2-е издание

Node.js в действии

Алекс Янг
Брэдли Мек
Майк Кантелон

А также

Тим Оксли
Марк Хартер
Т. Дж. Головайчук
Натан Райлих

MANNING

ПИТЕР

Node.js in Action

SECOND EDITION

ALEX YOUNG
BRADLEY MECK
MIKE CANTELON

WITH
TIM OXLEY
MARC HARTER
T.J. HOLOWAYCHUK
NATHAN RAJLICH

MANNING
SHELTER ISLAND

Алекс Янг, Брэдли Мек, Майк Кантелон
А также Тим Оксли, Марк Хартер, Т. Дж. Головайчук, Натан Райлих

Node.js в действии

2-е издание

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск
2018

ББК 32.988-02-018
УДК 004.738.5
Я60

Янг А., Мек Б., Кантелон М.

Я60 Node.js в действии. 2-е изд. — СПб.: Питер, 2018. — 432 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-03212-4

Второе издание «Node.js в действии» было полностью переработано, чтобы отражать реалии, с которыми теперь сталкивается каждый Node-разработчик. Вы узнаете о системах построения интерфейса и популярных веб-фреймворках Node, а также научитесь строить веб-приложения на базе Express с нуля. Теперь вы сможете узнать не только о Node и JavaScript, но и получить всю информацию, включая системы построения фронтэнда, выбор веб-фреймворка, работу с базами данных в Node, тестирование и развертывание веб-приложений.

Технология Node все чаще используется в сочетании с инструментами командной строки и настольными приложениями на базе Electron, поэтому в книгу были включены главы, посвященные обеим областям.

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

ББК 32.988-02-018
УДК 004.738.5

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

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

ISBN 978-1617292576 англ.
978-5-496-03212-4

© 2017 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Для профессионалов», 2018

Краткое содержание

Оглавление

Предисловие

С момента публикации первого издания «Node.js в действии» проект Node объединился с io.js, а модель управления радикально изменилась. Менеджер пакетов Node выделился в успешную новую компанию npm, а такие технологии, как Babel и Electron, изменили панораму разработки.

Тем не менее в базовых библиотеках Node изменений не так уж много. Сам язык JavaScript изменился: многие разработчики теперь используют возможности ES2015, поэтому исходные листинги были переписаны для «стрелочных» функций, констант и деструктуризации. При этом библиотеки и встроенные инструменты Node все еще очень похожи на свои аналоги Node до выхода версий 4.x, поэтому мы обратились к сообществу за идеями обновления этого издания.

Чтобы отразить реалии, с которыми теперь сталкивается Node-разработчик, мы изменили структуру книги. Express и Connect уделено меньше внимания, и книга в большей степени ориентирована на широкий спектр технологий. Вы найдете в книге всю информацию, необходимую для разработки в полном технологическом стеке, включая системы построения фронтэнда, выбор веб-фреймворка, работу с базами данных в Node, написание тестов и развертывание веб-приложений.

Помимо разработки веб-приложений в книгу также были добавлены главы о написании приложений командной строки и приложений Electron. Они помогут вам извлечь максимум практической пользы из ваших навыков использования Node и JavaScript.

Node и экосистема этой технологии — не единственная тема книги. Я по возможности постарался добавить историческую справку о том, что повлияло на развитие Node. Наряду с обычными темами Node и JavaScript рассматриваются такие вопросы, как философия Unix и корректное, безопасное использование баз данных. Хочется верить, что у вас сформируется достаточно широкая картина Node и JavaScript, которая поможет вам в поиске ваших собственных решений современных задач.

Алекс Янг

Благодарности

Эта книга основана на материале авторов предыдущего издания и очень многим обязана их труду: это Майк Кантелон (Mike Cantelon), Марк Хартер (Marc Harter), Т. Дж. Головайчук (T. J. Holowaychuk) и Натан Райлих (Nathan Rajlich). Это издание увидело свет только благодаря активной помощи рабочей группы из издательства Manning. Синтия Кейн (Cynthia Kane), руководитель проекта, помогала мне не отвлекаться от сути во время долгого процесса обновления исходного материала. Без тщательной научной редактуры, проведенной Дугом Уорреном (Doug Warren), книга и примеры кода не были бы и наполовину так хороши. Наконец, мы хотим поблагодарить многих рецензентов, поделившихся своим мнением во время написания и разработки: Остин Кинг (Austin King), Карл Хоуп (Carl Hope), Крис Солч (Chris Salch), Кристофер Рид (Christopher Reed), Дейл Фрэнсис (Dale Francis), Хафиз Вахид уд дин (Hafiz Waheed ud din), Харинат Маллепалли (Harinath Mallepally), Джефф Смит (Jeff Smith), Марк-Филипп Хуге (Marc-Philippe Huget), Мэттью Бертони (Matthew Bertoni), Филипп Шарьер (Philippe Charrière), Рэнди Камрадт (Randy Kamradt), Сандер Россел (Sander Rossel), Скотт Дирбек (Scott Dierbeck) и Уильям Уилер (William Wheeler).

Алекс Янг

О книге

Первое издание книги «Node.js в действии» было посвящено разработке веб-приложений, при этом особое внимание уделялось веб-фреймворкам Connect и Express. Второе издание «Node.js в действии» было переработано в соответствии с изменившимися требованиями в области разработки Node. Вы узнаете о системах построения интерфейса и популярных веб-фреймворках Node, а также научитесь строить веб-приложения на базе Express с нуля. Также в книге рассказано о том, как строить автоматизированные тесты и развертывать веб-приложения Node.

Технология Node все чаще используется в сочетании с инструментами командной строки и настольными приложениями на базе Electron, поэтому в книгу были включены главы, посвященные обеим областям.

Предполагается, что читатель знаком с основными концепциями программирования. В первой главе приведен краткий обзор JavaScript и ES2015 для читателей, которые только начинают вникать в тонкости современного JavaScript.

Структура

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 JavaScript и Node, с приведением примеров кода. Глава 2 проведет вас поэтапно через фундаментальные концепции программирования Node.js. Глава 3 представляет собой полное руководство по построению веб-приложений с нуля.

Часть II — самая большая — посвящена разработке веб-приложений. В главе 4 рассеиваются некоторые заблуждения по поводу frond-end систем сборки; если вы когда-либо использовали webpack или Gulp в проекте, но не понимали эти технологии в полной мере, эта глава написана для вас. В главе 5 описаны некоторые популярные фреймворки стороны сервера для Node, а в главе 6 технологии Connect и Express рассматриваются более подробно.

Глава 7 посвящена языкам сценариев, которые могут значительно повысить эффективность вашей работы при написании кода на стороне сервера. Большинству веб-приложений нужна база данных, поэтому в главе 8 рассматриваются многие

разновидности баз данных, которые могут использоваться с Node, от реляционных до баз данных NoSQL. В главах 9 и 10 рассматриваются процессы тестирования и разработки, здесь также упоминается развертывание в облаке.

Часть III выходит за рамки разработки веб-приложений. Глава 11 посвящена построению приложений командной строки с использованием Node, чтобы вы могли создавать текстовые интерфейсы, удобные для разработчика. Если вас интересуют перспективы построения настольных приложений на базе Node (например, Atom), обращайтесь к главе 12 — она полностью посвящена Electron.

Также в книгу включены три подробных приложения. В приложении А приведены инструкции по установке Node для систем MacOS и Windows. Приложение Б содержит подробное руководство по извлечению веб-данных (web scraping), а в приложении В представлены все компоненты среднего звена, официально поддерживаемые для веб-фреймворка Connect.

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 поясняют 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 выравнивается по вертикали с открывающей скобкой.

Примеры кода, используемые в книге, можно загрузить с веб-сайта *www.manning.com/books/node-js-in-action-second-edition*.

Об авторах

Алекс Янг

Алекс — веб-разработчик, живущий в Лондоне; автор книги *Node.js in Practice* (Manning, 2014). Алекс ведет популярный блог DailyJS по тематике JavaScript. В настоящее время работает на Sky старшим разработчиком для NOW TV. Он есть на GitHub (*https://github.com/alexyoung*) и в Twitter (*@alex_young*).

Брэдли Мек

Брэдли — участник TC39 и Node.js Foundation. В свободное время он разрабатывает инструментарий JavaScript, занимается садоводством и преподает. До работы в GoDaddy он долго использовал Node.js на благо других компаний, таких как NodeSource и Nodejitsu. Всегда готовый обучать и объяснять, он старается поддерживать у людей мотивацию, потому что учиться для него так же сложно, как и для всех остальных.

Иллюстрация на обложке

Иллюстрация на обложке второго издания книги называется «Man about Town» («Прожигатель жизни») и была позаимствована из изданного в XIX веке во Франции четырехтомного каталога Сильвена Марешаля (Sylvain Maréchal). В каталоге представлена одежда, характерная 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 явно выраженными. В наше время довольно трудно узнать жителей разных континентов, не говоря уже о жителях разных городов или регионов. Возможно, мы отказались от культурных различий в пользу более разнообразной личной жизни, и конечно, в пользу более разнообразной и стремительной технологической жизни.

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

Мы будем рады узнать ваше мнение!

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

Знакомство с Node

В наши дни Node уже можно назвать зрелой платформой веб-разработки. В главах 1–3 рассматриваются основные возможности Node, включая использование базовых модулей и npm. Также вы узнаете, как Node использует современные версии JavaScript и как построить веб-приложение с нуля. После чтения этих глав у вас сформируется хорошее понимание того, что может делать Node и как создавать ваши собственные проекты.

1 Знакомство с Node.js

Node.js — асинхронная управляемая событиями исполнительная платформа JavaScript с мощной, но компактной стандартной библиотекой. Ее сопровождением и поддержкой занимается Node.js Foundation — отраслевой консорциум с открытой моделью управления. Существует две активно поддерживаемые версии Node: текущая (Current) и пользующаяся долгосрочной поддержкой (LTS, Long Term Support). Если вы захотите больше узнать о том, как осуществляется управление Node, на официальном веб-сайте имеется достаточно подробная документация (*https://nodejs.org*).

С момента появления Node.js в 2009 году язык JavaScript прошел долгий путь от еле сносного браузерного языка до одного из важнейших языков во всех областях разработки программного обеспечения. Отчасти это изменение связано с появлением спецификации ECMAScript 2015, устранившей ряд важных недостатков в предыдущих версиях языка. Node использует JavaScript-ядро Google V8, основанное на шестой версии стандарта ECMAScript (иногда она называется ES6 и обозначается сокращением ES2015). Также на ситуацию повлияли такие инновационные технологии, как Node, React и Electron; они позволяют применять JavaScript буквально повсеместно: от сервера до браузера и в платформенных мобильных приложениях. Самые крупные компании постепенно принимают JavaScript, а компания Microsoft даже внесла свой вклад в успех Node.

В этой главе мы расскажем о технологии Node, о ее неблокирующей модели, управляемой событиями, а также о некоторых факторах, благодаря которым язык JavaScript стал отличным языком программирования общего назначения. Для начала давайте рассмотрим типичное веб-приложение Node.

1.1. Типичное веб-приложение Node

Одна из сильных сторон Node и JavaScript вообще — однопоточная модель программирования. Программные потоки (threads) являются стандартным источником

ошибок, и хотя некоторые из недавно появившихся языков программирования, включая Go и Rust, пытаются предоставить безопасные инструменты параллельного программирования, Node работает с моделью, используемой в браузере. Браузерный код представляет собой последовательность команд, которые выполняются одна за одной; код не выполняется параллельно. Для пользовательских интерфейсов такая модель не имеет смысла: пользователь не хочет дожидаться завершения медленных операций (например, обращений к данным по сети или к файлам). Для решения этой проблемы в браузерах используются события: когда пользователь щелкает на кнопке, инициируется событие, и выполняется функция, которая была определена ранее, но еще не выполнялась. Тем самым предотвращаются некоторые проблемы, встречающиеся в многопоточном программировании, включая взаимные блокировки (deadlocks) ресурсов и состояния гонки (race conditions).

1.1.1. Неблокирующий ввод/вывод

Что это означает в контексте программирования на стороне сервера? Ситуация аналогична: запросы ввода/вывода (например, обращения к диску или сетевым ресурсам) также выполняются относительно медленно, поэтому исполнительная среда не должна блокировать выполнение бизнес-логики во время чтения файлов или передачи сообщений по сети. Для этого в Node используются три концепции: события, асинхронные API, и неблокирующий ввод/вывод. *Неблокирующий* ввод/вывод — низкоуровневый термин с точки зрения программиста Node. Он означает, что программа может обратиться с запросом к сетевому ресурсу и заняться чем-то другим. А потом, когда сетевая операция будет завершена, выполняется функция обратного вызова, которая обработает результат.

На рис. 1.1 изображено типичное веб-приложение Node, использующее библиотеку веб-программирования Express для обработки заказов в магазине. Браузеры выдают запросы на приобретение продукта; приложение проверяет текущее состояние складских запасов, создает учетную запись для пользователя, отправляет квитанцию по электронной почте и возвращает HTTP-ответ в формате JSON. Одновременно могут происходить другие операции: квитанция отправляется по электронной почте, а база данных обновляется информацией от пользователя и данными заказа. По сути, перед нами прямолинейный императивный код JavaScript, но исполнительная среда работает параллельно, потому что она использует неблокирующий ввод/вывод.

На рис. 1.1 приложение обращается к базе данных по сети. В Node сетевые операции выполняются без блокировки, потому что Node при помощи библиотеки libuv (*http://libuv.org/*) использует неблокирующие сетевые вызовы операционной системы. Эта функциональность по-разному реализована для Linux, macOS и Windows, но вам придется иметь дело только с удобной библиотекой JavaScript для работы с базами данных. Хотя вы пишете команды типа `db.insert(query, err => {})`, Node во внутренней реализации выполняет оптимизированные неблокирующие сетевые операции.

