.

**Надстройки для управления порядком выполнения операций**

Для получения дополнительных сведений о надстройках, предназначенных для управления порядком выполнения операций и созданных сообществом разработчиков Node-приложений, обратитесь к статье «Virtual Panel: How to Survive Asynchronous Programming in JavaScript», написанной Вернером Шустером (Werner Schuster) и Дио Синодиносом (Dio Synodinos). Найти ее можно на веб-сайте InfoQ (http://mng.bz/wKnV).

В листинге 3.19 представлен пример использования надстройки Nimble для управления параллельным выполнением операций. В этом примере происходит одновременная загрузка двух файлов с последующим их архивированием.

**этот пример не работает в Microsoft Windows**

Поскольку в Windows команды tar и curl не поддерживаются, в данной операционной системе этот пример работать не будет.

В этом примере перед запуском процедуры архивирования происходит проверка факта завершения загрузки файлов.

**Листинг 3.19.** Использование в простом приложении инструментов управления порядком выполнения операций, созданных сообществом Node-разработчиков

```
var flow = require('nimble')
var exec = require('child_process').exec;

// Загрузка исходного кода для данной версии
function downloadNodeVersion(version, destination, callback) {
  var url = 'http://nodejs.org/dist/node-v' + version + '.tar.gz';
  var filepath = destination + '/' + version + '.tgz';
  exec('curl ' + url + ' >' + filepath, callback);
}

// Последовательное выполнение набора операций
flow.series([
  function (callback) {
    // Параллельная загрузка файлов
    flow.parallel([
      function (callback) {
        console.log('Downloading Node v0.4.6...');
        downloadNodeVersion('0.4.6', '/tmp', callback);
      },
      function (callback) {
        console.log('Downloading Node v0.4.7...');
        downloadNodeVersion('0.4.7', '/tmp', callback);
      }
    ], callback);
  },
```

```
  function(callback) {
    console.log('Creating archive of downloaded files...');
    // Создание файла архива
    exec(
      'tar cvf node_distros.tar /tmp/0.4.6.tgz /tmp/0.4.7.tgz',
      function(error, stdout, stderr) {
        console.log('All done!');
        callback();
      }
    );
  }
]);
```

Сценарий определяет вспомогательную функцию, которая загружает указанную версию исходного Node-кода. Затем последовательно выполняются следующие две операции: параллельная загрузка двух версий Node и архивирование этих версий путем создания нового файла архива.

## 3.4. Резюме

В этой главе мы узнали, как организовать код приложения, распределив его по многократно используемым модулям, а также научили асинхронный код вести себя так, как требуется.

Система модулей в Node, основанная на спецификации CommonJS (www.commonjs.org/specs/modules/1.0/), позволяет легко реализовать многократное использование модулей путем присваивания значений свойствам объектов exports и module.exports. Система поиска модулей обладает большой гибкостью, давая возможность размещать модули в любом месте файловой системы, а затем загружать их по требованию. Помимо включения модулей в дерево исходного кода приложения мы также можем воспользоваться папкой node_modules для предоставления нескольким приложениям совместного доступа к коду модуля. При загрузке модуля по требованию с помощью файла package.json можно указать, какой файл в дереве файлов с исходным кодом модулей должен анализироваться в первую очередь.

Чтобы держать асинхронную логику под контролем, используются обратные вызовы, генераторы событий и механизм управления порядком выполнения операций. Обратные вызовы можно задействовать для обработки одиночных событий, но при этом нужно постараться не допустить чрезмерного усложнения кода. Генераторы событий могут быть полезными для организации асинхронной

логики, поскольку позволяют связать ее с концептуальной сущностью и легко контролировать с помощью слушателей.

В рамках управления порядком выполнения операций определяется, как должны выполняться асинхронные операции, поочередно или одновременно. Реализовать такое управление можно самостоятельно, но проще воспользоваться инструментами, предлагаемыми сообществом Node-разработчиков. Выбор в пользу той или иной надстройки, предназначенной для управления порядком выполнения операций, зависит от ваших личных предпочтений и ограничений, связанных с проектом или дизайном.

Теперь, после изучения материала этой главы, мы готовы приступить к разработке Node-приложений. В главах части II рассматриваются наиболее важные функциональные возможности Node — прикладные программные интерфейсы технологии HTTP. Следующая глава посвящена основам разработки веб-приложений в Node.

1 См. «JavaScript: Your New Overlord» на YouTube (www.youtube.com/watch?v=Trurfqh_6fQ).

2 См. страницу «Chrome Experiments» с примерами (www.chromeexperiments.com/).

3 Jslinux — эмулятор PC, написанный на JavaScript (http://bellard.org/jslinux/).

4 См. статью «List of languages that compile to JS» (https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS).

5 Дополнительные сведения о стандарте ECMAScript можно найти в соответствующей статье Википедии (http://ru.wikipedia.org/wiki/ECMAScript).

6 Обратите внимание, что существуют исключения, которые «блокируют» выполнение кода в браузере, поэтому использование следующих запросов не рекомендуется: alert, prompt, confirm и синхронный запрос XHR.

7 Для получения дополнительных сведений, относящихся к этой проблеме, просмотрите статью «The C10K problem» (www.kegel.com/c10k.html).

8 Дополнительные сведения по этой теме можно найти на странице «About» веб-сайта проекта Node (http://nodejs.org/about/).

# Часть 2. Разработка веб-приложений на платформе Node

Встроенная в Node HTTP-функциональность делает эту платформу идеальной средой разработки веб-приложений. Именно веб-приложения чаще всего создаются на платформе Node, и именно этому посвящена данная часть книги.

Сначала мы узнаем, как использовать встроенную в Node HTTP-функциональность. Затем мы освоим программное обеспечение промежуточного уровня, позволяющее расширить функциональность приложений, например добавить способность обрабатывать данные, введенные в форму. И наконец, мы поговорим о популярной среде разработки Express, призванной ускорить разработку веб-приложений, и о развертывании созданных приложений.

## Глава 4. Создание веб-приложений в Node

- Обработка HTTP-запросов в Node с помощью API
- Создание веб-службы RESTful
- Обслуживание статических файлов
- Получение данных, вводимых пользователями в формы
- Обеспечение безопасности приложений с помощью HTTPS

В этой главе мы познакомимся с теми инструментами Node, которые предназначены для создания HTTP-серверов. Мы также узнаем о модуле fs (**f**ile**s**ystem — файловая система), предназначенном для обслуживания статических файлов. Мы научимся разрабатывать веб-приложения, соответствующие наиболее распространенным требованиям, в частности мы сможем создавать низкоуровневые веб-службы RESTful, принимать данные, вводимые пользователем в HTML-формы, отслеживать процесс загрузки файла и защищать веб-приложения с помощью реализованного в Node протокола SSL (Secure Sockets Layer — Слой защищенных сокетов).

Ядро Node представляет собой мощную потоковую систему синтаксического разбора HTTP-кода, состоящую примерно из 1500 строк оптимизированного кода, написанного на языке C Node-разработчиком Райаном Далом (Ryan Dahl). Сочетание этой системы с низкоуровневым прикладным программным интерфейсом TCP, который Node экспонирует для JavaScript-кода, предоставляет в ваше распоряжение чрезвычайно гибкий низкоуровневый HTTP-сервер.

Как и большинство модулей, образующих ядро платформы Node, модуль http характеризуется простотой. Высокоуровневая «начинка» прикладных программных интерфейсов (API), существенно упрощающая разработку веб-приложений, отдается «на откуп» таким средам разработки от независимых производителей, как Connect или Express. На рис. 4.1 представлена структура веб-приложения, создаваемого в Node. Показаны низкоуровневые прикладные программные интерфейсы, относящиеся к ядру Node, а также абстракции и реализации, которые надстраиваются над этими «строительными блоками».

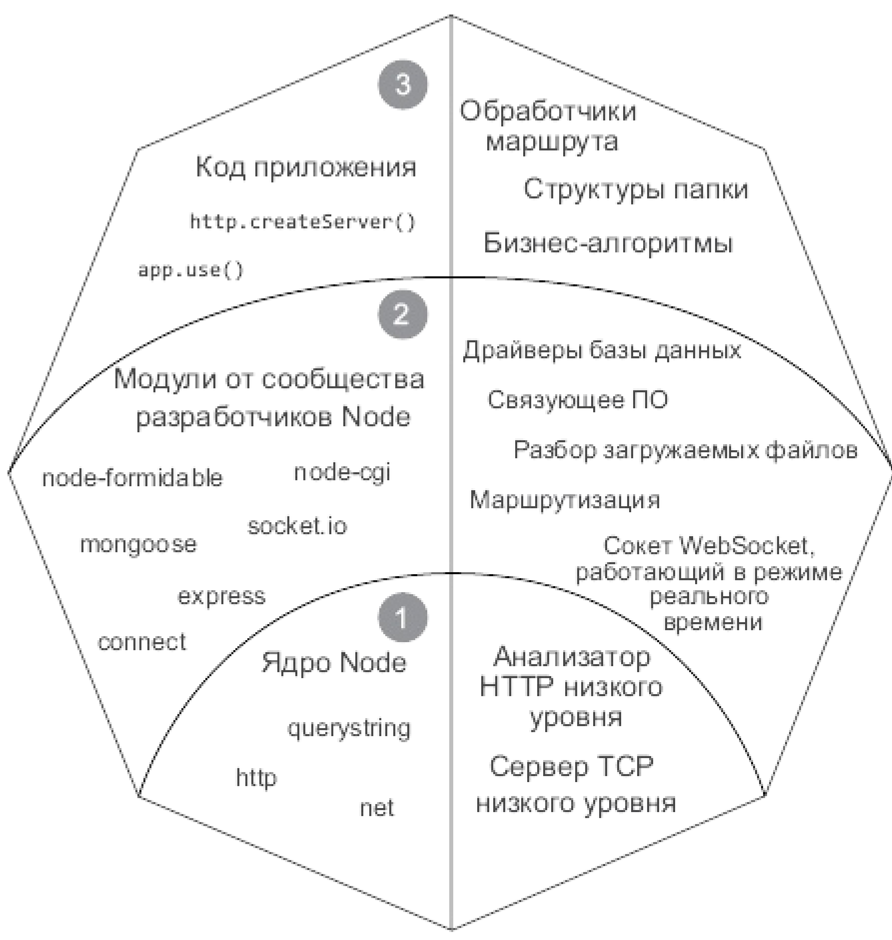



**Рис. 4.1.** Уровни, составляющие веб-приложение в Node

В этой главе непосредственно рассматриваются некоторые низкоуровневые API-интерфейсы, доступные в Node. Если же вас интересуют исключительно высокоуровневое программирование и среды разработки веб-приложений, такие как Connect или Express, которые рассматриваются в других главах, можете смело пропустить эту главу. Однако прежде чем приступить к созданию «продвинутых» веб-приложений с помощью Node, нужно познакомиться с фундаментальной API-библиотекой HTTP, которой можно пользоваться, не обращаясь к высокоуровневым инструментам и средам разработки приложений.

## 4.1. Знакомство с HTTP-сервером

Как уже отмечалось, в Node используется относительно низкоуровневый API-интерфейс. Интерфейс HTTP в Node также относится к низкоуровневым, если сравнивать его со средами разработки или языками программирования, такими как PHP, поэтому он остается быстродействующим и гибким.

Приступая к разработке надежных и производительных веб-приложений, мы узнаем:

- как Node предоставляет разработчикам входящие HTTP-запросы;
- как написать простейший HTTP-сервер, отвечающий сообщением «Hello World»;
- как считывать заголовки входящих запросов и устанавливать заголовки исходящих ответов;
- как установить код состояния HTTP-ответа.

Чтобы принимать входящие запросы, нужно создать HTTP-сервер. Давайте взглянем на интерфейс HTTP в Node.

### 4.1.1. Как Node предоставляет разработчикам входящие HTTP-запросы

Интерфейсы HTTP-сервера и HTTP-клиента в Node поддерживаются с помощью модуля http:

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

Чтобы создать HTTP-сервер, вызовите функцию http.createServer(). Эта функция имеет единственный аргумент — функцию обратного вызова, которая вызывается в каждом HTTP-запросе, принимаемом сервером. Функция обратного вызова *запроса* в качестве аргументов получает два объекта, request и response, которые обычно записываются в сокращенном виде, req и res:

```
var http = require('http');
var server = http.createServer(function(req, res){
  // обработка запроса
  });
```

Для каждого HTTP-запроса, получаемого сервером, вызывается функция обратного вызова, причем в качестве аргументов принимаются новые объекты req и res. Еще до выполнения обратного вызова Node выполняет синтаксический разбор запроса вплоть до конца HTTP-заголовков и предоставляет его как часть объекта req. Обратите внимание, что Node не может начать разбор тела запроса до тех пор, пока не будет сделан обратный вызов. Это поведение отличается от поведения серверных сред разработки, таких как PHP, в которых синтаксический разбор заголовков и тела запроса происходит до начала выполнения кода приложения. Node предлагает низкоуровневый интерфейс, позволяющий обрабатывать данные тела запроса по мере их разбора.

Node не выполняет запись отклика обратно клиенту в автоматическом режиме. После срабатывания обратного вызова запроса вы можете вручную завершить ответ с помощью метода res.end(), как показано на рис. 4.2. Это позволяет в течение жизни запроса (вплоть до завершения ответа) выполнять произвольную асинхронную логику. Если не завершить ответ, запрос «зависнет» до тех пор, пока клиент не отключится по тайм-ауту, или просто останется открытым.

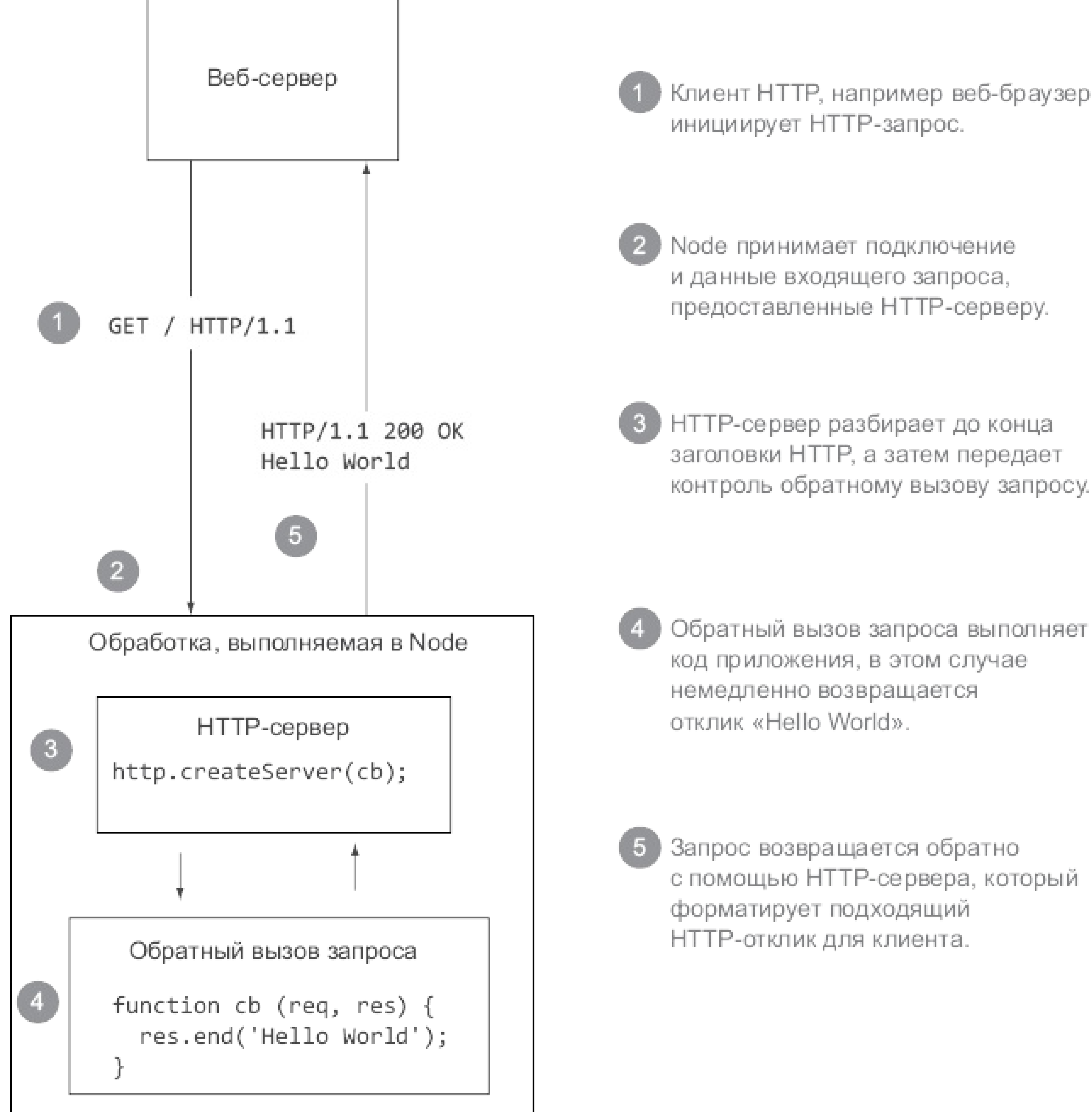



**Рис. 4.2.** Жизненный цикл HTTP-запроса, поддерживаемый в Node с помощью HTTP-сервера

Серверы в Node представляют собой длительные процессы, в течение своего жизненного цикла обслуживающие множество запросов.

### 4.1.2. Простейший HTTP-сервер, отвечающий сообщением «Hello World»

Чтобы реализовать простейший HTTP-сервер, наполним содержимым функцию обратного вызова запроса, упомянутую в предыдущем разделе.

Сначала вызовем метод res.write(), который записывает данные ответа в сокет, а затем использует метод res.end() для завершения ответа:

```
var http = require('http');
var server = http.createServer(function(req, res){
```

```
  res.write('Hello World');
  res.end();
});
```

Сокращенные формы записи методов res.write() и res.end() можно объединить в одну инструкцию, что может быть удобно для коротких ответов:

```
res.end('Hello World');
```

Теперь осталось выполнить привязку к порту, обеспечивающую возможность прослушивания входящих запросов. При этом используется метод server.listen(), принимающий комбинацию аргументов. С помощью этого метода прослушиваются соединения, установленные для выбранного порта. В процессе разработки обычно выполняется привязка к непривилегированному порту, например к порту 3000:

```
var http = require('http');
var server = http.createServer(function(req, res){
  res.end('Hello World');
});
server.listen(3000);
```

Теперь, когда Node прослушивает соединения с портом 3000, введите в адресной строке браузера адрес http://localhost: 3000. После этого вы должны получить страницу простого текста, содержащую слова «Hello World».

Создание HTTP-сервера — это только начало изучения методик программирования. В дальнейшем вам нужно узнать, как устанавливать коды состояния ответа и поля заголовка, соответствующим образом обрабатывать исключения и использовать API-интерфейсы, поддерживаемые Node. Но сначала давайте выясним, как формировать ответы на входящие запросы.

### 4.1.3. Чтение заголовков запроса и установка заголовков ответа

В примере «Hello World», рассмотренном в предыдущем разделе, демонстрируется минимум кода, которого достаточно для формирования подходящего HTTP-ответа. В этом примере используется заданный по умолчанию код состояния 200 (успешная операция), а также заданные по умолчанию заголовки ответа. Обычно же в отклик включается произвольное количество других HTTP-заголовков. Например, при отправке HTML-контента можно передать заголовок Content-Type вместе со значением text/html, что позволит браузеру вывести результат в формате HTML.

В Node доступно несколько методов, применяемых для поэтапного изменения полей заголовка HTTP-ответа: res.setHeader(field, value), res.getHeader(field) и res.removeHeader(field). Вот пример использования метода res.setHeader():

```
var body = 'Hello World';
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/plain');
res.end(body);
```

Добавлять и удалять заголовки можно в произвольном порядке, *но* только до первого вызова метода res.write() или res.end(). Завершив запись первой части ответа, Node очистит установленные HTTP-заголовки.

### 4.1.4. Установка кода состояния в HTTP-ответе

Обычно обратно передаются разные HTTP-коды состояния, которые отличаются от кода 200, заданного по умолчанию. Например, если запрашиваемого ресурса не существует, возвращается код состояния 404.

Чтобы задать другой код состояния, присвойте соответствующее значение свойству res.statusCode. Это можно сделать в любой момент времени, пока длится ответ приложения, то есть перед первым вызовом метода res.write() или res.end(). Это означает, что инструкцию res.statusCode = 302 можно поместить перед вызовами res.setHeader() или после них, как показано в следующем примере:

```
var url = 'http://google.com';
var body = '<p>Redirecting to <a href="' + url + '">'
        + url + '</a></p>';
res.setHeader('Location', url);
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/html');
res.statusCode = 302;
res.end(body);
```

Философия Node заключается в том, чтобы поддерживать небольшие по размеру, но надежные сетевые API-интерфейсы. Хотя такие API-интерфейсы не способны конкурировать с высокоуровневыми средами разработки, такими как Rails либо Django, они обеспечивают надежную платформу для построения собственных сред разработки. В силу этого ядро Node не поддерживает ни высокоуровневые концепции, такие как сеансы, ни базовые функциональные средства, такие как cookie-файлы. Все это отдается на откуп модулям независимых производителей.

Теперь, после знакомства с базовым API-интерфейсом HTTP, можно переходить к его применению. В следующем разделе с помощью этого API-интерфейса мы создадим простое приложение, совместимое с HTTP.

## 4.2. Создание веб-службы RESTful

Предположим, что с помощью Node нужно создать веб-службу, выполняющую типичные для списка запланированных дел действия: создание, чтение, обновление и удаление (Create, Read, Update, Delete, сокращенно CRUD). Эти действия могут быть реализованы многими способами, один из которых заключается в использовании веб-службы RESTful. Эта служба с помощью HTTP-методов, или глаголов, экспонирует компактный API-интерфейс REST.

Концепция REST (Representational State Transfer — передача репрезентативного состояния) была предложена в 2000 году Роем Филдингом (Roy Fielding)[9], одним из авторов спецификаций протоколов HTTP 1.0 и 1.1. В соответствии с этим соглашением HTTP-глаголы, такие как GET, POST, PUT и DELETE, проецируются для выборки, создания, обновления и удаления ресурсов, указанных с помощью URL-адреса. Веб-службы RESTful завоевали популярность благодаря простоте применения и реализации по сравнению с такими протоколами, как SOAP (Simple Object Access Protocol).

В рамках этого раздела взаимодействие с разрабатываемой веб-службой мы будем осуществлять с помощью веб-браузера cURL (http://curl.haxx.se/download.html). Веб-браузер cURL — это мощный HTTP-клиент, который работает в режиме командной строки и может использоваться для отправки запросов целевому серверу.

Для создания REST-совместимого сервера нужно реализовать четыре HTTP-глагола. Каждый глагол призван выполнять собственную операцию в списке запланированных дел:

- POST — добавление позиций в список;
- GET — вывод списка текущих позиций или подробностей, относящихся к заданной позиции;
- DELETE — удаление позиций из списка;
- PUT — изменение существующих позиций (в связи с ограниченным объемом книги этот метод подробно не рассматривается).

Конечный результат хорошо иллюстрирует следующий пример, в котором в списке запланированных дел с помощью команды curl создается новая позиция:

```
wavded@dev: ~
wavded@dev ~» curl -d 'buy node in action' http://localhost:3000
OK
```

А вот пример просмотра позиций в списке запланированных дел:

```
wavded@dev: ~
wavded@dev ~» curl http://localhost:3000
0) buy node in action
```

### 4.2.1. Создание ресурсов с помощью POST-запросов

В терминологии RESTful создание ресурса обычно проецируется на глагол POST. Соответственно в нашем приложении с помощью глагола POST мы создадим новый элемент в списке запланированных дел.

В Node проверка используемого HTTP-метода (глагола) осуществляется с помощью свойства req.method (листинг 4.1). Выяснив, какой именно метод используется в запросе, сервер будет знать, какую операцию выполнять.

Когда система синтаксического разбора HTTP-кода в Node считывает и разбирает данные запроса, эти данные становятся доступными в форме событий data, содержащих фрагменты разобранных данных, подготовленных к обработке программой:

```
var http = require('http')
var server = http.createServer(function(req, res){
  // События data вызываются после чтения нового фрагмента данных
  req.on('data', function(chunk){
    // По умолчанию фрагмент является объектом Buffer (байтовый массив)
    console.log('parsed', chunk);
  });
  // Событие end вызывается после считывания всех данных
  req.on('end', function(){
    console.log('done parsing');
    res.end()
  });
});
```

По умолчанию события data поддерживают объекты Buffer, представляющие собой Node-версию байтовых массивов. Если в списке запланированных дел используются текстовые элементы, двоичные данные не понадобятся. В этом

случае в качестве кодировки потока лучше всего выбрать вариант ascii или utf8. Это приведет к тому, что вместо событий data будут генерироваться строки. Чтобы изменить кодировку подобным образом, воспользуйтесь методом req.setEncoding(encoding):

```
// В качестве фрагмента вместо объекта Buffer используется utf8-строка
req.setEncoding('utf8')
req.on('data', function(chunk){
  console.log(chunk);
});
```

В случае с обработкой элемента списка запланированных дел в массив могут добавляться лишь целые строки. Один из способов получить целую строку — конкатенация всех фрагментов данных. Эта операция выполняется до появления события end, свидетельствующего о завершении запроса. После того как произойдет событие end, строка item списка дел полностью заполнится содержимым тела запроса, которое затем может быть помещено в массив списка дел. После добавления элемента в список дел можно завершить запрос строкой OK и заданным в Node по умолчанию кодом состояния 200. Эта методика проиллюстрирована в листинге 4.1 (файл todo.js).

**Листинг 4.1.** Буферизация строки тела POST-запроса

```
var http = require('http');
var url = require('url');
// В качестве хранилища данных используется обычный JavaScript-массив
// в памяти
var items = [];

var server = http.createServer(function(req, res){
  // req.method – это HTTP-метод запроса
  switch (req.method) {
    case 'POST':
    // Установка строкового буфера для входящей позиции
    var item = ' ';
    // Кодирование входящих событий data в виде строк в кодировке UTF-8
    req.setEncoding('utf8');
    req.on('data', function(chunk){
      // Конкатенация фрагмента данных в буфере
      item += chunk;
```

```
        });
        req.on('end', function(){
          // Помещение в массив items новой позиции
          items.push(item);
        res.end('OK\n');
        });
      break;
      }
    });
```

На рис. 4.3 представлен пример HTTP-сервера, обрабатывающего входящий HTTP-запрос и буферизующий вводимые данные перед завершением запроса.

**Рис. 4.3.** Конкатенация событий data в буфере тела запроса

Разработанное приложение теперь может добавлять позиции в список, но прежде, чем вы начнете использовать браузер cURL, следует закончить следующую операцию, чтобы получить список позиций.

### 4.2.2. Выборка ресурсов с помощью GET-запросов

Чтобы использовать глагол GET, добавьте его в ту же инструкцию switch, что и ранее, сопроводив кодом, выводящим позиции списка запланированных дел. В следующем примере сначала вызывается функция res.write(), которая сохраняет в заголовке содержимое полей, заданных по умолчанию, а также записывает переданные данные:

```
...
case 'GET':
items.forEach(function(item, i){
  res.write(i + ') ' + item + '\n');
});
res.end();
break;
...
```

После того как приложение получит возможность выводить позиции списка на экран, можно приступать к тестированию. Откройте окно терминала, запустите сервер и отправьте некоторые позиции методом POST, используя команду curl. В результате установки флага -d автоматически выбирается для запроса метод POST и значение передается в виде POST-данных:

```
$ curl -d 'buy groceries' http://localhost:3000
OK
$ curl -d 'buy node in action' http://localhost:3000
OK
```

После получения списка запланированных дел методом GET можно выполнить команду curl без флагов, поскольку глагол GET предлагается по умолчанию:

```
$ curl http://localhost:3000
0) buy groceries
1) buy node in action
```

### *Установка заголовка Content-Length*

Чтобы ускорить ответ, по возможности нужно отправлять вместе с ответом поле Content-Length. В случае с элементом списка ответ может быть предварительно сконструирован в памяти. В результате обеспечивается доступ к информации о длине строки, а также появляется возможность единовременной очистки всего списка. Установкой заголовка Content-Length неявно отключается фрагментарное кодирование в Node, что приводит к сокращению объема передаваемых данных с соответствующим увеличением производительности приложений.

Код оптимизированной версии обработчика метода GET может выглядеть следующим образом:

```
var body = items.map(function(item, i){
  return i + ') ' + item;
}).join('\n');
res.setHeader('Content-Length', Buffer.byteLength(body));
res.setHeader('Content-Type', 'text/plain; charset="utf-8"');
res.end(body);
```

На первый взгляд создается впечатление, что для заголовка Content-Length следует использовать значение body.length, но это не соответствует действительности, поскольку значение Content-Length должно представлять длину строки, выраженную в байтах, а не в символах. Эти два представления отличаются, если строка содержит символы, состоящие из нескольких байтов. Для решения этой проблемы воспользуйтесь методом Buffer.byteLength(), доступным в Node.

В следующем примере REPL-сеанса в Node иллюстрируется различие между вариантом с непосредственным использованием длины строки в качестве строки из пяти символов и семью байтами:

```
$ node
> 'etc …'.length
5
> Buffer.byteLength('etc …')
7
```

**repl в Node**

Node, как и многие другие языки программирования, предоставляет интерфейс REPL (read-eval-print-loop — цикл чтение-вычисление-печать), доступ к которому открывается при вводе в командной строке ключевого слова node без аргументов. В REPL-сеансе можно создавать фрагменты кода и получать результаты выполнения каждой инструкции. Этот интерфейс может применяться для изучения языка программирования, выполнения простых тестов и даже для отладки.

### 4.2.3. Удаление ресурсов с помощью DELETE-запросов

С помощью глагола DELETE элемент удаляется из списка запланированных дел. Чтобы выполнить эту операцию, приложение должно проверить запрошенный URL-

адрес, задающий элемент, который будет удаляться HTTP-клиентом. В рассматриваемом случае в качестве идентификатора используется индекс в массиве items, например DELETE /1 или DELETE /5.

Доступ к запрашиваемому URL-адресу обеспечивается с помощью свойства req.url, которое в зависимости от запроса может включать несколько компонентов. Например, если запрос записан в виде DELETE /1?api-key=foobar, это свойство будет включать название пути и строку запроса /1?api-key=foobar.

Для синтаксического разбора URL-адресов в Node используется модуль url, а также функция .parse(). Пример приведен в следующем REPL-сеансе в Node. Здесь выполняется разбор структуры URL-адреса в объект, включающий свойство pathname, которое вы будете использовать в обработчике DELETE:

```
$ node
> require('url').parse('http://localhost:3000/1?api-key=foobar')
{ protocol: 'http:',
  slashes: true,
  host: 'localhost:3000',
  port: '3000',
  hostname: 'localhost',
  href: 'http://localhost:3000/1?api-key=foobar',
  search: '?api-key=foobar',
  query: 'api-key=foobar',
  pathname: '/1',
  path: '/1?api-key=foobar' }
```

Функция url.parse() выполняет только синтаксический разбор пути, при этом идентификатор элемента остается строкой. Чтобы обрабатывать идентификатор в приложении, следует преобразовать его в число. Простой способ выполнения этой операции заключается в применении метода String#slice(), который возвращает часть строки, заключенную между двумя индексами. В данном случае этот метод может применяться для пропуска первого символа и отображения числовой части, все еще представляющей собой строку. Чтобы преобразовать строку в число, передайте ее глобальной JavaScript-функции parseInt(), которая возвращает числовое значение.

В листинге 4.2 продемонстрировано несколько приемов, применяемых для проверки значений, вводимых пользователями, а также для генерации ответов в ответ на запросы. Если значение является не числом ( JavaScript-значение NaN), код состояния получает значение 400, эквивалентное ошибке Bad Request (Неверный запрос). Затем код проверяет существование элемента и в случае его отсутствия

отображает сообщение об ошибке «404 Not Found». После завершения проверки вводимых значений элемент может быть удален из массива items, а приложение генерирует отклик 200, OK.

**Листинг 4.2.** Обработчик DELETE-запроса

```
...
// Добавление ветви DELETE в инструкцию switch
case 'DELETE':
  var path = url.parse(req.url).pathname;
  var i = parseInt(path.slice(1), 10);

  // Проверка корректности числа
  if (isNaN(i)) {
    res.statusCode = 400;
    res.end('Invalid item id');
  // Проверка существования запрошенного индекса
  } else if (!items[i]) {
    res.statusCode = 404;
    res.end('Item not found');
  } else {
    // Удаление запрошенного элемента
    items.splice(i, 1);
    res.end('OK\n');
  }
  break;
...
```

На первый взгляд кажется, что 15 строк кода, с помощью которых выполняется удаление элемента из массива, — это довольно много, но данное впечатление обманчиво. Если вы освоите работу с высокоуровневыми средами разработки, поддерживающими дополнительные API-интерфейсы, ваша задача станет намного проще. Освоение основ Node критически важно для понимания сути кода и принципов его отладки, а также позволит вам разрабатывать более мощные приложения и среды разработки.

В полноценной веб-службе RESTful также можно было бы реализовать HTTP глагол PUT, используемый для изменения существующего элемента в списке запланированных дел. Прежде чем перейти к следующему разделу, в котором рассматривается обслуживание статических файлов с помощью веб-приложения,

попробуйте реализовать этот последний обработчик самостоятельно с помощью методик, использованных при разработке REST-сервера.

## 4.3. Обслуживание статических файлов

Ко многим веб-приложениям выдвигаются схожие, если не идентичные, требования, и одно из них — обслуживание статических файлов, таких как CSS-файлы, JavaScript-код или графические файлы. Несмотря на то что создание надежного и эффективного статического файлового сервера относится к категории нетривиальных задач, к тому же сообществом Node-разработчиков предлагаются вполне надежные реализации файловых серверов, в этом разделе мы разработаем собственный статический файловый сервер, иллюстрирующий работу в Node низкоуровневого API-интерфейса файловой системы.

В этом разделе вы узнаете:

- как создать простой статический файловый сервер;
- как оптимизировать передачу данных с помощью метода pipe();
- как обрабатывать ошибки пользователя и файловой системы, устанавливая коды состояния.

А начнем мы с создания простейшего HTTP-сервера, предназначенного для обслуживания статических ресурсов.

### 4.3.1. Создание статического файлового сервера

Первыми и чаще всего используемыми файловыми серверами были традиционные HTTP-серверы, такие как Apache и IIS. Вполне возможно, один из этих серверо запущен на каком-нибудь старом веб-сайте, с которым вам иногда приходится работать. Попробуйте переместить этот сервер вместе с базовой функциональностью на платформу Node. Это превосходное упражнение, которое поможет вам лучше понять принципы работы старых добрых HTTP-серверов.

Каждый статический файловый сервер имеет папку root, которая является основной папкой, включающей обслуживаемые файлы. Для создаваемого сервера определяется переменная root, представляющая корневую папку статического файлового сервера:

```
var http = require('http');
var parse = require('url').parse;
```

```
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

...
```

*Магической* переменной __dirname, поддерживаемой в Node, присваивается путь к папке, в которой находится файл. Переменная называется «магической», поскольку может приманить различные значения в ходе выполнения одной программы, если файлы находятся в разных папках. В данном случае сервер будет обслуживать статические файлы, находящиеся в той же папке, что и сценарий, но переменной root можно присвоить путь и к другой папке.

Следующим шагом нужно получить доступ к переменной pathname модуля url, чтобы определить путь к запрашиваемому файлу. Если переменная pathname принимает значение /index.html и в качестве папки файла, заданной в переменной root, используется /var/www/example.com/public, можно просто объединить названия папок с помощью метода path.join() модуля. В результате будет создан абсолютный путь /var/www/example.com/public/index.html. Следующий код демонстрирует, каким образом это все работает:

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
});

server.listen(3000);
```

**атака обходом папок**

Файловый сервер, созданный в этом разделе, представляет собой довольно простой объект. Если вы собираетесь использовать файловый сервер на практике, более

тщательно проверяйте вводимые данные, чтобы предотвратить доступ пользователей к разделам файловой системы, получаемый с помощью атаки обходом папок. Подробнее этот тип атаки описан в Википедии (http://en.wikipedia.org/wiki/Directory_traversal_attack).

После завершения конструирования пути нужно передать содержимое файла. Для выполнения этой операции требуется высокоуровневый потоковый доступ к диску, и выполняемый с помощью одного из потоковых Node-классов fs.ReadStream. Этот класс генерирует события data в процессе последовательного чтения файла с диска. В листинге 4.3 представлен простой, но в то же время вполне функциональный файловый сервер.

**Листинг 4.3.** Простейший статический файловый сервер ReadStream

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

var server = http.createServer(function(req, res){
  var url = parse(req.url);
  // Конструирование абсолютного пути
  var path = join(root, url.pathname);
  // Создание класса fs.ReadStream
  var stream = fs.createReadStream(path);
  // Запись данных файла для ответа
  stream.on('data', function(chunk){
    res.write(chunk);
  });
  stream.on('end', function(){
    // Завершение ответа после закрытия файла
    res.end();
  });
});
```

```
server.listen(3000);
```

Этот файловый сервер в большинстве случаев вполне функционален, но при работе с ним следует учитывать некоторые нюансы. Далее вы научитесь оптимизировать передачу данных, сократив при этом объем кода сервера.

### *Оптимизация передачи данных с помощью метода Stream#pipe()*

При работе с Node следует знать принципы, на основе которых функционирует метод fs.ReadStream, а также иметь представление о гибкости, обеспечиваемой событиями. Кроме того, в Node поддерживается альтернативный высокоуровневый механизм, реализуемый методом Stream#pipe(). Этот метод позволяет серьезно упростить код сервера:

```
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  // При вызове stream.pipe() метод res.end() вызывается внутренне
  stream.pipe(res);
});
```

**каналы передачи данных и водопроводные трубы**

Чтобы лучше понять принцип работы каналов передачи данных в Node, представьте себе аналогию с водопроводными трубами. Чтобы вода поступала из какого-либо источника, например из водонагревателя например, в кухонный кран, нужно с помощью трубы связать источник воды с ее потребителем.

Аналогичная концепция применима к каналам передачи данных в Node, только здесь вместо воды вы имеете дело с данными, поступающими по «трубе» из источника (ReadableStream) потребителю (WritableStream). Соединение источника с потребителем осуществляется методом pipe:

```
ReadableStream#pipe(WritableStream);
```

В качестве примера использования каналов можно рассмотреть чтение файла (ReadableStream) и запись содержимого этого файла в другой файл (WritableStream):

```
var readStream = fs.createReadStream('./original.txt')
var writeStream = fs.createWriteStream('./copy.txt')
readStream#pipe(writeStream);
```

Произвольный поток чтения (ReadableStream) может быть направлен по каналу в любой поток записи (WritableStream). Например, объект HTTPrequest (req) является потоком чтения, и его содержимое может быть направлено в файл:

```
req.pipe(fs.createWriteStream('./req-body.txt'))
```

Чтобы глубже изучить концепцию потоков данных в Node, включая список доступных встроенных потоков данных, прочитайте книгу о потоках данных, опубликованную на веб-сайте GitHub по адресу https://github.com/substack/streamhandbook.

На рис. 4.4 показан HTTP-сервер, который считывает статический файл из файловой системы, а затем по каналу методом pipe() направляет результат HTTP-клиенту.

Теперь можно приступать к тестированию статического файлового сервера, выполнив команду curl. Флаг -i или —include заставляет браузер cURL вывести заголовок ответа:

```
$ curl http://localhost:3000/static.js –i
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
...
```

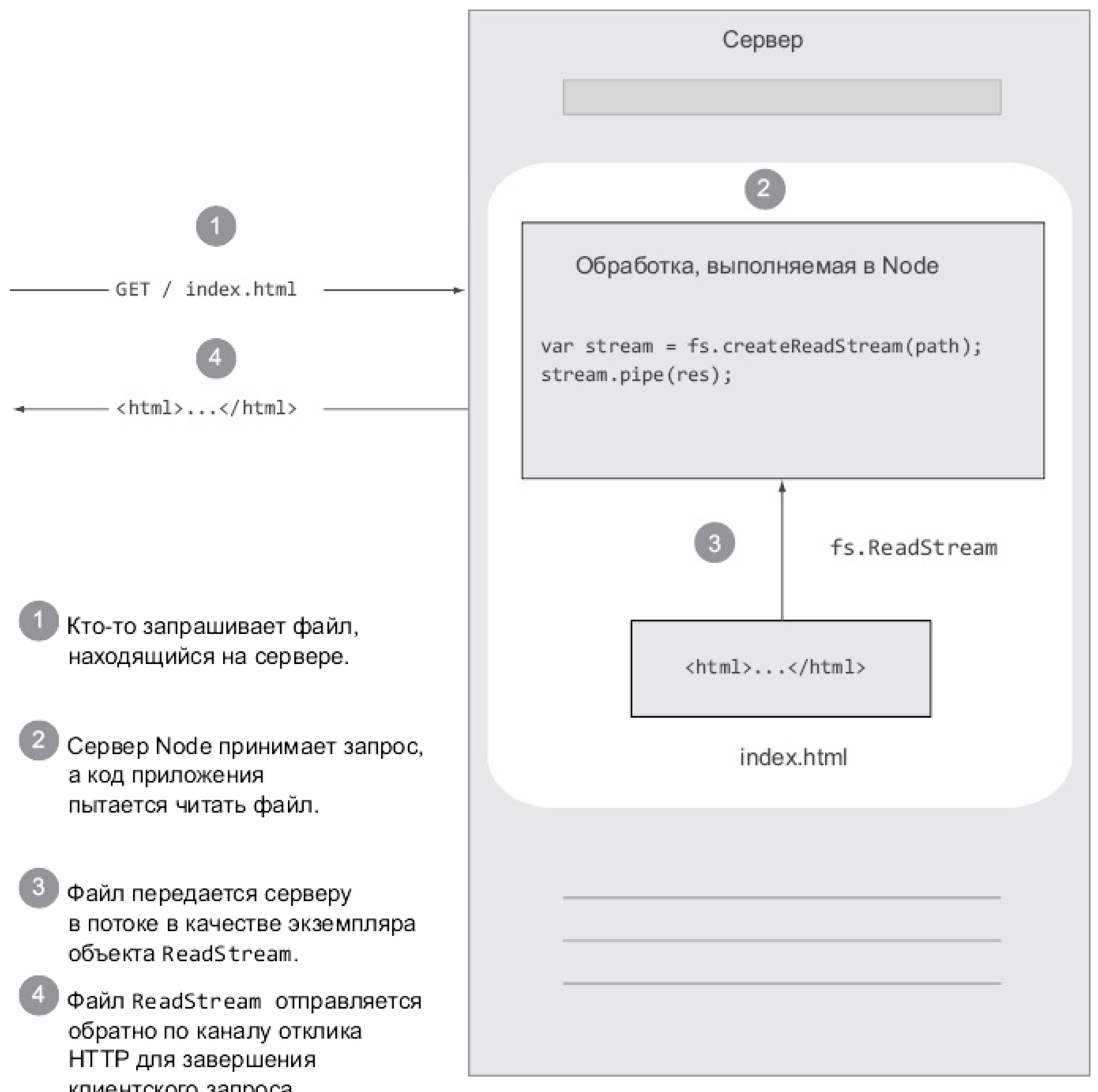



**Рис. 4.4.** HTTP-сервер в Node, обслуживающий с помощью метода fs.ReadStream статический файл в файловой системе

Как упоминалось ранее, в папке root находится сценарий статического файлового сервера, то есть предыдущая команда curl запрашивает сценарий сервера, который отсылается обратно в качестве тела отклика.

Создание статического файлового сервера на этом еще не завершается, к тому же такой сервер подвержен ошибкам. Всего одно необработанное исключение, такое как запрос пользователем несуществующего файла, приведет к завершению работы всего сервера. В следующем разделе рассказывается об обработке ошибок, возникающих при работе файлового сервера.

### 4.3.2. Обработка ошибок сервера

Создаваемый в этом разделе статический файловый сервер пока не способен обрабатывать ошибки, которые могут возникать в процессе использования класса fs.ReadStream. Ошибки возможны, если пытаться получить доступ к несуществующему или запрещенному файлу либо из-за других проблем, связанных с вводом-выводом данных файла. В этом подразделе мы выясним, как сделать более надежным файловый или любой другой Node-сервер.

В Node все, что наследуется от класса EventEmitter, может привести к генерированию события error. Поток данных, например fs.ReadStream, представляет собой специализированный класс генератора событий EventEmitter, который содержит предопределенные события, такие как упомянутые ранее события data и end. По умолчанию события error будут генерироваться в случае отсутствия слушателей. Это означает, что если не прослушиваются ошибки, они могут привести к отказу сервера.

Чтобы проиллюстрировать излагаемые концепции, попробуйте запросить несуществующий фал, такой как /notfound.js. В терминальном сеансе, в котором выполняется сервер, можно просмотреть трассу стека исключений, выводимого в поток stderr, как показано в следующем примере кода:

```
stream.js:99
throw arguments[1]; // Необработанное событие 'error'
^
Error: ENOENT, No such file or directory
  '/Users/tj/projects/node-in-action/source/notfound.js'
```

Чтобы предотвратить ошибки отказа сервера, нужно прослушивать сервер, зарегистрировав обработчик события error в fs.ReadStream, как в следующем фрагменте кода, который отвечает кодом состояния 500. Этот код означает, что произошла внутренняя ошибка сервера:

```
...
stream.pipe(res);
stream.on('error', function(err){
  res.statusCode = 500;
  res.end('Internal Server Error');
});
...
```

Регистрация обработчика события error помогает перехватывать любые предвиденные и непредвиденные ошибки, а также позволяет более элегантно отвечать клиенту.

### 4.3.3. Вытесняющая обработка ошибок с использованием файла fs.stat

Передаваемые файлы являются статическими, поэтому с помощью системного вызова stat() можно запрашивать информацию о файлах, такую как время изменения, размер в байтах и прочие сведения. Эта информация особенно ценна при поддержке условного метода GET, с помощью которого браузер может сгенерировать запрос на проверку актуальности кэша.

Код переработанного файлового сервера, показанный в листинге 4.4, вызывает метод fs.stat() и извлекает информацию о файле, такую как размер и код ошибки. Если именованного файла не существует, метод fs.stat() отвечает значением ENOENT в поле err.code. При этом может возвращаться код ошибки 404 свидетельствующей о том, что файл не найдет. При получении от метода fs.stat() других ошибок можно возвратить обобщенный код ошибки 500.

**Листинг 4.4.** Проверка существования файла и ответ с помощью поля Content-Length

```
var server = http.createServer(function(req, res){
  // Синтаксический разбор URL-адреса для получения пути
  var url = parse(req.url);
  // Конструирование абсолютного пути
  var path = join(root, url.pathname);
  // Проверка существования файла
  fs.stat(path, function(err, stat){
    if (err) {
      // Файл не существует
      if ('ENOENT' == err.code) {
        res.statusCode = 404;
        res.end('Not Found');
      // Некая другая ошибка
      } else {
        res.statusCode = 500;
        res.end('Internal Server Error');
      }
    } else {
    // Задание поля Content-Length с помощью объекта stat
    res.setHeader('Content-Length', stat.size);
    var stream = fs.createReadStream(path);
    stream.pipe(res);
```

```
        stream.on('error', function(err){
          res.statusCode = 500;
          res.end('Internal Server Error');
          });
        }
      });
    });
```

Теперь, когда мы познакомились с низкоуровневым обслуживанием файлов в Node, давайте рассмотрим еще одну обычную и, вероятно, даже более важную для веб-разработки функциональную возможность — получение данных, вводимых пользователями в HTML-формы.

## 4.4. Получение данных, вводимых в формы

В веб-приложениях пользователи обычно вводят данные с помощью форм. В Node поддерживается базовая функциональность, а реализация рабочих операций (таких, как проверка вводимых данных или загрузка файлов) возлагается на пользователей. На самом деле это не так уж плохо, поскольку стимулирует создание сред разработки независимыми производителями, предлагающими простой и эффективный низкоуровневый API-интерфейс.

В этом разделе вы узнаете:

- как обрабатывать введенные в форму данные;
- как обрабатывать выгружаемые файлы с помощью модуля formidable;
- как показывать ход загрузки файлов в режиме реального времени.

И начнем мы с вопросов обработки введенных в форму данных средствами Node.

### 4.4.1. Обработка заполненных полей форм

Обычно с запросами, выполняющими обработку данных форм, связаны следующие два значения Content-Type:

- application/x-www-form-urlencoded — значение, выбранное по умолчанию для HTML-форм;

- multipart/form-data — используется в случае, если форма содержит файлы, двоичные данные или текст, кодировка которого отличается от ASCII.

В этом разделе мы переработаем приложение со списком запланированных дел, которое было создано в предыдущем разделе, добавив в него средства использования формы и веб-браузера. После завершения в вашем распоряжении будет веб-приложение, поддерживающее список запланированных дел (рис. 4.5).

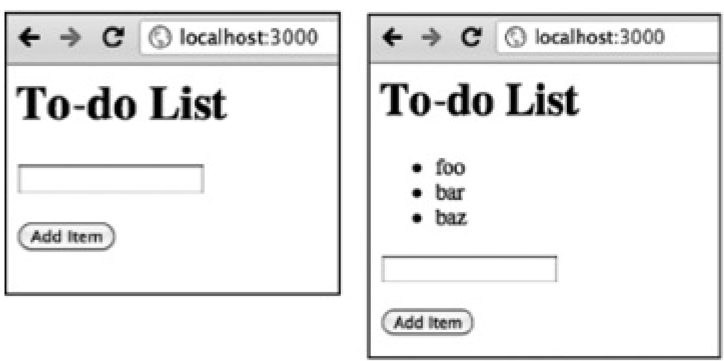



**Рис. 4.5.** Приложение со списком запланированных дел, использующее HTML-форму и веб-браузер. Экранный снимок слева иллюстрирует состояние приложения сразу же после загрузки, справа — после добавления в список нескольких позиций

Инструкция switch в методе req.method этого приложения служит для организации простого ветвления запросов. Применение этой инструкции продемонстрировано в листинге 4.5. Любой URL-адрес, не заданный *в точности* как "/", считается ответом 404 Not Found. Любой HTTP-глагол, если это не GET ил POST, расценивается как ответ 400 Bad Request. Имеющиеся в листинге функции-обработчики show(), add(), badRequest() и notFound() мы реализуем в этом разделе позже.

**Листинг 4.5.** HTTP-сервер, поддерживающий методы GET и POST

```
var http = require('http');
var items = [];
var server = http.createServer(function(req, res){
  if ('/' == req.url) {
    switch (req.method) {
      case 'GET':
        show(res);
        break;
      case 'POST':
        add(req, res);
        break;
```

```
      default:
        badRequest(res);
    }
  } else {
    notFound(res);
  }
});

server.listen(3000);
```

Хотя разметка обычно генерируется с помощью шаблонизаторов, в примере, приведенном в листинге 4.6, для простоты применяется конкатенация строк. При этом не нужно присваивать значение переменной res.statusCode, поскольку оно по умолчанию равно 200 OK. Результирующая веб-страница в окне браузера показана на рис. 4.5.

**Листинг 4.6.** Форма ввода данных и список элементов

```
function show(res) {
  var html = '<html><head><title>Todo List</title></head><body>'
        // Для наглядности в приложении вместо использования
        // шаблонизатора HTML-код просто подставляется
         + '<h1>Todo List</h1>'
         + '<ul>'
         + items.map(function(item){
           return '<li>' + item + '</li>'
           }).join('')
         + '</ul>'
         + '<form method="post" action="/">'
         + '<p><input type="text" name="item" /></p>'
         + '<p><input type="submit" value="Add Item" /></p>'
         + '</form></body></html>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}
```

Функция notFound() принимает объект ответа, присваивает коду состояния значение 404, а телу объекта ответа — строку 'Not Found':

```
function notFound(res) {
  res.statusCode = 404;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Not Found');
}
```

Реализация отклика 400 Bad Request практически идентична функции notFound() и указывает клиенту на то, что запрос некорректен:

```
function badRequest(res) {
  res.statusCode = 400;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Bad Request');
}
```

И напоследок в приложении нужно реализовать функцию add(), которая принимает объекты req и res. Код этой функции:

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

function add(req, res) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function(){
    var obj = qs.parse(body);
    items.push(obj.item);
    show(res);
  });
}
```

Для простоты в этом примере предполагается, что переменной Content-Type присвоено значение application/x-wwwform-urlencoded, что является заданным по умолчанию значением для HTML-форм. Чтобы выполнить синтаксический разбор этих данных, нужно просто конкатенировать фрагменты событий data, сформировав тем самым завершенную строку тела. Поскольку двоичные данные не обрабатываются, с помощью метода res.setEncoding() можно выбрать для кодировки запроса вариант utf8. Когда запрос генерирует событие end, все события data завершаются, а переменной body присваивается все тело ответа в виде строки.

#### Буферизация больших объемов данных

Буферизация хорошо работает, если тело запросов невелико и содержит не много JSON- или XML-кода, однако тут возможны проблемы. Если размер буфера должным образом не ограничен максимальным значением (см. главу 7), буферизация чревата возможной уязвимостью приложения. Чтобы минимизировать риск уязвимости, следует реализовать потоковый анализатор, сократить требования к памяти и снизить вероятность нехватки ресурсов. При этом выполняется синтаксический разбор фрагментов событий data по мере их генерирования, хотя это сложнее в реализации и применении.

### *Модуль querystring*

В серверной реализации функции add() для синтаксического разбора тела запроса используется Node-модуль querystring. Рассмотрим быстрый REPL-сеанс демонстрирующий принцип работы Node-функции querystring.parse(). Эта функция используется на сервере.

Предположим, что пользователь ввел в HTML-форму текст «take ferrets to the vet», предназначенный для включения в список запланированных дел:

```
$ node
> var qs = require('querystring');
> var body = 'item=take+ferrets+to+the+vet';
> qs.parse(body);
{ item: 'take ferrets to the vet' }
```

После добавления указанной позиции сервер возвращает пользователю обратно исходную форму путем вызова той же самой функции show(), которая была предварительно реализована. Подобное поведение характерно лишь для данного примера, в других случаях может выводиться какое-нибудь сообщение, например Added todo list item («Элемент добавлен в список дел»), или пользователь может перенаправляться обратно в папку /.

Попытайтесь протестировать этот пример. Добавьте в список несколько новых позиций, и вы обнаружите, что список запланированных дел не упорядочен. Можно также реализовать функцию удаления элементов, как мы уже делали ранее с помощью API-интерфейса REST.

### 4.4.2. Обработка выгружаемых файлов с помощью модуля formidable

Обработка операций по выгрузке файлов — еще один часто реализуемый и важный аспект разработки веб-приложений. Предположим, что нужно создать приложение, которое выполняет выгрузку коллекции фотографий на сайт и обеспечивает общий доступ к этим фотографиями после щелчка на соответствующей веб-ссылке. Выгрузку файлов можно выполнять с помощью HTML-формы, показываемой в окне браузера.

В следующем примере кода представлена форма, которая выгружает файл с помощью ассоциированного поля name:

```
<form method="post" action="/" enctype="multipart/form-data">
<p><input type="text" name="name" /></p>
<p><input type="file" name="file" /></p>
<p><input type="submit" value="Upload" /></p>
</form>
```

Чтобы корректно выполнить выгрузку файла и принять его содержимое, следует присвоить атрибуту enctype значение multipart/form-data, MIME-тип которого подходит для больших двоичных объектов (Binary Large OBjects, BLOB).

Быстрый синтаксический разбор запросов, состоящих из нескольких частей, в потоковом режиме представляет собой нетривиальную задачу, которая в этой книге не рассматривается. Она может быть решена с помощью нескольких модулей, созданных сообществом разработчиков Node-приложений. Один из подобных модулей, formidable, был создан Феликсом Гейзендорфером (Felix Geisendorfer) и предназначался для службы выгрузки файлов и кодирования видео (Transloadit). Этот модуль характеризуется высокой производительностью и надежностью.

Модуль formidable — это потоковая система синтаксического разбора, на вход которой поступают фрагменты получаемых данных, а на выходе генерируются заданные части, например упомянутые ранее заголовки ответов и их тела. Такой подход обеспечивает быстродействие, а также предотвращает чрезмерное расходование памяти, поскольку не требует буферизации. Все это позволяет не допустить краха процесса даже в случае очень больших файлов, например видеофайлов.

А теперь вернемся к рассмотрению нашего примера со службой обеспечения общего доступа к фотографиям. HTTP-сервер, код которого приведен в листинге 4.7, реализует базовую серверную функциональность по выгрузке файлов. HTTP-сервер в ответ на метод GET выводит HTML-форму. Также в код сервера включена пустая функция для метода POST; в нее в дальнейшем будет интегрирован модуль formidable, обрабатывающий выгрузку файлов.

**Листинг 4.7.** Подготовка HTTP-сервера к получению выгружаемых файлов

```
var http = require('http');
var server = http.createServer(function(req, res){
  switch (req.method) {
    case 'GET':
      show(req, res);
      break;
    case 'POST':
      upload(req, res);
      break;
  }
});

// Обслуживание HTML-формы, используемой для ввода файла
function show(req, res) {
  var html = ''
    + '<form method="post" action="/" enctype="multipart/form-data">'
    + '<p><input type="text" name="name" /></p>'
    + '<p><input type="file" name="file" /></p>'
    + '<p><input type="submit" value="Upload" /></p>'
    + '</form>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}

function upload(req, res) {
  // Логика выгрузки
}
```

Теперь, когда мы позаботились о GET-запросе, нужно реализовать функцию upload(), которая вызывается посредством обратного вызова запроса, когда приходит POST-запрос. Функция upload() должна принимать входящие выгружаемые данные при наличии модуля formidable. В оставшейся части раздела мы узнаем, как интегрировать модуль formidable в наше веб-приложение. Для этого требуется:

1. С помощью диспетчера Node-пакетов установить модуль formidable.

2. Создать экземпляр класса IncomingForm.

3. Вызвать функцию form.parse() для объекта HTTP-запроса.

4. Прослушивать события формы field, file и end.

5. Использовать высокоуровневый API-интерфейс модуля formidable.

Чтобы начать использовать модуль formidable в проекте, сначала нужно его установить. Для этого выполните следующую команду, которая позволяет установить модуль локально в папку ./node_modules:

```
$ npm install formidable
```

Чтобы получить доступ к API, как и в случае модуля http, следует воспользоваться функцией require():

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

Первым шагом на пути реализации функции upload() является ответ 400 Bad Request, генерируемый в том случае, если в запросе нет подходящего типа содержимого:

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request: expecting multipart/form-data');
    return;
  }
}


function isFormData(req) {
  var type = req.headers['content-type'] || '';
  return 0 == type.indexOf('multipart/form-data');
}
```

Вспомогательная функция isFormData() проверяет поле заголовка Content-Type, чтобы идентифицировать значение multipart/form-data. При этом используется JavaScript-метод String.indexOf(), который декларирует, что величина multipart/form-data находится в начале значения поля.

Теперь, когда мы познакомились с запросами, состоящими из несколько частей, нужно инициализировать новую форму formidable.IncomingForm и сгенерировать

вызов метода form.parse(req), в котором ключевое слово req обозначает объект запроса. В результате будет обеспечен доступ модуля formidable к событиям запроса data для синтаксического разбора:

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request');
    return;
  }
  var form = new formidable.IncomingForm();
  form.parse(req);
}
```

Объект IncomingForm сам по себе генерирует множество событий и по умолчанию направляет выгружаемые файлы в папку /tmp. Как показано в листинге 4.8, модуль formidable генерирует события после обработки элементов формы. Например, событие file генерируется после получения и обработки файла, а событие field — после завершения получения данных поля.

**Листинг 4.8.** Использование API-интерфейса модуля formidable

```
...
var form = new formidable.IncomingForm();

form.on('field', function(field, value){
  console.log(field);
  console.log(value);
});

form.on('file', function(name, file){
  console.log(name);
  console.log(file);
});

form.on('end', function(){
  res.end('upload complete!');
});
```

```
form.parse(req);
...
```

Проверив первые два вызова console.log() в обработчике событий field, вы можете увидеть, что в текстовое поле name введено значение my clock:

```
name
my clock
```

Событие file генерируется после завершения выгрузки файла. Объект file поддерживает размер файла, путь к папке form.uploadDir (по умолчанию — /tmp), исходное базовое имя и MIME-тип. Объект file, переданный методу console.log(), выглядит следующим образом:

```
{ size: 28638,
  path: '/tmp/d870ede4d01507a68427a3364204cdf3',
  name: 'clock.png',
  type: 'image/png',
  lastModifiedDate: Sun, 05 Jun 2011 02:32:10 GMT,
  length: [Getter],
  filename: [Getter],
  mime: [Getter],
...
}
```

Модуль formidable предоставляет также высокоуровневый API-интерфейс, фактически являясь, как уже отмечалось, оболочкой для API в единственном обратном вызове. Если методу form.parse() передается функция, в качестве первого аргумента используется объект error (при наличии ошибки). Если же ошибки отсутствуют, передаются два объекта: fields и files.

Содержимое объекта fields выводится с помощью метода console.log():

```
{ name: 'my clock' }
```

Объект files поддерживает экземпляры объекта File, которые генерируют событие file. При этом используется такое имя, как fields.

Обратите внимание, что можно прослушивать эти события даже при использовании обратного вызова, поэтому такие функции, как функция вывода сведений о ходе процесса, не мешают. В следующем примере кода иллюстрируется применение более краткой формы API для получения аналогичных результатов:

```
var form = new formidable.IncomingForm();
form.parse(req, function(err, fields, files){
  console.log(fields);
```

```
  console.log(files);
  res.end('upload complete!');
});
```

Теперь, зная основы, можно заняться вычислениями, позволяющими отражать ход выгрузки файлов — вполне естественный процесс для Node и цикла событий этой платформы.

### 4.4.3. Отражение хода выгрузки файлов в режиме реального времени

Событие progress модуля formidable генерирует значения, соответствующие количеству полученных и ожидаемых байтов. На основе этих значений можно реализовать индикатор процесса, иллюстрирующий ход выгрузки файлов. В следующем примере вычисляется процентное соотношение, которое выводится на консоль путем вызова метода console.log() при каждом генерировании события progress:

```
form.on('progress', function(bytesReceived, bytesExpected){
  var percent = Math.floor(bytesReceived / bytesExpected * 100);
  console.log(percent);
});
```

В результате выполнения этого сценария будет получен следующий результат:

```
1
2
4
5
6
8
...
99
100
```

Если вы изучите изложенные в этом разделе концепции, то сможете выполнить следующий очевидный шаг, который заключается в реализации индикатора процесса для пользовательского браузера. Это весьма полезное средство для любого приложения, в котором ожидается выгрузка больших объемов данных, причем Node прекрасно подходит для решения этой задачи. Например, с помощью протокола Web-Socket или с помощью модуля, выполняющегося в режиме реального времени, например Socket.IO, реализация индикатора процесса потребует лишь нескольких строк кода. Попробуйте в качестве упражнения

написать подобный код самостоятельно.

В последнем разделе этой главы рассматривается очень важная тема — защита приложения.

### 4.5. Защита приложения с помощью протокола HTTPS

Одно из требований, выдвигаемых к веб-сайтам электронной коммерции и сайтам, на которых хранятся важные данные, заключается в защите трафика, передаваемого между сервером и клиентом. В ходе стандартного HTTP-сеанса обмен информацией между сервером и клиентом происходит с помощью незакодированного текста, поэтому HTTP-трафик может легко перехватываться злоумышленниками.

Чтобы обеспечить безопасность веб-сеансов, можно воспользоваться протоколом HTTPS (Hypertext Transfer Protocol Secure — протокол защищенной передачи гипертекста). Протокол HTTPS представляет собой комбинацию протокола HTTP с транспортным уровнем TLS/SSL. Данные, пересылаемые по протоколу HTTPS, шифруются, что затрудняет их перехват злоумышленниками. В этом разделе рассматриваются основные сведения, касающиеся обеспечения безопасности приложений с помощью протокола HTTPS.

Если вы хотите воспользоваться преимуществами протокола HTTPS при разработке Node-приложений, начните с получения закрытого ключа и сертификата. Фактически закрытый ключ представляет собой «пароль», требуемый для расшифровки данных, передаваемых между сервером и клиентом. Закрытый ключ находится в файле, хранящемся в той части сервера, которая труднодоступна для не заслуживающих доверия пользователей. В этом разделе мы сгенерируем так называемый *самозаверенный сертификат*. Эта разновидность SSL-сертификатов не применяется на корпоративных веб-сайтах, поскольку в случае попытки получения доступа к странице с ненадежным сертификатом браузер выводит предупреждающее сообщение. Область применения подобных сертификатов — разработка веб-приложений и тестирование зашифрованного трафика.

Чтобы сгенерировать закрытый ключ, требуется пакет OpenSSL, который автоматически устанавливается в системе после установки Node. Чтобы сгенерировать закрытый ключ под названием key.pem, откройте окно командной строки и введите следующую команду:

```
openssl genrsa 1024 > key.pem
```

Помимо закрытого ключа нужен сертификат. В отличие от закрытого ключа, сертификат можно открыть всему миру; он содержит открытый ключ и сведения о владельце сертификата. С помощью открытого ключа шифруется трафик, направляемый от клиента серверу.

Закрытый ключ применяется для создания сертификатов. Чтобы сгенерировать сертификат key-cert.pem, введите следующую команду:

```
openssl req -x509 -new -key key.pem > key-cert.pem
```

Сгенерированные ключи следует хранить в безопасном месте. В рассматриваемом примере HTTPS-сервера, код которого приведен в листинге 4.9 ключи хранятся в той же папке, что и сценарий сервера. Обычно же ключи хранятся в другом месте, как правило, в папке ~/.ssh. Следующий код создает простой HTTPS-сервер, использующий сгенерированные ранее ключи.

**Листинг 4.9.** Простой HTTPS-сервер

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

var options = {
  // SSL-ключ и сертификат задаются объектом options
  key: fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./key-cert.pem')
};

// Объект options передается первым
https.createServer(options, function (req, res) {
  // Модули https и http имеют практически идентичные API-интерфейсы
  res.writeHead(200);
  res.end("hello world\n");}).listen(3000);
```

После запуска HTTPS-сервера можно подключиться к нему в безопасном режиме с помощью веб-браузера. Чтобы установить соединение, в адресной строке браузера введите строку https://localhost:3000/. Поскольку в примере HTTPS-сервера из листинга 4.9 не используется сертификат, заверенный подписью сертифицирующего органа, выводится соответствующее предупреждение. Можно проигнорировать это предупреждение, но если вы собираетесь разворачивать публичный веб-сайт, зарегистрируйтесь в сертифицирующем органе и получите реальный доверенный сертификат для использования на сервере.

## 4.6. Резюме

В этой главе, посвященной основам работы HTTP-сервера в Node, мы узнали, как отвечать на входящие запросы и как обрабатывать асинхронные исключения, что требуется для повышения надежности приложений. Также мы создали веб-

приложение RESTful, научились обслуживать статические файлы и даже разработали своеобразный вычислитель, рассчитывающий ход выгрузки файлов.

Возможно, вам, как разработчику веб-приложений, этот начальный этап изучения Node показался пугающим. Однако опытные веб-разработчики могут вам подтвердить, что вложенный труд не пропадет даром. Полученные знания помогут разобраться в том, как в Node выполняется отладка, создаются среды разработки с открытым кодом или модернизируются существующие среды разработки.

Излагаемые в этой главе фундаментальные концепции подготовили вас к освоению высокоуровневой среды разработки Connect, фантастическая функциональность которой может использоваться другими средами разработки веб-приложений. Позже мы познакомимся с Express — еще одним инструментом разработки веб-приложений в Node. Благодаря этим инструментам сведения, полученные в этой главе, станут понятнее, а создаваемые с их помощью веб-приложения будут более защищенными и полезными.

Но прежде чем двигаться дальше, следует подробнее узнать о том, как хранятся данные приложений. В следующей главе рассматриваются клиенты баз данных, созданные сообществом Node-разработчиков. Эти клиенты пригодятся нам в приложениях, которые мы будем разрабатывать в оставшейся части книги.

## Глава 5. Хранение данных Node-приложениями

- Хранилища данных в оперативной памяти и файловой системе
- Традиционные хранилища в виде реляционных баз данных
- Хранилища в виде нереляционных баз данных

Практически каждому приложению, будь то веб-приложение или приложение другого типа, требуется то или иное хранилище данных, и разрабатываемые в Node приложения не исключение. Выбор подходящего механизма хранения данных зависит от пяти факторов:

- какие данные хранятся;
- насколько быстро данные нужно считывать и записывать для достижения приемлемой производительности;
- каков объем имеющихся данных;

- как данные должны запрашиваться;
- каковы требования к надежности и сроку хранения данных.

Методы хранения данных варьируются, начиная от сохранения данных в памяти сервера и заканчивая использованием полномасштабной системы управления базами данных (СУБД). Независимо от выбранного метода всем им присущи определенные недостатки.

Механизмы, обеспечивающие долговременное хранение сложных структурированных данных и оснащенные мощными средствами поиска, весьма дорогостоящи, поэтому их использование не всегда оправданно. В то же время хранение данных в памяти сервера оптимально в плане производительности, но недостаточно надежно, поскольку данные могут быть потеряны в результате перезапуска сервера или отключения сервера от электросети.

Какой же механизм хранения данных следует выбрать? При разработке приложений в Node зачастую используется несколько механизмов хранения данных для достижения разных целей. В этой главе рассмотрены три варианта хранения данных:

- хранение данных без установки и конфигурирования СУБД;
- хранение данных с помощью реляционных СУБД, в частности MySQL и PostgreSQL;
- хранение данных с помощью баз данных, не поддерживающих SQL, в частности Redis, MongoDB и Mongoose.

Некоторые из перечисленных механизмов мы задействуем для создания приложений позже, а дочитав эту главу до конца, вы получите представление об использовании различных механизмов хранения данных при разработке собственных приложений.

А для начала давайте рассмотрим простейший и самый низший из доступных уровней хранения данных — бессерверное хранилище данных.

### 5.1. Бессерверное хранилище данных

С точки зрения системного администратора самыми удобными являются хранилища данных, не требующие для своего обслуживания СУБД. К таковым относятся хранилища данных в памяти и хранилища данных на базе файлов.

Установка приложения значительно упрощается, если не требует установки и конфигурирования СУБД.

Бессерверное хранилище данных идеально подходит для Node-приложений, которые пользователи запускают на собственных компьютерах, таких как веб-приложения и другие TCP/IP-приложения. Эта разновидность хранилищ данных также подходит для инструментов с интерфейсом командной строки (Command-Line Interface, CLI). При использовании управляемых Node инструментов командной строки хранилище данных может потребоваться, но вряд ли вы захотите устанавливать для этого собственный MySQL-сервер.

В этом разделе вы узнаете о том, когда и где используются эти две основные разновидности бессерверных хранилищ — в памяти и на базе файлов. И начнем мы с рассмотрения простейшего хранилища — хранилища данных в памяти.

### 5.1.1. Хранилище данных в памяти

В примерах приложений, рассмотренных в главах 2 и 4, хранилище данных, находящееся в памяти, применяется для отслеживания подробных сведений о пользователях чата и выполняемых операциях. В таком хранилище для хранения данных служат переменные. При этом чтение и запись данных осуществляются быстро, но как уже упоминалось, существует риск потери данных в случае перезапуска сервера или приложения.

Хранилища данных в памяти лучше всего использовать для хранения небольших фрагментов данных, к которым осуществляется частый доступ, например для создания счетчика, отслеживающего число просмотров страницы с момента последней перезагрузки приложения. Например, в следующем коде веб-сервер запускается через порт 8888 и подсчитывает каждый запрос:

```
var http = require('http');
var counter = 0;

var server = http.createServer(function(req, res) {
  counter++;
  res.write('I have been accessed ' + counter + ' times.');
  res.end();
}).listen(8888);
```

Что же касается приложений, надежность хранения информации которых не должна зависеть от перезапуска приложения или сервера, то более подходящим вариантом является хранилище данных на базе файлов.

### 5.1.2. Хранилище данных на базе файлов

В хранилищах данных на базе файлов для хранения данных применяется файловая система. Разработчики зачастую используют этот тип хранилища для хранения параметров приложения, а также постоянных данных, надежность хранения которых не должна зависеть от перезагрузки приложения или сервера.

#### Проблемы параллельного доступа

Несмотря на простоту использования хранилища данных на базе файлов, этот вариант хранения подходит не для всех типов приложений. Если, например, многопользовательское приложение сохраняет записи в файле, могут возникать проблемы, связанные с параллельным доступом. Два пользователя могут одновременно загрузить один и тот же файл, а затем изменить его, сохранив одну версию поверх другой. При этом изменения, внесенные одним из пользователей, будут потеряны. Поэтому для многопользовательских приложений лучше всего задействовать системы управления базами данных, поскольку возможность параллельного доступа предусмотрена в них изначально.

Чтобы проиллюстрировать использование хранилища данных на базе файлов, для веб-приложения со списком запланированных дел (см. главу 4) создадим в Node простую версию с поддержкой командной строки. Результат запуска этого варианта приложения показан на рис. 5.1.

```
1. Shell
$ node cli_tasks.js add Floss the cat.
Saved.
$ node cli_tasks.js list
Floss the cat.
$ node cli_tasks.js add Buy some hats.
Saved.
$ node cli_tasks.js list
Floss the cat.
Buy some hats.
$
```

**Рис. 5.1.** Список запланированных дел, реализованный в виде инструмента командной строки

Приложение будет сохранять задачи (запланированные дела) в файле .tasks, находящемся в той же папке, в которой выполняется сценарий. Задачи перед сохранением преобразуются в формат JSON. Эти же задачи при чтении из файла могут преобразовываться из формата JSON.

Чтобы создать приложение, нужно написать начальную логику, а затем определить вспомогательные функции, необходимые для выборки и сохранения задач.

### *Создание начальной логики*

Код начинается с объявления обязательных модулей, синтаксического разбора задачи и описания, полученных из аргументов командной строки, а также определения файла, в котором должны сохраняться задачи. Все это продемонстрировано в листинге 5.1.

**Листинг 5.1.** Выборка значений аргументов и определение пути к файлу базы данных

```
var fs = require('fs');
var path = require('path');
// Выборка аргументов из файла "node cli_tasks.js"
var args = process.argv.splice(2);
// Выборка первого аргумента (command)
var command = args.shift();
// Объединение оставшихся аргументов
var taskDescription = args.join(' ');
// Определение пути к базе данных относительно текущей рабочей папки
var file = path.join(process.cwd(), '/.tasks');
```

Если указать аргумент action, приложение либо выведет список сохраненных задач, либо добавит описание задачи в хранилище задач, как показано в листинге 5.2. Если же не указывать аргумент action, будет выведена подсказка по использованию приложения.

**Листинг 5.2.** Определение действия, выполняемого CLI-сценарием

```
switch (command) {
  // В варианте 'list' выводится список всех сохраненных задач
  case 'list':
    listTasks(file);
    break;

  // В варианте 'add' добавляется новая задача
  case 'add':
    addTask(file, taskDescription);
```

```
    break;

  // Во всех остальных случаях выводится подсказка по использованию
  default:
    console.log('Usage: ' + process.argv[0]
      + ' list|add [taskDescription]');
}
```

### *Определение вспомогательной функции для выборки задач*

На следующем шаге в коде приложения определяется вспомогательная функция loadOrInitializeTaskArray, предназначенная для выборки существующих задач. Как показано в листинге 5.3, она загружает текстовый файл, в котором находятся данные в формате JSON. В коде используются две асинхронные функции из модуля fs. Эти функции являются неблокирующими, обеспечивая продолжение цикла событий без ожидания возврата результатов файловой системой.

**Листинг 5.3.** Загрузка из текстового файла данных в формате JSON

```
function loadOrInitializeTaskArray(file, cb) {
  // Проверка существования файла с задачами
  fs.exists(file, function(exists) {
    var tasks = [];
    if (exists) {
      // Чтение из файла .tasks данных с задачами
      fs.readFile(file, 'utf8', function(err, data) {
        if (err) throw err;
        var data = data.toString();
        // Синтаксический разбор JSON-данных из списка
        // запланированных дел в массив задач
        var tasks = JSON.parse(data || '[]');
        cb(tasks);
      });
    } else {
      // Создание пустого массива задач в случае отсутствия файла tasks
      cb([]);
    }
  });
```

```
}
```

Далее вспомогательная функция loadOrInitializeTaskArray используется для реализации функциональности списка задач в функции listTasks.

**Листинг 5.4.** Функция списка задач

```
function listTasks(file) {
  loadOrInitializeTaskArray(file, function(tasks) {
    for(var i in tasks) {
      console.log(tasks[i]);
    }
  });
}
```

***Определение вспомогательной функции для хранения задач***

Теперь нужно определить вспомогательную функцию storeTasks, предназначенную для хранения задач в JSON-файле.

**Листинг 5.5.** Хранение задач на диске

```
function storeTasks(file, tasks) {
  fs.writeFile(file, JSON.stringify(tasks), 'utf8', function(err) {
    if (err) throw err;
    console.log('Saved.');
  });
}
```

Затем можно использовать вспомогательную функцию storeTasks, чтобы в функции addTask реализовать функциональность добавления задачи.

**Листинг 5.6.** Добавление задачи

```
function addTask(file, taskDescription) {
  loadOrInitializeTaskArray(file, function(tasks) {
    tasks.push(taskDescription);
    storeTasks(file, tasks);
  });
}
```

Благодаря использованию файловой системы в качестве хранилища данных можно относительно просто и быстро повысить надежность хранения данных. С помощью этих хранилищ можно также выполнять конфигурирование приложений. Если данные параметров приложений хранятся в текстовом файле и закодированы

в формате JSON, код, определенный ранее в функции loadOrInitializeTaskArray может быть переработан таким образом, чтобы выполнять чтение из JSON-файла и его синтаксический разбор.

В главе 13 приведены дополнительные сведения о манипулировании файловой системой с помощью Node. Ну а здесь мы рассмотрим традиционные хранилища данных, используемые в приложениях, — системы управления реляционными базами данных.

## 5.2. Системы управления реляционными базами данных

Системы управления реляционными базами данных (РСУБД) позволяют организовать хранение сложной информации и предоставляют средства для выборки этой информации. Такие системы традиционно применяются в относительно высокоуровневых приложениях, таких как системы управления контентом, системы управления связями с заказчиками, корзины интернет-магазинов и пр. При корректном использовании РСУБД хорошо выполняют возложенные на них функции, но при работе с ними нужны специальные знания по администрированию баз данных и знание языка SQL, требуется также доступ к серверу базы данных. Хотя существуют объектно-реляционные преобразователи (Object-Relational Mapper, ORM), API-интерфейсы которых способны генерировать SQL-код в фоновом режиме. Вопросы администрирования РСУБД, ORM и SQL выходят за рамки тем, рассматриваемых в книге, при желании обратитесь к одному из многочисленных интернет-ресурсов, чтобы получить сведения по этим технологиям.

У разработчика есть множество вариантов выбора реляционных баз данных, хотя большинство обычно выбирает базы данных с открытым исходным кодом, поскольку для этих баз данных доступна поддержка, они хорошо работают и к тому же совершенно бесплатны. В этом разделе рассматриваются две наиболее популярные полнофункциональные реляционные базы данных, MySQL и PostgreSQL. Если вы ранее не работали с базами данных, выберите базу данных MySQL, которая проще в установке и имеет большее число пользователей. Если же вы решите задействовать базу данных Oracle, вам понадобится модуль db-oracle (https://github.com/mariano/node-db-oracle), описание работы с которым также выходит за рамки темы этой книги.

Итак, начнем мы с базы данных MySQL, а затем перейдем к PostgreSQL.

### 5.2.1. MySQL

Одна из наиболее популярных баз данных в мире, MySQL, имеет широкую

поддержку среди сообщества Node-разработчиков. Если вы только начинаете работать с MySQL и хотите получить соответствующие знания, найдите официальный учебник в Интернете (http://dev.mysql.com/doc/refman/5.0/en/tutorial.html). Новички в SQL также могут найти множество справочников и книг в Интернете, включая книгу Криса Фехили (Chris Fehily) «SQL: Visual QuickStart Guide», Peachpit Press, 2008.

### *Создание с помощью MySQL приложения по учету работ*

Чтобы ознакомиться с преимуществами создания Node-приложений с помощью MySQL, рассмотрим соответствующий пример, создав бессерверное веб-приложение по учету работ. Вам потребуется указывать дату выполнения работы, потраченное на нее время и описание.

Создаваемое приложение будет иметь форму, используемую для ввода сведений о выполняемой работе (рис. 5.2).

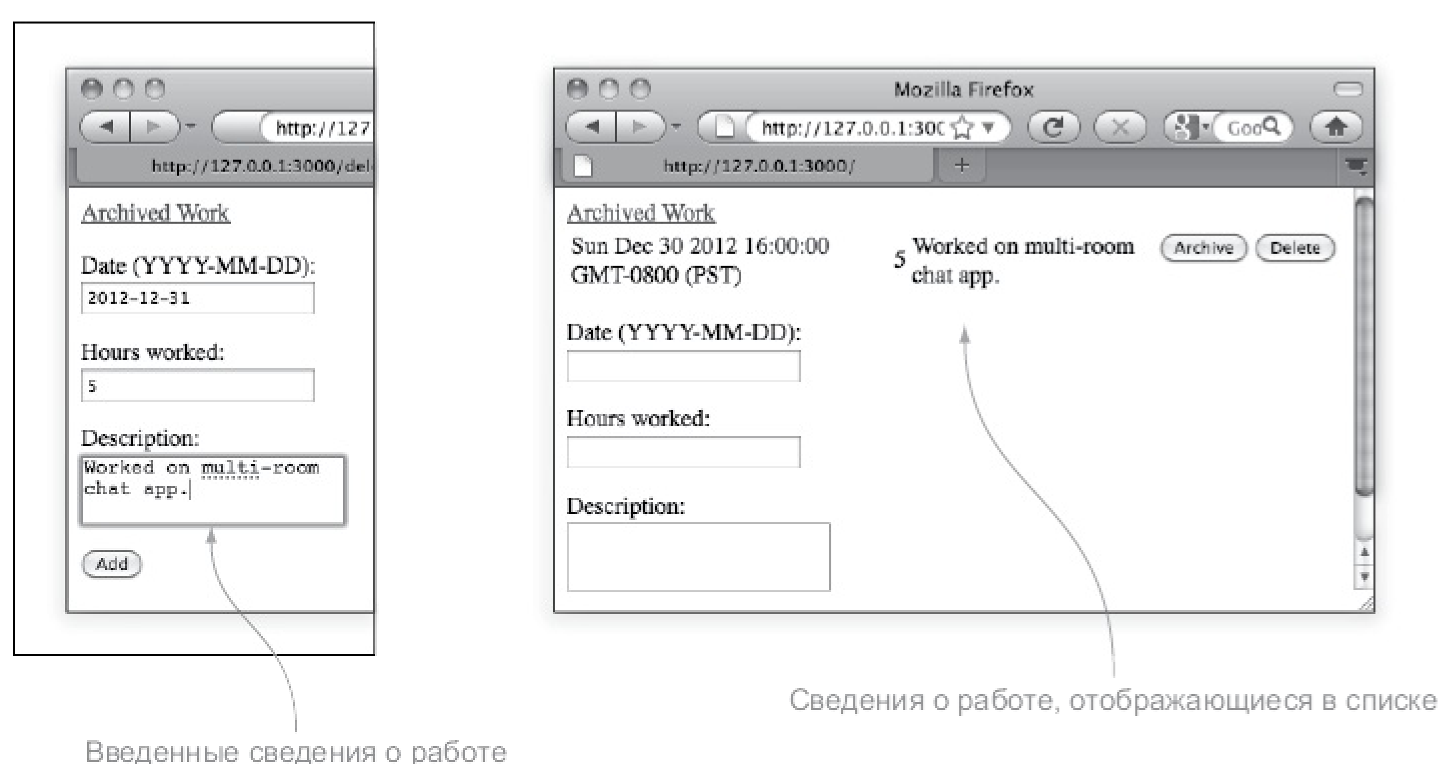



**Рис. 5.2.** Запись сведений о выполняемой работе

Введенные сведения о работе можно заархивировать или удалить, чтобы освободить место для ввода другой информации (рис. 5.3). После щелчка на ссылке Archived Work (Заархивированная работа) отображаются все ранее заархивированные позиции, относящиеся к выполняемой работе.

При создании этого веб-приложения в качестве простого хранилища данных можно воспользоваться файловой системой, хотя это может привести к трудностям при создании отчетов на основе этих данных. Например, если вы решите создать

отчет о работах, выполненных за последнюю неделю, придется считывать все записи и проверять дату выполнения работы в каждой записи. Если же хранить данные приложения в хранилище данных на основе РСУБД, вы сможете легкс генерировать отчеты с помощью SQL-запросов.

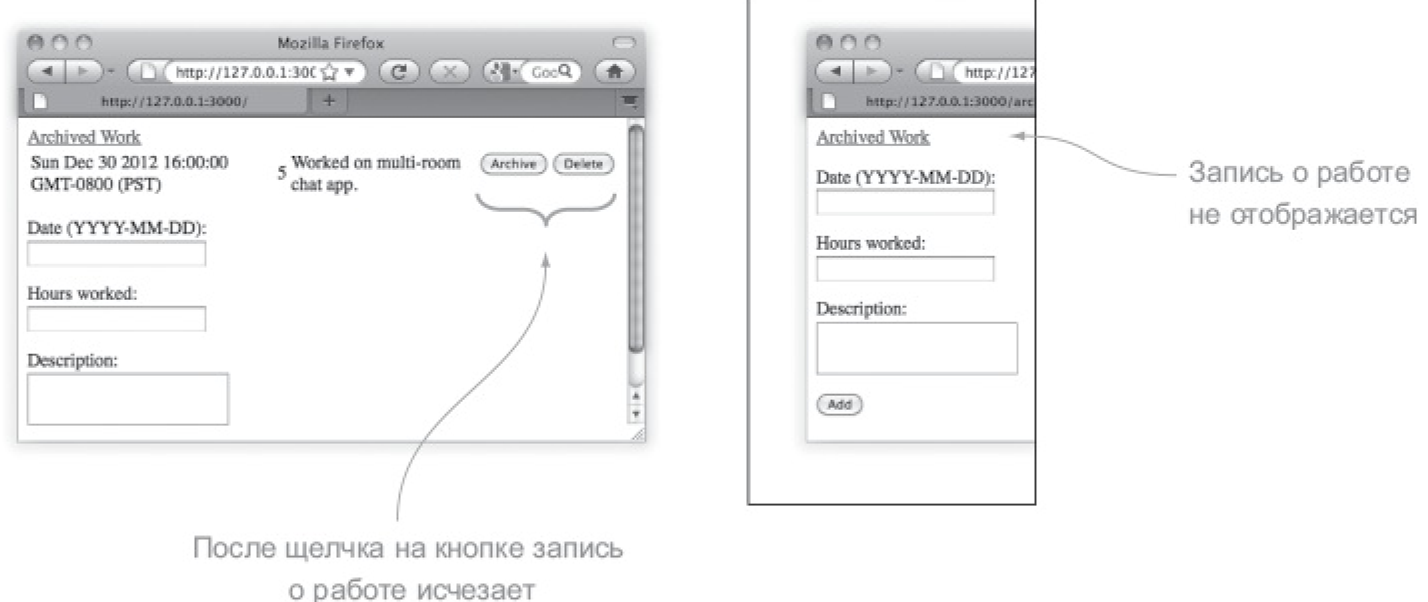



**Рис. 5.3.** Архивирование или удаление сведений о выполняемой работе

Чтобы создать приложение по учету работ, нужно:

- создать прикладную логику;
- разработать вспомогательные функции, обеспечивающие работоспособность приложения;
- написать функции для добавления, удаления, обновления и выборки данных с помощью MySQL;
- написать код визуализации HTML-записей и форм.

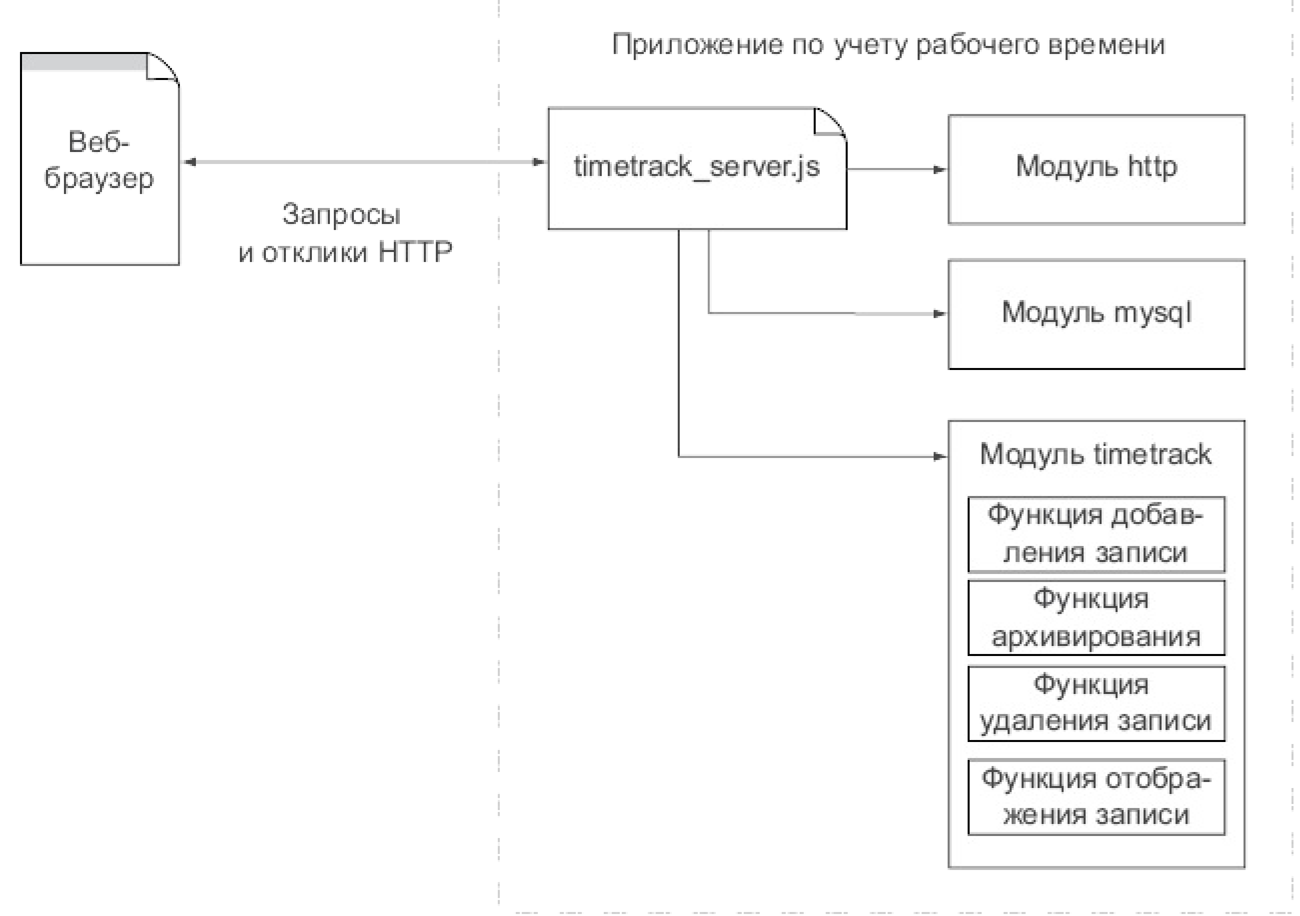



**Рис. 5.4.** Структура веб-приложения по учету работ

Приложение будет использовать встроенный в Node модуль http, реализующий функциональность сервера, и модуль от независимого производителя, обеспечивающий взаимодействие с MySQL-сервером. Нестандартный модуль timetrack должен поддерживать присущие приложению функции по хранению, изменению и выборке данных с помощью MySQL. Структура приложения представлена на рис. 5.4.

В конечном итоге мы создадим простое веб-приложение, позволяющее регистрировать сведения о выполняемой работе, а также просматривать, архивировать и удалять записи (рис. 5.5).

**Рис. 5.5.** Простое веб-приложение по учету работ

Чтобы обеспечить взаимодействие между Node и MySQL, мы воспользуемся популярным модулем nodemysql (https://github.com/felixge/node-mysql), разработанным Феликсом Гейзендорфером (Felix Geisendorfer). Но для начала установите Node-модуль mysql с помощью следующей команды:

```
npm install mysql
```

### *Создание прикладной логики*

Теперь нам нужно создать два файла, в которых будет находиться код приложения. Файл timetrack_server.js применяется для запуска приложения, а файл timetrack.js — это модуль, обеспечивающий функциональность приложения.

Начните с создания файла timetrack_server.js и добавьте в него код из листинга 5.7. Этот код включает APIhttp, код приложения и API mysql. В соответствии с вашей конфигурацией MySQL заполните поля host, user и password, в которых задаются название хоста, имя пользователя и пароль.

**Листинг 5.7.** Настройка приложения и инициализация соединения с базой данных

```
var http = require('http');
var work = require('./lib/timetrack');
// API-интерфейс MySQL, загружаемый по требованию
var mysql = require('mysql');

// Подключение к MySQL-серверу
var db = mysql.createConnection({
```

```
    host: '127.0.0.1',
    user: 'myuser',
    password: 'mypassword',
    database: 'timetrack'
});
```

Далее добавьте код из листинга 5.8, который определяет базовое поведение веб-приложения. С помощью этого приложения можно просматривать, добавлять или удалять записи, соответствующие работам. Также приложение должно позволять архивировать записи. Архивированная запись исчезает с главной страницы, но ее можно просмотреть на отдельной веб-странице.

**Листинг 5.8.** Маршрутизация HTTP-запросов

```
var server = http.createServer(function(req, res) {
  switch (req.method) {
    // Маршрутизация HTTP-запросов методом POST
    case 'POST':
      switch(req.url) {
        case '/':
        work.add(db, req, res);
        break;
      case '/archive':
        work.archive(db, req, res);
        break;
      case '/delete':
        work.delete(db, req, res);
        break;
      }
      break;
    // Маршрутизация HTTP-запросов методом GET
    case 'GET':
    switch(req.url) {
      case '/':
      work.show(db, res);
      break;
    case '/archived':
      work.showArchived(db, res);
```

```
      }
    break;
    }
  });
```

Код, представленный в листинге 5.9, представляет собой последнее дополнение, которое нужно внести в файл timetrack_server.js. Этот код создает таблицу базы данных, если таблица еще не создана, и запускает HTTP-сервер, прослушивающий IP-адрес 127.0.0.1, соответствующий TCP/IP-порту с номером 3000. Все запросы между Node и MySQL выполняются с помощью функции query.

**Листинг 5.9.** Создание таблицы базы данных

```
db.query(
  // SQL-код создания таблицы
  "CREATE TABLE IF NOT EXISTS work ("
  + "id INT(10) NOT NULL AUTO_INCREMENT, "
  + "hours DECIMAL(5,2) DEFAULT 0, "
  + "date DATE, "
  + "archived INT(1) DEFAULT 0, "
  + "description LONGTEXT,"
  + "PRIMARY KEY(id))",
  function(err) {
    if (err) throw err;
    console.log('Server started...');
    // Запуск HTTP-сервера
    server.listen(3000, '127.0.0.1');
  }
);
```

### *Создание вспомогательных функций для отправки HTML-запросов, создания форм и получения данных форм*

Теперь, когда мы полностью описали файл, используемый для запуска приложения, настало время создать файл, реализующий остальную функциональность. Создайте папку lib, поместите туда файл timetrack.js, включив в него код из листинга 5.10, который содержит встроенный в Node API-интерфейс querystring, а также определяет вспомогательные функции для отправки HTML-кода веб-страницы и приема данных, введенных в формы.

**Листинг 5.10.** Вспомогательные функции для отправки HTML-кода создания форм и получения данных форм

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

// Отправка HTML-ответа
exports.sendHtml = function(res, html) {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
};

// Синтаксический разбор POST-данных HTTP-запроса
exports.parseReceivedData = function(req, cb) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function() {
    var data = qs.parse(body);
    cb(data);
  });
};

// Визуализация простой формы
exports.actionForm = function(id, path, label) {
  var html = '<form method="POST" action="' + path + '">' +
    '<input type="hidden" name="id" value="' + id + '">' +
    '<input type="submit" value="' + label + '" />' +
    '</form>';
  return html;
};
```

### *Добавление данных с помощью MySQL*

Теперь, когда вспомогательные функции созданы, настало время определить логику добавления записи о выполненной работе в базу данных MySQL. Включите

код из листинга 5.11 в файл timetrack.js.

**Листинг 5.11.** Добавление записи о выполняемой работе

```
exports.add = function(db, req, res) {
  // Синтаксический разбор POST-данных HTTP-запроса
  exports.parseReceivedData(req, function(work) {
    db.query(
      // SQL-код, добавляющий запись о работе
      "INSERT INTO work (hours, date, description) " +
      " VALUES (?, ?, ?)",
      // Данные записей о работах
      [work.hours, work.date, work.description],
      function(err) {
        if (err) throw err;
        // Вывод пользователю списка записей о работах
        exports.show(db, res);
      }
    );
  });
};
```

Обратите внимание на использование символа вопросительного знака (?) в качестве заполнителя, определяющего местонахождение параметра. Каждый параметр еще до добавления в запрос автоматически экранируется с помощью метода query, что позволяет предотвращать так называемые атаки внедрения SQL-кода.

Также обратите внимание, что второй аргумент метода query представляет собой список значений, используемых вместо символов подстановки.

### *Удаление данных в MySQL*

Далее нужно добавить следующий код в файл timetrack.js. Этот код удаляет запись о работе.

**Листинг 5.12.** Удаление записи о работе

```
exports.delete = function(db, req, res) {
  // Синтаксический разбор POST-данных HTTP-запроса
  exports.parseReceivedData(req, function(work) {
    db.query(
```

```
      // SQL-код удаления записи о работе
      "DELETE FROM work WHERE id=?",
      // Идентификатор записи о работе
      [work.id],
      function(err) {
        if (err) throw err;
        // Вывод пользователю списка записей о работах
        exports.show(db, res);
      }
    );
  });
};
```

### *Обновление данных в MySQL*

Чтобы добавить логику, выполняющую обновление записи о работе и помечающую ее как архивированную, в файл timetrack.js включите код из листинга 5.13.

**Листинг 5.13.** Архивирование записи о работе

```
exports.archive = function(db, req, res) {
  // Синтаксический разбор POST-данных HTTP-запроса
  exports.parseReceivedData(req, function(work) {
    db.query(
      // SQL-код обновления записи о работе
      "UPDATE work SET archived=1 WHERE id=?",
      // Идентификатор записи о работе
      [work.id],
      function(err) {
        if (err) throw err;
        // Вывод пользователю списка записей о работах
        exports.show(db, res);
      }
    );
  });
};
```

### *Выборка данных в MySQL*

После создания логики добавления, удаления и обновления записей о работах нужно добавить код из листинга 5.14, который обеспечивает выборку данных, относящихся к записи о работе (архивированной или неархивированной), чтобы их можно было визуализировать в виде HTML-кода. Выполнение запроса обеспечивает обратный вызов, включающий аргумент rows для возвращаемых записей.

**Листинг 5.14.** Выборка записей о работе

```
exports.show = function(db, res, showArchived) {
  // SQL-код выборки записей о работе
  var query = "SELECT * FROM work " +
    "WHERE archived=? " +
    "ORDER BY date DESC";
  var archiveValue = (showArchived) ? 1 : 0;
  db.query(
    query,
    // Желательное архивное состояние записи о работе
    [archiveValue],
    function(err, rows) {
      if (err) throw err;
      html = (showArchived)
        ? ''
      : '<a href="/archived">Archived Work</a><br/>';
      // Форматирование результатов в виде HTML-таблицы
      html += exports.workHitlistHtml(rows);
      html += exports.workFormHtml();
      // Отправка HTML-ответа пользователю
      exports.sendHtml(res, html);
    }
  );
};

exports.showArchived = function(db, res) {
  // Вывод только архивированных записей о работах
  exports.show(db, res, true);
};
```

*Визуализация записей с помощью MySQL*

Добавьте код из листинга 5.15 в файл timetrack.js. Этот код позволяет визуализировать записи о работах в виде HTML-кода.

**Листинг 5.15.** Визуализация записей о работах в HTML-таблице

```
exports.workHitlistHtml = function(rows) {
  var html = '<table>';
  // Визуализация каждой записи о работе в виде строки HTML-таблицы
  for(var i in rows) {
    html += '<tr>';
    html += '<td>' + rows[i].date + '</td>';
    html += '<td>' + rows[i].hours + '</td>';
    html += '<td>' + rows[i].description + '</td>';
     // Вывод кнопки Archive, если запись о работе еще не заархивирована
    if (!rows[i].archived) {
      html += '<td>' + exports.workArchiveForm(rows[i].id) + '</td>';
    }
    html += '<td>' + exports.workDeleteForm(rows[i].id) + '</td>';
    html += '</tr>';
  }
  html += '</table>';
  return html;
};
```

*Визуализация HTML-форм*

И напоследок добавьте в файл timetrack.js код из листинга 5.16. Этот код призван визуализировать HTML-формы, используемые в приложении.

**Листинг 5.16.** HTML-формы для добавления, архивирования и удаления записей о работах

```
exports.workFormHtml = function() {
  // Визуализация пустой HTML-формы, предназначенной
  // для ввода новой записи о работе
  var html = '<form method="POST" action="/">' +
  '<p>Date (YYYY-MM-DD):<br/><input name="date" type="text"><p/>' +
  '<p>Hours worked:<br/><input name="hours" type="text"><p/>' +
```

```
    '<p>Description:<br/>' +
    '<textarea name="description"></textarea></p>' +
    '<input type="submit" value="Add" />' +
    '</form>';
    return html;
};

// Визуализация кнопки Archive в форме
exports.workArchiveForm = function(id) {
    return exports.actionForm(id, '/archive', 'Archive');
};

// Визуализация кнопки Delete в форме
exports.workDeleteForm = function(id) {
    return exports.actionForm(id, '/delete', 'Delete');
};
```

***Тестирование***

Теперь, когда приложение полностью определено, попробуем его запустить. Предварительно создайте базу данных timetrack с помощью интерфейса администратора базы данных MySQL. Затем запустите приложение, введя в командной строке следующую команду:

```
node timetrack_server.js
```

И наконец, в окне веб-браузера перейдите по адресу [http://127.0.0.1:3000](http://127.0.0.1:3000)/, чтобы воспользоваться приложением.

Хотя СУБД MySQL является одной из наиболее популярных реляционных ба данных, в силу ряда причин более предпочтительна база данных PostgreSQL. Давайте выясним, как использовать PostgreSQL в нашем приложении.

### 5.2.2. PostgreSQL

СУБД PostgreSQL пользуется заслуженным уважением по причине соответстви стандартам и надежности, поэтому многие разработчики Node-приложений отдают предпочтение этой РСУБД, выделяя ее среди других. В отличие от MySQ PostgreSQL поддерживает рекурсивные запросы и многие специализированные типы данных. Кроме того, в PostgreSQL используется множество методов

стандартной аутентификации, таких как протокол LDAP (Lightweight Directory Access Protocol) и интерфейс GSSAPI (Generic Security Services Application Program Interface). Также в PostgreSQL поддерживается синхронная репликация, обеспечивающая при необходимости масштабируемость или избыточность. Утрата данных, возможная при репликации такого типа, предотвращается путем верификации, выполняемой после завершения каждой операции с данными.

Если вы только начинаете использовать PostgreSQL и хотите изучить возможности этой РСУБД, обратитесь к официальному руководству, которое можно найти в Интернете (www.postgresql.org/docs/7.4/static/tutorial.html).

Наиболее часто применяемый и лучший API-интерфейс для PostgreSQL — это модуль node-postgres, созданный Брайаном Карлсоном (Brian Carlson). Его можно загрузить с веб-сайта по адресу https://github.com/brianc/node-Postgres.

**проблемы для пользователей windows**

Хотя модуль node-postgres предназначается для работы на платформе Windows, создатель модуля изначально тестировал его на платформах Linux и OS X. Что же касается пользователей Windows, они могут столкнуться с многочисленными проблемами, такими как фатальная ошибка во время установки. Поэтому на платформе Windows вместо PostgreSQL лучше использовать MySQL.

Установите модуль node-postgres с помощью следующей команды:

```
npm install pg
```

### *Подключение к базе данных PostgreSQL*

После установки модуля node-postgres можно подключиться к PostgreSQL и выбрать запрашиваемую базу данных. (Если пароль не задан, опустите часть :mypassword в строке подключения.)

```
var pg = require('pg');
var conString = "tcp://myuser:mypassword@localhost:5432/mydatabase";

var client = new pg.Client(conString);
client.connect();
```

### *Вставка строки в таблицу базы данных*

Для выполнения запросов к базе данных применяется метод query. В следующем примере кода демонстрируется вставка строки в таблицу базы данных:

```
client.query(
  'INSERT INTO users ' +
  "(name) VALUES ('Mike')"
);
```

Символы подстановки ($1, $2 и подобные) указывают место подстановки параметра. Каждый параметр экранируется перед добавлением в отчет, предотвращая атаки внедрения SQL-кода. Следующий пример кода демонстрирует вставку строки с помощью символов подстановки:

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES ($1, $2)",
  ['Mike', 39]
);
```

Чтобы получить значение первичного ключа, после вставки строки с помощью инструкции RETURNING укажите название строки, значение которой нужн вернуть. Затем добавьте обратный вызов в качестве последнего аргумента в вызове метода query, как показано в следующем примере:

```
client.query(
  "INSERT INTO users " +
  "(name, age) VALUES ($1, $2) " +
  "RETURNING id",
  ['Mike', 39],
  function(err, result) {
    if (err) throw err;
    console.log('Insert ID is ' + result.rows[0].id);
  }
);
```

### *Создание запроса, возвращающего результаты*

При создании запроса, возвращающего результаты, нужно сохранить в переменной возвращаемое значение клиентского метода query. Метод query возвращает объект, который наследует поведение класса EventEmitter, что позволяет воспользоваться преимуществами встроенной в Node функциональности. Этот объект генерирует

событие row для каждой выбираемой в базе данных строки. В листинге 5.17 показан код, который выводит данные из каждой строки, возвращаемой запросом. Обратите внимание на использование слушателей класса EventEmitter, определяющих операции, выполняемые со строками таблицы, а также операцию, которая выполняется после завершения выборки данных.

**Листинг 5.17.** Выборка строк из базы данных в PostgreSQL

```
var query = client.query(
  "SELECT * FROM users WHERE age > $1",
  [40]
);

// Обработка возврата строки
query.on('row', function(row) {
  console.log(row.name)
});

// Обработка завершения запроса
query.on('end', function() {
  client.end();
});
```

После выборки последней строки генерируется событие end, которое может являться сигналом для закрытия базы данных или выполнения другой прикладной логики.

Реляционные базы данных — это классические «рабочие лошадки», но если язык SQL вам не нужен, можно обратиться к альтернативным базам данных, которые завоевывают все большую популярность.

## 5.3. Базы данных, не поддерживающие SQL

На заре существования баз данных нереляционные базы данных были широко распространены. Однако постепенно их вытеснили реляционные базы данных, которые стали применяться при разработке различных приложений, как автономных, так и веб-приложений. В последние годы интерес к нереляционным СУБД возродился, что обусловлено такими преимуществами, как масштабируемость, простота и возможность применения в самых разных сценариях. Нереляционные базы данных, сокращенно называемые «NoSQL», либо вообще не используют SQL, либо наравне с SQL обращаются к другим технологиям.

В то время как в реляционных СУБД производительность приносится в жертву надежности, во многих нереляционных базах данных во главу угла ставится производительность. Поэтому нереляционные базы данных лучше всего выбирать для разработки систем реального времени, выполняющих аналитические операции и обмен сообщениями. В нереляционных базах данных также обычно не требуется предварительное определение схем данных, что может быть полезно для приложений, в которых данные хранятся в виде иерархии, но эта иерархия меняется.

В этом разделе рассматриваются две популярные нереляционные базы данных, Redis и MongoDB. Также рассматривается популярный API-интерфейс Mongoose обеспечивающий доступ к базе данных MongoDB. В этом API-интерфейсе появилось множество удобных инструментов, позволяющих экономить время. Установка и администрирование баз данных Redis и MongoDB выходят за рамки темы этой книги, но вы можете найти инструкции о том, как начать работать с Redis, на веб-сайте http://redis.io/topics/quickstart. С веб-сайта http://docs.mongodb.org/manual/installation/#installation-guides можно загрузить сведения о работе с MongoDB.

### 5.3.1. Redis

Хранилище данных Redis применяется для обработки простых данных, которые не требуют долговременного хранения, например мгновенных сообщений или игровых данных. Данные Redis хранит в оперативной памяти, а изменения, связанные со входом в систему, — на диске. Хотя объем хранилища ограничен, операции с данными в Redis могут выполняться очень быстро. Если Redis-сервер терпит крах и содержимое оперативной памяти «обнуляется», данные могут быть восстановлены с помощью журнала, находящегося на диске.

В Redis поддерживается словарь, включающий примитивные, но весьма полезные команды (http://redis.io/ commands), которые могут применяться для обработки многих структур данных. Большинство структур данных, поддерживаемых Redis, известны разработчикам, поскольку они аналогичны структурам, часто применяемым в программировании. Речь идет о хеш-таблицах, списках и парах ключ/значение, которые используются подобно простым переменным. Хеш-таблицы и пары ключ/значение проиллюстрированы на рис. 5.6. В Redis также поддерживается менее известная структура данных, которая называется *множество* (set) и с которой мы позже познакомимся.

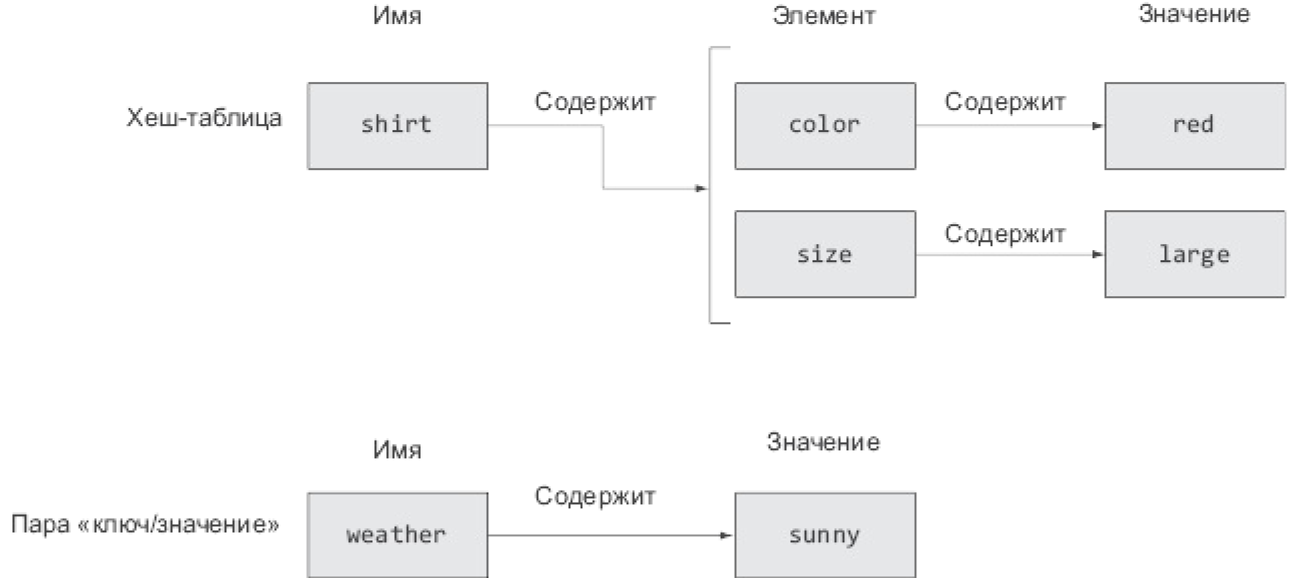



**Рис. 5.6.** В Redis поддерживаются простые типы данных, включая хеш-таблицы и пары ключ/значение

В этой главе мы не будем рассматривать все команды, доступные в Redis, зато познакомимся с множеством примеров, которые можно использовать как основу для разработки собственных приложений. Если вы только начинаете работать с Redis и до изучения примеров хотите получить представление о возможностях этой базы данных, прочитайте учебник «Try Redis» (http://try.redis.io/). А чтобы получить исчерпывающее представление о разработке приложений в Redis, прочтите книгу Джосайи Л. Карлсона ( Josiah L. Carlson) «Redis in Action», Manning, 2013.

Опытные разработчики могут воспользоваться API-интерфейсом для Redis — модулем node_redis, разработанным Мэттом Рэнни (Matt Ranney) и доступным на веб-сайте https://github.com/mranney/node_redis. Установите этот модуль с помощью следующей команды:

```
npm install redis
```

### *Подключение к Redis-серверу*

Следующий код устанавливает соединение с Redis-сервером. При этом используется заданный по умолчанию TCP/IP-порт на том же хосте. Созданный Redis-клиент включает унаследованное поведение класса EventEmitter, который генерирует событие error в случае, если возникают проблемы при попытке установки соединения с Redis-сервером. Как демонстрируется в этом примере, можно определить собственный код обработки ошибок, добавив слушателя для типа событий error:

```
var redis = require('redis');
var client = redis.createClient(6379, '127.0.0.1');
```

```
client.on('error', function (err) {
  console.log('Error ' + err);
});
```

### *Обработка данных в Redis*

После подключения к Redis-серверу приложение может начать обработку данных с помощью объекта client. Следующий пример кода демонстрирует хранение и выборку пар ключ/значение:

```
// Функция print выводит на печать результаты выполнения операции
// либо ошибку, если таковая имела место
client.set('color', 'red', redis.print);
client.get('color', function(err, value) {
  if (err) throw err;
  console.log('Got: ' + value);
});
```

### *Хранение и выборка значений с помощью хеш-таблицы*

В листинге 5.18 представлен код, выполняющий хранение и выборку значений в чуть более сложной структуре данных: в *хеш-таблице* (hash table), которая также известна как *хеш-карта* (hash map). Хеш-таблица фактически является таблицей идентификаторов, называемых *ключами* (keys), которые связаны с соответствующими *значениями* (values).

С помощью команды hmset в Redis элементам хеш-таблицы, которые идентифицируются по ключу, присваивается некое значение. С помощью команды hkeys в Redis выводятся ключи для каждого элемента хеш-таблицы.

**Листинг 5.18.** Хранение данных в элементах хеш-таблицы в Redis

```
// Установка элементов хеш-таблицы
client.hmset('camping', {
  'shelter': '2-person tent',
  'cooking': 'campstove'
},  redis.print);

// Получение значения элемента "cooking"
client.hget('camping', 'cooking', function(err, value) {
```

```
  if (err) throw err;
  console.log('Will be cooking with: ' + value);
});

// Получение ключей хеш-таблицы
client.hkeys('camping', function(err, keys) {
  if (err) throw err;
  keys.forEach(function(key, i) {
    console.log(' ' + key);
  });
});
```

### *Хранение и выборка данных с помощью списка*

Еще одной структурой данных, поддерживаемых в Redis, является список. Теоретически в таком списке может находиться свыше четырех миллиардов элементов, что ограничивается доступным объемом памяти.

Следующий код демонстрирует хранение и выборку значений в списке. Для добавления значений в список в Redis применяется команда lpush. С помощью команды lrange в Redis происходит выборка диапазона элементов списка, задаваемого аргументами, указывающими на начало и конец списка. С помощью аргумента -1 в этом примере кода задается последний элемент списка. В результате его использования командой lrange выбираются все элементы списка:

```
client.lpush('tasks', 'Paint the bikeshed red.', redis.print);
client.lpush('tasks', 'Paint the bikeshed green.', redis.print);
client.lrange('tasks', 0, -1, function(err, items) {
  if (err) throw err;
  items.forEach(function(item, i) {
    console.log(' ' + item);
  });
});
```

Список в Redis представляет собой упорядоченный список строк. Если, например, создается приложение, используемое для планирования конференций, в списке можно хранить маршруты, позволяющие добраться до места проведения конференции.

Списки в Redis концептуально подобны массивам, поддерживаемым во многих

языках программирования, предлагая знакомый многим программистам способ обработки данных. Спискам присущ единственный недостаток, заключающийся в недостаточном быстродействии при выборке данных. По мере увеличения длины списка скорость выборки в Redis уменьшается. Скорость выборки данных из списка оценивается как O($n$), где $n$ — количество элементов списка.

**нотация «большого О»**

В компьютерных науках с помощью нотации «большого О» алгоритмы классифицируются по степени сложности. Оценка алгоритма с применением нотации «большого О» позволяет быстро оценить его производительность и область применения. Если вы не знакомы с этой нотацией, обратитесь к статье Роба Белла (Rob Bell) «A Beginner's Guide to Big O Notation» на веб-сайте http://mng.bz/UJu7.

### *Хранение и выборка данных с помощью множеств*

В Redis множество представляет собой неупорядоченную группу строк. Если вы, например, создаете приложение, используемое для планирования конференций, в множестве можно хранить сведения об участниках конференции. Выборка данных из множеств осуществляется быстрее, чем из списков. Время, требуемое для выборки элемента множества, не зависит от размера этого множества и в нотации «большого О» оценивается как O(1).

Элементы множеств должны быть уникальны — если дважды попытаться сохранить в множестве одно и то же значение, вторая попытка будет проигнорирована.

Следующий код иллюстрирует хранение и выборку IP-адресов. Команда sadd в Redis делает попытку добавить значение в множество, а команда smembers возвращает сохраненные значения. В этом примере производится двойная попытка добавить IP-адрес 204.10.37.96, но как можно увидеть после вывода элементов множества, IP-адрес сохраняется только один раз:

```
client.sadd('ip_addresses', '204.10.37.96', redis.print);
client.sadd('ip_addresses', '204.10.37.96', redis.print);
client.sadd('ip_addresses', '72.32.231.8', redis.print);
client.smembers('ip_addresses', function(err, members) {
  if (err) throw err;
  console.log(members);
});
```

*Доставка данных с помощью каналов*

Благодаря *каналам* (channels) возможности баз данных в Redis превышают возможности традиционных хранилищ данных. Каналы представляют собой механизм доставки данных, поддерживающий средства публикации/подписки, как показано на рис. 5.7. Эти средства можно использовать при создании приложений для чатов и компьютерных игр.

**Рис. 5.7.** С помощью каналов в Redis реализуется простое решение, обеспечивающее универсальный сценарий передачи данных

Клиент Redis может осуществлять подписку или публиковать сообщения в любом выбранном канале. Подписка на канал означает, что вы будете получать произвольное сообщение, отправленное каналу. Публикация сообщения в канале означает, что это сообщение будут получать все клиенты, подписанные на этот канал.

В листинге 5.19 представлен пример TCP/IP-сервера для чата, реализующего доступную в Redis функциональность публикации/подписки.

**Листинг 5.19.** Простой сервер для чата, реализующий функциональность публикации/подписки в Redis

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

// Определение логики установки для каждого
// пользователя, подключающегося к серверу чата
var server = net.createServer(function(socket) {
  var subscriber;
  var publisher;

  socket.on('connect', function() {
```

```
    // Создание клиента подписки для каждого пользователя
    subscriber = redis.createClient();
    // Подписка на канал
    subscriber.subscribe('main_chat_room');

    // Если сообщение получено из канала, показываем его пользователю
    subscriber.on('message', function(channel, message) {
      socket.write('Channel ' + channel + ': ' + message);
    });

    // Создание клиента публикации для каждого пользователя
    publisher = redis.createClient();
  });

  socket.on('data', function(data) {
    // Публикация сообщения, введенного пользователем
    publisher.publish('main_chat_room', data);
  });

  socket.on('end', function() {
    // Если пользователь отключается, завершаем клиентские подключения
    subscriber.unsubscribe('main_chat_room');
    subscriber.end();
    publisher.end();
  });
});

// Запуск сервера для чата
server.listen(3000);
```

### *Максимизация производительности модуля node_redis*

При развертывании Node-приложений, использующих API-интерфейс node_redis можно задействовать модуль hiredis, разработанный Питером Нурдхуисом (Pieter Noordhuis). Этот модуль, доступный на веб-сайте https://github.com/pietern/hiredis-

node, позволяет существенно повысить производительность Redis за счет официальной C-библиотеки hiredis. API-интерфейс node_redis автоматически применяет библиотеку hiredis, если она установлена, не обращаясь к JavaScript-реализации.

Чтобы установить библиотеку hiredis, воспользуйтесь:

npm install hiredis

Поскольку библиотека hiredis компилируется из C-кода, а внутренние API-интерфейсы время от времени меняются, придется повторно скомпилировать hiredis после обновления Node.js. Чтобы повторно скомпилировать модуль hiredis, воспользуйтесь следующей командой:

npm rebuild hiredis

Теперь, когда мы познакомились с базой данных Redis, используемой для высокоскоростной обработки примитивов данных, рассмотрим более универсальную базу данных MongoDB.

### 5.3.2. MongoDB

База данных MongoDB является универсальной нереляционной базой данных. Она может применяться для разработки некоторых типов приложений, обычно создаваемых с помощью РСУБД.

В базе данных MongoDB документы хранятся в виде *коллекций* (collections.). Как показано на рис. 5.8, документам, относящимся к коллекции, нет нужды использовать одну и ту же схему, каждый документ может иметь собственную схему. Поскольку предварительно создавать схемы не нужно, в MongoDB базы данных являются более гибкими, чем традиционные РСУБД.

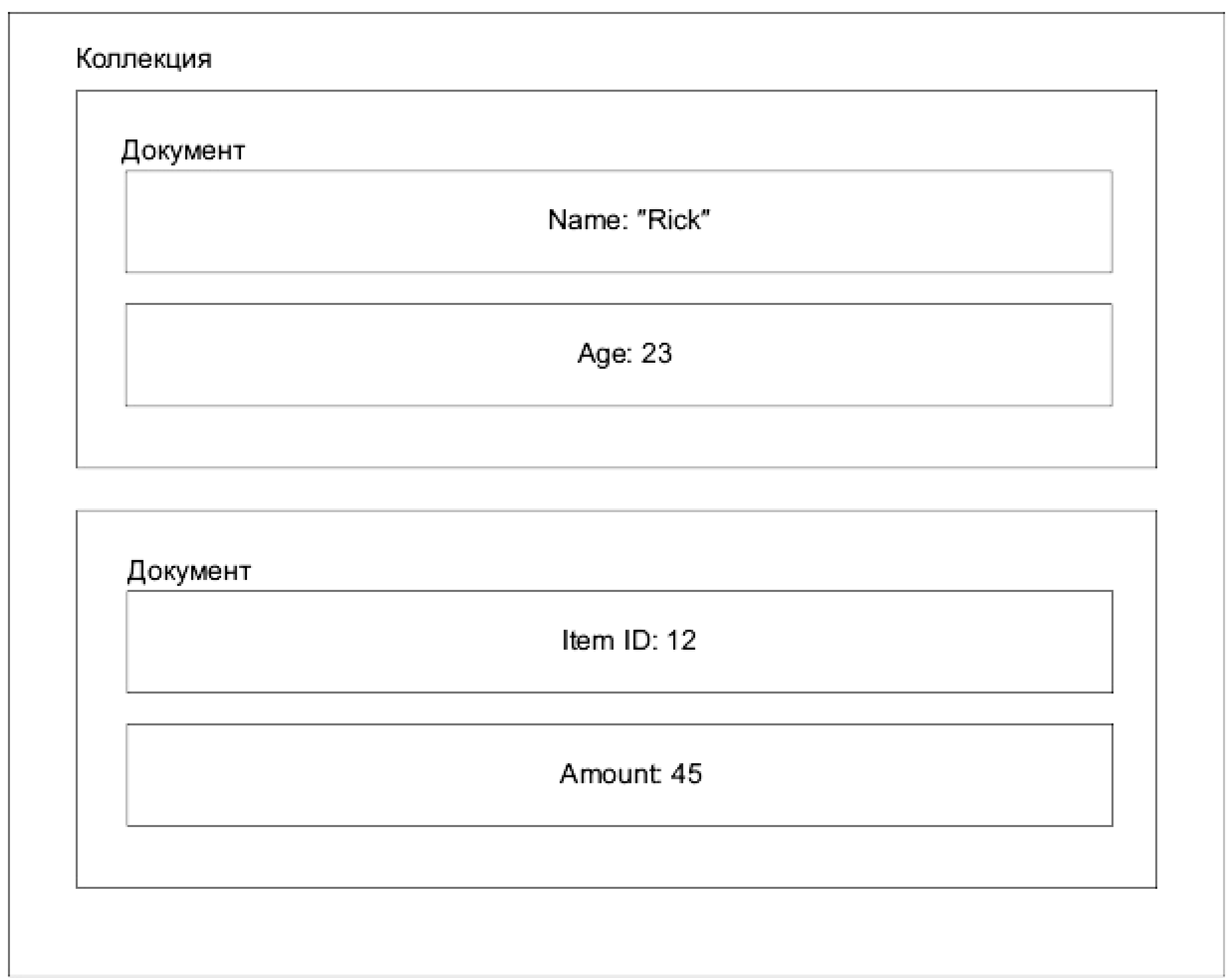



**Рис. 5.8.** Для каждого элемента коллекции в MongoDB может применяться собственная схема, отличающаяся от других

Чаще всего используется и лучше поддерживается API-интерфейсом MongoDB модуль node-mongodb-native, разработанный Кристианом Амором Квальхаймом (Christian Amor Kvalheim). Этот модуль можно загрузить с веб-сайта https://github.com/mongodb/node-mongodb-native. Чтобы его установить, воспользуйтесь следующей командой (обратите внимание, что для установки этого модуля в Windows понадобится файл msbuild.exe и Microsoft Visual Studio):

```
npm install mongodb
```

### *Подключение к MongoDB*

После установки модуля node-mongodb-native и запуска MongoDB-сервера для соединения с сервером воспользуйтесь следующим кодом:

```
var mongodb = require('mongodb');
var server = new mongodb.Server('127.0.0.1', 27017, {});

var client = new mongodb.Db('mydatabase', server, {w: 1});
```

***Получение доступа к коллекции в MongoDB***

С помощью следующего фрагмента кода можно получить доступ к коллекции при открытом соединении с базой данных:

```
client.open(function(err) {
  if (err) throw err;
  client.collection('test_insert', function(err, collection) {
    if (err) throw err;
  // Сюда нужно вставить код запроса
    console.log('We are now able to perform queries.');
  });
});
```

Чтобы после завершения выполняемых с базой данных операций закрыть соединение с базой данных, выполните в MongoDB команду client.close().

***Вставка документа в коллекцию***

В результате выполнения следующего кода в коллекцию включается документ, а также выводится на печать уникальный идентификатор документа:

```
collection.insert(
  {
    "title": "I like cake",
    "body": "It is quite good."
  },
  // Безопасный режим указывает на то, что операция с базой данных
  // должна быть завершена перед выполнением обратного вызова
  {safe: true},
  function(err, documents) {
    if (err) throw err;
    console.log('Document ID is: ' + documents[0]._id);
  }
);
```

**Безопасный режим**

Указывая в запросе инструкцию {safe: true}, вы показываете, что хотите, чтобы операция в базе данных была завершена перед выполнением обратного вызова. Если

код обратного вызова каким-либо образом зависит от завершения выполняемой в базе данных операции, укажите это инструкцией. Если же код обратного вызова не зависит от выполняемой операции, для ее завершения используйте инструкцию {}.

Хотя с помощью функции console.log идентификатор documents[0] _id можно вывести в виде строки, фактически это не строка. Идентификаторы документов в базе данных MongoDB закодированы в *формате BSON*(Binary JSON — двоичный формат JSON). Этот формат обмена данными преимущественно используется в MongoDB вместо JSON для перемещения данных между MongoDB-клиентом MongoDB-сервером. В большинстве случаев BSON обеспечивает более эффективно расходование пространства, чем JSON, а также позволяет выполнять более быстрый синтаксический разбор. В результате достигается более быстрое взаимодействие с базой данных.

***Обновление данных с помощью идентификаторов документов***

С помощью идентификаторов BSON-документов можно обновлять данные. Код листинга 5.20 выполняет обновление документа на основании его идентификатора.

**Листинг 5.20.** Обновление документа в MongoDB

```
var _id = new client.bson_serializer
            .ObjectID('4e650d344ac74b5a01000001');
collection.update(
  Put MongoDB
  query code here
  {_id: _id},
  {$set: {"title": "I ate too much cake"}},
  {safe: true},
  function(err) {
    if (err) throw err;
  }
);
```

***Поиск документов***

Для поиска документов в базе данных MongoDB используется метод find. В следующем примере кода выводятся все элементы коллекции с заголовком «I like

cake»:

```
collection.find({"title": "I like cake"}).toArray(
  function(err, results) {
  if (err) throw err;
    console.log(results);
  }
);
```

***Удаление документов***

Хотите что-либо удалить? Чтобы удалить запись, укажите ее внутренний идентификатор (либо выберите любой другой критерий) с помощью следующего кода:

```
var _id = new client
        .bson_serializer
        .ObjectID('4e6513f0730d319501000001');
collection.remove({_id: _id}, {safe: true}, function(err) {
  if (err) throw err;
});
```

MongoDB — это весьма мощная база данных, а с помощью модуля node-mongodb-native к ней обеспечивается быстрый доступ. Тем не менее многие пользователи предпочитают задействовать особый API-интерфейс, который абстрагирует доступ к базе данных, обрабатывая все его детали в фоновом режиме. Это позволяет ускорить разработку приложений и сократить объем кода. Один из наиболее популярных API-интерфейсов такого рода называется Mongoose.

### 5.3.3. Mongoose

API-интерфейс Mongoose производства LearnBoost — это Node-модуль облегчающий работу с базой данных MongoDB. Модели в Mongoose (в концепции «модель-представление-контроллер») поддерживают интерфейс для MongoDB-коллекций, а также дополнительную полезную функциональность, включая иерархии схем, программное обеспечение промежуточного уровня и верификацию. С помощью иерархий схем обеспечивается связь моделей между собой, что позволяет, например, отправлять посты в блоги, включая связанные с ними комментарии. С помощью промежуточного программного обеспечения промежуточного уровня преобразуются данные или вызывается код при выполнении операций с данными моделей, что делает возможным решение таких

задач, как «обрезание» дочерних данных после удаления родительской информации. Благодаря поддержке верификации данных в Mongoose можно определить данные, которые будут доступны на уровне схемы, не выполняя обработку этих данных вручную.

Если вы планируете использовать Mongoose при разработке приложений, а не только в качестве хранилища данных, обратитесь к документации в Интернете, которую можно найти на веб-сайте по адресу http://mongoosejs.com/.

В этом разделе мы познакомимся с основами работы с Mongoose, включая следующие моменты:

- как открыть и закрыть соединение в MongoDB;
- как зарегистрировать схему;
- как добавить задачу;
- как искать документ;
- как обновить документ;
- как удалить документ.

Но для начала нужно установить Mongoose с помощью следующей команды:

```
npm install mongoose
```

***Открытие и закрытие соединения***

Сразу же после установки Mongoose и запуска MongoDB-сервера воспользуйтеся следующим кодом для соединения с базой данных MongoDB, которая называется tasks:

```
var mongoose = require('mongoose');
var db = mongoose.connect('mongodb://localhost/tasks');
```

Если при выполнении приложения нужно закрыть соединение с базой данных, открытое с помощью Mongoose, используйте следующий код:

```
mongoose.disconnect();
```

***Регистрация схемы***

При управлении данными с помощью Mongoose нужно зарегистрировать

специальную схему. Чтобы выполнить эту операции для планируемых задач, воспользуйтесь следующим кодом:

```
var Schema = mongoose.Schema;
var Tasks = new Schema({
  project: String,
  description: String
});
mongoose.model('Task', Tasks);
```

Схемы в Mongoose могут применяться для решения многих проблем. Помимо определения структур данных, с помощью схем можно устанавливать значения, заданные по умолчанию, обрабатывать вводимые данные и выполнять верификацию. Дополнительные сведения, относящиеся к определению схем в Mongoose, приведены на веб-сайте http://mongoosejs.com/docs/schematypes.html.

### *Добавление задачи*

Когда схема зарегистрирована, к ней можно получить доступ, начав работать с Mongoose. Следующий пример кода показывает, как добавить задачу с помощью модели:

```
var Task = mongoose.model('Task');
var task = new Task();
task.project = 'Bikeshed';
task.description = 'Paint the bikeshed red.';
task.save(function(err) {
  if (err) throw err;
  console.log('Task saved.');
});
```

### *Поиск документа*

Поиск документа с помощью Mongoose тоже не представляет особого труда. С помощью метода find модели можно искать все документы или только требуемые документы, используя JavaScript-объект для задания критерия фильтрации. В следующем примере кода осуществляется поиск задач, связанных с выбранным проектом, и для каждой задачи выводятся уникальный идентификатор и описание:

```
var Task = mongoose.model('Task');
Task.find({'project': 'Bikeshed'}, function(err, tasks) {
```

```
  for (var i = 0; i < tasks.length; i++) {
    console.log('ID:' + tasks[i]._id);
    console.log(tasks[i].description);
  }
});
```

***Обновление документа***

Хотя для изменения и последующего сохранения документа можно использовать метод find модели, однако в Mongoose модели поддерживают также метод update, специально предназначенный для обновления документов. Следующий код реализует обновления документа с помощью Mongoose:

```
var Task = mongoose.model('Task');
Task.update(
  // Обновление по внутреннему идентификатору
  {_id: '4e65b793d0cf5ca508000001'},
  {description: 'Paint the bikeshed green.'},
  // Обновление только одного документа
  {multi: false},
  function(err, rows_updated) {
    if (err) throw err;
    console.log('Updated.');
  }
);
```

***Удаление документа***

После выборки документа в Mongoose его можно легко удалить. Выбрать и удалить документ можно по его внутреннему идентификатору (либо по любому другому критерию, если вместо метода findById применять метод find). При этом используется следующий код:

```
var Task = mongoose.model('Task');
Task.findById('4e65b3dce1592f7d08000001', function(err, task) {
  task.remove();
});
```

Изучая Mongoose, вы найдете массу полезного. Этот отличный инструмент

позволяет объединить гибкость и производительность базы данных MongoDB с простотой использования, традиционно связываемой с системами управления реляционными базами данных.

### 5.4. Резюме

Изучив технологии хранения данных, мы получили базовые знания, необходимые для воплощения в жизнь универсальных сценариев применения хранилищ данных.

При создании многопользовательских веб-приложений чаще всего используются те или иные СУБД. Если вы предпочитаете SQL, задействуйте РСУБД MySQL и PostgreSQL — эти системы обладают великолепной поддержкой. Если вас не устраивают ограничения в отношении производительности и гибкости, присущие реляционным базам данных, воспользуйтесь базой данных Redis или MongoDB. MongoDB представляет собой универсальную базу данных, в то время как Redis применяется для обработки часто меняющихся и менее сложных данных.

Если вам не нужны все «прибамбасы», характерные для полномасштабных систем управления базами данным, и вы хотите избежать связанных с ними сложностей, воспользуйтесь другими возможностями. Если во главу угла вы ставите быстродействие и производительность, а сохранность данных между сеансами для вас не критична, сохраняйте данные в памяти. Если же от приложения требуется высокая производительность, а сложные запросы по обработке данных не предусмотрены (например, в приложениях командной строки), сохраняйте данные в файлах.

Не бойтесь использовать в приложениях несколько механизмов хранения данных. Например, в системе управления контентом параметры конфигурирования веб-приложения можно хранить в файлах, пользовательские истории сохранять с помощью MongoDB, а рейтинги историй — с помощью Redis. Варианты хранения данных, доступные при разработке приложений, ограничены лишь вашим воображением.

Вооружившись знаниями основ разработки веб-приложений и хранения данных, можно приступать к созданию простых веб-приложений. Теперь мы готовы перейти к тестированию, которое позволит гарантировать, что код, написанный сегодня, останется работоспособным и завтра.

## Глава 6. Среда разработки Connect

- Создание Connect-приложения
- Принципы работы программного обеспечения промежуточного уровня в Connect

- Важность порядка следования программных компонентов промежуточного уровня
- Монтирование программного обеспечения промежуточного уровня и серверов
- Создание настраиваемого программного обеспечения промежуточного уровня
- Использование программного обеспечения промежуточного уровня для обработки ошибок

Connect — это среда разработки приложений, в которой с помощью модульных компонентов, *называемых программным обеспечением промежуточного уровня* (middleware), создается многократно используемый код веб-приложений. Программный компонент промежуточного уровня в Connect представляет собой функцию, которая перехватывает объекты request (запрос) и response (ответ), предоставляемые HTTP-сервером, выполняет программную логику, а затем либо завершает ответ, либо передает управление другому программному компоненту промежуточного уровня. В Connect программные компоненты промежуточного уровня объединяются вместе с помощью *диспетчера* (dispatcher).

В Connect можно создавать собственное программное обеспечение промежуточного уровня, а также использовать обычные компоненты, реализующие в приложениях сбор данных запросов, обслуживание статических файлов, синтаксический разбор тел запросов, управление сеансами и т.п. Благодаря простоте расширения и надстройки среда Connect с точки зрения разработчиков представляет собой некий абстрактный уровень, позволяющий им создавать собственные высокоуровневые среды веб-разработки. На рис. 6.1 показано, каким образом диспетчер строит Connect-приложение и организует программное обеспечение промежуточного уровня.

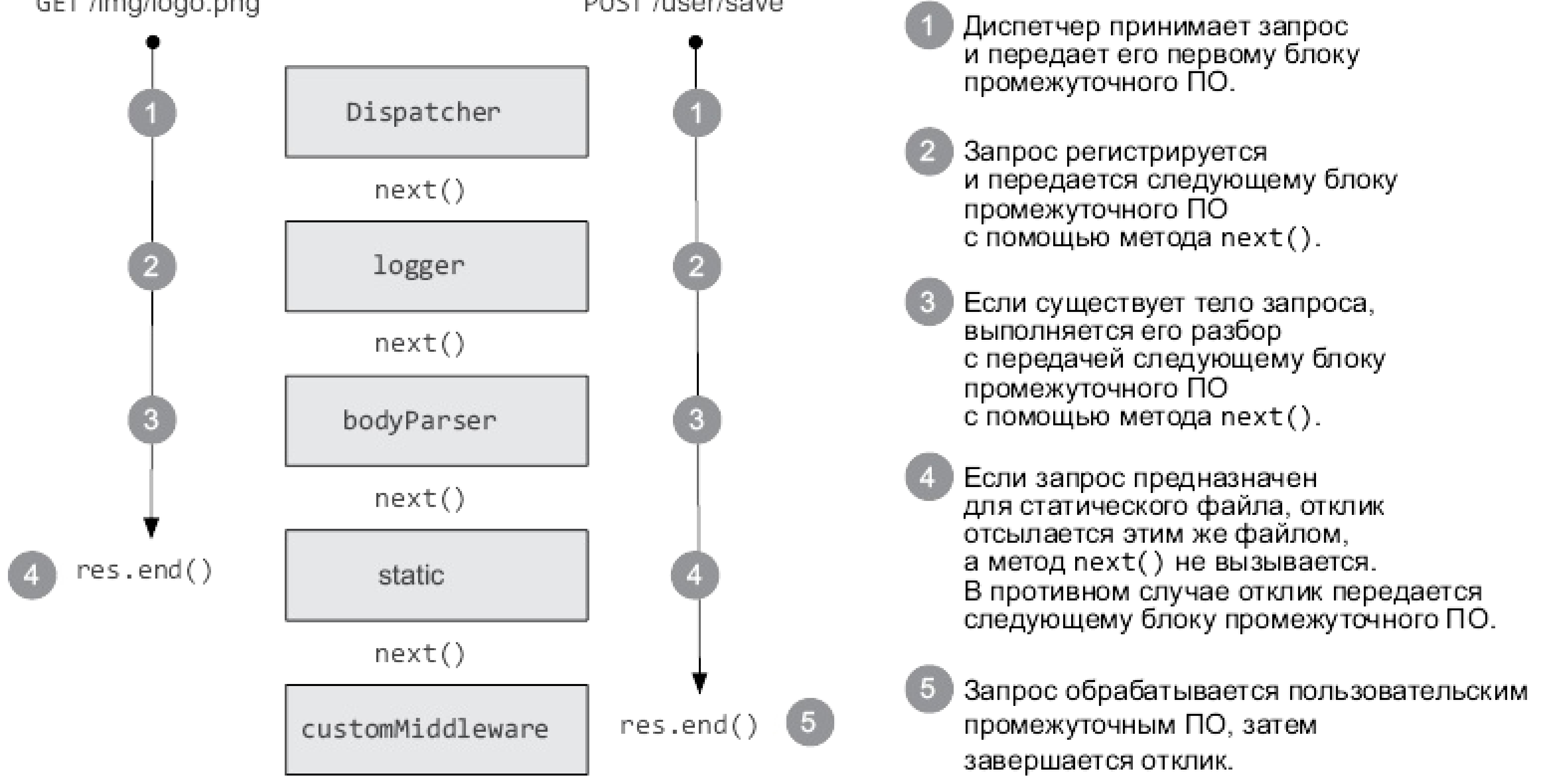



**Рис. 6.1.** Жизненный цикл двух HTTP-запросов, обрабатываемых Connect-сервером

**connect и Express**

Концепции, рассматриваемые в этой главе, непосредственно применимы и к более высокоуровневой среде разработки Express, поскольку, по сути, она представляет собой среду Connect, расширенную и надстроенную дополнительными высокоуровневыми «примочками». Прочитав эту главу, вы получите полное представление о том, как работает программное обеспечение промежуточного уровня и как объединять компоненты друг с другом при создании приложения.

В главе 8 с помощью среды Express мы будем создавать симпатичные веб-приложения с более высокоуровневым, чем в Connect, API-интерфейсом. Фактически большая часть функциональности, поддерживаемая в настоящий момент в Connect, изначально появилась в Express, однако после абстрагирования низкоуровневые «строительные блоки» остались в Connect, а выразительная высокоуровневая «начинка» была зарезервирована за Express.

Для начала давайте создадим простое Connect-приложение.

## 6.1. Создание Connect-приложения

Поскольку модуль connect разработан сторонним производителем, по умолчанию он при установке Node не устанавливается. Воспользуйтесь следующей командой, чтобы загрузить и установить среду Connect из mpm-хранилища:

```
$ npm install connect
```

Завершив установку, приступим к созданию простого Connect-приложения. Для создания такого приложения понадобится модуль connect, представляющий собой функцию, которая при вызове возвращает Connect-приложение.

В главе 4 показано, как метод http.createServer() принимает функцию обратного вызова, которая обрабатывает входящие запросы. Фактически «приложение», создаваемое в Connect, представляет собой JavaScript-функцию, которая принимает HTTP-запрос, а затем выполняет его диспетчеризацию выбранному программному компоненту промежуточного уровня.

В листинге 6.1 приведен код простого Connect-приложения. В этом приложении нет программных компонентов промежуточного уровня, поэтому диспетчер в ответ на любой принятый HTTP-запрос отвечает сообщением 404 Not Found.

**Листинг 6.1.** Простейшее Connect-приложение

```
var connect = require('connect');
var app = connect();
app.listen(3000);
```

После запуска сервера и передачи ему HTTP-запроса (с помощью команды curl или в окне веб-браузера) появится текст "Cannot GET /". Это означает, что приложение не сконфигурировано для обработки запрошенного URL-адреса. По первому примеру видно, как работает диспетчер в Connect: поочередно вызываются связанные между собой программные компоненты промежуточного уровня. Этот процесс длится до тех пор, пока один из компонентов не ответит на запрос. Если ни один из компонентов не отвечает, завершив перебор списка компонентов, приложение выводит сообщение об ошибке со статусом 404.

Теперь, создав простейшее Connect-приложение и поняв, как работает диспетчер, давайте попробуем построить приложение, делающее хоть *что-нибудь*. Чтобы создать подобное приложение, нужно определить и добавить программное обеспечение промежуточного уровня.

## 6.2. Принципы работы программного обеспечения промежуточного уровня в Connect

Программный компонент промежуточного уровня в Connect представляет собой JavaScript-функцию, которая по умолчанию принимает три аргумента. Первый аргумент — это запрос (объект request), второй аргумент — ответ (объект response),

а третий аргумент, который обычно называется next, задает функцию обратного вызова, показывающую, что компонент завершил работу и может выполняться следующий программный компонент промежуточного уровня.

Концепция программного обеспечения промежуточного уровня изначально была реализована в среде Ruby на платформе Rack. Эта среда разработки предоставляет очень простой модульный интерфейс, но из-за потоковой природы Node API-интерфейсы в Node и Rack не идентичны. Программные компоненты промежуточного уровня — это прекрасное решение, поскольку они невелики, самодостаточны и могут совместно использоваться приложениями.

В этом разделе мы познакомимся с программным обеспечением промежуточного уровня, с помощью которого к созданному в предыдущем разделе простейшему Connect-приложению добавим два простых программных компонента промежуточного уровня, которые наделят приложение новыми возможностями:

- программный компонент промежуточного уровня logger предназначен для записи на консоль данных запросов;
- программный компонент промежуточного уровня hello должен отвечать на запрос сообщением «hello world».

Начнем с создания простого программного компонента промежуточного уровня, который собирает данные запросов, поступающих на сервер.

### 6.2.1. Компонент logger

Предположим, что нужно создать файл журнала, в котором для каждого запроса, поступающего на сервер, записывать метод и URL-адрес запроса. Для решения этой задачи создадим функцию, называемую logger, которая в качестве параметров принимает объекты запроса и ответа, а также функцию обратного вызова next.

Функция next может быть вызвана программным компонентом промежуточного уровня, чтобы сообщить диспетчеру о том, что он завершил свою работу и управление можно передать следующему компоненту. Поскольку вместо возвращаемого значения используется функция обратного вызова, внутри программного компонента промежуточного уровня может быть реализована асинхронная логика, тем не менее диспетчер переходит к следующему компоненту только после завершения предыдущего. С помощью функции next() реализуется превосходный механизм обслуживания последовательности операций, выполняемых программными компонентами промежуточного уровня.

В компоненте logger вызывается метод console.log(), в качестве аргументов которого указываются метод и URL-адрес запроса, затем выводится текст "GET /user/1" и вызывается функция next(), передающая управление следующему компоненту:

```
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}
```

Итак, мы только что завершили разработку корректно функционирующего программного компонента промежуточного уровня, который для каждого полученного HTTP-запроса выводит на консоль метод и URL-адрес запроса, а затем вызывает функцию next(), возвращающую управление диспетчеру. Чтобы воспользоваться этим компонентом в приложении, вызовите метод .use(), передав его функции промежуточного уровня:

```
var connect = require('connect');
var app = connect();
app.use(logger);
app.listen(3000);
```

После отправки нескольких запросов серверу (как и раньше, с помощью команды curl или веб-браузера) на консоли вы увидите следующее:

```
GET /
GET /favicon.ico
GET /users
GET /user/1
```

Компонент сбора данных запросов — это лишь один программный компонент промежуточного уровня. Теперь нужно как-то отправить ответ клиенту. А для этого предназначен другой программный компонент промежуточного уровня.

### 6.2.2. Компонент, отвечающий сообщением «hello world»

Второй программный компонент промежуточного уровня нашего приложения будет отвечать на HTTP-запрос. Это тот же код, который используется в серверной функции обратного вызова «hello world» на домашней странице в Node:

```
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
```

Чтобы воспользоваться вторым компонентом нашего приложения, вызовите метод .use(). Количество вызовов этого метода, служащего для добавления компонентов промежуточного уровня, неограниченно.

В листинге 6.2 представлен код всего приложения. Добавление в код компонента hello приводит к тому, что сначала сервер вызывает компонент logger, который печатает текст на консоли, а затем отвечает на каждый HTTP-запрос текстом «hello world».

**Листинг 6.2.** Использование нескольких программных компонентов промежуточного уровня в Connect

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

// Вывод на консоль HTTP-метода и URL-адреса запроса и вызов метода next()
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}

// Завершение ответа на HTTP-запрос словами "hello world"
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}

connect()
.use(logger)
.use(hello)
.listen(3000);
```

В данном случае у компонента hello нет аргумента обратного вызова next. Это связано с тем, что данный компонент завершает HTTP-ответ и не нуждается в том, чтобы передавать управление обратно диспетчеру. В подобных случаях обратный вызов next не нужен. Это удобно, поскольку он соответствует сигнатуре функции обратного вызова http.createServer. Сказанное означает, что если вы уже написали код HTTP-сервера с применением модуля http, в вашем распоряжении оказывается полностью работоспособный программный компонент промежуточного уровня, который можно многократно использовать в Connect-приложении.

Функция use() возвращает экземпляр Connect-приложения, поддерживая формирование цепочки вызовов методов, как отписывалось ранее. Обратите внимание, что цепочка вызовов .use() не нужна, как показано в следующем примере кода:

```
var app = connect();
app.use(logger);
app.use(hello);
app.listen(3000);
```

Теперь, получив простое работоспособное приложение «hello world», давайте выясним, почему столь важен порядок вызовов методов .use() и каким образом этот порядок меняет поведение приложения.

## 6.3. Почему важен порядок вызова программных компонентов промежуточного уровня

Чтобы предоставить разработчикам приложений и сред разработки максимальную степень гибкости, в Connect не предусмотрен заранее заданный порядок вызова программных компонентов промежуточного уровня — вы сами может указать, в какой очередности они должны выполняться. Это очень простая концепция, но она не всегда принимается во внимание.

В этом разделе рассказывается, как порядок выполнения программного обеспечения промежуточного уровня в вашем приложении может радикально изменить его поведение. В частности, будут рассмотрены следующие вопросы:

- остановка выполнения оставшихся программных компонентов путем исключения функции next();
- использование мощного средства упорядочения программного обеспечения промежуточного уровня;
- использование программного обеспечения промежуточного уровня для идентификации.

Сначала посмотрим, как Connect работает с программным компонентом промежуточного уровня, который явно вызывает функцию next().

### 6.3.1. Когда функция next() не вызывается

В предыдущем примере приложения «hello world» сначала выполняется компонент logger, а за ним — компонент hello. В этом примере Connect записывает данные в

поток stdout, а затем отвечает на HTTP-запрос. А теперь посмотрим, что произойдет, если изменить этот порядок (листинг 6.3).

**Листинг 6.3.** Неправильно: компонент hello выполняется до компонента logger

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

// Всегда вызывается функция next() для перехода к следующему
// компоненту промежуточного уровня
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}

// Функция next() не вызывается, поскольку компонент отвечает на запрос
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}

var app = connect()
  // Компонент logger никогда не вызывается, поскольку
  // в hello нет вызова функции next()
  .use(hello)
  .use(logger)
  .listen(3000);
```

В этом примере компонент hello вызывается первым и отвечает на HTTP-запрос, как и ожидалось. Но поскольку компонент hello не вызывает функцию next(), он не возвращает управление диспетчеру, чтобы выполнить следующий программный компонент промежуточного уровня. В результате компонент logger вообще не выполняется. Если компонент промежуточного уровня не вызывает функцию next(), следующие за ним в цепочке компоненты промежуточного уровня не выполняются.

В данном случае помещение компонента hello перед компонентом logger вряд ли поможет, но при правильном использовании соответствующих методик от выбранного порядка размещения компонентов можно извлечь реальную пользу.

### 6.3.2. Выполнение аутентификации путем расположения компонентов в нужном порядке

Расположив компоненты промежуточного уровня в нужной последовательности, можно выполнять некоторые операции, например аутентификацию. Эта операция требуется почти в любом приложении. Пользователям приложения обычно нужно проходить процедуру входа (аутентификацию), а тем, кто не прошел аутентификацию, доступ к контенту блокируется. Реализовать аутентификацию нам поможет нужный порядок следования компонентов промежуточного уровня.

Предположим, что создан компонент restrictFileAccess, разрешающий доступ к файлу только аутентифицированным пользователям, которым разрешается получить доступ к следующему компоненту промежуточного уровня. Если же пользователь не проходит аутентификацию, функция next() не вызывается. В листинге 6.4 код компонента restrictFileAccess должен располагаться за кодом компонента logger, но перед кодом компонента serveStaticFiles.

**Листинг 6.4.** Ограничение доступа к файлу за счет правильного выбора порядка следования компонентов

```
var connect = require('connect');
connect()
  .use(logger)
  // Функция next() вызывается только в том случае, если пользователь
  // проходит аутентификацию
  .use(restrictFileAccess)
  .use(serveStaticFiles)
  .use(hello);
```

Теперь, когда вы получили представление о программном обеспечении промежуточного уровня и о том, какой это важный инструмент для управления прикладной логикой, рассмотрим другие средства среды Connect, помогающие использовать программное обеспечение промежуточного уровня.

## 6.4. Монтирование программного обеспечения промежуточного уровня и серверов

При создании приложений в Connect вы столкнетесь с концепцией*монтирования* (mounting). Эта концепция описывает простой, но мощный организационный инструмент, позволяющий определять префикс пути для программного обеспечения промежуточного уровня или приложений в целом. Путем монтирования можно создавать компоненты промежуточного уровня, как будто они находятся в корневой папке (значение /base свойства req.url), а затем использовать вместе с произвольным префиксом пути, не изменяя код.

Например, если компонент промежуточного уровня или сервер смонтирован в

папке /blog, указанная в коде ссылка req.url на папку /article/1 будет доступна для клиентского запроса в виде /blog/article/1. С помощью подобной методики можно обращаться к серверу блога из разных мест, не меняя код вызова. Например, если вы примете решение хранить посты в папке /articles (/articles/article/1) вместо папки /blog, понадобится лишь изменить префикс пути монтирования.

А теперь рассмотрим другой пример использования монтирования, характерный для приложений, обладающих собственными областями администрирования, такими как модерирование комментариев и принятие новых пользователей. В рассматриваемом примере область администрирования находится в папке /admin приложения. Нужно сделать так, чтобы папка /admin была доступна только пользователям, обладающим соответствующими полномочиями, а другие папки сайта были доступны всем пользователям.

Помимо перезаписи запросов из папки /base (значение /base свойства req.url), при монтировании можно вызывать программное обеспечение промежуточного уровня или приложение только в том случае, если запрос выполняется с определенным префиксом пути (точка монтирования). В листинге 6.5 второй и третий вызовы функции use() включают строку '/admin' в качестве первого аргумента, за которым следует компонент промежуточного уровня. Это означает, что следующие компоненты будут использованы только в том случае, если запрос выполняется с указанием префикса /admin. Так выглядит синтаксис, реализующий монтирование компонента промежуточного уровня или сервера в Connect.

**Листинг 6.5.** Синтаксис монтирования компонента промежуточного уровня или сервера

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


connect()
  .use(logger)
  // Если первым аргументом метода .use() является строка, Connect
  // будет вызывать программное обеспечение промежуточного уровня
  // только в случае соответствия префикса в URL-адресе
  .use('/admin', restrict)
  .use('/admin', admin)
  .use(hello)
  .listen(3000);
```

Используя технологии монтирования программного обеспечения промежуточного уровня и серверов, дополним приложение «hello world» областью администрирования. Мы будем использовать монтирование и добавим два новых

программных компонента промежуточного уровня:

- компонент restrict гарантирует, что только зарегистрированный пользователь получает доступ к странице;

- компонент admin предоставляет пользователю доступ к области администрирования.

И начнем мы с компонента, ограничивающего доступ к ресурсам пользователей, не имеющих соответствующих полномочий.

### 6.4.1. Программный компонент промежуточного уровня для аутентификации

Первый компонент промежуточного уровня должен выполнять аутентификацию. Речь идет об обобщенном компоненте аутентификации, который не привязан каким-либо образом к значению /admin свойства req.url. Однако при его монтировании в приложение компонент аутентификации будет вызываться только в том случае, если URL-адрес запроса начинается префиксом /admin. В этом случае выполняется аутентификация только тех пользователей, которые попытаются получить доступ с URL-адресом /admin; остальные пользователи входят в систему обычным образом.

В листинге 6.6 представлен «сырой» код базовой аутентификации. В этом простейшем механизме аутентификации используется поле заголовка HTTP-авторизации вместе с полномочиями, закодированными по алгоритму Base64. Дополнительные сведения о базовой аутентификации можно найти в статье Википедии по адресу http://wikipedia.org/wiki/Basic_access_authentication. Когда полномочия декодируются компонентом промежуточного уровня, имя пользователя и пароль проверяются на корректность. Если имя пользователя и пароль корректны, компонент вызывает функцию next(). Этот означает, что запрос выполнен и обработка данных может продолжаться. В противном случае генерируется ошибка.

**Листинг 6.6.** Компонент промежуточного уровня, выполняющий базовую HTTP-аутентификацию

```
function restrict(req, res, next) {
  var authorization = req.headers.authorization;
  if (!authorization) return next(new Error('Unauthorized'));

  var parts = authorization.split(' ')
```

```
  var scheme = parts[0]
  var auth = new Buffer(parts[1], 'base64').toString().split(':')
  var user = auth[0]
  var pass = auth[1];

  // Функция, проверяющая полномочия на доступ к базе данных
  authenticateWithDatabase(user, pass, function (err) {
    // Информируем диспетчера о происшедшей ошибке
    if (err) return next(err);
    // При наличии корректных полномочий вызывается функция next()
    // без аргументов
    next();
  });
}
```

И снова обратите внимание на то, что этот компонент промежуточного уровня не проверяет значение свойства req.url, пытаясь убедиться в том, что запрашивается именно папка /admin. Подобная проверка выполняется средой Connect. Благодаря этому обстоятельству можно создавать обобщенное программное обеспечение промежуточного уровня. Компонент restrict может применяться для аутентификации другой части сайта или другого приложения.

**вызов функции nexT с объектом ошибки в качестве аргумента**

Обратите внимание, как в предыдущем примере функция next вызывается с объектом ошибки, передаваемым в качестве аргумента. В результате выполнения этой операции Connect получает уведомление о том, что произошла ошибка приложения. А это означает, что в ответ на оставшуюся часть HTTP-запроса может вызываться только программное обеспечение промежуточного уровня, предназначенное для обработки ошибок. Эта разновидность программного обеспечения промежуточного уровня рассматривается в этой главе чуть позже. Ну а на данный момент вам достаточно знать, что Connect получает сообщение о том, что произошла ошибка и программное обеспечение промежуточного уровня завершает работу.

Если авторизация заканчивается без ошибок, Connect передает управление следующему компоненту промежуточного уровня, в данном случае — компоненту

admin.

### 6.4.2. Компонент, представляющий панель администрирования

Компонент промежуточного уровня admin реализует примитивный маршрутизатор с помощью инструкции switch, в качестве аргумента которой выступает URL-адрес запроса. Если в запросе содержится символ /, компонент admin выдает сообщение о перенаправлении, а если запрос имеет вид /users, он возвращает JSON-массив имен пользователей. В рассматриваемом примере имена пользователей жестко закодированы, в реальном же приложении имена пользователей обычно загружаются из базы данных.

**Листинг 6.7.** Маршрутизация запросов компонентом admin

```
function admin(req, res, next) {
  switch (req.url) {
    case '/':
      res.end('try /users');
      break;
    case '/users':
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify(['tobi', 'loki', 'jane']));
    break;
  }
}
```

Обратите внимание, что в коде используются строки / и /users, а не /admin и /admin/users. Это связано с тем, что перед вызовом программного обеспечения промежуточного уровня Connect удаляется префикс из свойства req.url. В результате URL-адреса интерпретируются так, как будто они смонтированы в папке /. Благодаря подобной простой методике приложения компоненты промежуточного уровня становятся более гибкими и могут использоваться в разных местах.

Например, благодаря монтированию приложение для блога может быть доступно как на веб-сайте http://foo.com/blog, так и на веб-сайте http://bar.com/posts. При этом не потребуется вносить какие-либо изменения в код приложения для блога или изменять URL-адреса, поскольку Connect изменяет значение свойства req.url, удаляя префикс при монтировании. То есть приложение для блога может находиться в любой папке, название которой начинается с символа /. Запросы будут использовать одни и те же компоненты промежуточного уровня и иметь одинаковое состояние. Обратите внимание на код, применяемый для установки сервера. Здесь гипотетическое приложение для блога используется

повторно за счет его монтирования в двух различных точках монтирования:

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

connect()
  .use(logger)
  .use('/blog', blog)
  .use('/posts', blog)
  .use(hello)
  .listen(3000);
```

***Тестирование***

Завершив создание компонентов промежуточного уровня, следует протестировать приложение с помощью команды curl. В процессе тестирования вы увидите, что при использовании URL-адресов, отличающихся от /admin, компонент hello выполняется так, как ожидалось:

```
$ curl http://localhost
hello world

$ curl http://localhost/foo
hello world
```

Обратите внимание, что для пользователя, не обладающего нужными полномочиями или имеющего некорректные полномочия, компонент restrict возвращает ошибку:

```
$ curl http://localhost/admin/users
Error: Unauthorized
  at Object.restrict [as handle]
  (E:\transfer\manning\node.js\src\ch7\multiple_connect.js:24:35)
  at next
  (E:\transfer\manning\node.js\src\ch7\node_modules\
    connect\lib\proto.js:190:15)
...
$ curl —user jane:ferret http://localhost/admin/users
Error: Unauthorized
  at Object.restrict [as handle]
  (E:\transfer\manning\node.js\src\ch7\multiple_connect.js:24:35)
```

```
  at next
  (E:\transfer\manning\node.js\src\ch7\node_modules\
    connect\lib\proto.js:190:15)
...
```

И наконец, обратите внимание, что только в том случае, если пользователь аутентифицирован под именем "tobi", вызывается компонент admin, а сервер отвечает JSON-массивом, содержащим имена пользователей:

```
$ curl —user tobi:ferret http://localhost/admin/users
["tobi","loki","jane"]
```

Как видите, методика монтирования проста в применении и дает превосходные результаты. А теперь рассмотрим некоторые способы создания настраиваемого программного обеспечения промежуточного уровня.

## 6.5. Создание настраиваемого программного обеспечения промежуточного уровня

Теперь, после изучения принципов создания простейших компонентов промежуточного уровня, пришло время углубиться в детали. В этом разделе мы создадим более обобщенные и многократно используемые компоненты промежуточного уровня. Одно из основных преимуществ программного обеспечения промежуточного уровня — возможность многократного использования, и в этом разделе мы создадим программное обеспечение промежуточного уровня, обладающее возможностью настройки механизмов сбора данных и маршрутизации запросов, URL-адресов и т.п. Такие компоненты можно многократно использовать в приложениях после минимальной настройки. Вам не понадобится создавать «с нуля» компоненты, требуемые для разрабатываемых приложений.

При создании программного обеспечения промежуточного уровня разработчики придерживаются простого соглашения, касающегося средств конфигурирования. В качестве подобного средства применяется функция, которая возвращает другую функцию. Это мощное средство языка JavaScript обычно называется*замыканием* (closure). Базовая структура настраиваемого программного обеспечения промежуточного уровня такого типа выглядит следующим образом:

```
function setup(options) {
  // Код настройки

  // Здесь происходит инициализация дополнительного
  // программного обеспечения промежуточного уровня
  return function(req, res, next) {
```

```
// Код программного обеспечения промежуточного уровня

// Параметры остаются доступными, даже если возвращена внешняя фун

  }
}
```

Этот тип программного обеспечения промежуточного уровня применяется следующим образом:

```
app.use(setup({some: 'options'}))
```

Обратите внимание, что функция setup вызывается в строке app.use. В предыдущих примерах в этой строке просто передавалась ссылка на функцию.

В этом разделе мы применим описанную методику для создания трех многократно используемых настраиваемых компонентов промежуточного уровня:

- компонент logger обеспечивает настройку формата печати;
- компонент router вызывает функции на основе указанных в запросе URL-адресов;
- компонент URL rewriter преобразует поля динамических данных URL-адреса і идентификаторы.

И для начала мы усовершенствуем компонент logger, сделав его настраиваемым.

### 6.5.1. Создание настраиваемого компонента logger

Созданный ранее компонент logger *не был* настраиваемым. Он был жестко закодирован и после вызова выводил на консоль значения свойств req.method и req.url, относящиеся к запросу. Что же делать в том случае, если вы захотите изменить выводимую на консоль информацию? Можно, конечно, изменить компонент logger вручную, но лучше изначально придать ему возможность настройки. Именно этим мы сейчас займемся.

Использование настраиваемого и жестко закодированного программного обеспечения промежуточного уровня практически ничем не отличается. Различие заключается только в том, что настраиваемому компоненту передается дополнительный аргумент, изменяющий его поведение. Вот как мог бы выглядеть фрагмент настраиваемого компонента logger, который принимает строку,

описывающую формат выводимой на консоль информации:

```
var app = connect()
  .use(logger(':method :url'))
  .use(hello);
```

Чтобы реализовать настраиваемый компонент logger, сначала нужно определить функцию setup, которая принимает один аргумент — строку. В рассматриваемом примере этот аргумент называется format. После вызова функции setup возвращается другая функция, которая фактически представляет собой компонент промежуточного уровня, используемый средой Connect. Возвращаемый компонент сохраняет доступ к переменной format даже после возврата из функции setup, поскольку компонент определен внутри одного и того же JavaScript-замыкания. Затем компонент logger заменяет символы в форматной строке связанными свойствами запроса для объекта req, выводит информацию в поток stdout и вызывает функцию next(), как показано в листинге 6.8.

**Листинг 6.8.** Настраиваемый компонент logger, предназначенный для среды Connect

```
// Функция setup может неоднократно вызываться
// с разными конфигурациями
function setup(format) {
  // Компонент logger с помощью переменной regexp
  // устанавливает соответствие со свойствами запроса
  var regexp = /:(\w+)/g;
  // Реальный компонент logger, который может
  // использоваться средой Connect
  return function logger(req, res, next) {
    // Форматирование журнальной записи запроса с помощью
    // переменной regexp
    var str = format.replace(regexp, function(match, property){
      return req[property];
    });
    // Вывод на консоль журнальной записи запроса
    console.log(str);
    // Передача управления следующему компоненту промежуточного уровн
    next();
  }
}
```

```
// Непосредственный экспорт функции setup компонента logger
module.exports = setup;
```

Поскольку компонент промежуточного уровня logger является настраиваемым, с помощью инструкции .use() его можно многократно использовать в одном приложении с разными вариантами настройки или в разных приложениях, которые вы создадите в будущем. Простая концепция настраиваемого программного обеспечения промежуточного уровня повсеместно применяется сообществом Connect-разработчиков. Эта же концепция используется при разработке компонентов промежуточного уровня ядра Connect с целью поддержания целостности.

В следующем разделе мы создадим чуть более сложный компонент промежуточного уровня — маршрутизатор, проецирующий входящие запросы на бизнес-логику.

### 6.5.2. Создание маршрутизирующего компонента промежуточного уровня

Маршрутизация — ключевая концепция веб-разработки. По сути, это метод проецирования URL-адресов входящих запросов на функции, реализующие бизнес-логику. На практике маршрутизация применяется в самых разных формах и масштабах. Это и находящиеся на верхнем уровне абстракции контроллеры, используемые средами разработки, такими как Rails на платформе Ruby, и менее абстрактная маршрутизация, основанная на HTTP-методах и путях, предлагаемая такими средами разработки, как Express и Sinatra на платформе Ruby.

Код простого маршрутизатора в вашем веб-приложении мог бы выглядеть так, как в листинге 6.9. В этом примере HTTP-глаголы и пути представлены в виде простого объекта и нескольких функций обратного вызова. Некоторые пути включают символы, предваряемые двоеточием (:). Эти символы представляют собой сегменты пути, которые вводит пользователь, например /user/12. В результате получается приложение с набором функций-обработчиков. Эти функции вызываются в том случае, если метод и URL-адрес запроса соответствуют одному из заданных маршрутов.

**Листинг 6.9.** Использование компонента промежуточного уровня router

```
var connect = require('connect');
// Компонент router, который определен позже в этом разделе
var router = require('./middleware/router');
  // Маршруты хранятся в виде объекта
var routes = {
```

```
  GET: {
    '/users': function(req, res){
      res.end('tobi, loki, ferret');
     },
     // Каждая запись проецируется на URL-адрес запроса
     // и содержит вызываемую функцию обратного вызова
     '/user/:id': function(req, res, id){
     res.end('user ' + id);
    }
  },
  DELETE: {
    '/user/:id': function(req, res, id){
      res.end('deleted user ' + id);
    }
  }
};
connect()
// Передача объекта routes функции настройки маршрутизации
.use(router(routes))
.listen(3000);
```

Поскольку количество компонентов промежуточного уровня в приложении неограниченно, а сами компоненты могут использоваться сколько угодно раз, можно задать несколько маршрутов в одном приложении. Это может быть полезно при решении организационных проблем. Предположим, что вы располагаете маршрутами, связанными с пользователями и администрированием. Эти маршруты можно хранить отдельно в файлах-модулях, а затем загружать по требованию для компонента router, как показано в следующем примере кода:

```
var connect = require('connect');
var router = require('./middleware/router');
connect()
  .use(router(require('./routes/user')))
  .use(router(require('./routes/admin')))
  .listen(3000);
```

На этом создание компонента router завершается. Он существенно сложнее, чем созданные ранее компоненты промежуточного уровня. Структура компонента

router представлена на рис. 6.2.

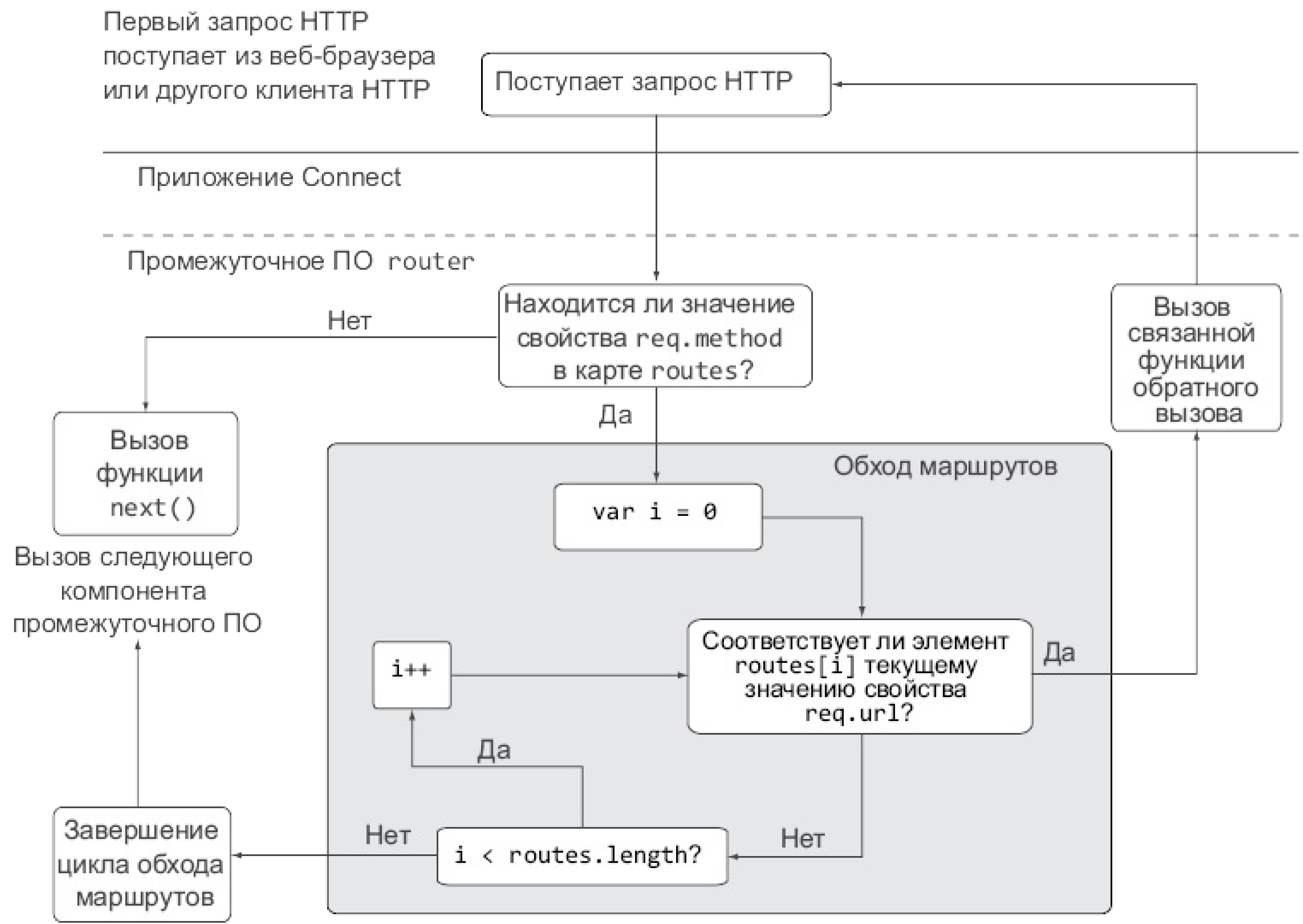



**Рис. 6.2.** Блок-схема программной логики компонента router

Блок-схема включает псевдокод, который поможет вам написать реальный код компонента router, приведенный в листинге 6.10.

**Листинг 6.10.** Простой компонент промежуточного уровня для маршрутизации

```
var parse = require('url').parse;
module.exports = function route(obj) {
  return function(req, res, next){
    // Проверяем, задано ли значение свойства req.method
    if (!obj[req.method]) {
      // Если значение свойства не определено, вызываем
      // функцию next() и прекращаем дальнейшее выполнение
      next();
      return;
    }
```

```
      // Поиск путей для метода req.method
      var routes = obj[req.method]
      // Синтаксический разбор URL-адреса для сопоставления с путем
      var url = parse(req.url)
      // Сохраняем пути из свойства req.method в качестве массива
      var paths = Object.keys(routes)

      // Циклический обход путей
      for (var i = 0; i < paths.length; i++) {
        var path = paths[i];
        var fn = routes[path];
        path = path
          .replace(/\//g, '\\/')
          .replace(/:(\w+)/g, '([^\\/]+)');
          // Строим регулярное выражение
          var re = new RegExp('^' + path + '$');
          var captures = url.pathname.match(re)
          // Пытаемся сопоставить с путем
          if (captures) {
            // Передаем группы перехваченных символов
            var args = [req, res].concat(captures.slice(1));
            fn.apply(null, args);
          // Если соответствие обнаружено, возвращаем управление,
          // чтобы предотвратить вызов следующей функции next()
          return;
        }
      }
      next();
    }
  };
```

Программный маршрутизатор, разработанный в этом разделе, представляет собой яркий пример настраиваемого программного обеспечения промежуточного уровня традиционного формата, включающего функцию настройки и возвращающего компонент промежуточного уровня, используемый в Connect-приложениях. Этот компонент принимает единственный аргумент, объект routes,

содержащий карту HTTP-глаголов, URL-адреса запросов и функции обратного вызова. Сначала выполняется проверка, позволяющая установить, определено ли текущее значение свойства req.method в карте маршрутов. Если результат проверки отрицательный, дальнейшая обработка в маршрутизаторе прекращается вызовом функции next(). Затем выполняется циклический обход заданных путей и проверяется их соответствие текущему значению свойства req.url. Если соответствие найдено, вызывается связанная функция обратного вызова, которая завершает HTTP-запрос.

Это законченный компонент промежуточного уровня, обладающий парой замечательных функциональных возможностей, но вы легко могли бы его усовершенствовать. Например, с помощью замыканий можно кэшировать регулярные выражения, которые обычно компилируются для каждого запроса.

Еще одна область применения программного обеспечения промежуточного уровня — перезапись URL-адресов. В следующем разделе рассматривается компонент промежуточного уровня, который обрабатывает поля динамических данных в URL-адресе, подставляя вместо них идентификаторы.

### 6.5.3. Создание компонента перезаписи URL-адресов

Перезапись URL-адресов может быть весьма полезной. Предположим, вам нужно принять запрос /blog/posts/my-post-title, найти идентификатор поста по завершающей части его названия, обычно называемой *полем динамических данных* (slug), и преобразовать URL-адрес таким образом, чтобы записать его в формате /blog/posts/<post-id>. Именно для решения подобных задач обычно используется программное обеспечение промежуточного уровня.

Маленькое приложение для блога в следующем примере сначала перезаписывает URL-адрес с помощью компонента rewrite, а затем передает управление компоненту showPost:

```
var connect = require('connect')
var url = require('url')
var app = connect()
  .use(rewrite)
  .use(showPost)
  .listen(3000)
```

Компонент rewrite, код которого приводится в листинге 6.11, выполняет синтаксический разбор URL-адреса, чтобы получить доступ к пути, а затем сравнивает его с регулярным выражением. Первая группа перехваченных символов (поле динамических данных) передается гипотетической функции findPostIdBySlug,

которая по полю динамических данных ищет идентификатор поста блога. Если поиск оказывается успешным, URL-адресу запроса (свойству req.url) присваивается любое выбранное вами значение. В рассматриваемом примере к пути /blog/post/ присоединяется идентификатор id, чтобы следующий компонент промежуточного уровня мог выполнить поиск поста блога по этому идентификатору.

**Листинг 6.11.** Компонент промежуточного уровня, перезаписывающий URL-адрес запроса на основе поля динамических данных

```
ar path = url.parse(req.url).pathname;

function rewrite(req, res, next) {
  var match = path.match(/^\/blog\/posts\/(.+)/)
  // Поиск только запросов /blog/posts
  if (match) {
    findPostIdBySlug(match[1], function(err, id) {
      // Если в процессе поиска произошла ошибка, информируем
      // обработчик ошибок и прекращаем работу
      if (err) return next(err);
      // Если не обнаруживается идентификатор, соответствующий полю
      // динамических данных, вызываем функцию next() с аргументом
      // Error, которому присвоено значение 'User not found'
      if (!id) return next(new Error('User not found'));
      // Перезаписываем значение свойства req.url, чтобы
      // следующий компонент промежуточного уровня
      // мог использовать реальный идентификатор
      req.url = '/blog/posts/' + id;
      next();
    });
  } else {
    next();
  }
}
```

### что показывают эти примеры

Рассмотренные в этом разделе примеры показали, что, создавая программное обеспечение промежуточного уровня, вам необходимо сосредоточиться на

небольших настраиваемых модулях. Приложение строится из множества именно таких маленьких, модульных, многократно используемых компонентов промежуточного уровня. За счет разбиения сложного приложения на небольшие фрагменты облегчаются его отладка и использование.

А теперь рассмотрим последнюю концепцию программного обеспечения промежуточного уровня в Connect — концепцию обработки ошибок в приложении.

## 6.6. Использование программного обеспечения промежуточного уровня для обработки ошибок

Ошибки присущи всем приложениям как на системном уровне, так и на уровне пользователя, поэтому очень важно умение подготовиться к возможным ошибкам, даже непредвиденным. Для обработки ошибок в Connect реализованы специальные компоненты промежуточного уровня, почти во всем подобные обычным компонентам промежуточного уровня. Отличие заключается в том, что компоненты, предназначенные для обработки ошибок, наравне с объектами запросов и ответов поддерживают объекты ошибок.

Обработка ошибок в Connect сведена к минимуму, что позволяет самому разработчику решить, как должна быть реализована система обработки ошибок. Так, с помощью программного обеспечения промежуточного уровня можно обрабатывать системные и прикладные ошибки (например, «foo is undefined»), пользовательские ошибки («password is invalid») или их комбинации. Connect позволяет выбрать тот вариант, который лучше всего подходит для вашего приложения.

В этом разделе мы рассмотрим обработку ошибок обоих типов, а также принципы, на которых основана система обработки ошибок промежуточного уровня. Также мы познакомимся с несколькими полезными методиками обработки ошибок:

- использование обработчика ошибок, предлагаемого в Connect по умолчанию;
- реализация собственной схемы обработки ошибок в приложении;
- использование нескольких компонентов промежуточного уровня, предназначенных для обработки ошибок.

Для начала давайте выясним, какая схема обработки ошибок доступна в

Connect, если не выполнять никакой дополнительной настройки.

### 6.6.1. Заданный по умолчанию обработчик ошибок в Connect

Рассмотрим следующий компонент промежуточного уровня, генерирующий ошибку ReferenceError, поскольку функция foo() не определена в приложении:

```
var connect = require('connect')



connect()
  .use(function hello(req, res) {
    foo();
    res.setHeader('Content-Type', 'text/plain');
    res.end('hello world');
  })
.listen(3000)
```

По умолчанию Connect генерирует ответ с кодом состояния 500. Тело ответа содержит текст «Internal Server Error» и дополнительные сведения, относящиеся собственно к ошибке. Этот ответ вполне информативен, но не обладает достаточной степенью гибкости. Дело в том, что в реальном приложении обычно требуется более специализированная обработка ошибок, например пересылка сообщений об ошибках демону журнала.

### 6.6.2. Собственная обработка ошибок приложения

В Connect можно также самостоятельно обрабатывать ошибки приложений, используя программное обеспечение промежуточного уровня, предназначенное для обработки ошибок. Например, в процессе разработки приложения может понадобиться генерировать клиенту ответ с представлением ошибки в формате JSON. В результате клиент получит быстро сгенерированный простой отчет. Если приложение развернуто на сервере, лучше формировать простой ответ типа «Server error», не экспонируя важную внутреннюю информацию (трассу стека, имена файлов и номера строк). Тем самым вы осложните жизнь потенциальному хакеру.

Функция промежуточного уровня, предназначенная для обработки ошибок, должна принимать четыре аргумента — err, req, res и next (листинг 6.12), в то время как обычная функция промежуточного уровня принимает только три аргумента: req, res и next.

**Листинг 6.12.** Программное обеспечение промежуточного уровня

для обработки ошибок в Connect

```
function errorHandler() {
  var env = process.env.NODE_ENV || 'development';
  // В программном обеспечении промежуточного уровня
  // для обработки ошибок определено четыре аргумента
  return function(err, req, res, next) {

    res.statusCode = 500;
    switch (env) {
      // Компонент errorHandler ведет себя по-разному
      // в зависимости от значения переменной NODE_ENV
      case 'development':
        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify(err));
        break;
      default:
        res.end('Server error');
    }
  }
}
```

**Выбирайте режим работы приложения с помощью переменной mode_env**

В соответствии с принятым в Connect соглашением переменная окружения NODE_ENV (process.env.NODE_ENV) применяется для переключения между различными средам сервера, например между средой разработки и средой выполнения.

Как только Connect обнаруживает ошибку, происходит вызов программного обеспечения промежуточного уровня, предназначенного для обработки ошибок (рис. 6.3).

Например, в разработанном ранее приложении admin в случае, если компонент, выполняющий маршрутизацию пользовательских путей, генерирует ошибку, из цепочки выполнения исключаются компоненты blog и admin. Причина заключается в том, что эти компоненты не обрабатывают ошибки — у них определено только три аргумента. Среда Connect обнаруживает, что компонент errorHandler

принимает аргумент ошибки, и вызывает этот компонент на выполнение:

```
connect()
  .use(router(require('./routes/user')))
  .use(router(require('./routes/blog')))  // пропускается
  .use(router(require('./routes/admin'))) // пропускается
  .use(errorHandler());
```

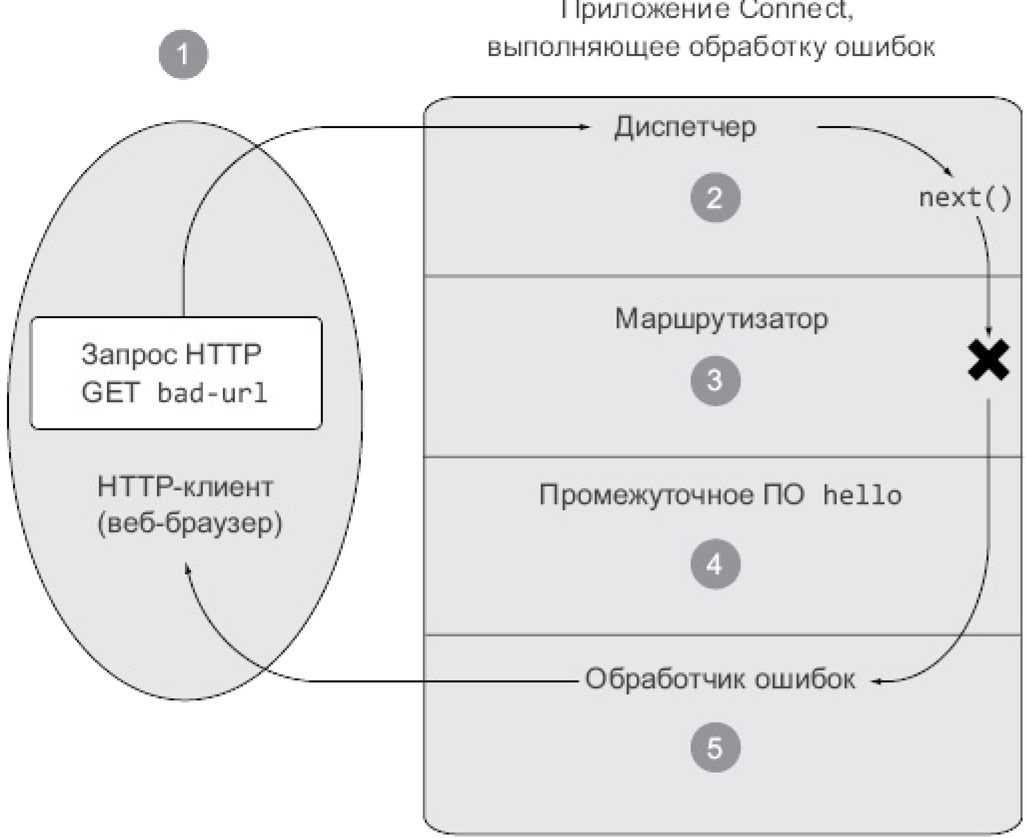



**Рис. 6.3.** Жизненный цикл HTTP-запроса, вызывающего ошибку сервера в Connect

### 6.6.3. Использование нескольких компонентов промежуточного уровня для обработки ошибок

Использование промежуточного ПО для обработки ошибок может быть полезно для раздельной обработки ошибок, если возникает такая необходимость. Предположим, что приложение включает веб-сервис, смонтированный в точке /api. Сообщения об ошибках веб-приложения, предназначенные для пользователя, нужно показывать на HTML-странице ошибок. В ответ на запросы /api нужно возвращать более подробные сведения об ошибках, например показывать их в формате JSON. Клиенты, принимающие подобные сообщения, смогут легко проанализировать ошибки и отреагировать соответствующим образом.

Чтобы посмотреть, как работает этот сценарий, создадим соответствующий код. В данном случае главное веб-приложение называется app, а сценарий api смонтирован в точке монтирования /api:

```
var api = connect()
  .use(users)
  .use(pets)
  .use(errorHandler);

var app = connect()
  .use(hello)
  .use('/api', api)
  .use(errorPage)
  .listen(3000);
```

Эта конфигурация представлена на рис. 6.4.

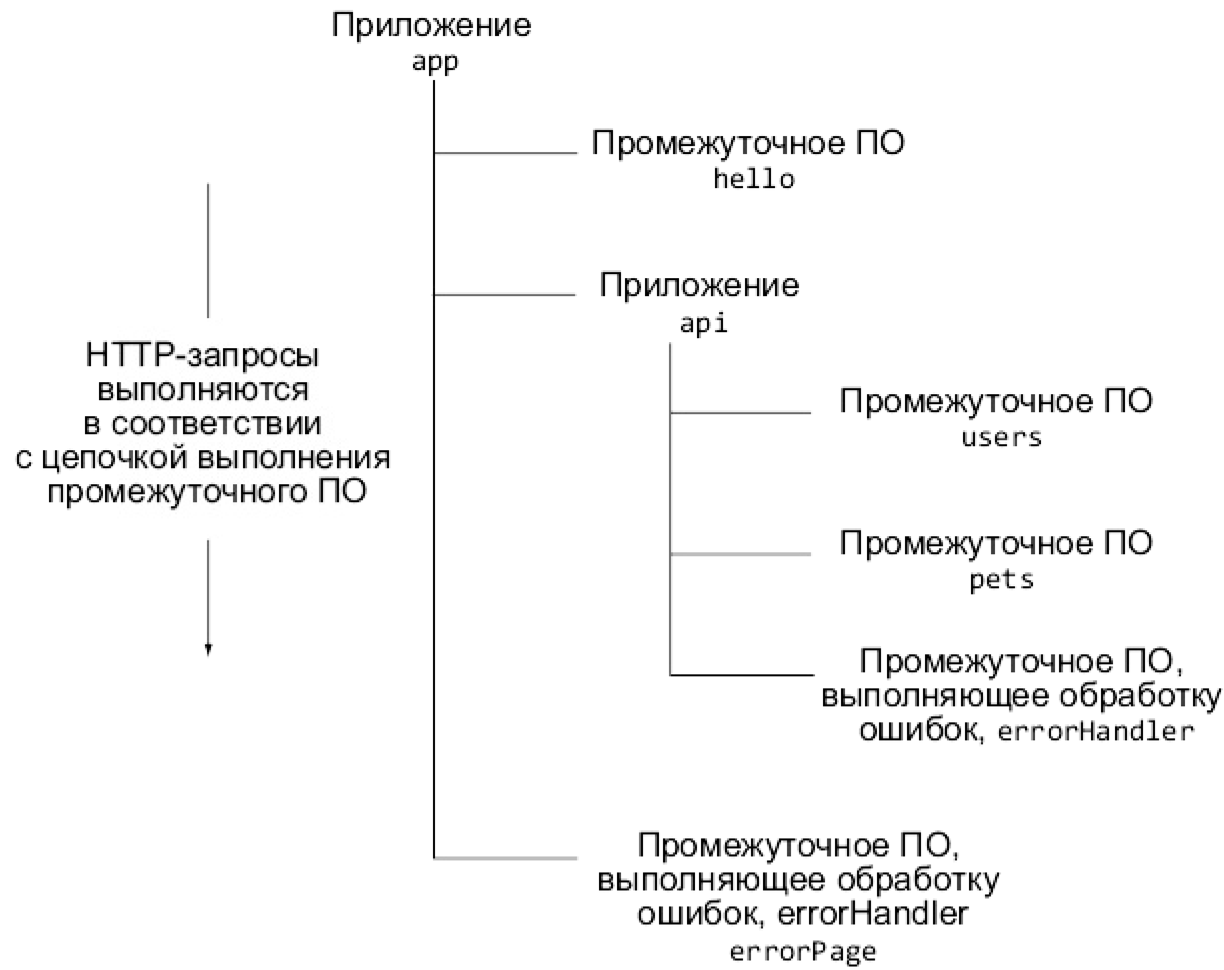



**Рис. 6.4.** Структура приложения с двумя компонентами обработки ошибок

Теперь для приложения нужно реализовать каждый компонент промежуточного уровня:

- компонент hello генерирует ответ "Hello World\n";
- компонент users генерирует ошибку notFoundError в случае отсутствия пользователя;
- компонент pets генерирует ошибку ReferenceError, демонстрируя обработку ошибок;
- компонент errorHandler предназначен для обработки любых ошибок в приложении api;
- компонент errorPage обрабатывает произвольные ошибки в главном приложении app.

### *Реализация компонента hello*

Компонент hello представляет собой функцию, которая выполняет сравнение

строки "/hello" с регулярным выражением, как показано в следующем фрагменте кода:

```
function hello(req, res, next) {
  if (req.url.match(/^\/hello/)) {
    res.end('Hello World\n');
  } else {
    next();
  }
}
```

При выполнении столь простой функции ошибки вряд ли возникнут.

### *Реализация компонента users*

Компонент users немного сложнее компонента hello. Как показано в листинге 6.13, с помощью регулярного выражения выполняется поиск соответствия для значений свойства req.url. Затем проверяется, существует ли индекс пользователя, с помощью первого набора соответствующих символов match[1]. Если пользователь найден, его имя выводится в формате JSON. Если же пользователь не найден, функции next() передается ошибка. При этом свойству not-Found присваивается значение true. Благодаря подобной методике унифицируется код обработки ошибок в компонентах, предназначенных для обработки ошибок.

**Листинг 6.13.** Компонент, выполняющий поиск пользователя в базе данных

```
var db = {
  users: [
    { name: 'tobi' },
    { name: 'loki' },
    { name: 'jane' }
  ]
};

function users(req, res, next) {
  var match = req.url.match(/^\/user\/(.+)/)
  if (match) {
    var user = db.users[match[1]];
    if (user) {
      res.setHeader('Content-Type', 'application/json');
```

```
        res.end(JSON.stringify(user));
      } else {
        var err = new Error('User not found');
        err.notFound = true;
        next(err);
      }
    } else {
      next();
    }
  }
```

***Реализация компонента pets***

Следующий фрагмент кода частично реализует компонент pets. Этот код иллюстрирует обработку ошибок на основе значений свойств, таких как булево свойство err.notFound. Этому свойству присваивается значение в компоненте users. В результате выполнения неопределенной функции foo() генерируется исключение, которое не будет иметь свойство .notFound:

```
  function pets(req, res, next) {
    if (req.url.match(/^\/pet\/(.+)/)) {
      foo();
    } else {
      next();
    }
  }
```

***Реализация компонента errorHandler***

И наконец, пришел черед компонента errorHandler. Контекстные сообщения об ошибках для веб-сервисов имеют особое значение — благодаря им веб-сервисы могут поддерживать адекватную обратную связь с пользователями, не предоставляя им слишком много информации. Вряд ли вам стоит экспонировать столь подробные сообщения об ошибках, как в строке "{"error":"foo is not defined"}", или же полную трассу стека — такая подробная информация об ошибках нужна разве что хакеру, планирующему организовать атаку на вас. Поэтому следует отвечать только такими сообщениями об ошибках, которые уберегут ваш сервер от хакерских атак. Именно это делает реализация компонента errorHandler (листинг

6.14).

**Листинг 6.14.** Компонент обработки ошибок, который не экспонирует лишних данных

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.setHeader('Content-Type', 'application/json');
  if (err.notFound) {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: err.message }));
  } else {
    res.statusCode = 500;
    res.end(JSON.stringify({ error: 'Internal Server Error' }));
  }
}
```

Этот компонент обработки ошибок использует настроенное ранее свойство err.notFound, которое предназначено для разделения ошибок сервера и клиента. При использовании другого подхода пришлось бы проверять, является ли ошибка экземпляром какой-либо другой ошибки (например, ошибки Validation-Error, относящейся к модулю верификации), а затем реагировать соответствующим образом.

При применении свойства err.notFound в случае, если сервер принимает HTTP-запрос, например /user/ronald, а сведения об этом пользователе отсутствуют в базе данных, компонент генерирует ошибку notFound. Эта ошибка передается компоненту errorHandler, что приводит к генерированию ошибки пути err.notFound. В результате возвращается код состояния 404 и свойство err.message в виде JSON-объекта. На рис. 6.5 показано сообщение об ошибке в окне веб-браузера.

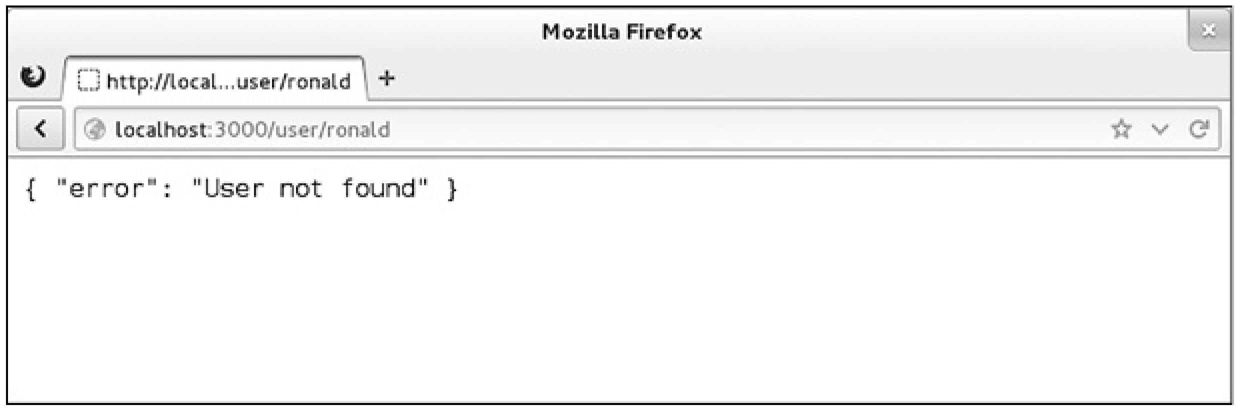



**Рис. 6.5.** JSON-объект для ошибки «User not found»

### *Реализация компонента errorPage*

Компонент errorPage — это второй компонент обработки ошибок, используемый в нашем учебном приложении. Поскольку предыдущий компонент, выполняющий обработку ошибок, никогда не вызывает функцию next(err), компонент errorPage будет выполняться только в случае ошибки, происшедшей при работе компонента hello.