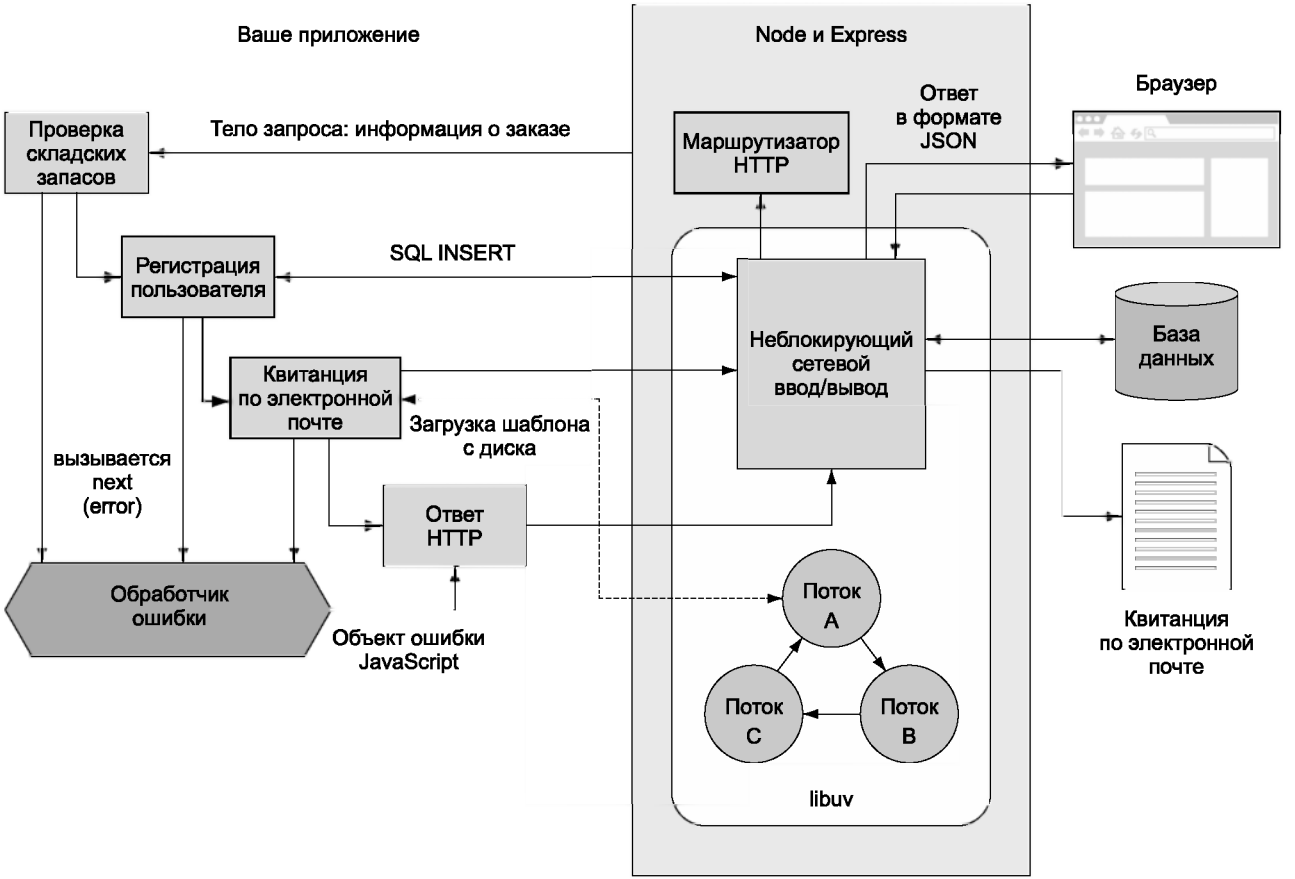


Рис. 1.1. Асинхронные и неблокирующие компоненты в приложениях Node

Обращения к диску происходят примерно так же, но, как ни странно, полного совпадения нет. Когда приложение генерирует квитанцию, отправляемую по электронной почте, и шаблон сообщения читается с диска, libuv использует пул потоков для создания иллюзии использования неблокирующего вызова. Управление пулом потоков — довольно тяжелое дело, но понять команду `email.send('template.ejs', (err, html) => {})` определенно намного проще.

Истинное преимущество использования асинхронных API с неблокирующими операциями ввода/вывода заключается в том, что Node может заниматься другими делами во время выполнения относительно медленных процессов. И хотя выполняться может только однопоточное и однопроцессное веб-приложение Node, в любой момент времени оно может обрабатывать сразу несколько подключений от тысяч потенциальных посетителей сайта. Чтобы понять, как это происходит, необходимо познакомиться с циклом событий.

1.1.2. Цикл событий

А теперь сосредоточимся на одном конкретном аспекте рис. 1.1: обработке запросов браузера. В этом приложении встроенная библиотека HTTP-сервера Node — базовый модуль с именем `http.Server` — обрабатывает запрос с использованием потоков, событий и парсера HTTP-запросов Node, который содержит платформенный код. При этом инициируется выполнение функции обратного вызова в вашем приложении, которая была добавлена средствами библиотеки веб-приложений Express (*https://expressjs.com/*). Выполняемая функция обратного вызова выдает запрос к базе данных, и в конечном итоге приложение отвечает данными JSON с использованием HTTP. Весь процесс использует как минимум три неблокируемых сетевых вызова: один для запроса, один для базы данных и один для ответа. Как Node планирует все эти неблокирующие сетевые операции? Ответ: при помощи цикла событий. На рис. 1.2 показано, как цикл событий используется для этих трех сетевых операций.

Цикл событий работает в одном направлении (он реализуется очередью FIFO — «первым пришел, первым вышел») и проходит через несколько фаз. На рис. 1.2 показана упрощенная последовательность важнейших фаз, выполняемых при каждой итерации цикла. Сначала выполняются таймеры, выполнение которых запланировано функциями JavaScript `setTimeout` и `setInterval`. Затем выполняются обратные вызовы ввода/вывода, поэтому, если один из неблокирующих сетевых вызовов вернул какой-либо ввод/вывод, в этот момент будет инициирован ваш обратный вызов. В фазе опроса читаются новые события ввода/вывода, а в конце инициируются обратные вызовы, запланированные функцией `setImmediate`. Это особый случай, потому что он позволяет запланировать немедленное выполнение обратных вызовов после текущих обратных вызовов ввода/вывода, уже находящихся в очереди. Пока это звучит абстрактно, но сейчас вы должны уяснить одно: Node использует однопоточную модель, но при этом предоставляет необходимые средства для написания эффективного и масштабируемого кода.

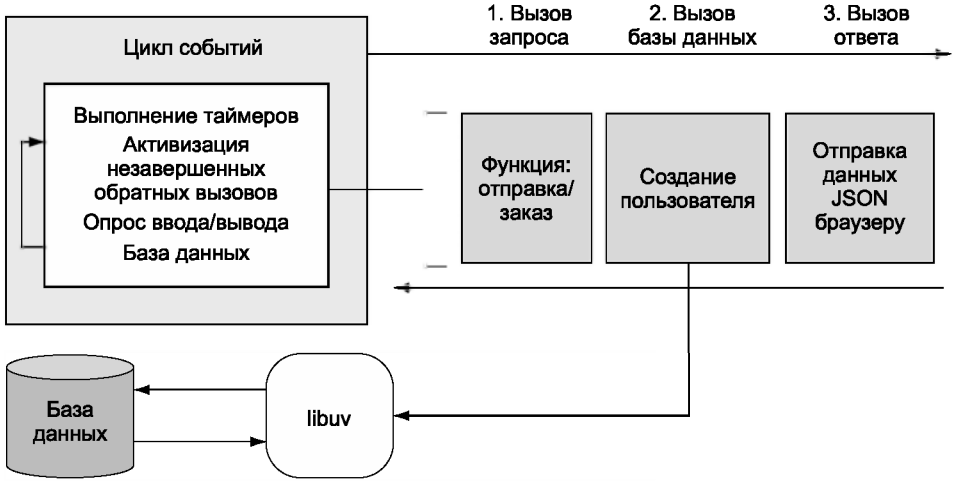


Рис. 1.2. Цикл событий

Обратите внимание на то, что в примерах используются «стрелочные» функции ES2015. Node поддерживает много новых функций JavaScript, и прежде чем двигаться дальше, мы рассмотрим некоторые новые языковые средства, используемые для написания более эффективного кода.

1.2. ES2015, Node и V8

Если вы когда-либо работали с JavaScript и вас огорчило отсутствие классов и странные правила видимости — радуйтесь: в Node многие проблемы были решены! Теперь вы можете создавать классы и использовать конструкции видимости `const` и `let` (вместо `var`). В Node 6 можно использовать параметры функций по умолчанию, оставшиеся параметры (rest parameters), оператор `spread`, циклы `for...of`, шаблонные строки, деструктуризацию, генераторы и многое другое. Замечательная сводка новых возможностей ES2015 в Node доступна по адресу *http://node.green*.

Начнем с классов. В ES5 и более ранних версиях для создания конструкций, напоминающих классы, использовались объекты-прототипы:

```
function User() {
  // Конструктор
}

User.prototype.method = function() {
  // Метод
};
```

В Node 6 и ES2015 тот же код можно написать с использованием классов:

```
class User {
  constructor() {}
  method() {}
}
```

Код получается более компактным и лучше читается. Но это еще не все: Node также поддерживает субклассирование, `super` и статические методы. Для пользователей с опытом работы на других языках поддержка синтаксиса классов делает технологию Node более доступной по сравнению с временами, когда мы были ограничены возможностями ES5.

Другая важная возможность Node 4 и выше — добавление конструкций `const` и `let`. В ES5 все переменные создавались с ключевым словом `var`. Проблема в том, что `var` определяет переменные в области видимости функции или глобальной области видимости, поэтому вы не могли определить переменную уровня блока в команде `if`, `for` или в другом блоке.

CONST ИЛИ LET?

Когда вы колеблетесь между const и let, почти всегда следует сделать выбор в пользу const. Так как в большей части вашего кода будут использоваться экземпляры ваших собственных классов, объектные литералы или значения, которые не должны изменяться, почти во всех случаях уместно const. Даже экземпляры объектов с изменяемыми свойствами могут объявляться с ключевым словом const — ведь const означает лишь то, что ссылка доступна только для чтения, а не неизменность самого значения.

В Node также поддерживаются обещания (promises) и генераторы. *Обещания* поддерживаются многими библиотеками, что позволяет писать асинхронный код в стиле динамических интерфейсов (fluent interface). Вероятно, вы уже знакомы с динамическими интерфейсами: их видел каждый, кто когда-либо использовал такие API, как jQuery, или хотя бы массивы JavaScript. Следующий короткий пример демонстрирует применение сцепленных вызовов для манипуляций с массивами в JavaScript:

```
[1, 2, 3]
  .map(n => n * 2)
  .filter(n => n > 3);
```

Генераторы нужны для использования синхронного стиля программирования с асинхронным вводом/выводом. Если вас интересует практический пример использования генераторов в Node, обратитесь к библиотеке веб-приложений

Koa (*http://koajs.com/*). При использовании обещаний или других генераторов с Koa вы можете прибегнуть к `yield` со значениями вместо вложения обратных вызовов.

Еще одна полезная возможность ES2015 в Node — *шаблонные строки*. В ES5 строковые литералы не поддерживали интерполяцию или деление на несколько строк. Теперь при использовании символа обратного апострофа (`) можно вставлять значения и использовать разбиения по строкам. Данная возможность особенно полезна при вставке коротких фрагментов HTML в веб-приложениях:

```
this.body = `
  <div>
    <h1>Hello from Node</h1>
    <p>Welcome, ${user.name}!</p>
  </div>
`;
```

В ES5 этот пример пришлось бы записывать следующим образом:

```
this.body = '\n';
this.body += '<div>\n';
this.body += '  <h1>Hello from Node</h1>\n';
this.body += '  <p>Welcome, ' + user.name + '</p>\n';
this.body += '<div>\n';
```

Старый стиль не только занимает больше места, но и повышает риск случайных ошибок. Последняя новая возможность, особенно важная для программистов Node, — *стрелочные функции*, которые способствуют упрощению синтаксиса. Например, если вы пишете функцию обратного вызова, которая получает один аргумент и возвращает значение, трудно представить себе более компактный синтаксис:

```
[1, 2, 3].map(v => v * 2);
```

В Node обычно используются два аргумента, потому что первым аргументом функции обратного вызова часто является объект ошибки. В этом случае аргументы должны заключаться в круглые скобки:

```
const fs = require('fs');
fs.readFile('package.json',
  (err, text) => console.log('Length:', text.length)
);
```

Если тело функции должно содержать более одной строки, необходимо использовать фигурные скобки. Полезность стрелочных функций не ограничивается упрощением синтаксиса; она также имеет отношение к областям видимости JavaScript. В ES5 и ранее при определении функций внутри других функций ссылка `this` становится

глобальным объектом. Из-за этого в следующем классе в стиле ES5 присутствует скрытая ошибка:

```
function User(id) {

// Конструктор
  this.id = id;
}

User.prototype.load = function() {
  var self = this;
  var query = 'SELECT * FROM users WHERE id = ?';
  sql.query(query, this.id, function(err, users) {
self.name = users[0].name;
  });
};
```

В строке, в которой присваивается значение `self.name`, невозможно использовать запись `this.name`, потому что `this` внутри функции будет глобальным объектом. Когда-то применялось обходное решение с сохранением значения `this` в точке входа родительской функции или метода. Со стрелочными функциями связывание выполняется правильно. В ES2015 предыдущий пример можно записать в гораздо более естественном виде:

```
class User {
  constructor(id) {
    this.id = id;
  }

  load() {
    const query = 'SELECT * FROM users WHERE id = ?';
    sql.query(query, this.id, (err, users) => {
    this.name = users[0].name;
    });
}
}
```

Вы не только можете использовать `const` для более эффективного моделирования запроса к базе данных, но и отпадает необходимость в неуклюжей переменной `self`. В ES2015 появилось много новых замечательных возможностей, благодаря которым код Node лучше читается; но давайте посмотрим, на чем строится их реализация в Node и как они связаны с уже рассмотренными средствами неблокирующего ввода/вывода.

1.2.1. Node и V8

Технология Node основана на ядре JavaScript V8, разработанном проектом Chromium для Google Chrome. К отличительным особенностям V8 относится прямая компиляция в машинный код и средства оптимизации, обеспечивающие высокую скорость

работы Node. В разделе 1.1.1 был упомянут еще один из встроенных компонентов Node — libuv. В этой части речь идет о вводе/выводе; V8 обеспечивает интерпретацию и выполнение кода JavaScript. Чтобы использовать libuv с V8, следует применить *связующую прослойку* C++. На рис. 1.3 изображены все программные компоненты, из которых состоит Node.