Однако вероятность ошибки при выполнении компонента hello весьма мала, поэтому, скорее всего, компонент errorPage никогда не будет выполнен. Поскольку этот компонент не имеет практического значения для рассматриваемого приложения, в качестве упражнения реализуйте его самостоятельно.

На этом разработка приложения завершается. Можно запустить сервер, который уже был настроен на прослушивание порта 3000. Попробуйте воспользоваться браузером, командой curl или другим HTTP-клиентом. Всесторонне протестируйте обработчик ошибок, запрашивая некорректного пользователя либо один из объектов pets.

Еще раз подчеркнем, что обработка ошибок является *важнейшим* аспектом любого приложения. Компоненты промежуточного уровня, предназначенные для обработки ошибок, предоставляют понятный и реализованный в одном месте механизм унификации логики обработки ошибок приложения. В приложение, развернутое на сервере, следует включить как минимум один компонент промежуточного уровня, реализующий обработку ошибок.

## 6.7. Резюме

В этой главе мы познакомились с компактной, но мощной средой Connect. Мы узнали, как работает диспетчер, научились создавать программное обеспечение промежуточного уровня, которое придает приложениям модульность и гибкость. Мы освоили технологию монтирования программного обеспечения промежуточного уровня по специальному базовому URL-адресу, что позволяет создавать приложения внутри других приложений. Мы познакомились с созданием настраиваемого программного обеспечения промежуточного уровня — благодаря возможности настройки его можно многократно использовать и адаптировать под конкретные нужды. И наконец, мы научились обрабатывать ошибки, возникающие при работе программного обеспечения промежуточного уровня.

Теперь, когда фундамент заложен, пришло время освоить программное обеспечение промежуточного уровня, входящее в комплект поставки Connect. Эта тема рассматривается в следующей главе.

# Глава 7. Встроенное в Connect программное обеспечение промежуточного уровня

- Программное обеспечение промежуточного уровня, применяемое для синтаксического разбора cookie-файлов, тел обычных запросов и строк информационных запросов
- Программное обеспечение промежуточного уровня, реализующее основные потребности веб-приложений
- Программное обеспечение промежуточного уровня, предназначенное для защиты веб-приложений
- Программное обеспечение промежуточного уровня, обслуживающее статические файлы

В предыдущей главе мы познакомились с концепцией программного обеспечения промежуточного уровня, узнали о том, как создавать программные компоненты промежуточного уровня и как применять их в Connect. Однако реальная сила Connect проявляется при использовании программного обеспечения промежуточного уровня, входящего в комплект поставки этой среды разработки. С помощью этих стандартных компонентов реализуется базовая функциональность веб-приложений, включая управление сеансами, синтаксический разбор cookie-файлов и тел запросов, сбор данных запросов и т.п. Такое программное обеспечение варьируется по сложности и может стать хорошей основой для создания как простых веб-серверов, так и высокоуровневых сред веб-разработки.

В этой главе объясняется принцип работы и демонстрируется большая часть наиболее востребованных встроенных программных компонентов промежуточного уровня (табл. 7.1).

**Таблица 7.1.** Краткий справочник по встроенному в Connect программному обеспечению промежуточного уровня

| Программный компонент промежуточного уровня | Раздел книги | Описание |
|---|---|---|
| cookieParser() | 7.1.1 | Поддерживает свойства req.cookies и req.signedCookies, используемые последующими компонентами промежуточного уровня |
| bodyParser() | 7.1.2 | Поддерживает свойства req.body и req.files, используемые последующими компонентами промежуточного уровня |
| | | Ограничивает размеры тела запроса на основании заданного ограничения по длине, |

| | | |
|---|---|---|
| limit() | 7.1.3 | выраженного в байтах. Этот компонент должен находиться перед компонентом bodyParser |
| query() | 7.1.4 | Поддерживает свойство req.query, используемое последующими компонентами промежуточного уровня |
| logger() | 7.2.1 | Записывает конфигурационную информацию о входящих HTTP-запросах в поток данных, такой как stdout, или в файл журнала |
| favicon() | 7.2.2 | Отвечает на HTTP-запросы /favicon.ico. Обычно находится перед компонентом logger, поэтому его нельзя увидеть в файлах журналов |
| methodOverride() | 7.2.3 | Позволяет задействовать фиктивный метод req.method для браузеров, которые не могут использовать нужный метод. Зависит от компонента bodyParser |
| vhost() | 7.2.4 | Использует заданный компонент и/или экземпляры HTTP-сервера на основе указанного имени хоста (например, nodejs.org) |
| session() | 7.2.5 | Установление HTTP-сеанса для пользователя и предоставление долговременного объекта req.session между запросами. Зависит от компонента cookieParser |
| basicAuth() | 7.3.1 | Поддержка базовой HTTP-аутентификации для вашего приложения |
| csrf() | 7.3.2 | Защита от атак подложными межсайтовыми запросами в HTTP-формах. Зависит от сеанса |
| errorHandler() | 7.3.3 | Если происходит ошибка сервера, возвращает клиенту трассу стека. Этот компонент полезен на этапе разработки, но не на этапе использования |
| static() | 7.4.1 | Обслуживает файлы из заданной папки для HTTP-клиентов. Хорошо работает со средством монтирования Connect |
| compress() | 7.4.2 | Оптимизирует HTTP-ответы путем gzip-архивирования |
| directory() | 7.4.3 | Предоставляет списки папок для HTTP-клиентов, обеспечивая оптимальный результат на основании клиентского заголовка запроса Accept (простой текст, формат JSON или HTML) |

А начнем мы с рассмотрения программного обеспечения промежуточного уровня, реализующего различные системы синтаксического разбора (парсеры), применяемые при разработке веб-приложений. Это программное обеспечение является той основой, на которой строится большая часть других программных компонентов промежуточного уровня.

## 7.1. Программное обеспечение промежуточного уровня для синтаксического разбора cookie-файлов, тел обычных запросов и строк информационных запросов

Ядро Node не предоставляет модулей, реализующих высокоуровневую функциональность веб-приложений, такую как синтаксический разбор cookie-файлов, буферизация тел обычных запросов или синтаксический разбор строк информационных запросов, — все это имеется в Connect. В этом разделе рассматриваются четыре встроенных программных компонента промежуточного уровня, предназначенных для синтаксического разбора данных:

- cookieParser() — синтаксический разбор cookie-файлов из веб-браузеров с сохранением результатов в свойстве req.cookies;

- bodyParser() — использование и синтаксический разбор тела запроса с сохранением результатов в свойстве req.body;
- limit() — применяется совместно с bodyParser(), препятствуя чрезмерному увеличению размера запросов;
- query() — синтаксический разбор строки URL-адреса информационного запроса с сохранением результатов в свойстве req.query.

Для начала мы рассмотрим синтаксический разбор cookie-файлов, так как такие файлы часто используются в веб-браузерах для имитации данных состояния, поскольку протокол HTTP, как известно, не поддерживает данных состояния.

### 7.1.1. Компонент cookieParser() — синтаксический разбор cookie-файлов

Парсер cookie-файлов, входящий в комплект поставки Connect, может выполнять синтаксический разбор обычных cookie-файлов, подписанных cookie-файлов и специальных cookie-файлов формата JSON. По умолчанию используются обычные неподписанные cookie-файлы, при этом заполняется значениями объект req.cookies. Если же для компонента session() нужна поддержка подписанных cookie-файлов, то при создании экземпляра cookieParser() ему передается секретная строка.

**настройка cookie на сервере**

Компонент cookieParser() не предоставляет никаких вспомогательных функций для настройки исходящих cookie-файлов. Для этого нужно использовать функцию res.setHeader() с Set-Cookie в качестве заголовка. Среда Connect исправляет заданную по умолчанию Node-функцию res.setHeader() таким образом, чтобы использовать заголовки Set-Cookie специального вида. В результате функция работает так, как надо.

#### *Применение*

Секретная строка, передаваемая функции cookieParser() в качестве аргумента, используется как подписанными, так и неподписанными cookie-файлами. С помощью этого значения Connect идентифицирует факт изменения содержимого cookie-файла. Это возможно, поскольку только приложению известно секретное

значение. Обычно в качестве подобного значения используется достаточно длинная строка, которая может генерироваться случайным образом.

В следующем примере используется секретная строка 'tobi is a cool ferret':

```
var connect = require('connect');
var app = connect()
  .use(connect.cookieParser('tobi is a cool ferret'))
  .use(function(req, res){
    console.log(req.cookies);
    console.log(req.signedCookies);
    res.end('hello\n');
  }).listen(3000);
```

Свойствам req.cookies и req.signedCookies присваиваются объекты, представляющие собой разобранный заголовок Cookie, отправляемый вместе с запросом. Если вместе с запросом не передается ни одного cookie-файла, оба объекта остаются пустыми.

### *Обычные cookie-файлы*

Если бы вы выполнили несколько HTTP-запросов к серверу из предыдущего примера с помощью команды curl(1) без поля заголовка Cookie, то в результате двух вызовов console.log() на консоль был бы выведен пустой объект:

```
$ curl http://localhost:3000/
{}
{}
```

Теперь попытаемся отправить несколько cookie-файлов. Как видите, оба cookie-файла доступны в свойстве req.cookies:

```
$ curl http://localhost:3000/ -H "Cookie: foo=bar, bar=baz"
{ foo: 'bar', bar: 'baz' }
{}
```

### *Подписанные cookie-файлы*

Подписанные cookie-файлы лучше всего использовать при передаче важных данных, поскольку целостность подобных cookie-файлов может верифицироваться. В результате предотвращаются так называемые атаки через посредника. После прохождения процедуры верификации подписанные cookie-файлы помещаются в объект req.signedCookies. Необходимость в использовании двух отдельных

объектов для хранения cookie-файлов продиктована соображениями безопасности. Если хранить оба cookie-файла (подписанный и неподписанный) в одном объекте, обычный cookie-файл может быть изменен таким образом, чтобы имитировать подписанный cookie-файл.

Содержимое подписанного cookie-файла может выглядеть следующим образом:

tobi.DDm3AcVxE9oneYnbmpqxoyhyKsk

Все, что находится слева от точки (.), — это значение cookie-файла, все, что справа, — хеш-код, сгенерированный сервером по алгоритму SHA-1 в формате HMAC (Hash-based Message Authentication Code). Если Connect предпримет попытк убрать подпись cookie-файла, ничего не получится, поскольку окажется измененным значение или HMAC-код.

Предположим, например, что в вашем распоряжении имеется подписанный cookie-файл с ключом name и значением luna. Компонент cookieParser закодирует cookie-файл таким образом, что получится значение luna.PQLM0wNvqOQEObZXUkWbS5m6Wlg. Хеш-часть cookie-файла проверяется д каждого запроса, и если cookie-файл был отправлен неизмененным, он будет доступен в виде req.signedCookies.name:

```
$ curl http://localhost:3000/ -H "Cookie:
  name=luna.PQLM0wNvqOQEObZXUkWbS5m6Wlg"
{}
{ name: 'luna' }
GET / 200 4ms
```

Если же значение cookie-файла изменится, как показано в следующей команде curl, он будет доступен как req.cookies.name, поскольку cookie-файл неправильный. Тем не менее его можно использовать при отладке или для внутренних потребностей приложения:

```
$ curl http://localhost:3000/ -H "Cookie:
  name=manny.PQLM0wNvqOQEObZXUkWbS5m6Wlg"
{ name: 'manny.PQLM0wNvqOQEObZXUkWbS5m6Wlg' }
{}
GET / 200 1ms
```

### *Cookie-файлы в формате JSON*

Специальный cookie-файл в формате JSON предваряется префиксом j:, которыі распознается Connect как сериализуемый формат JSON. Cookie-файлы в формате JSON могут быть как подписанными, так и неподписанными.

Среды разработки, такие как Express, могут использовать этот формат для создания интуитивно понятного интерфейса, позволяющего разработчикам получать доступ к cookie-файлам. При этом не нужно выполнять сериализацию вручную и разбирать значения cookie-файла в формате JSON. Обратите внимание на следующий пример, демонстрирующий синтаксический разбор cookie-файлов в формате JSON:

```
$ curl http://localhost:3000/ -H 'Cookie: foo=bar,
bar=j:{"foo":"bar"}'
{ foo: 'bar', bar: { foo: 'bar' } }
{}
GET / 200 1ms
```

Как уже упоминалось, cookie-файлы в формате JSON могут быть подписанными:

```
$ curl http://localhost:3000/ -H "Cookie:
  cart=j:{\"items\":[1]}.sD5p6xFFBO/4ketA1OP43bcjS3Y"
{}
{ cart: { items: [ 1 ] } }
GET / 200 1ms
```

### *Настройка исходящих cookie-файлов*

Как уже отмечалось, компонент cookieParser() не предлагает никакой функциональности для записи заголовков исходящих cookie-файлов для HTTP-клиента с помощью заголовка Set-Cookie. В Connect же имеется явная поддержка нескольких заголовков Set-Cookie, обеспечиваемая функцией res.set-Header().

Предположим, что нужно задать для cookie-файла foo строковое значение bar. В Connect эту операцию можно выполнить с помощью единственной строки кода, в которой вызывается функция res.setHeader(). Можно также устанавливать значения различных параметров, имеющих отношение к cookie-файлу, например дату прекращения действия. Установка этого параметра выполняется с помощью второго вызова функции set-Header(), как показано в следующем примере кода:

```
var connect = require('connect');
var app = connect()
.use(function(req, res){
  res.setHeader('Set-Cookie', 'foo=bar');
  res.setHeader('Set-Cookie', 'tobi=ferret;
   Expires=Tue, 08 Jun 2021 10:18:14 GMT');
  res.end();
```

}).listen(3000);

Можно проверить заголовки, передаваемые сервером в ответ на HTTP-запрос Для выполнения этой операции используется флаг —head команды curl. Как видите, заголовки Set-Cookie устанавливаются требуемым образом:

**$ curl http://localhost:3000/ —head**

HTTP/1.1 200 OK

Set-Cookie: foo=bar

Set-Cookie: tobi=ferret; Expires=Tue, 08 Jun 2021 10:18:14 GMT

Connection: keep-alive

На этом мы завершаем рассмотрение методик, используемых при передаче cookie-файлов с HTTP-запросом. В cookie-файлах можно хранить произвольные текстовые данные, но обычно в них хранят сведения об отдельных сеансах на стороне клиента, чтобы сервер мог получать полную информацию о состоянии пользователя. Требуемый механизм инкапсулирован в компоненте session(), который тоже рассматривается в этой главе.

Еще одной задачей, которую обычно приходится решать веб-приложениям, является синтаксический разбор тела входящего запроса. В следующем разделе рассматривается программный компонент промежуточного уровня bodyParser(), который предназначен для решения этой задачи.

### 7.1.2. Компонент bodyParser() — синтаксический разбор тел запросов

При использовании веб-приложений различных типов возникает необходимость принимать данные, вводимые пользователем. Предположим, что нужно принять файл, выгружаемый с помощью HTML-тега <input type="file">. Для решения этой задачи достаточно добавить единственную строку кода, в которой вызывается компонент bodyParser(). Этот чрезвычайно полезный компонент фактически состоит из трех подкомпонентов: json(), urlencoded() и multipart().

Компонент bodyParser() поддерживает свойство req.body приложения, которое может применяться для синтаксического разбора данных в формате JSON и формирования запросов вида x-www-form-urlencoded или multipart/form-data. Если запрос имеет формат multipart/form-data, например запрос на выгрузку файла, также будет доступен объект req.files.

#### *Применение*

Предположим, что нужно принять регистрационную информацию, поступившую от приложения в JSON-запросе. Для решения этой задачи достаточно поместить

компонент bodyParser() перед любым другим компонентом, имеющим доступ к объекту req.body. Дополнительно можно использовать объект options, который будет передаваться с помощью упомянутых ранее подкомпонентов (json(), urlencoded() и multipart()):

```
var app = connect()
.use(connect.bodyParser())
.use(function(req, res){
// .. выполнение операций по регистрации пользователя ..
  res.end('Registered new user: ' + req.body.username);
});
```

***Синтаксический разбор JSON-данных***

Следующий запрос curl(1) можно использовать для отправки данных в приложение. При этом передается JSON-объект, а свойству username этого объекта присваивается значение tobi:

```
$ curl -d '{"username":"tobi"}' -H "Content-Type: application/json"
  http://localhost
Registered new user: tobi
```

***Синтаксический разбор данных regular <form> data***

Поскольку компонент bodyParser() разбирает данные на основании значения Content-Type, формат входных данных абстрагируется. Поэтому форматирование результирующего объекта данных req.body выполняется приложением.

Например, хотя следующая команда curl(1) передает данные вида x-www-form-urlencoded, компонент работает так, как нужно, не требуя каких-либо изменений кода. Как и прежде, поддерживается свойство req.body.name:

```
$ curl -d name=tobi http://localhost
Registered new user: tobi
```

***Синтаксический разбор данных multipart <form> data***

Компонент bodyParser может разбирать данные вида multipart/form-data, которые типичны для выгрузки файлов. Эти операции выполняются с помощью рассмотренного в главе 4 модуля formidable от независимого производителя.

Чтобы протестировать соответствующую функциональность, можно собрать

данные обоих объектов, req.body и req.files:

```
var app = connect()
  .use(connect.bodyParser())
  .use(function(req, res){
    console.log(req.body);
    console.log(req.files);
    res.end('thanks!');
});
```

Чтобы сымитировать загрузку файла браузером, воспользуйтесь командой curl(1) с флагом -F или –form. После выполнения этой команды ожидается ввод имени поля и значения. В следующем примере кода выполняется загрузка единственного изображения photo.png. При этом в качестве имени поля используется tobi:

```
$ curl -F image=@photo.png -F name=tobi http://localhost
thanks!
```

Если вы обратите внимание на выводимые приложением данные, то увидите нечто, подобное показанному в следующем примере. В этом примере первый объект представляет req.body, второй — req.files. Как видите, объект req.files.image.path доступен в приложении, то есть вы можете переименовать файл, находящийся на диске, передать данные сотруднику для дальнейшей обработки, загрузить данные в сеть доставки контента или выполнить другую операцию, требуемую в приложении:

```
{ name: 'tobi' }
{ image:
  { size: 4,
    path: '/tmp/95cd49f7ea6b909250abbd08ea954093',
    name: 'photo.png',
    type: 'application/octet-stream',
    lastModifiedDate: Sun, 11 Dec 2011 20:52:20 GMT,
    length: [Getter],
    filename: [Getter],
    mime: [Getter] } }
```

Теперь, завершив рассмотрение систем синтаксического разбора тел запросов, можно подумать о безопасности. Поскольку компонент bodyParser() буферизует тела запросов json и x-www-form-urlencoded в памяти, создавая одну большую строку, возникает риск атаки отказа в обслуживании. Эта атака инициируется

путем генерирования экстремально больших тел запросов в формате JSON. Чтобы предотвратить подобную атаку, нужно воспользоваться компонентом limit(). При использовании этого компонента, рассматриваемого в следующем разделе, можно задать максимальный размер тела запроса.

### 7.1.3. Компонент limit() — ограничение размера тел запросов

Выполнить синтаксический разбор тела запроса зачастую недостаточно. Разработчики также нуждаются в правильной классификации допустимых запросов и в установке относящихся к ним ограничений. С помощью программного компонента промежуточного уровня limit() можно отклонять чрезмерно большие запросы независимо от их происхождения.

Например, пользователь сервера может случайно загрузить несжатую фотографию в формате RAW размером в несколько сотен мегабайтов. Хакер может передать огромную JSON-строку, которая приведет к блокированию компонента bodyParser() и, соответственно, метода JSON.parse() из V8. Вам нужно настроить сервер так, чтобы подобных ситуаций не возникало.

#### *Зачем нужен компонент limit()?*

Рассмотрим, каким образом хакер может вывести из строя уязвимый сервер. Сначала создадим небольшое Connect-приложение под названием server.js. Функциональность этого приложения ограничена синтаксическим разбором тела запроса с помощью компонента bodyParser():

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

var app = connect()
  .use(connect.bodyParser());

app.listen(3000);
```

А теперь создадим файл dos.js, представленный в листинге 7.1. Как видите, хакер может воспользоваться HTTP-клиентом в Node, чтобы атаковать предыдущее Connect-приложение. Для организации подобной атаки достаточно написать несколько мегабайтов JSON-данных.

**Листинг 7.1.** Атаки отказа в обслуживании на уязвимый HTTP-сервер

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

var req = http.request({
```

```
    method: 'POST',
    port: 3000,
    headers: {
      // Извещение сервера о передаче JSON-данных
      'Content-Type': 'application/json'
      }
});

// Начало передачи очень большого массива
req.write('[');
var n = 300000;
while (n—) {
  // Массив содержит 300000 строк "foo"
  req.write('"foo",');
}

req.write('"bar"]');

req.end();
```

Запустите сервер и выполните сценарий атаки:

```
$ node server.js &
$ node dos.js
```

Вы увидите, что платформе V8 понадобится до 10 секунд (в зависимости от применяемого оборудования), чтобы выполнить синтаксический разбор огромной JSON-строки. Это плохо, а предотвратить подобную атаку можно с помощьк компонента limit(), который создан специально для этого.

### *Использование*

Поместив компонент limit() *перед* компонентом bodyParser(), можно задать максимальный размер тела запроса путем указания количества байтов, например 1024, или строки в одном из следующих представлений: 1gb, 25mb или 50kb.

Если с помощью компонента limit() установить предел в 32kb, а затем запустить сервер и снова выполнить сценарий атаки, вы увидите, что Connect прервет выполнение запроса, если его размер составит 32 Кбайт:

```
var app = connect()
```

```
  .use(connect.limit('32kb'))
  .use(connect.bodyParser())
  .use(hello);

http.createServer(app).listen(3000);
```

***Создание оболочки для компонента limit() с целью повышения гибкости***

Ограничивать размер каждого тела запроса небольшими величинами, такими как 32kb, нельзя в приложениях, поддерживающих выгрузку файлов пользователями, поскольку размеры большинства выгружаемых графических файлов превышают этот предел. То же самое можно сказать о размерах видеофайлов. В то же время подобное ограничение вполне разумно для тел запросов, имеющих, например, формат JSON или XML.

Если приложение должно принимать тела запросов очень разных размеров, можно поместить компонент limit() в оболочку из функции на основе конфигурации определенного типа. Например, это можно сделать таким образом, чтобы была возможность задать тип контента, как показано в листинге 7.2.

**Листинг 7.2.** Ограничение размера тела запроса на основании значения Content-Type

```
// В данном случае fn представляет собой экземпляр функции limit()
function type(type, fn) {
  return function(req, res, next){
    var ct = req.headers['content-type'] || '';
    // Возвращаемый компонент сначала проверяет значение Content-type
    if (0 != ct.indexOf(type)) {
      return next();
    }
     // Затем компонент вызывает переданный компонент limit()
    fn(req, res, next);
  }
}

var app = connect()
  .use(type('application/x-www-form-urlencoded', connect.limit('64kb')))
  // Обрабатываем формы в формате JSON
```

```
    .use(type('application/json', connect.limit('32kb')))
    // Обрабатываем выгрузку изображений размером до 2 Мбайт
    .use(type('image', connect.limit('2mb')))
    // Обрабатываем выгрузку видео размером до 300 Мбайт
    .use(type('video', connect.limit('300mb')))
    .use(connect.bodyParser())
    .use(hello);
```

Еще один способ использования этого программного компонента промежуточного уровня — задействовать параметр limit для компонента bodyParser(), а затем явным образом вызвать limit().

В следующем разделе рассматривается маленький, но очень полезный программный компонент промежуточного уровня, который выполняет синтаксический разбор строк информационных запросов, используемых в приложении.

### 7.1.4. Компонент query() — синтаксический разбор строк информационных запросов

В предыдущих разделах мы рассматривали компонент bodyParser(), который применяется при синтаксическом разборе POST-запросов для форм. Теперь пришло время заняться GET-запросами для форм. Для обработки таких запросов применяется компонент query(), который выполняет синтаксический разбор строк информационных запросов, а также предоставляет объект req.query для использования в приложении. Если вы имеете опыт разработки PHP-приложений, то найдете сходство между этим компонентом и ассоциативным массивом $_GET. Подобно bodyParser(), компонент query() следует помещать перед любыми другими программными компонентами промежуточного уровня.

#### *Применение*

В следующем приложении используется компонент query(), отвечающий JSON-представлением строки, отправленной в запросе. Строковые параметры информационного запроса обычно служат для управления видом возвращаемых данных:

```
  var app = connect()
    .use(connect.query())
    .use(function(req, res, next){
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify(req.query));
```

});

Предположим, что вы разработали приложение для музыкальной библиотеки. В этом приложения вы могли бы предложить поисковый механизм и использовать строку информационного запроса для задания параметров поиска, например:

/songSearch?artist=Bob%20Marley&track=Jammin

Подобный информационный запрос сформирует такой объект res.query:

{ artist: 'Bob Marley', track: 'Jammin' }

Компонент query() использует тот же самый компонент qs от независимого производителя, что и bodyParser(). Например:

?images[]=foo.png&images[]=bar.png

Подобная сложная строка информационного запроса сформирует следующий объект req.query:

{ images: [ 'foo.png', 'bar.png' ] }

Если в HTTP-запросе строковые параметры информационных запросов не указывать, например /songSearch, объект req.query по умолчанию будет пустым:

{}

И закончим на этом. В следующем разделе мы рассмотрим встроенное программное обеспечение промежуточного уровня, призванное удовлетворить ключевые потребности веб-приложений, такие как сбор данных и управление сеансами.

### 7.2. Программное обеспечение промежуточного уровня для реализации ключевых функций веб-приложений

В Connect имеется встроенное программное обеспечение промежуточного уровня, предназначенное для выполнения большинства типовых операций, присущих веб-приложениям, поэтому вам не придется снова и снова разрабатывать нужные модули. Ключевые функции веб-приложений, к которым относятся сбор данных, управлением сеансами и виртуальный хостинг, реализованы компонентами, входящими в комплект поставки Connect.

В этом разделе рассматриваются пять программных компонентов промежуточного уровня, которые могут применяться в приложениях:

- logger() — поддержка гибкого сбора данных запросов;
- favicon() — автоматическая обработка запроса /favicon.ico без вмешательства пользователя;

- methodOverride() — позволяет клиентам прозрачным образом перезаписывать свойство req.method;
- vhost() — установка нескольких веб-сайтов на один сервер (виртуальный хостинг);
- session() — управление данными сеанса.

Ранее мы уже создавали собственный программный компонент промежуточного уровня, предназначенный для сбора данных. Теперь пришло время познакомиться со встроенным в Connect очень гибким компонентом logger(), который предназначен для решения той же задачи.

### 7.2.1. Компонент logger() — сбор данных запросов

Компонент logger() является гибким программным компонентом промежуточного уровня, который предназначен для сбора данных запросов с возможностью настройки формата этих данных. Кроме того, он обладает средствами буферизации вывода, что уменьшает количество операций записи на диск и позволяет задать нужный поток данных, если вы решите писать данные не на консоль, а куда-нибудь в другое место, например в файл или в сокет.

***Применение***

Чтобы воспользоваться Connect-компонентом logger() в разработанном вами приложении, вызовите его как функцию, которая возвращает экземпляр компонента logger(), как показано в листинге 7.3.

**Листинг 7.3.** Использование компонента logger()

```
var connect = require('connect');
var app = connect()
  // Если аргументы не указаны, используются заданные
  // по умолчанию параметры компонента logger
  .use(connect.logger())
  // hello – это гипотетический компонент промежуточного
  // уровня, отвечающий словами "Hello World"
  .use(hello)
  .listen(3000);
```

По умолчанию компонент logger использует следующий формат, который, хотя

и слишком подробен, содержит полезную информацию о каждом HTTP-запросе, а создаваемые при этом файлы журналов напоминают файлы журналов других веб-серверов, например Apache:

```
':remote-addr - - [:date] ":method :url HTTP/:http-version" :status
  :res[content-length] ":referrer" ":user-agent"'
```

Каждый из фрагментов, предваряемый двоеточием, представляет собой *маркеры* (tokens), которые в журнальной записи заменяются реальными значениями из HTTP-запроса. Например, в результате простого запроса curl(1) генерируется примерно такая строка журнала:

```
127.0.0.1 - - [Wed, 28 Sep 2011 04:27:07 GMT]
  "GET / HTTP/1.1" 200 - "-"
  "curl/7.19.7 (universal-apple-darwin10.0)
  libcurl/7.19.7 OpenSSL/0.9.8l zlib/1.2.3"
```

### *Настройка форматов журнала*

В принципе, компонент logger() можно применять без какой-либо настройки. Но при желании можно настроить формат журнала, чтобы записывать другую информацию, сделать ее менее подробной или изменить ее вид. Чтобы настроить формат журнала, нужно передать специальную строку маркеров, например:

```
var app = connect()
  .use(connect.logger(':method :url :response-time ms'))
  .use(hello);
```

В результате применения этого формата запись журнала приобретет примерно такой вид:

```
GET /users 15 ms
```

По умолчанию доступны следующие маркеры (обратите внимание, что в названиях заголовков регистр символов не важен):

```
:req[header] ex: :req[Accept]
:res[header] ex: :res[Content-Length]
:http-version
:response-time
:remote-addr
:date
:method
:url
```

:referrer

:user-agent

:status

Определять нестандартные маркеры несложно. Для этого достаточно указать название маркера и функцию обратного вызова для функции connect.logger.token. Например, чтобы записывать каждую строку запроса, можно использовать следующий код:

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


connect.logger.token('query-string', function(req, res){
  return url.parse(req.url).query;
});
```

Компонент logger() также поддерживает другие предопределенные форматы, такие как short и tiny, которые, в отличие от заданного по умолчанию формата, приводят к выводу сокращенной информации. Альтернативным предопределенным форматом является dev, позволяющий выводить краткие сведения для разработчиков веб-приложений. Этот формат может потребоваться в тех случаях, когда, например, нужно выяснить, находится ли пользователь на веб-сайте, а детали, связанные с HTTP-запросами, вас не интересуют. В этом формате также используются цвета, соответствующие коду состояния ответа. Например, ответам с кодами состояния 2xx соответствует зеленый цвет, с кодами 3xx — синий, с кодами 4xx — желтый и с кодами 5xx — красный. Благодаря подобной цветовой схеме существенно облегчается жизнь разработчиков.

Чтобы задействовать предопределенный формат, просто передайте его название компоненту logger():

```
var app = connect()
  .use(connect.logger('dev'))
  .use(hello);
```

Теперь, когда вы знаете, как форматировать выводимые компонентом logger() данные, давайте поговорим о его параметрах.

### *Параметры сбора данных: stream, immediate и buffer*

Как уже упоминалось, с помощью параметров поведение компонента logger() можно изменить.

Один из подобных параметров, stream, позволяет передать экземпляр Node-объекта Stream, который будет использоваться вместо stdout для сбора данных. С

помощью этого параметра можно перенаправить вывод компонента logger() в собственный файл журнала независимо от вывода сервера. При этом используется экземпляр объекта Stream, созданный на основе fs.createWriteStream.

При применении этих параметров в общем случае также рекомендуется включить свойство format. В следующем примере кода задействованы нестандартный формат и журналы, находящиеся в папке /var/log/myapp.log. А благодаря флагу присоединения ('a') файл не будет обрезаться при перезагрузке приложения:

```
var fs = require('fs')
var log = fs.createWriteStream('/var/log/myapp.log', { flags: 'a' })
var app = connect()
  .use(connect.logger({ format: ':method :url', stream: log }))
  .use('/error', error)
  .use(hello);
```

Еще один полезный параметр, immediate, обеспечивает запись строки журнала, если запрос получен в первый раз. В режиме ожидания ответа запись не производится. Этот параметр рекомендуется использовать в том случае, если сервер держит запросы открытыми на протяжении длительного времени и вам известно, когда было установлено соединение. Кроме того, этот параметр можно задействовать для отладки критически важных разделов приложения. Сказанное означает, что в этом режиме нельзя применять такие маркеры, как :status и :response-time, поскольку они связаны с ответом. Чтобы перейти в данный режим, присвойте значение true параметру immediate, как показано в следующем примере кода:

```
var app = connect()
  .use(connect.logger({ immediate: true }))
  .use('/error', error)
  .use(hello);
```

Третий доступный параметр называется buffer. Он позволяет минимизировать количество обращений к диску, на котором находится файл журнала. Это может быть особенно полезно, если журнал расположен в Сети, поскольку уменьшает сетевой трафик. Параметр buffer принимает числовое значение, определяющее заданный в миллисекундах интервал между очищением буфера. А чтобы использовать интервал, заданный по умолчанию, нужно передать значение true.

На этом изучение темы сбора данных завершается. В следующем разделе мы познакомимся с программным компонентом промежуточного уровня, позволяющим управлять значками сайтов.

### 7.2.2. Компонент favicon() — обслуживание значков сайтов

**Рис. 7.1.** По умолчанию предлагается значок сайта Connect

Маленький значок сайта (favicon) отображается в адресной строке и на ярлычках вкладок браузера. Чтобы вывести такой значок, браузер запрашивает файл /favicon.ico. Обычно используются файлы значков, игнорируемые другими компонентами приложения. Компонент favicon() по умолчанию предлагает значок сайта, относящийся к среде Connect. Этому компоненту аргументы не передаются. Полученный значок сайта показан на рис. 7.1.

***Применение***

Обычно компонент favicon() используется в самой верхней части стека, поэтому при запросах значков сайта игнорируются даже запросы на сбор данных. Затем значок кэшируется в памяти, чтобы ускорить последующие ответы. В представленном далее примере кода показан компонент favicon(), запрашивающий файл custom.ico путем передачи в качестве единственного аргумента пути к файлу:

```
connect()
  .use(connect.favicon(__dirname + '/public/favicon.ico'))
  .use(connect.logger())
  .use(function(req, res) {
   res.end('Hello World!\n');
});
```

Дополнительно с помощью аргумента maxAge можно передать значение, определяющее время кэширования значка сайта в памяти.

В следующем разделе рассматривается еще один небольшой, но весьма полезный программный компонент промежуточного уровня methodOverride(), позволяющий имитировать HTTP-метод запроса в случае, если возможности клиента ограниченны.

### 7.2.3. Компонент methodOverride() — имитация HTTP-методов

Если создается веб-сервер, использующий специальные HTTP-глаголы, такие как PUT или DELETE, возникает интересная проблема. Методами <form> браузера могут быть только GET или POST, доступ к другим методам ограничен приложением.

Для использования этого компонента нужно добавить рабочее окружение <input type=hidden>, при этом в качестве значения ему присваивается название используемого метода. Далее сервер будет проверять данное значение и имитировать метод, соответствующий запросу. Компонент methodOverride() реализует серверную часть данной технологии.

***Применение***

По умолчанию вводимым HTML-именем является _method, но вы можете передать собственное значение компоненту methodOverride(), как показано в следующем примере:

```
connect()
  .use(connect.methodOverride('__method__'))
  .listen(3000)
```

**Рис. 7.2.** Использование компонента methodOverride(), имитирующего запрос методом PUT, чтобы обновить форму

Чтобы продемонстрировать результат реализации компонента methodOverride(), создадим небольшое приложение, обновляющее информацию о пользователе. Приложение включает единственную форму, в которой выводится простое сообщение об успехе в том случае, если данные формы были переданы браузером и обработаны сервером, как показано на рис. 7.2.

Приложение обновляет данные о пользователе с помощью двух разных программных компонентов промежуточного уровня. В функции update функция

next() вызывается в том случае, если метод запроса отличается от PUT. Как упоминалось ранее, большинство браузеров не поддерживают атрибут method="put" формы, поэтому приложение из листинга 7.4 работает некорректно.

**Листинг 7.4.** Некорректно работающее приложение, обновляющее сведения о пользователе

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

function edit(req, res, next) {
  if ('GET' != req.method) return next();
  res.setHeader('Content-Type', 'text/html');
  res.write('<form method="put">');
  res.write('<input type="text" name="user[name]" value="Tobi" />');
  res.write('<input type="submit" value="Update" />');
  res.write('</form>');
  res.end();
}

function update(req, res, next) {
  if ('PUT' != req.method) return next();
  res.end('Updated name to ' + req.body.user.name);
}

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.bodyParser())
  .use(edit)
  .use(update);

app.listen(3000);
```

Корректный вариант приложения, обновляющего сведения о пользователе, приводится в листинге 7.5. В этом варианте кода в форме реализован дополнительный ввод, выполняемый с помощью имени _method. Следом за компонентом bodyParser() был добавлен компонент methodOverride(), поскольку компонент bodyParser() ссылается на свойство req.body, чтобы получить доступ к данным формы.

**Листинг 7.5.** Приложение, в котором реализован компонент methodOverride(), предназначено для обновления сведений о пользователе

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

function edit(req, res, next) {
  if ('GET' != req.method) return next();
  res.setHeader('Content-Type', 'text/html');
  res.write('<form method="post">');
  res.write('<input type="hidden" name="_method" value="put" />');
  res.write('<input type="text" name="user[name]" value="Tobi" />');
  res.write('<input type="submit" value="Update" />');
  res.write('</form>');
  res.end();
}

function update(req, res, next) {
  if ('PUT' != req.method) return next();
  res.end('Updated name to ' + req.body.user.name);
}

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.bodyParser())
  .use(connect.methodOverride())
  .use(edit)
  .use(update)

.listen(3000);
```

### *Получение доступа к исходному значению свойства req.method*

Компонент methodOverride() изменяет исходное значение свойства req.method, однако вы всегда можете получить доступ к исходному методу через свойство req.originalMethod. Это означает, что предыдущая форма могла бы выводить значения так:

```
console.log(req.method);
  // "PUT"

console.log(req.originalMethod);
  // "POST"
```

На первый взгляд создается впечатление, что для создания столь простой формы приходится выполнять довольно много работы, но мы обещаем, что строить такие формы будет гораздо проще, когда вы познакомитесь с высокоуровневыми средствами среды Express (см. главу 8) и шаблонизацией (см. главу 11).

В следующем разделе рассматривается небольшой программный компонент промежуточного уровня vhost(), предназначенный для обслуживания приложений на основе имен хостов.

### 7.2.4. Компонент vhost() — виртуальный хостинг

Компонент vhost() (виртуальный хост) представляет собой простое легковесное средство маршрутизации запросов с помощью заголовка Host запроса. Эта задача обычно решается реверсным прокси-сервером, который затем направляет запрос веб-серверу, выполняющемуся на том же локальном компьютере, но на другом порту. Компонент vhost() выполняет эту операцию в том же самом Node-процессе путем передачи управления HTTP-серверу, связанному с экземпляром vhost.

#### *Применение*

Подобно остальным встроенным в Connect программным компонентом промежуточного уровня, для использования компонента vhost() достаточно единственной строки кода. Этот компонент принимает два аргумента. Первый аргумент представляет собой имя хоста, которому должен соответствовать экземпляр vhost. Второй аргумент — это экземпляр http.Server, который будет использоваться в том случае, если HTTP-запрос делается с тем же именем хоста. Поскольку все Connect-приложения являются подклассами класса http.Server, экземпляр приложения также будет работоспособным.

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

var server = connect()
var app = require('./sites/expressjs.dev');
```

```
server.use(connect.vhost('expressjs.dev', app));

server.listen(3000);
```

Чтобы воспользоваться предыдущим модулем ./sites/expressjs.dev, назначьте HTTP-сервер свойству module.exports, как показано в следующем примере:

```
var http = require('http')
module.exports = http.createServer(function(req, res){
  res.end('hello from expressjs.com\n');
});
```

***Использование нескольких экземпляров vhost()***

Подобно другим программным компонентам промежуточного уровня, компонент vhost() можно использовать многократно в приложении для проецирования нескольких хостов на соответствующие приложения:

```
var app = require('./sites/expressjs.dev');
server.use(connect.vhost('expressjs.dev', app));

var app = require('./sites/learnboost.dev');
server.use(connect.vhost('learnboost.dev', app));
```

Вместо настройки компонента vhost() вручную можно сгенерировать список хостов в файловой системе. Соответствующая методика продемонстрирована в следующем примере, в котором метод fs.readdirSync() возвращает массив записей каталога:

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

var app = connect()
var sites = fs.readdirSync('source/sites');

sites.forEach(function(site){
  console.log(' ... %s', site);
  app.use(connect.vhost(site, require('./sites/' + site)));
});

app.listen(3000);
```

Компонент vhost() намного проще в применении, чем реверсный прокси-сервер. С его помощью можно управлять всеми приложениями как одним. Подобная методика идеально подходит для обслуживания нескольких небольших сайтов или сайтов, содержащих преимущественно статический контент. Недостаток этого компонента заключается в том, что если рухнет один из сайтов, все остальные сайты тоже выйдут из строя, поскольку все они работают в рамках одного процесса.

А теперь мы рассмотрим компонент управления сеансами — один из наиболее фундаментальных программных компонентов промежуточного уровня, входящих в комплект поставки Connect. Этот компонент, который называется session(), при подписании cookie-файлов использует компонент cookieParser().

### 7.2.5. Компонент session() — управление сеансами

В главе 4 вы узнали о том, что в Node поддерживаются все средства, необходимые для реализации концепции сеансов, однако изначально они не были реализованы. Следуя генеральной философии Node, заключающейся в том, чтобы иметь небольшое ядро и большую пользовательскую надстройку, сообществом Node-разработчиков компонент управления сеансами был реализован в виде надстройки и получил название session().

Connect-компонент session() предлагает надежный, интуитивно понятный и поддерживаемый сообществом разработчиков механизм управления сеансами. При этом предоставляются разнообразные хранилища сеансов, которые ранжируются от заданного по умолчанию хранилища данных в памяти до хранилищ сеансов на основе баз данных Redis, MongoDB и CouchDB, а также cookie-файлов. В этом разделе мы рассмотрим настройку программного обеспечения промежуточного уровня, работу с данными сеанса и использование Redis-хранилища ключ/значение в качестве альтернативного сеансового хранилища.

А для начала мы поговорим о настройке программного обеспечения промежуточного уровня и доступных параметрах.

#### *Применение*

Как упоминалось ранее, для работы компонента session() требуются подписанные cookie-файлы. Поэтому сначала нужно воспользоваться компонентом cookieParser() и передать секретную строку.

В листинге 7.6 представлен пример небольшого приложения, ведущего подсчет просмотров страницы. Это приложение требует минимальной настройки, параметры компоненту session() не передаются, а для хранения данных сеанса применяется заданное по умолчанию хранилище данных в памяти. По умолчанию

именем cookie-файла является connect.sid, а его значением — httpOnly. Следовательно, клиентские сценарии не могут получить доступ к этому значению. Значения указанных параметров можно изменить, о чем рассказано чуть позже.

**Листинг 7.6.** Connect-приложение, реализующее счетчик просмотров страницы с помощью сеансов

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

var app = connect()
  .use(connect.favicon())
  .use(connect.cookieParser('keyboard cat'))
  .use(connect.session())
  .use(function(req, res, next){
  var sess = req.session;
  if (sess.views) {
    res.setHeader('Content-Type', 'text/html');
    res.write('<p>views: ' + sess.views + '</p>');
    res.end();
    sess.views++;
  } else {
    sess.views = 1;
    res.end('welcome to the session demo. refresh!');
  }
});

app.listen(3000);
```

### *Установка времени истечения сеанса*

Предположим, что нужно установить время истечения сеанса в 24 часа, разрешить передачу cookie-файла сеанса только в том случае, если используется протокол HTTPS, и задать название cookie-файла. Для этого можно передать объект так:

```
var hour = 3600000;
var sessionOpts = {
  key: 'myapp_sid',
  cookie: { maxAge: hour * 24, secure: true }
```

```
};

...
  .use(connect.cookieParser('keyboard cat'))
  .use(connect.session(sessionOpts))
...
```

В среде Connect (и в среде Express, как показано в следующей главе) часто устанавливается значение переменной maxAge, определяющей количество миллисекунд, прошедших от заданного времени. С помощью этой переменной выражение будущих дат зачастую пишется интуитивно понятнее:

```
new Date(Date.now() + maxAge)
```

Теперь, после знакомства с настройкой сеансов, мы рассмотрим методы и свойства, доступные при работе с данными сеанса.

### *Работа с данными сеанса*

Управлять данными сеанса в Connect очень просто. При этом основной принцип заключается в том, что любые свойства, присвоенные объекту req.session, сохраняются после выполнения запроса. Эти свойства загружаются, если поступают следующие запросы от того же пользователя (браузера). Например, сохранить информацию о корзине покупок столь же просто, как назначить объект свойству cart:

```
req.session.cart = { items: [1,2,3] };
```

Когда вы получаете доступ к свойству req.session.cart в последующих запросах, становится доступным массив .items. А поскольку это обычный JavaScript-объект, в последующих запросах можно вызывать методы вложенных объектов, причем эти методы сохраняются именно так, как ожидается:

```
req.session.cart.items.push(4);
```

Следует иметь в виду, что если этот объект сеанса между запросами получает сериализованные данные (в формате JSON), на объект req.session накладываются те же ограничения, что и на формат JSON. Не допускаются циклические свойства, не могут применяться объекты function, не могут корректно сериализоваться объекты Date и т.п. Учитывайте эти ограничения при использовании объекта сеанса.

В Connect данные сеанса сохраняются автоматически, при этом вызывается метод Session#save([callback]), который также доступен в виде открытого API-интерфейса. Чтобы предотвратить атаки фиксации сеанса, для аутентификации пользователей часто применяются два полезных метода — Session#destroy() и Session#regenerate(), которые доступны также при построении приложений в среде

Express, о чем рассказывается в следующих главах.

А теперь давайте поговорим о манипулировании cookie-файлами сеанса.

### *Манипулирование cookie-файлами сеанса*

В Connect для сеансов можно задать глобальные параметры cookie-файлов, но можно также манипулировать конкретным cookie-файлом с помощью объекта Session#cookie, который по умолчанию устанавливает глобальные параметры.

Прежде чем начать настраивать свойства, давайте расширим предыдущее приложение для управления сеансами, чтобы можно было просматривать свойства cookie-файлов сеанса путем записи каждого свойства в отдельные теги <p> в HTML-коде ответа:

```
...
res.write('<p>views: ' + sess.views + '</p>');
res.write('<p>expires in: ' + (sess.cookie.maxAge / 1000) + 's</p>');
res.write('<p>httpOnly: ' + sess.cookie.httpOnly + '</p>');
res.write('<p>path: ' + sess.cookie.path + '</p>');
res.write('<p>domain: ' + sess.cookie.domain + '</p>');
res.write('<p>secure: ' + sess.cookie.secure + '</p>');
...
```

В Connect разрешается изменять программным способом все свойства cookie-файлов в каждом сеансе, включая expires, httpOnly, secure, path и domain. Например, можно задать продолжительность активного сеанса в 5 секунд:

```
req.session.cookie.expires = new Date(Date.now() + 5000);
```

В качестве альтернативы для задания длительности активного сеанса можно использовать более интуитивно понятный API-интерфейс, каковым является средство доступа .maxAge, позволяющее получить и установить значение в миллисекундах относительно текущего времени. Следующий код завершает активный сеанс через 5 секунд:

```
req.session.cookie.maxAge = 5000;
```

Оставшиеся свойства, такие как domain, path и secure, ограничивают *область видимости* (scope) cookie-файла, распространяя ее только на домен, путь или защищенные соединения, в то время как httpOnly предотвращает доступ к данным cookie-файла со стороны клиентских сценариев. Эти свойства могут обрабатываться аналогичным образом:

```
req.session.cookie.path = '/admin';
req.session.cookie.httpOnly = false;
```

До сих пор для хранения данных сеанса использовалось заданное по умолчанию хранилище в памяти, поэтому давайте выясним, как подключать альтернативные хранилища данных.

### *Хранилища данных сеанса*

Встроенный объект connect.session.MemoryStore — это простое хранилище данных в памяти, которое идеально подходит для тестирования приложений, поскольку не требует учитывать другие зависимости. Однако для разработки и эксплуатации приложения нужно иметь надежную масштабируемую базу данных, обеспечивающую резервирование данных сеанса.

Несмотря на то что практически любая база данных может служить хранилищем данных сеанса, для часто меняющихся данных лучше всего подходят хранилища вида ключ/значение, которым присуще малое время отклика. Сообщество Connect-разработчиков создало несколько хранилищ данных сеанса на основе таких баз данных, как CouchDB, MongoDB, Redis, Memcached, PostgreSQL и пр.

В нашем случае мы используем базу данных Redis, доступ к которой осуществляется с помощью модуля connect-redis. В главе 5 мы уже рассматривали взаимодействие с базой данных Redis с помощью модуля node_redis. Сейчас же мы выясним, как использовать Redis для хранения данных сеанса в Connect. База данных Redis представляет собой хорошее резервное хранилище, поскольку поддерживает политику истечения срока действия ключей, обеспечивает высокую производительность и проста в установке.

Скорее всего, вы уже установили базу данных Redis на своем компьютере, когда читали главу 5. Чтобы проверить этот факт, выполните команду redis-server:

```
$ redis-server
[11790] 16 Oct 16:11:54 * Server started, Redis version 2.0.4
[11790] 16 Oct 16:11:54 * DB loaded from disk: 0 seconds
[11790] 16 Oct 16:11:54 * The server is now ready to accept
  connections on port 6379
[11790] 16 Oct 16:11:55 - DB 0: 522 keys (0 volatile) in 1536 slots HT.
```

Теперь нужно установить модуль connect-redis, добавив его в файл package.json и выполнив команду npm install. Можно также непосредственно выполнить команду npm install connect-redis. Модуль connectredis экспортирует функцию, которая должна быть передана переменной connect:

```
var connect = require('connect')
var RedisStore = require('connect-redis')(connect);
```

```
var app = connect()
  .use(connect.favicon())
  .use(connect.cookieParser('keyboard cat'))
  .use(connect.session({ store: new RedisStore({ prefix: 'sid' }) }))
...
```

Передача в переменную connect ссылки на модуль connect-redis позволит выполнить наследование из прототипа connect.session.Store.prototype. Это важно, поскольку в Node один процесс может одновременно использовать несколько версий модуля. Путем передачи указанной версии Connect можно гарантировать, что connect-redis задействует корректную копию.

Экземпляр класса RedisStore передается session() как значение store, а произвольные параметры, которые вы собираетесь использовать, например префикс key для сеансов, могут передаваться конструктору RedisStore.

На этом рассмотрение компонентов программного обеспечения промежуточного уровня, реализующих основные функции веб-приложений, завершается. В следующем разделе мы начнем знакомство с программным обеспечением промежуточного уровня, обеспечивающим безопасность веб-приложений. Это очень важно для приложений, которым требуется защищать свои данные.

### 7.3. Программное обеспечение промежуточного уровня для защиты веб-приложений

Как уже неоднократно отмечалось, API-интерфейс ядра Node преднамеренно реализован низкоуровневым. Это означает, что он не поддерживает встроенные механизмы защиты или оптимальные методики, когда используется при создании веб-приложений. К счастью, компоненты обеспечения безопасности создаваемых веб-приложений доступны в Connect.

В этом разделе рассматриваются три встроенных в Connect программных компонента промежуточного уровня, которые предназначены для обеспечения безопасности веб-приложений:

- basicAuth() — поддерживает базовую HTTP-аутентификацию защищенных данных;
- csrf() — обеспечивает защиту против атак подложными межсайтовыми запросами (Cross-Site Request Forgery, CSRF);
- errorHandler() — помощь в отладке на этапе разработки.

Сначала мы рассмотрим компонент basicAuth(), реализующий базовую HTTP-аутентификацию, что обеспечивает защиту областей приложения с ограниченным доступом.

### 7.3.1. Компонент basicAuth() — базовая HTTP-аутентификация

В разделе 6.4 главы 6 мы создали программный компонент промежуточного уровня, реализующий базовую аутентификацию. Сейчас же мы рассмотрим встроенный в Connect компонент, предназначенный для выполнения той же операции. Как упоминалось ранее, базовая аутентификация — это простейшая HTTP-аутентификация, которую следует применять осторожно, поскольку полномочия пользователя могут быть легко перехвачены хакером, если не использовать протокол HTTPS. Учитывая это, можно сказать, что базовая HTTP аутентификация может быть полезной только для реализации быстрой и не слишком надежной системы аутентификации в небольших или персональных приложениях.

Если приложение использует компонент basicAuth(), при первой попытке подключения пользователя к приложению веб-браузер запросит полномочия, как показано на рис. 7.3.

**Рис. 7.3.** Запрос полномочий при базовой аутентификации

#### *Применение*

Компонент basicAuth() предлагает три варианта верификации полномочий. Первый обеспечивает передачу имени и пароля для единственного пользователя:

```
var app = connect()
  .use(connect.basicAuth('tj', 'tobi'));
```

#### *Поддержка функции обратного вызова*

Второй вариант верификации полномочий подразумевает передачу компоненту

basicAuth() обратного вызова, который возвращает true в случае успеха. Эта методика может применяться при проверке полномочий по хеш-таблице:

```
var users = {
  tobi: 'foo',
  loki: 'bar',
  jane: 'baz'
};

var app = connect()
  .use(connect.basicAuth(function(user, pass){
     return users[user] === pass;
});
```

***Поддержка асинхронной функции обратного вызова***

Последний вариант напоминает два предыдущих, за исключением того, что в этот раз обратный вызов передается компоненту basicAuth() с тремя аргументами, определяющими асинхронный поиск (листинг 7.7). Эта методика может пригодиться при аутентификации посредством дискового файла или информационного запроса к базе данных.

**Листинг 7.7.** Connect-компонент basicAuth выполняет асинхронный поиск

```
var app = connect();

app.use(connect.basicAuth(function(user, pass, callback){
  // Выполняем функцию верификации базы данных
  User.authenticate({ user: user, pass: pass }, gotUser);
  // Запускаем асинхронный обратный вызов
  // после ответа базы данных
  function gotUser(err, user) {
    if (err) return callback(err);
    // Предоставляем обратный вызов basicAuth()
    // с объектом user из базы данных
    callback(null, user);
  }
}));
```

***Пример с командой curl(1)***

Предположим, вам нужно ограничить доступ всем запросам, поступающим на сервер. Для решения этой задачи настройте приложение следующим образом:

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

var app = connect()
  .use(connect.basicAuth('tobi', 'ferret'))
  .use(function (req, res) {
     res.end("I'm a secret\n");
   });

app.listen(3000);
```

Если теперь попытаться сгенерировать HTTP-запрос серверу с помощью команды curl(1), вы увидите сообщение о том, что запрос не авторизован:

```
$ curl http://localhost –i
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Authorization Required"
Connection: keep-alive
Transfer-Encoding: chunked

Unauthorized
```

Выполнение того же запроса с полномочиями базовой HTTP-аутентификации (обратите внимание на начало URL-адреса) обеспечит доступ:

```
$ curl —user tobi:ferret http://localhost –i
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 22:42:06 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 22:41:02 GMT
ETag: "13-1318804862000"
Content-Type: text/plain; charset=UTF-8
Accept-Ranges: bytes
Content-Length: 13
Connection: keep-alive
```

I'm a secret

Продолжая тему безопасности, рассмотрим программный компонент промежуточного уровня csrf(), призванный помочь защититься от атак подложными межсайтовыми запросами.

### 7.3.2. Компонент csrf() — защита от атак подложными межсайтовыми запросами

При атаке подложными межсайтовыми запросами (Cross-Site Request Forgery, CSRF), по сути, эксплуатируется доверие веб-браузеру со стороны сайта. В ходе такой атаки аутентифицированный пользователь вашего приложения посещает другой сайт, созданный или скомпрометированный организатором атаки, после чего от имени пользователя выполняются подложные запросы, а он даже не подозревает об этом.

Эта атака организована довольно сложным образом, поэтому разбираться в ее принципах лучше на основе примера. Предположим, что в вашем приложении запрос DELETE /account вызывает удаление учетной записи пользователя (во время подключения пользователя к серверу). Предположим, что пользователь посетил форум, уязвимый для CSRF-атаки. Организатор атаки может передать сценарий, который генерирует запрос DELETE /account и удаляет учетную запись пользователя. Избежать подобной неприятности можно с помощью компонента csrf().

Компонент csrf() генерирует 24-символьный уникальный идентификатор, так называемый *маркер аутентичности* (authenticity token), и присваивает его сеансу пользователя в виде req.session._csrf. Затем этот маркер может быть включен в вводимые в форму данные в качестве скрытого значения под названием _csrf, и CSRF-компонент может верифицировать маркер при подписании. Этот процесс повторяется для каждого взаимодействия.

#### *Применение*

Чтобы убедиться в том, что компонент csrf() может получить доступ к req.body._csrf (скрытое значение вводимых данных) и req.session._csrf, следует поместить csrf() после компонентов body-Parser() и session(), как показано в следующем примере кода:

```
connect()
  .use(connect.bodyParser())
  .use(connect.cookieParser('secret'))
  .use(connect.session())
```

```
.use(connect.csrf());
```

Другой аспект веб-разработки заключается в обеспечении доступа к подробным журналам и детализированным отчетам об ошибках в среде разработки и в рабочей среде. Рассмотрим программный компонент промежуточного уровня errorHandler(), предназначенный для решения этих задач.

### 7.3.3. Компонент errorHandler() — обработка ошибок при разработке

Компонент errorHandler(), связанный с Connect, идеально подходит для разработки, предлагая подробные ответы с информацией об ошибках в форматах HTML, JSON и простого текста на базе поля Accept заголовка. Это означает, что данный компонент применяется только при разработке и не должен являться частью рабочей конфигурации.

#### *Применение*

Обычно компонент errorHandler() должен располагаться последним, поскольку предназначен для перехвата всех ошибок:

```
var app = connect()
  .use(connect.logger('dev'))
  .use(function(req, res, next){
    setTimeout(function () {
      next(new Error('something broke!'));
    }, 500);
  })

.use(connect.errorHandler());
```

#### *Получение HTML-ответа об ошибке*

Если просмотреть любую страницу в окне браузера, для которой был выполнен показанный здесь код, вы увидите Connect-страницу ошибки, подобную показанной на рис. 7.4. На ней выводится сообщение об ошибке, состояние ответа и полная трасса стека.

#### *Получение ответа об ошибке в формате простого текста*

Предположим, что мы тестируем встроенный в Connect API-интерфейс. Ответы

генерируемые в виде больших фрагментов HTML-кода, далеки от идеала, поэтому по умолчанию компонент errorHandler() отвечает в формате text/plain, который хорошо подходит для HTTP-клиентов командой строки, таких как curl(1). Эта концепция иллюстрируется следующим выводом:

```
$ curl http://localhost/
Error: something broke!
  at Object.handle (/Users/tj/Projects/node-in-action/source
    /connect-middleware-errorHandler.js:12:10)
  at next (/Users/tj/Projects/connect/lib/proto.js:179:15)
  at Object.logger [as handle] (/Users/tj/Projects/connect
    /lib/middleware/logger.js:155:5)
  at next (/Users/tj/Projects/connect/lib/proto.js:179:15)
  at Function.handle (/Users/tj/Projects/connect/lib/proto.js:192:3)
  at Server.app (/Users/tj/Projects/connect/lib/connect.js:53:31)
  at Server.emit (events.js:67:17)
  at HTTPParser.onIncoming (http.js:1134:12)
  at HTTPParser.onHeadersComplete (http.js:108:31)
  at Socket.ondata (http.js:1029:22)
```

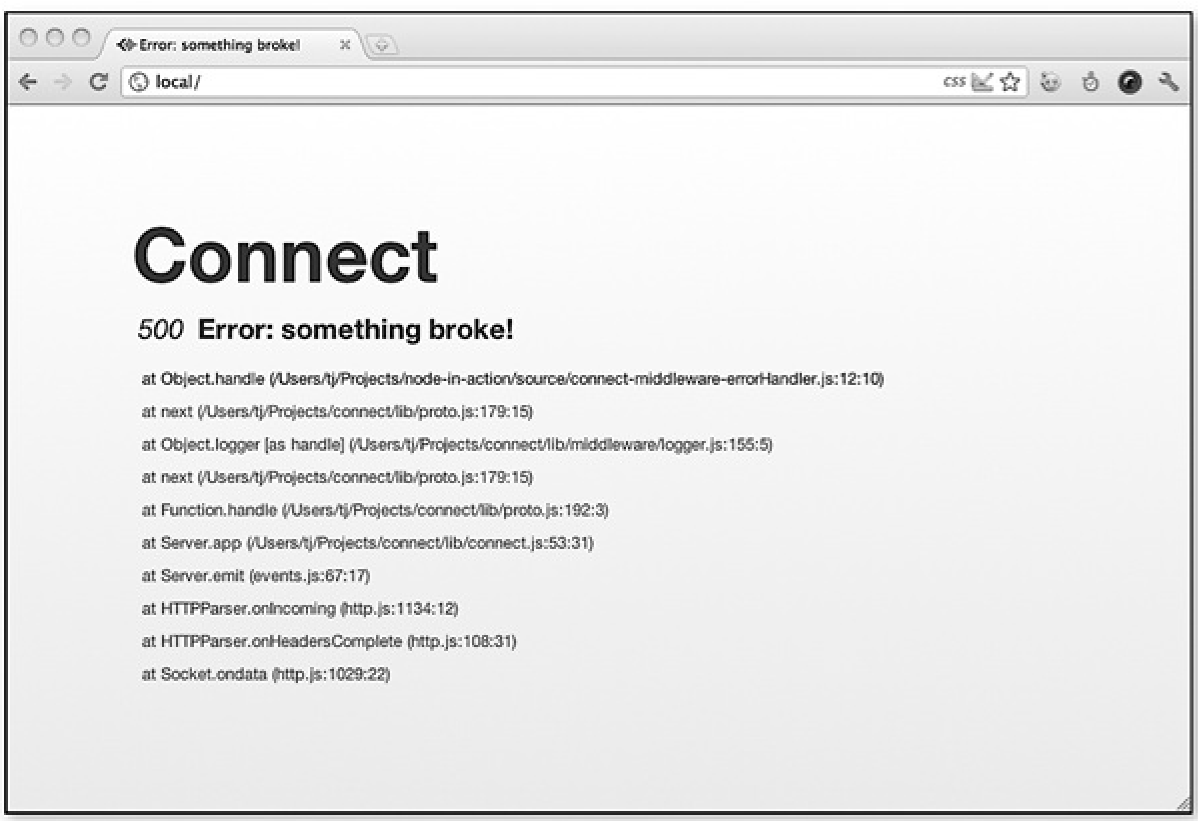



**Рис. 7.4.** Заданный по умолчанию Connect-компонент errorHandler() в окне веб-браузера

*Получение JSON-ответа об ошибке*

Если вы передаете HTTP-запрос, который включает HTTP-заголовок Accept application/json, то получите следующий ответ в формате JSON:

```
$ curl http://localhost/ -H "Accept: application/json"
{"error":{"stack":"Error: something broke!\n
    at Object.handle (/Users/tj/Projects/node-in-action
    /source/connect-middleware-errorHandler.js:12:10)\n
    at next (/Users/tj/Projects/connect/lib/proto.js:179:15)\n
    at Object.logger [as handle] (/Users/tj/Projects
    /connect/lib/middleware/logger.js:155:5)\n
    at next (/Users/tj/Projects/connect/lib/proto.js:179:15)\n
    at Function.handle (/Users/tj/Projects/connect/lib/proto.js:192:3)\n
    at Server.app (/Users/tj/Projects/connect/lib/connect.js:53:31)\n
    at Server.emit (events.js:67:17)\n
    at HTTPParser.onIncoming (http.js:1134:12)\n
    at HTTPParser.onHeadersComplete (http.js:108:31)\n
    at Socket.ondata (http.js:1029:22)","message":"something broke!"}}
```

Мы добавили дополнительное форматирование в JSON-ответ, что позволило упростить его чтение на странице, но когда Connect передает JSON-ответ, он уплотняется с помощью функции JSON.stringify().

Чувствуете ли вы теперь себя профессионалом в вопросах безопасности? Вероятно, нет, тем не менее вы знаете уже достаточно, чтобы обеспечить безопасность своих приложений с помощью встроенных в Connect программных компонентов промежуточного уровня. А теперь мы рассмотрим функцию, используемую во всех веб-приложениях и обеспечивающую обслуживание статических файлов.

## 7.4. Программное обеспечение промежуточного уровня для обслуживания статических файлов

Еще одной функцией, которая не реализована в ядре Node, но которая требуется многим веб-приложенияем, является обслуживание статических файлов. К счастью, нужная функциональность реализована в Connect.

В этом разделе мы рассмотрим три встроенных в Connect программных компонента промежуточного уровня, выполняющих обслуживание статических файлов файловой системы подобно тому, как это делают обычные HTTP-серверы:

- static() — обслуживает файлы, находящиеся в заданной папке root файловой системы;

- compress() — сжимает запросы и может применяться вместе с компонентом static();

- directory() — выводит содержимое папки по запросу.

Сначала давайте рассмотрим, каким образом можно обслуживать статические файлы с помощью единственной строки кода, в которой используется компонент static.

### 7.4.1. Компонент static() — обслуживание статических файлов

Встроенный в Connect компонент static() реализует высокопроизводительный, гибкий и многофункциональный статический файловый сервер, поддерживающий механизмы HTTP-кэша, запросы Range и т.п. И что более важно, этот компонент обеспечивает проверку безопасности для опасных путей, по умолчанию запрещает доступ к скрытым файлам (начинающимся с а .) и отклоняет вредоносные нулевые байты. В сущности, static() является очень надежным и хорошо управляемым компонентом обслуживания статических файлов, совместимым с различными HTTP-клиентами.

***Применение***

Предположим, что приложение следует типичному сценарию обслуживания статических ресурсов, находящихся в папке ./public. Доступ к этому приложению осуществляется с помощью единственной строки кода:

```
app.use(connect.static('public'));
```

В этой конфигурации компонент static() проверяет обычные файлы, которые находятся в папке ./public/, выбранной по URL-адресу запроса. Если файл существует, значение поля Content-Type ответа будет по умолчанию определяться расширением файла и передаваемыми данными. Если запрошенный путь не представляет файл, вызывается функция next(), которая реализует переход к следующему (если таковой имеется) программному компоненту промежуточного уровня для обработки запроса.

Чтобы протестировать компонент, создайте файл ./public/foo.js с вызовом console.log('tobi') и сгенерируйте запрос для сервера. Для этого используйте команду curl(1) с флагом -i, задающим печать HTTP-заголовков. Вы увидите, что

поля HTTP-заголовка, связанные с кэшем, получают требуемые значения, поле Content-Type отражает заданное расширение .js и контент передается успешно:

```
$ curl http://localhost/foo.js –i
HTTP/1.1 200 OK
Date: Thu, 06 Oct 2011 03:06:33 GMT
Cache-Control: public, max-age=0
Last-Modified: Thu, 06 Oct 2011 03:05:51 GMT
ETag: "21-1317870351000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Length: 21
Connection: keep-alive
console.log('tobi');
```

Поскольку путь в запросе используется в исходном виде, файлы в папках обслуживаются требуемым образом. Например, при запросах к серверу GET /javascripts/jquery.js и GET /stylesheets/app.css сервер будет обслуживать соответственно файлы ./public/javascripts/jquery.js и ./public/stylesheets/app.css.

### *Использование компонента static() с монтированием*

Иногда название приложения предваряется путем вида /public, /assets, /static и т.д. С учетом реализованной в Connect концепции монтирования упрощается обслуживание статических файлов, находящихся в разных папках. Для этого просто смонтируйте приложение в нужном месте. Как упоминалось в главе 6, программное обеспечение промежуточного уровня «не знает», где оно смонтировано, поскольку префикс пути удаляется.

Например, запрос GET /app/files/js/jquery.js с помощью компонента static(), смонтированного в точке /app/files, передается программному обеспечению промежуточного уровня в виде GET /js/jquery. Это хорошо работает с функциональностью префиксов, поскольку точка /app/files перестает быть частью разрешения файла:

```
app.use('/app/files', connect.static('public'));
```

Исходный запрос GET /foo.js в данном случае не сработает, поскольку программное обеспечение промежуточного уровня не вызывается до тех пор, пока присутствует точка монтирования, зато передачу файла обеспечит префиксная версия запроса GET /app/files/foo.js:

```
$ curl http://localhost/foo.js
```

```
Cannot get /foo.js

$ curl http://localhost/app/files/foo.js
console.log('tobi');
```

### *Абсолютный и относительный пути к папке*

Имейте в виду, что путь, переданный компоненту static(), для текущей рабочей папки является относительным. Это означает, что передача значения "public" в качестве пути разрешается, по сути, в виде process.cwd() + "public".

Иногда приходится использовать относительные пути, когда задается базовая папка. Для этого служит переменная __dirname, которая используется следующим образом:

```
app.use('/app/files', connect.static(__dirname + '/public'));
```

### *Обслуживание файла index.html при запросе папки*

Еще одно полезное свойство компонента static() заключается в его возможности обслуживать файлы index.html. Если запрашивается папка, в которой находится файл index.html, будет обслужен этот файл.

Теперь, когда вы научились обслуживать статические файлы с помощью единственной строки кода, давайте выясним, каким образом можно сжать данные ответа, используя программный компонент промежуточного уровня compress(), позволяющий сократить объем передаваемых данных.

## 7.4.2. Компонент compress() — сжатие статических файлов

С модулем zlib разработчик получает в свое распоряжение механизмы компрессии и декомпрессии данных с помощью алгоритмов gzip и deflate. Connect версии 2.0 и выше предлагает модуль zlib на уровне HTTP-сервера для компрессии исходящих данных, которую обеспечивает компонент compress().

Компонент compress() автоматически обнаруживает допустимые кодировки с помощью поля Accept-Encoding заголовка. Если это поле не содержит значений, используется идентичная кодировка, означающая, что ответ не сжимается. Если же поле содержит значение gzip или deflate либо оба этих значения, ответ будет сжат.

### *Применение*

Обычно Connect-компонент compress() добавляется в верхнюю часть стека, поскольку он является оболочкой методов res.write() и res.end().

В следующем примере обслуживаемые статические файлы будут поддерживать сжатие:

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

var app = connect()
  .use(connect.compress())
  .use(connect.static('source'));

app.listen(3000);
```

В показанном далее фрагменте кода обслуживается маленький 189-байтовый JavaScript-файл. По умолчанию команда curl(1) не передает поле Accept-Encoding, поэтому вы получите простой текст:

```
$ curl http://localhost/script.js –i
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 18:30:00 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 18:29:55 GMT
ETag: "189-1318789795000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Length: 189
Connection: keep-alive

console.log('tobi');
console.log('loki');
console.log('jane');
console.log('tobi');
console.log('loki');
console.log('jane');
console.log('tobi');
console.log('loki');
console.log('jane');
```

Следующая команда curl(1) добавляет поле Accept-Encoding, показывающее, что

вы хотите принять данные, сжатые по алгоритму gzip. Как видите, даже в случае столь небольшого файла степень сжатия передаваемых данных довольно ощутима, поскольку этот файл содержит повторяющиеся данные:

```
$ curl http://localhost/script.js -i -H "Accept-Encoding: gzip"
HTTP/1.1 200 OK
Date: Sun, 16 Oct 2011 18:31:45 GMT
Cache-Control: public, max-age=0
Last-Modified: Sun, 16 Oct 2011 18:29:55 GMT
ETag: "189-1318789795000"
Content-Type: application/javascript
Accept-Ranges: bytes
Content-Encoding: gzip
Vary: Accept-Encoding
Connection: keep-alive
Transfer-Encoding: chunked
K??+??I???O?P/?O?T??JF?????J?K???v?!?_?
```

Попробуйте протестировать этот же пример с параметром Accept-Encoding: deflate.

### *Использование нестандартной функции фильтрации*

По умолчанию компонент compress() поддерживает MIME-типы text/*, */json и */javascript, как определено в функции filter, заданной по умолчанию:

```
exports.filter = function(req, res){
  var type = res.getHeader('Content-Type') || '';
  return type.match(/json|text|javascript/);
};
```

Чтобы изменить это поведение, можно передать filter в объекте параметров компонента compress(), как показано в следующем фрагменте кода, который сжимает только обычный текст:

```
function filter(req) {
  var type = req.getHeader('Content-Type') || '';
  return 0 == type.indexOf('text/plain');
}
connect()
  .use(connect.compress({ filter: filter }))
```

### *Задание уровней сжатия и расходования памяти*

Привязки Node-модуля zlib поддерживают параметры производительности и сжатия, которые могут передаваться функции compress().

В следующем примере кода переменной level присваивается значение 3, что соответствует невысокой степени сжатия при высокой производительности, а переменной memLevel — значение 8, обеспечивающее ускоренное сжатие за счет большего расходования памяти. Эти значения полностью зависят от разрабатываемого приложения и доступных ресурсов (за дополнительными сведениями обратитесь к документации к Node-модулю zlib):

```
connect()
  .use(connect.compress({ level: 3, memLevel: 8 }))
```

А теперь давайте рассмотрим программный компонент промежуточного уровня directory(), который помогает компоненту static() обслуживать папки и показывать их содержимое в любых форматах.

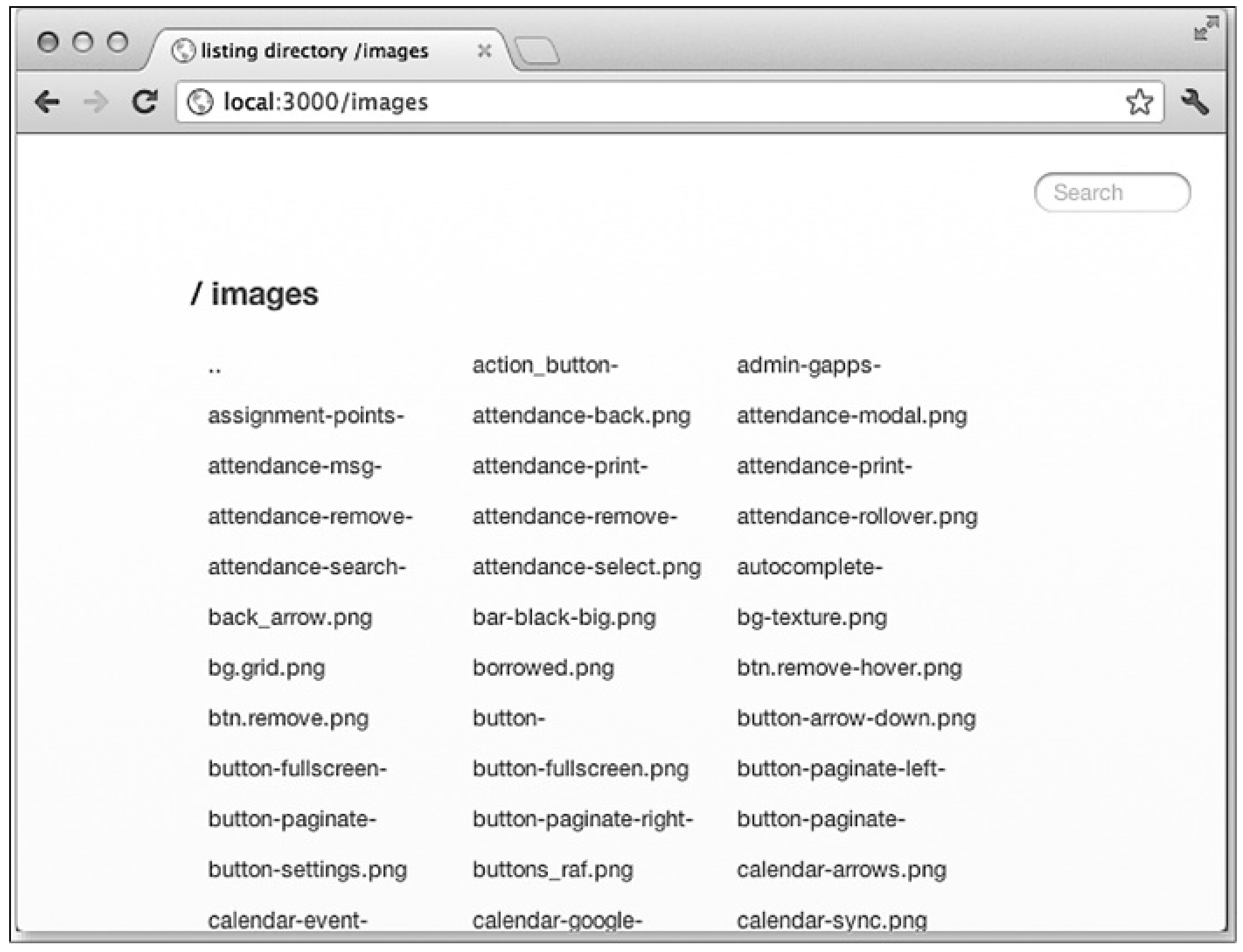



**Рис. 7.5.** Вывод содержимого папки с помощью Connect-компонента directory()

### 7.4.3. Компонент directory() — вывод содержимого папок

С помощью маленького компонента directory(), встроенного в Connect, пользователи могут просматривать файлы в удаленных папках. На рис. 7.5 показан интерфейс этого компонента с полем ввода поискового запроса, значками файлов и кликабельными средствами навигации.

#### *Применение*

Компонент directory() предназначен для совместной работы с компонентом static(), который выполняет фактическое обслуживание файлов, в то время как компонент directory() просто выводит списки. Настройка столь же проста, сколь прост следующий фрагмент кода, в котором запрос GET / призван обслужить папку ./public:

```
var connect = require('connect');
var app = connect()
  .use(connect.directory('public'))
  .use(connect.static('public'));
app.listen(3000);
```

#### *Использование компонента directory() с монтированием*

При монтировании можно задать префикс, позволяющий направить оба компонента, directory() и static(), по любому выбранному вами пути, как в следующем примере кода, в котором используется префикс GET /files. Здесь параметр icons задает режим показа значков, а параметр hidden включает для обоих компонентов режим показа и обслуживания скрытых файлов:

```
var app = connect()
  .use('/files', connect.directory('public',
    { icons: true, hidden: true }))
  .use('/files', connect.static('public', { hidden: true }));
app.listen(3000);
```

Теперь вы можете легко путешествовать по файлам и папкам.

## 7.5. Резюме

Реальная мощь Connect достигается за счет богатого набора многократно используемых программных компонентов промежуточного уровня, реализующих обычную функциональность веб-приложений, включая управление сеансами,

надежное обслуживание статических файлов, сжатие исходящих данных и т.п. Назначение Connect состоит в предоставлении разработчикам некой встроенной функциональности, чтобы им не приходилось снова и снова переписывать одни и те же фрагменты кода (возможно, не столь эффективные) для своих приложений и сред разработки.

Как было показано в этой главе, используя комбинации программных компонентов промежуточного уровня, Connect позволяет строить готовые веб-приложения. Тем не менее обычно Connect применяется в качестве строительного блока в высокоуровневых средах разработки, поскольку Connect, к примеру, не предлагает никаких средств маршрутизация или шаблонизации. Именно этот низкоуровневый подход делает Connect прекрасной основой для таких высокоуровневых сред разработки, как Express, в которую интегрирована среда Connect.

Наверно, у вас уже возник вопрос о том, почему нельзя просто использовать Connect для создания веб-приложений? Ответ — можно, но все же лучше обратиться к высокоуровневой среде веб-разработки Express, которая в полной мере реализует функциональность Connect, но при этом поднимает разработку приложений на следующую ступень. Express позволяет сделать разработку приложений быстрее и удобнее благодаря элегантной системе представлений, мощному механизму маршрутизации и нескольким методам выполнения запросов и ответов. О среде разработки Express рассказывается в следующей главе.

## Глава 8. Среда разработки Express

- Приступаем к созданию нового Express-приложения
- Настройка приложения
- Создание представлений в Express
- Загрузка и выгрузка файлов

В этой главе мы поговорим о более интересных вещах. Среда веб-разработки Express (http://expressjs.com) представляет собой надстройку над Connect, предлагая инструменты и структуру, которые делают разработку веб-приложений более быстрой, простой и увлекательной. В Express поддерживается унифицированная система представлений, позволяющая использовать практически любой шаблонизатор и простые утилиты для работы с различными форматами данных, передаваемыми файлами, маршрутными URL-адресами и т.п.

По сравнению с другими средами разработки приложений, такими как Django или Ruby на платформе Rails, Express гораздо компактнее. Философия, заложенная в Express, заключается в том, что приложения значительно отличаются по требованиям и реализациям, а легковесная среда разработки позволяет смастерить именно то, что требуется, ничего лишнего. Как сама среда Express, так и сообщество Node-разработчиков ориентированы на компактные и модульные кирпичики функциональности, а не на монолитные среды разработки.

В этой главе мы постепенно от начала до конца разработаем приложение для обмена фотографиями, чтобы на этом примере научиться создавать приложения с помощью Express. В процессе разработки вы научитесь:

- генерировать начальную структуру приложения;
- конфигурировать Express и приложение;
- визуализировать представления и интегрировать шаблонизаторы;
- обрабатывать формы и выгружать файлы;
- загружать ресурсы.

Готовое приложение для обмена фотографиями будет иметь вид списка, как показано на рис. 8.1.

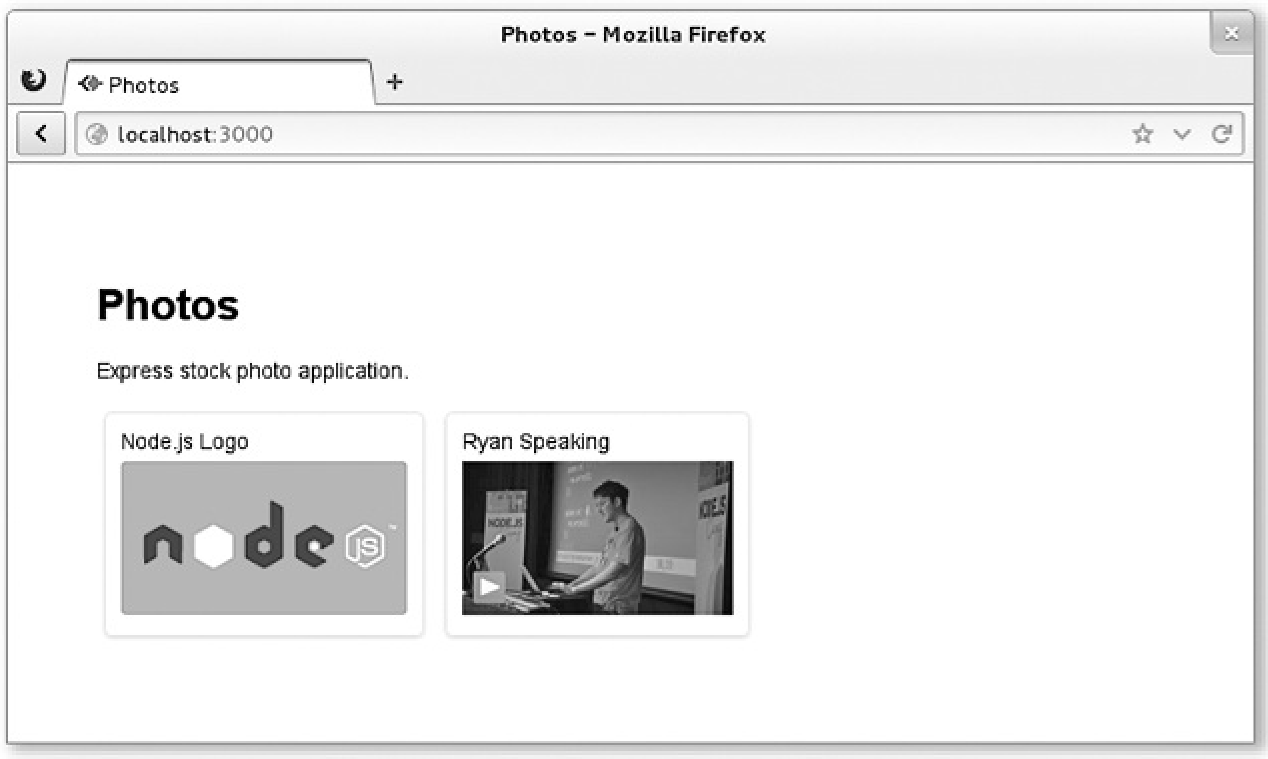



**Рис. 8.1.** Представление фотографий в виде списка

Кроме того, приложение будет иметь форму, предназначенную для выгрузки новых фотографий (рис. 8.2).

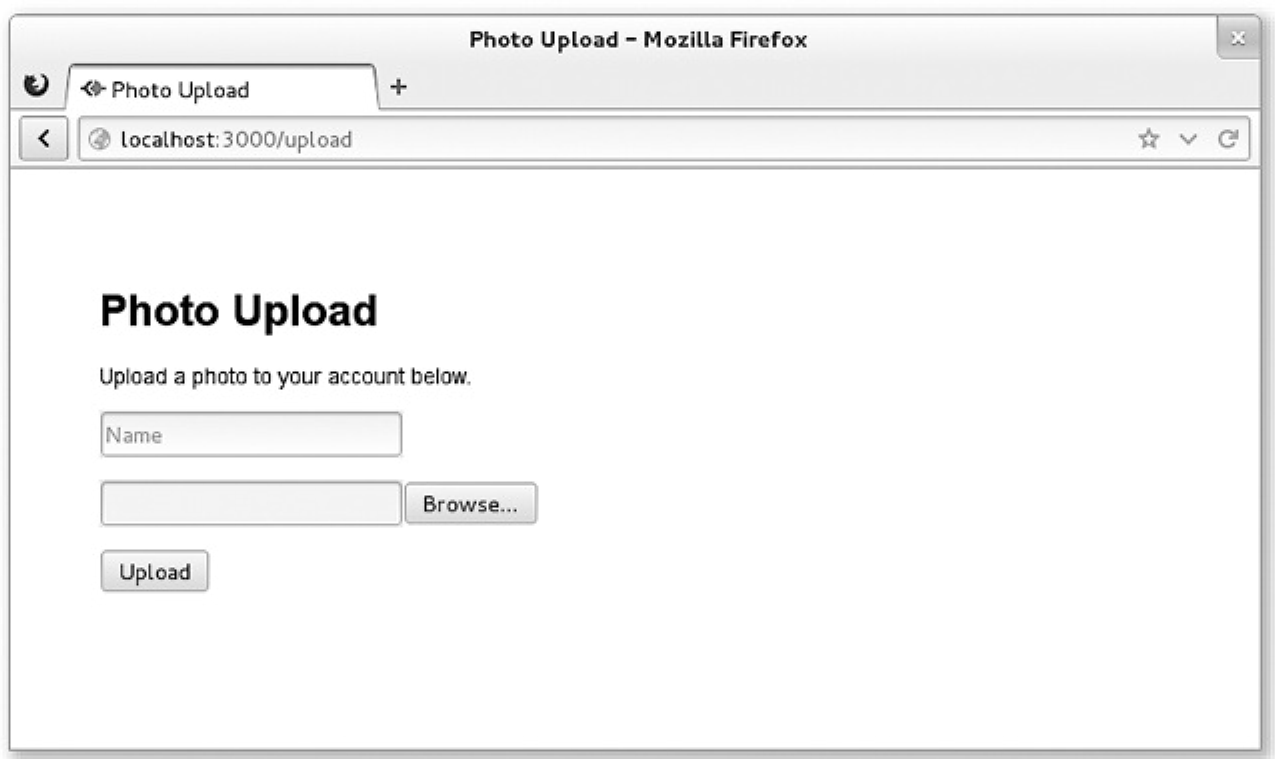



**Рис. 8.2.** Представление для выгрузки фотографий

И наконец, в нем будет реализован механизм для загрузки фотографий (рис. 8.3).

**Рис. 8.3.** Загрузка файла

А для начала давайте взглянем на структуру приложения.

### 8.1. Генерирование структуры приложения

Express не навязывает разработчику жесткую структуру приложения — вы можете размещать маршруты в любом количестве файлов, публиковать ресурсы в произвольной папке и т.п. Простейшее Express-приложение может быть весьма компактным, как показано в листинге 8.1, являясь тем не менее полнофункциональным HTTP-сервером.

**Листинг 8.1.** Простейшее Express-приложение

```
var express = require('express');
var app = express();

// Ответ на любой веб-запрос в папке /
app.get('/', function(req, res){
  // Передаем "Hello" в качестве текста ответа
  res.send('Hello');
});

// Слушаем порт 3000
app.listen(3000);
```

Исполняемый сценарий express(1), связанный с Express, может предоставить вам структуру приложения. Если вы только начинаете работать с Express, лучше всего воспользоваться таким автоматически сгенерированным приложением, поскольку вы получаете полностью готовое приложение с шаблонами, общедоступными ресурсами, конфигурацией и пр.

Заданную по умолчанию структуру приложения, сгенерированную сценарием express(1), составляют несколько папок и файлов, как показано на рис. 8.4. Эта структура ориентирована на разработчиков и с помощью Express ее можно запустить за считанные секунды, однако структуру рабочего приложения вам придется создавать самому.

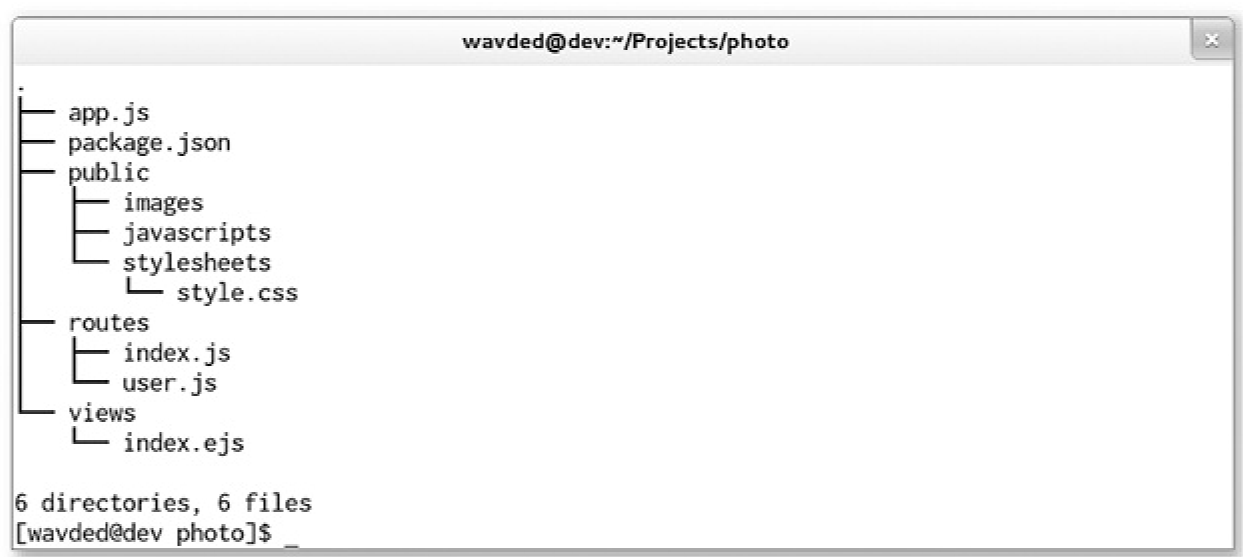



**Рис. 8.4.** Заданная по умолчанию структура приложения, в которой используются EJS-шаблоны

В примере, разрабатываемом в этой главе, мы будем использовать EJS-шаблоны, структура которых похожа на HTML-код. Формат EJS напоминает PHI JSP (для Java) и ERB (для Ruby), где серверный JavaScript-код внедряется в HTM документ и выполняется еще до передачи клиенту. Более подробно формат EJS рассматривается в главе 11.

К концу этой главы мы создадим приложение, имеющее схожую, но более расширенную структуру (рис. 8.5).

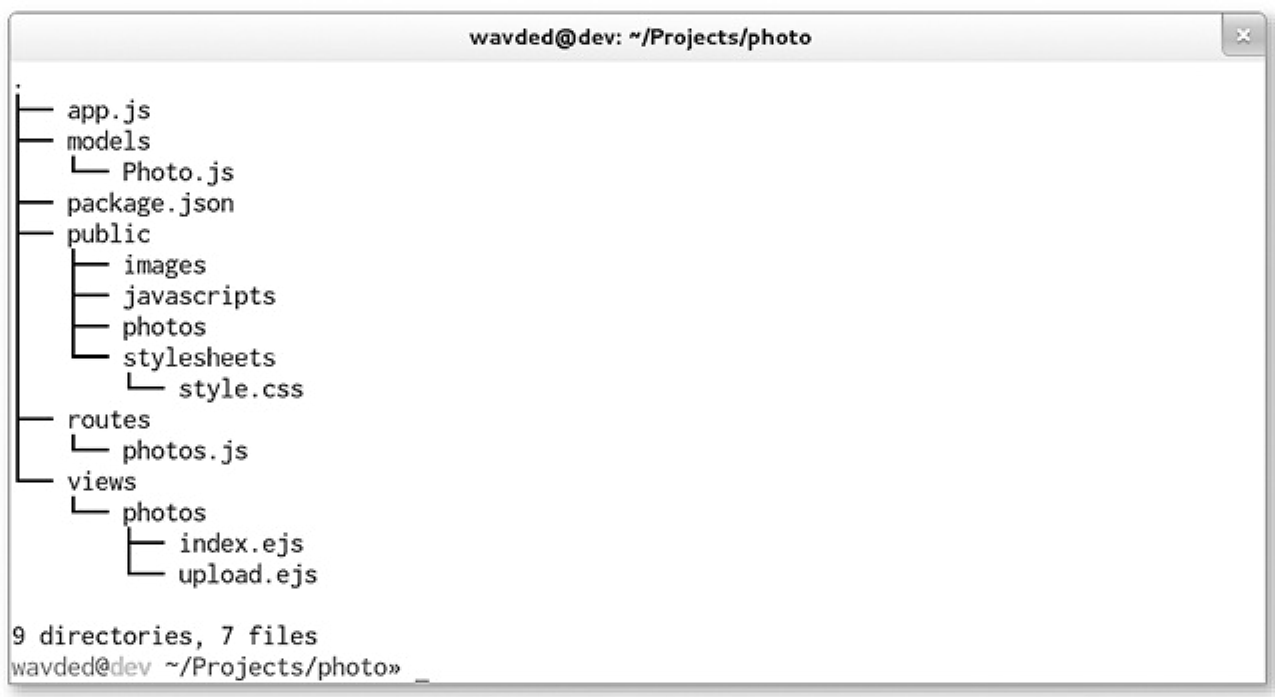



**Рис. 8.5.** Структура готового приложения

В этом разделе мы:

• глобально установим Express из npm-хранилища;

- сгенерируем приложение;
- исследуем приложение и зададим зависимости.

Итак, приступим.

### 8.1.1. Установка исполняемого файла среды Express

Сначала нужно глобально установить Express из хранилища с помощью следующей команды:

$ npm install -g express

После завершения установки воспользуйтесь флагом −help, чтобы просмотреть перечень доступных параметров (рис. 8.6).

```
wavded@dev:~/Projects/photo
[wavded@dev photo]$ express --help

  Usage: express [options]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -s, --sessions      add session support
    -e, --ejs           add ejs engine support (defaults to jade)
    -J, --jshtml        add jshtml engine support (defaults to jade)
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus) (defaults to plain c
ss)
    -f, --force         force on non-empty directory

[wavded@dev photo]$ _
```

**Рис. 8.6.** Справка в Express

В результате выбора некоторых параметров будут генерироваться небольшие фрагменты кода приложения. Например, можно заставить шаблонизатор сгенерировать пустой файл шаблона для другого шаблонизатора. Для решения этой задачи определите CSS-препроцессор с помощью параметра −css. При использовании параметра −sessions будет активизировано программное обеспечение промежуточного уровня, предназначенное для управления сеансами.

После завершения установки исполняемого файла сгенерируем заготовку, которая к концу главы превратится в приложение для обмена фотографиями.

### 8.1.2. Генерирование приложения

Для создаваемого приложения можно воспользоваться параметром -е (или —ejs), позволяющим подключить шаблонизатор EJS:

express -e photo

В результате выполнения этой команды в папке photo будет создано полнофункциональное приложение. Там появятся файл package.json, в котором описаны проект и зависимости, файл приложения, папки с общедоступными файлами и папка маршрутов (рис. 8.7).

```
wavded@dev:~/Projects
[wavded@dev Projects]$ express -e photo

   create : photo
   create : photo/package.json
   create : photo/app.js
   create : photo/public
   create : photo/public/javascripts
   create : photo/public/images
   create : photo/routes
   create : photo/routes/index.js
   create : photo/routes/user.js
   create : photo/views
   create : photo/views/index.ejs
   create : photo/public/stylesheets
   create : photo/public/stylesheets/style.css

   install dependencies:
     $ cd photo && npm install

   run the app:
     $ node app

[wavded@dev Projects]$ _
```

**Рис. 8.7.** Генерирование Express-приложения

### 8.1.3. Изучение приложения

А теперь познакомимся со структурой сгенерированного приложения. Чтобы увидеть зависимости приложения, откройте файл package.json в окне редактора (рис. 8.8). Express не делает предположений относительно версии зависимостей, предпочитаемой разработчиком, поэтому если вы не хотите столкнуться с непредвиденными ошибками, явно указывайте требуемые версии модулей. Например, с помощью следующей команды явно задается код, который будет использован во всех установленных копиях:

"express": "3.0.0"

Чтобы при использовании формата EJS добавить последнюю версию модуля, при установке модуля укажите флаг –save с командой npm. Выполните следующую команду, а затем снова откройте файл package.json, чтобы увидеть изменения:

$ npm install ejs —save

В листинге 8.2 показано содержимое файла приложения, сгенерированного с помощью команды express(1). Пока не изменяйте его. После прочтения двух предыдущих глав, посвященных среде Connect, вы уже должны быть знакомы с программными компонентами промежуточного уровня, сгенерированными в этом файле. Тем не менее стоит внимательно изучить этот код, чтобы получить представление о предлагаемой по умолчанию конфигурации программного обеспечения промежуточного уровня.

**Листинг 8.2.** «Каркас» сгенерированного Express-приложения

```
var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

var app = express();

app.configure(function(){
  app.set('port', process.env.PORT || 3000);
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  // Использование заданного по умолчанию значка сайта
  app.use(express.favicon());
  // Выводим журнал в цветовом оформлении, чтобы было удобно разработ
  app.use(express.logger('dev'));
  // Синтаксический разбор тел запросов
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  // Обслуживание статических файлов из папки ./public
  app.use(express.static(path.join(__dirname, 'public')));
});

app.configure('development', function(){
  // Выводим в процессе разработки страницы
```

```
  // ошибки, оформленные с помощью HTML-стилей
  app.use(express.errorHandler());
});

// Задаем маршруты для приложения
app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log("Express server listening on port " + app.get('port'));
});
```

У нас уже есть файлы package.json и app.js, но приложение не будет выполняться, поскольку еще не заданы зависимости. После завершения генерирования файла package.json с помощью команды express(1) нужно задать зависимости, как показано на рис. 8.9. Для этого выполните команду npm install, а затем команду node app.js, чтобы запустить приложение на выполнение. Проверьте приложение, указав в адресной строке браузера адрес http://localhost:3000. Приложение, предлагаемое по умолчанию, показано на рис. 8.10.

```
wavded@dev:~/Projects/photo
[wavded@dev Projects]$ cd photo
[wavded@dev photo]$ cat package.json
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "express": "3.0.0",
    "ejs": "*"
  }
[wavded@dev photo]$ _
```

**Рис. 8.8.** Сгенерированное содержимое файла package.json

```
wavded@dev:~/Projects/photo
[wavded@dev photo]$ npm install
ejs@0.8.3 node_modules/ejs

express@3.0.0 node_modules/express
├── methods@0.0.1
├── fresh@0.1.0
├── range-parser@0.0.4
├── cookie@0.0.4
├── crc@0.2.0
├── commander@0.6.1
├── debug@0.7.0
├── mkdirp@0.3.3
├── send@0.1.0 (mime@1.2.6)
└── connect@2.6.0 (pause@0.0.1, bytes@0.1.0, formidable@1.0.11, qs@0.5.1, send@0.0.4)
[wavded@dev photo]$ node app.js
Express server listening on port 3000
```

**Рис. 8.9.** Установка зависимостей и выполнение приложения

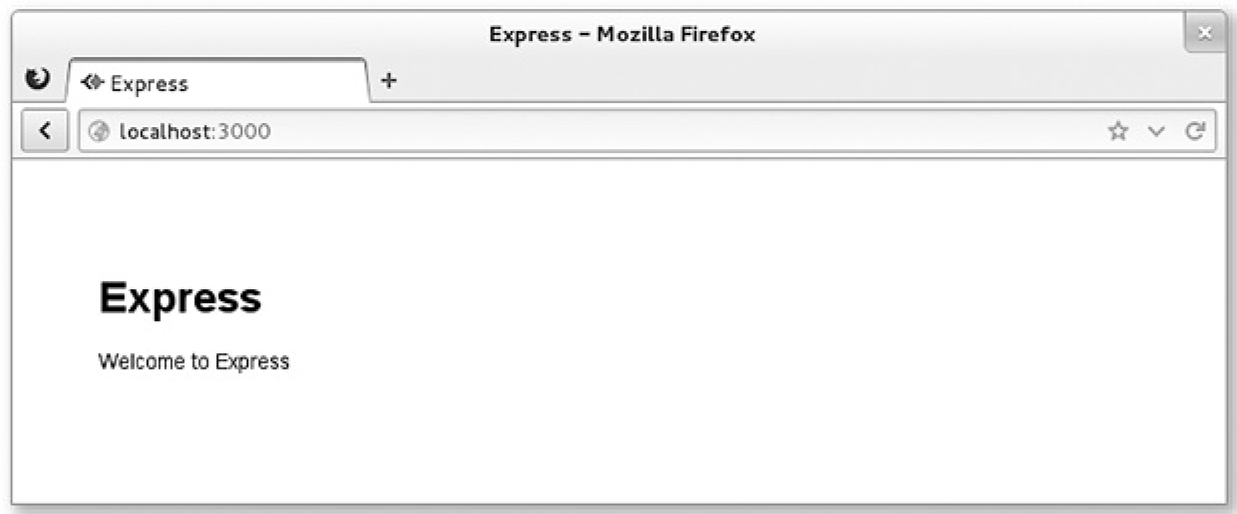



**Рис. 8.10.** Предлагаемое по умолчанию Express-приложение

После изучения сгенерированного приложения перейдем к его настройке, которая зависит от окружения.

## 8.2. Конфигурирование среды Express и приложения

Требования приложения зависят от окружения, в котором оно выполняется. Например, если программа находится на этапе разработки, лучше собирать подробные данные, а для готовой программы достаточно минимальных журналов и gzip-компрессии. Помимо конфигурирования функциональных средств, специфичных для окружения, может понадобиться задать некоторые параметры на уровне приложения, чтобы среда Express получила информацию о том, какой шаблонизатор используется и где найти шаблоны. В Express также допускается определить собственные конфигурационные пары ключ/значение.

Express имеет минималистскую систему конфигурирования, которая состоит из пяти методов, управляемых через переменную окружения NODE_ENV:

**app.configure()**

app.set()

app.get()

app.enable()

app.disable()

**установка переменных окружения**

Чтобы присвоить значение переменной окружения в системе UNIX, выполните следующую команду:

```
$ NODE_ENV=production node app
```

В Windows можно использовать такой код:

```
$ set NODE_ENV=production
$ node app
```

Эти переменные окружения будут доступны в приложении посредством объекта process.env.

В данном разделе мы узнаем, как использовать систему конфигурирования, чтобы настроить режимы работы Express и сделать разработку удобнее.

Но сначала давайте разберемся, что понимается под «зависимостью конфигурации от окружения».

### 8.2.1. Зависимость конфигурации от окружения

Хотя изначально переменная окружения NODE_ENV появилась в Express, позже он была адаптирована для многих других сред разработки на платформе Node в качестве средства извещения Node-приложения о текущем окружении, в котором по умолчанию происходит разработка.

Как показано в листинге 8.3, метод app.configure() принимает дополнительные строки, представляющие текущее окружение и некую функцию. Если текущее окружение соответствует переданной строке, тут же выполняется обратный вызов.

Если же передается только функция, она будет вызываться для всех вариантов окружения. Имена окружений выбираются совершенно произвольно. Например, могут применяться такие названия, как development, stage, test и production (или сокращенный вариант — prod).

**Листинг 8.3.** Настройка специфичных для окружения параметров с помощью метода app.configure()

```
app.configure(function(){
  // Все варианты окружения
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
...
});

// Только среда разработки
app.configure('development', function(){
  app.use(express.errorHandler());
});
```

Чтобы проиллюстрировать необязательность применения метода app.configure(), обратите внимание на листинг 8.4, который является эквивалентом предыдущего листинга, но вместо применения метода app.configure() в данном случае загрузка конфигурационных параметров выполняется из JSON- или YAML файла.

**Листинг 8.4.** Использование условных инструкций для установки зависящих от окружения параметров

```
// По умолчанию выбирается вариант "development"
var env = process.env.NODE_ENV || 'development';

app.set('views', __dirname + '/views');
// Все варианты окружения
app.set('view engine', 'ejs');
...
// Только среда разработки, использует
// инструкцию if вместо метода app.configure
if ('development' == env) {
  app.use(express.errorHandler());
}
```

Express использует систему конфигурирования внутренне, позволяя вам управлять режимами работы этой среды, но вы также можете задействовать систему конфигурирования в собственных целях. Для приложения, создаваемого в этой главе, нам потребуется лишь параметр photos, определяющий папку для хранения выгружаемых изображений. Значение этого параметра может меняться, что позволяет сохранять и обслуживать фотографии разного размера с выделением большего дискового пространства:

```
app.configure(function(){
  ...
  app.set('photos', __dirname + '/public/photos');
  ...
});

app.configure('production', function(){
  ...
  app.set('photos', '/mounted-volume/photos');
  ...
});
```

В Express также поддерживаются булевы варианты методов app.set() и app.get(). Например, метод app.enable(setting) эквивалентен методу app.set(setting, true), а с помощью метода app.enabled (setting) можно проверить, что значение параметра установлено. Дополнительными к методам app.enable (setting) и app.enabled (setting) являются методы app.disable (setting) и app.disabled(setting) соответственно. Последний позволяет проверить, что значение параметра не установлено.

Теперь, когда вы узнали, какими преимуществами обладает система конфигурирования, давайте поговорим о визуализации представлений в Express.

### 8.3. Визуализация представлений

В приложении, разрабатываемом в этой главе, мы будем использовать EJS-шаблоны, хотя, как уже отмечалось, может применяться почти любой шаблонизатор, созданный сообществом Node-разработчиков. Если вы еще не знакомы с EJS, не беспокойтесь. Этот язык подобен другим языкам создания шаблонов, таким как PHP, JSP, ERB. Некоторые базовые сведения о EJS вы получите в этой главе, а более подробно EJS и другие шаблонизаторы рассматриваются в главе 11.

Независимо от того, что визуализируется, — целая HTML-страница, HTML-

фрагмент или RSS-канал, — визуализация представлений критически важна практически для любого приложения. Эта операция основана на очень простой концепции: исходные данные передаются *представлению* (view), в котором эти данные преобразуются, как правило, в HTML-код для веб-приложений. Возможно, вы уже знакомы с концепций представлений, поскольку большая часть сред разработки предлагает схожую функциональность. На рис. 8.11 показано, как формируется новое представление данных.

Express предлагает два способа визуализации представлений. На уровне приложения применяется метод app.render(), а на уровне запросов или ответов — метод res.render(), который мы ранее задействовали внутренне. В этой главе мы будем использовать только метод res.render(). Если вы взглянете на файл ./routes/index.js, то обнаружите, что экспортируется единственная функция index. Эта функция вызывает метод res.render(), с помощью которого визуализируется шаблон ./views/index.ejs, как показано в следующем примере кода:

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

```
{ name: 'Tobi', species: 'ferret', age: 2 }

                    ↓

<h1>Tobi</h1>
<p>Tobi is a 2 year old ferret.</p>
```

**Рис. 8.11.** HTML-шаблон + данные = HTML-представление данных

В этом разделе мы узнаем:

- как конфигурировать систему представлений в Express;
- как искать файлы представлений;
- как экспонировать данные при визуализации представлений.

Прежде чем приступать к детальному изучению метода res.render(), давайте займемся конфигурированием системы представлений.

### 8.3.1. Конфигурирование системы представлений

Настройка системы представлений в Express достаточно проста. Тем не менее хотя команда express(1) сгенерирует для вас конфигурацию автоматически, надо понимать, что при этом происходит, чтобы иметь возможность внести изменения вручную. Сейчас мы поговорим о трех вещах:

- поиск представлений;
- конфигурирование шаблонизатора, предлагаемого по умолчанию;
- кэширование представлений с целью сокращения количества операций ввода-вывода.

И начнем мы с параметра views.

**__dirname**

Переменная __dirname (с двумя начальными символами подчеркивания) — это глобальная Node-переменная, идентифицирующая папку, в которой находится исполняемый файл. Зачастую при разработке приложений в качестве этой папки используется текущая рабочая папка, но в рабочем окружении может понадобиться запускать исполняемый Node-файл из другой папки. Благодаря использованию переменной __dirname упрощается поддержание совместимости путей к файлам в различных средах.

#### *Изменение папки поиска*

Следующий фрагмент кода демонстрирует параметр views, созданный исполняемым файлом среды Express:

```
app.set('views', __dirname + '/views');
```

Этот код создает папку, которую Express будет использовать при поиске представлений. Если задействовать переменную __dirname, определяющую корневую папку для приложения, исключается зависимость от текущей рабочей папки.

Теперь давайте поговорим о параметре view engine.

#### *Шаблонизатор, предлагаемый по умолчанию*

После генерирования приложения с помощью команды express(1) параметру view engine присваивается значение ejs. Это связано с тем, что при указании ключа командной строки -e выбирается шаблонизатор EJS. В результате вместо файла index.ejs визуализируется файл index. В противном случае Express требует указать расширение, чтобы определить, какой шаблонизатор должен использоваться.

Вас может удивить то, что Express проверяет расширения. Однако благодаря расширениям появляется возможность использовать несколько шаблонизаторов в рамках одного Express-приложения, при этом чистый API-интерфейс предлагается для наиболее распространенных вариантов применения, поскольку большинству приложений требуется единственный шаблонизатор.

Предположим, например, вы обнаружите, что писать RSS-каналы легче с помощью другого шаблонизатора, или просто решите перейти на другой шаблонизатор. В этом случае можно было бы по умолчанию использовать Jade, а с помощью расширения .ejs оставить EJS только для маршрута /feed, как показано в листинге 8.5.

**Листинг 8.5.** Выбор шаблонизатора по расширению имени файла

```
app.set('view engine', 'jade');

app.get('/', function(){
  // Предполагаем использовать шаблонизатор Jade, так как
  // именно он задан в качестве движка визуализации
  res.render('index');
});

app.get('/feed', function(){
  // Поскольку указано расширение .ejs, используем шаблонизатор EJS
  res.render('rss.ejs')
  ;
});
```

**синхронизация файла package.json**

Запомните, что любые дополнительные шаблонизаторы, которые вы собираетесь использовать, нужно добавлять в объект зависимостей package.json.

### *Кэширование представлений*

Чтобы предотвратить последовательные вызовы метода render() при выполнении дисковых операций ввода-вывода, в рабочей среде по умолчанию включен параметр view cache. В результате содержимое шаблонов сохраняется в памяти, что позволяет повысить быстродействие приложений. Побочный эффект заключается в том, что вы не сможете изменять файлы шаблонов без перезагрузки сервера, поэтому в среде разработки этот параметр отключают. При отладке вы, вероятно, тоже захотите его включить.

Если отключить параметр view cache, шаблон будет загружаться с диска при каждом запросе (рис. 8.12). В результате у вас появится возможность вносить изменения в шаблон без перезагрузки приложения. Если же отключить этот параметр, для каждого шаблона обращение к диску произойдет только однажды.

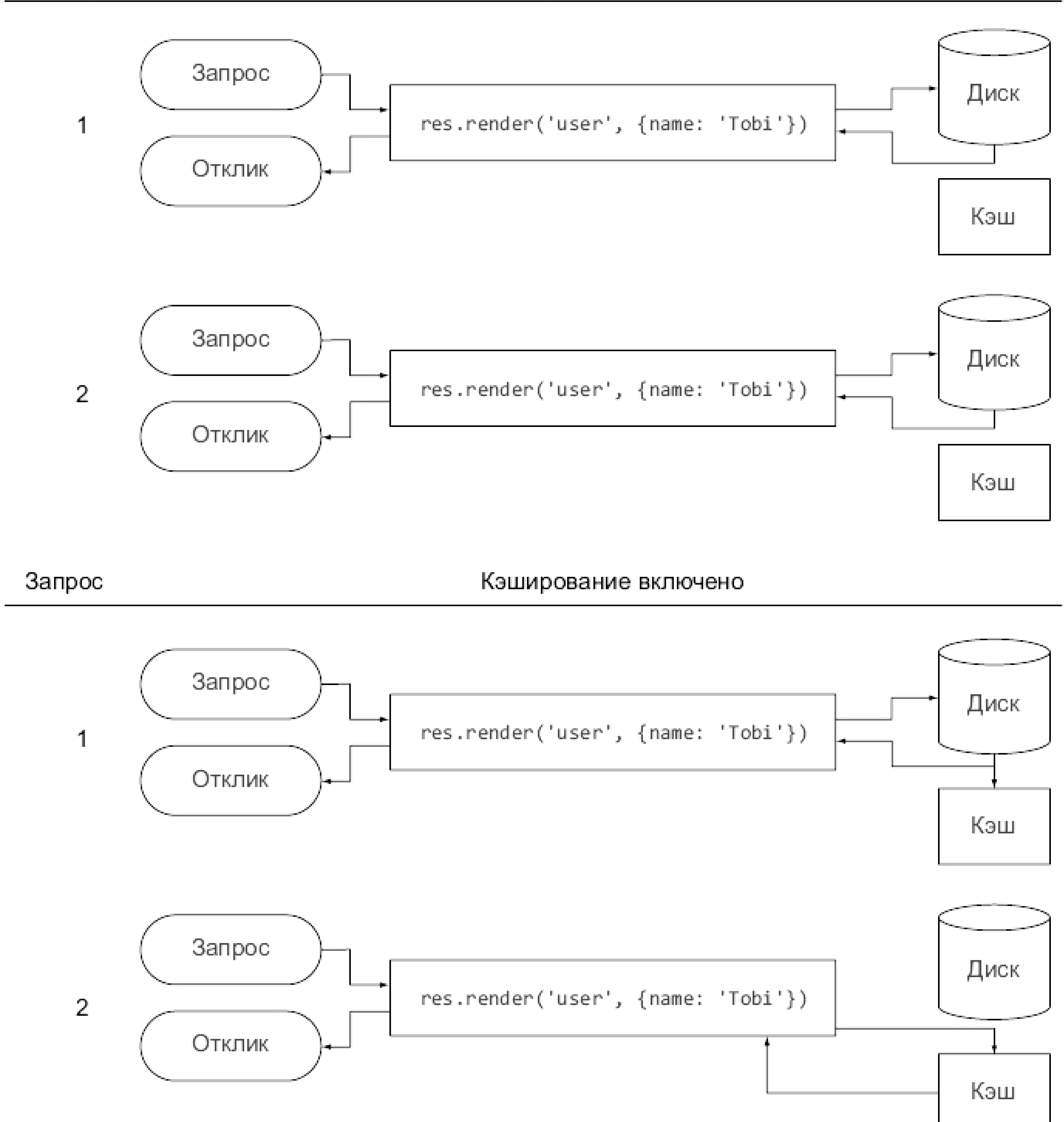



**Рис. 8.12.** Кэширование представлений

Мы только что завершили изучение механизма кэширования представлений, который позволяет повысить производительность приложений в рабочем окружении. Теперь давайте посмотрим, как Express ищет представления для визуализации.

### 8.3.2. Поиск представлений

Теперь, когда вы знаете, как происходит конфигурирование системы представлений, давайте разберемся в том, как Express ищет представление в папке,

в которой находится целевой файл представления. О создании соответствующих шаблонов можно пока не беспокоиться, о них мы поговорим позже.

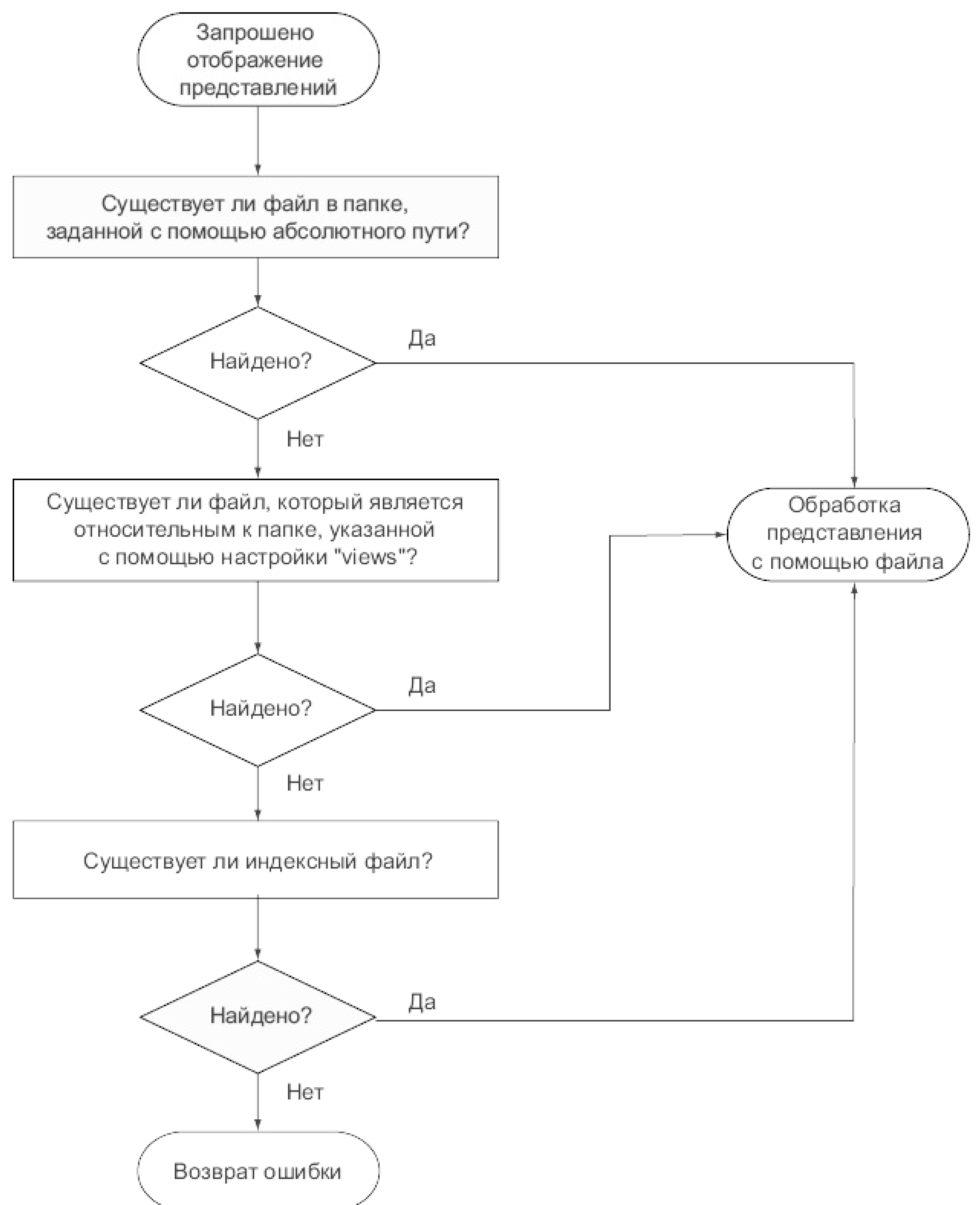



**Рис. 8.13.** Процесс поиска представлений в Express

Процесс поиска представления напоминает работу Node-функции require(). После вызова метода res.render() или app.render() Express сначала проверяет существование файла по абсолютному пути, а затем выполняет поиск относительно папки, заданной с помощью параметра views (см. п.8.3.1). Последним Express проверяет файл индекса (index).

Рисунок 8.13 демонстрирует описанный процесс в виде блок-схемы.

Поскольку по умолчанию используется шаблонизатор EJS, при вызове визуализатора расширение .ejs опускается, при этом вызов по-прежнему обрабатывается корректно.

По мере разрастания приложения ему потребуется все больше представлений, поэтому единственного ресурса будет недостаточно. Благодаря поиску представлений облегчается структурирование: например, можно использовать вложенные папки, связанные с ресурсом, и создавать в них представления, что иллюстрирует папка для фотографий на рис. 8.14.

**Рис. 8.14.** Поиск представлений в Express

Благодаря добавлению вложенных папок можно отказаться от избыточных частей в названиях файлов, например upload-photo.ejs и show-photo.ejs. Позже Express добавит расширение имени файла шаблонизатора и идентифицирует представление в виде ./views/photos/upload.ejs.

Далее Express проверит, находится ли файл index в этой папке. Если при именовании файлов используется множественный ресурс, такой как photos, обычно подразумевается список ресурсов. Соответствующий пример, res.render('photos'), показан на рис. 8.14.

Теперь, когда вы узнали, как Express ищет представления, давайте начнем создавать списки фотографий и пускать их в дело.

### 8.3.3. Экспонирование данных для представлений

Express предлагает несколько механизмов для экспонирования локальных переменных представлениям с целью их визуализации, но сами визуализируемые данные нужно где-то взять. В этом разделе для начального заполнения представления списком фотографий мы используем фиктивные данные.

Прежде чем вовлечь в дело базы данных, давайте подготовим данные для замены. Создайте файл ./routes/photos.js с маршрутами для фотографий. Затем создайте массив photos в файле, который будет нашей фиктивной базой данных. Соответствующий пример кода приведен в листинге 8.6.

**Листинг 8.6.** Фиктивные данные для фотографий, предназначенных для заполнения представления

```
var photos = [];
photos.push({
  name: 'Node.js Logo',
  path: 'http://nodejs.org/images/logos/nodejs-green.png'
});

photos.push({
  name: 'Ryan Speaking',
  path: 'http://nodejs.org/images/ryan-speaker.jpg'
});
...
```

Теперь, когда нужный контент у нас есть, требуется маршрут для вывода этого контента на экран.

### *Создание представления со списком фотографий*

Чтобы вывести фиктивные данные для фотографий на экран, нужно определить маршрут, призванный визуализировать представление с EJS-данными для фотографий (рис. 8.15).

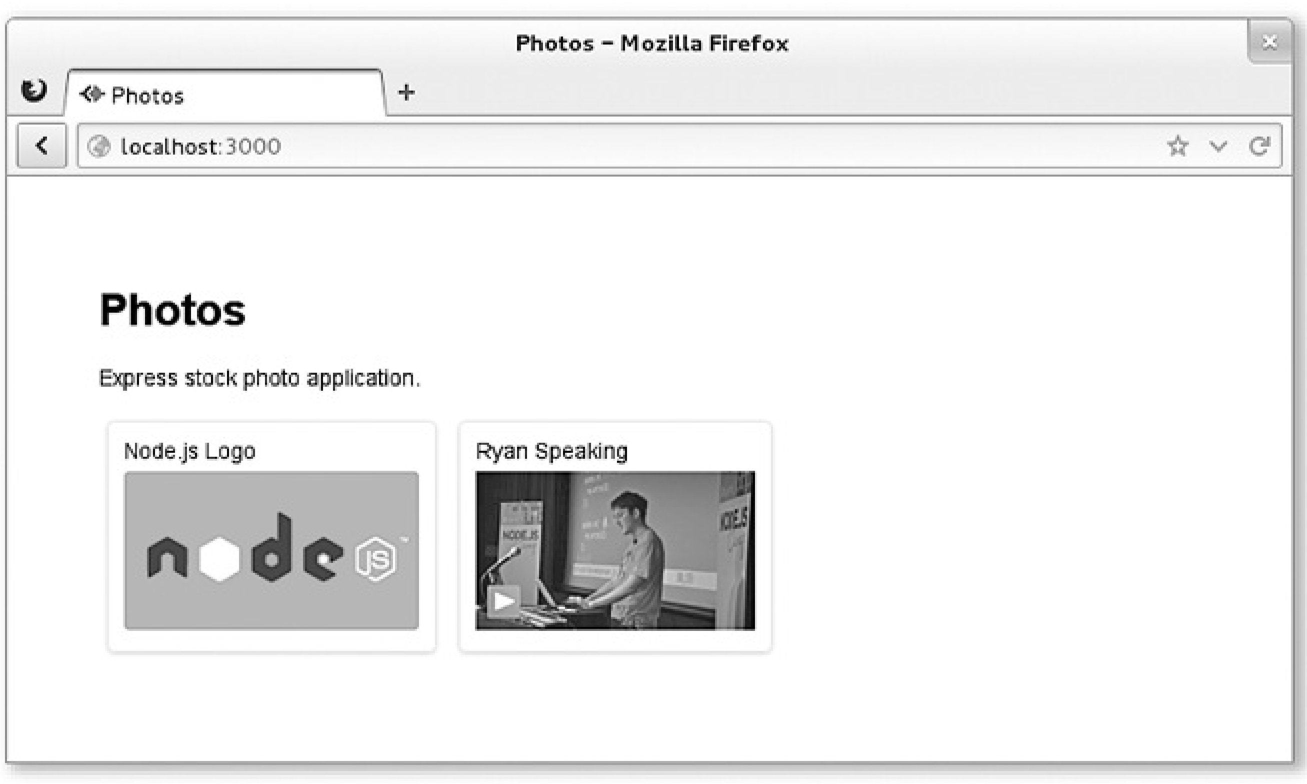


Рис. 8.15. Начальная версия представления со списком фотографий

Начните с открытия файла ./routes/photos.js и экспортируйте функцию list (листинг 8.7). Фактически для этой функции можно выбрать произвольное имя. Маршрутные функции в Connect ничем не отличаются от обычных функций программного обеспечения промежуточного уровня, они так же принимают объекты запроса и ответа и так же выполняют обратный вызов next(), который в данном случае не применяется. Передача объекта в функцию res.render() — это первый и главный метод передачи объектов представлению.

**Листинг 8.7.** Маршрутизация списка

```
exports.list = function(req, res){
  res.render('photos', {
    title: 'Photos',
    photos: photos
  });
};
```

Затем в файле ./app.js можно объявить модуль photos как загружаемый по требованию, чтобы получить доступ к только что созданной функции exports.list. Для вывода на экран фотографий, находящихся на странице индекса (/), передайте функцию photos.list методу app.get(), который служит для проецирования на эту функцию HTTP-метода GET и пути / (листинг 8.8).

**Листинг 8.8.** Добавление маршрута photos.list

```
...
var routes = require('./routes');
var photos = require('./routes/photos');
...
// Вместо app.get('/', routes.index)
app.get('/', photos.list);
```

Используя фиктивные данные и настроенный маршрут, можно создать представление для фотографий. Поскольку у нас будет несколько представлений, создадим папку ./views/photos с файлом index.ejs. С помощью JavaScript-инструкции forEach можно перебрать все объекты photo в объекте photos, который был передан методу res.render(). Затем название каждой фотографии и сама фотография выводятся на экран, как показано в листинге 8.9.

**Листинг 8.9.** Шаблон представления для списка фотографий

```
<!DOCTYPE html>
<html>
```

```
  <head>
    // Экранирование выводимых EJS-значений
    // с помощью <%= значение %>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Photos</h1>
    <p>Express stock photo application.</p>
    <div id="photos">
      // EJS вызывает JS с помощью символов <% код %>
      <% photos.forEach(function(photo) { %>
        <div class="photo">
          <h2><%=photo.name%></h2>
          <img src='<%=photo.path%>'/>
        </div>
      <% }) %>
    </div>
  </body>
</html>
```

Это представление генерирует разметку, похожую на представленную в листинге 8.10.

**Листинг 8.10.** HTML-код, созданный с помощью шаблона photos/index.ejs

```
...
<h1>Photos</h1>
<p>Express stock photo application.</p>
<div id="photos">
  <div class="photo">
    <h2>Node.js Logo</h2>
    <img src="http://nodejs.org/images/logos/nodejs-green.png" />
  </div>
...
```

Если вас интересует стилизация вашего приложения, в листинге 8.11 показан CSS-код из файла ./public/stylesheets/style.css.

**Листинг 8.11.** CSS-код, используемый для стилизации учебного приложения

этой главы

```
body {
  padding: 50px;
  font: 14px "Helvetica Neue", Helvetica, Arial, sans-serif;
}
a { color: #00B7FF; }
.photo {
  display: inline-block;
  margin: 5px;
  padding: 10px;
  border: 1px solid #eee;
  border-radius: 5px;
  box-shadow: 0 1px 2px #ddd;
}
.photo h2 {
  margin: 0;
  margin-bottom: 5px;
  font-size: 14px;
  font-weight: 200;
}
.photo img { height: 100px; }
```

С помощью команды node app запустите приложение и перейдите по адресу http://localhost:3000, введя его в адресной строке браузера. На экране появятся фотографии, как показано на рис. 8.15.

### *Методы экспонирования данных для представлений*

До сих пор мы выясняли, как передавать локальные переменные непосредственно в вызовы res.render(), но существуют и другие механизмы. Например, метод app.locals может применяться для переменных уровня приложения, а метод res.locals — для локальных переменных уровня запроса.

Значения, непосредственно передаваемые методу res.render(), будут иметь приоритет по сравнению со значениями, передаваемыми методам res.locals и app.locals, как показано на рис. 8.16.

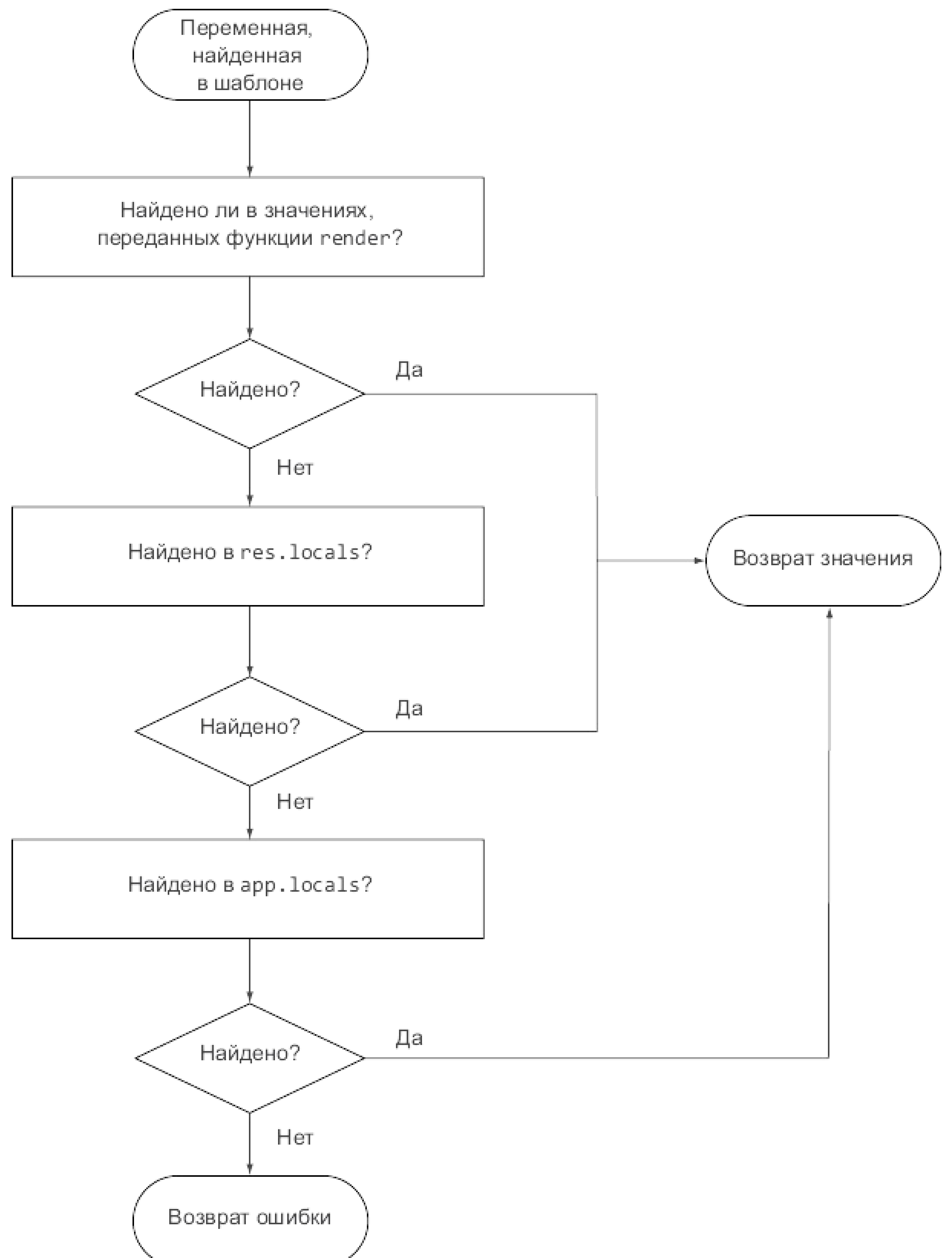



**Рис. 8.16.** Значения, передаваемые непосредственно в функцию render, при визуализации шаблона имеют приоритет

По умолчанию Express экспонирует представлениям только одну переменную уровня приложения, переменную settings. Эта переменная представляет собой объект, включающий все значения, установленные методом app.set(). Например, в результате вызова метода app.set('title', 'My Application') в шаблоне будет экспонировано свойство settings.title, как показано в следующем фрагменте EJS-кода:

```
<html>
  <head>
    <title><%=settings.title%></title>
  </head>
  <body>
    <h1><%=settings.title%></h1>
    <p>Welcome to <%=settings.title%>.</p>
  </body>
```

Внутренне Express экспонирует этот объект с помощью следующего JavaScript-кода:

```
app.locals.settings = app.settings;
```

На этом мы завершаем рассмотрение методов передачи данных представлениям.

А для порядка отметим, что app.locals тоже является JavaScript-функцией. После завершения передачи объекта все ключи объединяются, и если существуют объекты, которые нужно экспонировать полностью, такие как некоторые i18n-данные, воспользуйтесь следующим фрагментом кода:

```
var i18n = {
  prev: 'Prev',
  next: 'Next',
  save: 'Save
};
app.locals(i18n);
```

После выполнения этого кода всем шаблонам будут экспонированы свойства prev, next и save. В результате экспонируются вспомогательные функции представлений, которые помогают сократить объем кода в шаблонах. Например, если в вашем распоряжении имеется Node-модуль helpers.js с несколькими экспортированными функциями, экспонировать эти функции для представлений можно с помощью такой строки кода:

```
app.locals(require('./helpers'));
```

В следующем разделе мы наделим наше приложение способностью выгружать файлы на сайт и познакомимся со встроенным в Connect программным компонентом промежуточного уровня bodyParser, который делает это возможным.

## 8.4. Обработка форм и выгрузка файлов

В этом разделе мы реализуем механизм выгрузки фотографий. Удостоверьтесь, что

в вашем приложении задан параметр photos (см. п.8.2.1). Благодаря этому параметру можно свободно изменять папку для фотографий, адаптируя приложение для разных вариантов окружения. Теперь мы будем сохранять фотографии в папке ./public/photos, как показано в листинге 8.12. Поэтому создайте такую папку.

**Листинг 8.12.** С помощью специального параметра задается целевая папка для выгрузки фотографий

```
...
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.set('photos', __dirname + '/public/photos');
...
```

Для реализации механизма выгрузки фотографий нужно сделать три основных шага:

1. Определить модель Photo.
2. Создать форму для выгрузки фотографий;
3. Вывести список фотографий на экран.

### 8.4.1. Определение модели Photo

Для создания модели Photo мы воспользуемся простой Mongoose-моделью, о которой рассказывалось в главе 5. С помощью команды npm install mongoose −save установите Mongoose. Затем создайте файл ./models/Photo.js с определением модели из листинга 8.13.

**Листинг 8.13.** Модель Photo

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

// Устанавливаем соединение с базой данных mongodb на локальном
// хосте и используем photo_app в качестве хранилища фотографий
mongoose.connect('mongodb://localhost/photo_app');

var schema = new mongoose.Schema({
  name: String,
  path: String
```

```
});

module.exports = mongoose.model('Photo', schema);
```

Mongoose предоставит созданной модели все необходимые CRUD-методы (Photo.create, Photo.update, Photo.remove и Photo.find).

### 8.4.2. Создание формы выгрузки фотографий

Имея модель Photo, можно реализовать форму выгрузки и соответствующие маршруты. Как и в случае с большинством других страниц, для страницы выгрузки вам потребуются маршруты GET и POST.

Вы будете передавать папку для фотографий обработчику маршрута POST и возвращать обратный вызов маршрута, в результате обработчик получит доступ к папке. Добавьте новые маршруты в файл app.js, поместив их после маршрута, заданного по умолчанию (/):

```
...
app.get('/upload', photos.form);
app.post('/upload', photos.submit(app.get('photos')));
...
```

#### *Код создания формы выгрузки фотографий*

А теперь создадим собственно форму выгрузки фотографий (рис. 8.17). Эта форма содержит необязательное поле для названия фотографии и средства выбора графического файла.

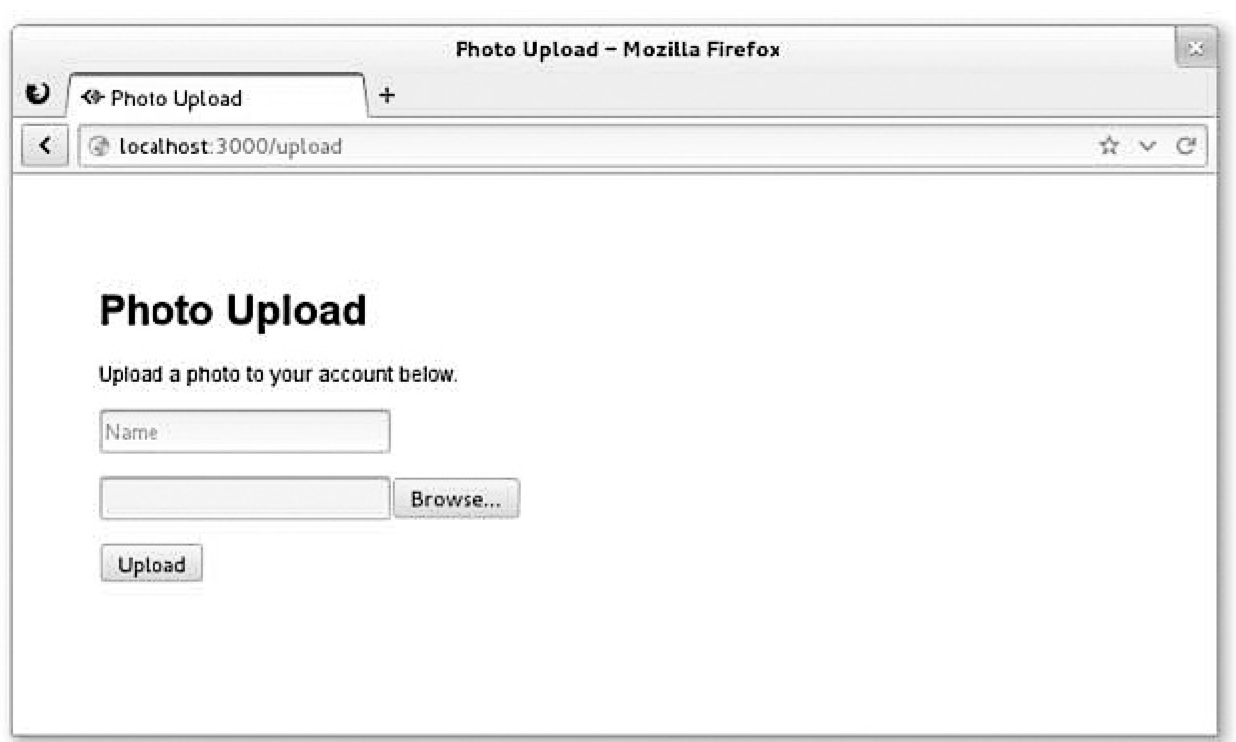



**Рис. 8.17.** Форма выгрузки фотографий

Создайте файл views/photos/upload.ejs с EJS-кодом из листинга 8.14.

**Листинг 8.14.** Код для формы выгрузки фотографий

```
<!DOCTYPE html>
<html>
  <head>
  <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Upload a photo to your account below.</p>
    <form method='post' enctype='multipart/form-data'>
      <p><input
            type='text', name='photo[name]', placeholder='Name'/>
            </p>
      <p><input type='file', name='photo[image]'/></p>
      <p><input type='submit', value='Upload'/></p>
    </form>
  </body>
</html>
```

А теперь выведем форму выгрузки фотографий на экран.

### *Вывод формы выгрузки фотографий на экран*

Теперь у нас есть форма выгрузки фотографий, но мы пока не знаем, как вывести ее на экран. Эту задачу можно решить с помощью функции photos.form. В файл ./routes/photos.js экспортируйте функцию form, которая будет визуализировать файл ./views/photos/upload.ejs (листинг 8.15).

**Листинг 8.15.** Вывод формы выгрузки фотографий на экран

```
exports.form = function(req, res){
  res.render('photos/upload', {
    title: 'Photo upload'
  });
};
```

### *Обработка заполненной формы выгрузки*

Теперь нам нужен маршрут для обработки заполненной формы выгрузки. Как упоминалось в главе 7, программный компонент промежуточного уровня bodyParser(), точнее, компонент multipart(), содержащий компонент bodyParser(), обеспечивает доступ к объекту req.files, представляющему файлы, которые были выгружены и сохранены на диске. Доступ к этому объекту можно получить с помощью свойства req.files.photo.image, а для доступа к полю на форме, photo[name], воспользуйтесь свойством req.body.photo.name.

С помощью метода fs.rename() файл «перемещается» в свое новое местоположение в параметре dir, передаваемом методу exports.submit(). В данном случае dir — это параметр photos, который определен в файле app.js. После завершения перемещения файла заполняется данными новый объект Photo, который сохраняется вместе с названием фотографии и путем к этой фотографии. При успешном сохранении пользователь перенаправляется на страницу индекса (листинг 8.16).

**Листинг 8.16.** Добавляем определение маршрута для обработки заполненной формы выгрузки

```
// Требуем модель Photo
var Photo = require('../models/Photo');
var path = require('path');
var fs = require('fs');
```

```
// Ссылка path.join, позволяющая назвать переменные "path"
var join = path.join;

...

exports.submit = function (dir) {
  return function(req, res, next){
    // По умолчанию в исходное имя файла
    var img = req.files.photo.image;
    var name = req.body.photo.name || img.name;
    var path = join(dir, img.name);
    // Переименование файла
    fs.rename(img.path, path, function(err){
      // Делегируем ошибки
      if (err) return next(err);
      Photo.create({
        name: name,
        path: img.name
      }, function (err) {
      // Делегируем ошибки
        if (err) return next(err);
        // Выполняем HTTP-перенаправление на страницу индекса
        res.redirect('/');
      });
    });
  };
};
```

Ура! Теперь мы можем выгружать фотографии. В следующем разделе мы реализуем логику вывода фотографий на странице индекса.

### 8.4.3. Вывод списка выгруженных фотографий

В разделе 8.3.3 мы реализовали маршрут app.get('/', photos.list) для фиктивных данных. Теперь пришла пора заменить их реальными данными.

Предыдущий обратный вызов маршрута делал лишь немного больше, чем

передача фиктивного массива фотографий шаблону, как показано в следующем примере кода:

```
exports.list = function(req, res){
  res.render('photos', {
    title: 'Photos',
    photos: photos
  });
};
```

В обновленной версии для получения каждой выгруженной фотографии используется файл Photo.find, предоставленный в Mongoose. Обратите внимание, что этот пример плохо работает с большими коллекциями фотографий. О том, как разбить коллекцию фотографий на страницы, мы поговорим в следующей главе.

Когда происходит обратный вызов с массивом объектов photos, оставшаяся часть маршрута остается такой же, как и перед выполнением асинхронного запроса (листинг 8.17).

**Листинг 8.17.** Измененный маршрут для списка

```
exports.list = function(req, res, next){
  // С помощью символов {} осуществляется поиск
  // всех записей в коллекции фотографий
  Photo.find({}, function(err, photos){
    if (err) return next(err);
    res.render('photos', {
      title: 'Photos',
      photos: photos
    });
  });
};
```

Давайте обновим шаблон ./views/photos/index.ejs таким образом, чтобы он был относительным для папки ./public/photos (листинг 8.18).

**Листинг 8.18.** Измененное представление, использующее параметры пути для фотографий

```
...
<% photos.forEach(function(photo) { %>
  <div class="photo">
    <h2><%=photo.name%></h2>
```

```
      <img src='/photos/<%=photo.path%>'/>
    </div>
  <% }) %>
  ...
```

Теперь на странице индекса появится динамический список фотографий, которые были выгружены приложением (рис. 8.18).

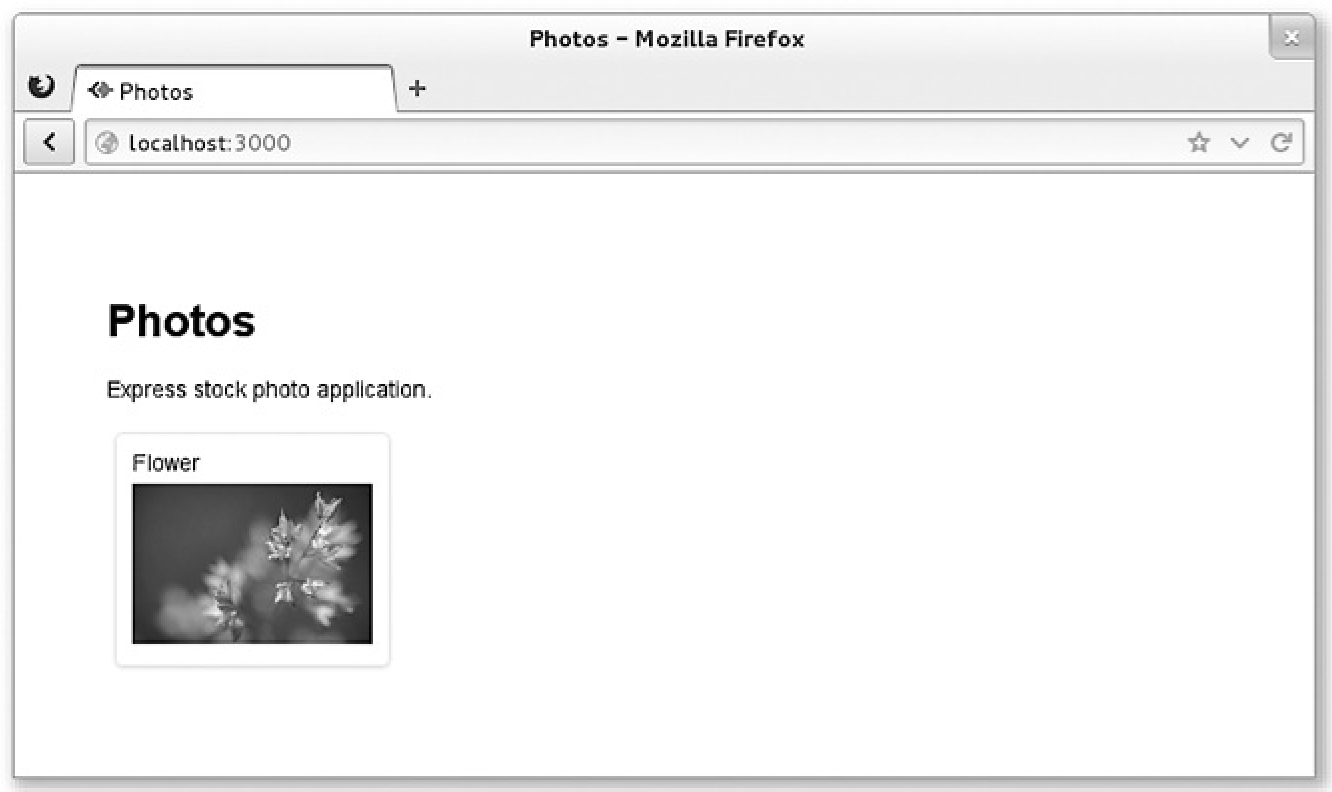



**Рис. 8.18.** Текущее состояние приложения для обмена фотографиями

До сих пор мы задавали простые маршруты, в которых не использовались символы подстановки. В следующем разделе мы более внимательно изучим доступные в Express средства маршрутизации.

## 8.5. Загрузка ресурсов

Ранее уже затрагивалась тема обслуживания статических файлов с помощью программного компонента промежуточного уровня express.static(), однако Express предлагает для передачи файлов несколько полезных методов ответа. Среди них метод res.sendfile(), применяемый для передачи файлов, и метод res.download(), предлагающий браузеру сохранить файл.

В этом разделе мы настроим приложение таким образом, чтобы загружать исходные фотографии, добавив для этого маршрут GET /photo/:id/download.

### 8.5.1. Создание маршрута загрузки фотографий

Сначала нужно добавить ссылку на фотографии, чтобы пользователи могли их загрузить. Откройте файл ./views/photos/index.ejs и переделайте его так, чтобы он соответствовал листингу 8.19. Изменения касаются добавления ссылки вокруг тега img, указывающей на маршрут GET /photo/:id/download.

**Листинг 8.19.** Добавление загрузочной гиперссылки

```
...
<% photos.forEach(function(photo) { %>
  <div class="photo">
    <h2><%=photo.name%></h2>
     // Mongoose предоставляет поле ID, которое может
     // применяться для поиска указанной записи
    <a href='/photo/<%=photo.id%>/download'>
      <img src='/photos/<%=photo.path%>'/>


    </a>
  </div>
<% }) %>
...
```

Вернувшись к файлу app.js, определим следующий маршрут в произвольном месте среди других маршрутов

```
app.get('/photo/:id/download', photos.download(app.get('photos')));
```

Прежде чем воспользоваться этим маршрутом, его нужно реализовать. Эта задача решается в следующем разделе.

### 8.5.2. Реализация маршрута загрузки фотографий

В файл ./routes/photos.js мы экспортируем функцию загрузки, код которой приведен в листинге 8.20. Этот маршрут обеспечивает загрузку запрошенной фотографии и передает файл в соответствии с заданным путем. В методе res.sendfile(), предоставляемом Express, используется тот же код, что и в методе express.static(), поэтому вы в качестве бесплатного приложения получаете HTTP-кэш, диапазон и другие механизмы. Кроме того, этот метод также получает такие же параметры, поэтому в качестве второго аргумента можно передавать значения вида { maxAge: oneYear }.

**Листинг 8.20.** Маршрут загрузки фотографий

```
// Задаем папку, из которой будут загружаться файлы
```

```
exports.download = function(dir){
  // Задаем обратный вызов маршрута
  return function(req, res, next){
    var id = req.params.id;
    // Загружаем запись для фотографии
    Photo.findById(id, function(err, photo){
      if (err) return next(err);
      // Конструируем абсолютный путь к файлу
      var path = join(dir, photo.path);
      // Передаем файл
      res.sendfile(path);
    });
  };
};
```

После запуска приложения и прохождения процедуры аутентификации вы получите возможность щелкать на фотографиях.

Полученный результат может не совпадать с ожидаемым. С помощью метода res.sendfile() происходит передача данных и их интерпретация браузером. В случае обнаружения изображений браузер будет показывать их в окне (рис. 8.19). В следующем разделе мы познакомимся с методом res.download(), который в окне браузера выводить приглашение к загрузке.

**Рис. 8.19.** Фотография, переданная методом res.sendfile()

**аргумент обратного вызова senDfile**

Обратный вызов можно также передать в качестве второго или третьего аргумента (при использовании параметров), чтобы известить приложение о завершении загрузки. Например, с помощью обратного вызова можно уменьшать значение счетчика файлов, разрешенных для загрузки пользователем.

### *Инициирование загрузки в браузере*

При замене метода res.sendfile() методом res.download() меняется поведение браузера после завершения загрузки. Полю Content-Disposition заголовка присваивается имя файла, соответственно браузер будет выводить приглашение к загрузке.

На рис. 8.20 показано, каким образом название исходной фотографии (littlenice_by_dhor.jpeg) было использовано в качестве имени загруженного файла. Не для всех приложений подобную методику можно назвать идеальной.

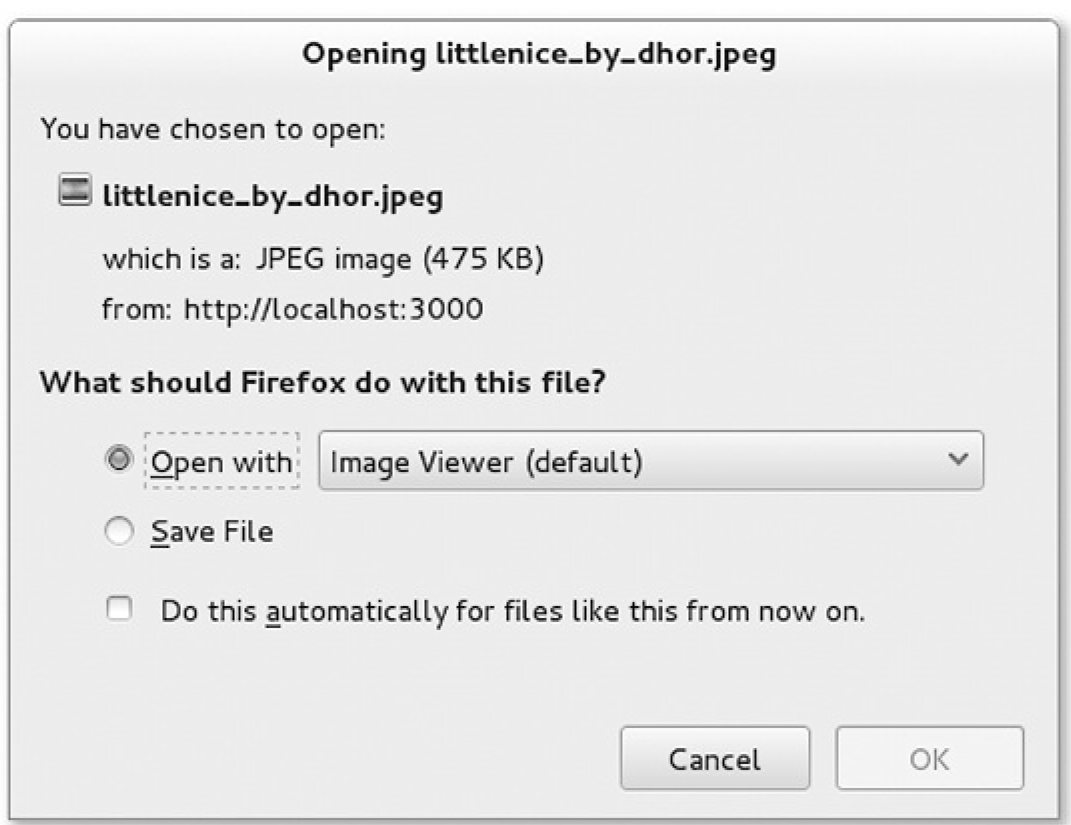



**Рис. 8.20.** Фотография была передана методом res.download()

В следующем разделе мы рассмотрим необязательный аргумент filename метода res.download().

*Задание имени загружаемого файла*

С помощью второго аргумента метода res.download() можно определить собственное имя, которое при загрузке будет использоваться вместо заданного по умолчанию исходного имени файла. Листинг 8.21 появился в результате изменения предыдущего листинга. Эта новая реализация подставляет то имя, которое было назначено фотографии при выгрузке на сервер, например Flower.jpeg.

**Листинг 8.21.** Маршрут загрузки фотографии с непосредственным заданием имени файла

```
...
var path = join(dir, photo.path);
res.download(path, photo.name+'.jpeg');
...
```

Если запустить приложение и щелкнуть на фотографии, появится приглашение к ее загрузке (рис. 8.21).

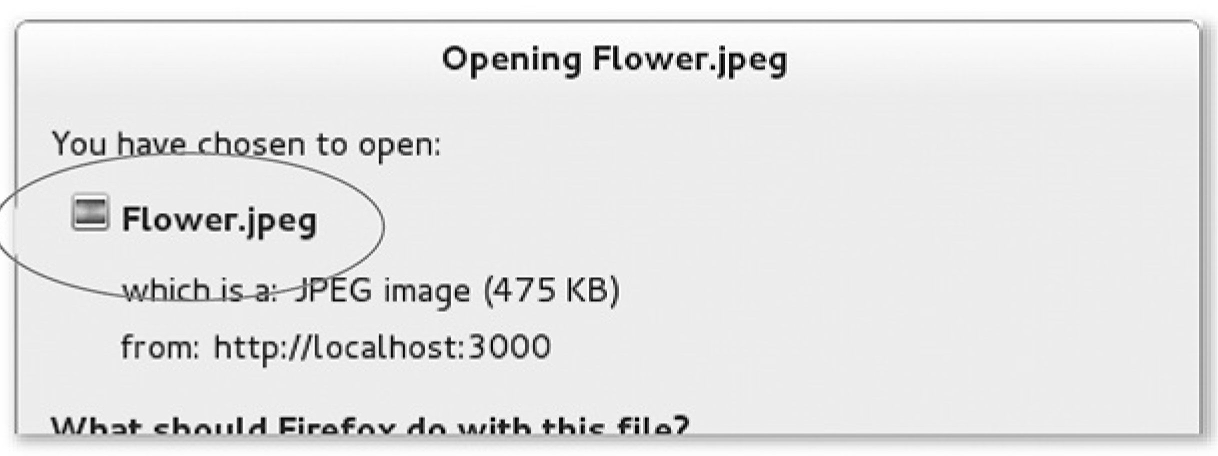



**Рис. 8.21.** Фотография с нестандартным именем, переданная методом res.download()

## 8.6. Резюме

В этой главе мы выяснили, как создать Express-приложение «с нуля» и как решать самые обычные задачи, возникающие при разработке веб-приложений.

Мы также узнали о том, как структурированы типичные папки Express-приложения и как использовать переменные окружения и метод app.configure, чтобы менять поведение приложения в различных вариантах окружения.

Базовыми компонентами Express-приложений являются маршруты и представления. Мы узнали, как визуализировать представления и как экспонировать для них данные, задавая свойства app.locals и res.locals, а также непосредственно передавая значения методом res.render(). Кроме того, мы познакомились с базовой маршрутизацией.

В следующей главе мы займемся более серьезными вещами, доступными в Express: например, научимся выполнять аутентификацию и маршрутизацию, использовать программное обеспечение промежуточного уровня и прикладной программный интерфейс REST.

## Глава 9. Нетривиальные возможности Express

- Реализация аутентификации
- Маршрутизация URL-адресов
- Создание API-интерфейса REST
- Обработка ошибок

В этой главе мы изучим ряд нетривиальных возможностей Express, которые позволят нам воспользоваться дополнительными преимуществами этой среды разработки.

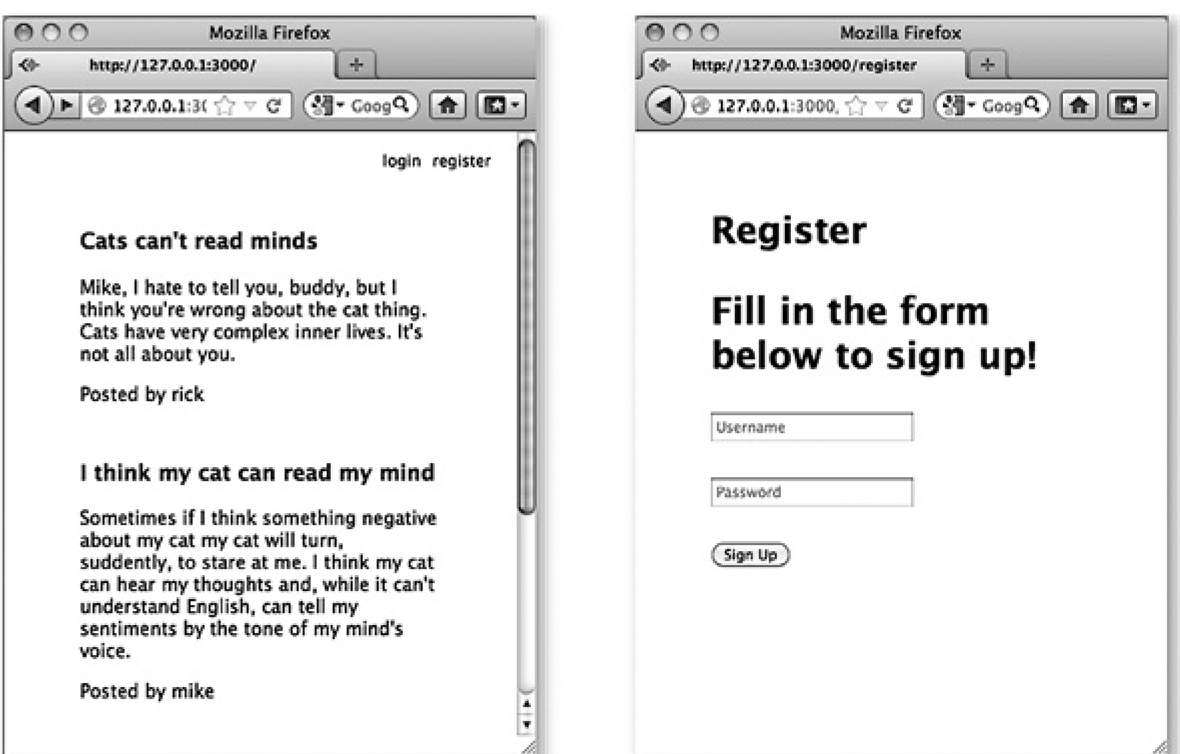



**Рис. 9.1.** Главная страница и страница регистрации приложения для чата

Чтобы продемонстрировать эти методики, мы создадим простое приложение, позволяющее людям регистрироваться и передавать общедоступные сообщения, которые выводятся на экран в обратном хронологическом порядке для просмотра посетителями веб-сайта. Подобный тип приложений известен под названием

приложений для чата. На рис. 9.1 показаны главная страница и страница регистрации пользователей, а на рис. 9.2 — страницы входа и передачи сообщений.

**Рис. 9.2.** Страницы входа и передачи сообщений

В это приложение мы добавим логику, обеспечивающую выполнение следующих операций:

- аутентификация пользователей;
- верификация и разбивка на страницы;
- предоставление API-интерфейса REST (Representational State Transfer — передач репрезентативного состояния) для отправки и получения сообщений.

Изучение нетривиальных возможностей Express мы начнем с аутентификации пользователей.

## 9.1. Аутентификация пользователей

В этом разделе мы начнем разрабатывать приложение для чата, создав «с нуля» систему аутентификации пользователей. При этом будут реализованы:

- логика хранения и аутентификации зарегистрированных пользователей;
- функциональность регистрации;
- функциональность входа в систему;

- программное обеспечение промежуточного уровня, по запросу загружающее информацию о пользователе (для пользователей, вошедших в систему).

Чтобы обеспечить аутентификацию пользователей, нужно как-то организовать хранение данных. Для нашего приложения мы будем использовать базу данных Redis (см. п.5.3.1). Эта база данных быстро устанавливается и требует минимум времени на освоение, поэтому хорошо нам подходит, тем более что мы собираемся сфокусироваться на логике приложения, а не на уровне базы данных. К тому же все, что в этой главе рассказывается о работе с базой данных, можно трансформировать почти на любую доступную базу данных, так что при желании можете заменить Redis любой другой базой данных, которая вам по вкусу. А теперь приступим к созданию модели пользователя.

### 9.1.1. Сохранение и загрузка информации о пользователях

В этом разделе мы пройдем ряд этапов, позволяющих реализовать загрузку и сохранение информации о пользователях, а также аутентификацию пользователей:

- определение зависимостей приложения с помощью файла package.json;
- создание модели пользователя;
- добавление логики загрузки и сохранения информации о пользователях с помощью Redis;
- защита пользовательских паролей с помощью модуля bcrypt;
- добавление логики аутентификации при входе в систему.

Модуль bcrypt от независимого производителя представляет собой функцию хеширования с солью. Она прекрасно подходит для хеширования паролей, поскольку позволяет эффективно отражать так называемые атаки методом грубой силы.

#### *Создание файла package.json*

Чтобы создать структуру приложения, которое поддерживает EJS и сеансы, начните сеанс командной строки, перейдите в папку разработки и введите команду:

```
express -e -s shoutbox
```

Здесь флаг -е включает поддержку EJS в файле app.js, а флаг –s — поддержку сеансов.

После создания структуры приложения измените его папку. Затем измените файл package.json, в котором определяются зависимости, включив в него папку дополнительных модулей. После внесения изменений в этот файл его содержимое должно выглядеть так, как показано в листинге 9.1.

**Листинг 9.1.** Файл package.json с дополнительными зависимостями для bcrypt и Redis

```
{
  "name": "shoutbox",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "express": "3.x",
    "ejs": "*",
    "bcrypt": "0.7.3",
    "redis": "0.7.2"
  }
}
```

Для установки зависимостей выполните команду npm install. В результате зависимости будут установлены в папку ./node_modules.

И наконец, выполните следующую команду, чтобы создать пустой файл EJS-шаблона, который мы позже наполним содержимым:

```
touch views/menu.ejs
```

Если не создавать такой файл, появится сообщение об ошибке, поскольку этот файл включается и в другие файлы шаблонов.

После завершения создания структуры приложения и установки зависимостей мы займемся определением в приложении модели пользователя.

### *Создание модели пользователя*

Теперь нужно создать папку lib, а в ней — файл user.js. В этот файл мы поместим код модели пользователя.

В листинге 9.2 представлен первый фрагмент кода, который будет добавлен в этот файл. В этом коде зависимости redis и bcrypt объявлены как загружаемые по требованию, а соединение с Redis открывается методом redis.createClient(). Функция User() получает объект и выполняет слияние свойств этого объекта в собственные свойства. Например, инструкция new User({ name: 'Tobi' }) создает объект и присваивает свойству name этого объекта значение Tobi.

**Листинг 9.2.** Начинаем создание модели пользователя

```
var redis = require('redis');
var bcrypt = require('bcrypt');
// Создание долговременного соединения с Redis
var db = redis.createClient();

// Экспорт функции User из модуля
module.exports = User;

function User(obj) {
  // Перебор ключей в переданном объекте
  for (var key in obj) {
    // Слияние значений
    this[key] = obj[key];
  }
}
```

### *Сохранение информации о пользователях в базе данных Redis*

Теперь нам нужно получить возможность сохранять информацию о пользователях в базе данных Redis. Метод save(), код которого представлен в листинге 9.3, проверяет, присвоен ли пользователю идентификатор. Если присвоен, вызывается метод update(), индексирующий идентификатор пользователя по имени и заполняющий хеш в Redis свойствами объекта. Если же идентификатор пользователя не обнаруживается, пользователь считается новым, и значение user:ids увеличивается на единицу, что дает пользователю возможность получить уникальный идентификатор; перед сохранением в Redis пароль хешируется с помощью того же метода update(). Добавьте код из этого листинга в файл lib/user.js.

**Листинг 9.3.** Реализация механизма сохранения модели пользователя

```
User.prototype.save = function(fn){
  // Пользователь уже существует
```

```
  if (this.id) {
    this.update(fn);
  } else {
    var user = this;
    // Создание уникального идентификатора
    db.incr('user:ids', function(err, id){
      if (err) return fn(err);
      // Назначение идентификатора перед сохранением
      user.id = id;
      // Хеширование пароля
      user.hashPassword(function(err){
        if (err) return fn(err);
        // Сохранение свойств пользователя
        user.update(fn);
      });
    });
  }
};

User.prototype.update = function(fn){
  var user = this;
  var id = user.id;
  // Индексирование идентификатора пользователя по имени
  db.set('user:id:' + user.name, id, function(err) {
    if (err) return fn(err);
    // Использование хеша в Redis для хранения данных
    db.hmset('user:' + id, user, function(err) {
      fn(err);
    });
  });
};
```

***Защита пользовательских паролей***

После создания пользователя свойству .pass нужно присвоить пароль пользователя.

Логика сохранения пользователя затем заменит свойство .pass хешем, сгенерированным с помощью пароля.

Хеш будет *с солью* (salted). Добавление соли к данным каждого пользователя помогает защититься от так называемых атак через радужные таблицы: с точки зрения механизма хеширования соль действует как закрытый ключ. Модуль bcrypt позволяет сгенерировать 12-символьную соль с помощью метода genSalt().

**Атака через радужные таблицы**

В атаках через радужные таблицы взлом хешированных паролей происходит с помощью особых предварительно вычисленных таблиц. Дополнительные сведения по этой теме можно найти в статье Википедии по адресу: http://en.wikipedia.org/wiki/Rainbow_table.

По окончании генерирования соли вызывается метод bcrypt.hash(), который хеширует свойство .pass и соль. Затем полученное значение хеша заменяет свойство .pass прежде, чем метод .update() сохранит это свойство в базе данных Redis. В результате вместо паролей в формате простого текста в базе данных Redis сохраняются только соответствующие хеши.

В листинге 9.4, который нужно добавить в файл lib/user.js, определяется функция, создающая хеш с солью и сохраняющая его в свойстве .pass пользователя.

**Листинг 9.4.** Добавляем поддержку bcrypt-шифрования в модель пользователя

```
User.prototype.hashPassword = function(fn){
  var user = this;
  // Генерируем 12-символьную соль
  bcrypt.genSalt(12, function(err, salt){
    if (err) return fn(err);
    // Задаем соль для сохранения
    user.salt = salt;
    // Генерируем хеш
    bcrypt.hash(user.pass, salt, function(err, hash){
      if (err) return fn(err);
      // Задаем хеш для сохранения
      user.pass = hash;
      fn();
```

```
    });
  });
};
```

Вот и все по этой теме.

### *Тестирование логики сохранения информации о пользователях*

Чтобы протестировать код, запустите Redis-сервер, введя команду redis-server в командной строке. Затем добавьте код из листинга 9.5 в нижнюю часть файла lib/user.js; этот код позволит создать «учебного» пользователя путем ввода в командной строке команды node lib/user.

**Листинг 9.5.** Тестирование модели пользователя

```
// Создание нового пользователя
var tobi = new User({
  name: 'Tobi',
  pass: 'im a ferret',
  age: '2'
});

// Сохранение пользователя
tobi.save(function(err){
  if (err) throw err;
  console.log('user id %d', tobi.id);
});
```

На основании данных, которые должны появиться на экране, например user id 1, можно сделать вывод о том, что создание пользователя прошло успешно. Завершив тестирование модели пользователя, удалите из файла lib/user.js код, представленный в листинге 9.5.

Если вы задействуете утилиту redis-cli, входящую в комплект поставки Redis, то с помощью команды hgetall можно выполнить выборку из хеша каждой пары ключ/значение, как демонстрирует сеанс командной строки в листинге 9.6.

**Листинг 9.6.** Изучение сохраненных данных с помощью утилиты redis-cli

```
// Запуск Redis в командной строке
$ redis-cli
// Поиск идентификатора последнего созданного пользователя
redis> get user:ids
```

"1"

// **Выборка данных из элемента хеш-карты**

redis> hgetall user:1

// **Свойства элемента хеш-карты**

1) "name"

2) "Tobi"

3) "pass"

4) "$2a$12$BAOWThTAkNjY7Uht0UdBku46eDGpKpK5iJcf0eLW08sMcfPL7.PN."

5) "age"

6) "2"

7) "id"

8) "4"

9) "salt"

10) "$2a$12$BAOWThTAkNjY7Uht0UdBku"

// **Выход из командной строки Redis**

redis> quit

Определив логику сохранения сведений о пользователях, давайте добавим логику выборки этих сведений.

**Другие Redis-команды, выполняемые с помощью утилиты redis-cli**

Чтобы получить дополнительные сведения о Redis-командах, обратитесь к справочнику по адресу http://redis.io/commands.

### *Выборка пользовательских данных*

Если пользователь пытается войти в систему через веб-приложение, он обычно вводит имя пользователя и пароль в форму. Затем эти данные передаются приложению для аутентификации. После передачи данных, введенных в форму входа в систему, в игру должен вступать некий метод, осуществляющий выборку сведений о пользователе по его имени.

В листинге 9.7 определен код функции User.getByName(). Эта функция сначала ищет идентификатор с помощью метода User.getId(), а затем передает найденный идентификатор методу User.get(), который получает для этого пользователя хешированные данные. Добавьте этот код в файл lib/user.js.

**Листинг 9.7.** Выборка сведений о пользователе из базы данных Redis

```
User.getByName = function(name, fn){
  // Поиск идентификатора пользователя по имени
  User.getId(name, function(err, id){
    if (err) return fn(err);
    // Выборка сведений о пользователе по идентификатору
    User.get(id, fn);
  });
};

User.getId = function(name, fn){
  // Получение идентификатора, индексированного по имени
  db.get('user:id:' + name, fn);
};

User.get = function(id, fn){
  // Выборка хеша объекта простого текста
  db.hgetall('user:' + id, function(err, user){
    if (err) return fn(err);
    // Преобразование объекта в новый объект User
    fn(null, new User(user));
  });
};
```

Завершив выборку хешированного пароля, давайте займемся аутентификацией пользователя.

### *Аутентификация пользователя при входе в систему*

Последний компонент, требуемый для выполнения аутентификации пользователя, представляет собой метод, код которого приведен в листинге 9.8. В этом коде задействованы функции, которые мы ранее создали для выборки информации о пользователе. Добавьте этот код в файл lib/user.js.

**Листинг 9.8.** Аутентификация пользователя по имени и паролю

```
User.authenticate = function(name, pass, fn){
  // Поиск пользователя по имени
  User.getByName(name, function(err, user){
```

```
      if (err) return fn(err);
      // Такого пользователя не существует
      if (!user.id) return fn();
      // Хеширование данного пароля
      bcrypt.hash(pass, user.salt, function(err, hash){
        if (err) return fn(err);
        // Соответствие найдено
        if (hash == user.pass) return fn(null, user);
        // Пароль неверный
        fn();
      });
    });
  };
```

Логика аутентификации начинается с выборки пользователя по имени. Если пользователь не найден, немедленно выполняется обратный вызов. В противном случае переданные соль и пароль пользователя хешируются, чтобы создать значение, которое должно быть идентичным сохраненному в хеше значению user.pass. Если переданный и сохраненный хеши не совпадают, значит, пользователь вошел в систему с некорректными полномочиями. Если искомого ключа не существует, Redis предоставит вам пустой хеш, который является причиной использования !user.id вместо !user.

Теперь, когда вы научились аутентифицировать пользователей, нужно найти способ их регистрировать.

### 9.1.2. Регистрация новых пользователей

Чтобы пользователи могли создавать новые учетные записи и входить через них в систему, нужно создать механизмы регистрации и входа.

В этом разделе мы займемся реализацией механизма регистрации, а для этого требуется:

- спроецировать регистрационные и входные маршруты на пути в URL-адресах;
- добавить маршрутную логику для вывода на экран формы регистрации;
- добавить логику сохранения пользовательских данных, переданных из формы.

Будущая форма регистрации показана на рис. 9.3.

**Рис. 9.3.** Форма регистрации пользователей

Эта форма появится на экране, если пользователь перейдет по ссылке /register в окне браузера. Позднее мы создадим похожую форму, которая даст пользователям возможность входить в систему.

### *Добавление маршрутов регистрации*

Чтобы получить показанную на рисунке форму регистрации, сначала нужно создать маршрут с целью визуализации этой формы и ее возвращения браузеру пользователя для вывода на экран.

В листинге 9.9 показан код измененного файла app.js, сгенерированный с помощью системы Node-модулей для импорта модуля, определяющего поведение маршрута регистрации из папки routes и связывающего HTTP-методы и URL-пути маршрутными функциями. В результате создается своего рода «фронтальный контроллер». Как видите, для маршрутов регистрации используются методы GET и POST.

**Листинг 9.9.** Добавление маршрутов регистрации

```
...
// Требующаяся маршрутная логика
var register = require('./routes/register');
...
// Добавление маршрутов
app.get('/register', register.form);
app.post('/register', register.submit);
```

Далее для определения маршрутной логики мы создадим пустой файл в папке routes, который называется register.js. Поведение маршрута регистрации (маршрут, который визуализирует шаблон регистрации) мы начнем определять путем экспорта следующей функции из файла routes/register.js:

```
exports.form = function(req, res){
  res.render('register', { title: 'Register' });
};
```

В этом маршруте для определения HTML-формы регистрации используется шаблон внедренного JavaScript-кода (Embedded JavaScript, EJS), который мы создадим следующим.

### *Создание формы регистрации*

Чтобы определить HTML-код для формы регистрации, создайте файл в папке views под названием register.ejs. Эту форму можно создать с помощью шаблона HTML/EJS-кода (листинг 9.10).

**Листинг 9.10.** Шаблон представления для формы регистрации

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
<body>
  //  Навигационные ссылки будут добавлены позже
  <% include menu %>

  <h1><%= title %></h1>
  <p>Fill in the form below to sign up!</p>

    // Логика вывода сообщений будет добавлена позже
    <% include messages %>

    <form action='/register' method='post'>
      <p>
        // Пользователь должен ввести имя
        <input type='text' name='user[name]' placeholder='Username' />
      </p>
      <p>
```

```
      <input type='password' name='user[pass]'
        // Пользователь должен ввести пароль
        placeholder='Password' />
    </p>
    <p>
      <input type='submit' value='Sign Up' />
    </p>
  </form>
</body>
</html>
```

Обратите внимание на команду include messages, которая фактически подключает другой шаблон — файл messages.ejs. Этот шаблон, который мы определим следующим, обеспечивает взаимодействие с пользователем.

### *Поддержка обратной связи с пользователем*

В процессе регистрации пользователя и во многих других областях применения типичного приложения необходимо поддерживать обратную связь с пользователем. Например, пользователь может попытаться зарегистрироваться под именем другого пользователя. В этом случае нужно дать знать пользователю о том, что введенное им имя уже занято и ему нужно выбрать другое имя.

В нашем приложении шаблон messages.ejs, который предназначен для вывода на экран сообщений об ошибках, может использоваться во многих других шаблонах.

Чтобы подготовить шаблон сообщений, в папке views создайте файл messages.ejs и поместите в этот файл представленный далее фрагмент кода. Логика шаблона проверяет, установлено ли значение переменной locals.messages, и если установлено, шаблон с помощью этой переменной выводит на экран объекты сообщений. Каждый объект сообщения включает свойство type, позволяющее выводить на экран уведомления, не имеющие отношения к ошибкам, и свойство string с текстом сообщения. Код приложения может выстраивать выводимые на экран сообщения в очередь, добавляя их в массив res.locals.messages. После вывода сообщений вызывается метод removeMessages, очищающий очередь сообщений:

```
<% if (locals.messages) { %>
  <% messages.forEach(function(message) { %>
    <p class='<%= message.type %>'><%= message.string %></p>
  <% }) %>
```

```
  <% removeMessages() %>
<% } %>
```

На рис. 9.4 показано, как выглядит форма регистрации при выводе сообщения об ошибке.

**Рис. 9.4.** Форма регистрации с сообщением об ошибке

В результате добавления сообщения в массив res.locals.messages реализуется простой механизм взаимодействия с пользователем, но объект res.locals после перенаправления не сохраняется, а чтобы сделать его более надежным, нужно с помощью сеансов сохранять сообщения между запросами.

### *Хранение транзитных сообщений в сеансах*

Обычным паттерном проектирования веб-приложений является паттерн PRG (Post/Redirect/Get — отправить/перенаправить/получить). В случае применения этого паттерна пользователь запрашивает форму, данные формы отправляются в виде HTTP-запроса POST, после чего пользователь перенаправляется на другую веб-страницу. Место, в которое попадает пользователь в результате перенаправления, зависит от того, корректны ли с точки зрения приложения данные формы. Если приложение считает, что данные формы некорректны, пользователь перенаправляется обратно на страницу формы. Если же данные в форме корректны, пользователь перенаправляется на новую веб-страницу. Обычно паттерн PRG не допускает дублирования при отправке данных формы.

В Express при перенаправлении пользователя происходит сброс содержимого объекта res.locals. То есть если сообщения для пользователя хранятся в объекте res.locals, они окажутся утраченными еще до того, как их можно будет вывести на экран. Однако если сохранять сообщения в переменной сеанса, с ними можно продолжить работу и по завершении последнего перенаправления отобразить на странице.

Чтобы с помощью переменной сеанса реализовать механизм выстраивания сообщений для пользователя в очередь, в приложение нужно включить еще один модуль. Создайте файл ./lib/messages.js и добавьте в него следующий код:

```
var express = require('express');
var res = express.response;


res.message = function(msg, type){
  type = type || 'info';
  var sess = this.req.session;
  sess.messages = sess.messages || [];
  sess.messages.push({ type: type, string: msg });
};
```

Функция res.message обеспечивает возможность добавления сообщений в переменную сеанса из любого Express-запроса. Объект express.response представляет собой прототип, который в Express используется для объектов ответа. Добавление свойств к объекту означает, что они будут доступны всему программному обеспечению промежуточного уровня и соответствующим маршрутам. В предыдущем примере кода объект express.response присваивается переменной res, чтобы упростить добавление свойств к объекту и сделать код более понятным.

Чтобы еще больше упростить добавление сообщений, используйте код из следующего фрагмента. Функция res.error позволяет легко добавить в очередь сообщений сообщение типа error. Она затрагивает также функцию res.message, которая ранее была определена в модуле:

```
res.error = function(msg){
  return this.message(msg, 'error');
};
```

Последний шаг заключается в экспонировании этих сообщений для шаблонов с целью их последующего вывода на экран. Если этого не сделать, пришлось бы передавать массив req.session.messages каждому вызову res.render() в приложении, что совсем не идеально.

Чтобы решить проблему, мы создадим программный компонент промежуточного уровня, который для каждого запроса будет заполнять массив res.locals.messages контентом массива res.session.messages, что обеспечивает эффективное экспонирование сообщений для каждого визуализируемого шаблона. До сих пор файл ./lib/messages.js применялся для расширения прототипа ответа, но он ничего не экспортирует. Если же добавить в этот файл следующий фрагмент

кода, будет экспортироваться требуемый программный компонент промежуточного уровня:

```
module.exports = function(req, res, next){
  res.locals.messages = req.session.messages || [];
  res.locals.removeMessages = function(){
    req.session.messages = [];
  };
  next();
};
```

Во-первых, переменная шаблона messages предназначается для хранения сообщений сеанса — некоего массива, который после выполнения предыдущего запроса может существовать или не существовать (не забывайте, что эти сообщения сохраняются между сеансами). Во-вторых, нам нужно придумать, как удалять сообщения из сеанса, иначе они будут неограниченно накапливаться.

Теперь для интегрирования этого нового механизма осталось добавить функцию require() в файл app.js. Вы должны смонтировать этот компонент после компонента сеанса, поскольку он зависит от определенного свойства req.session. Обратите внимание, что поскольку этот программный компонент промежуточного уровня не предназначен для приема параметров и не возвращает вторую функцию, можно вызвать функцию app.use(messages) вместо функции app.use(messages()). Как правило, перспективнее использовать программное обеспечение промежуточного уровня от стороннего производителя, которое позволяет задействовать функцию app.use(messages()) вне зависимости от того, принимает она параметры или нет:

```
...
var register = require('./routes/register');
var messages = require('./lib/messages');
...

app.use(express.methodOverride());
app.use(express.cookieParser('your secret here'));
app.use(express.session());
app.use(messages);
...
```

Теперь можно получить доступ к переменной messages и функции removeMessages() внутри любого представления, поэтому файл messages.ejs при включении в любой шаблон должен работать правильно.

С выведенной на экран формой регистрации и работающим механизмом поддержания обратной связи с пользователем можно переходить к обработке регистрационной информации, отправляемой формой.

### *Регистрация пользователя*

Теперь, когда форма регистрации определяется, а механизм поддержания обратной связи с пользователем добавлен, нам требуется создать маршрутную функцию, которая будет обрабатывать HTTP-запросы POST, направленные в папку /register. Эта функция будет называться submit.

Как упоминалось в главе 7, после отправки данных формы программный компонент промежуточного уровня bodyParser() заполняет отправленными данными свойство req.body. В форме регистрации используется объектная нотация user[name], которая после синтаксического разбора в Connect транслируется в свойство req.body.user.name. Аналогичным образом с полем ввода пароля работает свойство req.body.user.pass.

Осталось добавить небольшой фрагмент кода к маршруту подписки, который должен обеспечивать верификацию (проверяется, что имя пользователя еще не занято) и сохранение нового пользователя, как продемонстрировано в листинге 9.11.

**Листинг 9.11.** Создание пользователя на основе отправленных данных

```
var User = require('../lib/user');
...
exports.submit = function(req, res, next){
  var data = req.body.user;
  // Проверяем уникальность имени пользователя
  User.getByName(data.name, function(err, user){
    // Откладываем создание пользователя из-за ошибок
    // подключения к базе данных и других ошибок
    if (err) return next(err);
    // По умолчанию используется redis
     // Это имя пользователя уже занято
    if (user.id) {
      res.error("Username already taken!");
      res.redirect('back');
    } else {
      // Создание пользователя с помощью переданных данных
```

```
        user = new User({
        name: data.name,
        pass: data.pass
      });
      // Сохранение нового пользователя
      user.save(function(err){
        if (err) return next(err);
        // Сохранение идентификатора пользователя для аутентификации
        req.session.uid = user.id;
        // Перенаправление на страницу списка
        res.redirect('/');
         });
      }
    });
  };
```

После завершения регистрации объект user.id присваивается сеансу пользователя, чтобы позднее была возможность проверить и убедиться, что пользователь аутентифицирован. Если верификация проходит неудачно, сообщение экспонируется для шаблонов в виде переменной messages через массив res.locals.messages, а пользователь перенаправляется обратно к форме регистрации.

Чтобы реализовать эту функциональность, добавьте код из листинга 9.11 в файл routes/register.js.

Теперь можете запустить приложение, перейти по ссылке /register и зарегистрировать какого-нибудь пользователя. Следующий этап — научиться возвращать зарегистрированных пользователей к процедуре аутентификации через форму /login.

### 9.1.3. Вход в систему для зарегистрированных пользователей

Добавить функциональность входа в систему гораздо проще, чем функциональность регистрации, поскольку большая часть нужной логики уже реализована в определенном ранее универсальном методе аутентификации User.authenticate().

В этом разделе в наше приложение будет добавлено следующее:

- маршрутная логика для вывода на экран формы входа;

• логика аутентификации отправленных из формы пользовательских данных.

Форма входа в систему будет выглядеть так, как показано на рис. 9.5.

Начнем изменять файл app.js таким образом, чтобы входные маршруты загружались по требованию, а маршрутные пути устанавливались:

```
...
var login = require('./routes/login');
...
app.get('/login', login.form);
app.post('/login', login.submit);
app.get('/logout', login.logout);
...
```

А теперь добавим функциональность вывода на экран формы входа.

**Рис. 9.5.** Форма входа пользователей в систему

### *Вывод на экран формы входа в систему*

Первый шаг на пути к реализации формы входа заключается в создании файла, в котором будут храниться маршруты для входа в систему и выхода из нее: routes/login.js. Маршрутная логика вывода на экран формы входа практически идентична таковой для формы регистрации. Отличия заключаются только в имени выводимого шаблона и в заголовке страницы:

```
exports.form = function(req, res){
  res.render('login', { title: 'Login' });
};
```

Форма входа EJS-шаблона, которая определена в файле ./views/login.ejs (листинг 9.12), также весьма похожа на форму, определенную в файле register.ejs. Отличия заключаются только в выводимом на форме тексте инструкции и в маршруте

отправки данных.

**Листинг 9.12.** Шаблон представления для формы входа

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
    <body>
    <% include menu %>
    <h1><%= title %></h1>
    <p>Fill in the form below to sign in!</p>

    <% include messages %>

    <form action='/login' method='post'>
      <p>
        // Пользователь должен ввести имя
        <input type='text' name='user[name]' placeholder='Username' />
      </p>
      <p>
        <input type='password' name='user[pass]'
          // Пользователь должен ввести пароль
          placeholder='Password' />
      </p>
      <p>
        <input type='submit' value='Login' />
      </p>
    </form>
  </body>
</html>
```

После добавления маршрута и шаблона для вывода формы входа на экран в приложение нужно добавить логику обработки попыток входа в систему.

*Аутентификация попыток входа*

Для обработки попыток входа в систему вам нужно добавить в приложение маршрутную логику, которая проверяет имя пользователя и пароль, отправленные из формы. Если имя пользователя и пароль корректны, переменной session присваивается идентификатор пользователя, после чего пользователь перенаправляется на домашнюю страницу. Указанная логика реализована в листинге 9.13. Добавьте этот код в файл routes/login.js.

**Листинг 9.13.** Маршрут для обработки попыток входа

```
var User = require('../lib/user');
...
exports.submit = function(req, res, next){
  var data = req.body.user;
  // Проверка полномочий
  User.authenticate(data.name, data.pass, function(err, user){
    // Делегирование ошибок
    if (err) return next(err);
    // Обслуживаем пользователя с правильными полномочиями
    if (user) {
      // Сохраняем идентификатор пользователя для аутентификации
      req.session.uid = user.id;
      // Перенаправление к полному списку
      res.redirect('/');
    } else {
      // Экспонирование сообщения об ошибке
      res.error("Sorry! invalid credentials.");
      // Перенаправление обратно к форме входа
      res.redirect('back');
    }
  });
};
```

Как показано в листинге 9.13, если пользователь аутентифицируется с помощью метода User.authenticate(), значение свойства req.session.uid присваивается таким же образом, как и в случае с POST-маршрутом /register: это значение сохраняется в сеансе и может применяться в дальнейшем для выборки информации о пользователе или других данных, связанных с пользователем. Если соответствие не

обнаруживается, указывается ошибка и форма снова выводится на экран.

Некоторые пользователи предпочитают явно выходить из системы, поэтому нужно предусмотреть соответствующую ссылку. В файле app.js мы назначили значения свойства app.get('/logout', login.logout), поэтому в файле ./routes/login.js представленная далее функция удалит сеанс, обнаруженный программным компонентом промежуточного уровня session(), что приводит к установке сеанса для последующих запросов:

```
exports.logout = function(req, res){
  req.session.destroy(function(err) {
    if (err) throw err;
    res.redirect('/');
  })
};
```

Теперь, когда страницы регистрации и входа в систему созданы, нам нужно добавить меню, чтобы пользователи могли на них попасть. Этим мы и займемся.

### *Создание меню для аутентифицированных и анонимных пользователей*

В этом разделе мы создадим меню как для анонимных, так и для аутентифицированных пользователей. С его помощью можно будет входить в систему, регистрироваться, отправлять записи из формы и выходить из системы. На рис. 9.6 представлено меню для анонимного пользователя.

**Рис. 9.6.** Меню входа и регистрации обеспечивает доступ к созданным нами формам

Когда пользователь аутентифицируется, на экране появляется другое меню с именем пользователя и двумя ссылками: ссылкой на страницу для отправки сообщений в чат и ссылкой для выхода из системы. Это меню показано на рис. 9.7.

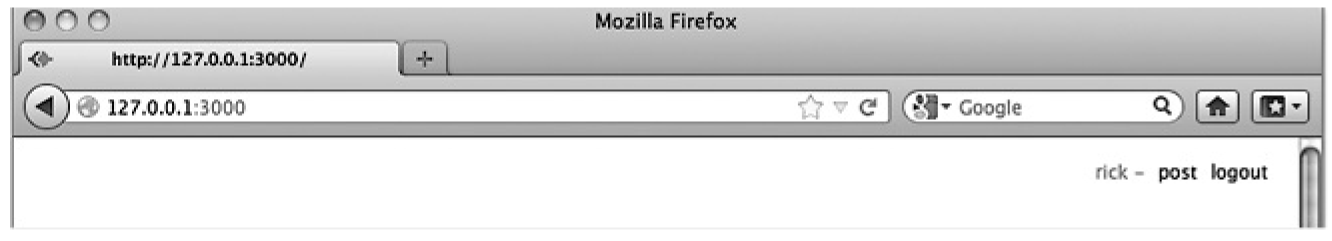



**Рис. 9.7.** Меню, которое появляется после аутентификации пользователя

Каждый созданный вами EJS-шаблон, представляющий страницу приложения, содержит код <% include menu %>, расположенный после тега <body>. Это код подключает шаблон ./views/menu.ejs, который мы сейчас создадим в соответствии с листингом 9.14.

**Листинг 9.14.** Шаблон меню для анонимных и аутентифицированных пользователей

```
<% if (locals.user) { %>


  // Меню для пользователей, вошедших в систему
  <div id='menu'>
    <span class='name'><%= user.name %></span>
    <a href='/post'>post</a>
    <a href='/logout'>logout</a>
  </div>
<% } else { %>
  // Меню для анонимных пользователей
  <div id='menu'>
    <a href='/login'>login</a>
    <a href='/register'>register</a>
  </div>
<% } %>
```

Глядя на это приложение, можно предположить, что раз переменная user экспонируется для шаблона, значит, пользователь аутентифицируется, поскольку иначе, как мы увидим далее, эта переменная не экспонировалась бы. Это означает, что когда переменная предоставлена, можно вывести на экран имя пользователя и ссылки для ввода данных и выхода из приложения. Если же сайт посещает анонимный пользователь, выводятся ссылки входа и регистрации.

У вас может возникнуть вопрос о том, откуда берется локальная переменная user, если мы ее еще не создали. В следующем разделе мы напишем код, реализующий для каждого запроса загрузку данных пользователя, вошедшего в систему, и открывающий шаблонам доступ к этим данным.

### 9.1.4. Программное обеспечение промежуточного уровня для загрузки пользовательских данных

При работе с веб-приложением часто возникает необходимость в загрузке из базы данных информации о пользователе, которая обычно предоставляется в виде

JavaScript-объекта. Благодаря наличию подобных легкодоступных данных облегчается взаимодействие с пользователем. В рамках приложения, разрабатываемого в этой главе, мы для каждого запроса будем загружать пользовательские данные с помощью программного обеспечения промежуточного уровня.

Соответствующий сценарий мы разместим в файле ./lib/middleware/user.js, то есть модель пользователя должна находиться в вышележащей папке (./lib). Сначала экспортируется функция промежуточного уровня, затем сеанс проверяется на предмет наличия идентификатора пользователя. Если идентификатор обнаруживается, значит, пользователь аутентифицирован, поэтому можно безопасно выполнять выборку данных из Redis.

Поскольку Node является однопоточной платформой, хранилище локальных программных потоков здесь отсутствует. В случае с HTTP-сервером переменные запроса и ответа являются единственными доступными контекстными объектами. Высокоуровневые среды разработки могут надстраиваться над Node с целью получения дополнительных объектов для хранения таких вещей, как данные аутентифицированных пользователей, однако в Express ставка делается на исходные объекты, предлагаемые платформой Node. В результате контекстные данные обычно хранятся в объекте запроса. Обратите внимание на листинг 9.15, где пользовательские данные сохраняются в виде свойства req.user; то есть последующие программные компоненты промежуточного уровня и маршруты могут получать доступ к пользовательским данным через это свойство.

**Листинг 9.15.** Программное обеспечение промежуточного уровня для загрузки данных пользователя, вошедшего в систему

```
var User = require('../user');
module.exports = function(req, res, next){
  // Получаем из сеанса идентификатор пользователя, вошедшего в систему
  var uid = req.session.uid;
  if (!uid) return next();
  // Получаем из Redis данные пользователя, вошедшего в систему
  User.get(uid, function(err, user){
    if (err) return next(err);
    // Экспонируем данные пользователя объекту ответа
    req.user = res.locals.user = user;
    next();
  });
};
```

Вас может удивить то, что для res.locals присваивание res.locals.user является объектом уровня запроса, который Express предоставляет с целью экспонирования данных для шаблонов, что очень напоминает app.locals. Это тоже некая функция, которая может применяться для объединения в себе существующих объектов.

Чтобы воспользоваться новым программным компонентом промежуточного уровня, сначала из файла app.js удалите все строки, содержащие текст «user». Затем, как обычно, можете объявить этот модуль загружаемым по требованию и передать его методу app.use(). В нашем приложении переменная user используется перед маршрутизатором, поэтому только маршруты и программный компонент промежуточного уровня, расположенные в коде после переменной user, будут иметь доступ к свойству req.user. Если применяется компонент, который загружает данные, компонент express.static middleware следует разместить перед ним, иначе каждый раз при обслуживании статического файла будет происходить ненужное обращение к базе данных для выборки пользовательских данных.

В листинге 9.16 продемонстрировано, как можно включить этот компонент в файл app.js.

**Листинг 9.16.** Включение компонента, загружающего пользовательские данные

```
var user = require('./lib/middleware/user');


...
app.use(express.session());
app.use(express.static(__dirname + '/public'));
// Добавляем в приложение программный компонент промежуточного уров
app.use(user);
app.use(messages);
app.use(app.router);
...
```

Если запустить приложение снова и в окне браузера перейти на страницу /login или /register, появится меню. Если нужно оформить меню с помощью стилей, добавьте в файл public/stylesheets/style.css CSS-разметку из листинга 9.17.

**Листинг 9.17.** CSS-разметка, добавляемая в файл style.css для оформления меню приложения с помощь

```
#menu {
  position: absolute;
  top: 15px;
  right: 20px;
  font-size: 12px;
```

```
  color: #888;
}

#menu .name:after {
  content: ' -';
}

#menu a {
  text-decoration: none;
  margin-left: 5px;
  color: black;
}
```

Теперь, когда меню готово, попробуйте зарегистрировать себя в качестве пользователя. Как только вы выполните эту операцию, вы должны увидеть меню для аутентифицированного пользователя, в котором есть ссылка Post.

В следующем разделе мы познакомимся с нетривиальными приемами маршрутизации, позволяющими реализовать в приложении функциональность отправки постов в чат.

## 9.2. Нетривиальные приемы маршрутизации

Основное назначение маршрутов в Express заключается в связывании URL-адреса с логикой ответа. Однако маршруты позволяют также связать URL-адрес с программным обеспечением промежуточного уровня. Это дает возможность с помощью такого программного обеспечения реализовать функциональность многократного использования на определенных маршрутах.

В этом разделе решаются следующие задачи:

- верификация отправленного пользователем из формы контента с помощью специализированного маршрутного программного обеспечения промежуточного уровня;
- реализация специализированной маршрутной верификации;
- реализация разбивки на страницы.

Давайте поговорим о том, как применять специализированное маршрутное программное обеспечение промежуточного уровня.

### 9.2.1. Верификация отправленного пользователем контента

Чтобы у нас было что-то, что можно верифицировать, давайте наделим наше приложение способностью отправлять приложению сообщения (посты). Чтобы иметь возможность отправлять посты в чат, нужно:

- создать модель записи;
- добавить маршруты, связанные с записью;
- создать форму записи;
- добавить логику создания записей на основе данных, отправленных из формы.

И начнем мы с создания модели записи.

#### *Создание модели записи*

Создайте файл, содержащий определение модели записи, и присвойте ему имя lib/entry.js. В этот файл включите код из листинга 9.18. Модель записи будет очень похожа на рассмотренную ранее модель пользователя, но в данном случае данные сохраняются в Redis-списке.

**Листинг 9.18.** Модель для записей

```
var redis = require('redis');
// Создание экземпляра Redis-клиента
var db = redis.createClient();

// Экспорт функции Entry из модуля
module.exports = Entry;

function Entry(obj) {
  // Циклический обход ключей в переданном объекте
  for (var key in obj) {
    // Слияние значений
    this[key] = obj[key];
  }
}
```

```
Entry.prototype.save = function(fn){
  // Преобразование сохраненных данных записи в JSON-строку
  var entryJSON = JSON.stringify(this);

  // Сохранение JSON-строки в Redis-списке
  db.lpush(
    'entries',
      entryJSON,
    function(err) {
      if (err) return fn(err);
      fn();
    }
  );
};
```

Подготовив базовую модель, теперь нужно добавить функцию getRange из листинга 9.19. Эта функция позволит нам извлекать записи.

**Листинг 9.19.** Код извлечения набора записей

```
Entry.getRange = function(from, to, fn){
  // Redis-функция lrange используется для извлечения записей
  db.lrange('entries', from, to, function(err, items){
    if (err) return fn(err);
    var entries = [];

    items.forEach(function(item){
      // Декодирование записей, предварительно
      // сохраненных в формате JSON
      entries.push(JSON.parse(item));
    });
    fn(null, entries);
  });
};
```

Теперь, завершив создание модели, можно добавлять маршруты в список и создавать записи.

### *Добавление связанных с записями маршрутов*

Прежде чем добавить в приложение маршруты, связанные с записями, нужно внести некоторые изменения в файл app.js. Для начала в верхнюю часть файла добавьте следующую инструкцию require:

```
var entries = require('./routes/entries');
```

Затем в том же файле app.js вместо строки, включающей текст app.get('/', подставьте указанную далее строку, чтобы все запросы к пути / возвращали список записей:

```
app.get('/', entries.list);
```

Ну а теперь займемся добавлением логики маршрутизации.

### *Добавление главной страницы с записями*

Начните с создания файла routes/entries.js, а затем добавьте в него код из листинга 9.20, придающий модели записи статус загружаемой по требованию и экспортирующий функцию визуализации списка записей.

**Листинг 9.20.** Организация списка записей

```
var Entry = require('../lib/entry');

exports.list = function(req, res, next){
  // Извлекаем записи
  Entry.getRange(0, -1, function(err, entries) {
    if (err) return next(err);

    // Визуализируем HTTP-ответ
    res.render('entries', {
      title: 'Entries',
      entries: entries,
    });
  });
};
```

Опираясь на маршрутную логику, определяющую список записей, нам нужно добавить EJS-шаблон для вывода этого списка на экран. В папку views поместите файл entries.ejs, в который добавьте EJS-шаблон из листинга 9.21.

**Листинг 9.21.** Модифицированный файл entries.ejs, поддерживающий разбивку на страницы

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>

    <% entries.forEach(function(entry) { %>
      <div class='entry'>
        <h3><%= entry.title %></h3>
        <p><%= entry.body %></p>
        <p>Posted by <%= entry.username %></p>
      </div>
     <% }) %>
  </body>
</html>
```

Если теперь запустить приложение, на главной странице появится список записей. Но так как ни одной записи у нас пока не создано, давайте перейдем к добавлению необходимых для этого компонентов.