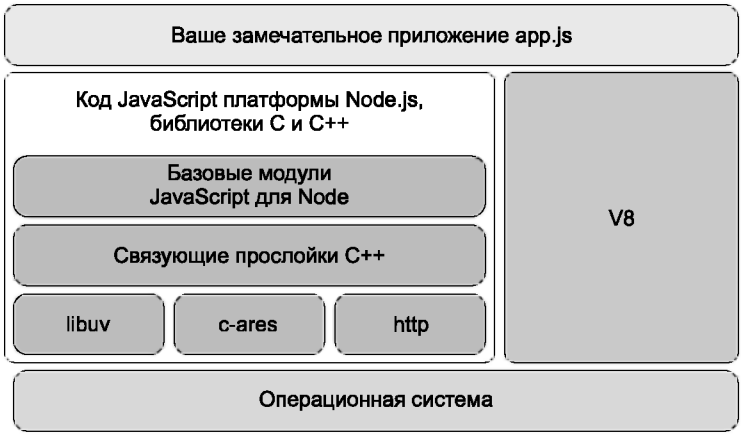


Рис. 1.3. Программный стек Node

Таким образом, конкретная функциональность JavaScript, доступная для Node, определяется набором возможностей, поддерживаемых V8. Управление этой поддержкой осуществляется через функциональные группы.

1.2.2. Работа с функциональными группами

Node включает функциональность ES2015, основанную на возможностях, поддерживаемых V8. Функциональность группируется по категориям: *завершенные*, *запланированные* и *находящиеся в разработке*. Завершенные возможности включаются по умолчанию, но для включения запланированных и находящихся в разработке возможностей используются флаги командной строки. Если вы хотите использовать запланированные возможности, которые почти готовы, но не считаются завершенными группой V8, запустите Node с флагом `--harmony`. Возможности, находящиеся в разработке, менее стабильны, а для их включения используются специальные флаги. Для получения списка всех возможностей Node, находящихся в разработке, в документации Node рекомендуется провести поиск по строке «in progress»:

```
node --v8-options | grep "in progress"
```

Список будет изменяться в зависимости от выпусков Node. У Node также имеется график управления версиями, определяющий доступные API.

1.2.3. График выпуска версий Node

Выпуски Node делятся на пользующиеся долгосрочной поддержкой, или LTS (Long-Term Support), текущие (Current) и ежедневные (Nightly). Для выпусков LTS обеспечивается 18-месячная поддержка с последующей 12-месячной поддержкой сопровождения. Для выпусков применяется система семантического управления версиями (SemVer). В этой системе выпуску назначается основной (major), дополнительный (minor) и оперативный (patch) номера версии. Например, у выпуска 6.9.1 имеется основной номер версии 6, дополнительный номер 9 и оперативный 1. Каждый раз, когда у Node изменяется основной номер версии, это может означать, что некоторые API стали несовместимыми с вашими проектами и вам придется заново протестировать их для новой версии Node. Кроме того, в терминологии выпуска Node увеличение основного номера версии означает появление новой текущей версии. Ежедневные сборки автоматически генерируются каждые 24 часа с новейшими изменениями, но, как правило, используются только для тестирования новой функциональности Node.

Выбор версии зависит от проекта и организации. Одни предпочитают LTS из-за более редких обновлений: этот вариант хорошо подходит для крупных организаций, в которых частые обновления создают проблемы. Но если вас интересуют новейшие усовершенствования в области быстродействия и функциональности, выбирайте текущую версию.

1.3. Установка Node

Простейший способ установить Node — использовать программу установки с сайта *https://nodejs.org*. Новейшая текущая версия (6.5 на момент написания книги) устанавливается средствами Mac или Windows. Загрузите исходный код самостоятельно или выполните установку при помощи менеджера пакетов вашей операционной системы. Такие пакеты существуют для Debian, Ubuntu, Arch, Fedora, FreeBSD, Gentoo и SUSE. Также имеются пакеты для Homebrew и SmartOS. Если пакета для вашей операционной системы нет, проведите построение из исходного кода.

ПРИМЕЧАНИЕ

Дополнительная информация об установке Node приведена в приложении А.

Полный список пакетов находится на сайте Node (*https://nodejs.org/en/download/package-manager/*), а исходный код доступен на GitHub (*https://github.com/nodejs/node*). Исходный код также пригодится вам в том случае, если вы хотите заняться изучением кода Node без его загрузки.

После того как платформа Node будет установлена, вы можете немедленно проверить результат — введите команду `node -v` в терминале. Команда должна вывести номер

только что загруженной и установленной версии Node. Создайте файл с именем hello.js, содержимое которого должно выглядеть так:

```
console.log("hello from Node");
```

Сохраните файл и запустите его командой `node hello.js`. Поздравляем! Теперь вы готовы к тому, чтобы заняться написанием собственных приложений на базе Node.

БЫСТРОЕ НАЧАЛО РАБОТЫ В WINDOWS, LINUX И MACOS

Если вы недавно занялись программированием и у вас еще нет любимого текстового редактора, для Node хорошо подойдет Visual Studio Code (https://code.visualstudio.com/). Этот редактор создан компанией Microsoft, но он распространяется с открытым кодом, доступен для бесплатной загрузки и поддерживает Windows, Linux и macOS.

Начинающие программисты оценят такие возможности Visual Studio Code, как цветовое выделение синтаксиса JavaScript и автозавершение для базовых модулей Node; ваш код JavaScript будет выглядеть более четко, и вы будете видеть списки поддерживаемых методов и объектов в процессе ввода. Вы также можете открыть интерфейс командной строки, в котором Node можно вызвать простым вводом команды Node. Этот способ особенно полезен для выполнения Node и команд npm. Возможно, пользователи Windows предпочтут его использованию cmd.exe. Все листинги были протестированы в Windows и Visual Studio Code, поэтому для выполнения примеров ничего особенного вам не потребуется.

Возможно, в начале работы вам стоит познакомиться с учебником по Node.js для Visual Studio Code (https://code.visualstudio.com/docs/runtimes/nodejs).

Когда вы установите Node, в вашем распоряжении также окажутся некоторые бесплатные инструменты. Node — не просто интерпретатор: это целое семейство инструментов, образующих платформу Node. В следующем разделе инструменты, включенные в поставку Node, рассматриваются более подробно.

1.4. Встроенные средства Node

В поставку Node входит встроенный менеджер пакетов, базовые модули JavaScript для поддержки самых разнообразных функций, от файлового и сетевого ввода/вывода до сжатия zlib, а также отладчик. Менеджер пакетов npm является важнейшим компонентом инфраструктуры, поэтому мы рассмотрим его более подробно.

Если вы хотите убедиться в том, что установка Node была выполнена правильно, выполните команды `node -v` и `npm -v` в командной строке. Эти команды выводят номера только что установленных версий Node и npm.

1.4.1. npm

Чтобы запустить программу командной строки npm, введите команду `npm`. Программа может использоваться для установки пакетов из центрального реестра npm, но также и для поиска и организации совместного использования ваших проектов с открытым или закрытым кодом. У каждого пакета npm в реестре существует сайт, на котором хранится Readme-файл, информация об авторе и статистика о загрузках.

Впрочем, это описание нельзя назвать исчерпывающим. Программой npm занимается npm, Inc. — компания, обеспечивающая работу сервиса npm и предоставляющая услуги коммерческим организациям. Сюда входит хостинг приватных пакетов npm; вы также можете вносить ежемесячную плату за хостинг исходного кода, принадлежащего вашей компании, чтобы разработчики JavaScript могли легко устанавливать его с использованием npm.

При установке пакетов командой `npm install` вам придется решить, хотите ли вы добавить их в свой текущий проект или установить их глобально. Глобальная установка пакетов обычно используется для служебных программ — чаще всего это программы, запускаемые из командной строки. Хорошим примером служит пакет gulp-cli.

Чтобы использовать npm, создайте файл **package.json** в каталоге, содержащем проект Node. Проще всего поручить создание файла **package.json** менеджеру npm.

Введите следующую команду в командной строке:

```
mkdir example-project
cd example-project
npm init -y
```

Открыв файл **package.json**, вы увидите простые данные в формате JSON с описанием вашего проекта. Если теперь установить модуль с сайта *www.npmjs.com* с ключом `--save`, npm автоматически обновит файл **package.json**. Попробуйте сами — введите команду `npm install`, или сокращенно `npm i`:

```
npm i --save express
```

Если теперь открыть файл **package.json**, вы увидите, что в свойстве `dependencies` добавился пакет `express`. Кроме того, в папке **node_modules** появился каталог **express**. Он содержит только что установленную версию Express. Вы также можете установить модули глобально с ключом `--global`. Старайтесь по возможности использовать локальные модули, но глобальные модули могут быть полезны для инструментов командной строки, которые должны использоваться за пределами JavaScript-кода Node. В качестве примера средства командной строки, устанавливаемого из npm, можно привести ESLint (*http://eslint.org/*).

Начиная работать с Node, вы часто будете пользоваться пакетами из npm. Node поставляется с множеством полезных встроенных библиотек, которые обычно называются *базовыми модулями*. Рассмотрим их более подробно.

1.4.2. Базовые модули

Базовые модули Node напоминают стандартные библиотеки других языков; эти инструменты понадобятся вам для написания кода JavaScript на стороне сервера. Сами стандарты JavaScript не включают никаких средств для работы с сетью — и даже файлового ввода/вывода в том виде, в котором его представляет большинство разработчиков серверного кода. Чтобы Node можно было использовать для серверного программирования, в эту платформу необходимо было добавить средства для работы с файлами и сетевых операций TCP/IP.

Файловая система

В поставку Node включается библиотека файловой системы (fs, path), клиенты и серверы TCP (net), поддержка HTTP (http и https) и разрешения доменных имен (dns). Также имеется полезная библиотека, которая используется в основном для написания тестов (assert), и библиотека операционной системы для запроса информации о платформе (os).

Node также содержит ряд уникальных библиотек. Модуль events — небольшая библиотека для работы с событиями — используется в качестве основы для многих API Node. Например, модуль stream использует модуль events для предоставления абстрактных интерфейсов для работы с потоками данных. Поскольку все потоки данных в Node используют одни и те же API, вы можете легко составлять программные компоненты; если у вас имеется объект чтения файлового потока, вы можете направить его через преобразование zlib, которое выполняет сжатие данных, а затем через объект записи в файловый поток для записи данных в файл.

В следующем листинге продемонстрировано использование модуля Node fs для создания потоков чтения и записи, которые могут направляться через другой поток (gzip) для преобразования данных — в данном случае их сжатия.

Листинг 1.1. Использование базовых модулей и потоков

```
const fs = require('fs');
const zlib = require('zlib');
const gzip = zlib.createGzip();
const outStream = fs.createWriteStream('output.js.gz');

fs.createReadStream('./node-stream.js')
  .pipe(gzip)
  .pipe(outStream);
```

Сетевые операции

Существует распространенное мнение, что создание простого сервера HTTP было настоящим примером программы «Hello World» для Node. Чтобы построить сервер на базе Node, необходимо загрузить модуль http и передать ему функцию. Функция

получает два аргумента: входной запрос и выходной ответ. В листинге 1.2 приведен пример, который вы можете выполнить в терминале.

Листинг 1.2. Программа «Hello World» с использованием модуля http для Node

```
const http = require('http');
const port = 8080;

const server = http.createServer((req, res) => {
  res.end('Hello, world.');
});

server.listen(port, () => {
  console.log('Server listening on: http://localhost:%s', port);
});
```

Сохраните листинг 1.2 в файле hello.js и запустите его командой `node hello.js`. Открыв адрес *http://localhost:8080*, вы увидите сообщение из строки 4.

Базовые модули Node предоставляют минимальную функциональность, но при этом они достаточно мощны. Часто многие задачи удается решить простым использованием этих модулей, даже без установки дополнительных пакетов из npm. За дополнительной информацией о базовых модулях обращайтесь по адресу *https://nodejs.org/api/*.

Последний встроенный инструмент Node — отладчик. В следующем разделе рассматривается пример его практического применения.

1.4.3. Отладчик

В поставку Node включается отладчик с поддержкой пошагового выполнения и REPL (Read-Eval-Print Loop). Работа отладчика основана на взаимодействии с вашей программой по сетевому протоколу. Чтобы запустить программу в отладчике, укажите аргумент `debug` в командной строке.

Предположим, вы отлаживаете код из листинга 1.2:

```
node debug hello.js
```

Результат должен выглядеть так:

```
< Debugger listening on [::]:5858
connecting to 127.0.0.1:5858 ... ok
break in node-http.js:1
> 1 const http = require('http');
  2 const port = 8080;
  3
```

Среда Node запускает вашу программу и активизирует ее отладку подключением к порту 5858. Теперь вы можете ввести команду `help`, чтобы просмотреть список

доступных команд, а затем команду `c` для продолжения выполнения программы. Node всегда запускает программу в *прерванном* состоянии, поэтому вам всегда придется продолжать ее выполнение, прежде чем делать что-либо другое.

Вы можете прервать выполнение отладчика, включив команду `debugger` в любую позицию кода. При обнаружении команды `debugger` отладчик останавливается, и вы можете вводить команды. Представьте, что вы написали REST API для создания учетных записей новых пользователей, а код создания не сохраняет хеш пароля нового пользователя в базе данных. Включите команду `debugger` в метод `save` класса `User`, выполните в пошаговом режиме каждую команду и посмотрите, что происходит.

ИНТЕРАКТИВНАЯ ОТЛАДКА

Node поддерживает отладочный протокол Chrome. Чтобы отладить сценарий с использованием средств разработчика Chrome, укажите флаг --inspect при запуске программы:

```
node --inspect --debug-brk
```

Node запускает отладчик и прерывает выполнение в первой строке. На консоль выводится URL-адрес; откройте его в Chrome для использования встроенного отладчика Chrome. Отладчик Chrome позволяет выполнять код строку за строкой, при этом он выводит значения всех переменных и объектов. Этот способ намного удобнее ввода команды console log.

Отладка более подробно рассматривается в главе 9. Если вы захотите опробовать ее прямо сейчас, лучше всего начать с описания отладки в документации Node (*https://nodejs.org/api/debugger.html*).