### *Создание формы записи*

На данный момент наше приложение способно показывать список записей, но не способно их добавлять. Далее мы наделим его этой способностью, но для начала включите следующие строки кода в раздел маршрутизации файла app.js:

```
app.get('/post', entries.form);
app.post('/post', entries.submit);
```

Затем добавьте в файл routes/entries.js указанный далее маршрут (эта маршрутная логика призвана визуализировать шаблон, содержащий форму):

```
exports.form = function(req, res){
  res.render('post', { title: 'Post' });
};
```

Теперь на основе EJS-шаблона, код которого приведен в листинге 9.22, мы создадим шаблон формы и сохраним его в файле views/post.ejs.

**Листинг 9.22.** Форма для ввода постов

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>

    <h1><%= title %></h1>
    <p>Fill in the form below to add a new post.</p>

    <% include messages %>
    <form action='/post' method='post'>
      <p>
        // Текст заголовка записи
        <input type='text' name='entry[title]' placeholder='Title' />
      </p>
      <p>
        // Текст тела записи
        <textarea name='entry[body]' placeholder='Body'></textarea>
      </p>
      <p>
        <input type='submit' value='Post' />
      </p>
    </form>
  </body>
</html>
```

Теперь, имея форму, которую можно вывести на экран, мы перейдем к созданию записей на основе отправленных из формы данных.

***Создание записей***

Чтобы реализовать функциональность создания записей на основе данных,

передаваемых из формы, в файл routes/entries.js добавьте код из листинга 9.23.

**Листинг 9.23.** Добавление записи на основе отправленных из формы данных

```
exports.submit = function(req, res, next){
  var data = req.body.entry;

  var entry = new Entry({
    "username": res.locals.user.name,
    "title": data.title,
    "body": data.body
  });

  entry.save(function(err) {
    if (err) return next(err);
    res.redirect('/');
  });
};
```

Если теперь в адресной строке браузера ввести адрес /post для доступа к приложению, вы сможете добавлять записи, выполнив процедуру входа в систему.

В следующем разделе мы поговорим о специализированном маршрутном программном обеспечении промежуточного уровня и о его использовании для верификации данных форм.

### 9.2.2. Специализированное маршрутное программное обеспечение промежуточного уровня

Предположим, вы хотите, чтобы на форме с записями п стов текстовое поле для записей было обязательным. Первое, что приходит в голову, — просто добавить его в обратный вызов маршрута, как показано в следующем фрагменте кода. Однако такой подход не идеален, поскольку тесно связывает логику верификации с конкретной формой. Во многих случаях логика верификации может быть абстрагирована в многократно используемых компонентах, что делает разработку более быстрой, простой и декларативной:

```
...
exports.submit = function(req, res, next){
  var data = req.body.entry;
```

```
    if (!data.title) {
      res.error("Title is required.");
      res.redirect('back');
      return;
    }

    if (data.title.length < 4) {
       res.error("Title must be longer than 4 characters.");
       res.redirect('back');
       return;
    }
...
```

Маршруты в Express могут дополнительно принимать программные компоненты промежуточного уровня, применяемые перед последним обратным вызовом маршрута только при соответствии маршрута. Сами по себе обратные вызовы маршрутов, используемые повсеместно в этой главе, особым образом не трактуются. Они такие же, как любые другие программные компоненты промежуточного уровня, даже те, которые вы планируете создать для верификации.

Чтобы приступить к работе со специализированным маршрутным программным обеспечением промежуточного уровня, давайте рассмотрим простой, но негибкий способ реализовать верификацию в виде специализированного маршрутного компонента.

### *Верификация формы с помощью специализированного маршрутного программного компонента промежуточного уровня*

Начнем с создания нескольких простых, но уже специализированных программных компонентов для верификации. Результат расширения POST-маршрута /post с помощью такого компонента мог бы выглядеть примерно так:

```
app.post('/post',
  requireEntryTitle,
  requireEntryTitleLengthAbove(4),
  entries.submit
);
```

Обратите внимание, что в предыдущем фрагменте кода, который представляет собой определение маршрута и в качестве аргументов обычно принимает путь и

логику маршрутизации, имеются два дополнительных аргумента, определяющих программное обеспечение промежуточного уровня для верификации.

Два примера программных компонентов промежуточного уровня в листинге 9.24 показывают, как можно абстрагировать исходные проверки. Тем не менее эти примеры еще не слишком модульные и могут работать только с единственным полем entry[title].

**Листинг 9.24.** Две перспективные, но пока еще далекие от идеала попытки создания программных компонентов промежуточного уровня для верификации

```
function requireEntryTitle(req, res, next) {
  var title = req.body.entry.title;
  if (title) {
    next();
  } else {
    res.error("Title is required.");
    res.redirect('back');
  }
}

function requireEntryTitleLengthAbove(len) {
  return function(req, res, next) {
    var title = req.body.entry.title;
    if (title.length > len) {
      next();
    } else {
      res.error("Title must be longer than " + len);
      res.redirect('back');
    }
  }
}
```

Более жизнеспособное решение могло бы заключаться в абстрагировании верификаторов и в передаче имени целевого поля. Давайте рассмотрим соответствующий пример.

### *Создание гибкого программного компонента промежуточного уровня для верификации*

В следующем фрагменте кода происходит передача имени поля. Это позволяет многократно использовать логику верификации, что означает сокращение объема, необходимого для написания кода :

```
app.post('/post',
        validate.required('entry[title]'),
        validate.lengthAbove('entry[title]', 4),
        entries.submit);
```

В разделе маршрутизации файла app.js подставьте этот фрагмент кода вместо строки:

```
app.post('/post', entries.submit);
```

Естественно, для верификации можно воспользоваться одной из многочисленных общедоступных библиотек, разработанных Express-сообществом. Однако умение разбираться в принципах работы предназначенного для верификации программного обеспечения промежуточного уровня и создавать его самостоятельно воистину бесценно.

А теперь приступим к работе. Создайте файл ./lib/middleware/validate.js и включите в него код из листинга 9.25. В этот файл будет экспортировано несколько программных компонентов промежуточного уровня — в нашем случае это validate.required() и validate.lengthAbove(). Детали реализации здесь не особо важны. Суть рассматриваемого примера заключается в том, чтобы с небольшими усилиями создать код, который может использоваться во многих приложениях.

**Листинг 9.25.** Реализация программного обеспечения промежуточного уровня для верификации

```
// Синтаксический разбор нотации entry[name]
function parseField(field) {
  return field
    .split(/\[|\]/)
    .filter(function(s){ return s });
}

// Поиск свойства по результатам синтаксического разбора,
// выполненного парсером parseField()
function getField(req, field) {
  var val = req.body;
  field.forEach(function(prop){
    val = val[prop];
```

```
  });
return val;
}

exports.required = function(field){
  // Однократный синтаксический разбор поля
  field = parseField(field);
  return function(req, res, next){
    // Для каждого запроса проверяем, содержит ли поле значение
    if (getField(req, field)) {
      // Если содержит, переходим к следующему
      // программному компоненту промежуточного уровня
      next();
    } else {
      // Если не содержит, выводим ошибку
      res.error(field.join(' ') + ' is required');
      res.redirect('back');
    }
  }
};

exports.lengthAbove = function(field, len){
  field = parseField(field);
  return function(req, res, next){
    if (getField(req, field).length > len) {
      next();
    } else {
      res.error(field.join(' ') + ' must have more than '
        + len + ' characters');
      res.redirect('back');
    }
  }
};
```

Чтобы сделать этот программный компонент промежуточного уровня доступным приложению, добавьте в верхнюю часть файла app.js следующую строку

кода:

```
var validate = require('./lib/middleware/validate');
```

Если вы попытаетесь сейчас запустить приложение, то сможете оценить эффект верификации. Естественно, API-интерфейс мог бы выполнить подобную верификацию гораздо шустрее, но мы оставляем эту задачу вам в качестве упражнения.

### 9.2.3. Разбивка на страницы

Разбивка на страницы — еще один прекрасный кандидат на реализацию с помощью специализированного маршрутного программного обеспечения промежуточного уровня. В этом разделе мы создадим компактную функцию промежуточного уровня, которая легко разобьет на страницы любой доступный вам ресурс.

#### *Разработка API-интерфейса пейджера*

API-интерфейс для функции промежуточного уровня page(), который мы создадим, будет походить на приведенный далее фрагмент кода, где Entry.count — это функция для подсчета общего количества записей, а 5 — это количество записей, показываемых на странице (по умолчанию на странице выводится 10 записей). В файле apps.js замените строку, содержащую символы app.get('/', следующим фрагментом кода:

```
app.get('/', page(Entry.count, 5), entries.list);
```

Чтобы подготовить приложение к использованию компонента разбивки на страницы, в верхнюю часть файла app.js добавьте строки из следующего фрагмента кода. Это обеспечит загрузку по требованию не только создаваемого вам компонента, но и модели записи:

```
...
var page = require('./lib/middleware/page');
var Entry = require('./lib/entry');
...
```

Затем нужно реализовать функцию Entry.count(). При использовании базы данных Redis эта задача не представляет особого труда. Откройте файл lib/entry.js и добавьте следующую функцию, которая с помощью команды llen получает количество элементов списка:

```
Entry.count = function(fn){
  db.llen('entries', fn);
};
```

Теперь все готово для создания самого программного компонента промежуточного уровня, выполняющего разбивку на страницы.

### *Создание компонента для разбивки на страницы*

Для разбивки на страницы нужно выяснить номер текущей страницы, используя значение в строке информационного запроса ?page=N. В файл ./lib/middleware/page.js добавьте функцию промежуточного уровня, представленную в листинге 9.26.

**Листинг 9.26.** Программное обеспечение промежуточного уровня, выполняющее разбивку на страницы

```
module.exports = function(fn, perpage){
  // По умолчанию выводим 10 записей на странице
  perpage = perpage || 10;
  // Возвращаем функцию промежуточного уровня
  return function(req, res, next){
    var page = Math.max(
      parseInt(req.param('page') || '1', 10),
      1
    // Разбор параметра страницы в целочисленное значение с основанием 1
    ) - 1;

    // Вызов переданной функции
    fn(function(err, total){
      // Делегирование ошибок
      if (err) return next(err);
      // Сохранение свойств страницы для будущей ссылки
      req.page = res.locals.page = {
        number: page,
        perpage: perpage,
        from: page * perpage,
        to: page * perpage + perpage - 1,
        total: total,
        count: Math.ceil(total / perpage)
      };
      // Передача управления следующему программному
```

```
    // компоненту промежуточного уровня
    next();
  });
 }
};
```

Компонент промежуточного уровня из листинга 9.26 перехватывает значение, присвоенное строке информационного запроса ?page=N, например ?page=1. Затем он выбирает общее количество результатов и экспонирует объект page с заранее вычисленными значениями для всех представлений, которые позднее могут быть визуализированы. Эти значения вычисляются вне шаблона, что позволяет создать более чистый шаблон, освобожденный от избыточного кода.

### *Использование пейджера в маршруте*

Теперь нужно обновить маршрут entries.list. Для этого достаточно изменить исходную функцию Entry.getRange(0, -1) таким образом, чтобы использовать диапазон, определяемый с помощью программного компонента промежуточного уровня page(), как показано в следующем примере кода:

```
exports.list = function(req, res, next){
  var page = req.page;
  Entry.getRange(page.from, page.to, function(err, entries){
    if (err) return next(err);
...
```

**несколько слов о req.param()**

Функция req.param() напоминает ассоциативный массив $_REQUEST языка PHP. С е помощью можно проверить строку информационного запроса, маршрут или тело запроса. Например, запрос ?page=1, /:page со значением /1 или даже передаваемый JSON-массив со значением {"page":1} были бы эквивалентны. Если бы вы попробовали получить непосредственный доступ к свойству req.query.page, можно было бы использовать только строку информационного запроса.

### *Создание шаблона для ссылок разбивки на страницы*

Теперь нужен шаблон, позволяющий реализовать сам пейджер. Добавьте в файл

./views/pager.ejs код из листинга 9.27, который представляет собой простой пейджер с кнопками Previous (предыдущая страница) и Next (следующая страница).

**Листинг 9.27.** EJS-шаблон для визуализации кнопок листания страниц

```
<div id='pager'>
  // При наличии единственной страницы кнопки
  // листания страниц не отображаются
  <% if (page.count > 1) { %>
    // Если мы находимся не на первой странице, показываем
    // ссылку на предыдущую страницу
    <% if (page.number) { %>
      <a id='prev' href='/?page=<%= page.number %>'>Prev</a>
     <% } %>
     // Если мы находимся не на последней странице, показываем
     // ссылку на следующую страницу
     <% if (page.number < page.count - 1) { %>
       <% if (page.number) { %>
           
       <% } %>
       <a id='next' href='/?page=<%= page.number + 2 %>'>Next</a>
     <% } %>
  <% } %>
</div>
```

### *Включение в шаблон ссылок для разбивки на страницы*

Теперь, когда для пейджера у нас есть программный компонент промежуточного уровня и шаблон, мы можем воспользоваться EJS-директивой include для включения шаблона пейджера в шаблон ./views /entries.ejs, представляющий список записей (листинг 9.28).

**Листинг 9.28.** Модифицированный шаблон entries.ejs, поддерживающий разбивк

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
     <link rel='stylesheet' href='/stylesheets/style.css' />
```

```
  </head>
  <body>
    <% include menu %>

    <% entries.forEach(function(entry) { %>
      <div class='entry'>
        <h3><%= entry.title %></h3>
        <p><%= entry.body %></p>
        <p>Posted by <%= entry.username %></p>
      </div>
    <% }) %>

    <% include pager %>
  </body>
</html>
```

***Чистая разбивка на страницы с помощью URL-адресов***

Вы могли бы удивиться тому, что для листания страниц вместо URL-адреса, такого как ?page=2, достаточно пути, например /entries/2. К счастью, для этого нужно внести всего два изменения в реализацию разбивки на страницы:

1. Изменить маршрутный путь для получения номера страницы.

2. Модифицировать шаблон страницы.

Первый шаг заключается в изменении маршрутного пути к списку записей таким образом, чтобы получать номер страницы. Для решения этой задачи можно было бы вызвать функцию app.get() со строкой /:page, но поскольку придется учитывать символ /, что эквивалентно /0, воспользуемся необязательной строкой /:page?. В маршрутных путях строки, такие как :page, называют *параметрами* (parameters) маршрута.

С этим необязательным параметром обе строки, /15 и /, являются корректными, и функция промежуточного уровня page() по умолчанию присваивает странице первый номер. Поскольку рассматриваемый маршрут находится на верхнем уровне (к примеру, /5, а не /entries/5), параметр :page может принимать значения, соответствующие таким маршрутам, как /upload. В этом случае можно применить

простое решение, которое заключается в том, чтобы поместить определение данного маршрута следом за другими — в результате он станет последним определенным маршрутом. Это приведет к тому, что еще до того, как мы до него доберемся, мы найдем соответствия для более специализированных маршрутов.

В рамках реализации описанной стратегии первый шаг заключается в удалении существующего маршрутного пути в файле app.js для маршрута /. То есть удалите следующую строку кода:

```
app.get('/', page(Entry.count, 5), entries.list);
```

Затем добавьте в файл app.js указанный маршрутный путь, поместив его после других определений маршрутов:

```
app.get('/:page?', page(Entry.count, 5), entries.list);
```

Осталось изменить шаблон пейджера. Нужно удалить строку информационного запроса, чтобы значение стало частью пути, а не URL-адреса. Для этого следующим образом измените код в файле views/pager.ejs:

```
<div id='pager'>
  <% if (page.count > 1) { %>
    <% if (page.number) { %>
      <a id='prev' href='/<%= page.number %>'>Prev</a>
    <% } %>
    <% if (page.number < page.count - 1) { %>
      <% if (page.number) { %>
          
      <% } %>
      <a id='next' href='/<%= page.number + 2 %>'>Next</a>
    <% } %>
  <% } %>
</div>
```

Если вы теперь запустите приложение, то обнаружите, что URL-адреса страниц стали чистыми.

### 9.3. Создание общедоступного API-интерфейса REST

В этом разделе мы реализуем в нашем чате общедоступный API-интерфейс для веб-службы RESTful, чтобы приложения сторонних производителей могли получать доступ к данным и добавлять данные для публикации. Идея технологии REST (Representational State Transfer — передача репрезентативного состояния) состоит в том, что данные приложений могут запрашиваться и изменяться с помощью

глаголов (verbs) и существительных (nouns), представленных соответственно HTTP-методами и URL-адресами. REST-запрос обычно возвращает данные в формате воспринимаемом компьютером, таком как JSON или XML.

Чтобы реализовать подобный API-интерфейс, нужно:

- разработать API-интерфейс, дающий пользователям возможность показывать, выводить в виде списка, удалять и отправлять записи;
- добавить базовую аутентификацию;
- реализовать маршрутизацию;
- предлагать ответы в форматах JSON и XML.

Для аутентификации и подписания API-запросов могут применяться различные методики, но реализация более сложных решений выходит за рамки темы этой книги. Чтобы показать, как интегрировать аутентификацию, мы будем использовать программный компонент промежуточного уровня basicAuth(), входящий в комплект поставки Connect.

### 9.3.1. Разработка API-интерфейса

Прежде чем перейти к реализации описанной технологии, определимся с применяемыми маршрутами. В нашем приложении мы будем предварять API-интерфейс для веб-службы RESTful путем /api, хотя при желании этот путь можно изменить. Например, можно использовать поддомен, такой как http://api.myapplication.com.

По следующему фрагменту кода можно понять причину, из-за которой функции обратного вызова следует помещать в отдельные Node-модули, а не определять их в качестве подставляемых (inline) с помощью вызовов app.VERB(). Единственный список маршрутов дает ясную картину того, что вы, собственно, реализовали и где находятся обратные вызовы:

```
app.get('/api/user/:id', api.user);
app.get('/api/entries/:page?', api.entries);
app.post('/api/entry', api.add);
```

### 9.3.2. Добавление базовой аутентификации

Как уже упоминалось, существует множество методик обеспечения безопасности API-интерфейса и организации необходимых ограничений, но их описание выходит

за рамки темы этой книги. Тем не менее процесс базовой аутентификации очень полезно проиллюстрировать.

Программный компонент промежуточного уровня api.auth будет абстрагировать этот процесс, а реализация останется модулю ./routes/api.js, который мы создадим позже. Как отмечено в главе 6, функции app.use() может передаваться путь, определяющий точку монтирования. Это означает, что имена запрашиваемых путей, которые начинаются с пути /api и любого HTTP-глагола, будут приводить к вызову данного программного компонента.

Строка app.use('/api', api.auth) должна располагаться перед программным компонентом промежуточного уровня, загружающим пользовательские данные, как показано в следующем фрагменте кода. В результате вы сможете в дальнейшем изменить программное обеспечение промежуточного уровня для загрузки пользовательских данных таким образом, чтобы загружать данные только для аутентифицированных пользователей API-интерфейса:

```
...
var api = require('./routes/api');
...
app.use('/api', api.auth);
app.use(user);
...
```

Далее создайте файл ./routes/api.js и сделайте загружаемыми по требованию модели express и user, как показано в следующем примере кода. Как отмечалось в главе 7, программный компонент промежуточного уровня basicAuth() использует специальную функцию для выполнения аутентификации, получая сигнатуру функции (имя пользователя, пароль, обратный вызов). В описанную схему полностью укладывается функция User.authentication:

```
var express = require('express');
var User = require('../lib/user');


exports.auth = express.basicAuth(User.authenticate);
```

Теперь, когда модуль аутентификации готов к применению, перейдем к реализации маршрутов API-интерфейса.

### 9.3.3. Реализация маршрутизации

Первым в этом разделе мы реализуем маршрут GET /api/user/:id. Соответствующая логика сначала будет пытаться найти пользователя по его идентификатору и при неудаче отвечать 404 Not Found. Если же пользователь обнаружится,

пользовательские данные будут переданы функции res.send() для сериализации, а приложение ответит JSON-представлением этих данных. В файл routes/api.js добавьте следующий фрагмент кода:

```
exports.user = function(req, res, next){
  User.get(req.params.id, function(err, user){
    if (err) return next(err);
    if (!user.id) return res.send(404);
    res.json(user);
  });
};
```

Затем добавьте маршрутный путь в файл app.js:

```
app.get('/api/user/:id', api.user);
```

Теперь все готово к тестированию приложения.

### *Тестирование механизма извлечения пользовательских данных*

Запустите приложение и протестируйте его с помощью утилиты командной строки cURL. На основе следующего фрагмента кода демонстрируется, каким образом можно тестировать REST-аутентификацию в приложении. Полномочия предоставляются в URL-адресе tobi:ferret, который утилита cURL использует для создания поля Authorization заголовка:

```
$ curl http://tobi:ferret@127.0.0.1:3000/api/user/1 -v
```

Результаты успешного тестирования приведены в листинге 9.29.

**Листинг 9.29.** Выводимые при тестировании данные

```
* About to connect() to local port 80 (#0)
*   Trying 127.0.0.1... connected
* Connected to local (127.0.0.1) port 80 (#0)
* Server auth using Basic with user 'tobi'
// Вывод на экран переданных HTTP-заголовков
> GET /api/user/1 HTTP/1.1
> Authorization: Basic Zm9vYmFyYmF6Cg==
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4
   OpenSSL/0.9.8r zlib/1.2.5
> Host: local
> Accept: */*
>
```

```
// Вывод на экран полученных HTTP-заголовков
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 150
< Connection: keep-alive
<
// Вывод на экран полученных JSON-данных
{
  "name": "tobi",
  "pass": "$2a$12$P.mzcfvmumS3MMO1EBN9wutf0Eiyw5X0VcGroeoVPGE7MLVtziYq
  "id": "1",
  "salt": "$2a$12$P.mzcfvmumS3MMO1EBN9wu"
}
```

### *Удаление конфиденциальных пользовательских данных*

Как вы можете видеть, в JSON-ответе содержатся пароль пользователя и соль Чтобы исправить ситуацию, можно реализовать метод .toJSON() для прототипа User.prototype в файле lib/user.js:

```
User.prototype.toJSON = function(){
  return {
    id: this.id,
    name: this.name
  }
};
```

Если метод .toJSON существует для объекта, его могут использовать вызовы JSON.stringify для получения JSON-данных. Если еще раз выполнит продемонстрированный ранее cURL-запрос, вы получите только свойства id и name:

```
{
  "id": "1",
  "name": "tobi"
}
```

Теперь мы займемся добавлением в API-интерфейс средства создания записей.

*Добавление записей*

Процессы добавления записей через HTML-форму и через API-интерфейс почти идентичны, поэтому, скорее всего, у вас возникнет желание повторно использовать маршрутную логику, реализованную ранее в функции entries.submit().

При добавлении записи маршрутная логика обеспечивает сохранение имени пользователя и добавляет запись к другим деталям. По этой причине вам потребуется изменить механизм загрузки пользовательских данных таким образом, чтобы заполнить объект res.locals.user пользовательскими данными, загруженными программным компонентом промежуточного уровня basicAuth. Компонент basicAuth сохраняет эти данные в свойстве объекта запроса req.remoteUser. Добавить соответствующую проверку в программный компонент промежуточного уровня, выполняющий загрузку пользовательских данных, несложно. Чтобы этот компонент работал с API-интерфейсом, просто измените определение модуля module.exports в файле lib/middleware/user.js так, как показано в следующем примере кода:

```
...
module.exports = function(req, res, next){
  if (req.remoteUser) {
    res.locals.user = req.remoteUser;
  }
  var uid = req.session.uid;
  if (!uid) return next();
  User.get(uid, function(err, user){
    if (err) return next(err);
    req.user = res.locals.user = user;
    next();
  });
};
```

После внесения этих изменений вы сможете добавлять записи через API-интерфейс.

Тем не менее было бы неплохо внести в реализацию еще одно изменение, касающееся организации дружественного к API-интерфейсу ответа вместо перенаправления на домашнюю страницу приложения. Чтобы добавить требуемую функциональность, измените вызов entry.save в файле routes/entries.js следующим образом:

```
...
```

```
entry.save(function(err) {
  if (err) return next(err);
  if (req.remoteUser) {
    res.json({message: 'Entry added.'});
  } else {
    res.redirect('/');
  }
});
...
```

И наконец, чтобы активизировать в нашем приложении API-интерфейс добавления записей, включите такую строку кода в раздел маршрутизации файла api.js:

```
app.post('/api/entry', entries.submit);
```

С помощью следующей cURL-команды можно протестировать механизм добавления записи через API-интерфейс. В нашем случае данные заголовка и тела передаются с помощью тех же имен полей, что и в HTML-форме:

```
$ curl -F entry[title]='Ho ho ho' -F entry[body]='Santa loves you'
  http://tobi:ferret@127.0.0.1:3000/api/entry
```

Теперь, когда мы наделили наше приложение способностью добавлять записи, нужно добавить ему способность извлекать данные записей.

### *Поддержка списка записей*

Следующим для API-интерфейса мы реализуем маршрут GET /api/entries/:page?. Реализация этого маршрута почти идентична реализации маршрута вывода существующих записей в виде списка из файла ./routes/entries.js. Можно использовать определенную ранее функцию промежуточного уровня page() для поддержки объекта req.page, применяемого для разбивки на страницы.

Поскольку логика маршрутизации будет получать доступ к записям, нам понадобится модель Entry, находящаяся в начале файла routes/api.js. Для загрузки этой модели используется такая строка кода:

```
var Entry = require('../lib/entry');
```

Затем добавьте следующую строку в раздел маршрутизации файла app.js:

```
app.get('/api/entries/:page?', page(Entry.count), api.entries);
```

А теперь добавьте приведенный далее код маршрутизации в файл routes/api.js. Различие между этой маршрутной логикой и похожей логикой из файла routes/entries.js связано с тем, что теперь больше не требуется визуализировать

данные с помощью шаблона, вместо этого данные выводятся в формате JSON:

```
exports.entries = function(req, res, next){
  var page = req.page;
  Entry.getRange(page.from, page.to, function(err, entries){
    if (err) return next(err);
    res.json(entries);
  });
};
```

Следующая cURL-команда запрашивает у API-интерфейса данные записей:

```
$ curl http://tobi:ferret@127.0.0.1:3000/api/entries
```

В результате выполнения этой cURL-команды на экране должны появиться примерно такие JSON-данные:

```
[
  {
    "username": "rick",
    "title": "Cats can't read minds",
   "body": "I think you're wrong about the cat thing."
  },
  {
    "username": "mike",
    "title": "I think my cat can read my mind",
    "body": "I think cat can hear my thoughts."
  },
...
```

Теперь, когда реализация API-интерфейса в основном закончена, давайте поговорим о том, как API-интерфейсы могут поддерживать несколько форматов ответа.

### 9.3.4. Согласование контента

Согласование контента — это тот механизм, который дает клиенту возможность указать, какие форматы он хочет получать и какой из них он бы предпочел. В этом разделе мы предложим потребителям нашего API-интерфейса JSON- и XMI представления API-контента, чтобы они могли выбрать тот, который им больше подходит.

В HTTP механизм согласования контента поддерживается с помощью поля

Accept заголовка. Например, если клиент предпочитает формат HTML, но в то же время не против принимать простой текст, он может воспользоваться следующим заголовком запроса:

Accept: text/plain; q=0.5, text/html

*Значение качества* (quality value, или qvalue), которое в рассматриваемом примере составляет q=0.5, указывает на то, то хотя формат text/html определен вторым, его приоритет на 50 % выше, чем приоритет формата text/plain. Express после синтаксического разбора этой информации предоставляет нормализованный массив req.accepted:

```
[{ value: 'text/html', quality: 1 },
 { value: 'text/plain', quality: 0.5 }]
```

В Express также поддерживается метод res.format(), который принимает массив MIME-типов и обратных вызовов. С его помощью Express определит, что клиент готов принять и что вы собираетесь предоставить, после чего сделает соответствующий обратный вызов.

### *Реализация механизма согласования контента*

Реализация механизма согласования контента для маршрута GET /api/entries может выглядеть примерно так, как показано в листинге 9.30. Механизм поддержки JSON-массива не изменился — записи сериализуются в JSON-массив с помощью метода res.send(). Обратный XML-вызов последовательно перебирает все записи и записывает результат в сокет. Обратите внимание, что устанавливать значение Content-Type явным образом не требуется. Функция res.format() установит соответствующий тип автоматически.

**Листинг 9.30.** Реализация механизма согласования контента

```
exports.entries = function(req, res, next){
  var page = req.page;
  // Выборка данных записи
  Entry.getRange(page.from, page.to, function(err, entries){
    if (err) return next(err);
                                       // Отвечаем по-
разному в зависимости от значения принятого заголовка
    res.format({
      'application/json': function(){
        // JSON-ответ
        res.send(entries);
```

```
      },
      'application/xml': function(){
        // XML-ответ
        res.write('<entries>\n');
        entries.forEach(function(entry){
          res.write(' <entry>\n');
          res.write(' <title>' + entry.title + '</title>\n');
          res.write(' <body>' + entry.body + '</body>\n');
          res.write(' <username>' + entry.username + '</username>\n');
          res.write(' </entry>\n');
        });
        res.end('</entries>');
      }
    })
  });
};
```

Если установить для обратного вызова формат ответа, предлагаемый по умолчанию, этот вызов будет выполняться в том случае, если пользователь явно не запросит нужный ему формат.

Метод res.format() также принимает расширение, которое проецируется на соответствующий MIME-тип. Например, вместо application/json и application/xml можно указать расширения json и xml, как в следующем примере кода:

```
...
res.format({
  json: function(){
    res.send(entries);
  },

  xml: function(){
    res.write('<entries>\n');
    entries.forEach(function(entry){
      res.write(' <entry>\n');
      res.write(' <title>' + entry.title + '</title>\n');
      res.write(' <body>' + entry.body + '</body>\n');
      res.write(' <username>' + entry.username + '</username>\n');
```

```
        res.write(' </entry>\n');
      });
      res.end('</entries>');
    }
  })
  ...
```

#### *Реализация XML-ответа*

Написать в маршруте кучу нестандартной логики для реализации XML-ответа — задача довольно сложная, и чтобы ее упростить, давайте воспользуемся системой представлений.

Создайте шаблон ./views/entries/xml.ejs с EJS-механизмом перебора записей для генерирования тегов <entry> (листинг 9.31).

**Листинг 9.31.** Использование EJS-шаблона для генерирования XML-кода

```
<entries>
// Циклический обход каждой записи
<% entries.forEach(function(entry){ %>
  <entry>
    // Вывод полей
    <title><%= entry.title %></title>
    <body><%= entry.body %></body>
    <username><%= entry.username %></username>
  </entry>
  <% }) %>
</entries>
```

Теперь обратный XML-вызов можно заменить единственным вызовом res.render(), передающим массив entries, как показано в следующем примере кода:

```
...
  xml: function(){
    res.render('entries/xml', { entries: entries });
  }
})
...
```

На данном этапе мы готовы протестировать XML-версию API-интерфейса Чтобы просмотреть выводимые XML-данные, в командной строке введите

следующую команду:

**curl -i -H 'Accept: application/xml'**
http://tobi:ferret@127.0.0.1:3000/api/entries

## 9.4. Обработка ошибок

До сих пор ни само приложение, ни API-интерфейс не отвечали страницами с ошибкой или сообщением 404 Not Found. Это означает, что при невозможности найти ресурс или разрыве соединения с базой данных Express ответит заданным по умолчанию кодом 404 или 500 соответственно. Как показано на рис. 9.8, этот ответ не слишком информативен, поэтому давайте его подправим. В этом разделе и для кода 404, и для ошибки мы создадим программный компонент промежуточного уровня, который будет отвечать данными в формате HTML, JSON или простого текста в зависимости от того, что принимает клиент.

**Рис. 9.8.** Стандартное сообщение об ошибке с кодом 404 в Connect

И начнем мы с реализации программного обеспечения промежуточного уровня для обработки ошибки с кодом 404, возникающей в случае отсутствия запрашиваемого ресурса.

### 9.4.1. Обработка ошибок с кодом 404

Как уже упоминалось, когда все программное обеспечение промежуточного уровня не отвечает, Connect по умолчанию отвечает кодом 404 и маленькой строкой в формате простого текста. Например, рассмотрим такую команду:

**$ curl http://tobi:ferret@127.0.0.1:3000/api/not/a/real/path –i**
-H "Accept: application/json"

Если запрошенной в этой команде записи не существует, Connect отвечает следующим образом:

**HTTP/1.1 404 Not Found**

```
Content-Type: text/plain
Connection: keep-alive
Transfer-Encoding: chunked

Cannot GET /api/not/a/real/path
```

Иногда подобный ответ может быть вполне приемлемым, но в идеале API-интерфейс для формата JSON должен отвечать в формате JSON:

```
$ curl http://tobi:ferret@127.0.0.1:3000/api/not/a/real/path
  -i -H "Accept: application/json"
```

Ответ на эту команду приведен в следующем фрагменте кода:

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Content-Length: 37
Connection: keep-alive

{ "message": "Resource not found" }
```

В реализации программного компонента промежуточного уровня для обработки ошибки с кодом 404 нет ничего особенного, подобная функциональность поддерживается как в Connect, так и в Express. Это обычная функция промежуточного уровня, которая вызывается в последнюю очередь. Если очередь до нее доходит, можно предположить, что никакой другой компонент так и не ответил, поэтому можете смело идти дальше и визуализировать шаблон или реагировать любым другим способом, который сочтете нужным.

HTML-ответ на ошибку с кодом 404, который мы собираемся создать, показан на рис. 9.9.

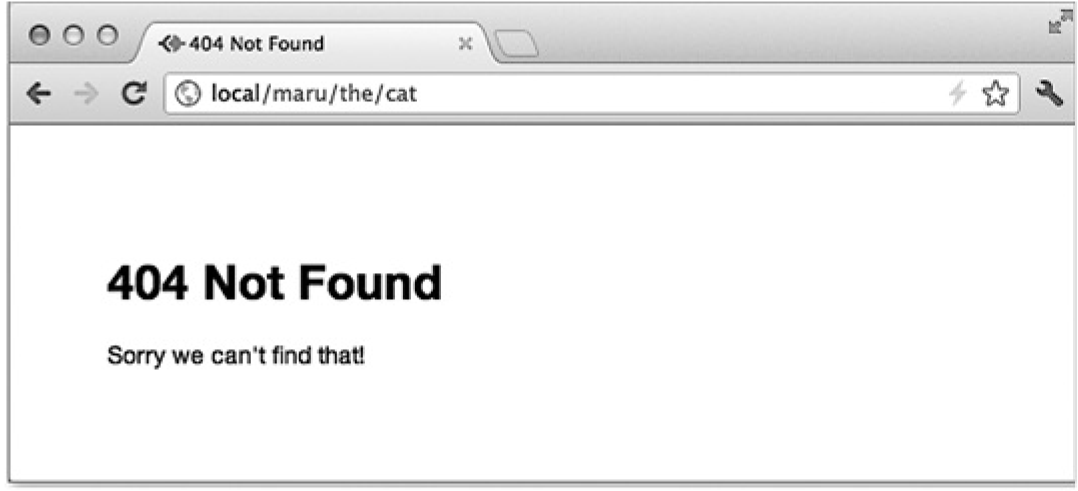



**Рис. 9.9.** Такое сообщение об ошибке с кодом 404 легче для восприятия, чем стандартное для Connect сообщение об ошибке

*Добавление маршрута для возвращения ответа на ошибку*

Откройте файл ./routes/index.js. На данный момент этот файл содержит лишь оригинальную функцию exports.index, сгенерированную командой express(1). Можете спокойно избавиться от этой функции, поскольку ее уже заменила функция entries.list.

Реализация функции ответа на ошибки зависит от потребностей приложения. В следующем примере кода применяется метод согласования контента res.format(), который в зависимости от предпочтений клиента предоставляет ему ответы text/html, application/json и text/plain. Метод ответа res.status(code) идентичен установке в Node свойства res.statusCode = code, однако он может выстраиваться в цепочку, как вы можете видеть в следующем примере кода по непосредственному вызову .format().

**Листинг 9.32.** Логика маршрута вида Not Found

```
exports.notfound = function(req, res){
  res.status(404).format({
    html: function(){
      res.render('404');
    },
    json: function(){
      res.send({ message: 'Resource not found' });
    },
    xml: function() {
      res.write('<error>\n');
      res.write(' <message>Resource not found</message>\n');
      res.end('</error>\n');
    },
    text: function(){
      res.send('Resource not found\n');
    }
  });
};
```

*Создание шаблона страницы ошибки*

Мы до сих пор еще не разработали шаблон для ошибки с кодом 404, поэтому создайте новый файл ./views/404.ejs, содержащий фрагмент EJS-кода из листинга

9.33. Дизайн этого шаблона — на ваше усмотрение.

**Листинг 9.33.** Пример страницы ошибки с кодом 404

```
<!DOCTYPE html>
<html>
  <head>
    <title>404 Not Found</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>

    <h1>404 Not Found</h1>
    <p>Sorry we can't find that!</p>
  </body>
</html>
```

***Подключение программного обеспечения промежуточного уровня***

Добавьте программный компонент промежуточного уровня routes.notfound следом за всеми остальными, и вы сможете обрабатывать ошибки с кодом 404 так, как пожелаете:

```
...
app.use(app.router);
app.use(routes.notfound);
...
```

Теперь, когда вы можете обрабатывать ошибки с кодом 404, давайте реализуем собственный программный компонент промежуточного уровня для обработки ошибок, призванный улучшить помощь пользователям в случае ошибок.

### 9.4.2. Обработка ошибок

До сих пор ошибки передавались функции next(). Однако в этом случае Connect по умолчанию будет откликаться заезженным ответом 500 Internal Server Error, который во многом подобен заданному по умолчанию ненавязчивому ответу 404. Как правило, передавать клиенту детальные сведения об ошибке не рекомендуется, поскольку это связано с потенциальной угрозой безопасности, но в данном случае предлагаемый по умолчанию ответ на ошибку очевидно бесполезен как для

потребителей вашего API-интерфейса, так и для посетителей, глядящих на него в браузере.

В этом разделе мы создадим обобщенный шаблон 5xx, с помощью которого будем отвечать клиентам в случае ошибки. Этот шаблон должен предлагать клиентам HTML-ответ, если они принимают формат HTML, и JSON-ответ, если он принимают формат JSON, как, например, потребители API-интерфейса.

Программное обеспечение промежуточного уровня может находиться где угодно, в нашем примере оно находится в файле ./routes/index.js вместе с функцией ошибки с кодом 404. Ключевое отличие функции промежуточного уровня exports.error заключается в том, что она принимает четыре параметра. Как отмечалось в главе 6, программный компонент промежуточного уровня для обработки ошибок должен принимать ровно четыре параметра.

### *Тестирование страниц ошибок с помощью условного маршрута*

Если ваше приложение надежно, вызвать ошибку будет довольно сложно. В этом случае удобно использовать *условные* (conditional) маршруты. Эти маршруты можно задействовать только с помощью флага конфигурации, переменной окружения или типа окружения (возможно, при разработке).

Следующий пример кода из файла app.js иллюстрирует возможность добавления маршрута /dev/error в приложение только в том случае, если указана переменная окружения ERROR_ROUTE. При этом создается фиктивная ошибка с помощью произвольного свойства err.type. Добавьте следующий код в раздел маршрутизации файла app.js:

```
if (process.env.ERROR_ROUTE) {
  app.get('/dev/error', function(req, res, next){
    var err = new Error('database connection failed');
    err.type = 'database';
    next(err);
  });
}
```

После этого можете запустить приложение, задав дополнительный маршрут с помощью показанной далее команды. Если вам интересно, что получится, в адресной строке браузера введите адрес /dev/error. Этот адрес предназначен только для тестирования обработчика ошибок:

```
$ ERROR_ROUTE=1 node app
```

***Реализация обработчика ошибок***

Чтобы реализовать обработчик ошибок в файле ./routes/index.js, код в листинге 9.34 начинается с вызова console.error(err.stack). Возможно, это самая главная строка в указанной функции. Она гарантирует, что когда ошибка, распространяющаяся в среде Connect, достигнет данной функции, вы тут же об этом узнаете. Причем сообщение об ошибке и трасса стека будут записаны в потоке ошибок (stderr) для последующего изучения.

**Листинг 9.34.** Обработчик ошибок, поддерживающий согласование контента

```
// Обработчики ошибок должны принимать четыре аргумента
exports.error = function(err, req, res, next){
  //  Запись ошибки в поток stderr
  console.error(err.stack);
  var msg;

  // Примеры ошибок специального вида
  switch (err.type) {
    case 'database':
      msg = 'Server Unavailable';
      res.statusCode = 503;
      break;
    default:
      msg = 'Internal Server Error';
      res.statusCode = 500;
  }
  res.format({
    // Визулизация шаблона, если принимается формат HTML
    html: function(){
      res.render('5xx', { msg: msg, status: res.statusCode });
    },

    // JSON-ответ, если принимается формат JSON
    json: function(){
      res.send({ error: msg });
    },
```

```
    // Ответ в формате простого текста
    text: function(){
      res.send(msg + '\n');
    }
  });
};
```

**оповещения об ошибках в ПРИЛОЖЕНИИ**

С помощью разработанного в этом разделе универсального обработчика ошибок можно решать некоторые дополнительные задачи, связанные с обработкой ошибок: например, можно известить команду разработчиков о возникших проблемах. Попробуйте самостоятельно протестировать этот обработчик. Загрузите один из модулей электронной почты от независимого производителя и создайте программный компонент промежуточного уровня, который будет отправлять оповещения по электронной почте, а затем вызывать функцию next(err), чтобы передать ошибку остальным программным компонентам обработки ошибок.

Чтобы предоставить более информативный ответ пользователю, но в то же время не экспонировать слишком подробные сведения об ошибке, можно было бы проверять свойства ошибки и отвечать соответственно. В нашем случае свойство err.type, которое мы добавили к маршруту /dev/error, проверяется с целью требуемым образом адаптировать сообщение об ошибке, после чего мы отвечаем в формате HTML, JSON или простого текста, что во многом напоминает обработчик ошибки с кодом 404.

### *Создание шаблона страницы ошибки*

EJS-шаблон для вызова res.render('5xx') , представленный в листинге 9.35, будет находиться в файле ./views/5xx.ejs.

**Листинг 9.35.** Пример страницы ошибки с кодом 500

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= status %> <%= msg %></title>
```

```
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>

    <h1><%= status %> Error</h1>
    <p><%= msg %></p>
    <p>
      Try refreshing the page, if this problem
          persists then we're already working on it!
    </p>
  </body>
</html>
```

***Подключение программного обеспечения промежуточного уровня***

Отредактируйте файл app.js, поместив код программного компонента промежуточного уровня routes.error после всех остальных компонентов, даже после компонента routes.notfound. Так вы обеспечите перехват компонентом routes.error всех ошибок в Connect, даже возможных ошибок компонента routes.notfound:

```
...
app.use(app.router);
app.use(routes.notfound);
app.use(routes.error);
});
```

Снова вызовите приложение с установленной переменной окружения ERROR_ROUTE. Обратите внимание на новую страницу ошибки (рис. 9.10).

Итак, мы только что создали полнофункциональное приложение для чата и параллельно изучили на практике некоторые важные приемы разработки приложений в Express.

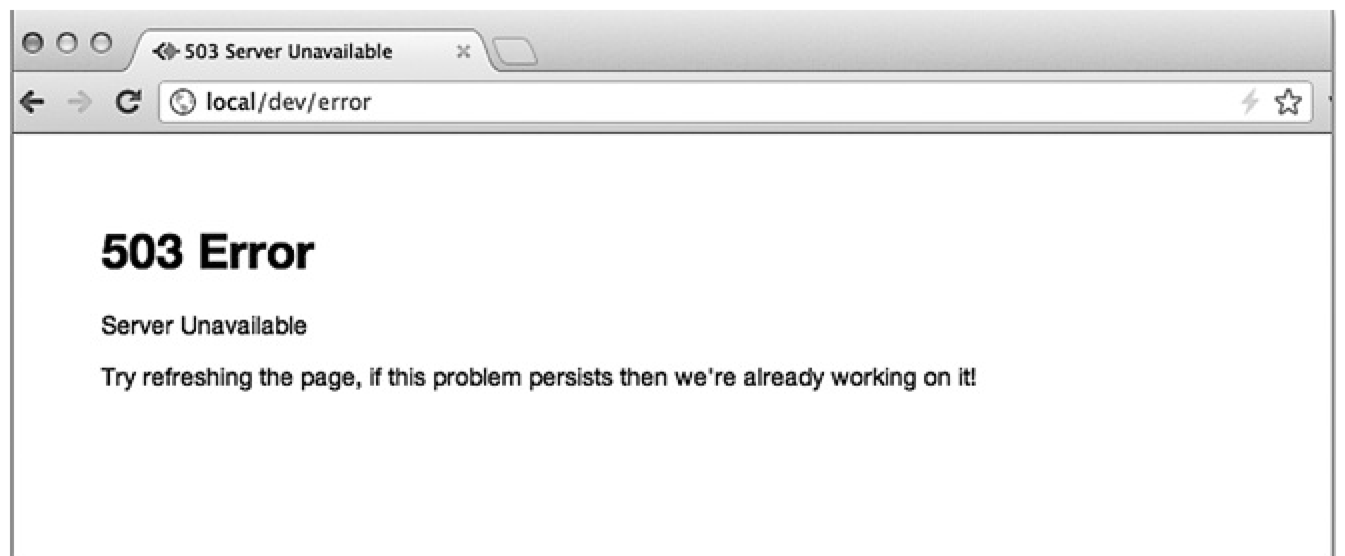



**Рис. 9.10.** Страница ошибки

### 9.5. Резюме

В этой главе мы создали простое веб-приложение, продемонстрировавшее многие возможности среды Express, о которых не было упомянуто в предыдущей главе. Методики, освоенные при чтении этой главы, помогут нам двигаться дальше по пути разработки веб-приложений.

Сначала мы создали многоцелевую систему аутентификации и регистрации пользователей, которая с помощью пользовательских сеансов сохраняет идентификаторы пользователей, вошедших в систему, а также любые предназначенные для пользователей системные сообщения.

Затем с помощью программного обеспечения промежуточного уровня мы задействовали систему аутентификации для создания API-интерфейса REST. Это API-интерфейс экспонирует для разработчиков выбранные данные приложения, причем после согласования контента делает их доступными в формате JSON или XML.

Получив навыки создания веб-приложений в последних двух главах, мы подготовились к изучению темы автоматизированного тестирования, которое может быть полезно для всего, что разрабатывается на платформе Node.

## Глава 10. Тестирование Node-приложений

- Тестирование Node-кода с помощью модуля assert
- Использование в Node сред модульного тестирования
- Эмулирование и контролирование веб-браузеров с помощью Node

По мере наделения нашего приложения различными функциональными средствами возрастает риск введения в него ошибок. Без тестирования приложение считается незавершенным, а поскольку ручное тестирование утомительно и чревато введением новых ошибок, обусловленных человеческим фактором, у разработчиков все более популярным становится автоматизированное тестирование. В этом случае вместо проверки функциональности приложения посредством его запуска вручную пишется логика автоматизированного тестирования кода.

Если вы ранее не сталкивались с концепцией автоматизированного тестирования, представьте себе робота, который вместо вас выполняет всю рутинную работу, предоставляя вам возможность заняться более интересными вещами. Тогда при внесении изменений в код вы будете просто сообщать о них роботу, который проверит, не вкрались ли туда ошибки. Даже если вы еще не закончили разработку приложения или же только начали создавать свое первое Node-приложение, вам полезно знать, как реализовать автоматизированное тестирование, поскольку тогда вы сможете писать тесты по мере разработки приложения.

В этой главе мы рассмотрим два типа автоматизированного тестирования: модульное и приемочное. При модульном тестировании происходит непосредственная проверка логики приложения, обычно на уровне функции или метода; модульное тестирование применимо ко всем типам приложений. В плане методологии модульное тестирование может быть разделено на две основные категории: разработка на основе тестирования (Test-Driven Development, TDD) и разработка на основе функционирования (Behavior-Driven Development, BDD). Вообще говоря, с практической точки зрения тесты в стиле TDD и BDD почти во всем одинаковы. Главное отличие заключается в языке описания тестов, с которым мы познакомимся на нескольких примерах. Между тестами в стиле TDD и BDD существуют и некоторые другие отличия, но их описание выходит за рамки темы этой книги.

Приемочное тестирование, представляющее собой дополнительный уровень тестирования, применяется преимущественно при отладке веб-приложений. Этот вид тестирования подразумевает контроль сценариев из браузера и проверку функциональности веб-приложений с его помощью.

Мы рассмотрим готовые решения, иллюстрирующие модульное и приемочное тестирование. В рамках модульного тестирования мы познакомимся с Node-модулем assert и со средами Mocha, nodeunit, Vows и should.js, а в рамках приемочного — со средами Tobi и Soda. Инструменты тестирования вместе с соответствующими методиками и подходами представлены на рис. 10.1.

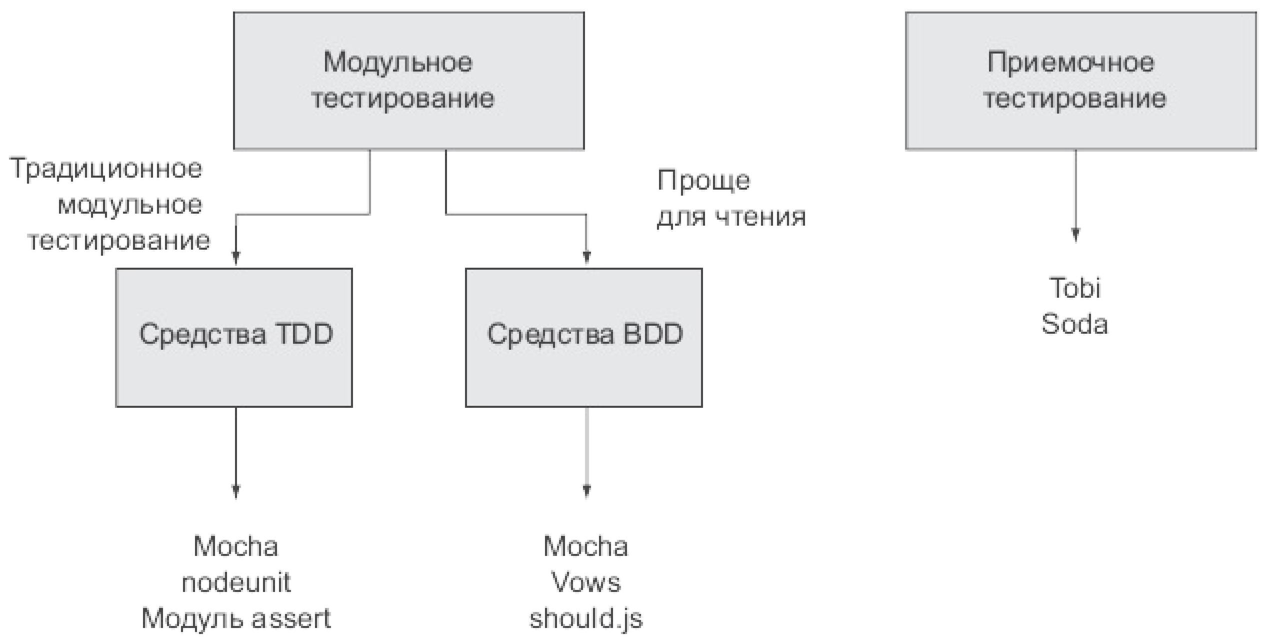



**Рис. 10.1.** Обзор сред тестирования приложений

И начнем мы с модульного тестирования.

## 10.1. Модульное тестирование

Модульное тестирование — это такая разновидность автоматического тестирования, в ходе которого проверяются отдельные части приложения. Благодаря тестам вы сможете более критично оценить дизайн приложения и избежать ловушек на ранних этапах разработки. Использование тестов позволит вам гарантировать, что последние изменения не ведут к появлению ошибок. Хотя написание модульных тестов требует определенного времени, вы сможете сэкономить время в дальнейшем, поскольку вам не придется вручную проверять приложение после внесения изменений.

Модульное тестирование может быть довольно сложным, причем асинхронная логика все еще больше усложняет. Асинхронные модульные тесты могут выполняться параллельно, поэтому при разработке тестов следует проявлять осторожность и следить, чтобы тесты не влияли друг на друга. Например, если тесты создают временные файлы на диске, будьте осторожны при удалении этих файлов после завершения каждого теста, чтобы не удалить рабочие файлы другого теста, который еще не завершился. Учитывая подобные особенности, многие среды модульного тестирования предусматривают средства управления порядком выполнения тестов.

В этом разделе мы рассмотрим:

- *встроенный в Node модуль assert* — неплохой выбор для пользователей, осваивающих автоматизированное тестирование в стиле TDD;
- *nodeunit* — безусловный фаворит у Node-сообщества среди сред тестирования в стиле TDD;
- *Mocha* — относительно новая среда тестирования в стиле TDD или BDD;
- *Vows* — широко используемая среда тестирования в стиле BDD;
- *should.js* — модуль, который надстраивается над Node-модулем assert и предназначен для тестирования в стиле BDD.

А начнем мы со знакомства с модулем assert, входящим в комплект поставки Node.

### 10.1.1. Модуль assert

В большинстве случаев модульное тестирование Node-приложений выполняется с помощью встроенного модуля assert. При этом выполняется проверка некоего условия, и если условие не выполняется, генерируется ошибка. Node-модуль assert способен использовать возможности многих сред тестирования сторонних производителей, но даже при их отсутствии он может быть весьма полезен для тестирования.

#### *Простой пример*

Предположим, что в вашем распоряжении имеется простое приложение, поддерживающее в памяти список запланированных дел. Это приложение нужно протестировать и убедиться, что все работает так, как нужно.

В коде листинга 10.1 определяется модуль, содержащий основную функциональность приложения. Логика модуля поддерживает создание, извлечение и удаление элементов списка запланированных дел. В этот модуль входит также простой метод doAsync, поэтому мы, в частности, можем видеть, как тестируются асинхронные методы. Присвоим этому файлу имя todo.js.

**Листинг 10.1.** Модель для списка запланированных дел

```
// Определяем базу данных для списка запланированных дел
function Todo () {
  this.todos = [];
```

```
}

// Добавляем элемент в список
Todo.prototype.add = function (item) {
  if (!item) throw new Error('Todo#add requires an item')
  this.todos.push(item);
}

// Удаляем все элементы из списка
Todo.prototype.deleteAll = function () {
  this.todos = [];
}

// Подсчитываем количество элементов списка
Todo.prototype.getCount = function () {
  return this.todos.length;
}

// Через 2 секунды обратный вызов со значением "true"
Todo.prototype.doAsync = function (cb) {
  setTimeout(cb, 2000, true);
}

// Экспорт функции Todo
module.exports = Todo;
```

Теперь можно воспользоваться Node-модулем assert для тестирования кода. В файл test.js введите следующий код из листинга 10.2 для загрузки требуемых модулей, создайте новый список запланированных дел и присвойте значение переменной, которая будет отражать ход тестирования.

**Листинг 10.2.** Настройка требуемых модулей

```
var assert = require('assert');
var Todo = require('./todo');
var todo = new Todo();
var testsCompleted = 0;
```

### *Тестирование значения переменной с помощью утверждения equal*

Теперь можно добавить тест, проверяющий функциональность удаления элемента из списка запланированных дел.

Обратите внимание на утверждение equal, которое является одним из наиболее часто применяемых утверждений модуля assert (листинг 10.3). С его помощью проверяется, действительно ли значение переменной равно значению второго аргумента. В рассматриваемом примере создается элемент списка запланированных дел, а затем все элементы списка удаляются.

**Листинг 10.3.** Тест на наличие элементов в списке запланированных дел после удаления

```
function deleteTest () {
  // Добавляем некие данные с целью тестирования
  // процедуры удаления
  todo.add('Delete Me');
  // Утверждаем, что данные были добавлены правильно
  assert.equal(todo.getCount(), 1, '1 item should exist');
  // Удаляем все записи
  todo.deleteAll();
  // Утверждаем, что запись была удалена
  assert.equal(todo.getCount(), 0, 'No items should exist');
  // Сообщаем о завершении теста
  testsCompleted++;
}
```

Если логика приложения работает должным образом, после завершения теста все элементы списка запланированных дел будут удалены, а значение переменной todo.getCount() станет равным 0. Если же возникают какие-либо проблемы, генерируется исключение. Когда значение переменной todo.getCount() оказывается ненулевым, результатом утверждения становится вывод на консоли трассы стека с сообщением об ошибке «No items should exist». После утверждения значение переменной testsCompleted увеличивается на единицу, показывая, что тест пройден успешно.

### *Тестирование логики на предмет возможных проблем с помощью утверждения notEaqual*

В качестве следующего шага добавьте в файл test.js код из листинга 10.4. С

помощью этого кода проверяются функциональность приложения, реализующая добавление элементов в список запланированных дел.

**Листинг 10.4.** Тест на функциональность добавления элементов в список запланированных дел

```
function addTest () {
  // Удаляем все существующие элементы
  todo.deleteAll();
  // Добавляем элемент
  todo.add('Added');
  // Утверждаем, что элементы существуют
  assert.notEqual(todo.getCount(), 0, '1 item should exist');
  // Сообщаем о завершении теста
  testsCompleted++;
}
```

Как видите, модуль assert допускает также утверждения notEqual. Этот тип утверждений применяется в тех случаях, когда код приложения генерирует определенное значение, свидетельствующее о наличии проблем в логике приложения.

В листинге 10.4 демонстрируется использование утверждения notEqual. Все элементы списка удаляются, элемент добавляется, затем логика приложения получает все элементы. Если количество элементов равно нулю, значит, утверждение несостоятельно и в результате генерируется исключение.

### *Использование дополнительных утверждений strictEqual, notStrictEqual, deepEqual, notDeepEqual*

Помимо утверждений equal и notEqual, модуль assert предлагает строгие (strict) версии утверждений, strictEqual и notStrictEqual. В этих утверждениях используется оператор строгого равенства (===) вместо более «свободной» версии (==).

Для сравнения объектов применяются утверждения deepEqual и notDeepEqual из модуля assert. Слово «deep» (глубокий), используемое в названиях утверждений, означает, что выполняется рекурсивное сравнение двух объектов, то есть сравниваются свойства двух объектов, а если свойства сами являются объектами, они тоже сравниваются.

### *Тестирование асинхронных значений на истинность с помощью утверждения ok*

Теперь пришло время протестировать метод doAsync в приложении списка запланированных дел (листинг 10.5). Поскольку тест является асинхронным, с помощью функции обратного вызова (cb) мы сигнализируем исполнителю о завершении теста. В данном случае мы не можем полагаться на функцию, возвращающую результаты, как в случае с синхронными тестами. Чтобы удостовериться в том, что значение doAsync равно true, применятся утверждение ok, которое предлагает простой способ удостовериться в том, что это значение равно true.

**Листинг 10.5.** Тест на передачу значения true при обратном вызове doAsync

```
function doAsyncTest (cb) {
  // Обратный вызов произойдет через 2 секунды
  todo.doAsync(function (value) {
    // Утверждаем, что значение истинно
    assert.ok(value,'Callback should be passed true');
    // Сообщаем о завершении теста
    testsCompleted++;
    // По завершении делаем обратный вызов
    cb();
  })
}
```

***Тестирование корректности механизма генерирования ошибок***

С помощью модуля assert можно также проверить корректность генерирования сообщений об ошибках, как показано в листинге 10.6. Второй аргумент вызова throws — это регулярное выражение, которое ищет в сообщении об ошибке текст «requires».

**Листинг 10.6.** Тест на корректность генерирования ошибки при отсутствии параметра

```
function throwsTest (cb) {
  // Вызываем todo.add без параметра
  assert.throws(todo.add, /requires/);
  // Сообщаем о завершении теста
  testsCompleted++
}
```

*Добавление логики запуска тестов*

После определения тестов можно добавить в файл логику запуска каждого теста. Код, приведенный в листинге 10.7, будет запускать каждый тест, а затем печатать, сколько тестов запущено и завершено.

**Листинг 10.7.** Выполнение тестов и информирование о завершении тестов

```
deleteTest();
addTest();
throwsTest();
doAsyncTest(function () {
  // Информируем о завершении
  console.log('Completed ' + testsCompleted + ' tests');
})
```

Для запуска тестов воспользуйтесь следующей командой:

```
$ node test.js
```

Если тесты не провалятся, сценарий проинформирует вас о количестве пройденных тестов. Этот сценарий достаточно «интеллектуален», чтобы отслеживать время запуска и завершения тестов во избежание проблем, связанных с некорректным выполнением отдельных тестов. Например, при прохождении теста программа может не достигнуть утверждения.

Чтобы использовать встроенную в Node функциональность, каждый вариант теста должен сопровождаться массой сопутствующих операций, в ходе которых тест нужно подготовить (например, удалить все элементы) и получить трассу его выполнения (счетчик «завершенности»). Все это отвлекает того, кто пишет варианты тестов, от главного, поэтому лучше воспользоваться специальной средой тестирования, которая выполнит за вас всю «черновую» работу, предоставив вам возможность заняться собственно логикой тестирования. Давайте посмотрим, как можно облегчить себе жизнь с помощью среды модульного тестирования nodeunit от независимого производителя.

### 10.1.2. Среда nodeunit

Благодаря средам модульного тестирования задача тестера существенно упрощается. Обычно среды тестирования отслеживают, сколько тестов выполняется, что упрощает одновременное выполнение нескольких сценариев тестирования.

Node-сообщество создало несколько превосходных сред тестирования. Знакомство с ними мы начинаем со среды nodeunit

(https://github.com/caolan/nodeunit), поскольку это испытанное временем решение, предназначенное для тех разработчиков Node-приложений, которые предпочитают тестирование в стиле TDD. Утилита командной строки nodeunit позволит выполнить для вашего приложения все тесты и покажет, сколько тестов запущено и провалено, избавив вас от необходимости специально для вашего приложения разрабатывать собственную утилиту тестирования.

В этом разделе мы узнаем, как с помощью nodeunit писать тесты, позволяющие тестировать как код Node-приложений, так и клиентский код, запускаемый с помощью веб-браузера. Кроме того, мы узнаем, как nodeunit решает проблему сохранения трассы выполнения асинхронных тестов.

### *Установка nodeunit*

Чтобы установить nodeunit, выполните следующую команду:

```
$ npm install -g nodeunit
```

После выполнения этой команды вам будет доступна новая команда nodeunit вместе с одной или несколькими папками или файлами с тестами, указываемыми в качестве аргумента. Команда nodeunit позволяет выполнять все сценарии с расширением .js, находящиеся в переданных папках.

### *Тестирование Node-приложений с помощью nodeunit*

Чтобы добавить nodeunit-тесты в проект, создайте для них папку (обычно под названием test). Каждый тестовый сценарий должен заполнить тестами объект exports.

Вот пример тестового nodeunit-файла для сервера:

```
exports.testPony = function(test) {
  var isPony = true;
  test.ok(isPony, 'This is not a pony.');
  test.done();
}
```

Обратите внимание, что в этом тестовом сценарии ни один модуль не объявлен как загружаемый по требованию. Среда nodeunit автоматически включает методы Node-модуля assert в объект, который передает эти методы каждой функции, экспортируемой тестовым сценарием. В нашем примере это объект test.

После завершения каждой функции, экспортируемой тестовым сценарием, должен вызываться метод done. Если этот метод не вызывается, тест оповещает об этом сообщением об ошибке «Undone tests». Таким образом среда nodeunit

контролирует, чтобы все запущенные тесты были завершены.

Кроме того, полезно проверять, чтобы все утверждения в тесте были соблюдены. Почему же утверждения могут не соблюдаться? При написании тестовых модулей всегда есть опасность ошибиться в логике тестирования, что ведет к необоснованной самоуверенности. Логика тестирования может быть написана так, что какое-то утверждение вообще никогда не будет оцениваться. В следующем примере кода показано, что при вызове метод test.done() может сообщить об успехе, хотя одно из утверждений оценено не было:

```
exports.testPony = function(test) {
  if (false) {
    test.ok(false, 'This should not have passed.');
  }
  test.ok(true, 'This should have passed.');
  test.done();
}
```

Чтобы избежать подобных проблем, можно реализовать счетчик утверждений вручную, как показано в листинге 10.8.

**Листинг 10.8.** Подсчет утверждений вручную

```
exports.testPony = function(test) {
  // Счетчик утверждений
  var count = 0;
  if (false) {
    test.ok(false, 'This should not have passed.');
    // Увеличиваем счетчик утверждений на единицу
    count++;
  }
  test.ok(true, 'This should have passed.');
  // Увеличиваем счетчик утверждений на единицу
  count++;
  // Проверяем счетчик утверждений
  test.equal(count, 2, 'Not all assertions triggered.');
  test.done();
}
```

Однако все это хлопотно. Среда nodeunit предлагает более изящный инструмент решения этой задачи — метод test.expect, который позволяет задать количество утверждений в каждом тесте. В результате удается убрать пару лишних строк кода:

```
exports.testPony = function(test) {
  test.expect(2);
  if (false) {
    test.ok(false, 'This should not have passed.');
  }
  test.ok(true, 'This should have passed.');
  test.done();
}
```

Помимо тестирования Node-модулей, nodeunit позволяет тестировать клиентский JavaScript-код, предлагая вам единый инструмент тестирования ваших веб-приложений. Для подробного ознакомления с этой и более современными технологиями просмотрите онлайновую документацию nodeunit по адресу https://github.com/caolan/nodeunit.

Теперь, после знакомства со средой nodeunit, позволяющей тестировать модули в стиле TDD, давайте выясним, как встроить в модульное тестирование стиль BDD.

### 10.1.3. Среда Mocha

Mocha является самой современной средой тестирования из описываемых в этой главе, к тому же ее несложно освоить. Хотя по умолчанию эта среда выполняет тестирование в стиле BDD, с ее помощью можно также тестировать приложения в стиле TDD. Mocha включает широкий набор средств, позволяющих, в частности, обнаруживать утечки глобальных переменных, кроме того, как и nodeunit, Mocha поддерживает тестирование серверных приложений.

По умолчанию логику Mocha-тестов определяют и задают BDD-функции describe, it, before, after, beforeEach и afterEach. В качестве альтернативы можно использовать TDD-интерфейс Mocha, который предусматривает замену describe на suite, it на test, before на setup и after на teardown. В нашем примере мы ограничимся заданным по умолчанию BDD-интерфейсом.

#### выявление утечек глобальных переменных

Надобность в глобальных переменных, которые доступны во всем приложении, возникает не часто, к тому же согласно передовой практике программирования их количество нужно стремиться минимизировать. Однако в JavaScript очень просто случайно создать глобальную переменную, просто забыв поставить ключевое слово var при объявлении переменной. Mocha помогает обнаруживать случайные утечки глобальных переменных, генерируя ошибку, когда при тестировании обнаруживает

команду создания глобальной переменной.

Чтобы отключить режим обнаружения утечек глобальных переменных, в командной строке после команды mocha укажите параметр —ignored-leaks. Если же вы хотите указать допустимое число используемых глобальных переменных, перечислите их через запятую после параметра командной строки –globals.

### *Тестирование Node-приложений с помощью Mocha*

Давайте создадим небольшой проект memdb, представляющий собой компактную базу данных в памяти, а затем с помощью Mocha протестируем эту базу данных. Сначала для проекта нужно создать файлы и папки:

```
$ mkdir -p memdb/test
$ cd memdb
$ touch index.js
$ touch test/memdb.js
```

После создания папки test, предназначенной для хранения тестов, нужно установить среду Mocha:

```
$ npm install -g mocha
```

По умолчанию Mocha будет использовать BDD-интерфейс. В листинге 10.9 показано, как он выглядит.

**Листинг 10.9.** Базовая структура Mocha-теста

```
var memdb = require('..');

describe('memdb', function(){
  describe('.save(doc)', function(){
    it('should save the document', function(){
    });
  });
});
```

Mocha поддерживает также интерфейсы в стиле TDD, qunit и exports, которые подробно описаны на веб-сайте проекта (http://visionmedia.github.com/mocha). Чтобы проиллюстрировать концепцию использования различных интерфейсов, взгляните на интерфейс exports:

```
module.exports = {
```

```
  'memdb': {
    '.save(doc)': {
      'should save the document': function(){
      }
    }
  }
}
```

Все эти интерфейсы предлагают одинаковую функциональность, но мы ограничимся BDD-интерфейсом и напишем наш первый тест, код которого приведен в листинге 10.10. Поместите его в файл test/memdb.js. В этом тесте для выполнения утверждений используется Node-модуль assert.

**Листинг 10.10.** Описание функциональности memdb.save

```
var memdb = require('..');
var assert = require('assert');

// Описание функциональности memdb
describe('memdb', function(){
  // Описание функциональности метода .save()
  describe('.save(doc)', function(){
    // Описание ожиданий
    it('should save the document', function(){
      var pet = { name: 'Tobi' };
      memdb.save(pet);
      var ret = memdb.first({ name: 'Tobi' });
      // Утверждаем, что домашнее животное найдено
      assert(ret == pet);
    })
  })
})
```

Чтобы проводить тесты, вам осталось запустить Mocha. Эта среда тестирования обращается к папке ./test, в которой по умолчанию находятся исполняемые JavaScript-файлы. Однако поскольку метод .save() не реализован, единственный определенный к настоящему времени тест провалится (рис. 10.2).

А теперь сделаем так, чтобы успешно пройти тест. Добавьте код из листинга 10.11 в файл index.js.

**Листинг 10.11.** Добавление функциональности save

```
var db = [];

exports.save = function(doc){
  // Добавляем документ в массив базы данных
  db.push(doc);
};

exports.first = function(obj) {
  // Извлекаем документы, которые соответствуют каждому свойству в obj
  return db.filter(function(doc){
    for (var key in obj) {
      // Соответствия нет; возвращаем false
      // и не выбираем этот документ
      if (doc[key] != obj[key]) {
        return false;
      }
    }
    // Соответствие есть; возвращаем и выбираем документ
    return true;
  // Нужен только первый документ или null
  }).shift();
};
```

```
wavded@dev: ~/Projects/memdb
wavded@dev ~/Projects/memdb» mocha

  .

  ✖ 1 of 1 test failed:

  1) memdb .save(doc) should save the document:
     TypeError: Object #<Object> has no method 'save'
      at Context.<anonymous> (/home/wavded/Projects/memdb/test/memdb.js:8:13)
      at Test.Runnable.run (/usr/local/lib/node_modules/mocha/lib/runnable.js:184:32)
      at Runner.runTest (/usr/local/lib/node_modules/mocha/lib/runner.js:300:10)
      at Runner.runTests.next (/usr/local/lib/node_modules/mocha/lib/runner.js:346:12)
      at next (/usr/local/lib/node_modules/mocha/lib/runner.js:228:14)
      at Runner.hooks (/usr/local/lib/node_modules/mocha/lib/runner.js:237:7)
      at next (/usr/local/lib/node_modules/mocha/lib/runner.js:185:23)
      at Runner.hook (/usr/local/lib/node_modules/mocha/lib/runner.js:205:5)
      at process.startup.processNextTick.process._tickCallback (node.js:244:9)


wavded@dev ~/Projects/memdb» _
```

**Рис. 10.2.** Проваленный Mocha-тест

Снова с помощью Mocha запустите тест, который на этот раз завершится успешно (рис. 10.3).

```
wavded@dev: ~/Projects/memdb
wavded@dev ~/Projects/memdb» mocha

  .

  ✓ 1 test complete (2ms)

wavded@dev ~/Projects/memdb» _
```

**Рис. 10.3.** Успешный Mocha-тест

### *Определение логики установки и очистки с помощью Mocha-хуков*

В данном варианте теста предполагается, что memdb.first() функционирует требуемым образом, поэтому можно добавить еще несколько вариантов тестов с ожиданиями, определенными с помощью функции it(). В измененном файле теста, код которого приведен в листинге 10.12, используется новая для Mocha концепция — *хуки* (hooks). Например, BDD-интерфейс экспонирует хуки beforeEach(), afterEach(), before() и after(), принимающие обратные вызовы. Это позволяет определять логику установки и очистки до и после вариантов и групп тестов, определенных в блоках describe().

**Листинг 10.12.** Добавление хука beforeEach

```
var memdb = require('..');
```

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

describe('memdb', function(){
  beforeEach(function(){
    // Очищаем базу данных перед выполнением каждого варианта
    // теста, чтобы тесты не зависели от состояния
    memdb.clear();
  })

  describe('.save(doc)', function(){
    it('should save the document', function(){
      var pet = { name: 'Tobi' };
      memdb.save(pet);
      var ret = memdb.first({ name: 'Tobi' });
      assert(ret == pet);
    })
  })

  describe('.first(obj)', function(){
    // Первое ожидание для метода .first()
    it('should return the first matching doc', function(){
      var tobi = { name: 'Tobi' };
      var loki = { name: 'Loki' };

      // Сохраняем два документа
      memdb.save(tobi);
      memdb.save(loki);

      // Проверяем корректность каждого возвращаемого документа
      var ret = memdb.first({ name: 'Tobi' });
      assert(ret == tobi);

      var ret = memdb.first({ name: 'Loki' });
      assert(ret == loki);
```

```
  })

  // Второе ожидание для метода .first()
  it('should return null when no doc matches', function(){
    var ret = memdb.first({ name: 'Manny' });
    assert(ret == null);
  })
 })
})
```

В идеальном случае варианты тестов вообще не должны зависеть от состояния. Чтобы добиться этого с базой данных memdb, нужно просто удалить все документы, реализовав метод .clear() в файле index.js:

```
exports.clear = function(){
  db = [];
};
```

Если запустить Mocha снова, вы увидите, что все три теста пройдены.

### *Тестирование асинхронной логики*

До сих пор мы еще не тестировали асинхронную логику в Mocha. Чтобы посмотреть, как это делается, внесите небольшое изменение в одну из функций, которые были ранее определены в файле index.js. Если изменить функцию save, как показано в следующем фрагменте, обратный вызов будет выполняться после небольшой задержки (эмулирующей выполнение асинхронной операции):

```
exports.save = function(doc, cb){
  db.push(doc);
  if (cb) {
    setTimeout(function() {
    cb();
    }, 1000);
  }
};
```

Варианты Mocha-тестов могут определяться как асинхронные простым добавлением специального аргумента в функцию, задающую логику тестирования. Этот аргумент обычно называется done. В листинге 10.13 представлены изменения, внесенные в исходный тест .save() для работы с асинхронным кодом.

**Листинг 10.13.** Тестирование асинхронной логики

```
describe('.save(doc)', function(){
  it('should save the document', function(done){
    var pet = { name: 'Tobi' };
    // Сохраняем документ
    memdb.save(pet, function(){
      // Выполняем обратный вызов с первым документом
      var ret = memdb.first({ name: 'Tobi' });
      // Утверждаем, что документ сохранен корректно
      assert(ret == pet);
      // Информируем Mocha о прохождении данного варианта теста
      done();
    });
  });
});
```

Аналогичное правило применяется ко всем хукам. Например, хук beforeEach(), предназначенный для очистки базы данных, мог бы добавить обратный вызов, и тогда среда Mocha ждала бы его завершения, прежде чем двигаться дальше. Если done() вызывается с ошибкой в качестве первого аргумента, Mocha сообщит об ошибке и пометит хук или вариант теста как неудачный:

```
beforeEach(function(done){
  memdb.clear(done);
})
```

Чтобы получить дополнительные сведения о Mocha, обратитесь к онлайновой документации по адресу http://visionmedia.github.com/mocha. Среда Mocha, как и nodeunit, может работать с JavaScript-клиентами.

**непараллельное тестирование в Mocha**

В Mocha тесты выполняются не параллельно, а последовательно, поэтому, хотя сами тесты проще писать, выполнение групп тестов происходит медленнее. Тем не менее Mocha не допустит, чтобы тест длился чрезмерно долго. По умолчанию Mocha-тест должен выполняться не дольше 2000 миллисекунд, после чего он считается проваленным. Если вам требуются более длительные тесты, запустите Mocha вместе с параметром командной строки –timeout, указав большее время тестирования.

В большинстве случаев тесты можно выполнять последовательно. Если в вашем случае этот способ не подходит, воспользуйтесь другими средами тестирования, работающими в параллельном режиме, такими как среда Vows, рассматриваемая в следующем разделе.

### 10.1.4. Среда Vows

Тесты, создаваемые с помощью среды модульного тестирования Vows, получаются более структурированными, чем в других средах тестирования, поэтому Vows-тесты проще читать и поддерживать.

В Vows используется собственная BDD-терминология, в рамках которой определяется структура тестов. В Vows группа тестов содержит один или более *пакетов* (batches). Пакет тестов можно трактовать как группу связанных *контекстов* (contexts), или концептуальных областей, которые вы хотели бы охватить тестами. Покеты и контексты запускаются параллельно. Контекст может содержать несколько вещей: *раздел* (topic), одно или больше *обязательств* (vows), а также один или больше связанных контекстов (вложенные контексты также могут выполняться параллельно). Раздел — это связанный с контекстом код тестирования. Обязательство представляет собой тест результата раздела. Структура Vows-тестов представлена на рис. 10.4.

Среда Vows, подобно nodeunit и Mocha, предназначена для автоматизированного тестирования приложений. Разница в основном касается оформления и параллелизма: для Vows-тестов требуются особые структура и терминология. В этом разделе мы познакомимся с примером теста для приложения и выясним, как использовать Vows для одновременного выполнения нескольких тестов.

Чтобы получить доступ к утилите командной строки vows, позволяющей запускать тесты, обычно нужно глобально установить Vows. Для установки воспользуйтесь следующей командой:

```
$ npm install -g vows
```

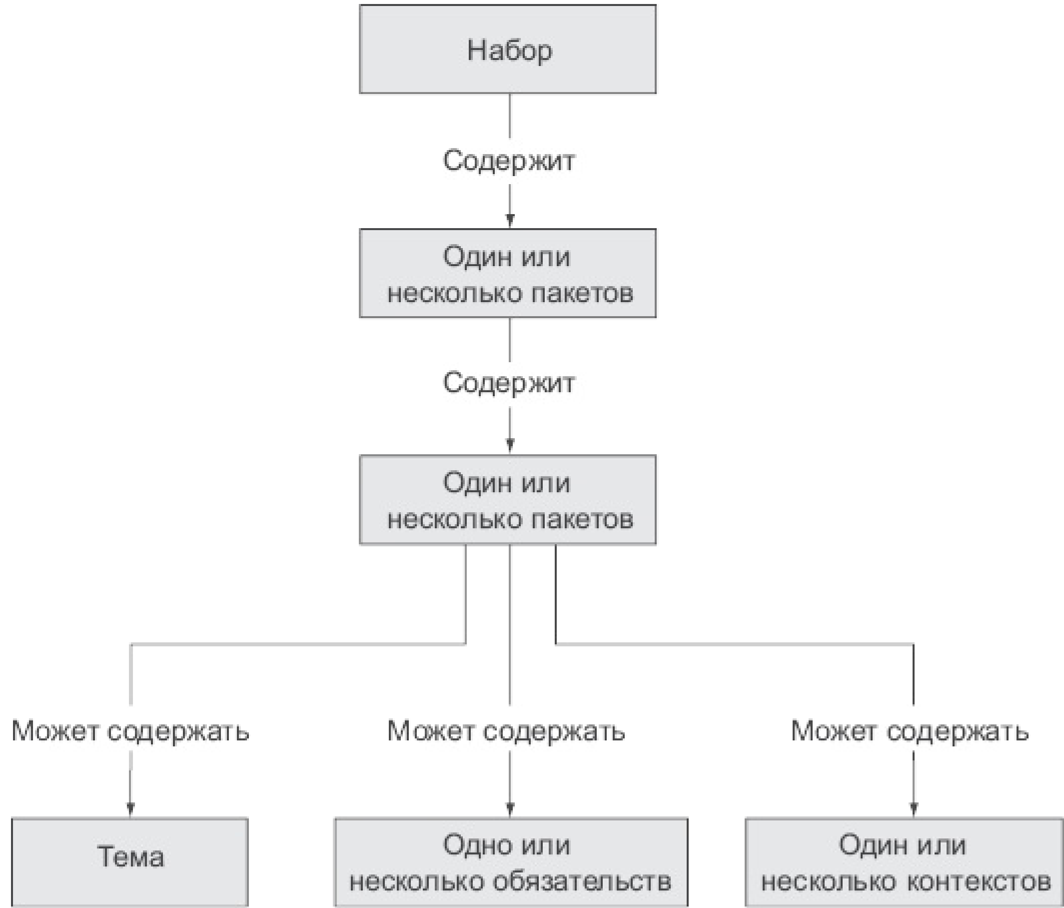



**Рис. 10.4.** Vows позволяет структурировать тесты в группе с помощью пакетов, контекстов, разделов и обязательств

### *Тестирование логики приложения с помощью Vows*

Чтобы начать тестирование в Vows, нужно выполнить либо сценарий, содержащий логику теста, либо утилиту командной строки vows. В следующем примере с помощью автономного тестового сценария (который запускается так же, как и любой другой Node-сценарий) проводится один из тестов основной логики приложения для списка запланированных дел.

В листинге 10.14 представлен пакет тестов. Внутри пакета можно определить контекст. Внутри контекста определяются раздел и обязательство. Обратите внимание, как используется обратный вызов для обработки асинхронной логики в разделе. Если раздел не является асинхронным, вместо передачи значения посредством обратного вызова осуществляется возврат этого значения.

**Листинг 10.14.** Использование Vows, чтобы протестировать приложение для списка запланированных дел

```
var vows = require('vows')
var assert = require('assert')
var Todo = require('./todo');
// Пакет
```

```
vows.describe('Todo').addBatch({
  // Контекст
  'when adding an item': {
    // Раздел
    topic: function () {
      var todo = new Todo();
      todo.add('Feed my cat');
      return todo;
    },
    // Обязательство
    'it should exist in my todos': function(er, todo) {
      assert.equal(todo.getCount(), 1);
    }
  }
}).run();
```

Чтобы включить код из предыдущего листинга в папку с тестами, из которой его можно будет запустить с помощью команды vows, измените последнюю строку кода следующим образом:

```
...
}).export(module);
```

Чтобы выполнить все тесты, находящиеся в папке test, введите следующую команду:

```
$ vows test/*
```

Чтобы получить дополнительные сведения о Vows, обратитесь к онлайновой документации проекта (http://vowsjs.org/). Страница этого веб-сайта показана на рис. 10.5.

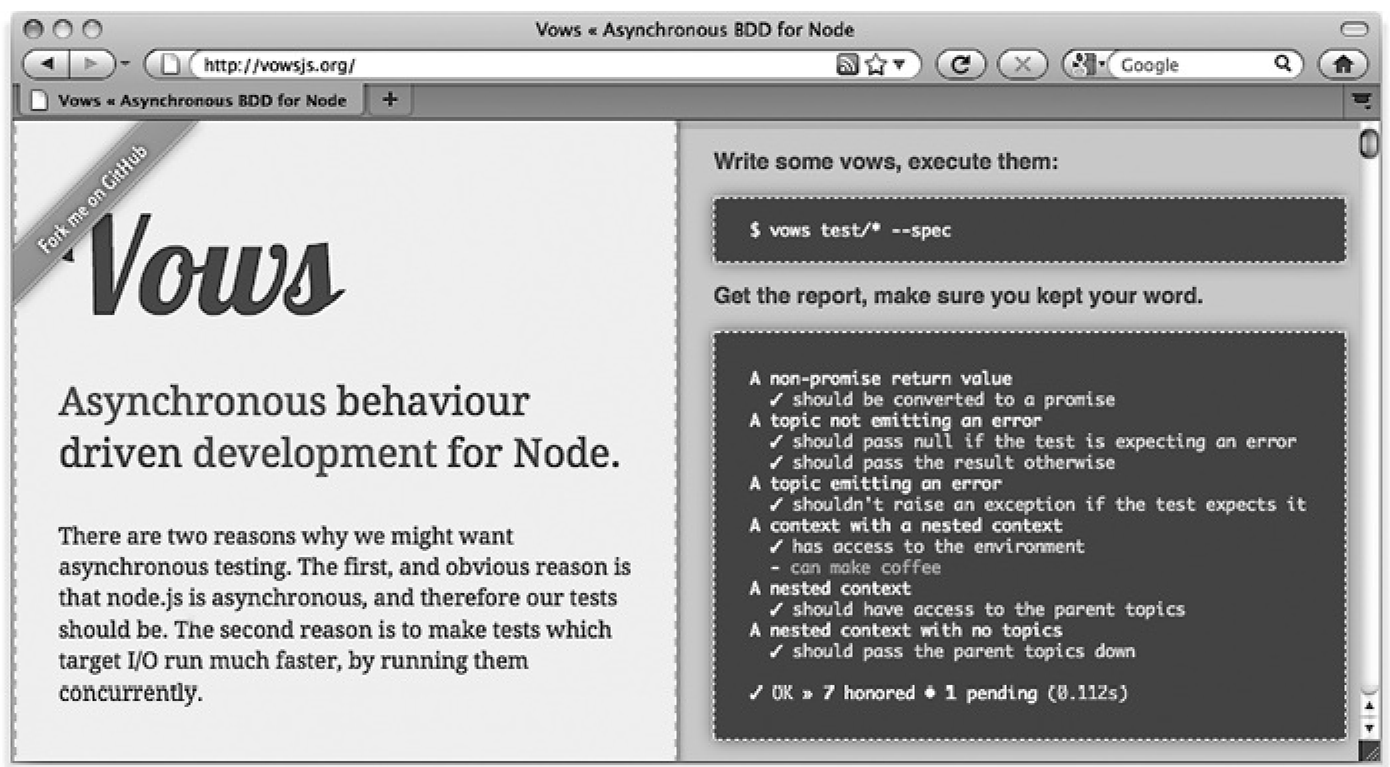



**Рис. 10.5.** Vows объединяет в себе полноценные инструменты BDD-тестирования с дополнительными возможностями, такими как макросы и средства контроля порядка выполнения

Vows предлагает всеобъемлющее решение для тестирования, однако далеко не всем понравится навязываемая этой средой структура тестирования, требующая использования пакетов, контекстов, разделов и обязательств. Либо, возможно, вам больше понравятся возможности другой среды тестирования, либо вы просто лучше знаете другую среду тестирования и не видите необходимости изучать Vows. Если сказанное относится к вам, хорошей альтернативой может стать библиотека should.js. В отличие от других сред тестирования эта библиотека представляет собой BDD-альтернативу Node-модулю assert.

### 10.1.5. Библиотека should.js

Библиотека should.js — это библиотека утверждений, которая может помочь вам сделать свои тесты понятнее, так как дает возможность выражать утверждения в стиле BDD. Поскольку она предназначена для использования вместе с другими средами тестирования, вам не придется отказываться от своей любимый среды. В этом разделе мы узнаем, как с помощью should.js писать утверждения, и напишем тест для собственного модуля.

Библиотеку should.js несложно использовать с другими средами тестирования, поскольку она просто дополняет Object.prototype единственным свойством should. В результате становится возможным создание таких выразительных утверждений, как user.role.should.equal("admin") или users.should.include("rick").

***Тестирование функциональности модуля с помощью should.js***

Давайте не будем мелочиться и напишем в Node калькулятор чаевых, запускаемый из командной строки; он поможет вам понять, кто сколько должен заплатить, когда вы с друзьями решите пропустить по стаканчику. Вам, естественно, придется написать тесты для проверки логики калькуляции, причем они должны быть понятны даже вашим друзьям-гуманитариям, чтобы они не решили, что вы пытаетесь их надуть.

Чтобы создать приложение для калькуляции чаевых, введите следующие команды, которые позволят выбрать папку для приложения и библиотеки тестирования should.js:

```
$ mkdir -p tips/test$ cd tips
$ touch index.js
$ touch test/tips.js
```

Теперь можно установить библиотеку should.js, выполнив следующую команду:

```
$ npm install should
```

Следующим шагом отредактируем содержимое файла index.js, который будет содержать логику, определяющую базовую функциональность приложения. Точнее, логика калькуляции чаевых будет содержать четыре вспомогательные функции:

- addPercentageToEach — увеличение каждого числа в массиве на заданную процентную величину;
- sum — вычисление суммы каждого элемента массива;
- percentFormat — форматирование выводимой процентной величины;
- dollarFormat — форматирование выводимой величины в долларах.

Добавьте описанную логику, включив в файл index.js код из листинга 10.15.

**Листинг 10.15.** Логика калькуляции чаевых

```
// Добавление процентной величины в массив элементов
exports.addPercentageToEach = function(prices, percentage) {
  return prices.map(function(total) {
    total = parseFloat(total);
    return total + (total * percentage);
  });
}
```

```
// Вычисление суммы элементов массива
exports.sum = function(prices) {
  return prices.reduce(function(currentSum, currentValue) {
    return parseFloat(currentSum) + parseFloat(currentValue);
  })
}

// Форматирование отображаемой процентной величины
exports.percentFormat = function(percentage) {
  return parseFloat(percentage) * 100 + '%';
}

// Форматирование отображаемой величины в долларах
exports.dollarFormat = function(number) {
  return '$' + parseFloat(number).toFixed(2);
}
```

Затем изменим тестовый сценарий в файле test/tips.js, используя код из листинга 10.16. Этот сценарий загружает модуль с логикой вычисления чаевых, определяет величину налога, вычисляет чаевые в процентах и пункты счета для теста, а также дополнительно тестирует процент для каждого элемента массива и общую сумму счета.

**Листинг 10.16.** Код, вычисляющий, кому сколько платить

```
// Использование модуля калькуляции чаевых
var tips = require('..');
var should = require('should');

// Определение процентов налоговых отчислений и чаевых
var tax = 0.12;
var tip = 0.15;
// Определение тестируемых пунктов счета
var prices = [10, 20];
var pricesWithTipAndTax = tips.addPercentageToEach(prices, tip + tax);

// Тестирование дополнительной суммы налога и чаевых
```

```
pricesWithTipAndTax[0].should.equal(12.7);
pricesWithTipAndTax[1].should.equal(25.4);

var totalAmount = tips.sum(pricesWithTipAndTax).toFixed(2);
// Тестирование общей суммы счета
totalAmount.should.equal('38.10');

var totalAmountAsCurrency = tips.dollarFormat(totalAmount);
totalAmountAsCurrency.should.equal('$38.10');

var tipAsPercent = tips.percentFormat(tip);
tipAsPercent.should.equal('15%');
```

Запустите сценарий с помощью следующей команды. Если код выполняется так, как ожидалось, сценарий ничего не должен выводить на экран, поскольку мы не делали никаких утверждений, и ваши друзья лишний раз удостоверятся в вашей честности:

```
$ node test/tips.js
```

Библиотека should.js поддерживает многие типы утверждений, от утверждений, использующих регулярные выражения, до утверждений, проверяющих свойства объектов. В результате обеспечивается исчерпывающее тестирование данных и объектов, генерируемых приложением. На странице проекта GitHub (http://github.com/visionmedia/should.js) есть полная документация, описывающая функциональность библиотеки should.js.

Теперь, завершив изучение инструментов модульного тестирования, давайте обратимся к совершенного другому стилю тестирования — приемочному.

## 10.2. Приемочное тестирование

В ходе *приемочного тестирования*, также называемого *функциональным тестированием*, тестируются результаты, а не логика. Теперь, когда мы завершили создание группы модульных тестов для нашего проекта, приемочное тестирование позволит обеспечить дополнительный уровень защиты от ошибок, которые не удалось выявить на этапе модульного тестирования.

Концептуально приемочное тестирование не отличается от тестирования, проводимого конечными пользователями, которые просто поочередно выполняют тесты в соответствии со списком проверки. Однако благодаря автоматизации приемочное тестирование позволяет ускорить проверку приложения и устраняет

влияние человеческого фактора.

Кроме того, приемочное тестирование дает возможность выявить аномалии в поведении JavaScript-клиентов. Если серьезные проблемы возникают из-за JavaScript-клиентов, модульное тестирование сервера тут не поможет, спасет ситуацию только всестороннее приемочное тестирование. Например, если ваше приложение верифицирует данные формы с помощью JavaScript-клиента, приемочное тестирование позволит гарантировать, что логика верификации работает и требуемым образом принимает или отклоняет водимые данные. Или, к примеру, вы можете столкнуться с функциональностью Ajax-администрирования, такой как навигация с целью выбора контента для начальной страницы веб-сайта, — эту операцию разрешается выполнять только аутентифицированным пользователям. Для решения задачи можно написать тест, гарантирующий, что Ajax-запрос возвратит требуемые результаты, только когда пользователь войдет в систему, в то время как другой тест мог бы гарантировать, что тот, кто не прошел аутентификацию, не получит доступа к данным.

В этом разделе мы узнаем, как использовать две среды приемочного тестирования, Tobi и Soda. Soda предлагает для приемочного тестирования богатый инструментарий реального браузера, в то время как среда Tobi, с которой мы будем знакомиться в первую очередь, проще в установке, изучении и применении.

### 10.2.1. Среда Tobi

Tobi (https://github.com/LearnBoost/tobi) — это простая в применении среда приемочного тестирования, которая эмулирует браузер и использует библиотеку should.js, предлагая доступ к ее механизму утверждений. В Tobi применяются два модуля от независимых производителей, jsdom и htmlparser, имитирующие веб-браузер и открывающие доступ к виртуальной модели DOM.

Tobi позволяет без проблем писать тесты, которые при необходимости смогут выполнять вход в созданное вами веб-приложение и отправлять вам веб-запросы, эмулирующие пользователей приложения. Если Tobi вернет непредвиденные результаты, ваш тест может предупредить вас о возникшей проблеме.

Поскольку среда Tobi должна имитировать действия пользователя и проверять результаты веб-запросов, ей приходится часто обрабатывать или исследовать DOM-элементы. В мире разработки клиентских JavaScript-приложений разработчики часто обращаются к библиотеке jQuery (http://jquery.com), если нужно взаимодействовать с моделью DOM. Разработчики могут также использовать jQuery на стороне сервера, а поскольку Tobi также задействует jQuery, минимизируется объем информации, которую приходится изучать для создания Tobi-тестов.

В этом разделе мы поговорим о том, как с помощью Tobi протестировать через

сеть любое запущенное веб-приложение, причем не обязательно Node-приложение. Кроме того, мы покажем, как использовать Tobi для тестирования веб-приложения, созданного с помощью Express, даже если это приложение не запущено.

### *Тестирование веб-приложений с помощью Tobi*

Прежде чем приступать к разработке тестов с помощью Tobi, создайте для них папку (либо воспользуйтесь имеющейся папкой приложения), затем измените папку в командной строке и введите команду для установки Tobi:

```
$ npm install tobi
```

Код, приведенный в листинге 10.17, показывает, как использовать Tobi для тестирования функциональности веб-приложения на веб-сайте, — в нашем случае запущено приложение для списка запланированных дел, которое мы тестировали в главе 5. Тест предпринимает попытку создания элемента списка, а затем ищет его на странице ответа. Если при запуске этого теста с помощью Node никаких исключений не генерируется, значит, тест пройден успешно.

**Листинг 10.17.** Тестирование веб-приложения с помощью HTTP-запросов

```
var tobi = require('tobi');
// Создание браузера
var browser = tobi.createBrowser(3000, '127.0.0.1');

// Получение формы для списка запланированных дел
browser.get('/', function(res, $){
  $('form')
  // Заполнение полей формы
  .fill({ description: 'Floss the cat' })
  // Отправка данных формы
  .submit(function(res, $) {
    $('td:nth-child(3)').text().should.equal('Floss the cat');
  });
});
```

Этот сценарий создает фиктивный браузер, выполняет с его помощью HTTP-запрос GET для главной страницы с формой ввода, заполняет поля формы и отправляет данные формы. Затем сценарий проверяет, находится ли в ячейке таблицы текст «Floss the Cat». Если текст обнаруживается, считается, что тест пройден.

Приложение можно протестировать, даже не запуская его. Для этого

воспользуйтесь следующим Tobi-тестом:

```
var tobi = require('tobi');
var app = require('./app');
var browser = tobi.createBrowser(app);

browser.get('/about', function(res, $){
  res.should.have.status(200);
  $('div').should.have.one('h1', 'About');
  app.close();
});
```

В состав Tobi не входит модуль запуска тестов, но его можно использовать со средами модульного тестирования, такими как Mocha или nodeunit.

### 10.2.2. Среда Soda

Soda (https://github.com/LearnBoost/soda) предлагает альтернативный подход к приемочному тестированию. В то время как другие Node-среды приемочного тестирования имитируют браузеры, Soda удаленно управляет реальными браузерами. Как показано на рис. 10.6, Soda делает это путем отправки инструкций на сервер Selenium Server (также известный как Selenium RC) или в службу тестирования по запросу Sauce Labs.

При тестировании сервер Selenium RC будет открывать браузеры на той машине, на которой он установлен, а служба Sauce Cloud будет открывать виртуальные браузеры на некоем сервере где-то в Интернете.

Взаимодействие с браузерами реализуют Selenium Server и Sauce Labs, но не Soda, тем не менее любую запрошенную информацию они возвращают обратно в среде Soda. Если вам нужно параллельно выполнить несколько тестов, но при этом вы не хотите загружать работой собственный компьютер, обратитесь к службе Sauce Labs.

**Рис. 10.6.** Среда приемочного тестирования Soda обеспечивает удаленное управление реальными браузерами. С помощью сервера Selenium RC или службы Sauce Labs среда Soda предоставляет API-интерфейс, который позволяет Node выполнять непосредственное тестирование, учитывающее реалии разных реализаций браузеров

В этом разделе мы поговорим о том, как установить Soda и Selenium Server, как тестировать приложения с помощью Soda и Selenium и как тестировать приложения с помощью Soda и Sauce Labs.

### *Установка Soda и Selenium Server*

Чтобы приступить к тестированию с помощью Soda, нужно установить среду Soda, а также сервер Selenium RC (если вы решите обойтись без Sauce Labs). Чтоб установить Soda, введите следующую команду:

```
$ npm install soda
```

Для работы сервера Selenium RC требуется Java. Если среда Java не установлена обратитесь на официальную страницу загрузки Java за инструкциями по установке для вашей операционной системы (www.java.com/ru/download/).

Установка сервера Selenium RC достаточно проста. Загрузите последнюю версию файла .jar со страницы загрузок Selenium (http://seleniumhq.org/download/). После завершения загрузки файла запустите сервер с помощью следующей команды (в названии вашего файла может использоваться другой номер версии):

```
java -jar selenium-server-standalone-2.6.0.jar
```

### *Тестирование веб-приложений с помощью Soda и Selenium*

После запуска сервера можно включить в сценарий следующий код для настройки выполняющихся тестов. В вызове createClient параметры host и port задают адрес хоста и номер порта, используемые для подключения к серверу Selenium RC. По умолчанию эти параметры имеют значения 127.0.0.1 и 4444 соответственно.

Параметр url в вызове createClient задает базовый URL-адрес, который должен открываться в окне браузера при тестировании, а значение browser задает браузер, применяемый для тестирования:

```
var soda = require('soda')
var assert = require('assert');
var browser = soda.createClient({
  host: '127.0.0.1',
  port: 4444,
  url: 'http://www.reddit.com',
  browser: 'firefox'
});
```

Чтобы получать информацию о том, что делает сценарий тестирования, можно воспользоваться следующим кодом. Этот код выводит на печать каждую Selenium-команду:

```
browser.on('command', function(cmd, args){
  console.log(cmd, args.join(', '));
});
```

Далее нужно включить в сценарий тестирования сами тесты. В листинге 10.18 представлен пример теста, в котором делается попытка реализовать вход пользователя на веб-сайт Reddit; тест считается проваленным, если на результирующей странице не появится текст «logout». Описание команд, таких как clickAndWait, можно найти на веб-сайте Selenium (http://release.seleniumhq.org/selenium-core/1.0.1/reference.html).

**Листинг 10.18.** В Soda-тесте допускаются команды управления действиями браузера

```
browser
// Выстраиваем методы в цепочку
.chain
// Начинаем Selenium-сеанс
.session()
// Открываем URL-адрес
.open('/')
// Вводим текст в поле формы
.type('user', 'mcantelon')
.type('passwd', 'mahsecret')
// Щелкаем на кнопке и ждем
```

```
.clickAndWait('//button[@type="submit"]')
// Удостоверяемся в существовании текста
.assertTextPresent('logout')
// Помечаем тест как пройденный
.testComplete()
// Завершаем Selenium-сеанс
.end(function(err){
  if (err) throw err;
  console.log('Done!');
});
```

### *Тестирование веб-приложений с помощью Soda и Sauce Labs*

Если вы решите тестировать веб-приложения с помощью службы Sauce Labs, подпишитесь на эту службу на веб-сайте Sauce Labs (https://saucelabs.com) и измените сценарий тестирования примерно так, как это сделано в листинге 10.19.

**Листинг 10.19.** Управление браузером Sauce Labs с помощью Soda

```
var browser = soda.createSauceClient({
  'url': 'http://www.reddit.com/',
  // Имя пользователя Sauce Labs
  'username': 'yourusername',
  // Ключ доступа к API-интерфейсу Sauce Labs
  'access-key': 'youraccesskey',
  // Предпочитаемая операционная система
  'os': 'Windows 2003',
  // Предпочитаемый тип браузера
  'browser': 'firefox',
  // Предпочитаемая версия браузера
  'browser-version': '3.6',
  'name': 'This is an example test',
  // Максимальная породолжительность теста (тест будет
  // считаться проваленным, если продлится дольше)
  'max-duration': 300
});
```

И на этом всё. Мы получили базовые сведения о мощной методике

тестирования, которая дополняет ваши модульные тесты и делает ваши приложения более устойчивыми к случайным ошибкам.

### 10.3. Резюме

Включая автоматизированное тестирование в процесс разработки приложений, вы существенно снижаете шансы на появление ошибок в коде и можете работать гораздо увереннее.

Если вы новичок в модульном тестировании, воспользуйтесь Mocha или nodeunit — это великолепная стартовая позиция, поскольку указанные среды тестирования просты в изучении, гибки и позволяют работать с библиотекой should.js, предоставляющей возможность задействовать механизм утверждений в стиле BDD. Если вам по душе стиль BDD и вы ищете систему, которая позволяет структурировать тесты и управлять порядком их выполнения, среда Vows будет прекрасным выбором.

Если вас интересует приемочное тестирование, начните с Tobi. Эта среда тестирования проста в установке и применении, а если вы к тому же знакомы с библиотекой jQuery, то сможете легко установить и запустить Tobi. Если же вам требуется приемочное тестирование, учитывающее особенности разных браузеров, выбирайте Soda, однако учтите, что Soda-тесты длятся дольше, к тому же вам придется освоить API-интерфейс Selenium.

Теперь, когда вы знаете, как в Node выполняется автоматизированное тестирование, давайте поглубже окунемся в тему шаблонизации веб-приложений в Node, познакомившись с несколькими шаблонизаторами, помогающими сделать веб-разработку приятнее и продуктивнее.

## Глава 11. Шаблонизация веб-приложений

Организация кода приложений путем шаблонизации

Создание шаблонов с помощью встроенного JavaScript-кода

Минималистский подход к шаблонизации в Hogan

Создание шаблонов с помощью шаблонизатора Jade

В главах 8 и 9 рассматривались некоторые элементы шаблонизации, применяемые при создании представлений в среде Express. В данной главе, которая полностью посвящена шаблонизации, мы рассмотрим три популярных

шаблонизатора, а также узнаем, как шаблонизация помогает сделать код веб-приложений чище за счет отделения программной логики от визуализирующей разметки.

Если вы знакомы с шаблонизацией и паттерном MVC (Model-View-Controller – модель-представление-контроллер), можете сразу переходить к разделу 11.2, с которого начинается подробное рассмотрение шаблонизаторов, включая EJS, Hogan и Jade. Если же шаблонизация вам в новинку, продолжайте читать — концептуально шаблонизация описывается в следующих нескольких разделах.

## 11.1. Поддержка чистоты кода путем шаблонизации

С помощью паттерна MVC можно разрабатывать как Node-приложения, так и веб-приложения почти всех остальных типов. Одна из ключевых концепций MVC заключается в разделении логики, данных и представления. В MVC-приложениях пользователь обычно запрашивает нужный ресурс на сервере, затем *контроллер* (controller) запрашивает данные приложения у *модели* (model) и передает эти данные *представлению* (view), которое показывает их конечному пользователю. MVC-представление часто реализуется с помощью одного из языков шаблонизации. Если в приложении используется шаблонизация, представление ретранслирует *шаблонизатору* (template engine) выбранные значения, возвращаемые моделью, и указывает, какой файл шаблона должен определять, как показывать эти значения.

На рис. 11.1 показано, как логика шаблонизации вписывается в общую архитектуру MVC-приложения.

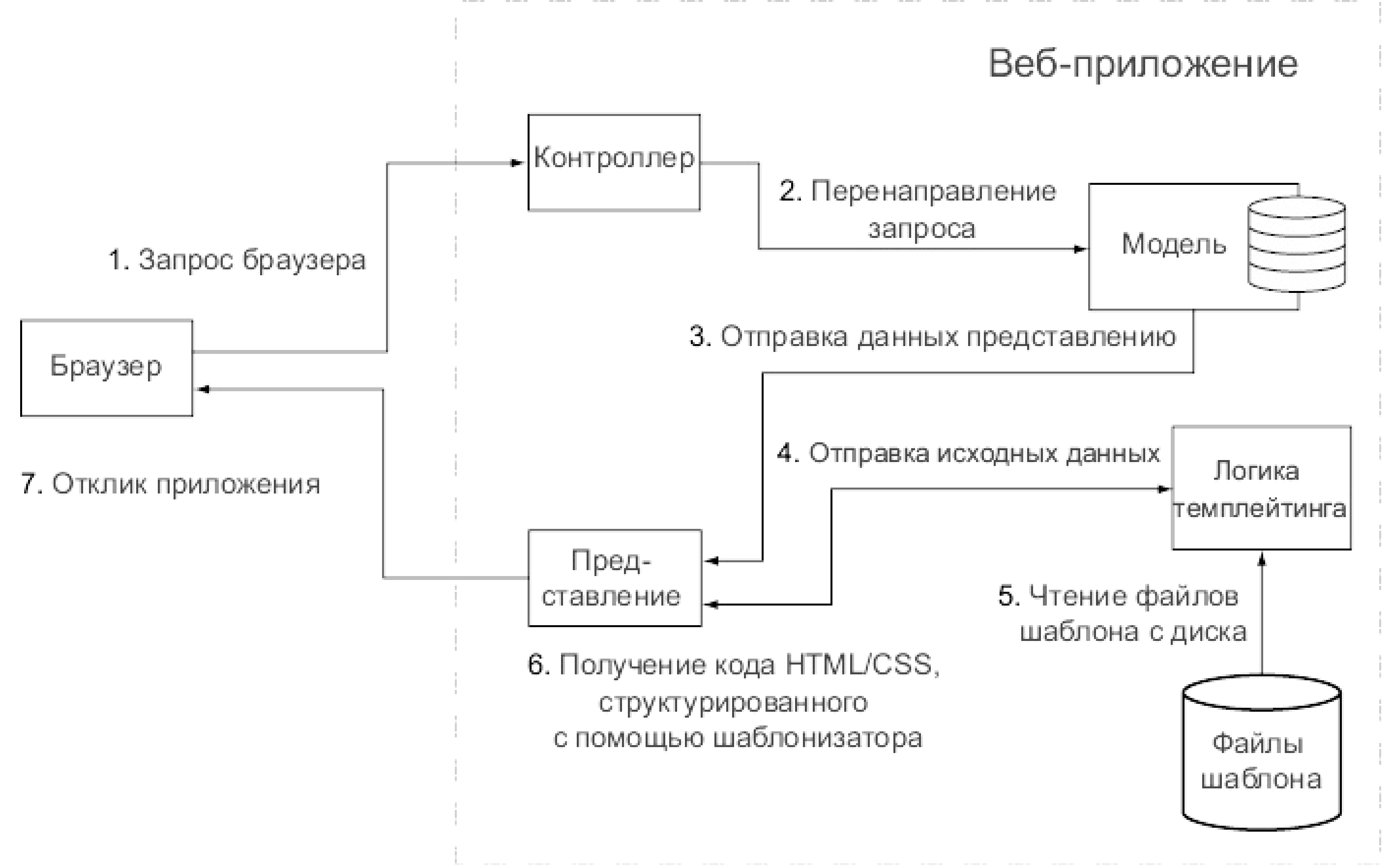



**Рис. 11.1.** Последовательность операций при работе MVC-приложения и его взаимодействие с уровнем шаблона

Файлы шаблонов обычно содержат местозаполнители для значений приложений, а также фрагменты HTML-, CSS- и иногда клиентского JavaScript-кода предназначенного для вывода на экран виджетов от независимых производителей (таких, как кнопка Like в Facebook) или для включения специального режима работы интерфейса (например, скрытия или показа частей страницы). А так как файлы шаблонов больше связаны с уровнем представления, чем программной логики, разработчикам клиентских и серверных приложений эти файлы помогают организовать разделение труда.

В этом разделе мы поговорим о HTML-визуализации, выполняемой как с помощью, так и без помощи шаблона, чтобы вы смогли почувствовать разницу. Однако сначала давайте разберем пример шаблонизации.

### 11.1.1.Шаблонизация в действии

Чтобы быстро понять, как действует шаблонизация, мы рассмотрим проблему элегантного HTML-вывода из простого приложения для блога. Каждая запись блога включает заголовок, дату создания и текст. В окне веб-браузера блог будет выглядеть так, как показано на рис. 11.2.

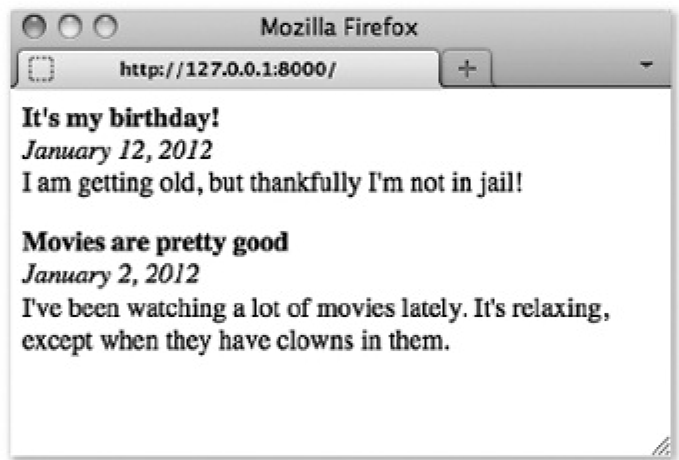



**Рис. 11.2.** Пример информации, выводимой в браузере приложением для блога

Записи блога могут считываться из текстового файла, отформатированного так, как фрагмент из файла entries.txt, показанный в листинге 11.1. Символы —- показывают, где заканчивается одна запись и начинается другая.

**Листинг 11.1.** Текстовый файл с записями блога

```
title: It's my birthday!
date: January 12, 2012
I am getting old, but thankfully I'm not in jail!
—-
title: Movies are pretty good
date: January 2, 2012
I've been watching a lot of movies lately. It's relaxing,
except when they have clowns in them.
```

Код приложения для блога, находящийся в файле blog.js, начинается с объявления модулей, загружаемых по требованию, и считывания записей блога, как показано в листинге 11.2.

**Листинг 11.2.** Логика синтаксического разбора записей блога, хранящихся в файле, в простом приложении для блога

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

// Функция для чтения и синтаксического разбора текста записи блога
function getEntries() {
  var entries = [];
  // Чтение данных записи блога из файла
  var entriesRaw = fs.readFileSync('./
```

```
    entries.txt', 'utf8');

  // Синтаксический разбор текста в отдельные записи блога
  entriesRaw = entriesRaw.split("—-");

  entriesRaw.map(function(entryRaw) {
    var entry = {};
    // Синтаксический разбор текста записей в отдельные строки
    var lines = entryRaw.split("\n");

    // Синтаксический разбор строк в свойства записей
    lines.map(function(line) {
      if (line.indexOf('title: ') === 0) {
        entry.title = line.replace('title: ', '');
      }
      else if (line.indexOf('date: ') === 0) {
        entry.date = line.replace('date: ', '');
      }
      else {
        entry.body = entry.body || '';
        entry.body += line;
      }
    });
    entries.push(entry);
  });
  return entries;
}
var entries = getEntries();
console.log(entries);
```

Следующий код, будучи добавленным в приложение для блога, задаст HTTP-сервер. Когда сервер получает HTTP-запрос, он возвращает страницу, содержащую все записи блога. Эта страница визуализируется с помощью функции blogPage, которую мы определим чуть позже:

```
var server = http.createServer(function(req, res) {
  var output = blogPage(entries);
```

```
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(output);
});

server.listen(8000);
```

Теперь нужно определить функцию blogPage, предназначенную для визуализации записей блога на HTML-странице, отправляемой браузеру пользователя. Для этого мы используем два подхода:

- визуализация HTML-кода без шаблона;
- визуализация HTML-кода с помощью шаблона.

Сначала рассмотрим визуализацию HTML-кода без шаблона.

### *Визуализация HTML-кода без шаблона*

Приложение для блога могло бы непосредственно выводить HTML-код на экран, но объединение на экране HTML-разметки и программной логики приложения привело бы к хаосу. В листинге 11.3 функция blogPage показывает, как вывести на экран записи блога без помощи шаблона.

**Листинг 11.3.** Шаблонизатор разделяет детали представления и прикладную логику

```
function blogPage(entries) {
  // HTML-разметка слишком сильно "перемешана" с прикладной логикой
  var output = '<html>'
        + '<head>'
        + '<style type="text/css">'
        + '.entry_title { font-weight: bold; }'
        + '.entry_date { font-style: italic; }'
        + '.entry_body { margin-bottom: 1em; }'
        + '</style>'
        + '</head>'

        + '<body>';
  entries.map(function(entry) {
```

```
    output += '<div class="entry_title">' + entry.title + "</div>\n"
        + '<div class="entry_date">' + entry.date + "</div>\n"
        + '<div class="entry_body">' + entry.body + "</div>\n";
  });

  output += '</body></html>';

  return output;
}
```

Обратите внимание, что контент, CSS-определения и HTML-разметка связанные с представлением, приводят к значительному увеличению числа строк в коде приложения.

### *Визуализация HTML-кода с помощью шаблона*

Визуализация HTML-кода путем шаблонизации позволяет убрать из прикладной логики HTML-разметку, что делает код приложения существенно чище.

Чтобы проверять рассматриваемые в этом разделе примеры, нужно установить в папку приложения модуль ejs, позволяющий работать с внедренным JavaScript-кодом (Embedded JavaScript, EJS). Для этого в командой строке введите следующую команду:

```
npm install ejs
```

Показанный далее код загружает шаблон из файла, а затем определяет новую версию функции blogPage, которая теперь использует шаблонизатор EJS (о нем мы поговорим в разделе 11.2):

```
var ejs = require('ejs');
var template = fs.readFileSync('./template/blog_page.ejs', 'utf8');

function blogPage(entries) {
  var values = {entries: entries};
  return ejs.render(template, {locals: values});
}
```

Файл EJS-шаблона содержит HTML-разметку, которая исключается и прикладной логики, а также местозаполнители, позволяющие указать, где должны располагаться данные, передаваемые шаблонизатору. Файл EJS-шаблона, применяемый для вывода записей блога на экран, включает HTML-код и

местозаполнители (листинг 11.4).

**Листинг 11.4.** EJS-шаблон для вывода на экран записей блога

```
<html>
  <head>
    <style type="text/css">
      .entry_title { font-weight: bold; }
      .entry_date { font-style: italic; }
      .entry_body { margin-bottom: 1em; }
    </style>
  </head>

  <body>
    // Местозаполнитель, который циклически перебирает записи блога
    <% entries.map(function(entry) { %>
      // Местозаполнители для ввода фрагментов данных в каждой записи
      <div class="entry_title"><%= entry.title %></div>
      <div class="entry_date"><%= entry.date %></div>
      <div class="entry_body"><%= entry.body %></div>
    <% }); %>
  </body>
</html>
```

Модули, разработанные Node-сообществом, также предлагают шаблонизаторы, причем много и разных. Если вы полагаете, что HTML- и/или CSS-код недостаточно элегантен, так как HTML-код требует закрывающих тегов, а CSS-код – открывающих и закрывающих скобок, обратите внимание на шаблонизаторы. Они позволяют файлам шаблонов использовать специальные «языки» (такие, как язык Jade, о котором мы поговорим в этой главе позже), предлагающие быстрый способ спецификации HTML- и/или CSS-кода.

Хотя благодаря шаблонизаторам можно сделать шаблоны более «чистыми», вы просто можете не захотеть тратить свое время на изучение альтернативных способов спецификации HTML- и CSS-кода. В конце концов, принятие окончательного решения зависит от ваших личных предпочтений.

В оставшейся части главы мы поговорим о том, как встраивать шаблонизацию в ваши Node-приложения. Для этого используются три популярных шаблонизатора:

- шаблонизатор на базе внедренного JavaScript-кода;

- минималистский шаблонизатор Hogan;
- шаблонизатор Jade.

Каждый из этих шаблонизаторов предлагает альтернативный способ написания HTML-кода. Давайте начнем с шаблонизатора на базе внедренного JavaScript-кода.

## 11.2. Шаблонизация с использованием внедренного JavaScript-кода

Внедренный JavaScript-код (Embedded JavaScript, EJS) предлагает совершенн прямолинейный подход к шаблонизации (https://github.com/visionmedia/ejs), который покажется очень близким тому, кто использует шаблонизаторы в других языках, таких как JSP ( Java), Smarty (PHP), ERB (Ruby) и подобных. В это шаблонизаторе EJS-теги внедряются в HTML-код в качестве местозаполнителе для данных. Кроме того, EJS позволяет выполнять в шаблонах исходную JavaScrip логику для таких операций, как условное ветвление и итерация, как это делается в PHP.

В этом разделе мы узнаем:

- как создавать EJS-шаблоны;
- как использовать EJS-фильтры для поддержки обычной функциональности представлений, такой как манипулирование текстом, сортировка и итерация;
- как интегрировать EJS с Node-приложениями;
- как использовать EJS для клиентских приложений.

А теперь давайте глубже погрузимся в мир EJS-шаблонизации.

### 11.2.1. Создание шаблона

В мире шаблонизации данные, отсылаемые шаблонизатору для визуализации, иногда называют *контекстом* (context). Вот пример использования EJS в Node для визуализации простого шаблона с помощью контекста:

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {message: 'Hello template!'};
```

```
console.log(ejs.render(template, {locals: context}));
```

Обратите внимание на применение параметра locals во втором аргументе функции render. Второй аргумент может включать параметры визуализации, а также контекстные данные. Параметр locals гарантирует, что отдельные биты контекстных данных не будут интерпретироваться как EJS-параметры. Однако в большинстве случаев в качестве второго параметра можно передавать сам контекст, как в следующем вызове render:

```
console.log(ejs.render(template, context));
```

Если вы передаете контекст EJS-шаблону непосредственно во втором аргументе вызова render, убедитесь, что при именовании контекстных значений не используются следующие термины: cache, client, close, compileDebug, debug, filename, open или scope. Эти значения зарезервированы с целью изменения параметров шаблонизатора.

### *Экранирование символов*

В процессе визуализации EJS экранирует все специальные символы в контекстных значениях, заменяя их кодами HTML-сущностей. Это позволяет предотвратить атаки межсайтового скриптинга (Cross-Site Scripting, XSS), в ходе которых злонамеренные пользователи веб-приложения пытаются в качестве данных отправить зловредный JavaScript-код в надежде, что этот код будет выполнен при выводе данных на экран в браузере другого пользователя. Следующий фрагмент демонстрирует, как работает механизм экранирования символов в EJS-шаблоне:

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {message: "<script>alert('XSS attack!');</script>"};

console.log(ejs.render(template, context));
```

В результате выполнения этого кода на экране появится следующее:

```
&lt;script&gt;alert('XSS attack!');&lt;/script&gt;
```

Если вы доверяете используемым в шаблоне данным и не хотите экранировать контекстные значения в EJS-шаблоне, в теге шаблона используйте символы <%- вместо символов <%=, как показано в примере:

```
var ejs = require('ejs');
var template = '<%- message %>';
var context = {
  message: "<script>alert('Trusted JavaScript!');</script>"
```

```
};

console.log(ejs.render(template, context));
```

Если вам не нравятся символы, используемые в EJS-шаблоне для спецификации тегов, вы можете легко их изменить:

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

ejs.open = '{{:'
ejs.close = '}}:'
var template = '{{= message }}';
var context = {message: 'Hello template!'};

console.log(ejs.render(template, context));
```

Теперь, когда кое-какие базовые знания о EJS у вас есть, давайте посмотрим, как EJS позволяет упростить управление представлением данных.

### 11.2.2. Манипулирование данными шаблона с помощью EJS-фильтров

В EJS поддерживаются *фильтры* (filters) — специальные механизмы, облегчающие преобразование легковесных данных. Признак использования фильтра — двоеточие (:) после открывающих символов EJS-тега, например:

- символы <%=: служат для экранирования выводимых данных с помощью фильтра;
- символы <%-: позволяют отменить экранирование выводимых данных с помощью фильтра.

Фильтры могут *выстраиваться в цепочки*. Это означает, что можно поместить несколько фильтров в единственный EJS-тег, получив эффект совместного применения всех фильтров (это напоминает концепцию «канала» в UNIX). В следующих нескольких разделах мы рассмотрим фильтры, которые могут быть полезны в самых обычных сценариях.

#### *Фильтры для обработки выделения*

EJS-фильтры помещаются в EJS-теги. Чтобы получить представление о полезности подобных фильтров, представьте себе приложение, дающее пользователю

возможность проинформировать других о фильмах, которые он посмотрел. Это могла бы быть, например, информация о самом последнем просмотренном фильме.

В следующем примере EJS-тег в шаблоне выводит на экран данные последнего фильма из массива фильмов с помощью фильтра last, который предназначен для вывода на экран только последнего элемента массива:

```
var ejs = require('ejs');
var template = '<%=: movies | last %>';
var context = {'movies': [
  'Bambi',
  'Babe: Pig in the City',
  'Enter the Void'
]};

console.log(ejs.render(template, context));
```

Обратите внимание, что допустимым также является фильтр first, который позволяет выбрать первый элемент массива. Если вы хотите выбрать конкретный элемент в массиве, воспользуйтесь фильтром get. EJS-тег <%=: movies | get:1 %> выведет на экран второй элемент массива movies (0 соответствует первому элементу массива). Фильтр get можно также использовать для вывода свойств, если в качестве контекстного значения используется объект, а не массив.

### *Фильтры для изменения регистра символов*

EJS-фильтры могут также применяться для изменения регистра символов. EJS-тег в следующем шаблоне имеет фильтр, который для заданного контекстного значения делает первую букву прописной. В рассматриваемом примере вместо «bob» на экран выводится «Bob»:

```
var ejs = require('ejs');
var template = '<%=: name | capitalize %>';
var context = {name: 'bob'};

console.log(ejs.render(template, context));
```

Если нужно вывести на экран контекстное значение исключительно с применением символов верхнего регистра, воспользуйтесь фильтром upcase. Если же, наоборот, требуются только символы нижнего регистра, обратитесь к фильтру downcase.

### *Фильтры для обработки текста*

С помощью EJS-фильтров можно выполнять обработку текста, например усекать текст, добавлять символы в начале или в конце текстового фрагмента и даже удалять часть текста.

Путем усечения текста до определенного количества символов можно избежать проблем, возникающих при использовании длинных строк текста в HTML-макетах. Например, следующий код усекает текст заголовка до 20 символов, в результате на экран выводится надпись «The Hills are Alive».

```
var ejs = require('ejs');
var template = '<%=: title | truncate:20 %>';
var context = {title: 'The Hills are Alive With the Sound of Critters'};

console.log(ejs.render(template, context));
```

Существует также EJS-фильтр, позволяющий усекать текст до заданного количества слов. Чтобы усечь контекстное значение до двух слов, в предыдущем примере кода замените EJS-тег следующим:

```
<%=: title | truncate_words:2 %>
```

В результате на экране останется только надпись «The Hills».

В фильтре replace неявно используется прототип String.prototype.replace(pattern), поэтому этот фильтр принимает либо строку, либо регулярное выражение. В следующем коде представлен пример автоматического сокращения слова с помощью EJS-фильтра:

```
var ejs = require('ejs');
var template = "<%=: weight | replace:'kilogram','kg' %>";
var context = {weight: '40 kilogram'};

console.log(ejs.render(template, context));
```

Чтобы добавить текст в конце, воспользуйтесь фильтром append:'some text'. Если же нужно добавить текст в начале, обратитесь к фильтру prepend:'some text'.

### *Фильтры для сортировки*

EJS-фильтры могут применяться для сортировки. Возвращаясь к предыдущему примеру с цитированием названия фильма, воспользуемся EJS-фильтрами для сортировки фильмов по названию и выведем на экран первый элемент массива в алфавитном порядке (рис. 11.3).

Показанную на рис. 11.3 блок-схему реализует следующий код:

```
var ejs = require('ejs');
var template = '<%=: movies | sort | first %>';
var context = {'movies': [
  'Bambi',
  'Babe: Pig in the City',
  'Enter the Void'
]};

console.log(ejs.render(template, context));
```

| movies | sort | first |
|---|---|---|
| Bambi | Babe: Pig in the City | Babe: Pig in the City |
| Babe: Pig in the City | Bambi | |
| Enter the Void | Enter the Void | |

**Рис. 11.3.** Блок-схема, иллюстрирующая использование EJS-фильтров для обработки текстового массива

Если вы хотите отсортировать массив объектов, но предпочли бы при этом сравнивать свойства объектов, можете сделать это с помощью фильтров:

```
var ejs = require('ejs');
var template = "<%=: movies | sort_by:'name' | first | get:'name' %>";
var context = {'movies': [
  {name: 'Babe: Pig in the City'},
  {name: 'Bambi'},
  {name: 'Enter the Void'}
]};

console.log(ejs.render(template, context));
```

Обратите внимание на фильтр get:'name' в конце цепочки фильтров. Применение этого фильтра обусловлено тем, что в результате сортировки возвращается объект, у которого нужно выбрать выводимое на экран свойство.

### *Фильтр для проецирования*

С помощью EJS-фильтра map можно выбрать то свойство объекта, которое должно обрабатываться следующими фильтрами. В предыдущем примере кода в цепочку фильтров можно было бы включить фильтр map. Вместо выбора свойства с помощью фильтра sort_by и выделения выводимого на экран свойства с помощью фильтра get можно воспользоваться фильтром map для создания массива свойств объекта. Результирующий EJS-шаблон мог бы выглядеть так:

```
<%=: movies | map:'name' | sort | first %>
```

### *Создание собственных фильтров*

Хотя EJS поставляется с большим количеством готовых фильтров, пригодных для решения большинства задач, всегда может возникнуть необходимость в создании дополнительного фильтра. Например, в EJS отсутствует готовый фильтр, выполняющий округление числа до случайного десятичного разряда. Подобный фильтр можно создать самостоятельно, как показано в листинге 11.5.

**Листинг 11.5.** Создание собственных нестандартных EJS-фильтров

```
var ejs = require('ejs');
var template = '<%=: price * 1.145 | round:2 %>';
var context = {price: 21};

// Определение функции для объекта ejs.filters
ejs.filters.round = function(number, decimalPlaces) {
  // Первый аргумент – это вводимое значение, контекст
  // или результат применения предыдущего фильтра
  number = isNaN(number) ? 0 : number;
  decimalPlaces = !decimalPlaces ? 0 : decimalPlaces;

  var multiple = Math.pow(10, decimalPlaces);
  return Math.round(number * multiple) / multiple;
};

console.log(ejs.render(template, context));
```

Как видите, EJS-фильтры — это прекрасный инструмент для сокращения объема кода, обеспечивающего подготовку данных к выводу на экран. Чтобы перед визуализацией шаблона не заниматься преобразованием данных вручную, можно

воспользоваться встроенными EJS-механизмами, облегчающими и ускоряющими такое преобразование.

### 11.2.3. Интеграция EJS-шаблонов в приложение

Поскольку хранить шаблоны в файлах наравне с кодом приложения неудобно и приводит к загромождению кода, воспользуемся API-интерфейсом файловой системы Node, позволяющим загружать шаблоны из отдельных файлов.

Перейдите в рабочую папку и создайте файл app.js, содержащий код из листинга 11.6.

**Листинг 11.6.** Хранение кода шаблонов в отдельных файлах

```
var ejs = require('ejs');
var fs = require('fs');
var http = require('http');
// Местоположение файла шаблона
var filename = './template/students.ejs';

// Данные, передаваемые шаблонизатору
var students = [
  {name: 'Rick LaRue', age: 23},
  {name: 'Sarah Cathands', age: 25},
  {name: 'Bob Dobbs', age: 37}
];

// Создание HTTP-сервера
var server = http.createServer(function(req, res) {
  if (req.url == '/') {
    // Считывание шаблона из файла
    fs.readFile(filename, function(err, data) {
      var template = data.toString();
      var context = {students: students};
      // Визуализация шаблона
      var output = ejs.render(template, context);
      res.setHeader('Content-type', 'text/html');
      // Отправка HTTP-ответа
      res.end(output);
```

```
    });
  } else {
    res.statusCode = 404;
    res.end('Not found');
  }
});

server.listen(8000);
```

Теперь создайте дочернюю папку template, в которой будут находиться шаблоны. После создания файла students.ejs в папке template структура приложения должна выглядеть так, как показано на рис. 11.4.

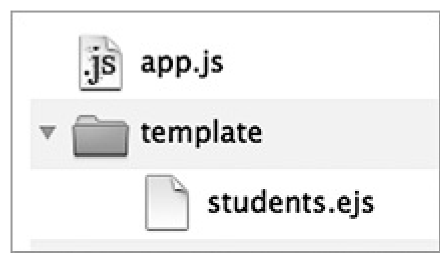



**Рис. 11.4.** Структура приложения с папкой для EJS-шаблонов

Введите код из листинга 11.7 в файл students.ejs.

**Листинг 11.7.** EJS-шаблон для визуализации массива студентов

```
<% if (students.length) { %>
  <ul>
    <% students.forEach(function(student) { %>
      <li><%= student.name %> (<%= student.age %>)</li>
    <% }) %>
  </ul>
<% } %>
```

### *Кэширование EJS-шаблонов*

EJS поддерживает кэширование функций шаблона в памяти. Это означает, что после однократного синтаксического разбора файла шаблона функция, созданная в ходе синтаксического разбора, сохраняется в памяти. Визуализация кэшированного шаблона осуществляется быстрее, поскольку пропускается этап синтаксического разбора.

Если вы занимаетесь разработкой веб-приложения в Node и хотите, чтобы

изменения, внесенные в файлы шаблона, тут же отражались на экране, кэширование следует отключить. Если же вы уже развернули приложение в рабочей среде, кэширование позволяет быстро и просто получить ощутимый выигрыш в производительности. Условное включение кэширования осуществляется через переменную окружения NODE_ENV.

Чтобы проверить, как работает кэширование, измените вызов EJS-функции render из предыдущего примера:

```
var cache = process.env.NODE_ENV === 'production';
var output = ejs.render(
  template,
  {students: students, cache: cache, filename: filename}
);
```

Обратите внимание, что в качестве значения параметра filename не обязательно используется файл. В данном случае может применяться уникальное значение, показывающее, какой шаблон вы визуализируете.

Теперь, когда вы знаете, как интегрировать EJS-шаблоны в Node-приложение, давайте рассмотрим альтернативный вариант применения EJS — в веб-браузерах.

### 11.2.4. Создание клиентских приложений с помощью EJS

Мы уже видели примеры использования EJS с Node, а теперь познакомимся с применением EJS в браузере. Чтобы воспользоваться EJS-шаблоном на сторон клиента, сначала нужно загрузить EJS в рабочую папку с помощью следующих команд:

```
cd /your/working/directory
curl https://raw.github.com/visionmedia/ejs/master/ejs.js -o ejs.js
```

После загрузки файла ejs.js шаблонизатор EJS можно использовать в клиентском коде. В листинге 11.8 представлено простое клиентское приложение с EJS-шаблоном.

**Листинг 11.8.** Добавление в клиентское приложение средств EJS-шаблонизации

```
<html>
  <head>
  <title>EJS example</title>
  <script src="ejs.js"></script>
    // Подключение библиотеки jQuery для DOM-манипуляций
    <script
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js">
```

```
    </script>
  </head>
  <body>

    // Местозаполнитель для вывода визуализируемого шаблона
    <div id='output'></div>

    <script>
      // Шаблон для визуализации контента
      var template = "<%= message %>";
      // Данные для использования с шаблоном
      var context = {message: 'Hello template!'};

        // Ожидание загрузки страницы браузером
        $(document).ready(function() {
          // Визуализация шаблона на устройстве
          // с идентификатором "output"
        $('#output').html(
          ejs.render(template, context)
        );
      });
    </script>
  </body>
</html>
```

Теперь мы знаем, как использовать в Node полнофункциональный шаблонизатор, поэтому настало время познакомиться с шаблонизатором Hogan, в котором функциональность шаблонизации намеренно ограничена.

### 11.3. Использование языка Mustache с шаблонизатором Hogan

Шаблонизатор hogan.js (https://github.com/twitter/hogan.js) был создан в компании Twitter для собственных нужд, касающихся шаблонизации. Hogan представляет собой реализацию популярного стандарта языка шаблонизации Mustache (http://mustache.github.com/), разработанного Крисом Ванстратом (Chris Wanstrath) для проекта GitHub.

В Mustache выбран минималистский подход к шаблонизации. В отличие от EJS и

стандарт Mustache сознательно не включены ни условная логики, ни встроенные средства фильтрации, исключение сделано лишь для экранирующего контента, позволяющего предотвращать XSS-атаки. Разработчики Mustache постарались сделать код шаблона максимально простым.

В этом разделе мы узнаем:

- как создать и реализовать в приложении Mustache-шаблоны;
- какие теги шаблонов определены стандартом Mustache;
- как организовать шаблоны с помощью «компонентов»;
- как выполнить точную настройку Hogan с помощью нестандартных ограничителей и других параметров.

Итак, давайте взглянем на альтернативный подход к шаблонизации, который предлагает Hogan.

### 11.3.1. Создание шаблона

Чтобы использовать Hogan в приложении или выполнить любой из примеров, предлагаемых в этом разделе, нужно установить Hogan в папку приложения. Для этого в командной строке введите следующую команду:

```
npm install hogan.js
```

Далее представлен код Node-приложения, использующего Hogan для визуализации простого шаблона на основе контекста:

```
var hogan = require('hogan.js');
var template = '{{message}}';
var context = {message: 'Hello template!'};

var template = hogan.compile(template);
console.log(template.render(context));
```

В результате выполнения этого кода на экран выводится текст «Hello template!».

Теперь, когда вы знаете, как с помощью Hogan обрабатывать Mustache-шаблоны, давайте рассмотрим теги, предлагаемые стандартом Mustache.

### 11.3.2. Mustache-теги

Концептуально Mustache-теги не отличаются от EJS-тегов. Mustache-теги служат

местозаполнителями для значений переменных, показывая, где нужна итерация, что позволяет расширить функциональность Mustache или добавить в шаблон комментарий.

### *Вывод на экран простых значений*

Чтобы вывести на экран контекстное значение в Mustache-шаблоне, заключите имя значения в двойные фигурные скобки. Подобные скобки среди Mustache-разработчиков получили жаргонное название «mustaches» (дословно — усы). Если, например, нужно вывести на экран значение для контекстного элемента name, воспользуйтесь Hogan-тегом {{name}}.

Подобно большинству других шаблонизаторов, Hogan по умолчанию экранирует контент, чтобы предотвратить возможные XSS-атаки. Для вывода на экран неэкранированного значения в Hogan можно добавить либо третью фигурную скобку, либо предварить контекстный элемент символом амперсанда. Если вернуться к предыдущему примеру, то для вывода неэкранированного контекстного значения воспользуйтесь тегом {{{name}}} или {{&name}}.

Для добавления в Mustache-шаблон комментария применяется такой формат:

```
{{! Это комментарий }}
```

### *Секции — перебор нескольких значений*

Хотя Hogan не позволяет включать логику в шаблоны, этот шаблонизатор предлагает элегантный механизм перебора нескольких значений в контекстном элементе с помощью *секций* (sections).

Например, следующий контекст содержит элемент с массивом значений:

```
var context = {
  students: [
    { name: 'Jane Narwhal', age: 21 },
    { name: 'Rick LaRue', age: 26 }
  ]
};
```

Предположим, вы решили создать шаблон, который выводит на экран данные каждого студента в отдельном HTML-абзаце:

```
<p>Name: Jane Narwhal, Age: 21 years old</p>
<p>Name: Rick LaRue, Age: 26 years old</p>
```

Чтобы получить такой HTML-код, достаточно использовать следующий Hogan-

шаблон:

```
{{#students}}
  <p>Name: {{name}}, Age: {{age}} years old</p>
{{/students}}
```

### *Инвертированные секции — заданный по умолчанию HTML-код при отсутствии значений*

Что произойдет, если элемент students в контекстных данных окажется не массивом? Если, например, это один объект, шаблон просто выведет его на экран. Однако секции не отображаются, если значение соответствующего элемента не определено, ложно или представляет собой пустой массив.

Чтобы шаблон выводил на экран сообщение об отсутствии значения в какой-то секции, Hogan предлагает то, что в терминологии Mustache называется *инвертированными секциями* (inverted sections). Следующий код после включения в описанный ранее шаблон с данными студентов выведет на экран сообщение о том, что в контексте отсутствуют данные о студенте:

```
{{^students}}
  <p>No students found.</p>
{{/students}}
```

### *Лямбда-секция — включение в секции нестандартной функциональности*

Чтобы помочь разработчикам расширить предлагаемую Mustache функциональность, стандарт допускает определение тегов секций, обрабатывающих контент шаблона путем вызова функций, а не перебором элементов массива. Подобная секция называется *лямбда-секцией* (section lambda).

В качестве примера рассмотрим, как с помощью лямбда-секции добавить поддержку парсера Markdown в механизм визуализации шаблона. В этом примере используется модуль github-flavored-markdown, который нужно установить, введя в командной строке команду:

```
npm install github-flavored-markdown
```

В листинге 11.9 объект **Name** в шаблоне визуализируется в разметку <strong>Name</strong> при передаче через парсер Markdown, который вызывается кодом лямбда-секции.

**Листинг 11.9.** Использование лямбда-секции в Hogan

```
var hogan = require('hogan.js');
```

```
// Загружаем парсер Markdown по требованию
var md = require('github-flavored-markdown');

// Mustache-шаблон содержит также Markdown-форматирование
var template = '{{#markdown}}'
        + '**Name**: {{name}}'
        + '{{/markdown}}';

var context = {
  name: 'Rick LaRue',
  markdown: function() {
    return function(text) {
      // Контекст шаблона включает лямбда-секцию, выполняющую
      // синаксический разбор Markdown-формата в шаблон
      return md.parse(text);
    };
  }
};

var template = hogan.compile(template);
console.log(template.render(context));
```

С помощью лямбда-секций можно легко реализовать в шаблонах такие вещи, как кэширование и трансляция.

### *Компоненты — повторное использование шаблонов в других шаблонах*

При написании шаблонов можно избежать нежелательного повторения одного и того же кода в нескольких шаблонах. Один из способов добиться этого — создать *компоненты* (partials). Компоненты — это шаблоны, которые в качестве строительных блоков могут включаться в другие шаблоны. Еще один вариант применения компонентов — разбиение сложных шаблонов на более простые.

Например, в листинге 11.10 с помощью компонента код шаблона, применяемый для вывода на экран данных о студентах, выделяется из главного шаблона.

**Листинг 11.10.** Использование компонентов в Hogan

```
// Код шаблона, используемый для компонента
var hogan = require('hogan.js');
```

```
var studentTemplate = '<p>Name: {{name}}, '
                + 'Age: {{age}} years old</p>';
// Код главного шаблона
var mainTemplate = '{{#students}}'
              + '{{>student}}'
              + '{{/students}}';

var context = {
  students: [{
    name: 'Jane Narwhal',
    age: 21
  },{
    name: 'Rick LaRue',
    age: 26
  }]
};

// Компилирование главного шаблона и компонента
var template = hogan.compile(mainTemplate);
var partial = hogan.compile(studentTemplate);

// Визуализация главного шаблона и компонента
var html = template.render(context, {student: partial});
console.log(html);
```

### 11.3.3. Тонкая настройка Hogan

Использовать Hogan очень просто — достаточно изучить словарь тегов, и можно приступать к работе. В процессе работы шаблонизатора вам может понадобиться изменить всего пару параметров.

Если вам не нравятся фигурные скобки в стиле Mustache, достаточно переопределить используемый разделитель путем передачи методу compile нужного разделителя в качестве параметра. В следующем примере показано, как в Hogan выполнить компиляцию с помощью разделителя в стиле EJS:

```
hogan.compile(text, {delimiters: '<% %>'});
```

Если вам не нравится использовать теги секций, которые начинаются с символа # после открывающей фигурной скобки, задайте параметр sectionTags метода compile. Можно, например, задействовать другой формат для тегов секций, в которых выполняется развертывание лямбда-секций. В результате выполнения кода из листинга 11.11 изменится код, представленный ранее в листинге 11.9. В данном случае префикс в виде подчеркивания позволит отличить тег секции markdown от тегов следующих секций, которые выполняют циклический перебор вместо развертывания лямбда-секций.

**Листинг 11.11.** Использование нестандартных тегов секций в Hogan

```
var hogan = require('hogan.js');
// Загружаем парсер markdown по требованию
var md = require('github-flavored-markdown');

// Нестандартный тег, применяемый в шаблоне
var template = '{{_markdown}}'
          + '**Name**: {{name}}'
          + '{{/markdown}}';

var context = {
  name: 'Rick LaRue',
  // Лямбда-секция для нестандартного тега
  _markdown: function(text) {
    return md.parse(text);
  }
};

var template = hogan.compile(
  template,
  // Заданы нестандартные открывающий и закрывающий теги
  {sectionTags: [{o: '_markdown', c: 'markdown'}]}
);
console.log(template.render(context));
```

При использовании Hogan не нужно изменять какие-либо параметры, чтобы включить кэширование, поскольку оно встроено в функцию compile и включается по умолчанию.

После освоения двух достаточно простых Node-шаблонизаторов давайте

перейдем к шаблонизатору Jade, который подходит к разметке представлений иначе, чем EJS и Hogan.

## 11.4. Шаблонизация с использованием Jade

Jade (http://jade-lang.com) предлагает альтернативный способ спецификации HTML-кода. Ключевое различие между Jade и большинством других систем шаблонизации заключается в том, что в Jade применяются значимые пробельные символы.

При создании Jade-шаблонов задаются отступы, определяющие уровень вложенности HTML-тегов. Кроме того, HTML-теги нельзя явно закрывать, что позволяет исключить проблему случайной преждевременной вставки закрывающих тегов или их полного отсутствия. Благодаря отступам шаблоны визуально становятся менее плотными, что облегчает их сопровождение.

Рассмотрим небольшой пример, демонстрирующий использование Jade-шаблонов для представления следующего HTML-кода:

```
<html>
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <div id="main" class="content">
      <strong>"Hello world!"</strong>
    </div>
  </body>
</html>
```

Для моделирования этого HTML-кода можно задействовать такой Jade-шаблон:

```
html
  head
    title Welcome
  body
    div.content#main
      strong "Hello world!"
```

В Jade, как и в EJS, допускается внедрение JavaScript-кода как на стороне сервера, так и на стороне клиента. Jade предлагает также дополнительные механизмы, такие как наследование шаблонов и примеси (mixins). Благодаря примесям можно определять простые многократно используемые мини-шаблоны, предназначенные для представления в HTML-разметке наиболее востребованных

визуальных элементов, таких как списки и поля. Примеси напоминают применяемые в Hogan компоненты, о которых рассказывалось в предыдущем разделе. Благодаря наследованию облегчается организация Jade-шаблонов, необходимых для визуализации одной HTML-страницы в нескольких файлах. Далее мы подробно рассмотрим эти механизмы.

Чтобы установить Jade в папку Node-приложения, в командной строке введите следующую команду:

```
npm install jade
```

Если при установке Jade указать глобальный флаг −g, открывается доступ к утилите командной строки jade, обеспечивающей быструю визуализацию шаблонов в HTML-разметку. Благодаря использованию этой утилиты файл template/sidebar.jade будет визуализирован в файл sidebar.html, находящийся в папке шаблона. Утилита jade предлагает простой способ для экспериментов с Jade-синтаксисом:

```
jade template/sidebar.jade
```

В этом разделе мы узнаем:

- как в Jade задаются имена классов, атрибуты и блочное расширение, а также получим другую базовую информацию о Jade;
- как вводить программную логику в Jade-шаблоны с помощью встроенных ключевых слов;
- как организовать шаблоны, применяя наследование, блоки и примеси.

Для начала давайте получим основные сведения о Jade, касающиеся использования и синтаксиса этого шаблонизатора.

### 11.4.1. Основные сведения о Jade

В Jade используются те же названия тегов, что и в HTML, но в Jade можно не применять открывающие и закрывающие символы (< и >), а уровень вложенности тегов разрешается выражать отступами.

Тег может включать один или больше CSS-классов, связанных с ним посредством инструкции .<classname>. Элемент div, к которому применены классы content и sidebar, может быть представлен следующим образом:

```
div.content.sidebar
```

CSS-идентификаторы назначаются тегу с помощью префикса #. Вы можете назначить CSS-идентификатор featured_content в предыдущем примере с помощью

следующего Jade-представления:

```
div.content.sidebar#featured_content
```

**Сокращенная запись тега div**

Поскольку тег div используется в HTML-разметке повсеместно, Jade предлагает сокращенную форму его спецификации. Следующий пример кода визуализирует ту же HTML-разметку, что и предыдущий:

```
.content.sidebar#featured_content
```

Теперь, когда вы научились специфицировать HTML-теги и соответствующие им CSS-классы и идентификаторы, давайте займемся спецификацией атрибутов HTML-тегов.

### *Спецификация атрибутов тегов*

Чтобы специфицировать атрибуты тега, заключите их в скобки. Отдельные атрибуты, находящиеся в скобках, разделяются запятыми. Чтобы специфицировать ссылку, которая откроется в другой вкладке, воспользуйтесь следующим Jade-представлением:

```
a(href='http://nodejs.org', target='_blank')
```

Поскольку спецификация атрибутов тегов в Jade может привести к появлению длинных строк, шаблонизатор предлагает более гибкий подход. Следующее Jade-представление вполне корректно и эквивалентно предыдущему:

```
a(href='http://nodejs.org',
  target='_blank')
```

Можно также специфицировать атрибуты, которые не требуют значений. Следующий пример Jade-кода демонстрирует спецификацию HTML-формы которая включает элемент select вместе с предварительно выбранным параметром:

```
strong Select your favorite food:
form
  select
    option(value='Cheese') Cheese
    option(value='Tofu', selected) Tofu
```

### *Спецификация контента тегов*

Мы уже встречали примеры контента тегов. В предыдущем фрагменте кода к нему относится контент Select your favorite food после тега strong, Cheese после первого тега option и Tofu после второго тега option.

Это стандартный, но не единственный способ спецификации контента тегов в Jade. Хотя подобный стиль прекрасно подходит для спецификации небольших фрагментов контента, если контента окажется много, это может привести к Jade-шаблонам со слишком длинными строками. Чтобы избежать подобной проблемы, при спецификации контента в Jade можно использовать символ |:

```
textarea
  | This is some default text
  | that the user should be
  | provided with.
```

Если HTML-тег, такой как style или script, принимает только текст (то есть вложенные HTML-элементы не допускаются), символы | могут полностью исключаться, как показано в следующем примере кода:

```
style
  h1 {
    font-size: 6em;
    color: #9DFF0C;
  }
```

Благодаря наличию двух разных способов выражения длинного и короткого контента тегов облегчается создание элегантных Jade-шаблонов. Кроме того, в Jade поддерживается альтернативный механизм выражения вложенности, который называется *блочным расширением* (block expansion).

### *Упорядочение кода путем блочного расширения*

В Jade вложенные теги обычно выделяются с помощью отступов, но иногда это может привести к слишком большому количеству пробельных символов.

Например, в этом Jade-шаблоне с помощью отступов определяется простой список ссылок:

```
ul
  li
    a(href='http://nodejs.org/') Node.js homepage
  li
    a(href='http://npmjs.org/') NPM homepage
```

```
li
  a(href='http://nodebits.org/') Nodebits blog
```

Благодаря блочному расширению в Jade можно более компактно представить предыдущий пример. В случае блочного расширения для иллюстрации вложенности после тега указывается двоеточие. Следующий код генерирует ту же разметку, что и предыдущий, но вместо семи строк кода у нас осталось только четыре:

```
ul
  li: a(href='http://nodejs.org/') Node.js homepage
  li: a(href='http://npmjs.org/') NPM homepage
  li: a(href='http://nodebits.org/') Nodebits blog
```

Теперь, когда мы узнали, как с помощью Jade моделируется разметка, давайте выясним, как интегрировать Jade-шаблон в веб-приложение.

### *Включение данных в Jade-шаблоны*

Данные передаются в Jade-шаблон так же, как и в EJS-шаблон. Сначала шаблон компилируется в функцию, которая затем вызывается вместе с контекстом, чтобы визуализировать HTML-вывод. Например:

```
var jade = require('jade');
var template = 'strong #{message}';
var context = {message: 'Hello template!'};

var fn = jade.compile(template);
console.log(fn(context));
```

В этом примере кода конструкция #{message} в шаблоне определяет местозаполнитель, который заменяется контекстным значением.

С помощью контекстных значений можно также предоставлять значения для атрибутов. Например:

```
var jade = require('jade');
var template = 'a(href = url)';
var context = {url: 'http://google.com'};

var fn = jade.compile(template);
console.log(fn(context));
```

В этом примере кода визуализируется следующая разметка:

```
<a href="http://google.com"></a>
```

Теперь, когда мы узнали, как с помощью Jade моделируется HTML-разметка и как данные приложений включаются в Jade-шаблоны, пришло время узнать, как встроить в Jade-шаблон программную логику.

### 11.4.2. Программная логика в Jade-шаблонах

Когда вы получаете в свое распоряжение Jade-шаблоны вместе с прикладными данными, вам требуется логика, чтобы как-то обрабатывать эти данные. В Jade можно непосредственно внедрять в шаблоны строки JavaScript-кода — именно с помощью JavaScript-кода вы определяете программную логику в своих шаблонах. Чаще всего в качестве такого кода используются инструкции if, циклы for и объявления var. Но прежде чем углубиться в детали, давайте рассмотрим пример Jade-шаблона, визуализирующего список контактов. Этот пример хорошо иллюстрирует то, как Jade-логика может применяться в приложении:

```
h3.contacts-header My Contacts

if contacts.length
  each contact in contacts
    - var fullName = contact.firstName + ' ' + contact.lastName
    .contact-box
      p fullName
      if contact.isEditable
        p: a(href='/edit/+contact.id) Edit Record
      p
        case contact.status
          when 'Active'
            strong User is active in the system
          when 'Inactive'
            em User is inactive
          when 'Pending'
            | User has a pending invitation
else
  p You currently do not have any contacts
```

Сначала давайте взглянем на различные способы обработки выводимых данных шаблонизатором Jade при внедрении JavaScript-кода.

### *Использование JavaScript-кода в Jade-шаблонах*

Если предварить строку JavaScript-кода символом -, это приведет к выполнению JavaScript-кода без возвращения какого-либо значения в выводимых шаблоном данных. Если же строку JavaScript-кода предварить символом =, возвращаемое этим кодом значение попадет в выводимые данные, причем оно будет экранировано для предотвращения XSS-атак. Если же генерируемое JavaScript-кодом значение экранировать не нужно, можно предварить код символами !=. Перечень доступных префиксов приведен в табл. 11.1.

**Таблица 11.1.** Префиксы, используемые внедренным в Jade-шаблон JavaScript-кодом

| Префикс | Вывод |
|---|---|
| = | Выводимые данные экранируются (для не заслуживающих доверия или непредсказуемых значений, для защиты от XSS-атак) |
| != | Выводимые данные не экранируются (для заслуживающих доверия или предсказуемых значений) |
| - | Выводимые данные отсутствуют |

В Jade включено несколько наиболее востребованных условных и итеративных инструкций, которые можно писать без префиксов: if, else if, else, case, when, default, until, while, each и unless.

Кроме того, в Jade можно определять переменные. В следующем примере кода показаны два эквивалентных способа присваивания значений переменным в Jade:

```
- var count = 0
count = 0
```

Как и упомянутые ранее инструкции с префиксом -, инструкции без префиксов не порождают никаких данных.

### *Циклический перебор объектов и массивов*

В Jade значения, переданные в контексте, доступны JavaScript-коду. В следующем примере кода мы считываем Jade-шаблон из файла, а затем передам в Jade-шаблон контекст, содержащий пару сообщений, которые мы собираемся показывать в массиве:

```
var jade = require('jade');
var fs = require('fs');
var template = fs.readFileSync('./template.jade');
var context = { messages: [
  'You have logged in successfully.',
```

```
  'Welcome back!'
]};

var fn = jade.compile(template);
console.log(fn(context));
```

В результате Jade-шаблон должен включать следующий код:

```
- messages.forEach(function(message) {
  p= message
- })
```

Финальная HTML-разметка должна выглядеть следующим образом:

```
<p>You have logged in successfully.</p><p>Welcome back!</p>
```

В Jade также поддерживается отличная от JavaScript форма итераций, реализуемая инструкцией each. С ее помощью можно циклически перебирать элементы массивов и свойства объектов.

Вот как выглядит эквивалент предыдущего примера, в котором используется инструкция each:

```
each message in messages
  p= message
```

Немного изменив код, можно выполнить циклический перебор свойств объекта:

```
each value, key in post
  div
    strong #{key}
    p value
```

***Код условной визуализации в шаблоне***

Иногда в шаблонах требуется реализовать вывод тех или иных данных на экран в зависимости от значения этих данных. В следующем примере кода показано условие, в котором на протяжении половины всего времени тег script выводится в формате HTML:

```
- var n = Math.round(Math.random() * 1) + 1
- if (n == 1) {
  script
    alert('You win!');
- }
```

В Jade условные выражения могут также создаваться в более чистой

альтернативной форме:

```
- var n = Math.round(Math.random() * 1) + 1
  if n == 1
    script
      alert('You win!');
```

Если в условиях используются отрицания, например if (n != 1), в Jade их можно заменить ключевым словом unless:

```
- var n = Math.round(Math.random() * 1) + 1
  unless n == 1
    script
      alert('You win!');
```

***Использование в Jade инструкции case***

В Jade также поддерживаются условия с инструкцией case, которые напоминают JavaScript-условия с инструкцией switch. С помощью инструкции case вы можете выбрать в шаблоне вариант выводимых данных на основе нескольких сценариев.

Следующий пример шаблона показывает, как с помощью инструкции case вывести на экран результаты поиска в блоге тремя разными способами. Если в результате поиска ничего не найдено, выводится соответствующее сообщение. Если найден единственный пост, он показывается полностью. Если найдено несколько постов блога, с помощью инструкции each выполняется циклический перебор постов с выводом их заголовков:

```
case results.length
  when 0
    p No results found.
  when 1
    p= results[0].content
  default
    each result in results
      p= result.title
```

### 11.4.3. Организация Jade-шаблонов

После определения шаблонов их нужно как-то организовать. Как и в случае с прикладной логикой, нужно стремиться к уменьшению размеров файлов шаблонов. При этом предполагается, что один файл шаблона концептуально должен

соответствовать одному строительному блоку приложения, например странице, врезке или контенту поста в блоге.

В этом разделе мы рассмотрим несколько механизмов, с помощью которых разные файлы шаблонов для визуализации контента объединяются вместе:

- структурирование нескольких шаблонов путем наследования;
- реализация макетов путем добавления блока в начало и в конец шаблона;
- включение шаблона;
- многократное использование программной логики шаблона с помощью примесей.

Сначала мы поговорим о наследовании шаблонов в Jade.

### *Структурирование нескольких шаблонов путем наследования*

Наследование шаблонов — это один из механизмов структурирования шаблонов. В рамках этой концепции шаблоны трактуются аналогично классам в парадигме объектно-ориентированного программирования. Один шаблон может расширять другой шаблон, который в свою очередь может расширять еще один шаблон. Количество уровней наследования определяется лишь соображениями целесообразности.

В качестве простого примера давайте выясним, как путем наследования шаблонов получить базовую HTML-оболочку, в которую можно «упаковать» контент страницы. В рабочем каталоге создайте папку template, в которой будет находиться Jade-файл примера. Для шаблона страницы мы создадим файл layout.jade со следующим Jade-кодом:

```
html
  head
    block title
  body
    block content
```

Шаблон layout.jade содержит только определение HTML-страницы в виде двух *блоков* (blocks). С помощью блоков при наследовании шаблонов задается место, в котором будет находиться контент, предоставляемый производным шаблоном. В файле layout.jade для этого служат два блока: блок title позволяет производному шаблону задать заголовок, а блок content — то, что должно быть показано на

странице.

Затем в папке шаблона, находящейся в рабочем каталоге, создайте файл page.jade. Этот файл шаблона будет наполнять содержимым блоки title и content:

```
extends layout

block title
  title Messages

block content
  each message in messages
    p= message
```

И наконец, добавьте программную логику из листинга 11.12 (это модифицированная версия предыдущего примера данного раздела), предназначенную для вывода результатов применения шаблона и показывающую наследование в действии.

**Листинг 11.12.** Наследование шаблонов в действии

```
var jade = require('jade');
var fs = require('fs');
var templateFile = './template/page.jade';
var iterTemplate = fs.readFileSync(templateFile);
var context = {messages: [
  'You have logged in successfully.',
  'Welcome back!'
]};

var iterFn = jade.compile(
  iterTemplate,
  {filename: templateFile}
);

console.log(iterFn(context));
```

А теперь давайте рассмотрим другое применение механизма наследования шаблонов: вставку блоков в начало и в конец шаблона

***Реализация макетов путем вставки блоков в начало и в конец шаблона***

В предыдущем примере блоки, определенные в файле layout.jade, не содержали контента, что делало задачу включения контента в шаблон page.jade совершенно элементарной. Если же блок в унаследованном шаблоне *содержит* контент, этот контент может не заменяться производными шаблонами, а наращиваться путем добавления блоков в начало и в конец шаблона. Это позволит вам определять самый обычный контент и добавлять его в шаблон, не заменяя существующий контент.

Следующий шаблон, layout.jade, содержит дополнительный блок scripts с контентом (тегом script, который предназначен для загрузки библиотеки jQuery):

```
html
  head
    block title
    block scripts
      script(src='//ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js')
  body
    block content
```

Если вы хотите, чтобы шаблон page.jade дополнительно загружал UI-библиотеку jQuery, можете воспользоваться шаблоном из листинга 11.3.

**Листинг 11.13.** Использование механизма добавления блока в конец шаблона для загрузки дополнительного JavaScript-файла

```
// Этот шаблон расширяет шаблон макета
extends layout
baseUrl = "http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/"

block title
  title Messages

// Определение стилевого блока
block style
  link(rel="stylesheet", href= baseUrl+"themes/flick/jquery-ui.css")

// Добавление блока сценариев к блоку, определенному в макете
block append scripts
  script(src= baseUrl+"jquery-ui.js")

block content
```

```
count = 0
each message in messages
  - count = count + 1
  script
    $(function() {
      $("#message_#{count}").dialog({
        height: 140,
        modal: true
      });
    });
  != '<div id="message_' + count + '">' + message + '</div>'
```

Наследование шаблонов — не единственный путь объединения нескольких шаблонов. Вы также можете использовать для этого Jade-команду include.

### *Включение шаблонов*

Еще одним инструментом организации шаблонов является Jade-команда include. Эта команда встраивает в шаблон контент другого шаблона. Если в шаблон layout.jade из предыдущего примера добавить строку include footer, мы получим следующий шаблон:

```
html
  head
    block title
    block style
    block scripts
      script(src='//ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js')
  body
    block content
    include footer
```

Этот шаблон включает контент шаблона footer.jade в визуализируемую шаблоном layout.jade разметку, как показано на рис. 11.5.

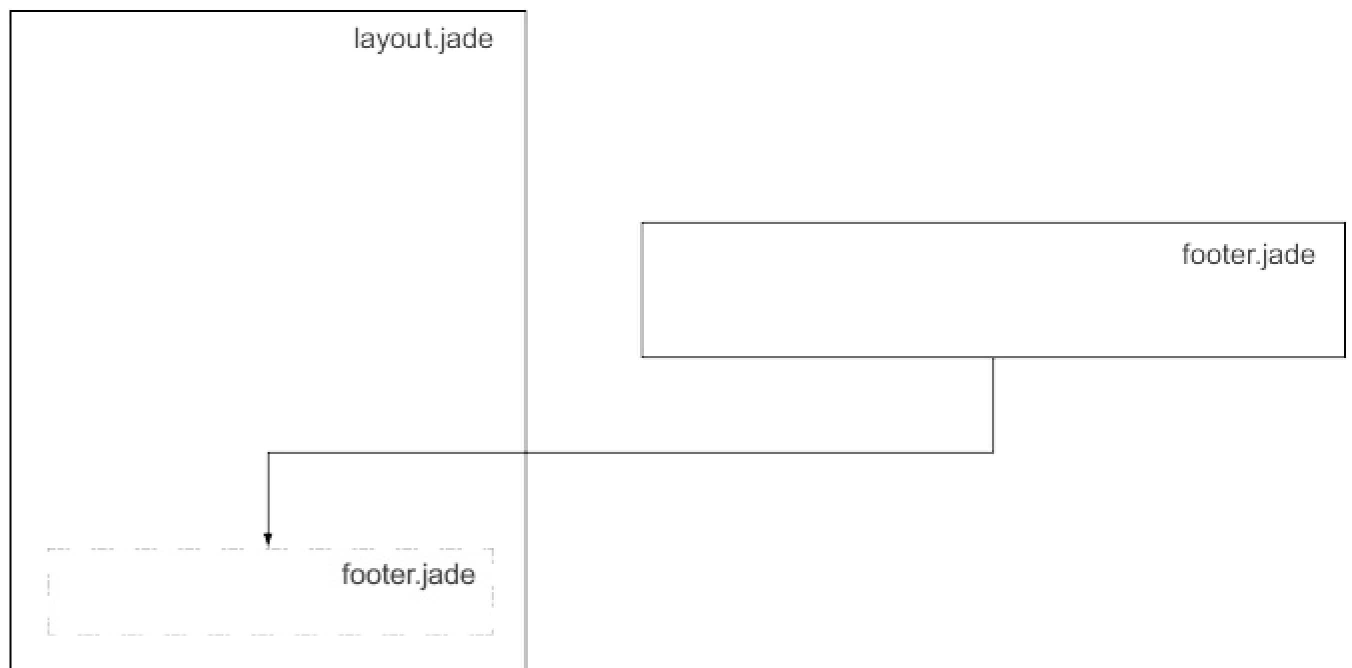



**Рис. 11.5.** В Jade имеется механизм, поддерживающий простой способ встраивания контента одного шаблона в другой шаблон в процессе визуализации

Этот механизм может применяться, например, для добавления в файл layout.jade информации о веб-сайте или элементов дизайна. Указав расширение, можно также включать в шаблон файлы, не являющиеся Jade-файлами, например:

```
include twitter_widget.html
```

### *Многократное использование программной логики шаблона с помощью примесей*

Хотя для использования готовых фрагментов кода может применяться Jade-команда include, с ее помощью вряд ли удастся создать библиотеку многократно используемой функциональности, с которой могли бы совместно работать страницы и приложения. Для решения этой задачи в Jade служит команда mixin, которая позволяет определять многократно используемые фрагменты Jade-кода.

Jade-команда mixin является аналогом JavaScript-функции. Как и функция, команда mixin может принимать аргументы, на основе которых генерируется Jade-код.

Предположим, например, что наше приложение обрабатывает следующую структуру данных:

```
var students = [
  {name: 'Rick LaRue', age: 23},
  {name: 'Sarah Cathands', age: 25},
  {name: 'Bob Dobbs', age: 37}
];
```

Чтобы определить способ вывода HTML-списка, получаемого из значений

заданного свойства каждого объекта, можно задействовать следующую команду mixin:

```
mixin list_object_property(objects, property)
  ul
    each object in objects
      li= object[property]
```

Затем с помощью команды mixin можно вывести данные на экран, воспользовавшись следующей строкой Jade-кода:

```
mixin list_object_property(students, 'name')
```

Благодаря наследованию шаблонов инструкции include и команде mixin вы без проблем можете многократно использовать визуализирующую разметку и не допускать чрезмерного «разбухания» файлов шаблона.

## 11.5. Резюме

Теперь, когда вы узнали, как работают три популярных HTML-шаблонизатора, вы можете применить технику шаблонизации для приведения в порядок прикладной логики и визуализирующей разметки вашего приложения. Node-сообщество создало множество шаблонизаторов, которыми можно воспользоваться, если троица, рассмотренная в этой главе, по тем или иным причинам вас не устроит (https://npmjs.org/browse/keyword/template).

Например, шаблонизатор Handlebars.js (https://github.com/wycats/handlebars.js/) расширяет возможности языка создания шаблонов Mustache за счет дополнительных механизмов, таких как условные теги и глобально доступные лямбда-выражения. В шаблонизаторе Dustjs (https://github.com/akdubya/dustjs) на первое место ставятся быстродействие и такие средства, как потоки данных. Чтобы получить список Node-шаблонизаторов, воспользуйтесь проектом consolidate.js (https://github.com/visionmedia/consolidate.js), который предлагает API-интерфейс, абстрагирующий возможности шаблонизаторов, что упрощает использование в одном приложении сразу нескольких шаблонизаторов. Ну а если идея осваивать другой язык создания шаблонов для вас совершенно неприемлема, обратитесь к шаблонизатору Plates (https://github.com/flatiron/plates), тогда вы сможете средствами языка HTML и программной логики шаблонизатора Plates проецировать в разметке данные приложения на CSS-идентификаторы и классы.

Если вам нравится продвигаемая в Jade идея разделения представления и прикладной логики, обратите внимание на Stylus (https://github.com/LearnBoost/stylus). В этом проекте подобный подход применяется при создании CSS-стилей.

Теперь вы получили в свое распоряжение последние крупицы знаний, необходимые для профессионального создания веб-приложений. В следующей главе мы займемся развертыванием веб-приложений, чтобы представить их всему миру.

9 Roy Thomas Fielding, «Architectural Styles and the Design of Network-based Software Architectures» (PhD diss, University of California, Irvine, 2000), www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

# Часть 3. Идем с Node дальше

В последней части книги мы узнаем, как использовать Node для решения задач, не относящихся к традиционной разработке веб-приложений, и как с помощью Socket.io включать в веб-приложения компоненты реального времени. Вам предстоит познакомиться с методикой создания в Node серверов, не поддерживающих технологию HTTP TCP/IP, и даже утилит командной строки.

Вдобавок вы узнаете, как устроена и работает экосистема Node-сообщества, и каким образом вы можете помочь этой системе, пополнив своими наработками хранилище Node Package Manager.

## Глава 12. Развертывание и обеспечение доступности Node-приложений

- Выбор хоста для Node-приложения
- Развертывание типичного приложения
- Обеспечение доступности и повышение производительности

После завершения разработки веб-приложения наступает этап ввода этого приложения в эксплуатацию. Для каждой веб-платформы существуют методики, направленные на повышение надежности и производительности, и Node не является исключением из этого правила.

Первым этапом развертывания веб-приложения является выбор хоста. Затем нужно выбрать методики мониторинга приложения и поддержания его в работоспособном состоянии. Также следует добиться его максимальной производительности. Все это и является предметом рассмотрения данной главы.

Но сначала мы займемся выбором хоста для приложения.

### 12.1. Хостинг Node-приложений

Большинство разработчиков веб-приложений знакомо с концепцией хостинга PHP-приложений. Когда сервер Apache с PHP-поддержкой получает HTTP-запрос, он проецирует часть запрашиваемого URL-адреса, относящуюся к пути, на требуемый файл, после чего PHP исполняет содержимое этого файла. Подобная технология упрощает процесс развертывания PHP-приложений: вы просто выгружаете PHP файлы в выбранное место файловой системы, где они становятся доступны для веб-браузеров. Помимо простоты развертывания снижается стоимость хостинга

PHP-приложений, поскольку зачастую серверы параллельно обслуживают множество пользователей.

Развертывание Node-приложений с помощью ориентированных на Node служб облачного хостинга, предлагаемых компаниями Joyent, Heroku, Nodejitsu, VMware и Microsoft, более не представляет сложности. Обращаться к таким службам нужно в тех случаях, если вы хотите избежать проблем администрирования своего сервера или хотите воспользоваться инструментами диагностики Node-приложений, такими как Joyent SmartOS, который проверяет, какая программная логика в Node-приложении выполняется медленнее всего. Веб-сайт Cloud9, сам созданный с помощью Node.js, предлагает даже интегрированную среду разработки (Integrated Development Environment, IDE) на базе браузера, с помощью которой можно клонировать GitHub-проекты, работать над ними через браузер, а затем развертывать с помощью различных служб облачного хостинга, ориентированных на Node (табл. 12.1).

**Таблица 12.1.** Ориентированные на Node службы облачного хостинга и IDE-службы

| Название | Веб-сайт |
|---|---|
| Heroku | www.heroku.com/ |
| Nodejitsu | www.nodejitsu.com/ |
| Cloud Foundry от VMware | www.cloudfoundry.com/ |
| Библиотека Microsoft Azure SDK для Node.js | www.windowsazure.com/en-us/develop/nodejs/ |
| IDE Cloud9 | http://c9.io/ |

Альтернативой облачному хостингу Node-приложений является запуск собственного сервера. В качестве платформы для Node-серверов часто выбирают Linux. Эта платформа предлагает больший уровень гибкости, чем службы облачного хостинга приложений, поскольку в Linux можно легко устанавливать произвольные приложения, например серверы баз данных. В то же время службы облачного хостинга Node-приложений обычно предлагают ограниченный набор связанных приложений.

Однако администрирование Linux-серверов требует специальных знаний и навыков. И если вы выбираете этот вариант развертывания, вам потребуется изучить соответствующую документацию, чтобы знать, как выполнять в Linux процедуры установки и сопровождения.

**VIRTUALBOX**

Если вы ранее не имели опыта администрирования серверов, попробуйте поработать

с такими программами, как VirtualBox (www.virtualbox.org/). Эта программа позволит вам запустить виртуальный Linux-компьютер на своей рабочей станции независимо от типа операционной системы, под управлением которой эта станция работает.

Если вы уже знакомы с различными вариантами применения серверов для хостинга Node-приложений, сразу переходите к разделу 12.2, посвященному основам развертывания Node-приложений. Ну а мы сначала посмотрим, какие варианты нам доступны:

- выделенные серверы;
- виртуальные частные серверы;
- многоцелевые облачные серверы.

Давайте определимся, что дает каждый из этих вариантов для хостинга ваших Node-приложений.

### 12.1.1. Выделенные и виртуальные частные серверы

Ваш сервер может быть как реальным (физическим), его обычно называют *выделенным* (dedicated server), так и виртуальным. Виртуальные серверы выполняются на физических серверах, получая доступ к оперативной памяти, процессору и диску физического сервера. Поскольку виртуальные серверы эмулируют физические серверы, вы можете администрировать их так же, как физические. На одном физическом сервере может выполняться несколько виртуальных серверов.

Выделенные серверы обычно обходятся дороже, чем виртуальные, к тому же для установки и настройки выделенных серверов требуется больше времени, поскольку отдельные компоненты нужно разместить по местам, выполнить их сборку и конфигурирование. В то же время виртуальные частные серверы (Virtual Private Servers, VPS) можно установить достаточно быстро, поскольку они создаются на существующих физических серверах.

Если не предвидится быстрый рост количества пользователей, для хостинга веб-приложений лучше всего использовать виртуальные частные серверы. Эти серверы дешевы, к тому же им можно быстро выделять дополнительные ресурсы, такие как место на диске и оперативная память. Соответствующая технология уже устоялась, а многие компании, такие как Linode (www.linode.com/) и Prgmr

(http://prgmr.com/xen/), предлагают решения, позволяющие легко их устанавливать и запускать.

Подобно выделенным серверам, виртуальные частные серверы обычно нельзя создать по требованию. Не позволяют они также справиться с быстрым ростом количества пользователей, поскольку такой рост требует оперативного подключения дополнительных серверов без участия человека. Чтобы справляться с этими задачами, вам нужен облачный хостинг.

### 12.1.2. Облачный хостинг

Облачные серверы подобны виртуальным частным серверам, поскольку тоже являются виртуальными эмуляторами выделенных серверов. Преимущество облачных серверов по сравнению с выделенными и виртуальными частными серверами заключается в том, что управление облачными серверами полностью автоматизировано. С помощью удаленного интерфейса или прикладного программного интерфейса (API) облачные серверы можно создавать, останавливать, запускать и удалять.

Зачем это может понадобиться? Предположим, вы основали компанию, которая использует внутрикорпоративную сеть на базе Node. Предположим также, что вы хотите, чтобы клиенты могли подписаться на предоставляемые услуги, а после оформления подписки получали доступ к собственному серверу, на котором выполняется ваше программное обеспечение. Конечно, можно нанять персонал, который установит и развернет серверы, круглосуточно обслуживающие клиентов, но пока вы не реализуете собственный центр обработки данных, клиентам придется координировать свои усилия с провайдерами выделенного или виртуального частного сервера, чтобы получать своевременный доступ к нужным ресурсам. Управлять облачным сервером можно с помощью API-интерфейса, пересылая инструкции провайдеру облачного хостинга, который при необходимости будет предоставлять вам доступ к новым серверам. Благодаря подобному уровню автоматизации вы сможете предлагать услуги клиентам быстро и без вмешательства человека. На рис. 12.1 показано, как использовать облачный хостинг, чтобы автоматизировать создание и удаление серверов для нашего приложения.

**Рис. 12.1.** Создание, запуск, остановку и удаление облачных серверов можно полностью автоматизировать

Недостатки, связанные с применением облачных серверов, заключаются в их дороговизне (по сравнению с виртуальными частными серверами). Также потребуются некоторые познания, касающиеся специфики облачных платформ.

## *Amazon Web Services*

Amazon Web Services (AWS) — старейшая и наиболее популярная облачная платформа (http://aws.amazon.com/). Эта платформа представляет собой группу различных служб, связанных с хостингом, включая службу доставки электронной почты, сети доставки контента и многие другие. Одной из основных служб AWS является Amazon Elastic Compute Cloud (EC2), которая при необходимости обеспечивает создание облачных серверов.

Виртуальные EC2-серверы называются *экземплярами* и могут управляться как из командной строки, так и через веб-консоль управления (рис. 12.2). Поскольку освоение командной строки для платформы AWS требует определенного времени, начинающим пользователям рекомендуется применять веб-консоль.

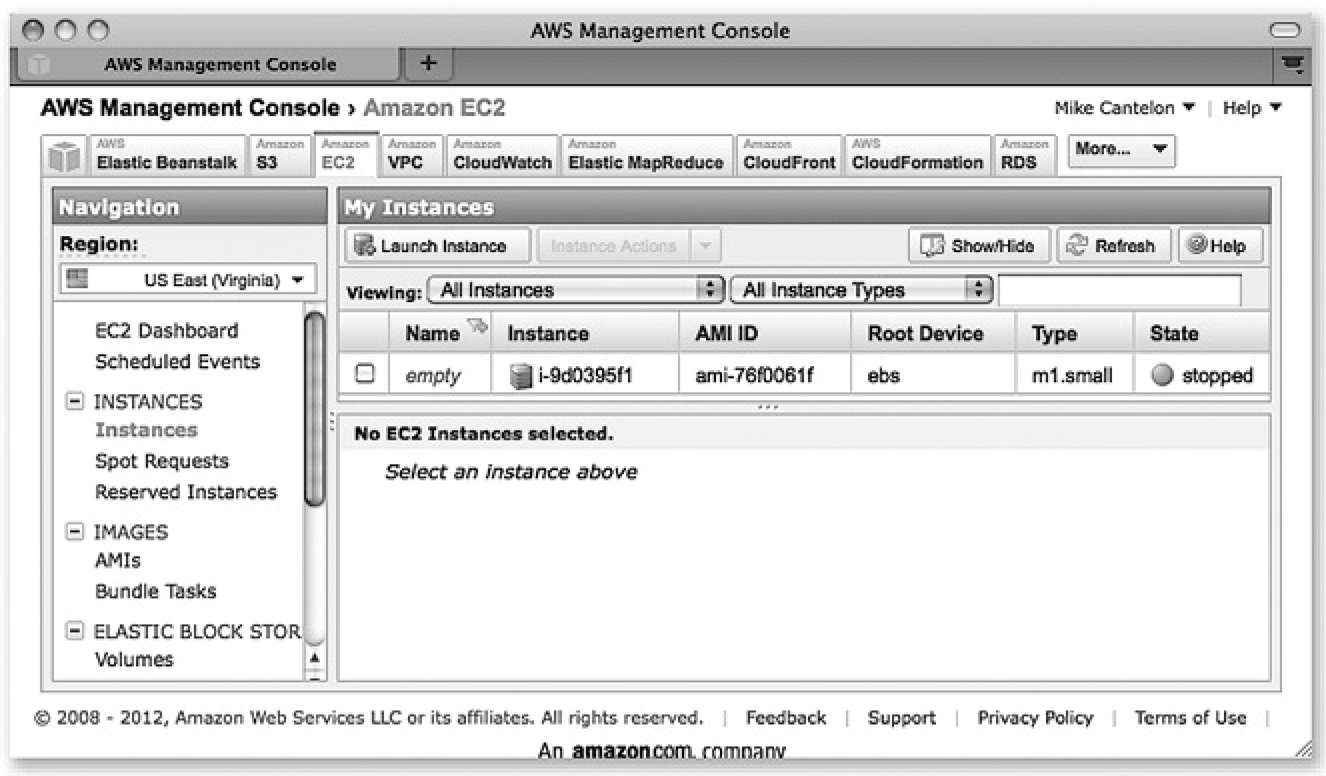



**Рис. 12.2.** Веб-консоль AWS предлагает начинающим пользователям более простой, чем командная строка, способ управления облачными серверами Amazon

Благодаря популярности AWS в Интернете можно легко найти руководства пользователя и прочую документацию, например учебник от Amazon «Getting Started with Amazon EC2 Linux Instances» (http://mng.bz/cw8n).

## *Rackspace Cloud*

Платформа Rackspace Cloud (www.rackspace.com/cloud/) проста в изучении и применении. Однако эта простота оборачивается недостатками, один из которых заключается в том, что Rackspace Cloud предлагает меньший набор облачных решений и функций, чем AWS. К тому же Rackspace Cloud имеет не столь удобный веб-интерфейс. Серверы платформы Rackspace Cloud могут управляться как с помощью веб-интерфейса, так и из командной строки.

В табл. 12.2 обобщаются варианты хостинга, рассмотренные в этом разделе.

**Таблица 12.2.** Обзор вариантов хостинга

| Допустимый рост трафика | Вариант хостинга | Относительная стоимость |
|---|---|---|
| Медленный | Выделенный сервер | Средняя |
| Линейный | Виртуальный частный сервер | Низкая |
| Непредсказуемый | Облачный сервер | Высокая |

После ознакомления с вариантами хостинга Node-приложений давайте

выясним, как запустить Node-приложение на сервере.

## 12.2. Основы развертывания Node-приложений

Предположим, вы создали веб-приложение, которым хотите похвастаться, или коммерческое приложение, которое нужно протестировать перед вводом в эксплуатацию. Вероятно, вы начнете с простого развертывания приложения, а затем проделаете определенную работу, чтобы максимизировать время доступности и производительность приложения. В этом разделе мы проведем простое временное развертывание с помощью программы Git, а также выясним, как с помощью программы Forever сохранять приложение работоспособным и работающим. Временно развернутое приложение не сохраняется после перезагрузки сервера, но зато может быстро выполняться.

### 12.2.1. Развертывание из Git-хранилища

В этом разделе мы познакомимся с базовым развертыванием с помощью Git-хранилища, чтобы вы поняли, из каких основных этапов оно состоит.

Обычно развертывание осуществляется за четыре этапа.

1. Подключение к серверу по протоколу SSH.

2. Установка на сервере платформы Node и при необходимости инструментов контроля версий (например, Git или Subversion).

3. Загрузка из хранилища контроля версий на сервер файлов приложения, включая Node-сценарии, изображения и таблицы CSS-стилей.

4. Запуск приложения.

Вот пример кода приложения, которое запускается после загрузки файлов приложения с помощью программы Git:

```
git clone https://github.com/Marak/hellonode.git
cd hellonode
node server.js
```

Подобно PHP, Node не может выполняться в фоновом режиме. Поэтому базовое развертывание, про которое мы упомянули, требует открытия SSH-соединения. Как только SSH-соединение будет закрыто, выполнение приложения завершится. К счастью, поддерживать выполнение приложения очень легко, если прибегнуть к помощи одного простого инструмента.

**Автоматизированное развертывание**

Существует много способов автоматизировать развертывание Node-приложений. Один из способов заключается в использовании таких инструментов, как Fleet (https://github.com/substack/fleet), с помощью которого можно развертывать приложения на одном или нескольких серверах с помощью команды git push. Более традиционный подход заключается в применении инструмента Capistrano, как описано в статье «Deploying node.js applications with Capistrano» Эвана Тайлера (Evan Tahler), опубликованной в блоге Bricolage (http://mng.bz/3K9H).

### 12.2.2. Поддержание работы Node-приложения

Предположим, вы создали персональный блог с помощью специального приложения для разработки блогов, Cloud9 Nog (https://github.com/c9/nog), и собираетесь развернуть его. При этом нужно гарантировать продолжение выполнения приложения даже в случае разрыва SSH-соединения.

Чаще всего для решения задач подобного рода применяется созданный Node-сообществом инструмент Nodejitsu Forever (https://github.com/nodejitsu/forever). Он позволяет сохранить Node-приложение работающим даже после разрыва SSH-соединения и дополнительно обеспечивает его перезапуск после сбоя. На рис. 12.3 концептуально показано, как работает программа Forever.

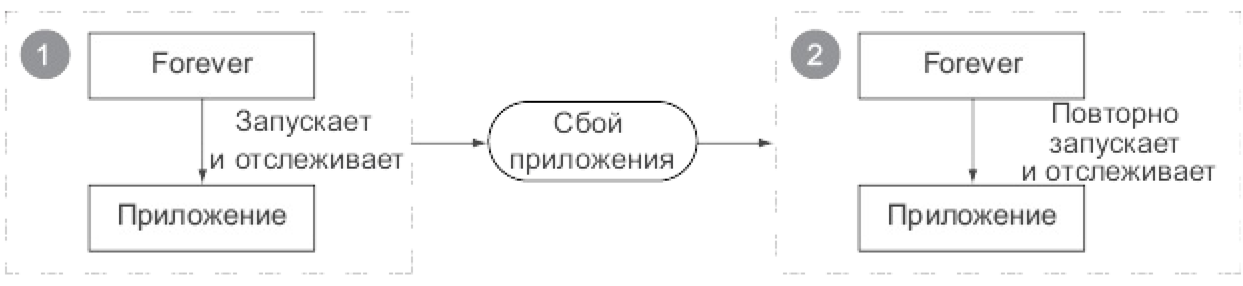



**Рис. 12.3.** Forever обеспечивает выполнение приложения даже в случае сбоя

Чтобы выполнить глобальную установку Forever, воспользуйтесь командой sudo.

Чтобы установить Forever, выполните следующую команду:

```
sudo npm install -g forever
```

После завершения установки воспользуйтесь программой Forever для запуска

блога и поддержания его работающим:

```
forever start server.js
```

**Команда SUDO**

Зачастую в процессе глобальной установки модуля npm (с флагом -g) команду npm нужно предварять командой sudo (www.sudo.ws/). В результате утилита npm запустится с привилегиями суперпользователя. Если команда sudo выполняется впервые, понадобится ввести пароль. Только тогда команда, указанная после пароля, будет выполнена.

Если нужно остановить выполнение блога, введите такую команду:

```
forever stop server.js
```

Чтобы получить список приложений, которыми можно управлять с помощью Forever, воспользуйтесь командой list:

```
forever list
```

Еще одна полезная способность Forever заключается в том, что этот инструмент способен перезагрузить приложение в случае изменения какого-либо исходного файла. Это избавит вас от необходимости каждый раз перезагружать его вручную после добавления какого-либо нового механизма или исправления ошибки.

Чтобы запустить Forever в этом режиме, воспользуйтесь флагом -w:

```
forever -w start server.js
```

Хотя Forever является чрезвычайно полезным для развертывания приложений инструментом, иногда приходится обращаться к иным решениям с более широкими возможностями. В следующем разделе мы познакомимся с корпоративными инструментами мониторинга выполняющихся приложений, а также узнаем, как повысить производительность приложений.

## 12.3. Максимизация времени доступности и производительности приложений

После завершения разработки Node-приложения нужно убедиться в том, что оно запускается и останавливается, когда сервер запускается и останавливается, а также автоматически перезапускается после сбоев сервера. Довольно просто забыть о необходимости остановить приложение перед перезагрузкой сервера или о необходимости перезапустить приложение после перезагрузки сервера.

Также не следует забывать о том, чтобы предпринять шаги по максимизации

производительности приложения. Например, если приложение выполняется на сервере с четырехъядерным процессором, не стоит загружать только одно ядро. Если в этом случае резко возрастет трафик веб-приложения, вычислительных возможностей одного ядра может не хватить для обслуживания трафика, в результате ваше приложения не сможет стабильно отвечать.

Вдобавок к необходимости загрузить все имеющиеся ядра центрального процессора старайтесь не использовать Node с целью хостинга статических файлов для больших корпоративных сайтов. Платформа Node «заточена» под интерактивные приложения, такие как веб-приложения и протоколы TCP/IP, поэтому не может обслуживать статические файлы так же эффективно, как специально предназначенное для этого программное обеспечение. Для обслуживания статических файлов используются такие решения, как программа Nginx (http://nginx.org/en/), которая оптимизирована для решения подобных задач. Также можно выгрузить все статические файлы в сеть доставки контента (Content Delivery Network, CDN), такую как Amazon S3 (http://aws.amazon.com/s3/), а затем ссылаться на эти файлы из приложения.

В этом разделе вы найдете несколько рекомендаций, касающихся доступности и производительности:

- использование программы Upstart для сохранения вашего приложения доступным и работоспособным при перезапусках и сбоях;
- использование кластерного API-интерфейса в Node, чтобы загрузить работой многоядерные процессоры;
- обслуживание статических файлов Node-приложения с помощью программы Nginx.

Начнем мы с рассмотрения очень мощного и удобного инструмента Upstart, предназначенного для поддержания доступности приложений.

### 12.3.1. Поддержание доступности приложения с помощью программы Upstart

Допустим, вы в восторге от своего приложения и собираетесь продать его остальному миру. Вы хотите иметь железную гарантию того, что при перезапуске сервера вы не забудете затем перезапустить приложение. Кроме того, вы хотите иметь гарантию того, что в случае сбоя приложения оно не только автоматически перезапустится, но и информация о сбое будет записана и вас об этом известят, что позволит вам диагностировать причину сбоя.

Проект Upstart (http://upstart.ubuntu.com) предлагает элегантный способ управления запуском и остановкой любого Linux-приложения, включая Node-приложение. Поддержку программы Upstart обеспечивают, в частности, современные версии платформ Ubuntu и CentOS.

Чтобы установить Upstart на платформе Ubuntu, воспользуйтесь следующей командой:

```
sudo apt-get install upstart
```

Для установки Upstart на платформе CentOS используйте команду:

```
sudo yum install upstart
```

После установки Upstart для каждого из ваших приложений нужно добавить конфигурационный файл программы Upstart. Указанные файлы создаются в каталоге /etc/init и имеют названия, подобные имя_приложения.conf. Конфигурационные файлы не обязательно должны помечаться как исполняемые.

С помощью следующей команды создается пустой конфигурационный файл программы Upstart, предназначенный для учебного приложения, рассматриваемого в этой главе:

```
sudo touch /etc/init/hellonode.conf
```

Далее добавьте в конфигурационный файл код из листинга 12.1. Этот код запустит приложение после запуска сервера и остановит его после остановки сервера. То, что должно запускаться программой Upstart, находится в разделе exec.

**Листинг 12.1.** Типичный конфигурационный файл программы Upstart

```
// Указываем имя разработчика приложения
author "Robert DeGrimston"

// Указываем название или описание приложения
description "hellonode"

// Запускаем приложение от имени пользователя nonrootuser
setuid "nonrootuser"



// Запускаем приложение в ходе процедуры начальной загрузки,
// как только становятся доступными файловая система и сеть
start on (local-filesystems and net-device-up IFACE=eth0)

// Останавливаем приложения в ходе процедуры завершения работы
```

stop on shutdown

**// Перезапускаем приложение в случае сбоя**

respawn

**// Записываем информацию в потоки stdin и stderr**

**// в файле /var/log/ upstart/yourapp.log**

console log

**// Устанавливаем все переменные окружения**,

**// необходимые для приложения**

env NODE_ENV=production

**// Задаем команду выполнения приложения**

exec /usr/bin/node /path/to/server.js

Этот конфигурационный файл сохранит ваш процесс запущенным и работающим после перезапуска сервера и даже после его неожиданного сбоя. Весь вывод, генерируемый приложением, будет направляться в файл журнала /var/log/upstart/hellonode.log, причем Upstart проконтролирует обновление журнала, чтобы у вас была самая последняя информация.

После создания конфигурационного файла программы Upstart запустите приложение с помощью следующей команды:

sudo service hellonode

В случае успешного запуска появится сообщение:

hellonode start/running, process 6770

Upstart можно настраивать в очень широких пределах. Описание всех доступных параметров можно найти в онлайновой документации (http://upstart.ubuntu.com/cookbook/).

**Upstart и respawn**

Если указан параметр respawn, Upstart в случае сбоя по умолчанию будет постоянно пытаться перезагрузить приложение до тех пор, пока количество попыток перезагрузки не достигнет 10 в течение 5 секунд. Чтобы изменить это ограничение,

воспользуйтесь параметром respawn limit КОЛИЧЕСТВО ИНТЕРВАЛ. Зд КОЛИЧЕСТВО — это количество попыток внутри заданного (в секундах) ИНТЕРВА. времени. Например, с помощью следующих команд можно задать 20 попыток в течение 5 секунд:

```
respawn
respawn limit 20 5
```

Если приложение пытается перезагрузиться 10 раз на протяжении 5 секунд (ограничение, заданное по умолчанию), это может означать наличие ошибок в коде или конфигурации, способных привести к полной невозможности запуска приложения. Чтобы сохранить ресурсы, которые могут потребоваться другим процессам, Upstart прекращает попытки перезапуска приложения после достижения заданного ограничения.

Поэтому при проверке «здоровья» приложения рекомендуется не полагаться на Upstart, а найти возможность оперативно уведомлять разработчиков о проблемах по электронной почте или с помощью других средств быстрого обмена информацией. Такая проверка может сводиться просто к посещению веб-сайта и ожиданию правильного ответа. При этом можно использовать собственные методы или специальные инструменты, такие как Monit (http://mmonit.com/monit/) или Zabbix (www.zabbix.com/).

Теперь, когда мы узнали, как сделать так, чтобы приложение продолжало работать независимо от сбоев и перезагрузок сервера, следующий логичный предмет нашей заботы — производительность. И в этом нам поможет кластерный API-интерфейс платформы Node.

### 12.3.2. Кластерный API-интерфейс: задействуем несколько ядер

Несмотря на то что у большинства современных процессоров несколько ядер, Node-процесс использует только одно из них. Если вы размещаете Node-приложение на сервере и хотите максимизировать загрузку сервера, можно, хотя и сложно, вручную запустить несколько экземпляров приложения на разных TCP/IP-портах, а затем с помощью системы балансировки загрузки распределить веб-трафик по этим экземплярам.

Чтобы упростить использование нескольких ядер одним приложением, в Node

был включен кластерный API-интерфейс. С его помощью приложению легко запустить несколько рабочих процессов, которые одновременно будут делать одни и те же действия на разных ядрах и использовать один и тот же TCP/IP-порт. На рис. 12.4 показано, как на сервере с четырехъядерным процессором организуется работа приложения с применением кластерного API-интерфейса.

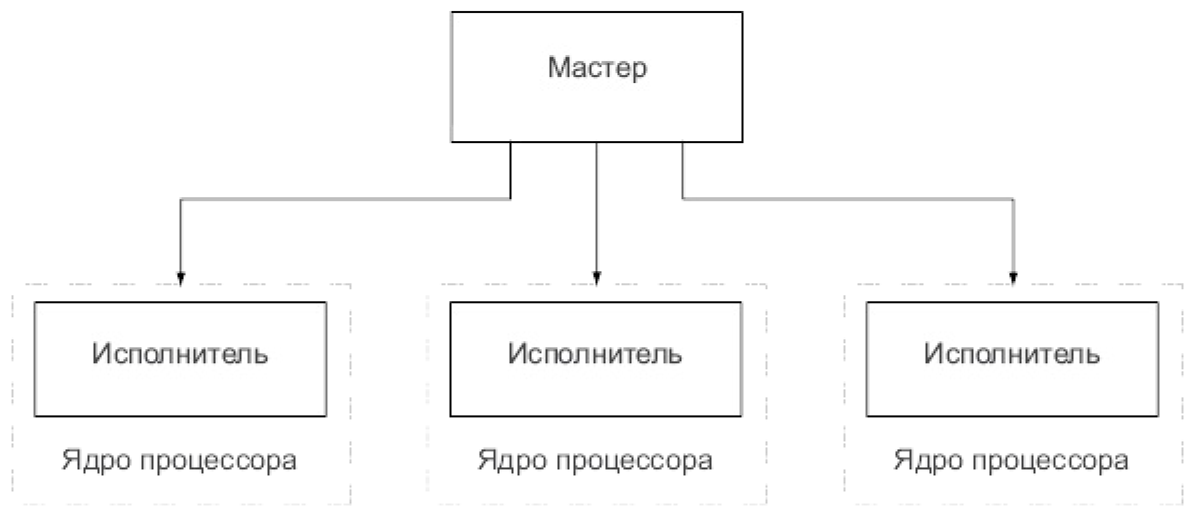



**Рис. 12.4.** Для выполнения на четырехъядерном процессоре мастер породил трех работников

Код из листинга 12.2 автоматически порождает главный процесс (мастер) и дополнительно рабочие процессы (работники) для каждого ядра процессора.

**Листинг 12.2.** Демонстрация кластерного API-интерфейса платформы Node

```
var cluster = require('cluster');
var http = require('http');
// Выясняем количество ядер на процессоре сервера
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Создаем процесс для каждого ядра
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', function(worker, code, signal) {
    console.log('Worker ' + worker.process.pid + ' died.');
  });
} else {
  // Определяем задание, которое должен выполнить каждый работник
  http.Server(function(req, res) {
```

```
    res.writeHead(200);
    res.end('I am a worker running in process ' + process.pid);
  }).listen(8000);
}
```

Поскольку мастер и работники выполняются в разных системных процессах, что является необходимым условием, если они предназначены для работы на разных ядрах, они не могут обмениваться данными состояния через глобальные переменные. Однако здесь на помощь приходит кластерный API-интерфейс, который предоставляет мастеру и работникам средства общения.

В листинге 12.3 показан пример обмена сообщениями между мастером и работниками. Счетчик всех запросов хранится у мастера, и когда какой-то работник рапортует об обработке запроса, об этом сообщается каждому работнику.

**Листинг 12.3.** Демонстрация кластерного API-интерфейса платформы Node

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;
var workers = {};
var requests = 0;

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    workers[i] = cluster.fork();

    (function (i) {
      // Слушаем сообщения от работника
      workers[i].on('message', function(message) {
        if (message.cmd == 'incrementRequestTotal') {
          // Увеличиваем счетчик общего количества запросов
          requests++;
          for (var j = 0; j < numCPUs; j++) {
            // Отправляем каждому работнику новое
            // значение общего числа запросов
            workers[j].send({
              cmd: 'updateOfRequestTotal',
              requests: requests
```

```
            });
          }
        }
      });
    // Используем замыкание для сохранения значения работника
    })(i);
  }

  cluster.on('exit', function(worker, code, signal) {
    console.log('Worker ' + worker.process.pid + ' died.');
  });
} else {
  // Слушаем сообщения от мастера
  process.on('message', function(message) {
    if (message.cmd == 'updateOfRequestTotal') {
      // Обновляем счетчик запросов, основываясь на сообщении мастера
      requests = message.requests;
    }
  });
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end('Worker in process ' + process.pid
      + ' says cluster has responded to ' + requests
      + ' requests.');
    // Даем мастеру знать о необходимости увеличить общее число запросов
    process.send({cmd: 'incrementRequestTotal'});
  }).listen(8000);
}
```

С помощью кластерного API-интерфейса платформы Node очень просто создавать приложения, способные использовать преимущества современного оборудования.

### 12.3.3. Хостинг статических файлов и представительство

Платформа Node в первую очередь предназначена для обслуживания

динамического веб-контента, поэтому она не столь эффективна, когда речь идет о статическом контенте, таком как изображения, таблицы CSS-стилей или клиентский JavaScript-код. Обслуживание статических файлов с помощью протокола HTTP — весьма специфическая задача, для решения которой были оптимизированы специфические программные проекты, которые разрабатывались много лет.

К счастью, веб-сервер с открытым исходным кодом, Nginx (http://nginx.org/en/), оптимизированный для обслуживания статических файлов, можно легко установить вместе с Node именно с целью обслуживания статических файлов. Если рассмотреть типичную конфигурацию Nginx/Node, то Nginx-сервер сначала обрабатывает каждый веб-запрос и транслирует обратно Node те запросы, которые не требуют обработки статических файлов. Соответствующая конфигурация представлена на рис. 12.5.

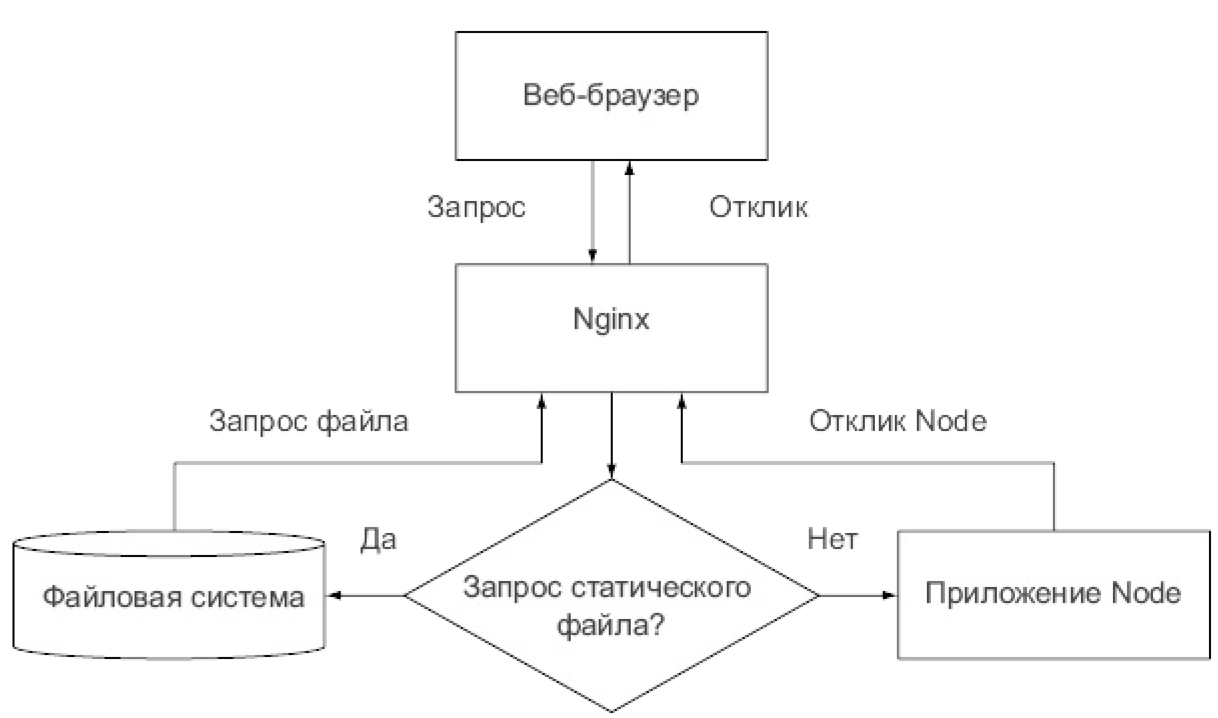



**Рис. 12.5.** Nginx-сервер можно использовать как представитель Node, который быстро возвращает статические ресурсы обратно веб-клиентам

Реализует эту конфигурацию код из листинга 12.4, который помещается в раздел http конфигурационного файла Nginx-сервера. Конфигурационный файл обычно хранится в каталоге /etc Linux-сервера и называется /etc/nginx/nginx.conf.

**Листинг 12.4.** Конфигурационный файл Nginx-сервера, используемого в качестве представителя Node при обслуживании статических файлов

```
http {

  upstream my_node_app {
    // IP-адрес и порт Node-приложения
```

```
    server 127.0.0.1:8000;
  }

  server {

    // Порт, на который представитель будет получать запросы
    listen 80;
    server_name localhost domain.com;
    access_log /var/log/nginx/my_node_app.log;

    // Обрабатываем запросы файлов для URL-путей, которые
    // начинаются с префикса /static/
    location ~ /static/ {
      root /home/node/my_node_app;
      if (!-f $request_filename) {
        return 404;
      }
    }

    // Определяем URL-путь для ответа представителя
    location / {
      proxy_pass http://my_node_app;
      proxy_redirect off;

      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_set_header X-NginX-Proxy true;
    }
  }
}
```

Благодаря использованию Nginx для обработки статических веб-ресурсов можно гарантировать, что Node будет делать только то, что лучше всего умеет.

### 12.4. Резюме

В этой главе мы познакомились с несколькими вариантами хостинга Node-приложений, включая хостинг в стиле Node, хостинг на выделенном сервере, хостинг на виртуальном частном сервере и облачный хостинг. Каждый вариант подходит для своего случая.

Когда вы будете готовы развернуть Node-приложение для ограниченной категории пользователей, можете быстро загрузить и запустить программу Forever для мониторинга приложения. В случае долговременного развертывания можно подумать о том, чтобы автоматизировать запуск и остановку приложения с помощью программы Upstart.

Чтобы включить в работу большую часть ресурсов сервера, можно воспользоваться кластерным API-интерфейсом платформы Node, позволяющим одновременно запускать экземпляры приложения на нескольких ядрах процессора. Если вашему веб-приложению приходится обслуживать статические ресурсы, такие как изображения и PDF-документы, можно запустить Nginx-сервер, который при обработке статических ресурсов будет играть роль представителя вашего Node-приложения.

Теперь, когда у нас есть хорошая ручка, чтобы открывать и закрывать двери веб-приложений, самое время взглянуть, на что еще способна платформа Node. В следующей главе мы познакомимся со всеми остальными вариантами использования Node, начиная от разработки инструментов командной строки и заканчивая добыванием данных с веб-сайтов.

## Глава 13. За пределами веб-серверов

- Использование модуля Socket.IO для обмена данными между браузерами в режиме реального времени
- Реализация TCP/IP-сетей
- Использование API-интерфейсов платформы Node для взаимодействия с операционной системой
- Разработка и использование инструментов командной строки

Благодаря асинхронной природе Node обеспечивается возможность решения задач, связанных с интенсивным вводом-выводом. Подобные задачи невозможно или затруднительно решать в синхронной среде. До сих пор в этой книге мы в

основном говорили о разработке HTTP-приложений, а что можно сказать о разработке приложений других типов? Для чего еще может пригодиться платформа Node?

На самом деле с помощью Node можно создавать не только HTTP-приложения но и другие типы приложений, ориентированные на ввод-вывод данных. На платформе Node можно разрабатывать приложения практически любых типов, в том числе утилиты командной строки, сценарии администрирования систем и веб-приложения реального времени.

В этой главе мы узнаем, как создавать веб-серверы реального времени, которые выходят за рамки модели традиционного HTTP-сервера. Мы также узнаем о некоторых прикладных программных интерфейсах (API) платформы Node, которые можно использовать для создания приложений других типов, таких как TCP-серверы или утилиты командной строки.

И начнем мы с рассмотрения модуля Socket.IO, с помощью которого обеспечивается обмен данными в режиме реального времени между браузерами и сервером.

## 13.1. Socket.IO

Модуль Socket.IO (http://socket.io) является, возможно, наиболее известным модулем, созданным Node-сообществом. Те, кто интересуется разработкой веб-приложений реального времени и вообще никогда даже не слышал о Node, рано или поздно узнаˆют о модуле Socket.IO, который и приводит их прямиком к Node. С помощью Socket.IO можно создавать веб-приложения реального времени, использующие двунаправленный канал обмена данными между сервером и клиентом.

В своей простейшей форме Socket.IO имеет API-интерфейс, очень похожий на API-интерфейс WebSocket (http://www.websocket.org). Отличие заключается в наличии встроенных модулей совместимости, предназначенных для устаревших браузеров, не поддерживающих новые механизмы. Модуль Socket.IO предлагает также удобные API-интерфейсы для широковещания, неустойчивых сообщений и многого другого. Эти возможности сделали модуль Socket.IO очень популярным у разработчиков игр, запускаемых через веб-браузер, приложений для чата и потоковых приложений.

HTTP относится к категории протоколов без сохранения состояния. Это означает, что клиент может передавать серверу только простые недолговечные запросы и сервер не получает реальных данных о подключенных или отключенных пользователях. Наличие подобного ограничения привело к стандартизации протокола WebSocket, определяющего для браузеров механизм поддержания

полнодуплексного соединения с возможностью одновременно получать и отправлять данные на обоих концах соединения. Благодаря API-интерфейсам модуля WebSocket приложения, требующие обмена данными в режиме реального времени между клиентом и сервером, получили «второе дыхание».

Проблема протокола WebSocket заключается в его незавершенности. И хотя некоторые веб-браузеры его уже поддерживают, этого нельзя сказать про многие устаревшие браузеры, в том числе про Internet Explorer. Модуль Socket.IO решает эту проблему, применяя протокол WebSocket при его поддержке браузером, а если такая поддержка отсутствует — теми или иными способами в зависимости от браузера имитируя поведение протокола WebSocket даже для устаревших браузеров.

В этом разделе с помощью модуля Socket.IO мы создадим два учебных приложения:

- минимальное Socket.IO-приложение, сообщающее время сервера подключенным клиентам;
- Socket.IO-приложение, обновляющее страницы при модификации CSS-файлов.

После создания этих учебных приложений мы познакомимся с еще несколькими вариантами применения модуля Socket.IO, вернувшись не некоторое время к учебному приложению из главы 4, показывающему ход выгрузки файлов. Но начнем мы с изучения основ.

### 13.1.1. Создание минимального Socket.IO-приложения

Предположим, мы хотим создать быстрое компактное веб-приложение, которое в режиме реального времени непрерывно обновляет в браузере показания системных часов сервера (UTC-время). С его помощью можно было бы заметить разницу между показаниями часов сервера и клиента. В принципе, подобное приложение можно создать с помощью модуля http или сред разработки, рассмотренных в предыдущих главах. Действительно, так даже можно получить что-то работающее, используя всякие трюки вроде длинных опросов (long-polling), но модуль Socket.IO предлагает для решения этой задачи более прозрачный интерфейс. Реализовать это приложение с помощью Socket.IO почти так же просто, как получить его в подарок.

Прежде чем приступать к созданию приложения, установим модуль Socket.IO с помощью следующей команды:

```
npm install socket.io
```

В листинге 13.1 находится серверный код этого приложения. Сохраните файл с

кодом, мы его проверим, когда у нас также будет клиентский код.

**Листинг 13.1.** Socket.IO-сервер, обновляющий показания часов у клиентов

```
var app = require('http').createServer(handler);
// Наращиваем обычный HTTP-сервер до Socket.IO-сервера
var io = require('socket.io').listen(app);
var fs = require('fs');
var html = fs.readFileSync('index.html', 'utf8');

// Код HTTP-сервера всегда обращается к файлу index.html
function handler (req, res) {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html, 'utf8'));
  res.end(html);
}

// Получаем UTC-представление текущего времени
function tick () {
  var now = new Date().toUTCString();
  // Отправляем показания времени всем подключенным сокетам
  io.sockets.send(now);
}

// Вызываем функцию tick каждую секунду
setInterval(tick, 1000);

app.listen(8080);
```

Как видите, модуль Socket.IO минимизирует объем кода, который нужно добавить в базовый HTTP-сервер. Появляются всего лишь две дополнительные строки, в которых определяется переменная io (переменная для экземпляра Socket.IO-сервера), используемая с целью обмена сообщениями в режиме реального времени между сервером и клиентами. В рассматриваемом примере кода, получающего доступ к системным часам сервера, каждую секунду вызывается функция tick(), которая оповещает всех клиентов, подключенных к серверу, о показаниях системных часов сервера.

Код сервера сначала считывает в память файл index.html, который теперь уже

пора разработать. В листинге 13.2 показана клиентская часть нашего приложения.

**Листинг 13.2.** Socket.IO-клиент, выводящий на экран время сервера, полученное путем широковещательной рассылки

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="/socket.io/socket.io.js">
    </script>
    <script type="text/javascript">
      // Подключение к Socket.IO-серверу
      var socket = io.connect();
      // При получении события message сервер отсылает показания времени
      socket.on('message', function (time) {
        // Обновление элемента time показаниями часов сервера
        document.getElementById('time').innerHTML = time;
      });
    </script>
  </head>
  <body>Current server time is: <b><span id="time"></span></b>
  </body>
</html>
```

***Тестирование***

Теперь все готово к запуску сервера. Запустите сервер с помощью команды node clock-server.js, и вы увидите ответ «info−socket.io started». Это означает, что модуль Socket.IO настроен и готов к приему сообщений. Введите в адресной строке браузера URL-адрес http://localhost:8080/. При наличии некоторого везения вы увидите вывод, представленный на рис. 13.1. Показания времени обновляются каждую секунду на основании сообщений, получаемых с сервера. А теперь откройте другой браузер и в адресной строке введите тот же самый URL-адрес. Вы увидите показания времени, которые изменяются синхронно с показаниями времени сервера.

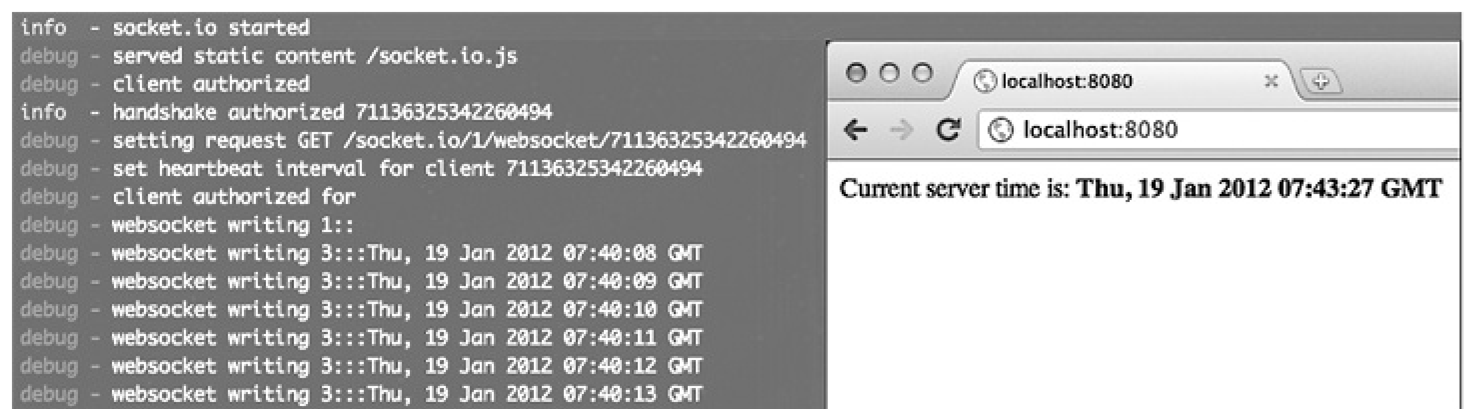



**Рис. 13.1.** Сервер с системными часами в окне терминала и клиентский браузер, подключенный к серверу

Благодаря модулю Socket.IO нам удалось с помощью нескольких строк кода наладить обмен данными между клиентом и сервером.

**Другие разновидности обмена сообщениями с помощью Socket.IO**

Отправка сообщения всем подключенным сокетам — это только один из способов взаимодействия с подключенными пользователями, доступными в Socket.IO. Можно также отправлять сообщения отдельным сокетам, транслировать сообщение всем сокетам за исключением одного, отправлять неустойчивые (необязательные) сообщения и т.п. Дополнительные сведения по этой теме можно найти в документации по Socket.IO (http://socket.io/#how-to-use).

Теперь, когда вы знаете, как с помощью модуля Socket.IO делать простые вещи, давайте рассмотрим другой пример, демонстрирующий, как могут быть полезны для разработчиков события, переданные севером.

### 13.1.2. Использование модуля Socket.IO для вызова страницы и перезагрузки CSS-стилей

Сначала рассмотрим типичную последовательность операций, которую приходится реализовывать разработчику веб-страниц.

1. Открыть веб-страницу в нескольких браузерах.

2. Найти для страницы стили, которые нужно изменить.

3. Внести изменения в одну или несколько таблиц стилей.

4. Вручную выполнить обновление страницы во *всех* веб-браузерах.

5. Вернуться к этапу 2.

Этапом, который хотелось бы автоматизировать, является этап 4, на котором разработчику приходится вручную переходить в окно каждого браузера и щелкать на кнопке Refresh (Обновить). Выполнение этой операции требует особенно много времени, поскольку разработчик должен протестировать разные браузеры на нескольких различных компьютерах и мобильных устройствах.

Можно ли полностью устранить этап ручного обновления веб-страницы? Предположим, когда мы сохраняем таблицу стилей в текстовом редакторе, *все* веб-браузеры, в которых открыта эта страница, автоматически ее обновляют с учетом новых стилей в таблице CSS-стилей. Благодаря исключению этапа ручного обновления разработчики веб-страниц смогли бы экономить массу времени. Реализовать эту функциональность можно с помощью модуля Socket.IO, а также Node-функций fs.watchFile и fs.watch, добавив всего лишь несколько строк кода.

Чтобы обеспечить совместимость со всеми платформами, в данном примере мы используем функцию fs.watchFile() вместо более современной функции fs.watch(), а функцию fs.watch() мы детально рассмотрим позднее.

**Функции fs.watchfile() и fs.watch()**

Node.js предлагает два API-интерфейса для просмотра файлов: функция fs.watchFile() (http://mng.bz/v6dA) является более затратной в смысле расходуемых ресурсов, но зато она более надежна и может работать на многих платформах. Функция fs.watch() (http://mng.bz/5KSC) прекрасно оптимизирована для каждой платформы, но ее применение на отдельных платформах имеет определенные особенности (детали см. в п. 13.3.2).

В этом примере мы совместно используем среду Express и модуль Socket.IO которые могут работать сообща, как было с модулями http.Server и Socket.IO в предыдущем примере.

Сначала рассмотрим серверный код в целом. Сохраните код из листинга 13.3 в файле watchserver.js, если хотите в дальнейшем иметь возможность его запускать.

**Листинг 13.3.** Express/Socket.IO-сервер, который генерирует события при изменении файла

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

```
var http = require('http');
var path = require('path');
var express = require('express');
// Создание сервера Express-приложений
var app = express();
var server = http.createServer(app);
// Оболочка HTTP-сервера, используемая для создания экземпляра Socket.IO
var io = require('socket.io').listen(server);
var root = __dirname;

// Использование компонента промежуточного уровня, чтобы
// начать мониторинг файлов, возвращаемых статическим
// программным обеспечением промежуточного уровня
app.use(function (req, res, next) {
  // Регистрация события static, генерируемого компонентом
  // промежуточного уровня static()
  req.on('static', function () {
    var file = url.parse(req.url).pathname;
    var mode = 'stylesheet';
    if (file[file.length - 1] == '/') {
      file += 'index.html';
      mode = 'reload';
    }
    // Идентификация имени файла, обслуживаемого и вызываемого
    // функцией createWatcher()
    createWatcher(file, mode);
  });
  next();
});

// Установка сервера в качестве базового для обслуживания статических фа
app.use(express.static(root));

// Поддержка списка отслеживаемых активных файлов
```

```
var watchers = {};

function createWatcher (file, event) {
  var absolute = path.join(root, file);
  if (watchers[absolute]) {
    return;
  }

  // Начало мониторинга файла на предмет изменений
  fs.watchFile(absolute, function (curr, prev) {
    // Изменено ли значение mtime (последнее измененное время)?
    // Если изменено, Socket.IO-сервер генерирует событие
    if (curr.mtime !== prev.mtime) {
      io.sockets.emit(event, file);
    }
  });

  // Пометка файла как отслеженного
  watchers[absolute] = true;
}

server.listen(8080);
```

К данному моменту мы создали полнофункциональный сервер статических файлов, подготовленный к передаче событий reload и stylesheet через сеть клиенту с помощью модуля Socket.IO.

А теперь рассмотрим базовый клиентский код. Сохраните код из листинга 13.4 в файле index.html, чтобы в следующий раз после запуска сервера этот код обслуживался в корневой папке.

**Листинг 13.4.** Клиентский код, выполняющий перезагрузку таблиц стилей после получения событий сервера

```
<!DOCTYPE html>
<html>
  <head>
    <title>Socket.IO dynamically reloading CSS stylesheets</title>
    <link rel="stylesheet" type="text/css" href="/header.css" />
```

```
    <link rel="stylesheet" type="text/css" href="/styles.css" />
    <script type="text/javascript" src="/socket.io/socket.io.js">
    </script>
    <script type="text/javascript">
      window.onload = function () {
        // Подключение к серверу
        var socket = io.connect();

        // Получение события reload от сервера
        socket.on('reload', function () {
          window.location.reload();
        });

        // Получение события stylesheet от сервера
        socket.on('stylesheet', function (sheet) {
          var link = document.createElement('link');
          var head = document.getElementsByTagName('head')[0];
          link.setAttribute('rel', 'stylesheet');
          link.setAttribute('type', 'text/css');
          link.setAttribute('href', sheet);
          head.appendChild(link);
        });
      }
    </script>
  </head>
  <body>
    <h1>This is our Awesome Webpage!</h1>
    <div id="body">
      <p>If this file (<code>index.html</code>) is edited, then the
      server will send a message to the browser using Socket.IO telling
      it to refresh the page.</p>

      <p>If either of the stylesheets (<code>header.css</code> or
      <code>styles.css</code>) are edited, then the server will send a
```

```
        message to the browser using Socket.IO telling it to dynamically
        reload the CSS, without refreshing the page.</p>
      </div>
      <div id="event-log"></div>
    </body>
  </html>
```

***Тестирование***

Чтобы приведенный код был работоспособным, нужно создать два CSS-файла, header.css и styles.css, тогда файл index.html в процессе своей загрузки будет загружать и эти два файла.

Теперь, когда в нашем распоряжении имеется серверный код, а также файл index.html и файлы CSS-стилей, используемые браузером, можно приступать к тестированию. Начнем с запуска сервера:

```
$ node watch-server.js
```

После запуска сервера в адресной строке веб-браузера введите адрес [http://localhost:8080](http://localhost:8080), и вы увидите простую HTML-страницу, уже обслуженную и визуализированную. Теперь попробуйте изменить один из CSS-файлов (например, задайте фоновый цвет в теге body). И в окне браузера вы увидите, что таблица стилей перезагружается прямо у вас на глазах, причем сама страница не перезагружается. Попытайтесь открыть эту страницу в нескольких браузерах.

В этом примере используются нестандартные события reload и stylesheet, которые мы определили в приложении и которые не относятся к API-интерфейсу модуля Socket.IO. Это показывает, что объект socket действует как двунаправленный генератор событий EventEmitter, что позволяет применять его для генерирования событий, передаваемых модулем Socket.IO через сеть.

### 13.1.3. Другие применения модуля Socket.IO

Как вы знаете, изначально протокол HTTP никогда не предназначался для поддержания какого бы то ни было обмена данными в режиме реального времени. Однако благодаря появлению таких браузерных технологий, как WebSocket, и модулей, подобных Socket.IO, это ограничение ушло в прошлое. В результате стало возможным появление новых приложений для браузера, обладающих невиданными ранее возможностями.

В главе 4 мы отмечали, что модуль Socket.IO прекрасно подошел бы для ретрансляции обратно в браузер событий, отражающих ход выгрузки файла, чтобы

за выгрузкой мог наблюдать пользователь. Для этого пригодилось бы также нестандартное событие progress:

```
// Обновили предыдущий пример (см. п. 4.4.3)
form.on('progress', function(bytesReceived, bytesExpected) {
  var percent = Math.floor(bytesReceived / bytesExpected * 100);

  // Ретранслируем процентную долю выгруженного файла с помощью Socke
  socket.emit('progress', { percent: percent });
});
```

Для ретрансляции нужно получить доступ к экземпляру сокета, соответствующего тому браузеру, который выполняет выгрузку файла. Однако данный вопрос выходит за рамки темы данной книги, поэтому обратитесь к соответствующим ресурсам в Интернете. Для начала прочитайте статью Даниэля Баулига (Daniel Baulig) «Socket.io and Express: tying it all together», котора опубликована в его блоге *blinzeln* (www.danielbaulig.de/socket-ioexpress.)

Как уже отмечалось, разработчики, интересующиеся средствами создания веб-приложений реального времени, обращаются к Socket.IO, часто даже не подозревая о существовании платформы Node.js. Степень важности и влияния модуля Socket.IO трудно переоценить. Причем это влияние постоянно набирает обороты в сообществах разработчиков веб-игр, позволяя создавать такие креативные игры и приложения, которые прежде казались невозможными. Кроме того, модуль Socket.IO очень часто используется в приложениях, создаваемых на платформе Node.js, например в приложении Node Knockout (http://nodeknockout.com). Какие же восхитительные вещи вы сможете делать с его помощью?

## 13.2. TCP/IP-сети

Платформа Node хорошо подходит для создания сетевых приложений, поскольку они обычно характеризуются интенсивным вводом-выводом. Если не считать HTTP-серверов, которые мы подробно рассмотрим в следующих разделах, Node поддерживает все типы сетей, использующих протокол TCP. Например, Node — это прекрасная платформа для написания почтового сервера, файлового сервера или прокси-сервера, она также может использоваться в качестве клиента этих служб. Node предлагает несколько инструментов, нацеленных на написание высококачественных и производительных приложений ввода-вывода, и в данном разделе мы обязательно их обсудим.

В некоторых сетевых протоколах требуется чтение значений — символов, целых, дробных и других типов данных, включая двоичные — на уровне отдельных байтов.

Для этого в Node используется собственный тип данных, Buffer, который выступает в качестве части двоичных данных фиксированной длины. С помощью этого типа данных обеспечивается доступ к низкоуровневым байтам, требуемый для реализации других протоколов.

В этом разделе рассматриваются следующие темы:

- работа с буферами и двоичными данными;
- создание TCP-сервера;
- создание TCP-клиента.

Сначала мы подробно поговорим о том, как Node обрабатывает двоичные данные.

### 13.2.1. Работа с буферами и двоичными данными

Разработчикам приложений платформа Node предлагает специальный тип данных Buffer. Он действует как фрагмент необработанных данных фиксированной длины. Буфер можно рассматривать как эквивалент функции malloc() языка С или оператора new языка С++. Буферы являются очень быстрыми и легковесными объектами, которые повсеместно используются в API-интерфейсах ядра Node. Например, по умолчанию эти объекты возвращаются в событиях data всеми потоковыми классами.

Node экспонирует конструктор Buffer глобально, что дает нам возможность использовать его в качестве расширения обычных типов данных в JavaScript. С программистской точки зрения буферы подобны массивам. Отличия заключаются в том, что размеры буфера неизменны, причем буфер может содержать только значения в диапазоне от 0 до 255. Несмотря на наличие подобных ограничений, буферы идеально подходят для хранения двоичных данных. Поскольку буферы рассчитаны на работу с байтами в чистом виде, вы можете применять их для реализации любых низкоуровневых протоколов.

#### *Текстовые и двоичные данные*

Предположим, вам нужно сохранить в памяти число 121234869, используя тип данных Buffer. По умолчанию Node предполагает, что вы собираетесь работать с текстовыми данными в буферах, поэтому после передачи в функцию конструктора Buffer строки "121234869" создается новый объект Buffer, в который записывается строковое значение:

```
var b = new Buffer("121234869");

console.log(b.length);
9
console.log(b);
<Buffer 31 32 31 32 33 34 38 36 39>
```

В данном случае возвращается 9-байтовый буфер Buffer. Подобное представление объясняется тем, что строка записывается в Buffer с использованием заданной по умолчанию кодировки UTF-8, в которой каждый символ представляется одним байтом.

В Node также входят вспомогательные функции, выполняющие чтение и запись двоичных (воспринимаемых компьютером) целочисленных данных. Эти функции служат для реализации машинных протоколов, которые передают через сеть необработанные данные (типов int, float, double и т.д.). Процедуру сохранения числовых данных, реализуемую в данном примере, можно сделать более эффективной, если воспользоваться вспомогательной функцией writeInt32LE(). С помощью этой функции число 121234869 сохраняется в виде двоичного целочисленного значения в 4-байтовом буфере Buffer (для процессоров, поддерживающих формат «от младшего бита к старшему»).

Существуют и другие вариации вспомогательных функций класса Buffer:

- writeInt16LE() — для маленьких целочисленных значений;
- writeUInt32LE() — для значений без знака;
- writeInt32BE() — для значений, которые ориентированы на процессоры, поддерживающие формат «от старшего бита к младшему».

Чтобы получить дополнительные сведения по этой теме, обратитесь к документации по API-интерфейсу Buffer (http://nodejs.org/docs/latest/api/buffer.html).

В следующем примере кода число записывается с помощью двоичной вспомогательной функции writeInt32LE:

```
var b = new Buffer(4);
b.writeInt32LE(121234869, 0);

console.log(b.length);
4
```

```
console.log(b);
<Buffer b5 e5 39 07>
```

Благодаря сохранению в памяти значения в виде двоичного целого, а не текстовой строки объем памяти, занимаемой данными, уменьшается в два раза (с 9 до 4 байт). На рис. 13.2 представлена структура двух буферов памяти, а также проиллюстрированы различия между протоколами, поддерживающими данные, воспринимаемые человеком (текст) и компьютером (двоичные данные).

Независимо от того, с каким протоколом вы работаете, Node-класс Buffer сможет организовать вам правильное представление данных.

**Порядок байтов**

Термин «порядок байтов» относится к порядку следования байтов в многобайтовой последовательности. Если байты расположены в последовательности от младшего к старшему, младший значащий бит (Least Significant Byte, LSB) хранится первым, а последовательность байтов считывается в направлении справа налево. В случае расположения байтов от старшего к младшему сначала сохраняется старший байт (Most Significant Byte, MSB), а последовательность байтов считывается в направлении слева направо. В Node.js существуют эквивалентные вспомогательные функции для типов данных, поддерживающих следование байтов от младшего к старшему и от старшего к младшему.

**Рис. 13.2.** Различные представления на байтовом уровне (порядок байтов от младшего к старшему) числа 121234869: в виде текстовой строки и в виде двоичного целого

А теперь самое время начать использовать объекты Buffer, чтобы создать TCP-сервер и обмениваться с ним данными.

### 13.2.2. Создание TCP-сервера

API-интерфейс ядра Node является низкоуровневым, экспонируя для созданных на его основе модулей только самое необходимое. Хороший пример, иллюстрирующий это утверждение, — Node-модуль http. Этот модуль надстраивается над модулем net для реализации протокола HTTP. Другие протоколы, такие как SMTP (для электронной почты) или FTP (для передачи данных), также надстраиваются над модулем net, поскольку API-интерфейс ядра Node не реализует никаких других высокоуровневых протоколов.

#### *Запись данных*

Модуль net предлагает исходный интерфейс TCP/IP-сокета, предназначенный для использования приложениями. API-интерфейс для создания TCP-сервера очень похож на таковой для создания HTTP-сервера. Нужно вызвать метод net.createServer() и передать ему функцию обратного вызова, которая будет

выполняться для каждого подключения. Основное отличие при создании TCP-сервера заключается в том, что функция обратного вызова принимает только один аргумент (обычно он называется socket), который является объектом Socket, а не два аргумента — req и res, — как при создании HTTP-сервера.

**Класс Socket**

В Node класс Socket используется клиентской и серверной частью модуля net Подкласс этого класса Stream включает свойства readable (доступен для чтения) и writable (доступен для записи), то есть он двунаправленный. Этот подкласс генерирует события data в случае, если вводимые данные считаны из сокета. Он также включает функции write() и end(), которые применяются для отправки выводимых данных.

А теперь вкратце рассмотрим класс net.Server, который ожидает подключения, а затем выполняет функцию обратного вызова. В данном случае код функции обратного вызова просто записывает «Hello World!» в сокет и чисто закрывает подключение:

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

net.createServer(function (socket) {
  socket.write('Hello World!\r\n');
  socket.end();
}).listen(1337);
console.log('listening on port 1337');
```

Запустите сервер, чтобы выполнить тестирование:

```
$ node server.js
listening on port 1337
```

Если вы попытаетесь подключиться к серверу с помощью веб-браузера, это может не получиться, поскольку сервер «не говорит на языке HTTP» — только на исходном TCP. Чтобы подключиться к серверу для просмотра сообщений, при подключении нужно использовать подходящий TCP-клиент, например netcat(1):

```
$ netcat localhost 1337
Hello World!
```

Отлично! А теперь попробуем воспользоваться клиентом telnet(1):

```
$ telnet localhost 1337
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello World!
Connection closed by foreign host.
```

Команда telnet обычно выполняется в интерактивном режиме, поэтому тоже выводит свои сообщения. Сообщение «Hello World!» печатается непосредственно перед сообщением о закрытии соединения, как и ожидалось.

Как видите, запись данных в сокет — довольно простой процесс. Достаточно использовать вызовы write(), а вызовом end() все завершить. Этот API-интерфейс намеренно сделан таким, чтобы соответствовать API-интерфейсу HTTP-объекта res при записи ответа клиенту.

### *Чтение данных*

Обычно серверы следуют парадигме «запрос-ответ», предусматривающей после подключения клиента немедленную отправку запроса. Сервер считывает запрос и генерирует ответ, который записывается обратно в сокет. Именно в соответствии с этим принципом работает протокол HTTP, а также большинство других сетевых протоколов. Поэтому нужно не только знать, как записывать данные, но и иметь представление о чтении данных.

Если вы помните, как считывать тело запроса, поступающего из HTTP-объекта req, то чтение из TCP-сокета покажется вам просто забавой. С помощью доступного для чтения интерфейса Stream нужно просто слушать события data, содержащие входные данные, считанные из сокета:

```
socket.on('data', function (data) {
  console.log('got "data"', data);
});
```

По умолчанию сокету не назначается какая-либо кодировка, поэтому в качестве аргумента data будет выступать экземпляр Buffer. Обычно так и нужно (именно поэтому данный вариант предлагается по умолчанию), но когда это удобнее, можно вызвать функцию setEncoding(), чтобы аргумент data включал закодированные строки, а не буферы. Благодаря прослушиванию события end вы также будете знать, когда клиент закрыл свой конец сокета и прекратил отправку данных:

```
socket.on('end', function () {
  console.log('socket has ended');
```

```
});
```

Можно легко написать быстрый TCP-клиент, который ищет строку версии заданного SSH-сервера, просто ожидая первое событие data:

```
var net = require('net');
var socket = net.connect({ host: process.argv[2], port: 22 });
socket.setEncoding('utf8');
socket.once('data', function (chunk) {
  console.log('SSH server version: %j', chunk.trim());
  socket.end();
});
```

Выполните тестирование. Обратите внимание, что в этом чрезвычайно упрощенном примере предполагается, что строка версии целиком находится в одном блоке данных. Как правило, это предположение корректно, но более совершенная программа могла бы буферизировать ввод до тех пор, пока не найдет символ \n. Давайте проверим, что использует SSH-сервер github.com:

```
$ node client.js github.com
SSH server version: "SSH-2.0-OpenSSH_5.5p1 Debian-6+squeeze1+github8"
```

### *Объединение двух потоков данных с помощью функции socket.pipe()*

Использование функции pipe() (http://mng.bz/tuyo) вместе с доступными либо для чтения, либо для записи частями объекта Socket — тоже хорошая идея. Фактически если вы решите написать базовый TCP-сервер, который просто отправляет в ответ клиенту все передаваемые клиентом данные (эхо-сервер), можно воспользоваться единственной строкой кода в функции обратного вызова:

```
socket.pipe(socket);
```

Этот пример показывает, что с помощью одной строки кода можно реализовать протокол IETF Echo Protocol (http://tools.ietf.org/rfc/rfc862.txt), но еще важнее здесь то, что вы можете использовать функцию pipe() как *в направлении* объекта socket, так и в *обратном направлении*. Естественно, обычно это делается для более значимых потоковых экземпляров, таких как поток filesystem или gzip.

### *Обработка нечистых отключений*

Последнее, что нужно рассказать о TCP-серверах, касается необходимости предусмотреть таких клиентов, которые отключаются, но при этом не чисто закрывают сокет. В случае TCP-клиента netcat(1) подобная ситуация случается,

когда вы нажимаете комбинацию клавиш Ctrl+C, которая убивает процесс, вместо Ctrl+D, чтобы чисто закрыть соединение. Для обнаружения подобной ситуации слушайте событие close:

```
socket.on('close', function () {
  console.log('client disconnected');
});
```

Если очистка должна выполняться при отключенном сокете, выполняйте ее после события close, а не после события end, поскольку событие end не генерируется, если соединение не было чисто закрыто.

### *Объединяем все вместе*

Давайте возьмем все упомянутые события и создадим простой эхо-сервер, который выводит информацию на терминал в случае каких-либо событий. Код этого сервера показан в листинге 13.5.

**Листинг 13.5.** Простой TCP-сервер, который просто возвращает обратно клиенту все полученные от него данные

```
var net = require('net');
net.createServer(function (socket) {
  console.log('socket connected!');
  // Событие data может возникать неоднократно
  socket.on('data', function (data) {
    console.log('"data" event', data);
  });
  // Событие end может возникать только один раз для каждого сокета
  socket.on('end', function () {
    console.log('"end" event');
  });
  // Событие close может возникать только один раз для каждого сокета
  socket.on('close', function () {
    console.log('"close" event');
  });
  // Задаем обработчик ошибок, предотвращающий
  // возможность появления неперехваченных исключений
  socket.on('error', function (e) {
    console.log('"error" event', e);
```

```
  });
  socket.pipe(socket);
}).listen(1337);
```

Запустите сервер, подключитесь к нему с помощью команды netcat или telnet, а затем немного «поиграйте» с ним. Вы должны увидеть, что вызовы console.log() для событий печатаются в поток stdout сервера, в то время как вы набираете на клавиатуре в клиентском приложении какую-то мешанину.

Учитывая, что теперь вы умеете разрабатывать низкоуровневые TCP-серверы в Node, вам, вероятно, интересно узнать, как писать в Node клиентские программы, предназначенные для взаимодействия с этими серверами. Тогда вперед!

### 13.2.3. Создание TCP-клиента

Платформа Node предназначена для разработки не только серверных программ, создавать клиентские сетевые программы в Node столь же полезно, сколь просто.

Для создания соединений с TCP-серверами используется функция net.connect(). Эта функция принимает аргумент options со значениями host и port, а затем возвращает экземпляр класса socket. Объект socket, возвращаемый функцией net.connect(), инициирует отключение от сервера, поэтому прежде чем начинать какую-либо работу с сокетом, нужно послушать событие connect:

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


var socket = net.connect({ port: 1337, host: 'localhost' });
socket.on('connect', function () {
  // начинаем писать ваш "запрос"
  socket.write('HELO local.domain.name\r\n');
...
});
```

Как только экземпляр socket подключается к серверу, он начинает себя вести подобно экземплярам socket в функции обратного вызова net.Server.

Давайте напишем базовую реплику команды netcat(1), представленную в листинге 13.6. Обычно программа подключается к указанному удаленному серверу и направляет поток stdin из программы в сокет, а затем направляет ответ сокета в поток stdout программы.

**Листинг 13.6.** Базовая реплика команды netcat(1), реализованная с помощью Noc

```
var net = require('net');
var host = process.argv[2];
```

```
// Синтаксический разбор в аргументах командной
// строки аргументов host и port
var port = Number(process.argv[3]);

// Создаем экземпляр socket и начинаем подключение к серверу
var socket = net.connect(port, host);

// Обрабатываем событие connect при установлении соединения с сервером
socket.on('connect', function () {
  // Направляем поток stdin процесса в сокет
  process.stdin.pipe(socket);
  // Направляем данные сокета в поток stdout процесса
  socket.pipe(process.stdout);
  // Вызываем resume() для потока stdin, чтобы начать считывать данные
  process.stdin.resume();
});

// Приостанавливаем поток sdtdin в случае события end
socket.on('end', function () {
  process.stdin.pause();
});
```

Можете воспользоваться этим клиентом для подключения к разработанным ранее учебным TCP-серверам. Если же вы являетесь фанатом «Звездных войн» вызовите сценарий реплики netcat со следующими аргументами, чтобы увидеть особое «пасхальное яйцо»:

```
$ node netcat.js towel.blinkenlights.nl 23
```

А теперь сидите и наслаждайтесь выводом, показанным на рис. 13.3. Вы заслужили отдых.

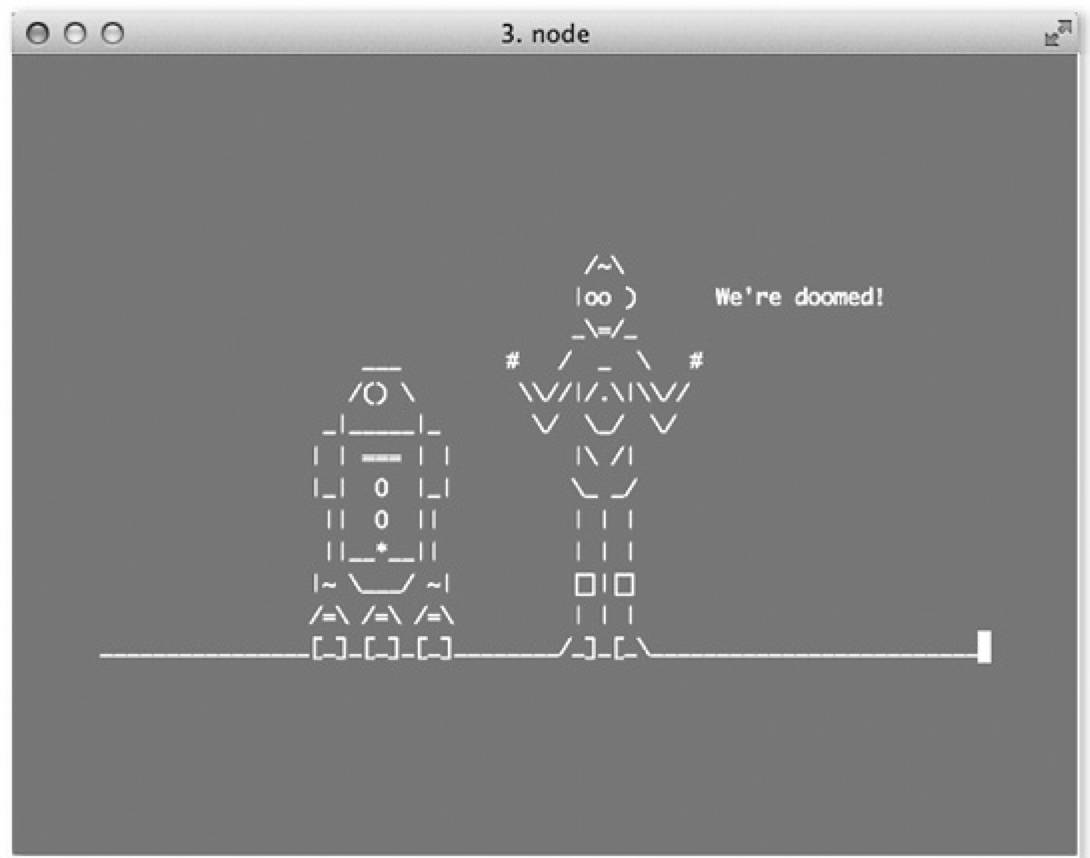



**Рис. 13.3.** Подключение к ASCII-серверу «Звездных войн» с помощью сценария netcat.js

На этом мы завершаем написание низкоуровневых TCP-клиентов и TCP серверов с помощью Node.js. Модуль net предоставляет простой, но в то же время многогранный API-интерфейс, а класс Socket — интерфейсы Stream для чтения и записи. Фактически модуль net — это своеобразная витрина фундаментальных возможностей ядра Node.

Давайте снова переключим передачу и взглянем на API-интерфейсы ядра Node позволяющие взаимодействовать с окружением процесса и запрашивать информацию о времени выполнения и операционной системе.

## 13.3. Инструменты для взаимодействия с операционной системой

Зачастую возникает необходимость во взаимодействии с окружением, в котором выполняется Node. Сюда можно отнести проверку переменных окружения, чтобы включить режим сбора данных о системе в режиме отладки, реализацию в Linux драйвера джойстика, используя низкоуровневые fs-функции для взаимодействия с файлом /dev/js0 (файл устройства для игрового джойстика), или запуск внешнего дочернего процесса, такого как php, для компиляции устаревшего PHP-сценария.

Все эти виды деятельности требуют использования ряда API-интерфейсов ядра Node, которые рассматриваются в этом разделе:

- *Глобальный объект* process содержит информацию о текущем процессе, такую как

переданные ему аргументы, и заданных переменных окружения.

- *Модуль fs* содержит высокоуровневые классы ReadStream и WriteStream, с которыми мы уже знакомы, а также низкоуровневые функции, с которыми мы еще познакомимся.

- *Модуль* child_process содержит как высокоуровневые, так и низкоуровневые интерфейсы, для порождения дочерних процессов, а также специальный механизм порождения экземпляров node с каналом двусторонней передачи сообщений.

Объект process — это один из тех API-интерфейсов, с которым взаимодействует подавляющее большинство программ, поэтому начнем с него.

### 13.3.1. Глобальный процесс-одиночка

У каждого Node-процесса есть один глобальный объект process, доступ к которому имеет каждый модуль. В этом объекте содержится полезная информация о процессе и контекст, в котором он запущен. Например, аргументы, которые были вызваны в Node для выполнения текущего сценария, можно получить с помощью объекта process.argv, а получить или установить переменные окружения — с помощью объекта process.env. Однако самое интересное в объекте process — то, что он является экземпляром EventEmitter, генерирующим весьма специфические события, такие как exit и uncaughtException.

Объект process имеет массу всяких «примочек», и те API-интерфейсы, которые в данном разделе не рассматриваются, будут описаны в этой главе далее. А сейчас мы сфокусируемся на следующем:

- использование объекта process.env для получения и установки переменных окружения;

- прослушивание сгенерированных объектом process специальных событий, таких как exit и uncaught-Exception;

- прослушивание сгенерированных объектом process сигнальных событий, таких как SIGUSR2 и SIGKILL.

***Использование объекта process.env для получения и установки переменных окружения***

С помощью переменных окружения можно изменять режим выполнения программы или модуля. Например, используя эти переменные, можно сконфигурировать сервер, указав, какой порт слушать. Либо операционная система может установить переменную TMPDIR, чтобы указать место для вывода временных файлов, которые позднее можно будет удалить.

**переменные окружения**

Если вы еще не знакомы с переменными окружения, то знайте, что это такие пары «ключ/значение», с помощью которых можно изменить поведение любого процесса. Например, все операционные системы используют переменную окружения PATH для поиска программы по имени среди путей к файлам (при этом ls разрешается в /bin/ls).

Предположим, что в ходе разработки или отладки своего модуля вы хотите включить режим сбора данных о системе, но не хотите, чтобы он оставался включенным при обычном использовании, поскольку это раздражало бы потребителей вашего модуля. Прекрасный инструмент для этого — переменные окружения. Чтобы выяснить, чему равна переменная окружения DEBUG, нужно проверить объект process.env.DEBUG, как показано в листинге 13.7.

**Листинг 13.7.** Определение функции debug по значению переменной окружения DEBUG

```
var debug;
// Задаем функцию debug на основе объекта process.env.DEBUG
if (process.env.DEBUG) {
  debug = function (data) {
    // Если переменная DEBUG задана, функция debug
    // пишет аргумент в поток stderr
    console.error(data);
  };
} else {
  // Если переменная DEBUG не задана, функция debug
  // ничего не делает (она пустая)
  debug = function () {};
}
```

```
// Вызываем функцию debug в разных местах кода
debug('this is a debug call');

console.log('Hello World!');

debug('this another debug call');
```

Если попытаться выполнять этот сценарий, не устанавливая значение переменной окружения process.env.DEBUG, вы увидите, что при вызове функции debug ничего не происходит, поскольку вызываемая функция пуста:

```
$ node debug-mode.js
Hello World!
```

Чтобы протестировать режим отладки, нужно установить переменную окружения process.env.DEBUG. Простейший способ сделать это — запустить экземпляр Node, указав впереди команду DEBUG=1. Тогда в режиме отладки вызовы функции debug будут печататься на консоли вперемешку с обычными данными:

```
$ DEBUG=1 node debug-mode.js
this is a debug call
Hello World!
this is another debug call
```

Это прекрасный способ получения при отладке диагностической информации в потоке stderr с целью устранения ошибок в коде.

Модуль debug, разработанный Т. Дж Головайчуком (T. J. Holowaychuk) (https://github.com/visionmedia/debug), инкапсулирует эту функциональность вместе с несколькими дополнительными функциональными средствами. Если вам понравился описанный здесь прием отладки, вы определенно должны проверить этот модуль в деле.

### *Специальные события, генерируемые процессом*

Обычно объект process генерирует два специальных события:

- событие exit генерируется непосредственно перед завершением процесса;
- событие uncaughtException генерируется при появлении необработанной ошибки.

Событие exit важно для любого приложения, которое должно что-либо сделать

непосредственно перед завершением программы, например очистить объект или вывести на консоль последнее сообщение. Обратите внимание, что событие exit генерируется уже после остановки цикла событий, поэтому после возникновения этого события у вас не будет возможности инициировать какую бы то ни было асинхронную операцию. Код завершения передается событию exit в качестве первого аргумента и в случае успешного завершения он равен нулю.

Давайте напишем сценарий, который слушает событие exit, чтобы напечатать сообщение «Exiting...»:

```
process.on('exit', function (code) {
  console.log('Exiting...');
});
```

Еще одно специальное событие, генерируемое объектом process, — это uncaughtException. В идеальной программе неперехваченных исключений не бывает, но в реальной программе бывает всякое. Событию uncaughtException передается один аргумент — неперехваченный объект Error.

Если у событий «ошибки» не окажется слушателей, любые неперехваченные ошибки будут приводить к краху процесса (это предлагаемое по умолчанию поведение большинства приложений), поэтому нужен хотя бы один слушатель, чтобы решить, что делать с ошибкой. Платформа Node не завершается автоматически, однако считается, что сделать это она должна в собственном обратном вызове. В документации к Node.js есть явно сформулированное предупреждение, суть которого заключается в том, что при любом использовании этого события в обратном вызове должен содержаться вызов функции process.exit(), в противном случае вы оставите приложение в неопределенном состоянии, что всегда плохо.

Давайте послушаем событие uncaughtException и выбросим неперехваченную ошибку, чтобы увидеть все в действии:

```
process.on('uncaughtException', function (err) {
   console.error('got uncaught exception:', err.message);
   process.exit(1);
});


throw new Error('an uncaught exception');
```

Теперь в случае появления неперехваченной ошибки вы сможете перехватить ее и выполнить необходимую очистку перед завершением процесса.

### *Перехват сигналов, отправленных процессу*

В UNIX применяется концепция *сигналов* (signals), которая представляет собой базовую форму взаимодействия между процессами (InterProcess Communication, IPC). Эти сигналы являются весьма примитивными, допуская использование лишь фиксированного набора имен и не имея аргументов для передачи.

Node поддерживает заданные по умолчанию варианты поведения для нескольких сигналов, которые мы сейчас рассмотрим:

- SIGINT. Отправляется оболочкой после нажатия клавиш Ctrl+C. Заданное п умолчанию поведение Node заключается в уничтожении процесса. Это поведение можно переопределить с помощью единственного слушателя сигнала SIGINT для объекта process.

- SIGUSR1. Если получен этот сигнал, Node осуществляет переход к встроенном отладчику.

- SIGWINCH. Отправляется оболочкой при изменении размеров терминала. Пр получении этого сигнала Node сбрасывает свойства process.stdout.rows и process.stdout.columns, а также генерирует событие resize.

Эти три сигнала Node обрабатывает по умолчанию, но вы также можете слушать любой из этих сигналов и выполнять функцию обратного вызова, слушая сигнал для объекта process.

Предположим, вы написали сервер, но когда нажимаете комбинацию клавиш Ctrl+C, чтобы убить сервер, происходит нечистое отключение, что приводит к обрыву всех ожидающих соединений. Для решения проблемы нужно перехватить сигнал SIGINT и запретить серверу принимать соединения, тем самым позволи всем соединениям завершиться перед окончанием процесса. Для этого нужно слушать событие process.on('SIGINT', ...). Название сгенерированного события будет совпадать с названием сигнала:

```
process.on('SIGINT', function () {
  console.log('Got Ctrl-C!');
  server.close();
});
```

Теперь после нажатия комбинации клавиш Ctrl+C из оболочки Node-процессу будет отправлен сигнал SIGINT, в результате вместо уничтожения процесса произойдет выполнение зарегистрированного обратного вызова. Поскольку заданное по умолчанию поведение большинства приложений заключается в завершении процесса, то же самое можно сделать с вашим обработчиком сигналов SIGINT после выполнения всех требуемых действий по завершению работы. В этом

случае запрет серверу на прием входящих соединений поможет делу. Подобная методика может применяться и в Windows, несмотря на недостаточное количество подходящих сигналов, поскольку Node обрабатывает эквивалентные действия Windows и моделирует искусственные Node-сигналы.

Аналогичную методику можно применять для перехвата любых UNIX-сигналов передаваемых Node-процессу. Эти сигналы описаны в статье Википедии, посвященной UNIX-сигналам:http://wikipedia.org/wiki/Unix_signal#POSIX_signals. К сожалению, в Windows сигналы обычно неработоспособны, за исключением нескольких имитированных сигналов: SIGINT, SIGBREAK, SIGHUP и SIGWINCH.

### 13.3.2. Использование модуля fs

В модуль fs включены функции, применяемые для взаимодействия с файловой системой компьютера, на котором выполняется Node. Большинство из этих функций однозначно соответствуют своим аналогам из языка С, но существуют также высокоуровневые абстракции, такие как классы fs.readFile(), fs.writeFile(), fs.ReadStream и fs.WriteStream, которые являются надстройками над функциями open(), read(), write() и close().

#### Синхронные функции в Node.js

Как вы уже знаете, в Node большую часть API-интерфейса составляют асинхронные функции, которые никогда не блокируют цикл событий, но тогда возникает вопрос: зачем беспокоиться о синхронных версиях этих функций файловой системы? Ответ в том, что собственная Node-функция require() является синхронной, а для ее реализации используются функции модуля fs, поэтому и нужны синхронные аналоги. В Node синхронные функции следует применять только во время запуска или начальной загрузки модуля, и никогда более.

Способы применения практически всех низкоуровневых функций идентичны их С-аналогам. Более того, в документации к Node при описании подобных функций указываются ссылки на страницы, описывающие соответствующие С-функции. Вы сможете легко найти эти низкоуровневые функции, поскольку у каждой из них всегда имеется синхронный аналог. Например, функции fs.stat() и fs.statSync() являются низкоуровневыми привязками для С-функции stat(2).

В следующих разделах рассматриваются примеры взаимодействия с файловой системой.

*Перемещение файла*

Относительно простая и очень распространенная задача, возникающая при взаимодействии с файловой системой, заключается в перемещении файла из одной папки в другую. В UNIX для этого используется команда mv, в Windows — команда move. И в Node, наверное должно быть нечто похожее, верно?

Если исследовать модуль fs в REPL-сеансе или обратиться к документации (http://nodejs.org/api/fs.html), вы не найдете там функции перемещения fs.move(). Зато там есть функция переименования fs.rename(), которая, если хорошо подумать, делает то же самое. Так что ура!

Но не спешите радоваться, не все так просто. Функция fs.rename() соответствует C-функции rename(2), которая не поддерживает физические устройства (например, жесткие диски). Это означает, что следующий код не будет корректно выполняться и вызовет ошибку EXDEV:

```
fs.rename('C:\\hello.txt', 'D:\\hello.txt', function (err) {
  // err.code === 'EXDEV'
});
```

Что же делать в этой ситуации? У вас остается возможность создавать новые файлы на диске D:\ и считывать файлы с диска C:\, поэтому копирование файла будет работать. Учитывая это, мы можем создать оптимизированную функцию move(), которая при возможности вызывает очень быструю функцию fs.rename(), а если она не срабатывает, то копирует файл с одного устройства на другое, используя классы fs.ReadStream и fs.Write-Stream. Одна из возможных реализаций представлена в листинге 13.8.

**Листинг 13.8.** Функция move(), которая при возможности выполняет переименование или возвращается к копированию

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


module.exports = function move (oldPath, newPath, callback) {
  // Вызов функции fs.rename() в надежде, что она сработает
  fs.rename(oldPath, newPath, function (err) {
    if (err) {
      // Возвращение к копированию в случае ошибки EXDEV
      if (err.code === 'EXDEV') {
        copy();
      } else {
        // Сбой и информирование вызывающей
```

```
      // функции в случае ошибки другого рода
      callback(err);
    }
    return;
  }
  // Если функция fs.rename() сработала, дело сделано
  callback();
});

function copy () {
  // Считывание исходного файла и его направление в целевую папку
  var readStream = fs.createReadStream(oldPath);
  var writeStream = fs.createWriteStream(newPath);
  readStream.on('error', callback);
  writeStream.on('error', callback);
  readStream.on('close', function () {
    // Отсоединение (удаление) исходного файла
    // после завершения копирования
    fs.unlink(oldPath, callback);
  });
  readStream.pipe(writeStream);
}
}
```

При желании можно протестировать этот модуль непосредственно в REPL-сеансе:

```
$ node
> var move = require('./copy')
> move('copy.js', 'copy.js.bak', function (err) { if (err) throw err })
```

Обратите внимание, что функция копирования работает только с файлами, но не с папками. Чтобы заставить ее работать с папками, сначала нужно проверить, ведет ли заданный путь к папке, и если ведет, то при необходимости вызвать функции fs.readdir() и fs.mkdir(). Можете самостоятельно реализовать этот вариант.

### Коды ошибок модуля fs

Модуль fs возвращает стандартные UNIX-имена для кодов ошибок файловой систем

(www.gnu.org/software/libc/manual/html_node/Error-Codes.html), поэтому их желательно знать. С помощью библиотеки libuv эти имена нормализуются даже в Windows, поэтому вашему приложению достаточно проверять по одному коду ошибки за раз. В соответствии с GNU-документацией ошибка EXDEV происходи если «была зафиксирована попытка создания неправильной ссылки между файловыми системами».

### *Отслеживание изменений в папке или файле*

Функция fs.watchFile() известна уже давно. Использование этой функции на некоторых платформах сопряжено с определенными издержками, поскольку для отслеживания измененных файлов требуется опрос. В процессе отслеживания она вызывает для файла функцию stat(), некоторое время ждет, а затем снова вызывает функцию stat() с образованием непрерывного цикла и вызовом функции отслеживания при изменении файла.

Предположим, мы пишем модуль, призванный записывать изменения, происходящие в системном журнале. Для решения задачи нам нужно обращаться к функции обратного вызова всякий раз, когда изменяется глобальный файл system.log:

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

fs.watchFile('/var/log/system.log', function (curr, prev) {
  if (curr.mtime.getTime() !== prev.mtime.getTime()) {
    console.log('"system.log" has been modified');
  }
});
```

Переменные curr и prev представляют текущий и предыдущий объекты fs.Stat, которые должны иметь различные временные метки, соответствующие времени изменения файла. В рассматриваемом примере сравниваются значения mtime, поскольку нам нужно знать лишь о том, когда файл был изменен, а не когда к нему осуществлялся доступ.

Функция fs.watch() появилась в Node версии v0.6. Как упоминалось ранее, эта функция оптимизирована лучше, чем fs.watchFile(), поскольку при отслеживании файлов она использует собственный API-интерфейс платформы, извещающий об изменении файла. Благодаря этому API-интерфейсу функция также может отслеживать изменения в любом файле в папке. На практике функция fs.watch()

менее надежна, чем функция fs.watchFile(), в силу различий между базовыми механизмами отслеживания файлов различных платформ. Например, параметр filename не может использоваться для получения информации относительно отслеживаемых папок в OS X, и задача Apple — решить эту проблему в будущих версиях OS X. Описание подобных проблем можно найти в документации к Node по адресу: http://nodejs.org/api/fs.html#fs_caveats.

### *Использование модулей сообщества fstream и filed*

Как вы уже знаете, модуль fs, подобно всем другим API-интерфейсам ядра Node, является строго низкоуровневым. Это означает, что он оставляет достаточно места для инноваций и удивительных абстракций, надстраиваемых на верхнем уровне. Коллекция Node-модулей в npm-хранилище день ото дня растет, и как вы уже, наверно, догадались, среди них есть модули, превосходящие по своим возможностям модуль fs.

Например, модуль fstream, разработанный Исааком Шлютером (Isaac Schlueter), является одной из основных составных частей самого mpm-хранилища (https://github.com/isaacs/fstream). Изначально этот модуль создавался в качестве составной части npm. Затем благодаря своей универсальности он был выделен в качестве отдельной единицы и начал применяться для разработки различных приложений командной строки и сценариев системного администрирования. Модуль fstream выделяется благодаря возможности «бесшовной» обработки разрешений и символических ссылок, поддерживаемой по умолчанию при копировании файлов и папок.

С помощью модуля fstream можно делать то же самое, что делает команда cp -rp sourceDir destDir (рекурсивное копирование папки вместе с содержимым, а также передача прав владения и разрешений), просто направляя экземпляр Reader экземпляру Writer. В следующем примере также применяется фильтр модуля fstream для условного исключения файлов в зависимости от функции обратного вызова:

```
fstream
  .Reader("path/to/dir")
  .pipe(fstream.Writer({ path: "path/to/other/dir", filter: isValid )


// Проверяем файл, который мы собираемся записать, и сообщаем,
// нужно его копировать поверх существующего или нет
function isValid () {
  // Игнорирование временных файлов текстовыми
```

```
  // редакторами, такими как TextMate
  return this.path[this.path.length - 1] !== '~';
}
```

Еще один популярный модуль — filed (https://github.com/mikeal/filed), разработанный Майклом Роджерсом (Mikeal Rogers). Отчасти популярность этого модуля объясняется тем, что его написал автор сверхпопулярного модуля request. Известность этим модулям принесло то, что они поддерживают новую разновидность управления порядком выполнения экземпляров Stream: прослушивание события pipe и реагирование различным образом в зависимости от контента, направляемого к ним (или от них).

Чтобы убедиться в перспективности подобного подхода, обратите внимание на то, каким образом модуль filed превращает обычный HTTP-сервер в полнофункциональный статический файловый сервер с помощью всего лишь одной строки кода:

```
http.createServer(function (req, res) {
  req.pipe(filed('path/to/static/files')).pipe(res);
});
```

Этот код обеспечивает передачу свойства Content-Length вместе с нужными кэшированными заголовками. Если у браузера уже есть кэшированный файл, модуль filed ответит на HTTP-запрос кодом 304 Not Modified, пропуская шаги по открытию и чтению файла из дискового процесса. Эти две разновидности оптимизации, которые используются при обработке события pipe, стали возможными, поскольку экземпляр модуля filed получает доступ к объектам req и res HTTP-запроса.

В этом разделе мы рассмотрели два хороших примера модулей, разработанных сообществом и расширяющих возможности модуля fs. С их помощью можно делать невероятные вещи и экспонировать великолепные API-интерфейсы, но это далеко не все существующие модули. Для поиска опубликованных модулей, предназначенных для решения конкретной задачи, воспользуйтесь командой npm search. Предположим, что нужно найти модуль, который облегчает копирование файлов из одного местоположения в другое. Для поиска требуемого модуля введите команду npm search copy. Когда вы найдете интересующий вас опубликованный модуль, выполните команду npm info module-name, чтобы получить сведения о модуле, такие как описание, домашняя страница и опубликованные версии. Просто не забывайте о том, что прежде чем приступать к написанию собственного модуля с целью решения той или иной конкретной задачи, никогда не мешает поискать подходящий модуль в npm-хранилище.

### 13.3.3. Порождение внешних процессов

Node предоставляет модуль child_process, предназначенный для создания дочерних подпроцессов сервера или сценария. Для этого существует два API-интерфейса: высокоуровневая функция exec() и низкоуровневая функция spawn(). Выбор зависит от того, что вы хотите получить. Существует также особый способ создания дочерних процессов самой платформы Node с использованием специального встроенного IPC-канала fork(). Все перечисленные здесь функции имеют собственные варианты применения:

- cp.exec() — высокоуровневый API-интерфейс для порождения команд и буферизации результата в обратном вызове;
- cp.spawn() — низкоуровневый API-интерфейс для порождения одиночных команд в объекте Child-Process;
- cp.fork() — специальный механизм порождения дополнительных Node-процессов с помощью встроенного IPC-канала.

Мы рассмотрим каждую из этих функций в следующих разделах.

#### Преимущества и недостатки дочерних процессов

Использование дочерних процессов имеет свои преимущества и недостатки. Один очевидный недостаток заключается в том, что исполняемая программа должна устанавливаться на машине пользователя, что делает ее зависимостью для вашего приложения. Альтернативный подход заключается в использовании JavaScript-кода для выполнения всего того, что обычно выполняет дочерний процесс. Хорошим примером, иллюстрирующим эту концепцию, является команда npm, которая изначально использовалась системной командой tar для извлечения Node-пакетов. При этом возникали проблемы, причиной которых были конфликты из-за несовместимости версий tar, а также из-за того, что на компьютерах под управлением Windows команда tar устанавливалась очень редко. В результате появилась команда node-tar (https://github.com/isaacs/node-tar), которая была полностью написана на JavaScript и не использовала дочерние процессы.

В то же время использование внешних приложений позволяет разработчикам окунуться в мир других языков программирования. Например, приложение gm (http://aheckmann.github.com/gm/) представляет собой модуль, использующий

библиотеки GraphicsMagick и ImageMagick для выполнения в Node-приложении все
разновидностей преобразования изображений.

### *Буферизация результатов выполнения команды с помощью функции cp.exrc()*

Высокоуровневый API-интерфейс cp.exec() применяется в том случае, когда вам нужен только результат выполнения команды и не нужен доступ к данным из полученных дочерних потоков ввода-вывода. Этот API-интерфейс позволяет вводить полные последовательности команд, включающие множество процессов, направляемых от одного к другому.

Одно из показательных применений API-интерфейса exec() — получение от пользователя команд для исполнения. Предположим, вы написали IRC-бот и хотите выполнять команды, когда пользователь вводит что-то, начинающееся с точки (.). Например, если пользователь в качестве IRC-сообщения набирает .ls, бот выполняет команду ls и печатает выводимые данные обратно в IRC-комнату. Как показано в листинге 13.9, нужно еще установить параметр timeout, чтобы бесконечные процессы по истечении некоторого периода времени автоматически уничтожались.

**Листинг 13.9.** Применение API-интерфейса cp.exec() для выполнения команд, вводимых пользователем через IRC-бот

```
var cp = require('child_process');

// Объект room представляет подключение к IRC-комнате
// (из некоего теоретического IRC-модуля)
room.on('message', function (user, message) {
  // Событие message генерируется для каждого IRC-сообщения,
  // отправленного в комнату
  if (message[0] === '.') {
    // Проверяем наличие точки в контенте сообщения
    var command = message.substring(1);
    cp.exec(command, { timeout: 15000 },
      function (err, stdout, stderr) {
        // Порождаем дочерний процесс и возвращаем результат
        // Node-буферизации в обратном вызове, время ожидания 15 секунд
        if (err) {
```

```
        room.say(
          'Error executing command "' + command +
            '": ' + err.message
        );
        room.say(stderr);
      } else {
        room.say('Command completed: ' + command);
        room.say(stdout);
      }
    }
  );
}
});
```

В mpm-хранилище есть несколько хороших модулей, которые реализуют протокол IRC, поэтому если у вас нет желания писать собственный IRC-бот, вам определенно нужно воспользоваться одним из существующих модулей (в mpm-хранилище популярностью пользуются модули irc и irc-js).

Если вам нужно буферизировать вывод команды, но вы хотите, чтобы платформа Node автоматически экранировала аргументы, воспользуйтесь функцией execFile(). Эта функция принимает четыре аргумента (вместо трех), исполняемый файл передается вместе с массивом аргументов, которые вызываются с исполняемым файлом. Это может быть полезно, если нужно постепенно создать аргументы, которые собирается использовать дочерний процесс:

```
cp.execFile('ls', [ '-l', process.cwd() ],
function (err, stdout, stderr) {
  if (err) throw err;
  console.error(stdout);
});
```

### *Порождение команд потоковым интерфейсом с помощью функции cp.spawn()*

Низкоуровневым API-интерфейсом для порождения дочерних Node-процессов является функция cp.spawn(). Эта функция отличается от функции cp.exec(), поскольку возвращает объект ChildProcess, с которым можно взаимодействовать. Вместо того чтобы предоставлять единственную функцию обратного вызова после завершения процесса, как в случае функции cp.exec(), функция cp.spawn() позволяет

взаимодействовать с каждым потоком ввода-вывода в дочернем процессе индивидуально.

Основная форма применения функции cp.spawn() выглядит следующим образом:

```
var child = cp.spawn('ls', [ '-l' ]);

// stdout – это регулярный экземпляр Stream, генерирующий
// события 'data', 'end', и .т.д.
child.stdout.pipe(fs.createWriteStream('ls-result.txt'));

child.on('exit', function (code, signal) {
  // генерируется при завершении дочернего процесса
});
```

Первый аргумент — это программа, которую вы хотите выполнить. Это может быть единственное имя программы, которое ищется в текущей переменной окружения PATH или по абсолютному пути к программе. Второй аргумент — это массив строковых аргументов для вызова с процессом. По умолчанию объект ChildProcess содержит три встроенных экземпляра класса Stream, с которыми, естественно, должен взаимодействовать сценарий:

- child.stdin — доступный для записи класс Stream, представляющий дочерний поток stdin;
- child.stdout — доступный для чтения класс Stream, представляющий дочерний поток stdout;
- child.stderr — доступный для чтения класс Stream, представляющий дочерний поток stderr.

С этими потоками можно выполнять различные действия, например направлять их в файл, в сокет или в поток, доступный для записи. Можно также их полностью игнорировать, если хотите.

Еще одно интересное событие — exit; оно генерируется объектами ChildProcess, когда процесс завершается, а связанные с этим процессом потоковые объекты уже завершены.

Хороший пример модуля, который абстрагирует использование функции cp.spawn() в полезную функциональность, — это модуль node-cgi (https://github.com/TooTallNate/node-cgi). С его помощью в HTTP-сервера

платформы Node можно многократно использовать устаревшие CGI-сценарии (Common Gateway Interface — общий шлюзовый интерфейс). Реально CGI-интерфейс был просто стандартом, который применялся для ответа на HTTP-запросы путем вызова CGI-сценариев в качестве дочерних процессов HTTP-сервера со специальными переменными окружения, описывающими запрос. Например, можно создать CGI-сценарий, использующий sh в качестве CGI-интерфейса:

```
#!/bin/sh
echo "Status: 200"
echo "Content-Type: text/plain"
echo
echo "Hello $QUERY_STRING"
```

Если бы вы решили присвоить этому файлу имя hello.cgi (не забудьте выполнить команду chmod +x hello.cgi, чтобы сделать этот файл исполняемым), то смогли бы легко вызывать его для HTTP-запросов HTTP-сервера в качестве программной логики ответа. При этом используется единственная строка кода:

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

var server = http.createServer( cgi('hello.cgi') );
server.listen(3000);
```

При установленном сервере и поступлении этому серверу HTTP-запроса модуль node-cgi обрабатывает запрос, делая две вещи:

- порождает новый сценарий hello.cgi в качестве дочернего процесса с помощью функции cp.spawn();
- передает новому процессу контекстную информацию о текущем HTTP-запросе с помощью специального набора переменных окружения.

Сценарий hello.cgi использует одну из специфичных для CGI переменных окружения, QUERY_STRING, которая содержит строку информационного запроса, являющуюся частью URL-адреса запроса. Эту переменную сценарий использует в ответе, записывающем информацию в поток stdout сценария. Если вы запустите этот учебный сервер и отправите HTTP-запрос с помощью команды curl, вы увидите что-то подобное следующему:

```
$ curl http://localhost:3000/?nathan
Hello nathan
```

Существует множество хороших вариантов использования дочерних Node-процессов, и модуль node-cgi — один из примеров. Когда ваш сервер (или приложение) начнет делать то, что вы от него хотите, вы неизбежно придете к мысли о том, что он (оно) вам еще не раз где-нибудь пригодится.

### *Распределение рабочей нагрузки с помощью функции cp.fork()*

Последний API-интерфейс модуля child_process предлагает особый способ порождения дополнительных Node-процессов, но со специальным встроенным IPC-каналом. Поскольку вы уже порождали саму платформу Node, то знаете, что первый аргумент, передаваемый в cp.fork(), представляет собой путь к исполняемому модулю Node.js.

Подобно cp.spawn(), функция cp.fork() возвращает объект ChildProcess. Основное отличие заключается в API-интерфейсе, добавленном за счет IPC-канала. В результате дочерний процесс имеет функцию child.send (message), а сценарий, вызываемый функцией fork(), может слушать события .on('message') процесса.

Предположим, вы хотите написать на платформе Node такой HTTP-сервер который вычисляет последовательность чисел Фибоначчи. Вы могли бы попытаться создать его «в одном флаконе», как показано в листинге 13.10.

**Листинг 13.10.** Неоптимальная реализация в Node.js HTTP-сервера вычисляющего числа Фибоначчи

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

function fib (n) {
// Вычисляем число Фибоначчи
  if (n < 2) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}

var server = http.createServer(function (req, res) {
  var num = parseInt(req.url.substring(1), 10);
  res.writeHead(200);
  res.end(fib(num) + "\n");
});
```

```
server.listen(8000);
```

Если вы запустите сервер с файлом fibonacci-naive.js и отправите HTTP-запрос по адресу http://localhost:8000, сервер будет работать, как ожидалось, но вычисление последовательности чисел Фибоначчи для заданного значения потребует чрезмерного расходования вычислительных ресурсов процессора. Пока результат вычисляется в единственном программном потоке сервера, дополнительные HTTP-запросы не могут обслуживаться. К тому же для выполнения вычислений используется только одно ядро центрального процессора, а остальные простаивают. Это плохо.

Лучшее решение заключается в том, чтобы разветвлять Node-процессы для каждого HTTP-запроса и поручать дочернему процессу выполнение сложных вычислений с возвращением результата. Функция cp.fork() предлагает для этого чистый интерфейс.

В таком решении используются два файла:

- файл fibonacci-server.js будет сервером;
- файл fibonacci-calc.js будет выполнять вычисления.

Сначала рассмотрим код сервера:

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

var server = http.createServer(function(req, res) {
  var child = cp.fork(__filename, [ req.url.substring(1) ]);
  child.on('message', function(m) {
    res.end(m.result + '\n');
  });
});
server.listen(8000);
```

Сервер использует функцию cp.fork() для размещения логики, вычисляющей числа Фибоначчи, в отдельном Node-процессе, который будет отправлять результаты обратно родительскому процессу с помощью функции process.send(), как показано в следующем сценарии fibonacci-calc.js:

```
function fib(n) {
  if (n < 2) {
    return 1;
```

```
    } else {
      return fib(n - 2) + fib(n - 1);
    }
  }

  var input = parseInt(process.argv[2], 10);
  process.send({ result: fib(input) });
```

Можете запустить сервер с файлом fibonacci-server.js и снова отправить HTTP-запрос по адресу http://localhost:8000.

Это прекрасный пример того, что разделение различных компонентов приложения на несколько процессов дает ощутимую выгоду. Функция cp.fork() предоставляет функции child.send() и child.on('message') для отправки сообщений потомку и получения сообщений от потомка соответственно. Внутри самого дочернего процесса функции process.send() и process.on('message') отправляют и получают сообщения. Пользуйтесь ими!

Давайте еще раз переключим передачу и обратимся к разработке в Node инструментов командной строки.

### 13.4. Создание инструментов командной строки

Еще одна задача, традиционно решаемая с помощью Node-сценариев, — создание инструментов командной строки. К данному моменту вы уже должны быть знакомы с весьма мощным инструментом командной строки, созданным в Node, — диспетчером Node-пакетов (Node Package Manager, npm). Этот диспетчер выполняет массу операций в файловой системе, а также порождает дочерние процессы, причем все это делается с помощью платформы Node и ее асинхронных API-интерфейсов. В результате npm может устанавливать пакеты параллельно, а не последовательно, что делает этот процесс гораздо быстрее. И тот факт, что *столь* сложный инструмент командной строки был создан на платформе Node, свидетельствует о ее поистине безграничных возможностях.

Потребности большей части программ командной строки связаны с обычными для процессов операциями, такими как синтаксический разбор аргументов командной строки, чтение из потока stdin и запись в потоки stdout и stderr. В этом разделе рассматриваются общие требования к написанию полнофункциональных программ командной строки, в том числе:

- синтаксический разбор аргументов командной строки;
- работа с потоками stdin и stdout;

• цветовое оформление выводимых данных с помощью модуля ansi.js.

Чтобы приступить к созданию программ командной строки, нужно уметь читать аргументы, с которыми пользователь вызывает программу. Этим мы сейчас и займемся.

### 13.4.1. Синтаксический разбор аргументов командной строки

Синтаксический разбор аргументов является простым и предсказуемым процессом. Для его реализации Node предлагает свойство process.argv, представляющее собой строковый массив, в котором находятся аргументы, используемые при вызове Node. Первая запись в массиве — исполняемый файл платформы Node, вторая запись — имя сценария. Для синтаксического разбора и дальнейшего использования этих аргументов нужно просто перебрать все записи массива и проверить значение каждого аргумента.

Давайте создадим небольшой сценарий args.js, печатающий результаты синтаксического разбора свойства process.argv. Поскольку первые две записи массива большую часть времени не потребуются, можете «отсечь» их с помощью функции slice():

```
var args = process.argv.slice(2);
console.log(args);
```

Если запустить этот сценарий сам по себе, получится пустой массив. Причина в том, что массиву не были переданы дополнительные аргументы:

```
$ node args.js
[]
```

Если же передать аргументы «hello» и «world», массив будет содержать строковые значения, как и ожидалось:

```
$ node args.js hello world
[ 'hello', 'world' ]
```

Как и в случае с любым другим терминальным приложением, для объединения аргументов, содержащих пробелы, используются кавычки. Это правило диктует не Node, а используемая оболочка (bash в UNIX или cmd.exe в Windows):

```
$ node args.js "tobi is a ferret"
[ 'tobi is a ferret' ]
```

В соответствии с соглашением, принятым в UNIX, в ответ на установку флагов -h и —help для произвольной программы командной строки выводится инструкция по использованию этой программы, затем осуществляется выход из программы. В

листинге 13.11 представлен пример применения функции Array#forEach() для циклического перебора аргументов и синтаксического разбора в обратный вызов, а также для печати инструкций по использованию, если встречается ожидаемый флаг.

**Листинг 13.11.** Синтаксический разбор свойства process.argv с помощью функции

```
// Отсечение двух записей, которые неинтересны
var args = process.argv.slice(2);

// Циклический перебор аргументов в поиске флага -h или –help
args.forEach(function (arg) {
  switch (arg) {
    case '-h':
    case '—help':
    printHelp();
    // При необходимости добавляем сюда
    // дополнительные переключатели для флагов
    break;
  }
});

// Выводим на печать информативное сообщение и выходим
function printHelp () {
  console.log(' usage:');
  console.log(' $ AwesomeProgram <options> <file-to-awesomeify>');
  console.log(' example:');
  console.log(' $ AwesomeProgram —make-awesome not-yet.awesome');
  process.exit(0);
}
```

Можно легко расширить блок switch, чтобы включить в синтаксический разбор дополнительные переключатели. Node-сообщество предлагает ряд модулей, таких как commander.js, nopt, optimist и nomnom, которые можно использовать вместо блока switch для синтаксического разбора аргументов. Как это обычно бывает в программировании, для решения любой задачи существует несколько способов.

Еще одна задача, которую должна решать любая программа командной строки, — чтение ввода из потока stdin и запись структурированных данных в поток stdout. Давайте посмотрим, как она решается в Node.

### 13.4.2. Работа с потоками stdin и stdout

Как правило, UNIX-программы компактны, автономны и направлены на решение единственной задачи. Эти программы комбинируются с помощью каналов передачи данных, когда результаты выполнения одного процесса направляются другому процессу — и так до конца цепочки команд. Например, чтобы воспользоваться стандартными UNIX-командами для выборки уникального списка авторов из заданного Git-хранилища, можно скомбинировать команды git log, sort и uniq:

```
$ git log —format='%aN' | sort | uniq
Mike Cantelon
Nathan Rajlich
TJ Holowaychuk
```

Эти команды выполняются параллельно, выводимые первым процессом данные направляются следующему процессу до тех пор, пока цепочка процессов не завершится. В рамках идиомы использования каналов в Node поддерживаются два объекта Stream, с которыми работает программа командной строки:

- process.stdin — объект ReadStream, с которого считываются вводимые данные;
- process.stdout — объект WriteStream, в который записываются выводимые данные.

Эти объекты ведут себя подобно потоковым интерфейсам, с которыми мы уже знакомы.

***Запись выводимых данных с помощью потока proccess.stdout***

Доступный для записи поток process.stdout используется неявно всякий раз, когда вызывается функция console.log(). Внутренне функция console.log() вызывает функцию process.stdout.write() после форматирования аргументов ввода. Однако консольные функции больше подходят для отладки и инспектирования объектов. Чтобы записывать структурированные данные в поток sdtdout, можно непосредственно вызывать функцию process.stdout.write().

Предположим, что программа по протоколу HTTP подключилась к URL-адресу и записывает ответ в поток stdout. В этом контексте хорошо работает функция Stream#pipe():

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

```
var target = url.parse(process.argv[2]);
var req = http.get(target, function (res) {
  res.pipe(process.stdout);
});
```

Вуаля! Минимальная реплика curl занимает всего семь строк кода. Не так уж и плохо, не правда ли? А теперь займемся объектом process.stdin.

### *Чтение вводимых данных с помощью потока process.stdin*

Прежде чем получить возможность чтения данных из потока stdin, нужно вызвать функцию process.stdin.resume(), чтобы дать знать о том, что сценарий нуждается в чтении данных из потока sdtdin. После этого поток stdin будет вести себя подобно любому другому доступному для чтения потоку, генерирующему события data после получения данных от другого процесса или нажатия пользователем клавиш в окне терминала.

В листинге 13.12 представлена программа командной строки, которая запрашивает у пользователя возраст и на основании этой информации принимает решение о дальнейшем выполнении.

**Листинг 13.12.** Программа, ограничивающая доступ пользователя на основании его возраста

```
// Установка возрастного ограничения
var requiredAge = 18;

// Задаем вопрос пользователю
process.stdout.write('Please enter your age: ');

// Настраиваем поток stdin для генерирования строк UTF-8 вместо буферов
process.stdin.setEncoding('utf8');

process.stdin.on('data', function (data) {
  // Синтаксический разбор данных в число
  var age = parseInt(data, 10);
  // Если пользователь ввел некорректное число, выводим
  // соответствующее сообщение
  if (isNaN(age)) {
    console.log('%s is not a valid number!', data);
```

```
  // Если возраст пользователя меньше 18 лет, выводим сообщение,
  // в котором приглашаем пользователя зайти через несколько лет
  } else if (age < requiredAge) {
    console.log('You must be at least %d to enter, ' +
    'come back in %d years',
    requiredAge, requiredAge - age);
  } else {
    // Если предыдущие условия удовлетворены, продолжаем выполнение
    enterTheSecretDungeon();
  }
  // Ожидание единственного события data перед закрытием потока stdin
  process.stdin.pause();
});


process.stdin.resume();
// Вызываем функцию resume(), чтобы начать чтение, поскольку
// поток process.stdin начинает выполняться после приостановки
function enterTheSecretDungeon () {
  console.log('Welcome to The Program :)');
}
```

***Диагностика входа в систему с помощью потока process.stderrr***

В каждом Node-процессе имеется еще один доступный для записи поток, который называется process.stderr и ведет себя подобно потоку process.stdout. Отличие заключается в том, что запись данных производится в поток stderr. Поскольку поток stderr обычно резервируется для отладочных целей, а не для отправки структурированных данных и использования каналов, вместо непосредственного доступа к потоку process.stderr обычно вызывается функция console.error().

После изучения встроенных в Node потоков ввода-вывода, которые нужны при создании любой программы командной строки, давайте познакомимся с чем-то чуть более красочным (простите за каламбур).

### 13.4.3. Окрашивание выводимых данных

В большинстве инструментов командной строки текст окрашивается, что позволяет лучше различать выводимые на экран данные. В Node окрашивание выводимых данных происходит в REPL-сеансе, как это реализовано в npm-хранилище для разных уровней записи. Это приятное дополнительное средство будет нелишним в любой программе командной строки. Окрашивание данных довольно просто реализовать, особенно с помощью модулей, предлагаемых Node-сообществом.

### *Создание и запись управляющих ANSI-кодов*

Цвета на терминале воспроизводятся с помощью *управляющих кодов* кодировки ANSI (American National Standards Institute — Американский национальны институт стандартов). Управляющие коды представляют собой простые текстовые последовательности, которые записываются в поток stdout и приводят к определенным изменениям в терминальном окне, позволяя, в частности, изменить цвет текста, позицию курсора на экране, воспроизвести звук и т.д.

Начнем с простого примера. Чтобы напечатать на экране слово «hello», окрашенное в зеленый цвет, достаточно одного вызова функции console.log():

```
console.log('\033[32mhello\033[39m');
```

Если вы присмотритесь к выводимым в результате данным, то заметите, что слово «hello», находящееся в середине строки, с двух сторон ограничено странными символами. Назначение этих символов объясняет рис. 13.4, на котором окрашенная в зеленый цвет строка «hello» разбита на три части.



**Рис. 13.4.** Вывод зеленого слова «hello» с помощью управляющих ANSI-кодов

#### Управляющие ANSI-Коды в Windows

Технически ни Windows, ни командная строка этой операционной системы (cmd.exe) не поддерживают управляющие ANSI-коды. К счастью, Node интерпретируе управляющие ANSI-коды в Windows, когда ваш сценарий пишет их в поток stdout, а затем вызывает соответствующие Windows-функции, которые возвращают те же

самые результаты. Это интересно, хотя вряд ли вам понадобиться при разработке собственных Node-приложений.

Терминал распознает множество управляющих кодов, запоминать которые просто бессмысленно. Да и не нужно, поскольку Node-сообщество предоставляет модули colors.js, clicolor и ansi.js, облегчающие использование цвета в программах.

### *Форматирование цветов переднего плана с помощью модуля ansi.js*

Давайте рассмотрим модуль ansi.js (https://github.com/TooTallNate/ansi.js), который можно установить с помощью команды npm install ansi. Этот модуль более гибкий по сравнению с другими модулями для работы с цветом (которые обрабатывают строки поочередно), поскольку реализует очень тонкий слой, надстроенный над исходными ANSI-кодами. В модуле ansi.js можно задать *режимы* (modes) потока (например, «bold»), которые будут активны до тех пор, пока вы не очистите их одним из вызовов reset(). В качестве дополнительного бонуса примите к сведению тот факт, что ansi.js — это первый модуль, который поддерживает 256-цветные терминалы. Также этот модуль может конвертировать цветовые CSS-коды (например, #FF0000) в цветовые ANSI-коды.

В модуле ansi.js используется концепция *курсора* (cursor), который представляет собой оболочку вокруг экземпляра потока, доступного для записи, содержащую множество удобных функций, предназначенных для записи ANSI-кодов в поток данных. Все эти функции поддерживают цепочки команд. Например, чтобы с помощью синтаксиса ansi.js опять вывести на печать слово «hello», окрашенное в зеленый цвет, нужен следующий код:

```
var ansi = require('ansi');
var cursor = ansi(process.stdout);


cursor
  .fg.green()
  .write('Hello')
  .fg.reset()
  .write('\n');
```

Как видите, сначала с помощью модуля ansi.js создается экземпляр cursor из потока, доступного для записи. Поскольку нужно окрасить выводимые программой данные, в качестве доступного для записи потока, который будет использовать

курсор, передается поток process.stdout. При наличии класса cursor вы сможете вызывать любой из предлагаемых им методов, чтобы изменить способ визуализации на терминале текстовых данных. В данном случае получается результат, который эквивалентен результату рассмотренного ранее вызова функции console.log():

- cursor.fg.green() — выбирает зеленый цвет переднего плана;
- cursor.write('Hello') — записывает окрашенный в зеленый цвет текст «Hello» на терминал;
- cursor.fg.reset() — восстанавливает цвет переднего плана, заданный по умолчанию;
- cursor.write('\n') — завершает вывод символом новой строки.

Программная настройка вывода с помощью класса cursor позволяет получить чистый интерфейс, способный изменять цвета.

### *Форматирование фоновых цветов с помощью модуля ansi.js*

Модуль ansi.js поддерживает также фоновые цвета. Чтобы установить фоновый цвет вместо цвета переднего плана, замените символы fg в вызове символами bg. Например, для выбора красного фонового цвета воспользуйтесь командой cursor.bg.red().

А сейчас мы быстро напишем программу, которая выводит на терминал разноцветное название книги (рис. 13.5).

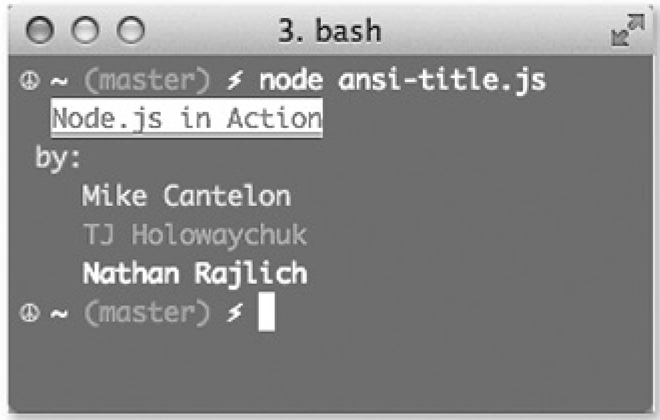



**Рис. 13.5.** Результат выполнения сценария ansi-title.js, который печатает название книги и имена авторов разными цветами

Код, окрашивающий выводимые данные в разные цвета, довольно многословен,

хотя и чрезвычайно прост. Каждый вызов функции непосредственно проецируется на соответствующий управляющий код, записанный в поток. Код, показанный в листинге 13.13, включает две строки инициализации, за которыми следует длинная цепочка вызовов функций, записывающих цветовые коды и строки в поток process.stdout.

**Листинг 13.13.** Простая программа, выводящая на печать разноцветные название книги и имена авторов

```
var ansi = require('ansi');
var cursor = ansi(process.stdout);


cursor
  .reset()
  .write(' ')
  .bold()
  .underline()
  .bg.white()
  .fg.black()
  .write('Node.js in Action')
  .fg.reset()
  .bg.reset()
  .resetUnderline()
  .resetBold()
  .write(' \n')
  .fg.green()
  .write(' by:\n')
  .fg.cyan()
  .write(' Mike Cantelon\n')
  .fg.magenta()
  .write(' TJ Holowaychuk\n')
  .fg.yellow()
  .write(' Nathan Rajlich\n')
  .reset()
```

Цветовые коды — только один из ключевых механизмов модуля ansi.js. Мы не стали касаться кодов, применяемых для позиционирования курсора, не узнали, как подать звуковой сигнал, как скрыть или показать курсор. Чтобы получить ответы на

эти вопросы, загляните в документацию к модулю ansi.js и примерам его применения.

### 13.5. Резюме

Платформа Node изначально предназначалась для разработки приложений, ориентированных на ввод-вывод данных, таких как HTTP-серверы. Однако, как вы узнали из этой главы, Node отлично подходит для решения самых разных задач, таких как создание интерфейса командной строки для вашего сервера приложений, клиентской программы, способной подключить вас к ASCII-серверу сериала «Звездные войны», или программы, которая извлекает и выводит на экран статистику с серверов фондового рынка, — возможности Node-приложений ограничены лишь вашим воображением. Обратите внимание на модули npm и node-gyp, представляющие собой два примера сложных программ командной строки, написанных с помощью Node. Эти примеры достойны изучения.

В этой главе мы поговорили о паре созданных Node-сообществом коммуникационных модулей, которые могли бы стать вашим ориентиром при разработке приложений. В следующей главе мы сосредоточимся на том, как находить такие восхитительные модули в Node-сообществе и как предоставлять собственные модули на суд сообщества с целью их совершенствования и получения отзывов от членов сообщества. Социальное общение — штука весьма увлекательная!

## Глава 14. Экосистема Node

- Поиск онлайновой помощи по Node
- Совместная разработка Node-приложений с помощью веб-сайта GitHub
- Публикация результатов работы с помощью диспетчера Node-пакетов

Чтобы добиться максимальных результатов при разработке Node-приложений, нужно знать, где получить помощь и как объединить ваши усилия с усилиями остальных членов Node-сообщества.

Как и в большинстве сообществ разработчиков приложений с открытым кодом, создание Node-приложений и связанных проектов выполняется путем взаимодействия через Интернет. Разработчики отсылают друг другу код, просматривают присланный код, создают документацию для проектов и отчеты по обнаруженным ошибкам. Завершив подготовку новой версии Node, разработчики

публикуют ее на официальном веб-сайте Node. Если появляется достойный внимания модуль от независимого производителя, он публикуется в npm-хранилище. К модулю, находящемуся в хранилище, могут получить доступ все желающие и затем установить его на своих компьютерах. Благодаря онлайновым ресурсам обеспечивается поддержка, требуемая для работы с Node и со связанными проектами.