До настоящего момента мы говорили о том, как работает платформа Node и какие возможности она предоставляет разработчикам. Вероятно, вы с нетерпением ждете рассказа о том, что можно сделать с помощью Node на практике. В следующем разделе рассматриваются разные типы программ, создаваемых на платформе Node.

1.5. Три основных типа программ Node

Программы Node делятся на три основных типа: веб-приложения, средства командной строки и демоны и настольные приложения. К категории веб-приложений относятся простые приложения, предоставляющие одностраничные приложения, микрослужбы REST и веб-приложения полного стека. Возможно, вы уже использовали средства командной строки, написанные с использованием Node, например npm, Gulp и webpack. *Демоны* (daemons) представляют собой фоновые службы. Хорошим примером служит менеджер процессов PM2 (*www.npmjs.com/package/pm2*).

Настольные приложения обычно строятся с применением фреймворка Electron (*http://electron.atom.io/*), использующего Node во внутренней реализации для настольных веб-приложений. Примеры такого рода — текстовые редакторы Atom (*https://atom.io/*) и Visual Studio Code (*https://code.visualstudio.com/*).

1.5.1. Веб-приложения

Технология Node предназначена для работы с JavaScript на стороне сервера, поэтому ее выбор в качестве платформы для построения веб-приложений выглядит логично. Выполнение JavaScript на стороне клиента и сервера открывает возможности для повторного использования кода в разных средах. Веб-приложения Node обычно пишутся с применением таких фреймворков, как Express (*http://expressjs.com/*). В главе 6 рассматриваются основные серверные фреймворки, доступные для Node. Глава 7 посвящена Express и Connect, а темой главы 8 станет шаблонизация веб-приложений.

Чтобы создать простейшее веб-приложение Express, создайте новый каталог и установите модуль Express:

```
mkdir hello_express
cd hello_express
npm init -y
npm i express --save
```

Теперь включите следующий код JavaScript в файл с именем server.js.

Листинг 1.3. Веб-приложение Node

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

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Express web app on localhost:3000');
});
```

Введите команду `npm start`, и в вашей системе появляется веб-сервер Node, работающий на порту 3000. Открыв адрес *http://localhost:3000* в браузере, вы увидите текст из строки `res.send`.

Node также играет важную роль в мире разработки интерфейсов, потому что это основной инструмент, используемый при транспиляции других языков (например, TypeScript в JavaScript). Транспиляторы преобразуют код, написанный на одном высокоуровневом языке, на другой язык; в этом отношении они отличаются от традиционных компиляторов, которые преобразуют код с высокоуровневого языка на

низкоуровневый. Глава 4 посвящена системам фронтэнда; в ней рассматриваются сценарии npm, Gulp и webpack.

Не вся веб-разработка связана с построением веб-приложений. Иногда разработчику приходится решать такие задачи, как извлечение данных со старого сайта, которые должны использоваться при его обновлении. Приложение Б полностью посвящено извлечению веб-данных для демонстрации возможностей исполнительной среды JavaScript в Node с моделью DOM (Document Objet Model), а также использования Node за пределами привычной «комфортной зоны» типичных веб-приложений Express. Если же вы просто хотите быстро создать простейшее веб-приложение, в главе 3 приведено обучающее руководство по построению веб-приложений Node.

1.5.2. Средства командной строки и демоны

Node используется для написания средств командной строки, таких как менеджеры процессов и транспиляторы JavaScript, используемые разработчиками JavaScript. Однако Node также может стать удобной платформой для написания удобных средств командной строки, которые выполняют другие операции, включая преобразование графики, и сценариев для управления воспроизведением мультимедийных материалов.

Ниже приведен простой пример, который вы можете опробовать. Создайте новый файл с именем **cli.js** и добавьте в него следующие строки:

```
const [nodePath, scriptPath, name] = process.argv;
console.log('Hello', name);
```

Запустите сценарий командой `cli.js yourName`; вы увидите сообщение `Hello yourName`. В сценарии функциональность деструктуризации ES2015 используется для извлечения третьего аргумента из `process.argv`. Объект `process` доступен в любой программе; с его помощью программа получает аргументы, заданные пользователем при запуске.

У программ командной строки Node также имеются полезные возможности. Если добавить в начало программы строку, которая начинается с `#!`, и предоставить файлу разрешение на исполнение (`chmod +x cli.js`), вы сможете заставить командный интерпретатор использовать Node при запуске программы. После этого программы Node можно будет запускать точно так же, как любые другие сценарии командного интерпретатора. Для систем семейства Unix добавляется следующая строка:

```
#!/usr/bin/env node
```

При таком использовании Node вы сможете заменить свои сценарии командного интерпретатора командами Node. Это означает, что Node можно будет использовать с любыми другими инструментами командной строки, включая фоновые

программы. Программы Node могут запускаться при помощи cron или работать в фоновом режиме как демоны.

Если все эти термины вам незнакомы, не беспокойтесь: глава 11 познакомит вас с написанием программ командной строки, и вы узнаете, как программы этого типа помогают Node проявить свои сильные стороны. Например, в программах командной строки широко используются потоки данных (streams) как универсальный API, а потоки также относятся к числу самых полезных возможностей Node.

1.5.3. Настольные приложения

Если вы работали с текстовым редактором Atom или Visual Studio Code — знайте, что вы все это время использовали Node. Фреймворк Electron использует Node для реализации своих операций, поэтому каждый раз, когда требуется выполнить ввод/вывод с диском или сетью, Electron обращается к Node. Также Electron использует Node для управления зависимостями; это означает, что вы сможете добавлять пакеты из npm в проекты Electron.

Чтобы быстро опробовать Electron на практике, клонируйте репозиторий Electron и запустите приложение:

```
git clone https://github.com/electron/electron-quick-start
cd electron-quick-start
npm install && npm start
curl localhost:8081
```

О том, как написать приложение на базе Electron, рассказано в главе 12.

1.5.4. Приложения, хорошо подходящие для Node

Мы рассмотрели некоторые разновидности приложений, которые можно построить с Node, но существуют некоторые типы приложений, в которых Node особенно заметно превосходит конкурентов. Node обычно используется для создания веб-приложений реального времени; к этой категории может относиться что угодно, от пользовательских приложений (скажем, чат-серверов) до внутренних систем сбора аналитики. Так как функции в JavaScript являются полноправными объектами, а в Node имеется встроенная модель событий, написание асинхронных программ реального времени проходит более естественно, чем в других языках сценариев.

Если вы строите традиционные веб-приложения MVC (Model-View-Controller), Node хорошо подходит и для них. На базе Node строятся многие популярные системы ведения блогов, например Ghost (*https://ghost.org/*); платформа Node хорошо зарекомендовала себя для подобных веб-приложений. Стиль разработки отличается от системы WordPress, построенной с использованием PHP, но Ghost поддерживает

многие аналогичные возможности, включая шаблоны и многопользовательские средства администрирования.

Node также может делать то, что в других языках делается намного сложнее. Node базируется на JavaScript, поэтому в Node возможно выполнение браузерного кода JavaScript. Сложные клиентские приложения могут адаптироваться для выполнения на сервере Node, что позволяет серверам заранее генерировать веб-приложения; это сокращает время визуализации страниц в браузере и упрощает работу поисковых систем.

Наконец, если вы занимаетесь построением настольных или мобильных приложений, опробуйте фреймворк Electron, работающий на базе Node. В наши дни, когда веб-интерфейсы пользователя не уступают настольным, настольные приложения Electron могут конкурировать с платформенными веб-приложениями при сокращенном времени разработки. Electron также поддерживает три основные платформы, поэтому ваш код будет работать в Windows, Linux и macOS.

1.6. Заключение

- Node — неблокирующая, управляемая событиями платформа для построения приложений JavaScript.
- В качестве исполнительной среды JavaScript используется ядро V8.
- Библиотека libuv обеспечивает быстрый кросс-платформенный неблокирующий ввод/вывод.
- Node содержит небольшую стандартную библиотеку — базовые модули, которые добавляют в JavaScript поддержку сетевого и дискового ввода/вывода.
- В поставку Node входит отладчик и менеджер зависимостей (npm).
- Node используется для построения веб-приложений, средств командной строки и даже настольных приложений.

2 Основы программирования Node

В отличие от многих платформ с открытым кодом, Node легко настраивается и не предъявляет особых требований к памяти и дисковому пространству. Программирование не требует применения сложных интегрированных средств разработки или систем построения. Тем не менее некоторые фундаментальные знания сильно помогут вам в начале работы. В этой главе рассматриваются две основные проблемы, с которыми сталкиваются начинающие разработчики Node:

- организация кода;
- асинхронное программирование.

В этой главе будут представлены важные приемы асинхронного программирования, которые позволят вам держать под контролем процесс выполнения вашего приложения. Вы научитесь:

- реагировать на одноразовые события;
- реагировать на повторяющиеся события;
- определять последовательность отработки асинхронной логики.

Но для начала посмотрим, как для решения проблемы организации кода используются *модули* — основной механизм Node для организации кода и его упаковки для удобства повторного использования.

2.1. Структурирование и повторное использование функциональности Node

При создании приложения, с использованием Node или без, часто наступает момент, когда хранить весь код в одном файле становится слишком неудобно. В традиционном решении этой проблемы, представленном на рис. 2.1, разработчик брал файл с большим объемом кода и разбивал его на отдельные файлы.

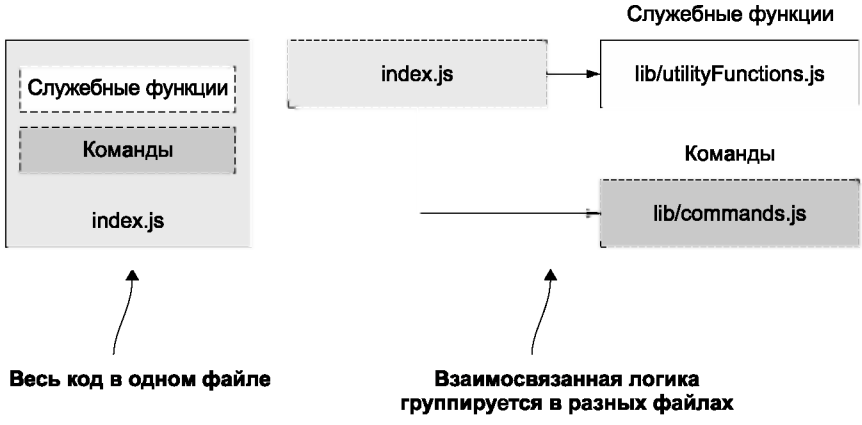


Рис. 2.1. В коде удобнее ориентироваться, если он разбит по разным каталогам и файлам, а не хранится в одном огромном файле

В реализациях некоторых языков (таких как PHP и Ruby) внедрение логики из другого файла (*включаемого*) означает, что вся логика, выполняемая в файле, влияет на глобальную область видимости. Все создаваемые переменные и все функции, объявленные во включаемом файле, заменяют переменные и функции, созданные и объявленные приложением.

Предположим, вы программируете на PHP, и ваше приложение содержит следующую логику:

```
function uppercase_trim($text) {
  return trim(strtoupper($text));
}
include('string_handlers.php');
```

Если файл string_handlers.php также попытается определить функцию `uppercase_trim`, вы получите сообщение об ошибке:

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

В PHP эта проблема решается за счет использования *пространств имен*; аналогичную функциональность Ruby предлагает в виде *модулей*. Однако Node в принципе избегает эту потенциальную проблему, не предоставляя простых возможностей для случайного загрязнения глобального пространства имен.

ПРОСТРАНСТВА ИМЕН PHP, МОДУЛИ RUBY

Пространства имен PHP описаны в руководстве по языку PHP по адресу http://php.net/manual/en/language.namespaces.php. Модули Ruby рассматриваются в документации Ruby: http://ruby-doc.org/core-2.3.1/Module.html.

Модули Node упаковывают код для повторного использования, но при этом не изменяют глобальную область видимости. Предположим, вы разрабатываете систему управления контентом (CMS) с открытым кодом на языке PHP и хотите использовать стороннюю библиотеку API, которая не использует пространства имен. Библиотека может содержать класс с таким же именем, как у вашего приложения, и этот класс нарушит работу вашего приложения, если только вы не переименуете свой класс в приложении или библиотеке. Однако изменение имени класса в приложении может создать проблемы у других разработчиков, которые пользуются вашей CMS-системой в своих проектах. После переименования класса в библиотеке вам придется помнить о необходимости повторять этот трюк каждый раз, когда вы обновляете библиотеку в дереве исходного кода вашего приложения. Конфликты имен — проблема, которую лучше избегать.

Имена модулей позволяют выбрать, какие функции и переменные из включенного файла будут доступны для приложения. Если модуль возвращает более одной функции или переменной, модуль может указать их заданием свойств объекта с именем `exports`. Если модуль возвращает одну функцию или переменную, вместо этого можно задать свойство `module.exports`. На рис. 2.2 показано, как работает эта схема.

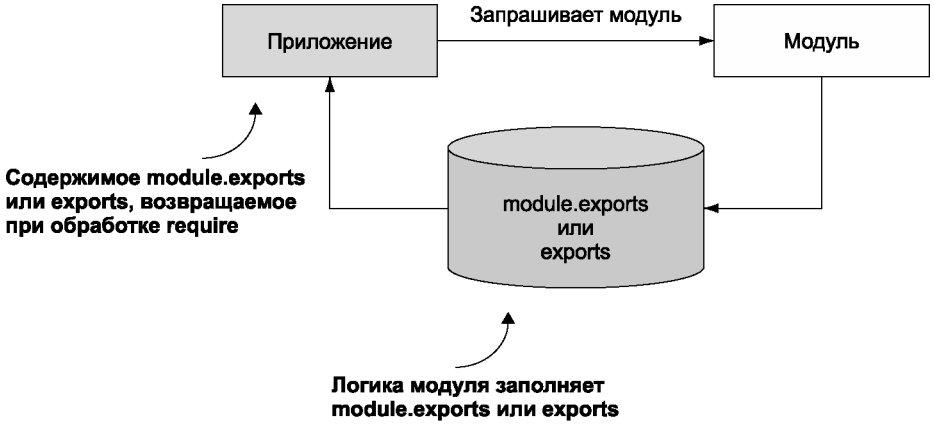


Рис. 2.2. Заполнение свойства module.exports или объекта exports позволяет модулю выбрать, какая информация должна быть доступна приложению

Если описание кажется запутанным, не беспокойтесь; мы разберем некоторые примеры в этой главе. Избегая загрязнения глобальной области видимости, система модулей Node предотвращает конфликты имен или упрощает повторное использование кода. После этого модули могут быть опубликованы в реестре npm (менеджер пакетов), сетевой подборке готовых модулей Node, и распространяться в сообществе Node. При этом пользователям модулей не нужно беспокоиться о том, что модуль случайно заменит переменные и функции другого модуля.

Чтобы помочь вам в организации логики по модулям, мы рассмотрим следующие вопросы:

- как создаются модули;
- как модули хранятся в файловой системе;
- о чем нужно знать при создании и использовании модулей.

Чтобы с ходу взяться за изучение системы модулей Node, мы создадим новый проект Node, а затем определим для него простой модуль.

2.2. Создание нового проекта Node

Создать новый проект Node несложно: создайте папку и выполните команду `npm init`. И все! Команда `npm` задаст несколько вопросов, вы можете ответить на все вопросы утвердительно. Полный пример:

```
mkdir my_module
cd my_module
npm init -y
```

Флаг `-y` означает подтверждение (*yes*). Таким образом, `npm` создаст файл package.json со значениями по умолчанию. Если вы хотите полностью управлять процессом создания проекта, опустите флаг `-y`; `npm` задаст ряд вопросов о лицензии проекта, имени автора и т. д. После того как все будет сделано, просмотрите содержимое файла package.json. Вы можете отредактировать его вручную, но помните: файл должен содержать корректную разметку JSON.

Пустой проект успешно создан, теперь можно переходить к созданию модуля.

2.2.1. Создание модулей

Модуль может представлять собой как отдельный файл, так и каталог с одним или несколькими файлами (рис. 2.3). Если модуль оформлен в виде каталога, основному файлу в этом каталоге, который будет обрабатываться, обычно присваивается имя index.js (хотя и его можно переопределить: см. раздел 2.5).

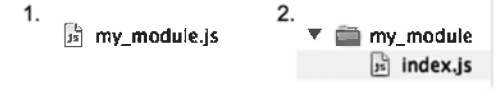


Рис. 2.3. Модули Node могут создаваться либо в виде файлов (пример 1), либо в виде каталогов (пример 2)

Чтобы создать типичный модуль, создайте файл, определяющий свойства объекта `exports` с любыми данными (строками, объектами, функциями и т. д.).

Чтобы продемонстрировать, как создаются базовые модули, мы добавим простейшую функциональность перерасчета валют в файл currency.js. Этот файл, представленный в следующем листинге, содержит две функции для пересчета канадских долларов в американские, и наоборот.

Листинг 2.1. Определение модуля Node (currency.js)

```
const canadianDollar = 0.91;

function roundTwo(amount) {
  return Math.round(amount * 100) / 100;
}
exports.canadianToUS = canadian => roundTwo(canadian * canadianDollar);
exports.USToCanadian = us => roundTwo(us / canadianDollar);
```

Функция canadianToUS определяется в exports, чтобы она могла использоваться в коде, требующем этот модуль.

Функция USToCanadian также определяется в exports.

Обратите внимание: задаются только два свойства объекта `exports`. Таким образом, для приложения, включающего модуль, будут доступны только две функции: `canadianToUS` и `USToCanadian`. Переменная `canadianDollar` закрыта от внешнего доступа: она влияет на логику работы `canadianToUS` и `USToCanadian`, но приложение не сможет обратиться к ней напрямую.

Чтобы использовать новый модуль в программе, воспользуйтесь функцией Node `require` и передайте путь к нужному модулю в аргументе. Node выполняет синхронный поиск модуля и загружает его содержимое. Сначала Node ищет файлы среди базовых модулей, затем в текущем каталоге, и наконец, в каталоге `node_modules`.

REQUIRE И СИНХРОННЫЙ ВВОД/ВЫВОД

Require — одна из немногих синхронных операций ввода/вывода в Node. Так как модули часто используются и обычно включаются в начале файла, синхронность require помогает сохранить чистоту, упорядоченность и удобочитаемость кода.

Старайтесь избегать использования require в частях вашего приложения, выполняющих интенсивный ввод/вывод. Любой синхронный вызов блокирует работу, не позволяя Node делать что-либо до завершения вызова. Например, если вы запустили HTTP-сервер, выполнение require для каждого входного запроса снизит быстродействие приложения. Как правило, именно по этой причине require и другие синхронные операции используются только при начальной загрузке приложения.

В файле test-currency.js (листинг 2.2) модуль currency.js включается вызовом `require`.

Листинг 2.2. Включение модуля (test-currency.js)

```
Использует функцию canadianToUS
модуля currency.

   const currency = require('./currency');   <- В пути означает, что модуль находится в одном
                                                каталоге со сценарием приложения.
   console.log('50 Canadian dollars equals this amount of US dollars:');
   console.log(currency.canadianToUS(50));
   console.log('30 US dollars equals this amount of Canadian dollars:');
   console.log(currency.USToCanadian(30));   <- Использует функцию USToCanadian
                                                модуля currency.
```

Включение модуля с путем, начинающимся с `./`, означает, что если вы создаете свой сценарий приложения с именем **test-currency.js** в каталоге **currency_app**, то файл модуля **currency.js** (рис. 2.4) также должен существовать в каталоге **currency_app**. При необходимости расширение **.js** подставляется по умолчанию, так что при желании его можно опустить. Если расширение **.js** не указано, Node также проверит файл **.json** с заданным именем. Файлы JSON загружаются как объекты JavaScript.

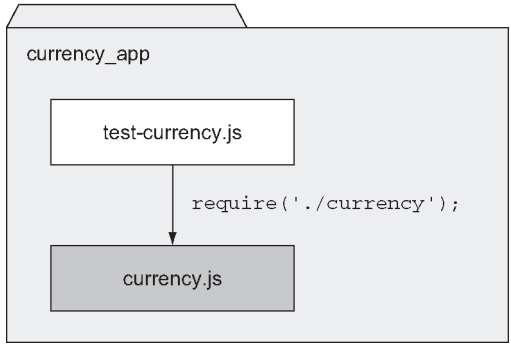


Рис. 2.4. Если аргумент require начинается с ./, Node проверяет тот каталог, в котором находится выполняемый файл

После того как Node найдет и обработает ваш модуль, функция `require` возвращает содержимое объекта `exports`, определенного в модуле. После этого вы сможете использовать две функции, возвращаемые модулем, для пересчета сумм в другую валюту.

Если вы захотите определить структуру модулей, разместите модули в подкаталогах. Если, например, вы хотите хранить модуль currency в папке **lib/**, приведите строку `require` к следующему виду:

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

Заполнение объекта `exports` модуля предоставляет простой механизм распределения повторно используемого кода по разным файлам.

2.3. Настройка создания модуля с использованием module.exports

Хотя заполнение объекта `exports` функциями и переменными хорошо подходит для большинства случаев, время от времени требуется создать модуль с отклонением от этой модели.

Например, модуль пересчета валют из предыдущего раздела можно переписать таким образом, чтобы он возвращал одну функцию-конструктор `Currency` вместо объекта, содержащего функции. Объектно-ориентированная реализация может выглядеть примерно так:

```
const Currency = require('./currency');
const canadianDollar = 0.91;
const currency = new Currency(canadianDollar);
console.log(currency.canadianToUS(50));
```

Возвращение функции из `require` (вместо объекта) сделает ваш код более элегантным — если это единственное, что требуется от модуля.

Казалось бы, чтобы создать модуль, возвращающий одну переменную или функцию, нужно присвоить `exports` то, что вы хотите вернуть. Однако такое решение не сработает, потому что Node не ожидает, что `exports` будет присваиваться любой другой объект, функция или переменная. Код модуля в следующем листинге пытается присвоить `exports` функцию (листинг 2.3).

Листинг 2.3. Модуль работать не будет

```
class Currency {
  constructor(canadianDollar) {
    this.canadianDollar = canadianDollar;
  }

  roundTwoDecimals(amount) {
    return Math.round(amount * 100) / 100;
  }

  canadianToUS(canadian) {
    return this.roundTwoDecimals(canadian * this.canadianDollar)
  }

  USToCanadian(us) {
    return this.roundTwoDecimals(us / this.canadianDollar);
  }
}
exports = Currency;     <-------  Ошибка; Node не позволяет переписывать exports.
```

Чтобы приведенный код модуля работал так, как ожидается, нужно заменить `exports` на `module.exports`. Механизм `module.exports` позволяет экспортировать одну переменную, функцию или объект. Если вы создаете модуль, который заполняет как `exports`, так и `module.exports`, то возвращается `module.exports`, а `exports` игнорируется.

ЧТО В ДЕЙСТВИТЕЛЬНОСТИ ЭКСПОРТИРУЕТСЯ

В конечном итоге в вашем приложении экспортируется module.exports. exports задается как глобальная ссылка на module.exports — изначально это пустой объект, к которому можно добавлять свойства. exports.myFunc — сокращенная запись для module.exports.myFunc.

В результате присваивание exports другой ссылки разрывает связь между module.exports и exports. Так как экспортируется module.exports, exports работает не так, как ожидается, — он уже не ссылается на module.exports. Чтобы сохранить эту связь, снова включите в module.exports ссылку на exports:

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

Использовав exports или module.exports в зависимости от ваших потребностей, вы сможете распределить функциональность по модулям и избежать проблем с постоянно растущими сценариями приложения.

2.4. Повторное использование модулей с папкой node_modules

Включение модулей с указанием их местонахождения в файловой системе относительно приложения полезно для организации кода, специфического для данного приложения. Иначе дело обстоит с кодом, предназначенным для использования в нескольких приложениях или распространения среди других разработчиков. Node имеет уникальный механизм повторного использования кода, который позволяет включать модули без точной информации об их местонахождении в файловой системе. Этот механизм основан на использовании каталогов node_modules.

В приведенном ранее примере включался модуль `./currency`. Если убрать `./` и включить просто `currency`, Node начинает искать этот модуль по схеме, представленной на рис. 2.5.

Переменная окружения `NODE_PATH` позволяет выбрать альтернативные каталоги для хранения модулей Node. Если переменная `NODE_PATH` используется, ей должен быть присвоен список каталогов, разделенных символом точки с запятой (;) в Windows или двоеточием (:) в других операционных системах.

2.5. Потенциальные проблемы

Хотя система модулей Node устроена достаточно прямолинейно, вы должны учитывать два обстоятельства.

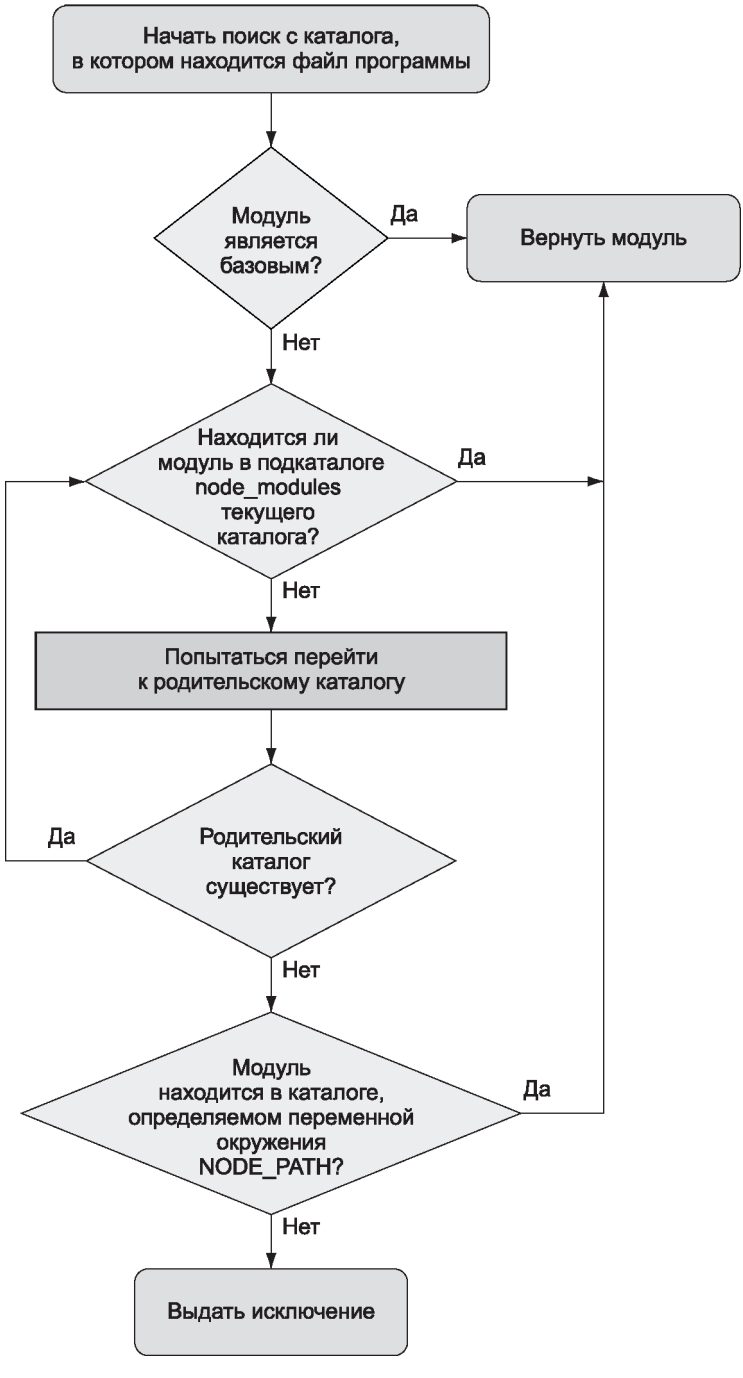


Рис. 2.5. Последовательность поиска модуля

Во-первых, если модуль представляет собой каталог, файл в каталоге модуля, который будет обработан, должен называться index.js, если только обратное не указано в файле package.json в каталоге модуля. Чтобы задать другой файл вместо index.js, файл package.json должен содержать данные JSON (JavaScript Object Notation), определяющие объект с ключом `main` и значением, которое определяет путь к основному файлу в каталоге модуля. Эти правила обобщены в блок-схеме на рис. 2.6.

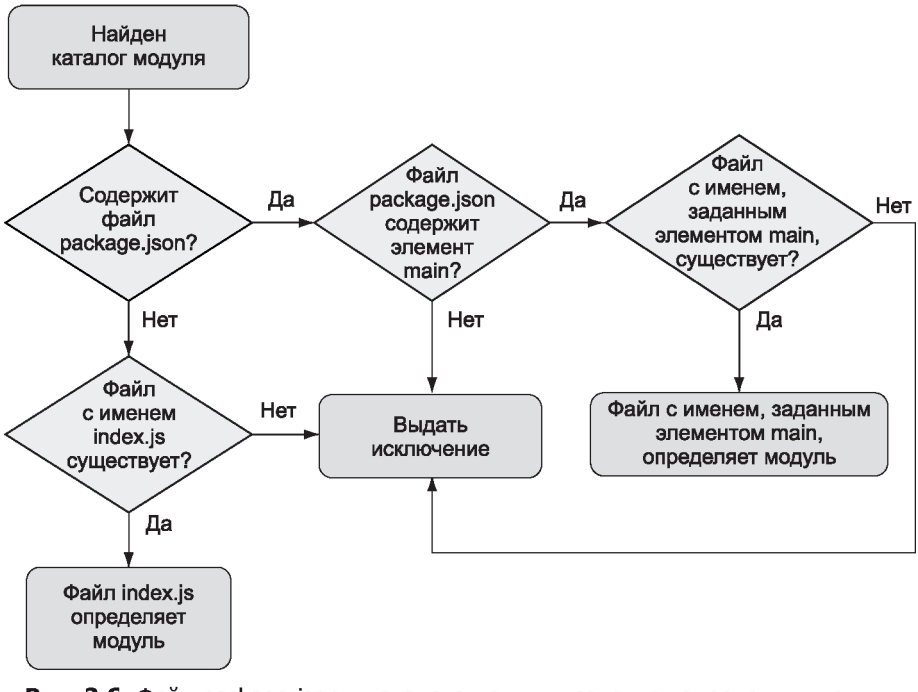


Рис. 2.6. Файл package.json в каталоге модуля позволяет определить модуль с использованием файла, отличного от index.js

Пример файла package.json, который назначает основным файлом currency.js:

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

Другое обстоятельство, о котором следует помнить, — это способность Node кэшировать модули как объекты. Если два файла в приложении включают один и тот же модуль, то данные, возвращенные для первого вызова `require`, будут сохранены в памяти, поэтому второму вызову `require` не нужно будет проверять исходные файлы модуля. А это означает, что загрузка модуля с вызовом `require` в том же процессе вернет тот же объект. Представьте, что вы создаете веб-приложение MVC, у которого имеется

основной объект приложения. Вы можете настроить этот объект, экспортировать его, а затем включить в любой точке проекта вызовом `require`. Если в объект приложения были добавлены полезные данные конфигурации, вы сможете обратиться к ним из других файлов — при условии, что проект имеет следующую структуру каталогов:

```
проект
   app.js
   models
      post.js
```

На рис. 2.7 показано, как работает эта схема.

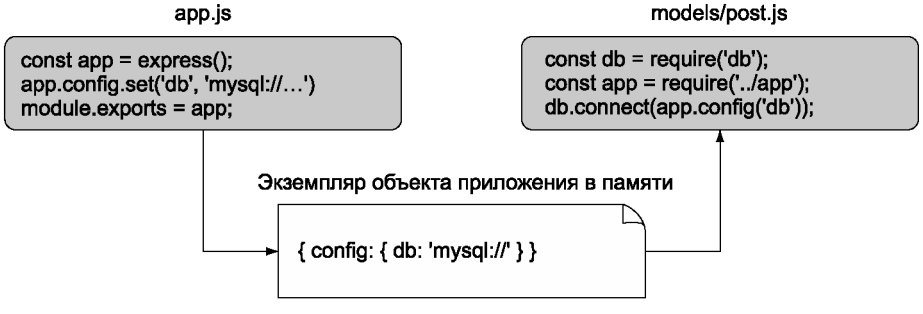


Рис. 2.7. Общий объект приложения в веб-приложении

Лучший способ освоиться с системой модулей Node — поэкспериментировать с ней и самостоятельно проверить поведение, описанное в этом разделе. Теперь, когда вы в общих чертах понимаете, как работают модули, можно переходить к средствам асинхронного программирования.

2.6. Средства асинхронного программирования

Если вы когда-либо занимались разработкой фронтэнда, в которой события (например, щелчки мышью) инициируют выполнение логики, — значит, вы уже занимались асинхронным программированием. Асинхронное программирование на стороне сервера работает по тем же принципам: происходят события, по которым срабатывает ответная логика. В мире Node для управления логикой ответов используются две популярные модели: обратные вызовы и слушатели событий.

Обратные вызовы обычно определяют логику для одноразовых ответов. Например, при выполнении запроса к базе данных можно назначить функцию обратного вызова, определяющую, что нужно сделать с результатами запроса. Функция обратного вызова может вывести результаты, выполнить с ними некие вычисления или выполнить другую функцию обратного вызова, использовав результаты запроса в качестве аргумента.

Слушатели событий представляют собой обратные вызовы, связанные с концептуальной сущностью (событием). Скажем, щелчок кнопкой мыши — это событие, которое обрабатывается в браузере. А код Node в сервере HTTP генерирует событие `request` при выдаче запроса HTTP. Вы можете прослушать событие `request` и добавить логику ответа. В следующем примере функция `handleRequest` будет вызываться каждый раз, когда генерируется событие `request`; для этого вызов метода `EventEmitter.prototype.on` связывает слушателя события с сервером:

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

Экземпляр сервера HTTP в Node является примером *генератора событий* — класса (`EventEmitter`), который может использоваться при наследовании и который добавляет возможность генерирования и обработки событий. Многие аспекты базовой функциональности Node наследуются от `EventEmitter`; вы также можете создать собственный генератор событий.

Итак, мы выяснили, что логика ответа обычно организуется в Node одним из двух способов. Теперь можно перейти непосредственно к асинхронному программированию; будут рассмотрены следующие темы:

- как обрабатывать одноразовые события с обратными вызовами;
- как реагировать на повторяющиеся события при помощи слушателей событий;
- как преодолеть некоторые трудности асинхронного программирования.

Начнем с одного из самых распространенных способов организации асинхронного кода: функций обратного вызова.

2.7. Обработка одноразовых событий в обратных вызовах

Функция обратного вызова (callback) — функция, которая передается в аргументе асинхронной функции и описывает, что нужно делать по завершении асинхронной операции. Функции обратного вызова часто применяются в разработке Node — чаще, чем генераторы событий, и с ними проще работать.

Чтобы продемонстрировать использование обратных вызовов в приложении, посмотрим, как создать простой сервер HTTP, который:

- асинхронно читает заголовки последних сообщений, хранящиеся в файле JSON;
- асинхронно читает базовый шаблон HTML;
- строит страницу HTML с заголовками;
- отправляет страницу HTML пользователю.

Примерный результат показан на рис. 2.8.

Файл JSON (titles.json), приведенный в листинге 2.4, отформатирован в виде массива с заголовками сообщений.

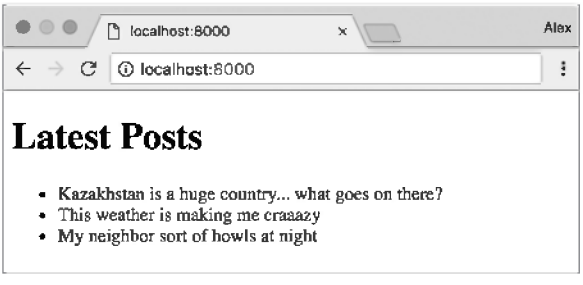


Рис. 2.8. HTML-ответ от веб-сервера, который читает заголовки из файла JSON и возвращает результаты в виде веб-страницы

Листинг 2.4. Список заголовков сообщений

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

Файл шаблона HTML (template.html) из листинга 2.5 включает только базовую структуру для вставки заголовков сообщений.

Листинг 2.5. Базовый шаблон HTML для вывода заголовков

```
<!doctype html>
<html>
  <head></head>
  <body>
    <h1>Latest Posts</h1>
    <ul><li>%</li></ul>
  </body>
</html>
```

% заменяется данными заголовка.

Код чтения данных из файла JSON и построения веб-страницы приведен в листинге 2.6 (файл blog_recent.js).

Листинг 2.6. Использование обратных вызовов в простом приложении

```
const http = require('http');
const fs = require('fs');
http.createServer((req, res) => {
  if (req.url == '/') {
    fs.readFile('./titles.json', (err, data) => {
```

Создает сервер HTTP и использует обратный вызов для определения логики ответа.

Читает файл JSON и использует обратный вызов для определения того, как поступить с его содержимым.

```
      if (err) {
        console.error(err);
        res.end('Server Error');
      } else {
        const titles = JSON.parse(data.toString());
        fs.readFile('./template.html', (err, data) => {
          if (err) {
            console.error(err);
            res.end('Server Error');
          } else {
          const tmpl = data.toString();
            const html = tmpl.replace('%', titles.join('</li><li>'));
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(html);
          }
        });
      }
    });
  }
}).listen(8000, '127.0.0.1');
```

Если возникает ошибка, зарегистрировать ее и вернуть клиенту сообщение «Server Error».

Разбирает данные из текста JSON.

Читает шаблон HTML и использует обратный вызов при завершении загрузки.

Собирает страницу HTML с заголовками сообщений.

Отправляет страницу HTML пользователю.

В этом примере используются три уровня вложенности обратных вызовов:

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

В трехуровневой структуре нет ничего плохого, но чем больше уровней вы используете, тем более громоздко выглядит ваш код и тем больше проблем возникает с его рефакторингом и тестированием, поэтому глубину вложения желательно ограничить. Создавая именованные функции для обработки отдельных уровней, вы можете выражать ту же логику способом, который требует больше строк кода — но этот код будет проще в сопровождении, тестировании и рефакторинге. Следующий листинг функционально эквивалентен листингу 2.6.

Листинг 2.7. Сокращение вложенности за счет определения промежуточных функций

```
const http = require('http');
const fs = require('fs');
http.createServer((req, res) => {
  getTitles(res);
}).listen(8000, '127.0.0.1');

function getTitles(res) {
  fs.readFile('./titles.json', (err, data) => {
    if (err) {
      hadError(err, res);
    } else {
      getTemplate(JSON.parse(data.toString()), res);
    }
```

Клиентский запрос поступает здесь.

Управление передается getTitles.

getTitles извлекает заголовки и передает управление getTemplate.

```
  });
}
function getTemplate(titles, res) {
  fs.readFile('./template.html', (err, data) => {
    if (err) {
      hadError(err, res);
    } else {
      formatHtml(titles, data.toString(), res);
    }
  });
}
function formatHtml(titles, tmpl, res) {
  const 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');
}
```

getTemplate читает файл шаблона и передает управление formatHtml.

formatHtml читает заголовки и шаблон, после чего строит ответ для клиента.

Если возникает ошибка, hadError выводит ее на консоль и возвращает клиенту сообщение «Server Error».

Уровень вложенности также можно сократить блоками `if/else` с другой идиомой в разработке Node — ранним возвратом из функции. Следующий листинг функционально эквивалентен предыдущему, но избегает дальнейшего вложения за счет раннего возврата управления. Кроме того, он явно указывает, что функция не должна продолжать выполнение.

Листинг 2.8. Сокращение вложения за счет раннего возврата

```
const http = require('http');
const fs = require('fs');
http.createServer((req, res) => {
  getTitles(res);
}).listen(8000, '127.0.0.1');
function getTitles(res) {
  fs.readFile('./titles.json', (err, data) => {
    if (err) return hadError(err, res);
    getTemplate(JSON.parse(data.toString()), res);
  });
}
function getTemplate(titles, res) {
  fs.readFile('./template.html', (err, data) => {
    if (err) return hadError(err, res);
    formatHtml(titles, data.toString(), res);
  });
}
function formatHtml(titles, tmpl, res) {
  const html = tmpl.replace('%', titles.join('</li><li>'));
  res.writeHead(200, { 'Content-Type': 'text/html'});
  res.end(html);
}
function hadError(err, res) {
```

Вместо того чтобы создавать ветвь else, вы возвращаете управление, потому что в случае возникновения ошибки продолжать выполнение этой функции не нужно.