На рис. 14.1 показано, как использовать онлайновые ресурсы для разработки, распространения и сопровождения связанных с Node программных продуктов.

Обычно потребность в технической поддержке возникает раньше, чем необходимость в совместной работе. Поэтому сначала давайте выясним, как получить необходимую помощь в Интернете.

**Рис. 14.1.** Node-проекты создаются при онлайновом взаимодействии, часто через веб-сайт GitHub. Затем эти проекты публикуются в npm-хранилище, а документирование и сопровождение осуществляются через онлайновые ресурсы

## 14.1. Онлайновые ресурсы Node-разработчика

Поскольку мир Node постоянно меняется, используйте Интернет для поиска актуальных ссылок на требуемые ресурсы. В вашем распоряжении находятся многочисленные веб-сайты, онлайновые дискуссионные группы и комнаты чата, где можно найти нужную информацию.

### 14.1.1. Ссылки, относящиеся к Node и модулям

В табл. 14.1 перечислены ссылки и онлайновые ресурсы, относящиеся к Node. Наиболее полезными сайтами, содержащими ссылки на API-интерфейсы платформы Node и рассказывающими о модулях от независимых производителей, являются соответственно сайт Node.js и домашняя страница npm-диспетчера.

Если вы собираетесь что-нибудь реализовать с помощью Node или любого из встроенных в Node модулей, домашняя страница платформы Node — это воистину бесценный ресурс. На сайте документировано все, что связано с Node, включая каждый из прикладных программных интерфейсов этой платформы (рис. 14.2). Вы всегда найдете на сайте информацию о последней версии Node. В официальном блоге также документируются последние достижения, касающиеся Node, и предлагаются всеобщему вниманию важные новости Node-сообщества. Там есть даже вакансии по трудоустройству.

**Таблица 14.1.** Полезные ссылки, имеющие отношение к Node.js

| Ресурс | URL-адрес |
|---|---|
| Домашняя страница Node.js | http://nodejs.org/ |
| Обновленная документация по ядру Node.js | http://nodejs.org/api/ |
| Блог, посвященный Node.js | http://blog.nodejs.org/ |
| Доска объявлений о работе, связанной с Node.js | http://jobs.nodejs.org/ |
| Домашняя страница диспетчера Node-пакетов (npm) | http://npmjs.org/ |

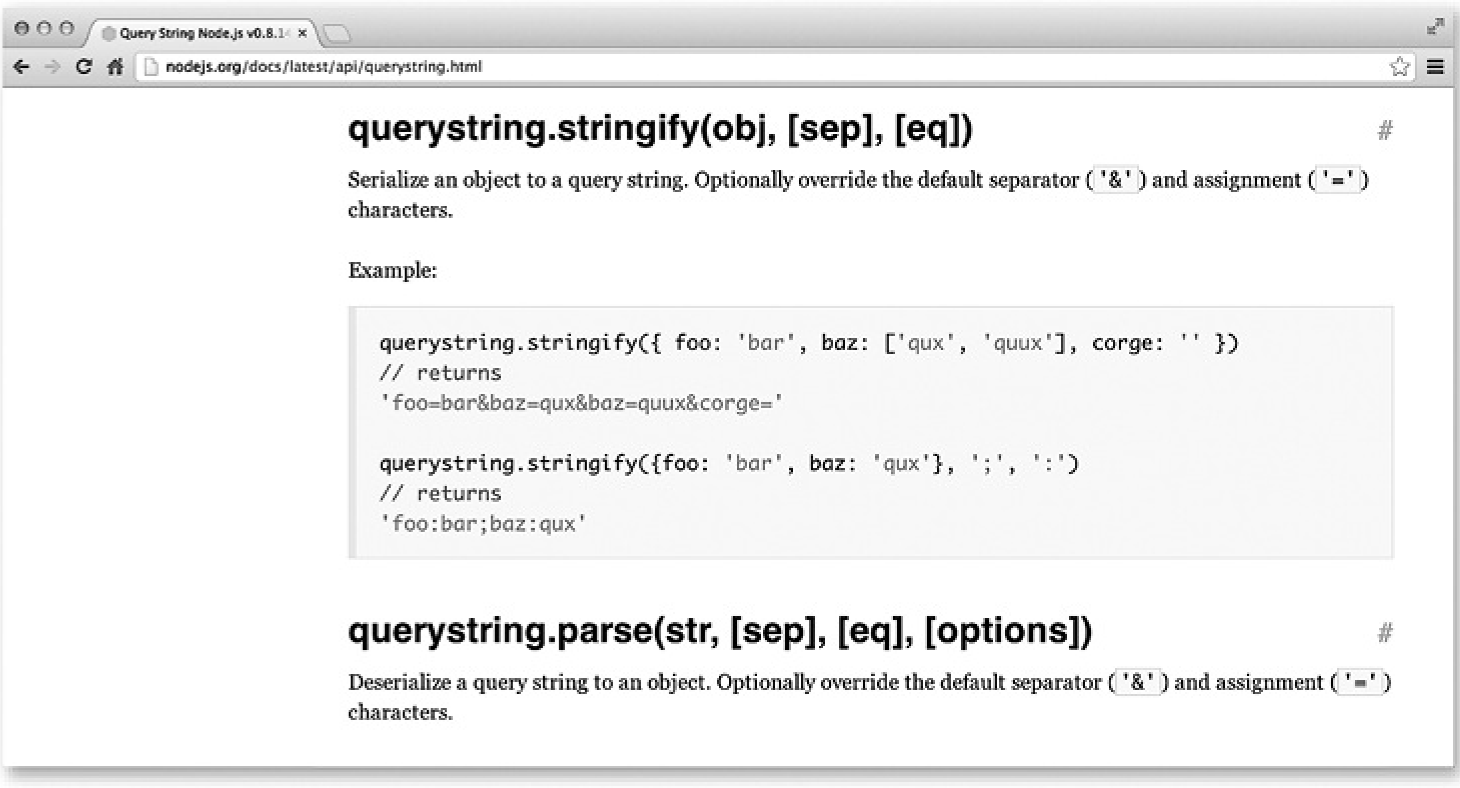


**Рис. 14.2.** Помимо ссылок на полезные ресурсы, связанные с Node, на веб-сайте nodejs.org имеется авторитетная документация по каждой вышедшей версии Node

Если вы собираетесь купить модуль от независимого производителя, обратитесь к странице поиска npm-хранилища. С помощью ключевых слов можно найти нужный модуль среди тысяч доступных модулей. Если вы нашли модуль, который вас заинтересовал, щелкните на его названии для вывода страницы сведений. На этой странице можно найти ссылки на домашнюю страницу проекта модуля (при наличии таковой), а также другую полезную информацию, например сведения о npm-пакетах, которые зависят от данного модуля, описание зависимостей самого модуля, список версий Node, с которыми совместим модуль, лицензионную информацию.

Однако при всем при том эти веб-сайты могут не дать ответы на все вопросы, связанные с использованием Node или модулей от независимых производителей, поэтому давайте поговорим о других великолепных местах, где можно получить онлайновую помощь.

### 14.1.2. Google Groups

На форуме Google Groups созданы группы дискуссий, в которых обсуждаются вопросы, связанные с Node и некоторыми другими популярными модулями от независимых производителей, в том числе npm, Express, node-mongodb-native и Mongoose.

В Google Groups можно задавать как общие, так и конкретные вопросы. Например, если у вас возникают проблемы при удалении MongoDB-документов с помощью модуля nodemongodb-native, на форуме Google Groups перейдите в группу node-mongodb-native (https://groups.google.com/forum/?fromgroups#!forum/node-mongodb-native) и найдите сообщение от пользователя, у которого возникла аналогичная проблема. Если подобная проблема ни у кого еще не возникала, присоединитесь к этой группе и отправьте свой вопрос. Можно создавать длинные сообщения, чтобы задавать сложные вопросы, требующие подробной детализации.

Не существует единого централизованного списка, включающего описание всех связанных с Node групп в Google Groups. Эти группы могут упоминаться в документации проекта, но обычно их приходится искать в Интернете. Например, в поле поиска Google можно ввести строку nameofsomemodule node.js google group, чтобы проверить наличие группы, в которой обсуждается соответствующий модуль от независимого производителя.

Недостаток, связанный с форумом Google Groups, заключается в том, что ответа можно ожидать часами и днями, в зависимости от параметров выбранной группы.

Если же у вас простой вопрос, требующий незамедлительного ответа, задайте его в комнате тематического чата.

### 14.1.3. IRC

Ретранслируемый интернет-чат (Internet Relay Chat, IRC) появился в далеком 1988 году. Хотя некоторые пользователи считают IRC-чат устаревшим, он по-прежнему существует и активно используется. Более того, именно IRC остается лучшим онлайновым средством для быстрого получения ответов на вопросы, которые касаются программ с открытым исходным кодом. Комнаты IRC-чата называются *каналами* (channels). Существуют отдельные каналы для Node и различных модулей от независимых производителей. Вряд ли вы найдете список IRC-каналов, связанных с Node, но если для модуля от независимого производителя подобный канал существует, скорее всего, соответствующее упоминание будет в документации по этому модулю.

Чтобы получить ответ на вопрос, заданный в IRC, подключитесь к IRC-сети (http://chatzilla.hacksrus.com/faq/#connect), выберите соответствующий канал и отправьте вопрос в этот канал. Проявите уважение к подписчикам канала и предварительно попробуйте самостоятельно поискать ответ. Если таким образом ответ найти не удалось, задавайте свой вопрос.

Если вы раньше не использовали IRC-чат, подключитесь к нему с помощью веб-клиента. Для сети ITC Freenode, в которой находится большинство IRC-каналов связанных с Node, существует веб-клиент, доступный на веб-сайте http://webchat.freenode.net/. Чтобы подключиться к каналу, введите соответствующее имя в форме подключения. При этом вам не придется регистрироваться, к тому же можно использовать любой псевдоним, выбранный на ваше усмотрение. Если выбранный вами псевдоним уже кем-то занят, в конце автоматически появится символ подчеркивания (_), который сделает ваш псевдоним уникальным.

После щелчка на кнопке Connect (Подключиться) вы подключитесь к каналу и окажетесь вместе с другими пользователями в одной комнате, название которой будет находиться на правой боковой панели.

### 14.1.4. Очередь проблем на сайте GitHub

Если разработка проекта происходит с помощью веб-сайта GitHub, для получения ответа на возникшие вопросы можно воспользоваться очередью проблем (issue queue) сайта GitHub. Чтобы получить доступ к очереди проблем, перейдите на главную страницу проекта GitHub и щелкните на вкладке Issues (Проблемы). С

помощью поля поиска можно попытаться найти описание ошибки, которое поможет решить возникшую у вас проблему. Пример очереди проблем показан на рис. 14.3.

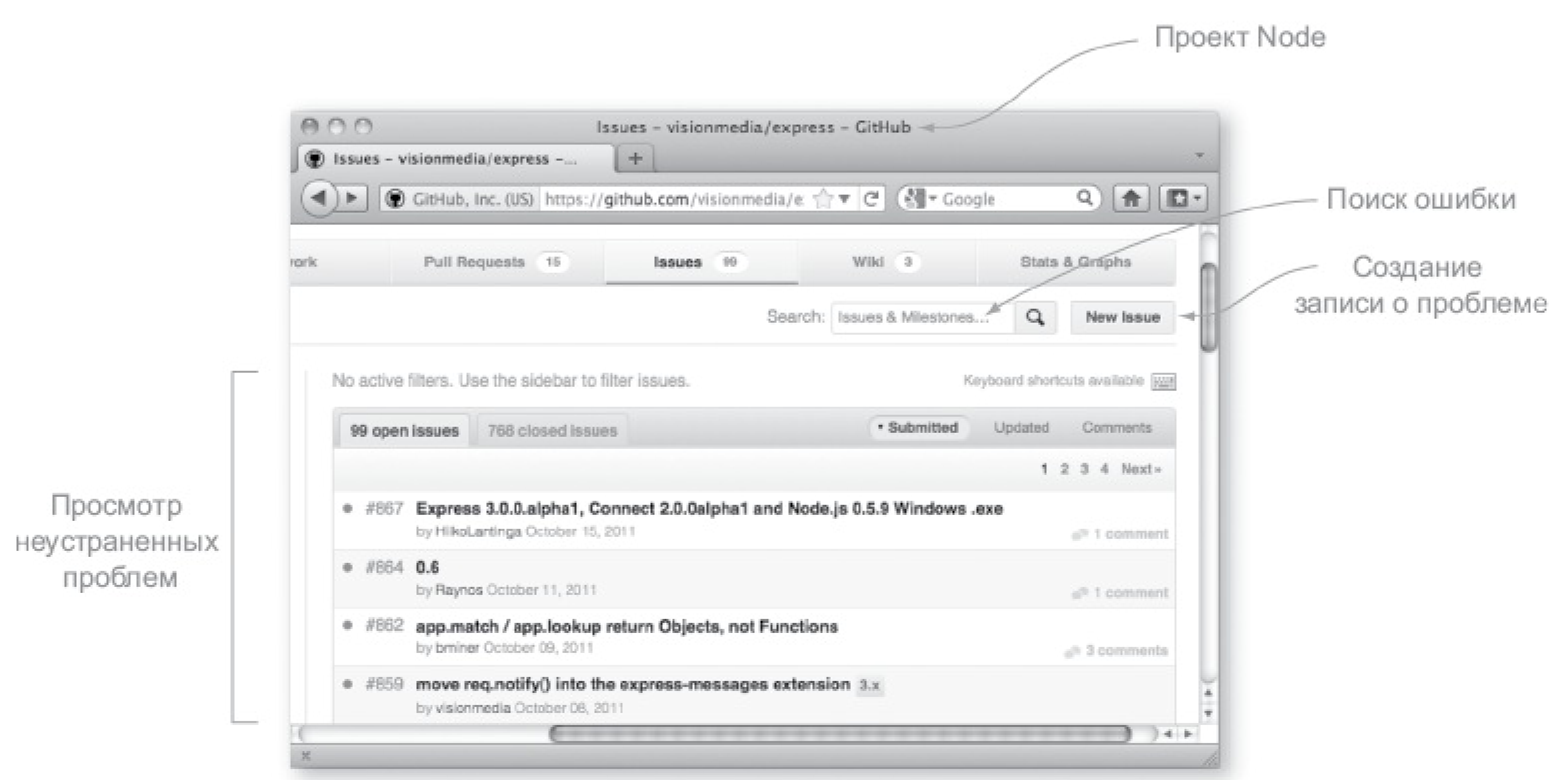



**Рис. 14.3.** Если вы подозреваете о наличии ошибки в коде проекта, воспользуйтесь очередью проблем на веб-сайте GitHub

Если вы не можете найти описание своей проблемы, но уверены, что ошибка кроется в коде, щелкните на кнопке New Issue (Новая проблема), находящейся на странице ошибок, и опишите возникшую у вас проблему. Когда вы создадите описание ошибки, специалисты, ответственные за поддержку проекта, смогут ответить на ваш вопрос на странице проблем. Они либо помогут вам устранить ошибку, либо зададут уточняющие вопросы, которые помогут им найти решение вашей проблемы.

**Не путайте систему отслеживания ошибок с форумом поддержки**

При выполнении некоторых проектов считается недопустимым задавать общие вопросы технического плана, используя систему отслеживания ошибок сайта GitHub. Для этого есть другие проекты, предлагающие общую техническую поддержку, например Google Group. Просмотрите файл README проекта, в котором указан желательная тематика вопросов, относящихся к проекту.

Теперь, когда вы получили представление о том, где получить ответы на вопросы, относящиеся к проекту, поговорим о роли GitHub, не связанной с оказанием технической поддержки. Рассмотрим использование веб-сайта GitHub для организации совместной работы над Node-проектами.

## 14.2. GitHub

Веб-сайт GitHub является «центром притяжения» мира программного обеспечения с открытым исходным кодом и критически важным инструментом разработчиков Node-приложений. GitHub-сервис предоставляет хостинг для Git, мощную систему контроля версий (Version Control System, VCS) и удобный веб-интерфейс для просмотра Git-хранилищ. Для проектов с открытым исходным кодом GitHub-сервис бесплатен.

### GIT

Git пользуется наибольшей популярностью среди разработчиков проектов с открытым исходным кодом. Для Git существует также распределенная система контроля версий (Distributed Version Control System, DVCS), которую, в отличие от Subversion и многих других систем контроля версий, можно использовать без сетевого подключения к серверу. Система Git появилась в 2005 году, причем ее появление было инспирировано патентованной системой контроля версий BitKeeper. Издатель BitKeeper предоставил группе разработчиков ядра Linux права на свободное использование этой программы, но позже отозвал их в силу возникшего подозрения, что группа разработчиков пытается выяснить, как работают внутренние механизмы BitKeeper. После этого Линус Торвальдс (Linus Torvalds), разработчик Linux, решил создать альтернативную систему контроля версий со сходной функциональностью, и уже через несколько месяцев система Git использовалась командой разработчиков ядра Linux.

Помимо хостинга Git, GitHub поддерживает проекты, предлагающие функциональность отслеживания ошибок, вики и хостинга веб-страниц. Поскольку большинство Node-проектов, находящихся в npm-хранилище, имеют поддержку на GitHub, знакомство с GitHub будет полезным для разработчиков Node-приложений поскольку GitHub предлагает удобные средства просмотра кода, поиска неустраненных ошибок, а в случае необходимости обеспечивает внесение правок и дополнение документации.

Еще одно применение GitHub заключается в *отслеживании* (watch) проекта. При этом вы будете получать уведомления в случае внесения каких-либо изменений в проект. Зачастую количество людей, отслеживающих проект, является мерилом популярности этого проекта.

Хотя возможности GitHub велики, как их использовать? Давайте углубимся в эту тему в следующем разделе.

### 14.2.1. Приступаем к работе с GitHub

Если вас посетит некая идея, касающаяся Node-проекта или модуля от независимого производителя, то, чтобы упростить себе доступ к средствам хостинга на сайте GitHub, вы, скорее всего, создадите там учетную запись, естественно, если она там еще не создана. После этого вы сможете добавлять туда свои проекты, как описано в следующем разделе.

Поскольку GitHub требует использования системы Git, вам придется сначала сконфигурировать эту систему, а затем перейти к работе с GitHub. Облегчить настройку GitHub помогут страницы помощи, предлагаемые для платформ Macintosh, Windows и Linux (https://help.github.com/articles/set-up-git). После завершения конфигурирования Git нужно настроить GitHub, зарегистрировавшись на этом сайте и предоставив открытый ключ защищенной оболочки (Secure SHell, SSH). SSH-ключ нужен для безопасного взаимодействия с GitHub.

Детали, относящиеся к каждому из описанных действий, рассматриваются в следующем разделе. Обратите внимание, что эти действия нужно выполнить только один раз, а не при каждом добавлении проекта в GitHub.

#### *Конфигурирование Git и регистрация на сайте GitHub*

Для работы с GitHub нужно сконфигурировать систему Git. При этом потребуется указать ваши имя и адрес электронной почты, воспользовавшись следующими двумя командами:

**git config —global user.name "Bob Dobbs"**

git config —global user.email subgenius@example.com

Затем нужно зарегистрироваться на сайте GitHub. Перейдите на страницу регистрации (https://github.com/signup/free), заполните поля и щелкните на кнопке Create an Account (Создать учетную запись).

#### *Добавление открытого SSH-ключа на сайте GitHub*

После регистрации нужно предоставить сайту GitHub открытый SSH-ключ

(https://help.github.com/articles/generating-ssh-keys). Этот ключ потребуется для аутентификации ваших Git-транзакций. Чтобы добавить ключ, выполните следующие действия:

1. В окне браузера введите адрес https://github.com/settings/ssh.

2. Щелкните на кнопке Add SSH Key (Добавить SSH-ключ).

Дальнейшие действия зависят от используемой операционной системы. GitHub идентифицирует вашу операционную систему и выведет на экран соответствующие инструкции.

### 14.2.2. Добавление проекта на сайт GitHub

После завершения настройки GitHub можете добавить проект к своей учетной записи и начать наполнять его контентом.

Сначала нужно создать GitHub-хранилище для вашего проекта, о чем мы скоро расскажем. Затем нужно создать Git-хранилище для локальной рабочей станции, с которым мы будем работать, прежде чем перейдем к GitHub-хранилищу. Описанный процесс проиллюстрирован на рис. 14.4.

**Рис. 14.4.** Действия, предпринимаемые для добавления Node-проекта на сайт GitHub

Для просмотра файлов проекта можно воспользоваться веб-интерфейсом сайта

GitHub.

### *Создание GitHub-хранилища*

Чтобы создать GitHub-хранилище, нужно пройти следующие этапы.

1. С помощью браузера выполнить процедуру входа на сайт github.com.
2. Перейти по ссылке https://github.com/new.
3. Заполнить поля предоставленной формы, описывающей хранилище, и щелкнуть на кнопке Create Repository (Создать хранилище).
4. GitHub создает пустое Git-хранилище и очередь ошибок для вашего проекта.
5. GitHub выводит на экран пошаговую инструкцию, описывающую действия, которые нужно предпринять в Git для передачи кода в GitHub.

Поскольку полезно понимать, что происходит на каждом этапе, мы пройдем все этапы и на примере продемонстрируем основные приемы работы с Git.

### *Создание пустого Git-хранилища*

Чтобы добавить учебный проект в GitHub, сначала нужно создать учебный Node-модуль. В рамках данного примера создадим модуль, включающий некую программную логику сокращения URL-адресов, и назовем его node-elf.

В первую очередь нужно создать временную папку для проекта с помощью следующей команды:

**mkdir -p ~/tmp/node-elf**

cd ~/tmp/node-elf

Чтобы использовать эту папку в качестве Git-хранилища, введите такую команду:

git init

В результате будет создана папка .git, предназначенная для хранения метаданных.

### *Добавление файлов в Git-хранилище*

После создания пустого хранилища нужно заполнить его файлами. В рамках

текущего примера добавим файл, включающий программную логику сокращения URL-адресов. Сохраните код из листинга 14.1 в файле index.js, а файл — в той же папке .git.

**Листинг 14.1.** Node-модуль для сокращения URL-адресов

```
// Функция инициализации вызывается неявно методами shorten() и expanc
exports.initPathData = function(pathData) {
  pathData = (pathData) ? pathData : {};
  pathData.count = (pathData.count) ? pathData.count : 0;
  pathData.map = (pathData.map) ? pathData.map : {};
}

// Принимаем строку "path" и возвращаем сокращенный
// URL-адрес, полученный путем проецирования
exports.shorten = function(pathData, path) {
  exports.initPathData(pathData);
  pathData.count++;
  pathData.map[pathData.count] = path;
  return pathData.count.toString(36);
}

// Принимаем предварительно сокращенный URL-адрес
// и возвращаем расширенный URL-адрес
exports.expand = function(pathData, shortened) {
  exports.initPathData(pathData);
  var pathIndex = parseInt(shortened, 36);
  return pathData.map[pathIndex];
}
```

Теперь нужно сообщить системе Git о том, что вы хотите поместить файл в хранилище. Git-команда add работает иначе, чем в других системах контроля версий. Вместо добавления файлов в Git-хранилище команда add добавляет файлы в так называемую *зону подготовки* (staging area). Эту зону можно представить в виде контрольного списка, в котором указываются только что добавленные файлы или файлы, которые вы изменили и хотели бы включить в хранилище при его очередном обновлении:

```
git add index.js
```

Теперь система Git «знает», что данный файл нужно отслеживать. При желании в зону подготовки можно добавлять другие файлы, но на данный момент достаточно одного.

Чтобы сообщить Git о вашем желании обновить хранилище, с помощью команды commit включите в него измененные файлы, выбранные вами в зоне подготовки. Команду commit нужно выполнять с флагом −m, чтобы на экран было выведено сообщение с описанием изменений, происшедших после обновления хранилища:

```
git commit -m "Added URL shortening functionality."
```

После этого хранилище на вашей рабочей станции будет обновлено. Чтобы просмотреть список изменений, введите следующую команду:

```
git log
```

### *Передача данных из Git в GitHub*

Если сейчас в вашу рабочую станцию вдруг ударит молния, вы потеряете результаты всей работы. Чтобы застраховаться от неожиданностей и насладиться всеми преимуществами веб-интерфейса GitHub, все изменения, сделанные вами в Git-хранилище, нужно передать в учетную запись на сайте GitHub. Однако прежде, чем это сделать, нужно сообщить Git о том, куда отправлять изменения. А для этого нужно сначала добавить удаленное Git-хранилище. Подобные хранилища обычно называют *удаленными узлами* (remotes).

С помощью следующей строки кода можно добавить удаленный GitHub-узел в ваше хранилище. Вместо параметра username подставьте свое имя пользователя. Также обратите внимание, что имя проекта задается с помощью параметра node-elf.git:

```
git remote add origin git@github.com:username/node-elf.git
```

После добавления удаленного узла можно отправлять изменения в GitHub. В терминологии Git такая отправка называется *продвижением* (push). Следующая команда указывает Git на необходимость продвижения вашей работы в удаленный узел *исходного кода* (origin), определенный с помощью предыдущей команды. Каждое Git-хранилище может иметь одну или несколько ветвей, применяемых для концептуального разделения рабочих областей в хранилище. Результаты работы можно продвинуть в *главную* (master) ветвь:

```
git push -u origin master
```

Параметр −u команды push указывает Git на то, что данный удаленный узел является одновременно *восходящим* (upstream) узлом и ветвью. Восходящий узел является удаленным узлом, используемым по умолчанию.

После первого продвижения данных с помощью параметра −u можно продолжить продвижение, используя следующую простую для запоминания команду:

git push

Если теперь перейти на сайт GitHub и обновить страницу хранилища, вы должны увидеть свой файл. Создание модуля и его хостинг на сайте GitHub — это быстрый, но не слишком чистый способ обеспечить его многократное использование. Например, с помощью следующих команд можно задействовать наш учебный модуль в проекте:

**mkdir ~/tmp/my_project/node_modules**

cd ~/tmp/my_project/node_modules

git clone https://github.com/mcantelon/node-elf.git elf

cd ..

Команда require('elf') обеспечивает доступ к модулю. Обратите внимание, что при клонировании хранилища используется последний аргумент командной строки, задающий имя папки, в которую будет клонировано хранилище.

Теперь вы знаете, как добавлять проекты на сайт GitHub, в том числе как создать хранилище на сайте GitHub, как создать Git-хранилище на своей рабочей станции и добавить туда файлы, как продвинуть Git-хранилище со своей рабочей станции на сайт GitHub. Дополнительную информацию по всем перечисленным темам можно найти на многочисленных онлайновых ресурсах. Если вы ищете исчерпывающую информацию по работе с Git, обратитесь к книге Скотта Чакона (Scott Chacon), одного из основателей проекта GitHub, «Pro Git». Эту книгу можно купить или получить доступ к бесплатной электронной копии (http://progit.org/). Если же вам больше нравится изучать новый материал на практических примерах, обратитесь к странице документации официального веб-сайта Git. На этой странице можно найти множество готовых примеров (http://git-scm.com/documentation).

### 14.2.3. Совместная работа с помощью GitHub

Теперь, когда вы знаете, как создать GitHub-хранилище «с нуля», давайте выясним, как с помощью GitHub организовать совместную работу.

Предположим, что при использовании модуля от независимого производителя происходит ошибка. В этом случае можно проверить исходный код модуля и определить способ устранения ошибки. Также можно отправить электронное сообщение автору кода с описанием вашего варианта исправления и вложением с исправлениями. Без сайта GitHub автору кода пришлось бы выполнить довольно

утомительную работу, сравнивая файлы с последней версией программы и внося в код присланные вами исправления. Если же автор программы пользуется сайтом GitHub, вы можете *клонировать* (clone) хранилище проектов автора, внести соответствующие изменения, а затем проинформировать автора об исправленной ошибке с помощью того же сайта GitHub. После этого GitHub представит автору кода веб-страницу, на которой будут видны различия между его кодом и клонированной версией, и если он согласится с исправлением, оно войдет в последнюю версию кода после единственного щелчка мышью.

На языке GitHub дублирование хранилища называется *ветвлением* (forking). После ветвления проекта с копией можно делать все что угодно, на оригинальном хранилище это никак не скажется. Для ветвления никакого разрешения брать у автора исходного проекта не требуется. Любой пользователь может выполнить ветвление произвольного проекта и отправить результаты своей работы обратно в исходный проект. Автор исходного проекта может не одобрить ваш вклад, но даже в этом случае вам не возбраняется оставить исправленную вами версию, которую вы сможете развивать и поддерживать независимо от исходного проекта. Если ваш вариант проекта завоюет популярность, другие пользователи смогут выполнять ветвление уже вашей ветви, чтобы внести свой вклад в развитие проекта.

Когда вы внесете изменения в ветвь, можете сообщить автору исходного проекта об изменении с помощью *запроса на получение* (pull request), представляющего собой сообщение, в котором автору хранилища предлагается *получить* (pull) сделанные вами изменения. На языке Git *получение* (pulling) изменений означает импорт результатов работы из ветви и объединение этой работы с вашей. На рис. 14.5 продемонстрирован сценарий совместной работы на сайте GitHub.

**Рис. 14.5.** Типичный сценарий GitHub-разработки

Рассмотрим пример ветвления GitHub-хранилища, выполняемого в рамках совместной работы. Этот процесс показан на рис. 14.6.

**Рис. 14.6.** Процесс совместной работы на сайте GitHub, реализуемый путем ветвления

Ветвление инициирует процесс совместной работы путем дублирования GitHub-хранилища в вашу учетную запись (A). Затем ветвь хранилища клонируется на рабочую станцию (B), после чего в него вносятся изменения, которые одобряются

(C). Далее результаты вашей работы продвигаются обратно в GitHub (D), где все завершается отправкой владельцу исходного хранилища запроса на получение изменений, в котором ему предлагается изучить изменения (E). Если он решит включить предлагаемые вами изменения в свое хранилище, он одобрит ваш запрос на получение вашей ветви хранилища.

Предположим, вам нужно создать ветвь хранилища node-elf, созданного в этой главе ранее, и добавить код, который экспортирует версию модуля. Это бы позволило каждому пользователю вашего модуля быть уверенным в том, что он работает с нужной версией.

Для начала войдите на сайт GitHub и перейдите на главную страницу хранилища по адресу https://github.com/mcantelon/node-elf, где щелкните на кнопке Fork (Ветвь), чтобы продублировать хранилище. Полученная страница ничем не будет отличаться от страницы исходного хранилища, но под названием хранилища появится примерно такой текст: «forked from mcantelon/node-elf» (создание ветви mcantelon/node-elf).

После создание ветви нужно клонировать хранилище на рабочую станцию, подготовить изменения и продвинуть их на сайт GitHub. Следующие команды реализуют описанный сценарий для хранилища node-elf:

```
mkdir -p ~/tmp/forktest
cd ~/tmp/forktest
git clone git@github.com:chickentown/node-elf.git
cd node-elf
echo "exports.version = '0.0.2';" >> index.js
git add index.js
git commit -m "Added specification of module version."
git push origin master
```

После продвижения изменений на сайт GitHub щелкните на кнопке Pull Request (Запрос на получение), находящейся на странице ветви хранилища, после чего введите тему и содержимое сообщения, описывающего изменения, и щелкните на кнопке Send Pull Request (Отправить запрос на получение). На рис. 14.7 показан экранный снимок типичного сообщения.

**Рис. 14.7.** Детали GitHub-запроса на получение

Запрос на получение добавляется в очередь проблем исходного хранилища. Затем владелец исходного хранилища после просмотра предлагаемых вами изменений может принять их, щелкнув на кнопке Merge Pull Request (Объединить запрос на получение), ввести подтверждающее сообщение и щелкнуть на кнопке Confirm Merge (Подтвердить объединение). После этого проблемная ветвь автоматически закрывается.

После создания вместе с другими пользователями самого прекрасного в мире модуля нужно представить его миру. Наилучший способ для этого — добавить его в npm-хранилище.

## 14.3. Пополнение npm-хранилища

Предположим, вы разработали модуль, создающий сокращенные версии URL-адресов, и считаете, что он мог бы быть полезным другим пользователям платформы Node. Чтобы открыть доступ к этому модулю для всех заинтересованных лиц, отправьте соответствующие сообщение, описывающее возможности своего модуля, в те группы форума Google Groups, которые имеют отношение к Node. Однако в этом случае вы сможете охватить лишь ограниченное число пользователей. К тому же при таком подходе невозможно сообщить пользователям вашего модуля о появлении обновлений и изменений.

Чтобы решить проблему доступности и обновлений, опубликуйте ваш модуль в npm-хранилище. С помощью npm-хранилища можно легко определять зависимости проекта, поскольку они автоматически устанавливаются вместе с модулем. Если вы создали модуль, предназначенный для хранения комментариев, относящихся к контенту (например, постов блога), то в качестве зависимости можно было бы иметь модуль, обрабатывающий хранящиеся в MongoDB-хранилище данные комментариев. Либо модуль, который поддерживает инструмент командной строки, в качестве зависимости мог бы иметь вспомогательный модуль, предназначенный для синтаксического разбора аргументов командной строки.

До сих пор мы использовали npm-хранилище для установки самых разных программ, от сред тестирования до драйверов баз данных, но ничего еще там не публиковали. В следующих разделах мы рассмотрим этапы публикации созданных вами программ в npm-хранилище.

1. Подготовка пакета.

2. Написание спецификации пакета.

3. Тестирование пакета.

4. Публикация пакета.

Начнем с этапа подготовки пакета.

### 14.3.1. Подготовка пакета

Любой Node-модуль, который вы собираетесь представить на суд общественности, должен сопровождаться связанными ресурсами, такими как документация, примеры, тесты и соответствующие утилиты командной строки. В комплект поставки модуля должен также входить файл README с информацией необходимой для быстрого освоения модуля.

Папка, в которой находится пакет, должна иметь ряд вложенных папок. В табл. 14.2 перечислены типичные вложенные папки (bin, docs, example, lib и test), которые могут применяться для размещения файлов проекта.

**Таблица 14.2.** Обычные папки Node-проекта

| Вложенная папка | Описание |
|---|---|
| bin | Сценарии командной строки |
| docs | Документация |
| example | Примеры использования приложения |
| lib | Основные функциональности приложения |
| test | Сценарии тестов и связанные ресурсы |

После размещения файлов проекта во вложенных папках нужно подготовить его для публикации в npm-хранилище, написав спецификацию пакета.

### 14.3.2. Написание спецификации пакета

При публикации пакета в npm-хранилище нужно включить в пакет файл спецификации, который сможет распознать компьютер. Этот файл в формате JSON называется package.json и содержит информацию о модуле, такую как имя, описание, версия, зависимости и другие характеристики. На веб-сайте Nodejitsu можно найти пример файла package.json вместе с описанием назначения каждой части файла. Это описание появляется после наведения указателя мыши на соответствующую часть контента (http://package.json.nodejitsu.com/).

В файле package.json обязательными являются только имя и номер версии. Другие характеристики не обязательны, но если они будут определены, модуль

станет более полезным. Например, указав характеристику bin, вы тем самым дадите npm знать о том, какие файлы в пакете являются инструментами командной строки, после чего npm сделает их глобально доступными.

Пример спецификации может выглядеть следующим образом:

```
{
  "name": "elf"
  , "version": "0.0.1"
  , "description": "Toy URL shortener"
  , "author": "Mike Cantelon <mcantelon@example.com>"
  , "main": "index"
  , "engines": { "node": "0.4.x" }
}
```

Чтобы получить исчерпывающее описание доступных параметров файла package.json, введите следующую команду:

```
npm help json
```

Поскольку создание JSON-файла вручную не намного проще написания XML кода, имеет смысл воспользоваться специальными инструментами, облегчающими эту задачу. Один из подобных инструментов, ngen, представляет собой npm-пакет, после установки которого добавляется инструмент командной строки ngen. После ответа на ряд вопросов ngen сгенерирует файл package.json. При этом также генерируются другие файлы, которые обычно входят в npm-пакеты, такие как файл Readme.md.

Чтобы установить ngen, выполните следующую команду:

```
npm install -g ngen
```

После установки ngen вы сможете получить доступ к глобальной команде ngen, которая при запуске в корневом каталоге проекта выведет на экран вопросы, имеющие отношение к проекту. После ответа на эти вопросы генерируется файл .json пакета, а также другие файлы, которые обычно включаются в Node-проекты. Некоторые сгенерированные файлы просто не нужны, поэтому вы можете смело их удалить. Среди сгенерированных файлов имеется файл .gitignore, определяющий множество файлов и каталогов, которые обычно не следует добавлять в Git-хранилище проекта, публикуемого в npm-хранилище. Также создается файл .npmignore со схожими функциями. С его помощью npm получает сведения о файлах, которые могут игнорироваться при публикации пакета в npm-хранилище.

Вот пример выводимых после выполнения команды ngen данных:

**Project name: elf**

Enter your name: Mike Cantelon

```
Enter your email: mcantelon@gmail.com
Project description: URL shortening library

create : /Users/mike/programming/js/shorten/node_modules/.gitignore
create : /Users/mike/programming/js/shorten/node_modules/.npmignore
create : /Users/mike/programming/js/shorten/node_modules/History.md
create : /Users/mike/programming/js/shorten/node_modules/index.js
...
```

Генерирование файла package.json — это сложнейший этап процесса публикации в npm-хранилище. По окончании этого этапа вы будете готовы к публикации модуля.

### 14.3.3. Тестирование и публикация пакета

Публикация модуля в npm-хранилище выполняется в три этапа, которые мы рассмотрим в этом разделе.

1. Локальное тестирование установленной копии пакета.
2. Добавление npm-пользователя, если он еще не добавлен.
3. Публикация пакета в npm-хранилище.

#### *Тестирование установленной копии пакета*

Чтобы выполнить локальное тестирование пакета, воспользуйтесь npm-командой link, находясь в корневом каталоге модуля. В результате выполнения этой команды пакет станет глобально доступным на рабочей станции, и Node сможет использовать его подобно пакетам, устанавливаемым обычным образом из npm-хранилища.

```
sudo npm link
```

Теперь, чтобы глобально связать ваш проект, можете установить его в отдельную тестовую папку с помощью команды link, указав после команды название пакета:

```
npm link elf
```

После установки пакета проведите быстрый тест требований модуля, выполнив функцию require в REPL-сеансе, как показано в следующем примере кода. Это позволит вам увидеть переменные или функции, которые предоставляет ваш модуль:

```
node
> require('elf');
{ version: '0.0.1',
  initPathData: [Function],
  shorten: [Function],
  expand: [Function] }
```

После того как пакет пройдет процедуру тестирования и вы завершите его развертывание, воспользуйтесь npm-командой unlink, находясь в корневой папке модуля:

```
sudo npm unlink
```

После этого модуль больше не будет глобально доступным на рабочей станции, но позднее, после завершения публикации модуля в npm-хранилище, можно будет выполнить его стандартную установку с помощью команды install.

По завершении тестирования npm-пакета на следующем этапе нужно создать учетную запись публикации, если вы еще этого не сделали.

### *Добавление пользователя npm-хранилища*

Чтобы создать собственную учетную запись для публикации в npm-хранилище, выполните следующую команду:

```
npm adduser
```

Вам будет предложено ввести имя пользователя, адрес электронной почты и пароль. Если добавление учетной записи пройдет успешно, сообщения об ошибке вы не увидите.

### *Публикация в npm-хранилище*

Следующий этап — публикация. Чтобы опубликовать пакет в npm-хранилище, введите такую команду:

```
npm publish
```

На экране может появиться предупреждение «Sending authorization over an insecure channel» («Выполнение аутентификации через незащищенный канал»), но если при этом дополнительные сообщения об ошибках не появятся, значит, публикация прошла успешно. Чтобы удостовериться в успешной публикации пакета, воспользуйтесь npm-командой view:

```
npm view elf description
```

Не возбраняется включить в хранилище одно или несколько частных хранилищ

в качестве зависимостей npm-пакета. Такая ситуация может возникнуть, если в вашем распоряжении имеется модуль с полезными вспомогательными функциями, которые вы хотели бы использовать, но этот модуль не опубликован в npm-хранилище.

При добавлении частной зависимости можно указать любое имя, которое отличается от имен других зависимостей, а там, где обычно помещается номер версии, будет находиться URL-адрес Git-хранилища. В следующем примере приводится фрагмент файла package.json, в котором последняя зависимость представляет собой частное хранилище:

```
"dependencies" : {
  "optimist" : ">=0.1.3",
  "iniparser" : ">=1.0.1",
  "mingy": ">=0.1.2",
  "elf": "git://github.com/mcantelon/node-elf.git"
},
```

Отметьте, что любой частный модуль также должен содержать файл package.json. Чтобы обезопасить себя от случайной публикации такого модуля, присвойте свойству private в файле package.json значение true:

```
"private": true,
```

Теперь вы полностью оснащены для установки, тестирования и публикации собственных модулей в npm-хранилище.

## 14.4. Резюме

Как и большинство успешных проектов с открытым исходным кодом, Node имеет собственное онлайновое сообщество, где вы сможете найти массу доступных онлайновых ресурсов и быстро получить ответы на вопросы с помощью онлайновых ссылок, форума Google Groups, IRC-чата или очереди проблем сайта GitHub.

Помимо системы отслеживания ошибок в проектах, GitHub предлагает Git-хостинг и механизм просмотра кода в Git-хранилище с помощью веб-браузера. Используя GitHub, другие разработчики могут легко разветвлять проект с открытым исходным кодом, внося свой вклад в устранение ошибок, добавление программ или придание проекту нового дыхания. Можно также легко отправлять внесенные в ветвь изменения обратно в исходное хранилище.

После того как Node-проект достигает такой степени готовности, когда его пора представлять остальному миру, проект можно отправить в хранилище диспетчера Node-пакетов (Node Package Manager, npm). Проект после включения в npm-

хранилище будет проще найти другим пользователям, а если он представляет собой модуль, его будет проще установить.

Теперь вы знаете, как при необходимости найти помощь, работать совместно с другими и вносить свой вклад в общее дело. Платформа Node возникла и стала популярной благодаря активной деятельности сообщества энтузиастов. Будьте активны и станьте частью Node-сообщества!

## Установка Node и дополнительных модулей

Платформа Node без особого труда устанавливается в большинстве операционных систем. Для этого можно воспользоваться обычной программой установки или командной строкой. Устанавливать с помощью командной строки проще в OS X и Linux, для Windows этот вариант не рекомендуется.

В следующих разделах рассматривается процесс установки Node в операционных системах OS X, Windows и Linux. В последнем разделе приложения рассказывается о поиске и установке полезных надстроек с помощью диспетчера Node-пакетов (Node Package Manager, npm).

### А.1. Установка в OS X

Процесс установки Node в OS X достаточно прост. Для установки предварительно скомпилированной версии Node и npm воспользуйтесь официальной программой установки (http://nodejs.org/#download), экран которой показан на рис. А.1.

Если вы предпочитаете выполнять установку из исходного кода, воспользуйтесь инструментом Homebrew (http://mxcl.github.com/homebrew/), который автоматизирует такую установку, или проведите установку из исходного кода вручную. Однако для этого у вас на машине уже должен быть установлен набор инструментов разработчика Xcode.

Независимо от выбранного метода установки вам понадобится вводить команды с помощью интерфейса командной строки OS X, доступ к которому обеспечивает приложение Terminal. Это приложение обычно находится в папке Utilities (Утилиты) основной папки Applications (Приложения).

Если вы планируете выполнять компиляцию исходного кода, обратитесь к п.А.4 за дополнительными сведениями.

**Рис. А.1.** Официальная программа установки Node для OS X

**XCODE**

Если набор Xcode у вас еще не установлен, загрузите его с веб-сайта Apple (http://developer.apple.com/downloads/). Вы можете совершенно бесплатно зарегистрироваться в Apple в качестве разработчика, чтобы получить доступ к странице загрузки. При полной установке Xcode занимает примерно 4 Гбайт. В качестве альтернативы Apple предлагает пакет Command Line Tools for Xcode, который доступен для загрузки на той же веб-странице и обеспечит минимальную функциональность, требуемую для компиляции Node и других программных проектов с открытым кодом. Чтобы быстро проверить, установлен ли пакет Xcode на вашем компьютере, запустите приложение Terminal и выполните команду xcodebuild.

Если пакет Xcode на вашем компьютере установлен, после выполнения этой команды появится сообщение об ошибке: «Does not contain an Xcode project» («В текущей папке отсутствует проект Xcode»).

## А.1.1. Установка с помощью Homebrew

Чтобы упростить установку Node в OS X, можно воспользоваться приложением Homebrew, специально предназначенным для установки программ с открытым исходным кодом.

Чтобы установить Homebrew, введите следующую команду в командной строке:

```
ruby -e "$(curl -fsSkL raw.github.com/mxcl/homebrew/go)"
```

Как только приложение Homebrew будет установлено, установите Node с помощью такой команды:

```
brew install node
```

По мере компиляции Homebrew-кода на экране будет появляться масса текстовой информации, связанной с процессом компиляции. Можете не обращать на нее внимания.

## А.2. Установка в Windows

В среде Windows платформу Node проще всего устанавливать с помощью отдельной официальной программы установки (http://nodejs.org/#download). После завершения установки вы получите возможность запускать Node и npm из командной строки Windows.

Альтернативный способ установки Node в Windows заключается в компиляции исходного кода. Этот способ более сложный и требует использования проекта Cygwin, который предоставляет UNIX-совместимое окружение. Вероятно, вы не захотите связываться со столь сложным способом установки Node, если, конечно, вам не нужны модули, которые работают в Windows только с помощью Cygwin или требуют компиляции, как некоторые модули драйверов баз данных.

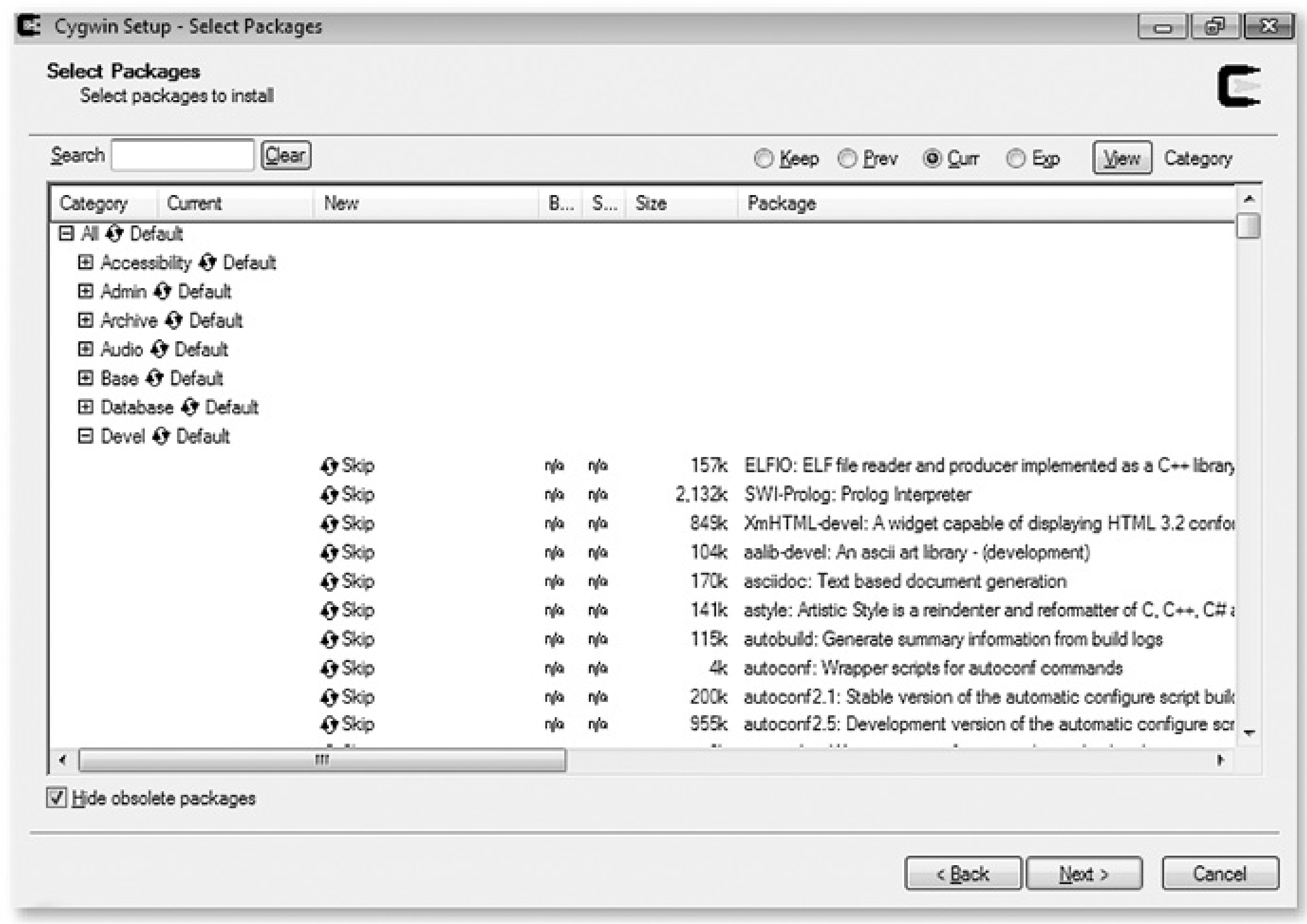



**Рис. А.2.** С помощью окна выбора пакетов Cygwin можно выбрать устанавливаемую программу с открытым кодом

Чтобы установить Cygwin, в окне веб-браузера перейдите по ссылке загрузки программы установки Cygwin (http://cygwin.com/install.html) и загрузите файл setup.exe. Дважды щелкните на этом файле, чтобы начать установку. Затем последовательно щелкайте на кнопке Next (Далее), чтобы принимать заданные по умолчанию параметры до тех пор, пока не дойдете до окна Choose a Download Site (Выберите сайт загрузки). Выберите в списке произвольный сайт загрузки, затем щелкните на кнопке Next. Если появится предупреждение о наличии более новой версии Cygwin, щелкните на кнопке OK для продолжения.

На экране появится окно выбора пакетов Cygwin, показанное на рис. А.2.

С помощью этого окна можно выбрать ту функциональность, которую вы хотите установить в вашем UNIX-подобном окружении. В табл. А.1 перечислены т устанавливаемые пакеты, которые относятся к разработке Node-приложений.

**Таблица А.1.** Cygwin-пакеты, необходимые для запуска Node

| Категория | Пакет |
|---|---|
| Разработка | gcc4-g++ |
| Разработка | git |

| | |
|---|---|
| Разработка | make |
| Разработка | openssl-devel |
| Разработка | pkg-config |
| Разработка | zlib-devel |
| Сеть | inetutils |
| python | python |
| Интернет | wget |

Завершив выбор пакетов, щелкните на кнопке Next.

На экране появится список пакетов, состав которого зависит от выбранных вами параметров. Если вы согласны с этим списком и хотите установить все пакеты, снова щелкните на кнопке Next. После этого Cygwin загрузит требуемые пакеты. По завершении загрузки щелкните на кнопке Finish (Готово).

Запустите Cygwin, щелкнув на значке рабочего стола или выбрав соответствующую команду в меню Пуск (Start). На экране появится командная строка. После этого можно компилировать Node (см. п.А.4, чтобы получить соответствующие инструкции).

## А.3. Установка в Linux

Обычно установка Node в среде Linux довольно проста. В этом разделе рассматривается установка путем компиляции исходного кода с помощью двух популярных дистрибутивов Linux: Ubuntu и CentOS. Платформу Node также можно установить с помощью диспетчера Node-пакетов, используя другие дистрибутивы. Соответствующие инструкции по установке находятся на сайте GitHub (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager).

### А.3.1. Предварительные условия установки в среде Ubuntu

Прежде чем устанавливать Node в среде Ubuntu, нужно установить пакеты, обеспечивающие нужные условия установки. Для этого на платформе Ubuntu 11.04 или более поздней версии достаточно выполнить единственную команду:

```
sudo apt-get install build-essential libssl-dev
```

**SUDO**

Команда sudo позволяет выполнить другую команду от имени «суперпользователя» (также называемого пользователем «root»). Команда Sudo часто применяется во время установки программ, поскольку при этом файлы размещаются в защищенных областях файловой системы, к которым может получать доступ только

суперпользователь.

### А.3.2. Предварительные условия установки в среде CentOS

Прежде чем установить Node в среде CentOS, нужно установить пакеты, обеспечивающие нужные условия установки. Для этого в среде CentOS 5 выполните следующие команды:

**sudo yum groupinstall 'Development Tools'**

sudo yum install openssl-devel

После установки пакетов, необходимых для установки Node, переходите к компиляции исходного кода платформы Node.

## А.4. Компиляция исходного кода платформы Node

Чтобы скомпилировать Node, в любой операционной системе требуется выполнить одни и те же действия.

Сначала в командной строке введите следующую команду, которая создаст для Node временную папку загрузки исходного кода:

mkdir tmp

Перейдите в эту папку:

cd tmp

Введите следующую команду:

curl -O http://nodejs.org/dist/node-latest.tar.gz

На экране появится текст, сопровождающий процесс загрузки. Когда индикатор загрузки достигнет значения 100 %, на экране вновь появится приглашение командной строки. Чтобы разархивировать полученные файлы, введите такую команду:

tar zxvf node-latest.tar.gz

На экране начнут появляться сообщения, иллюстрирующие процесс разархивирования, затем вновь отобразится приглашение командной строки. В командной строке введите команду ls, которая выводит список файлов в текущей папке, среди которых будет название только что разархивированной папки.

Для перехода в папку с разархивированными файлами введите следующую команду:

cd node-v*

Вы перейдете в папку с исходным кодом платформы Node. Чтобы подготовить установку в соответствии с вашей операционной системой, запустите следующий

сценарий настройки:

./configure

Затем в командной строке введите команду компиляции кода платформы Node:

make

Компиляция займет не много времени. Проявите терпение и дождитесь ее завершения. По мере компиляции на экране будет отображаться соответствующий текст, который можно игнорировать.

**Проблемы с CYGWIN**

Если вы используете пакет Cygwin на платформе Windows 7 или Vista, на этом шаге могут возникать ошибки. Причина появления ошибок связана с Cygwin, а не с Node. Чтобы устранить проблему, выйдите из оболочки Cygwin и запустите приложение командной строки ash.exe. Это приложение находится в каталоге Cygwin, обычно c:\cygwin\bin\ash.exe. В командной строке введите команду /bin /rebaseall -v. По завершении перезагрузите компьютер. Эта процедура должна помочь устранить проблему.

На этом компиляции практически завершается. Когда на экране перестанет прокручиваться текст и вновь появится приглашение командной строки, введите завершающую команду процесса установки:

sudo make install

Затем введите следующую команду, чтобы запустить Node и посмотреть номер версии. Это позволит вам убедиться в успешном завершении процесса установки:

node -v

Итак, вы установили Node на своей машине!

### А.5. Использование диспетчера Node-пакетов

После установки Node можно использовать встроенные модули, предоставляющие API-интерфейсы для реализации сетевых операций, взаимодействия с файловой системой и выполнения других действий, характерных для типичного приложения. Встроенные Node-модули считаются*ядром* платформы Node. Несмотря на массу полезной функциональности, предлагаемой ядром, вы, вероятно, захотите также задействовать функциональность модулей, предлагаемых Node-сообществом. На рис. А.3 показана концептуальная связь между ядром платформы Node и

дополнительными модулями.

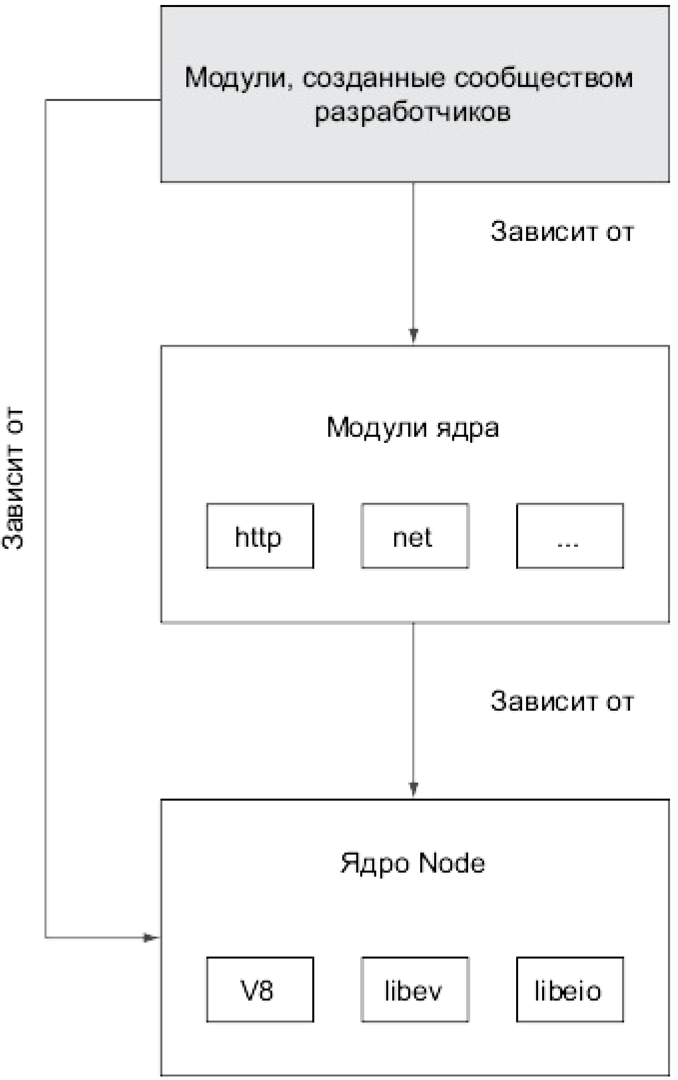



**Рис. А.3.** В Node-стек входят глобально доступная функциональность, модули ядра и модули, разработанные сообществом

В зависимости от используемого вами в работе языка программирования вам может быть знакома идея хранилищ наращиваемой функциональности, поддерживаемых сообществом разработчиков. Хранилища сродни библиотекам в том отношении, что содержат «строительные блоки» приложений, которые помогут вам делать то, что средствами исходного языка сделать сложно. Однако в отличие от библиотек хранилища, как правило, имеют модульную структуру, поэтому обычно вместо целой библиотеки достаточно извлечь из хранилища лишь нужные надстройки.

Node-сообщество располагает собственным инструментом управления надстройками, разработанными сообществом, — это диспетчер Node-пакетов (Node Package Manager, npm). В этом разделе мы поговорим о применении npm для поиска разработанных сообществом надстроек, просмотра сопутствующей документации и изучения исходного кода надстроек.

**диспетчер Node-пакетов не установлен в моей системе**

Как правило, npm устанавливается при установке Node. Чтобы это проверить, в командной строке выполните команду npm и посмотрите, сможет ли система найти эту команду. Если система ее не найдет, установите npm с помощью следующих команд:

```
cd /tmp
git clone git://github.com/isaacs/npm.git
cd npm
sudo make install
```

Завершив установку npm, в командной строке введите показанную далее команду, проверяющую работоспособность npm. Эта команда выводит номер версии npm:

```
npm -v
```

Если диспетчер Node-пакетов установлен правильно, на экране появится номер версии:

```
1.0.3
```

Если при установке npm возникли проблемы, посетите страницу проекта npm на сайте GitHub (http://github.com/isaacs/npm). Там вы найдете самые свежие инструкции относительно установки.

### А.5.1. Поиск пакетов

Утилита командной строки npm предоставляет удобный доступ к надстройкам, разработанным сообществом. Модули надстроек называются *пакетами* и хранятся в онлайновом хранилище. Аналогом npm для пользователей языков PHP, Ruby и Perl являются утилиты PEAR, Gem и CPAN соответственно.

Утилита npm невероятно удобна в применении. С ее помощью можно загружать и устанавливать пакеты одной командой. Можно также легко искать новые пакеты, просматривать относящуюся к пакетам документацию, исследовать исходный код пакетов и публиковать собственные пакеты, которые будут доступны Node-сообществу.

Чтобы искать имеющиеся в хранилище пакеты, воспользуйтесь npm-командой search. Например, для поиска пакета XML generator введите следующую команду:

```
npm search xml generator
```

Если поиск выполняется первый раз, придется немного подождать, пока загружается информация из хранилища. В дальнейшем поиск пойдет быстрее.

В качестве альтернативы поиску в режиме командной строки npm предлагает веб-интерфейс http://search.npmjs.org/. Этот веб-сайт (его начальная страница показана на рис. А.4) предоставляет, в частности, статистику: количество существующих пакетов, пакеты, которые в наибольшей степени зависят от других пакетов, пакеты, которые были недавно обновлены.

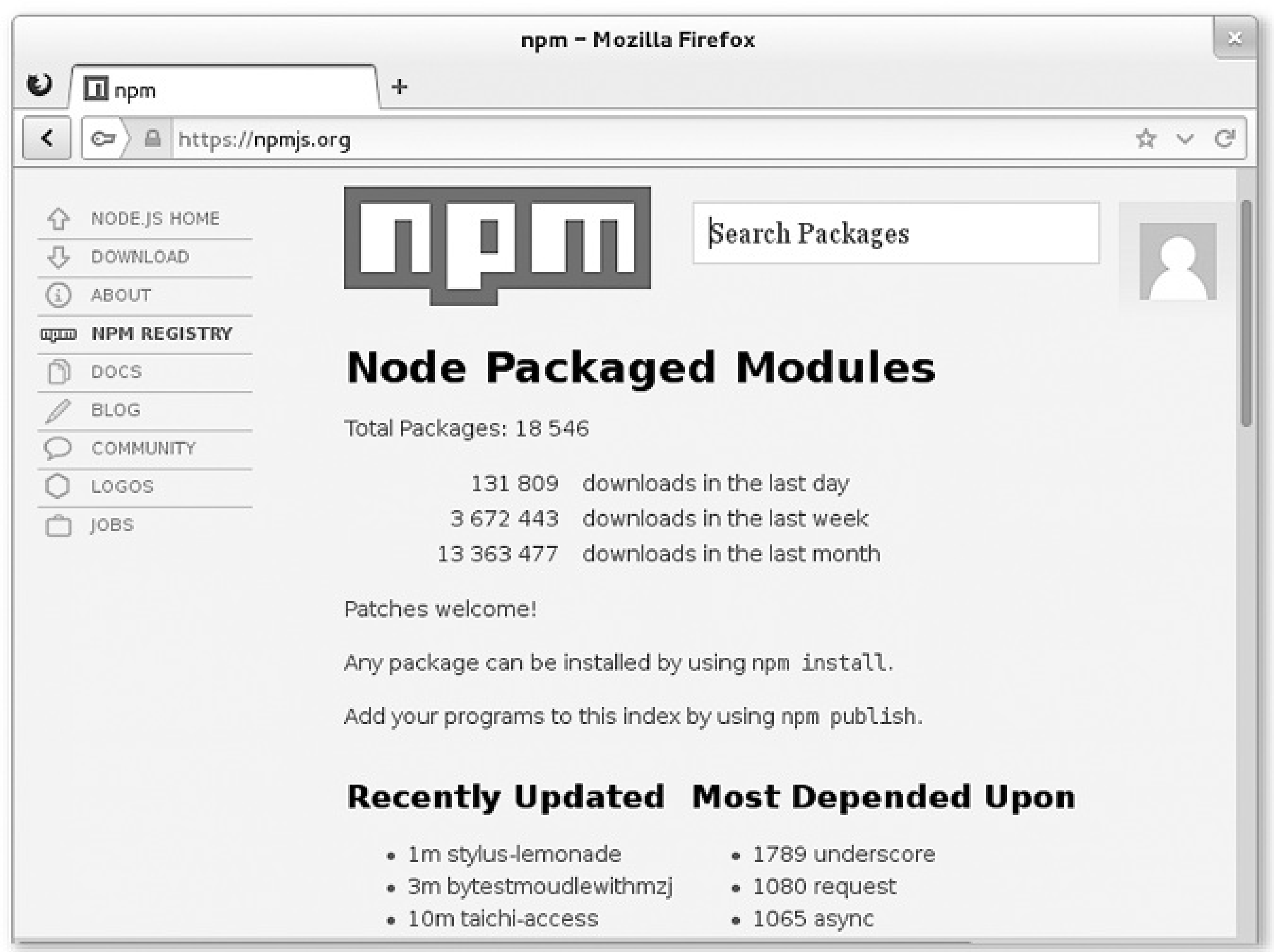



**Рис. А.4.** Веб-сайт, применяемый для поиска в npm-хранилище, показывает полезную статистику о npm-модулях

С помощью веб-интерфейса поиска в npm-хранилище можно также просматривать отдельные пакеты, выводить на экран полезную информацию, такую как зависимости пакета и местонахождение в Интернете хранилища контроля версий проекта.

### А.5.2. Установка пакетов

Когда вы найдете пакеты, которые нужно установить, выберите один из способов установки. В npm доступно два основных варианта установки: локальный и глобальный.

В случае *локальной* установки пакета загруженный модуль помещается в папку node_modules, которая находится в текущем рабочем каталоге. Если эта папка не существует, npm создаст ее.

Обратите внимание на пример локальной установки пакета express:

```
npm install express
```

В случае *глобальной* установки пакета загружаемый модуль помещается в каталог /usr/local, поддерживаемый в операционных системах, отличных от Windows. Обычно UNIX использует этот каталог для хранения приложений установленных пользователем. В Windows глобально установленные npm-модули хранятся в папке Appdata\Roaming\npm, вложенной в папку пользователя.

Вот пример глобальной установки пакета express:

```
npm install -g express
```

Если вы не обладаете необходимыми для доступа к файлу полномочиями, при выполнении глобальной установки предваряйте команду установки ключевым словом sudo. Например:

```
sudo npm install -g express
```

После установки пакета нужно разобраться в том, как он работает. С помощью npm это сделать несложно.

### А.5.3. Изучение документации и кода пакета

Диспетчер Node-пакетов предлагает удобный способ просмотра онлайновой документации, входящей в комплект поставки пакета. В результате выполнения команды npm docs в окне веб-браузера открывается страница документации пакета. Чтобы просмотреть документацию по пакету express, выполните следующую команду:

```
npm docs express
```

Документацию по пакету можно просматривать даже в том случае, когда сам пакет не установлен.

Если документация по пакету неполная или не очень понятная, может потребоваться просмотр исходных файлов пакета. Диспетчер Node-пакетов предлагает простой способ порождения подоболочки, в качестве рабочего каталога которой выбран каталог верхнего уровня с исходными файлами пакета. С помощью следующей команды можно изучать исходные файлы локально установленного пакета express:

```
npm explore express
```

Чтобы исследовать исходные файлы глобально установленного пакета, добавьте параметр командной строки -g после команды npm. Например:

npm -g explore express

Исследование пакетов — прекрасный способ научиться чему-то новому. Читая исходный Node-код, вы познакомитесь с новыми для вас приемами программирования и способами организации кода.

## Отладка Node-приложений

При разработке приложений, а также при изучении нового языка программирования или среды разработки вам может помочь знание инструментов и приемов отладки. В этом приложении вы познакомитесь с несколькими приемами, позволяющими точно понять, что происходит с кодом вашего Node-приложения.

### Б.1. Анализ кода в JSHint

При разработке приложений случаются синтаксические и логические ошибки. Чтобы выяснить причину ошибки, сначала нужно изучить проблемный код. Если при этом вы не сможете обнаружить причину ошибки, попробуйте воспользоваться утилитой для анализа исходного кода.

Одной из таких утилит является JSHint. Эта утилита может предупреждать о фатальных ошибках в коде, таких как вызовы функций, которые нигде не определены, или стилистических ошибках, таких как нарушение принятых в JavaScript соглашений о применении прописных букв в названиях конструкторов классов. Даже если вам никогда не придется запускать JSHint, знакомство с типами ошибок, распознаваемых JSHint, поможет вам избежать подобных ошибок в программах.

Утилита JSHint разработана на базе JSLint — весьма популярного инструмента анализа исходного JavaScript-кода. Однако JSLint не слишком хорошо поддается настройке, и здесь в игру вступает JSHint.

По мнению многих пользователей, утилита JSLint предлагает чрезмерно строгие рекомендации в отношении стиля. В то же время в JSHint можно указать не только те конструкции, которые нужно проверять, но и те конструкции, которые проверять не нужно. Например, точки с запятой технически требуются интерпретаторам JavaScript-кода, но в большинстве интерпретаторов поддерживается автоматическая вставка точки с запятой (Automated Semicolon Insertion, ASI). Зная об этом, многие разработчики не вставляют точки с запятой, чтобы лишний раз не «загрязнять» код. В то время как утилита JSLint «жалуется» на ошибки, вызванные отсутствием точек с запятой, утилиту JSHint можно сконфигурировать таким образом, чтобы игнорировать подобную «ошибку» и

проверять только те ошибки, которые вызывают сбой приложения.

После установки JSHint становится доступным инструмент командной строки jshint, который проверяет исходный код. Утилиту JSHint нужно установить глобально, выполнив следующую npm-команду:

npm install -g jshint

После установки JSHint введите следующую команду для проверки JavaScript-файлов:

jshint my_app.js

Для утилиты JSHint можно создать конфигурационный файл, показывающий, что именно нужно проверять. Создать такой файл можно путем изменения заданного по умолчанию конфигурационного файла, который доступен на сайте GitHub (https://github.com/jshint/node-jshint/blob/master/.jshintrc). Скопируйте этот файл на свою рабочую станцию, а затем модифицируйте его требуемым образом.

Если вы присвоите своей версии конфигурационного файла расширение .jshintrc и поместите его в папке приложения или в произвольной родительской для папки приложения папке, JSHint сможет находить и использовать этот файл автоматически.

Чтобы указать местоположение конфигурационного файла, можно также выполнить команду jshint с флагом config. Вот как выглядит команда запуска утилиты JSHint, которая использует конфигурационный файл с нестандартным именем:

jshint my_app.js —config /home/mike/jshint.json

Дополнительные сведения о каждом конфигурационном параметре можно найти на веб-сайте JSHint (http://www.jshint.com/docs/#options).

## Б.2. Вывод отладочной информации

Если код выглядит вполне работоспособным, но ведет себя непредсказуемым образом, выведите отладочную информацию, которая поможет вам разобраться в сути происходящего.

### Б.2.1. Отладка с помощью модуля console

Встроенный в Node модуль console обеспечивает вывод информации, в том числе отладочной, на консоль.

#### *Вывод информации о состоянии приложения*

Для вывода информации о состоянии приложения в стандартный поток вывода применяется функция console.log, которая также может называться console.info. В отношении принимаемых аргументов эта функция напоминает функцию printf() (http://ru.wikipedia.org/wiki/Printf):

```
console.log('Counter: %d', counter);
```

Функции console.warn и console.error похожи на функцию console.log, но служат для вывода предупреждений и сообщений об ошибках. Единственное отличие заключается в том, что вместо стандартного потока вывода вывод происходит в поток ошибок. При использовании этих функций можно перенаправить предупреждения и сообщения об ошибках в файл журнала, как показано в следующем примере кода:

```
node server.js 2> error.log
```

Функция console.dir выводит содержимое объекта. Соответствующий пример кода, выполняющего подобный вывод, может выглядеть так:

```
{ name: 'Paul Robeson',
  interests: [ 'football', 'politics', 'music', 'acting' ] }
```

### *Вывод информации о времени выполнения программы*

В состав модуля console включены две функции, которые при совместном использовании позволят оценить время выполнения отдельных блоков кода, причем нескольких одновременно.

Чтобы приступить к оценке времени выполнения кода, добавьте следующую строку в ту точку кода программы, с которой начнется отсчет времени выполнения кода:

```
console.time('myComponent');
```

Для завершения замера, позволяющего оценить время выполнения программы, в соответствующую точку кода добавьте такую строку:

```
console.timeEnd('myComponent');
```

Эта строка выведет время выполнения кода.

### *Вывод трассы стека*

В результате трассировки стека можно получить сведения о функциях, выполняемых до достижения определенной точки кода. Например, если в процессе выполнения Node-приложения произойдет ошибка, путем трассировки стека можно получить сведения о причинах ошибки и обнаружить место ее возникновения в коде приложения.

В любой момент выполнения приложения можно вывести трассу стека без остановки выполняемого приложения. Для этого используется функция console.trace().

В результате будет сгенерирована примерно такая трасса стека :

```
Trace:
  at lastFunction (/Users/mike/tmp/app.js:12:11)
  at secondFunction (/Users/mike/tmp/app.js:8:3)
  at firstFunction (/Users/mike/tmp/app.js:4:3)
  at Object.<anonymous> (/Users/mike/tmp/app.js:15:3)
...
```

Обратите внимание, что на трассе стека ход выполнения программы отражается в обратном хронологическом порядке.

### Б.2.2. Использование модуля debug для управления выводом отладочной информации

Вывод отладочной информации, несомненно, полезен, но если вы активно не занимаетесь поиском и устранением ошибок, обилие выводимой на экран отладочной информации может лишь раздражать. В идеале было бы неплохо, чтобы режим вывода отладочной информации можно было включать и выключать.

Один из способов управления выводом отладочной информации заключается в использовании переменной окружения. Удобным инструментом для управления выводом отладочной информации с помощью переменной окружения DEBUG является модуль debug, созданный TJ Головайчуком (TJ Holowaychuk). Подробно об этом рассказывается в главе 13.

## Б.3. Встроенный в Node отладчик

Если ваши потребности в плане отладки не ограничиваются простым получением отладочной информации, воспользуйтесь встроенным в Node отладчиком командной строки. Чтобы запустить отладчик, в команду запуска приложения включите ключевое слово debug:

```
node debug server.js
```

Если подобным образом запустить Node-приложение, появятся первые несколько строк кода и приглашение командной строки отладчика (рис. Б.1).

```
2. Shell
$ node debug server.js
< debugger listening on port 5858
connecting... ok
break in server.js:1
  1 var http = require('http');
  2
  3 http.createServer(function (req, res) {
debug>
```

**Рис. Б.1.** Запуск встроенного в Node отладчика

Строка break in server.js:1 означает, что отладчик остановлен перед первой строкой кода.

### Б.3.1. Навигация с помощью отладчика

С помощью отладчика можно управлять выполнением приложения. Чтобы выполнить следующую строку кода, введите в командной строке отладчика команду next (или просто n). Если же в командной строке отладчика ввести команду cont (или просто с), код будет выполняться вплоть до прерывания.

Выполнение отладчика может прерываться при завершении работы приложения или в *точках прерывания* (breakpoints). В точке прерывания отладчик останавливает выполнение приложения, что позволяет вам проверить состояние приложения.

Один из способов добавить точку прерывания в программу — вставить строку в то место кода приложения, в котором вы хотите установить точку прерывания. В этой строке должна находиться инструкция debugger;, как показано в листинге Б.1. При обычном режиме выполнения кода строка, содержащая инструкцию debugger;, на выполнение Node-приложения не влияет.

**Листинг Б.1.** Программное добавление точки прерывания

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

function handleRequest(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}

http.createServer(function (req, res) {
  // Добавление точки прерывания в код
  debugger;
  handleRequest(req, res);
```

```
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

Если запустить код из листинга Б.1 в режиме отладки, первое прерывание кода произойдет в первой строке. Если в командной строке отладчика ввести команду cont, будет создан HTTP-сервер, ожидающий подключения. Если создать соединение, введя в адресной строке браузера адрес http://127.0.0.1:1337, выполнение кода остановится на строке с инструкцией debugger;.

Чтобы перейти к следующей строке, в командной строке отладчика введите команду next. После этого текущей станет строка, в которой находится вызов функции handleRequest. Если снова ввести команду next, чтобы перейти к следующей строке кода, отладчик не будет переходить к каждой строке кода внутри функции handleRequest. Если же ввести команду step, отладчик перейдет к коду функции handleRequest и будет выполнять поиск ошибок в коде этой функции. Если вы передумаете отлаживать код функции, в командной строке отладчика введите команду out (или o), чтобы выйти из кода функции и вернуться к основному коду приложения.

Помимо исходного кода точки прерывания можно устанавливать для самого отладчика. Чтобы установить точку прерывания в текущей строке отладчика, в командной строке отладчика введите команду set-Breakpoint() (или sb()). Можно задать точку прерывания в определенной строке кода (sb(line)) или для определенной вызываемой функции (sb('fn()')).

Чтобы удалить точку прерывания, воспользуйтесь функцией clearBreakpoint() (cb()). Эта функция принимает те же аргументы, что и функция setBreakpoint(), только решает обратную задачу.

### Б.3.2. Проверка и изменение состояния с помощью отладчика

Чтобы отслеживать определенные значения в приложении, нужны *наблюдатели* (watchers). С помощью наблюдателей вы сможете получать информацию о значениях переменных, переходя по коду от инструкции к инструкции.

Например, если в отладочный код из листинга Б.1 ввести команду watch ("req.headers['user-agent']"), то на каждом шаге выполнения кода будет отображаться тип браузера, выполнившего запрос. Для просмотра списка наблюдателей введите команду watchers. Чтобы удалить наблюдатель, воспользуйтесь командой unwatch, которая может, например, применяться в такой форме:

```
unwatch("req .headers['useragent']")
```

Если в какой-то момент отладки нужно полностью проверить состояние или выполнить какие-либо манипуляции, воспользуйтесь командой repl для открытия REPL-сеанса. В REPL-сеансе можно ввести любое JavaScript-выражение и оценит его состояние. Чтобы выйти из REPL-сеанса и вернуться в окно отладчика, нажмите комбинацию клавиш Ctrl+C.

Чтобы выйти из окна отладчика после завершения отладки, дважды нажмите комбинацию клавиш Ctrl+C или Ctrl+D либо введите команду .exit.

На этом рассмотрение основных сведений об отладчике завершается. За дополнительной информацией по использованию отладчика обратитесь на официальный веб-сайт Node по адресу http://nodejs.org/api/debugger.html.

## Б.4. Node-инспектор

Node-инспектор (Node Inspector) — это альтернатива встроенному в Node отладчику. В качестве интерфейса вместо командной строки в Node-инспекторе используется браузер на базе WebKit, такой как Chrome или Safari.

### Б.4.1. Запуск Node-инспектора

Прежде чем приступать к отладке приложений с помощью Node-инспектора, установите этот отладчик следующей npm-командой:

npm install -g node-inspector

После завершения установки в системе станет доступна команда node-inspector.

Чтобы отладить Node-приложение, запустите его с параметром —debug-brk:

node —debug-brk server.js

Параметр командной строки —debug-brk приведет к запуску процесса отладки и вставке точки прерывания перед первой строкой кода приложения. Если подобное поведение отладчика вас не устраивает, укажите параметр командной строки —debug.

После запуска приложения запустите Node-инспектор:

node-inspector

Node-инспектор интересен тем, что в нем используется тот же код, что и в веб-инспекторе движка WebKit, только подключен этот код к JavaScript-движку платформы Node, поэтому веб-разработчики должны чувствовать себя комфортно.

После запуска Node-инспектора в окне WebKit-браузера перейдите по адресу http://127.0.0.1:8080/debug? port=5858, и вы попадете в окно Node-инспектора. Если Node-инспектор запускался с параметром —debug-brk, в окне отладчика появится первый сценарий вашего приложения (рис. Б.2). Если же применялся параметр —

debug, для выбора отлаживаемого сценария можно воспользоваться селектором сценариев (на рисунке выбран сценарий step.js).

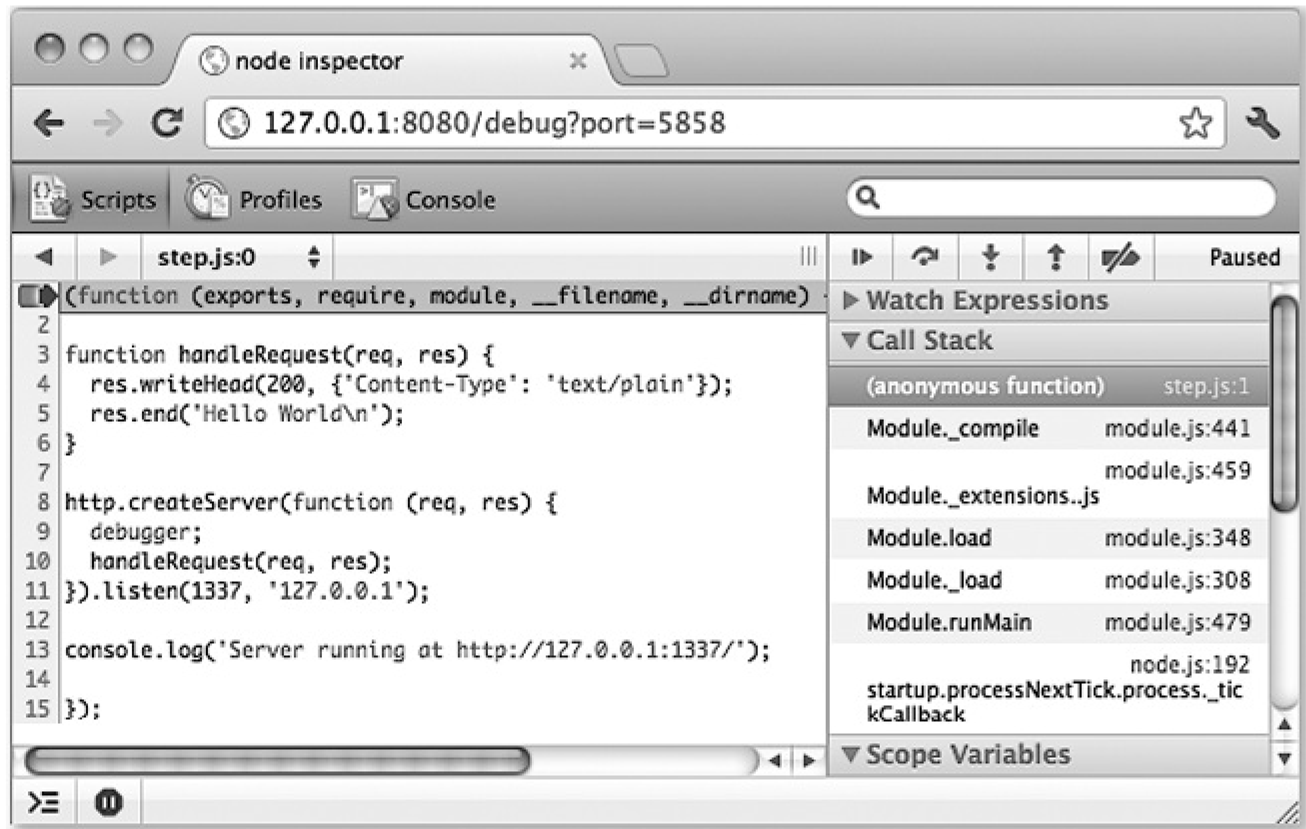



**Рис. Б.2.** Окно Node-инспектора

Красная стрелка, находящаяся слева от строки кода, показывает, что эта строка выполняется следующей.

## Б.4.2. Навигация в окне Node-инспектора

Чтобы перейти к следующему вызову функции в коде приложения, щелкните на кнопке в виде маленького круга, по окружности которого направлена стрелка. Node-инспектор, подобно отладчику командной строки в Node, позволит вам перейти к коду функции. Когда красная стрелка окажется слева от вызова функции, можете перейти к этому коду, щелкнув на значке в виде маленькой окружности со стрелкой, указывающей в нижнюю часть окружности. Чтобы выйти из кода функции, щелкните на значке в виде маленькой окружности со стрелкой, которая указывает в верхнюю часть окружности. Если вы используете модуль ядра Node или модуль от Node-сообщества, отладчик переключится к файлам сценариев этих модулей после выполнения шага отладки в коде приложения. Не волнуйтесь понапрасну, через некоторое время отладчик вернется к коду вашего приложения.

Чтобы в окне Node-инспектора добавить точку прерывания, щелкните на номере строки слева от любой строки сценария. Чтобы удалить все точки прерывания, щелкните на кнопке, находящейся справа от кнопки шага с выходом

(стрелка, направленная вверх).

Node-инспектор предлагает еще один интересный механизм, позволяющий изменить код в процессе выполнения приложения. Если вы хотите изменить строку кода, просто щелкните на ней, измените ее, а затем щелкните за пределами строки кода.

### Б.4.3. Просмотр состояния в окне Node-инспектора

В процессе отладки приложения можно проверять его состояние, используя свертываемые панели. Эти панели находятся под кнопками, с помощью которых можно перемещаться в окне инспектора (рис. Б.3). С помощью этих панелей можно проверять стек вызовов и переменные, которые задействованы в данный момент времени. Для изменения переменной нужно дважды щелкнуть на ней, а затем изменить значение. Как и в случае со встроенным в Node отладчиком командной строки, можно добавлять команды отслеживания, которые при отладке кода приложения позволят получать нужную информацию по мере перехода от строки к строке.

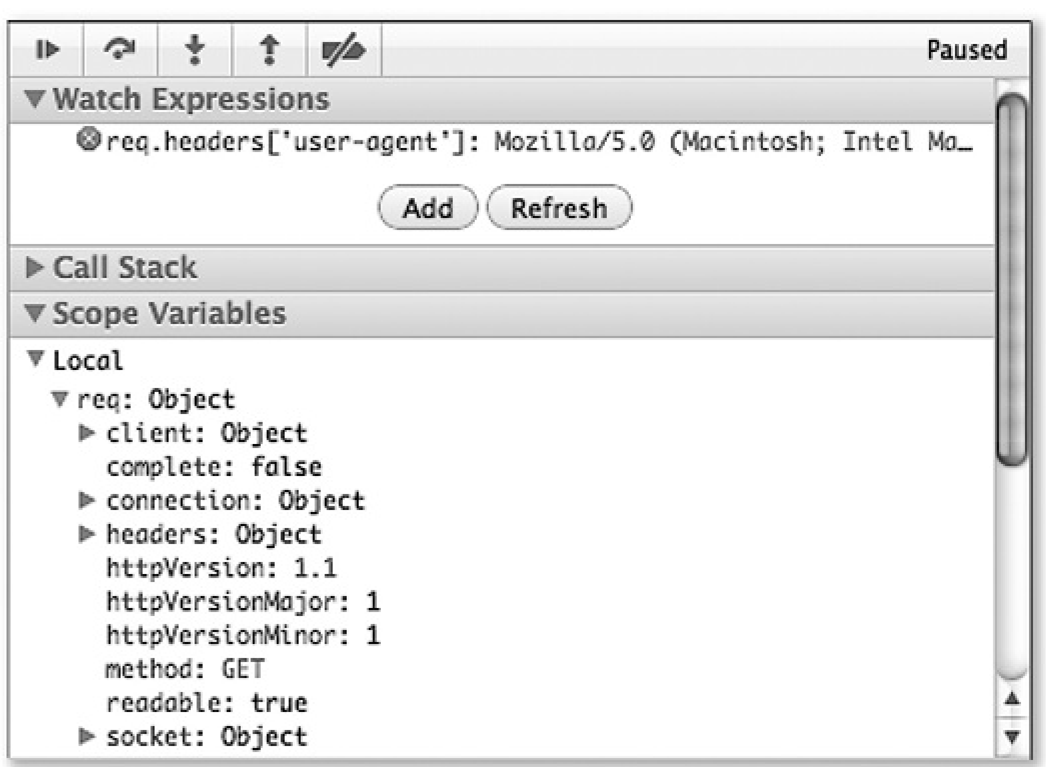



**Рис. Б.3.** Просмотр состояния приложения с помощью Node-инспектора

Чтобы получить дополнительные сведения о работе Node-инспектора, посетите соответствующую страницу проекта GitHub (https://github.com/dannycoates/node-inspector/).

**сомневаешься? обнови!**

Если при использовании Node-инспектора вы замечаете какие-либо странности, попробуйте обновить окно браузера. Если это не поможет, попробуйте перезапустить и Node-приложение, и Node-инспектор.

## Расширение и конфигурирование среды Express

Среда разработки Express способна на многое и без дополнительной настройки, но если вы расширите среду Express и настроите ее конфигурацию, то сможете упростить разработку приложений и выжмете из нее максимум полезного.

### В.1. Расширение среды Express

Давайте начнем с прояснения вопроса о расширении среды Express. В этом разделе мы узнаем:

- как создавать собственные шаблонизаторы;
- как использовать шаблонизаторы, созданные сообществом;
- как совершенствовать приложения с помощью модулей, расширяющих среду Express.

#### В.1.1. Регистрация шаблонизаторов

Шаблонизаторы могут изначально поддерживать среду Express, экспортируя метод __express. Но далеко не каждый шаблонизатор на это способен, к тому же вы можете создавать собственные шаблонизаторы. В Express создание шаблонизаторов облегчается благодаря методу app.engine(). В этом разделе мы займемся созданием компактного шаблонизатора markdown, поддерживающего подстановку переменных для динамического контента.

Метод app.engine() *проецирует* (maps) расширение файла на функцию обратного вызова, чтобы указать Express на способ обработки соответствующих файлов. В листинге В.1 расширение .md передается для того, чтобы визуализаторы, такие как res.render('myview.md'), использовали функцию обратного вызова для визуализации файла. Благодаря подобной абстракции в среде разработки можно задействовать практически любой шаблонизатор. В нашем нестандартном шаблонизаторе фигурные скобки вокруг локальных переменных обеспечивают возможность динамического ввода: например, строка {name}, встретившаяся в шаблоне, будет

приводить к выводу значения переменной name.

**Листинг В.1.** Обработка расширения .md

```
var express = require('express');
var http = require('http');
// Требуется реализация markdown
var md = require('github-flavored-markdown').parse;
var fs = require('fs');

var app = express();
// Связывание обратного вызова с файлами .md
app.engine('md', function(path, options, fn){
  // Считывание содержимого файла в виде строки
  fs.readFile(path, 'utf8', function(err, str){
     // Делегирование ошибок в Express
    if (err) return fn(err);
    try {
      // Преобразование markdown-строки в HTML-код
      var html = md(str);
       // Подстановка фигурных скобок
      html = html.replace(/\{([^}]+)\}/g, function(_, name){
        // Значение, заданное по умолчанию, равно '' (пустая строка)
        return options[name] || '';
        });
        // Передача визуализированного HTML-кода в Express
        fn(null, html);
      } catch (err) {
      // Перехват всех выброшенных ошибок
      fn(err);
    }
  });
});
```

Шаблонизатор, код которого приведен в листинге В.1, позволяет создавать динамические представления для шаблонизатора markdown. Например, если вы хотите поздравить пользователя со скидкой[10], воспользуйтесь следующим примером кода:

```
# {name}

Greetings {name}! Nice to have you check out our application {appName}.
```

### B.1.2. Шаблоны, поддерживаемые consolidate.js

Проект consolidate.js был разработан специально для Express 3.x; он предоставляет простой унифицированный API-интерфейс для многих Node-шаблонизаторов. В результате среда Express 3.x без дополнительной настройки позволяет использовать более 14 различных шаблонизаторов. Если же задействовать библиотеку шаблонов, вы с помощью consolidate.js получите доступ к огромному числу шаблонизаторов.

Например, идея создания шаблонизатора Swig была навеяна Django. Этот шаблонизатор использует теги, внедренные в HTML, для определения программной логики, как показано в следующем примере кода:

```
<ul>
  {% for pet in pets %}
    <li>{{ pet.name }}</li>
  {% endfor %}
</ul>
```

В зависимости от применяемого шаблонизатора и наличия в редакторе средств поддержки выделения синтаксических конструкций, возможно, было бы здорово, чтобы шаблонизаторы в стиле HTML использовали расширение .html, а не расширение, основанное на названии шаблонизатора, такое как .swig. Для этого можно использовать Express-метод app.engine(). После единственного вызова, в результате которого Express визуализирует файл .html, в дальнейшем будет применяться Swig:

```
var cons = require('consolidate');
app.engine('html', cons.swig);
```

Шаблонизатор EJS также, вероятно, мог бы проецироваться на .html, поскольку тоже использует внедренные теги:

```
<ul>
  <% pets.forEach(function(pet){ %>
    <li><%= pet.name %></li>
  <% }) %>
</ul>
```

Некоторые шаблонизаторы применяют совсем другой синтаксис, поэтому не имеет смысла проецировать его на файлы .html. Хороший пример тому —

шаблонизатор Jade, поддерживающий собственный декларативный язык. Jade мог бы проецироваться со следующим вызовом:

```
var cons = require('consolidate');
app.engine('jade', cons.jade);
```

Чтобы получить дополнительные сведения и список поддерживаемых шаблонизаторов, посетите хранилище проекта consolidate.js по адресу https://github.com/visionmedia/consolidate.js.

### B.1.3. Расширение функциональных возможностей среды Express и доступные среды Express

Вы могли бы удивиться, какие возможности доступны разработчикам при использовании более структурированных сред разработки, таких как Ruby on Rails. Express предлагает в этом плане несколько возможностей.

Express-сообщество разработало несколько высокоуровневых сред разработки, надстраиваемых над Express. Благодаря этому поддерживается структура каталогов, а также высокоуровневые механизмы, такие как контроллеры в стиле Rails. В дополнение к этим средам Express предлагает множество подключаемых модулей, обеспечивающих расширение функциональных возможностей среды Express.

#### *Подключаемый модуль express-expose*

Подключаемый модуль express-expose может применяться с целью экспонирования для клиентов серверных JavaScript-объектов. Например, если вы хотите экспонировать JSON-представление аутентифицированного пользователя, можете вызвать функцию res.expose(), чтобы предоставить объект express.user клиентскому коду:

```
res.expose(req.user, 'express.user');
```

#### *Подключаемый модуль express-resource*

Еще один насыщенный ресурсами подключаемый модуль, express-resource, используется для структурированной маршрутизации.

Маршрутизация может выполняться различными способами, но все они в конечном итоге сводятся к использованию метода запроса и пути, которые изначально поддерживаются Express. А более высокоуровневые концепции надстраиваются сверху.

Представленный далее пример кода демонстрирует, каким образом можно определить действия, выполняемые для вывода на экран, создания и обновления в декларативном стиле некоего пользовательского ресурса. Сначала добавим следующую строку кода в файл app.js:

```
app.resource('user', require('./controllers/user'));
```

А код модуля контроллера ./controllers/user.js мог бы выглядеть так, как показано в листинге В.2.

**Листинг В.2.** Ресурсный файл user.js

```
exports.new = function(req, res){
  res.send('new user');
};

exports.create = function(req, res){
  res.send('create user');
};

exports.show = function(req, res){
  res.send('show user ' + req.params.user);
};
```

Чтобы увидеть полный список надстроек, шаблонизаторов и сред разработки, обратитесь к вики-странице, посвященной Express, на сайте https://github.com/visionmedia/express/wiki.

## В.2. Дополнительное конфигурирование

В предыдущих разделах мы выяснили, как конфигурировать Express с помощью функции app.configure(), и рассмотрели многочисленные параметры конфигурирования. В этом разделе мы узнаем о дополнительных возможностях конфигурирования, которые можно использовать для изменения заданного по умолчанию поведения и разблокирования дополнительной функциональности.

В табл. В.1 перечислены параметры конфигурирования Express, которые не упоминались в главе 8.

**Таблица В.1.** Встроенные варианты настройки Express

| | |
|---|---|
| default engine | Заданный по умолчанию шаблонизатор |
| views | Путь доступа к представлениям |
| | |

| json replacer | Функция для обработки JSON-ответов |
|---|---|
| json spaces | Количество пробелов, используемых для форматирования JSON-ответов |
| jsonp callback | Поддержка формата JSONP с помощью функций res.json() и res.send() |
| trust proxy | Заслуживающий доверия реверсный прокси-сервер |
| view cache | Функции кэширования шаблонизатора |

Конфигурационный параметр views просто показывает, где находятся шаблоны представлений. Когда в командной строке с помощью команды express создается каркас приложения, параметру views автоматически присваивается название вложенной папки, в которой находятся представления для приложения.

Давайте взглянем на более востребованный конфигурационный параметр — json_replacer.

### В.2.1. Манипулирование JSON-ресурсами

Предположим, вы хотите задать для объекта user приватные свойства, например идентификатор объекта (_id). По умолчанию в ответ на вызов метода res.send(user) будет сформирован JSON-ответ, например такой:

```
{"_id":123,"name":"Tobi"}
```

Параметр json replacer принимает функцию, которую Express передаст методу JSON.stringify() во время вызовов res.send() и res.json().

Отдельное Express-приложение, код которого приведен в листинге В.3, иллюстрирует использование этой функции для игнорирования в любом JSON-ответе свойств, начинающихся с символа подчеркивания. В данном примере ответ будет выглядеть так:

```
{"name":"Tobi"}
```

**Листинг В.3.** Использование параметра json_replacer для обработки и модификации JSON-данных

```
var express = require('express');
var app = express();

app.set('json replacer', function(key, value){
  if ('_' == key[0]) return;
  return value;
});

var user = { _id: 123, name: 'Tobi' };
```

```
app.get('/user', function(req, res){
  res.send(user);
});

app.listen(3000);
```

Обратите внимание, что отдельные объекты или прототипы объектов могут быть реализованы методом .toJSON(). Этот метод используется функцией JSON.stringify() при преобразовании объекта в JSON-строку. Это прекрасна альтернатива обратному вызову json_replacer, если ваши манипуляции не применяются к каждому объекту.

Теперь, когда мы узнали, как управлять JSON-данными, которые экспонируются при выводе, давайте выясним, как можно тонко настроить формат JSON-ответов.

### B.2.2. Форматирование JSON-ответов

Конфигурационный параметр json spaces влияет на вызовы функции JSON.stringify() в Express. Этот параметр задает количество пробельных символов используемых для форматирования JSON-данных в виде строки.

По умолчанию данный метод возвращает сжатый JSON-массив, который может выглядеть примерно так:

```
{"name":"Tobi","age":2,"species":"ferret"}
```

Сжатый JSON-массив идеален для рабочего окружения, поскольку позволяет уменьшить размер ответа. Однако на этапе разработки лучше обеспечить вывод несжатых данных, поскольку их гораздо проще читать.

Параметру json spaces автоматически присваивается значение 0 в рабочем окружении, а в среде разработки он получает значение 2, что приводит к следующему выводу:

```
{
  "name": "Tobi",
  "age": 2,
  "species": "ferret"
}
```

### B.2.3. Поля заголовка заслуживающего доверия реверсного прокси-сервера

По умолчанию Express в любом окружении не доверяет полям заголовка реверсного прокси-сервера. Хотя изучение реверсных прокси-серверов выходит за рамки темы этой книги, отметим, что если ваше приложение выполняется под защитой

реверсного прокси-сервера, такого как Nginx, HAProxy и Varnish, вам нужно установить параметр trust proxy, чтобы среда Express расценивала поля заголовка как безопасные.

[10] Markdown переводится как «скидка».