```
  console.error(err);
  res.end('Server Error');
}
```

Итак, теперь вы знаете, как использовать обратные вызовы для обработки одноразовых событий для таких задач, как определение ответов при чтении файлов и запросов веб-серверов, и мы можем перейти к организации событий с использованием генераторов событий.

СОГЛАШЕНИЯ NODE ДЛЯ АСИНХРОННЫХ ОБРАТНЫХ ВЫЗОВОВ

Большинство встроенных модулей Node использует обратные вызовы с двумя аргументами: первый аргумент предназначен для ошибки (если она возникнет), а второй — для результатов. Аргумент ошибки error часто сокращается до err.

Типичный пример сигнатуры функции:

```
const fs = require('fs');
fs.readFile('./titles.json', (err, data) => {
  if (err) throw err;
  // Если ошибки не было, что-то сделать с данными
});
```

2.8. Обработка повторяющихся событий с генераторами событий

Генераторы событий инициируют события и включают возможность обработки этих событий при их инициировании. Некоторые важные компоненты Node API — серверы HTTP, серверы TCP и потоки — реализуются как генераторы событий. Разработчик также может создавать собственные генераторы.

Как упоминалось ранее, события обрабатываются при помощи слушателей. *Слушатель* (listener) представляет связь события с функцией обратного вызова, которая выполняется каждый раз при возникновении события. Например, у сокета TCP в Node имеется событие с именем `data`, которое инициируется каждый раз при появлении новых данных в сокете:

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

Посмотрим, как использовать события `data` для создания эхо-сервера.

2.8.1. Пример генератора событий

Простой пример повторяющихся событий встречается в эхо-сервере. Когда вы отправляете данные эхо-серверу, он просто выводит полученные данные (рис. 2.9).

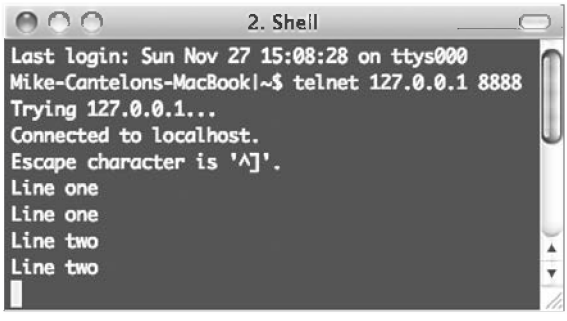


Рис. 2.9. Эхо-сервер повторяет отправленные ему данные

В листинге 2.9 приведен код, необходимый для реализации эхо-сервера. Каждый раз при подключении клиента создается сокет. Сокет представляет собой генератор событий, к которому можно добавить слушателя методом `on` для реакции на события `data`. Эти события данных генерируются каждый раз при появлении новых данных в сокете.

Листинг 2.9. Использование метода on для реакции на события

```
const net = require('net');
const server = net.createServer(socket => {
  socket.on('data', data => {
    socket.write(data);
  });
});
server.listen(8888);
```

События data обрабатываются каждый раз при чтении новых данных.

Данные записываются на сторону клиента.

Эхо-сервер запускается следующей командой:

```
node echo_server.js
```

После того как эхо-сервер будет запущен, вы можете подключиться к нему следующей командой:

```
telnet 127.0.0.1 8888
```

Каждый раз, когда подключенный сеанс telnet отправляет данные серверу, эти данные будут продублированы в сеансе telnet.

TELNET В СИСТЕМЕ WINDOWS

Если вы используете операционную систему Microsoft Windows, поддержка telnet может не устанавливаться по умолчанию, и вам придется устанавливать ее самостоятельно. Инструкции для различных версий Windows доступны на Technet: http://mng.bz/egzr.

2.8.2. Реакция на событие, которое должно происходить только один раз

Слушатели могут определяться для многократной реакции на события, как в предыдущем примере, или же для однократной реакции. В следующем листинге, использующем метод `once`, приведенный ранее пример с эхо-сервером изменяется так, чтобы он воспроизводил только первый фрагмент отправленных ему данных.

Листинг 2.10. Использование метода once для реакции на одно событие

```
const net = require('net');
const server = net.createServer(socket => {
  socket.once('data', data => {
    socket.write(data);
  });
});
server.listen(8888);
```

Событие data будет обработано всего один раз.

2.8.3. Создание генераторов событий: публикация/подписка

В предыдущем примере использовался встроенный Node API, использующий генераторы событий. Однако встроенный модуль событий Node позволяет создавать собственные генераторы событий.

Следующий код определяет генератор событий `channel` с одним слушателем, который реагирует на присоединение пользователя к каналу. Обратите внимание на использование `on` (или в более длинной альтернативной форме `addListener`) для добавления слушателя к генератору событий:

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

Однако функция обратного вызова `join` никогда не будет вызвана, потому что никакие события еще не генерируются. Добавьте в листинг строку, которая инициирует событие функцией `emit`:

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

ИМЕНА СОБЫТИЙ

События — ключи, с которыми может быть связано любое строковое значение: data, join и вообще произвольное длинное имя. Только одно событие с именем error играет особую роль; вскоре мы рассмотрим его.

Давайте посмотрим, как реализовать собственную логику публикации/подписки с использованием `EventEmitter` для создания канала передачи информации. Запустив сценарий из листинга 2.11, вы получите простой чат-сервер. Канал чат-сервера реализуется как генератор событий, который реагирует на события `join`, генерируемые клиентами. Когда клиент присоединяется к каналу, логика слушателя `join` в свою очередь добавляет дополнительного слушателя для данного клиента к каналу широковещательного события, который будет записывать все рассылаемые сообщения в клиентский сокет. Имена типов событий (такие, как `join` и `broadcast`) выбираются абсолютно произвольно. При желании вы можете использовать другие имена для этих типов событий.

Листинг 2.11. Простая система «публикация/подписка» с использованием генератора событий

```
const events = require('events');
const net = require('net');
const channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};
channel.on('join', function(id, client) {
  this.clients[id] = client;
  this.subscriptions[id] = (senderId, message) => {
    if (id != senderId) {
      this.clients[id].write(message);
    }
  };
  this.on('broadcast', this.subscriptions[id]);
});
const server = net.createServer(client => {
  const id = `${client.remoteAddress}:${client.remotePort}`;
  channel.emit('join', id, client);
  client.on('data', data => {
    data = data.toString();
    channel.emit('broadcast', id, data);
  });
});
server.listen(8888);
```

Добавляет для события join слушателя, который сохраняет объект client, чтобы приложение могло отправить данные обратно пользователю.

Игнорирует данные, если они передаются напрямую в широковещательной рассылке.

Добавляет слушателя, относящегося к текущему пользователю, для события broadcast.

Генерирует событие join при подключении пользователя к серверу, с указанием идентификатора пользователя и объекта client.

Генерирует событие широковещательной рассылки в канале с указанием идентификатора пользователя и сообщения при отправке данных любым пользователем.

После того как чат-сервер заработает, откройте новое окно командной строки и введите следующий код для входа в чат:

```
telnet 127.0.0.1 8888
```

Если вы открыли несколько окон командной строки, вы увидите, что все данные, вводимые в одном окне, воспроизводятся в других.

У этого чат-сервера есть серьезная проблема: при закрытии подключений и выходе пользователя из чат-комнаты остается слушатель, который будет пытаться

записывать данные клиенту, который уже не подключен. Конечно, произойдет ошибка. Чтобы решить ее, необходимо добавить слушателя в генератор событий `channel` (листинг 2.12) и добавить логику в слушателя события `close` для генерирования события `leave` канала. Событие `leave` удаляет слушателя `broadcast`, который был изначально добавлен для клиента.

Листинг 2.12. Создание слушателя для завершения при отключении клиента

```
...
channel.on('leave', function(id) {
  channel.removeListener(
    'broadcast', this.subscriptions[id]
  );
  channel.emit('broadcast', id, `${id} has left the chatroom.\n`);
});
const server = net.createServer(client => {
  ...
  client.on('close', () => {
    channel.emit('leave', id);
  });
});
server.listen(8888);
```

Создает слушателя для события leave.

Удаляет широковещательного слушателя для конкретного клиента.

Генерирует события leave для отключения клиента.

Если по какой-то причине вы захотите отключить чат без завершения сервера, вы также можете использовать метод генератора события `removeAllListeners` для удаления всех слушателей заданного типа. Следующий код показывает, как эта возможность реализуется для нашего чат-сервера:

```
channel.on('shutdown', () => {
  channel.emit('broadcast', '', 'The server has shut down.\n');
  channel.removeAllListeners('broadcast');
});
```

Затем можно добавить поддержку команды чата, завершающей общение. Для этого измените слушателя для события `data` в следующем коде:

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

Теперь при вводе любым участником в чате команды `shutdown` все участники будут отключены.

ОБРАБОТКА ОШИБОК

Стандартный прием, часто применяемый при создании генераторов событий, — генерирование события типа error вместо прямого инициирования ошибки. Это позволяет вам определять специальную логику реакции на события посредством назначения одного или нескольких слушателей для данного типа события.

Следующий код показывает, как слушатель событий обрабатывает сгенерированную ошибку, выводя информацию на консоль:

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

Если слушатель для этого типа события не определен при генерировании типа события error, генератор события выводит трассировку стека (список команд программы, выполнявшихся до точки возникновения ошибки) и прерывает выполнение. В трассировке стека обозначен тип ошибки, заданный вторым аргументом вызова emit. Это поведение присуще исключительно событиям типа error; при генерировании других типов событий, не имеющих слушателей, ничего не происходит.

Если событие типа error генерируется без передачи объекта error во втором аргументе, в трассировке стека будет указано «неперехваченное неопределенное событие ошибки 'error'», и приложение аварийно завершается. Существует устаревший механизм, который может применяться для обработки таких ошибок, — вы можете определить собственную реакцию, определив глобальный обработчик в следующем коде:

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

Были разработаны некоторые альтернативы для такого решения — например, домены (http://nodejs.org/api/domain.html), но они не считаются готовыми для применения в реально эксплуатируемом коде.

Если вы захотите предоставить пользователям, подключающимся к чату, счетчик пользователей, активных в настоящий момент, можно использовать следующий метод `listeners`, который возвращает массив слушателей для заданного типа события:

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

Чтобы увеличить количество слушателей у генератора событий и чтобы избежать предупреждений, которые Node выдает при более 10 слушателях, также можно воспользоваться методом `setMaxListeners`. Если взять наш генератор событий в качестве примера, для увеличения количества разрешенных слушателей можно использовать следующий код:

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

2.8.4. Доработка генератора событий: отслеживание содержимого файлов

Чтобы расширить поведение генератора событий, вы можете создать новый класс JavaScript, наследующий от генератора событий. Например, вы можете создать класс `Watcher` для обработки файлов, находящихся в заданном каталоге файловой системы. После этого класс может использоваться для создания программ, отслеживающих содержимое каталога (все файлы, помещенные в каталог, переименовываются с приведением символов к нижнему регистру, после чего копируются в отдельный каталог).

После создания объекта `Watcher` необходимо дополнить методы, унаследованные от `EventEmitter`, двумя новыми методами из листинга 2.13.

Листинг 2.13. Расширение функциональности генератора событий

```
const fs = require('fs');
const events = require('events');

class Watcher extends events.EventEmitter {
  constructor(watchDir, processedDir) {
    super();
    this.watchDir = watchDir;
    this.processedDir = processedDir;
  }

  watch() {
    fs.readdir(this.watchDir, (err, files) => {
      if (err) throw err;
      for (var index in files) {
        this.emit('process', files[index]);
      }
    });
```

Расширяет объект EventEmitter методом, обрабатывающим файлы.

Обрабатывает каждый файл в отслеживаемом каталоге.

```
  }

  start() {                                    Добавляет метод для
    fs.watchFile(this.watchDir, () => {        начала отслеживания.
      this.watch();
    });
  }
}

module.exports = Watcher;
```

Метод `watch` перебирает содержимое каталога и обрабатывает все найденные файлы. Метод `start` запускает отслеживание каталога. Для отслеживания используется функция Node `fs.watchFile`; когда в каталоге что-то происходит, срабатывает метод `watch`, который перебирает содержимое каталога и выдает событие `process` для каждого обнаруженного файла.

После того как класс `Watcher` будет определен, его можно использовать для создания объекта `Watcher` следующей командой:

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

Для созданного объекта `Watcher` вызывается метод `on`, унаследованный от класса генератора события; он задает логику обработки каждого файла:

```
watcher.on('process', (file) => {
  const watchFile = `${watchDir}/${file}`;
  const processedFile = `${processedDir}/${file.toLowerCase()}`;
  fs.rename(watchFile, processedFile, err => {
    if (err) throw err;
  });
});
```

Вся необходимая логика готова. Запустите отслеживание каталога следующим кодом:

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

После размещения кода `Watcher` в сценарии и создания каталогов watch и done запустите сценарий с использованием Node и скопируйте файлы в каталог отслеживания — вы увидите, что файлы появляются (с переименованием в нижний регистр) в итоговом каталоге. Этот пример показывает, как использовать генератор событий для создания новых классов.

Научившись использовать обратные вызовы для определения асинхронной логики и использовать генераторы событий для многократного срабатывания асинхронной логики, вы приблизитесь на шаг к умению управлять приложениями Node. Однако отдельный обратный вызов или слушатель генератора событий может включать логику выполнения дополнительных асинхронных задач. Если порядок выполнения

этих задач важен, вы можете столкнуться с новой проблемой: как управлять тем, когда именно будет выполняться каждая задача в серии асинхронных задач?

Прежде чем заняться управлением потоком выполнения (эта тема будет рассматриваться в разделе 2.10), стоит рассмотреть некоторые проблемы, с которыми вы с большой вероятностью столкнетесь при написании асинхронного кода.

2.9. Проблемы с асинхронной разработкой

При создании асинхронных приложений необходимо внимательно следить за потоком выполнения приложения и состоянием приложения: условиями цикла событий, переменными приложения и любыми другими ресурсами, изменяющимися в ходе выполнения логики программы.

Цикл событий Node, например, отслеживает асинхронную логику, выполнение которой еще не завершилось. Пока остается незавершенная асинхронная логика, выход из процесса Node не происходит. Поведение постоянно выполняемого процесса Node хорошо подходит для такого приложения, как веб-сервер, но нежелательно для процессов, которые должны завершиться по истечении некоторого периода времени (например, программ командной строки). Цикл событий отслеживает все подключения к базе данных, пока они не будут закрыты, не позволяя Node завершить работу приложения.

Переменные приложения также могут неожиданно изменяться при недостаточном соблюдении осторожности. В листинге 2.14 приведен пример того, как порядок выполнения асинхронного кода может вызвать путаницу. Если бы пример кода выполнялся синхронно, можно предположить, что было бы выведено сообщение «The color is blue». Но поскольку пример выполняется асинхронно, значение переменной `color` изменяется до выполнения `console.log`, и выводится сообщение «The color is green».

Листинг 2.14. Ошибки, вызванные изменением области видимости

```
function asyncFunction(callback) {
  setTimeout(callback, 200);
}
let color = 'blue';
asyncFunction(() => {
  console.log(`The color is ${color}`); ◄—— Выполняется в последнюю очередь (на 200 мс позже).
});
color = 'green';
```

Чтобы «заморозить» содержимое переменной `color`, можно изменить логику и использовать *замыкание* (closure) JavaScript. В листинге 2.15 вызов `asyncFunction` заключен в анонимную функцию, которая получает аргумент `color`. Затем анонимная функция, которой отправляется текущее содержимое `color`, немедленно выполняется. Так как

`color` становится аргументом анонимной функции, значение становится локальным для области видимости этой функции, и при изменении значения `color` за пределами анонимной функции локальная копия остается без изменений.

Листинг 2.15. Использование анонимной функции для сохранения значения глобальной переменной

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

let color = 'blue';

(color => {
  asyncFunction(() => {
    console.log('The color is', color);
  });
})(color);

color = 'green';
```

Это всего лишь один из многочисленных приемов из области программирования на JavaScript, которые встретятся вам в ходе разработки Node.

ЗАМЫКАНИЯ

За дополнительной информацией о замыканиях обращайтесь к документации JavaScript от Mozilla: https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Closures.

Теперь вы знаете, как использовать замыкания для управления состоянием приложения, и мы можем перейти к теме упорядочения асинхронной логики для управления потоком выполнения приложения.

2.10. Упорядочение асинхронной логики

В процессе выполнения асинхронной программы некоторые задачи могут выполняться в любой момент независимо от того, что происходит в остальном коде программы, без каких-либо проблем. Однако некоторые задачи должны выполняться только до или после других задач.

В сообществе Node концепцию упорядочения групп асинхронных задач принято называть «потоком выполнения». Существует две разновидности потока выполнения: последовательный и параллельный (рис. 2.10).

Задачи, которые должны выполняться одна после другой, называются *последовательными*. Простой пример — задачи создания каталога и сохранения файла в каталоге. Файл невозможно сохранить до того, как каталог будет создан.

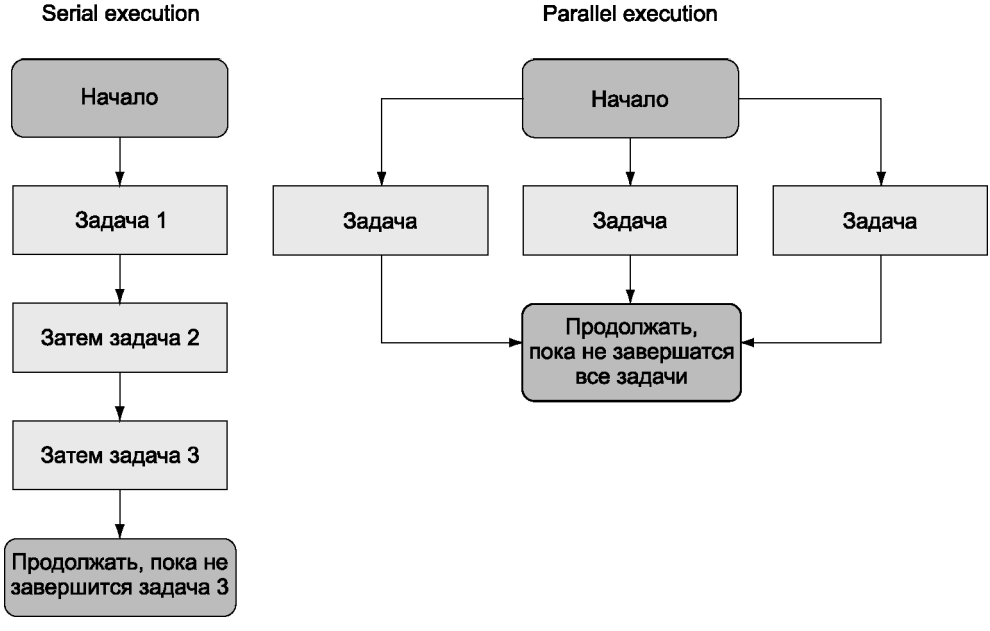


Рис. 2.10. Последовательное выполнение асинхронных задач концептуально сходно с синхронной логикой: задачи выполняются одна за другой. С другой стороны, параллельные задачи могут выполняться одновременно друг с другом

Задачи, которые не обязаны выполняться одна за другой, называются *параллельными*. Относительный порядок запуска и остановки этих задач может быть неважен, но все эти задачи должны быть завершены до выполнения дальнейшей логики. Пример — загрузка нескольких файлов, которые позднее будут сжаты в zip-архив. Файлы могут загружаться одновременно, но все загрузки должны быть завершены до создания архива.

Управление последовательным и параллельным потоком выполнения требует некоторых служебных операций на программном уровне. При реализации последовательного потока выполнения приходится отслеживать задачу, выполняемую в настоящий момент, или же поддерживать очередь невыполненных задач. При реализации параллельного выполнения необходимо хранить информацию о том, сколько задач отработало до завершения.

Средства управления потоком выполнения берут на себя рутинные операции, что упрощает группировку асинхронных последовательных или параллельных задач. В сообществе было создано достаточно много дополнений, предназначенных для упорядочения асинхронной логики, самостоятельная реализация этих средств снимает с них покров тайны и помогает вам глубже понять, как решаются проблемы асинхронного программирования.

В следующих разделах мы покажем:

- когда использовать последовательный поток выполнения;
- как реализовать последовательный поток выполнения;
- как реализовать параллельный поток выполнения;
- как использовать сторонние модули для управления потоком выполнения.

Для начала стоит разобраться, когда и как в асинхронном мире применяется последовательный поток выполнения.

2.11. Когда применяется последовательный поток выполнения

Для последовательного выполнения нескольких асинхронных задач можно воспользоваться обратными вызовами. Но если таких задач достаточно много, их придется как-то организовать. Если этого не сделать, код получится слишком громоздким из-за чрезмерного вложения обратных вызовов.

Ниже приведен пример последовательного выполнения задач с применением обратных вызовов. В этом примере функция `setTimeout` используется для моделирования задач, выполнение которых требует времени: первая задача выполняется за одну секунду, вторая — за половину секунды, и последняя — за десятую долю секунды. Вызов `setTimeout` — искусственная модель; реальный код может читать файлы, выдавать запросы HTTP и т. д. И хотя код этого примера достаточно короткий, он не отличается элегантностью и не позволяет легко добавить еще одну задачу на программном уровне.

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

Также для выполнения этих задач можно воспользоваться средствами управления потоком выполнения — например, Async (*http://caolan.github.io/async/*). Модуль Async прост в использовании, а среди его достоинств можно выделить малую кодовую базу (всего 837 байт после минификации и сжатия). Установка Async устанавливается следующей командой:

```
npm install async
```

Код в следующем листинге заново реализует приведенный выше фрагмент с последовательным потоком выполнения.

Листинг 2.16. Организация последовательного выполнения с использованием дополнений, разработанных в сообществе

```
const async = require('async');
async.series([   <-------  Массив функций, которые должны быть выполнены Async – одна за другой.
  callback => {
    setTimeout(() => {
      console.log('I execute first.');
      callback();
    }, 1000);
  },
  callback => {
    setTimeout(() => {
      console.log('I execute next.');
      callback();
    }, 500);
  },
  callback => {
    setTimeout(() => {
      console.log('I execute last.');
      callback();
    }, 100);
  }
]);
```

Хотя реализация, использующая поток выполнения, повышает количество строк кода, обычно она создает меньше проблем с чтением и сопровождением. Скорее всего, вы не будете использовать это решение постоянно, но, когда вы столкнетесь с ситуацией, в которой потребуется избежать вложения обратных вызовов, этот прием поможет вам получить более удобочитаемый код.

Итак, мы рассмотрели пример организации последовательного управления потоком с использованием специализированных средств. Теперь посмотрим, как реализовать его с нуля.

2.12. Реализация последовательного потока выполнения

Чтобы организовать последовательное выполнение нескольких асинхронных задач с использованием явного управления потоком, сначала нужно поместить задачи в массив в желательном порядке их выполнения. Как видно из рис. 2.11, этот массив определяет очередь: при завершении одной задачи из массива извлекается следующая.

Каждая задача представлена в массиве функцией. При завершении задача должна вызвать функцию-обработчик для указания статуса ошибки и результатов. Функ-

Рис. 2.11. Управление последовательным потоком выполнения

ция-обработчик в этой реализации прерывает выполнение в случае ошибки. Если ошибки не было, обработчик извлекает из очереди следующую задачу и выполняет ее.

Чтобы продемонстрировать реализацию управления последовательным потоком выполнения, мы создадим простое приложение, которое выводит заголовок одной статьи и URL-адрес случайно выбранного канала RSS. Список возможных каналов RSS хранится в текстовом файле. Вывод приложения выглядит примерно так:

```
Of Course ML Has Monads!
http://lambda-the-ultimate.org/node/4306
```

В нашем примере используются два вспомогательных модуля из реестра npm. Откройте окно командной строки и введите следующие команды, чтобы создать каталог для примера и установить вспомогательные модули:

```
mkdir listing_217
cd listing_217
npm init -y
npm install --save request@2.60.0
npm install --save htmlparser@1.7.7
```

Модуль `request` реализует упрощенного клиента HTTP, который может использоваться для выборки данных RSS. Модуль `htmlparser` обладает функциональностью, которая позволяет преобразовать низкоуровневые данные RSS в структуры данных JavaScript.

Затем создайте в новом каталоге файл **index.js** с кодом из листинга 2.17.

Листинг 2.17. Реализация управления последовательным потоком выполнения в простом приложении

```
const fs = require('fs');
const request = require('request');
const htmlparser = require('htmlparser');
const configFilename = './rss_feeds.txt';
function checkForRSSFile() {
```

Задача 1: Убедиться в том, что файл со списком URL-адресов канала RSS существует.

```
  fs.exists(configFilename, (exists) => {
    if (!exists)
      return next(new Error(`Missing RSS file: ${configFilename}`));
    next(null, configFilename);
  });
}
function readRSSFile(configFilename) {
  fs.readFile(configFilename, (err, feedList) => {
    if (err) return next(err);
    feedList = feedList
      .toString()
      .replace(/^\s+|\s+$/g, '')
      .split('\n');
    const random = Math.floor(Math.random() * feedList.length);
    next(null, feedList[random]);
  });
}
function downloadRSSFeed(feedUrl) {
  request({ uri: feedUrl }, (err, res, body) => {
    if (err) return next(err);
    if (res.statusCode !== 200)
      return next(new Error('Abnormal response status code'));
    next(null, body);
  });
}
function parseRSSFeed(rss) {
  const handler = new htmlparser.RssHandler();
  const parser = new htmlparser.Parser(handler);
  parser.parseComplete(rss);
  if (!handler.dom.items.length)
    return next(new Error('No RSS items found'));
  const item = handler.dom.items.shift();
  console.log(item.title);
  console.log(item.link);
}
const tasks = [
  checkForRSSFile,
  readRSSFile,
  downloadRSSFeed,
  parseRSSFeed
];
function next(err, result) {
  if (err) throw err;
  const currentTask = tasks.shift();
  if (currentTask) {
    currentTask(result);
  }
}
next();
```

При возникновении ошибки вернуть управление.

Задача 2: Прочитать и разобрать файл с URL-адресами канала.

Преобразует список URL-адресов поставки канала в строку, а затем в массив.

Выбирает случайный URL-адрес из массива.

Задача 3: Выполнить запрос HTTP и получить данные выбранного канала.

Задача 4: Разобрать данные RSS в массив.

Выводит заголовок и URL-адрес первого элемента, если он существует.

Добавляет каждую выполняемую задачу в массив в порядке выполнения.

Функция с именем next выполняет каждую задачу.

Выдает исключение, если в ходе выполнения задачи происходит ошибка.

Следующая задача берется из массива задач.

Выполняет текущую задачу.

Запускает последовательное выполнение задач.

Прежде чем тестировать приложение, создайте файл **rss_feeds.txt** в одном каталоге со сценарием приложения. Если у вас нет под рукой ни одного канала RSS, опробуйте канал блога Node по адресу *http://blog.nodejs.org/feed/*. Поместите URL-адреса

канала RSS в текстовый файл, по одному в каждой строке файла. После того как файл будет создан, откройте окно командной строки и введите следующие команды, чтобы перейти в каталог приложения и выполнить сценарий:

```
cd listing_217
node index.js
```

Последовательный поток выполнения, как показывает реализация примера, позволяет вводить обратные вызовы в действие тогда, когда они необходимы — вместо простого вложения.

Разобравшись с реализацией последовательного потока выполнения, можно переходить к параллельному выполнению асинхронных задач.

2.13. Реализация параллельного потока выполнения

Чтобы несколько асинхронных задач выполнялись параллельно, задачи также необходимо поместить в массив, но на этот раз порядок их следования неважен. Каждая задача должна вызывать функцию-обработчик, которая увеличивает количество завершенных задач. После того как все задачи будут завершены, функция-обработчик должна выполнить некоторую завершающую логику.

В качестве примера параллельного потока выполнения мы создадим простое приложение, которое читает содержимое текстовых файлов и выводит частоты использования слов в файлах.

Чтение содержимого текстовых файлов будет производиться при помощи асинхронной функции `readFile`, чтобы операции чтения из файлов могли осуществляться параллельно. На рис. 2.12 показано, как работает это приложение.

Результат выглядит примерно так (хотя, скорее всего, он будет намного длиннее):

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

Откройте окно командной строки и введите следующие команды, чтобы создать два каталога: один для примера, а другой, внутри него, — для текстовых файлов, которые нужно проанализировать:

```
mkdir listing_218
cd listing_218
mkdir text
```

Затем создайте в каталоге listing_218 файл word_count.js, содержащий код из листинга 2.18.

Листинг 2.18. Реализация управления параллельным потоком выполнения в простом приложении

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

function checkIfComplete() {
  completedTasks++;
  if (completedTasks === tasks.length) {
    for (let index in wordCounts) {
      console.log(`${index}: ${wordCounts[index]}`);
    }
  }
}

function addWordCount(word) {
  wordCounts[word] = (wordCounts[word]) ? wordCounts[word] + 1 : 1;
}

function countWordsInText(text) {
  const words = text
    .toString()
    .toLowerCase()
    .split(/\W+/)
    .sort();
  words
    .filter(word => word)
    .forEach(word => addWordCount(word));
}

fs.readdir(filesDir, (err, files) => {
  if (err) throw err;
  files.forEach(file => {
    const task = (file => {
      return () => {
        fs.readFile(file, (err, text) => {
          if (err) throw err;
          countWordsInText(text);
          checkIfComplete();
        });
      };
    })(`${filesDir}/${file}`);
    tasks.push(task);
  }
  tasks.forEach(task => task());
});
```

Когда все задачи будут завершены, вывести список всех слов, использованных в файлах, и количество вхождений каждого слова.

Подсчитывает вхождения слов в тексте.

Получает список файлов в каталоге.

Определяет задачу для обработки каждого файла. Каждая задача включает вызов функции, которая асинхронно читает содержимое файла, после чего подсчитывает в нем вхождения слов.

Добавляет каждую задачу в массив функций, вызываемых для параллельного выполнения.

Запускает параллельное выполнение задач.

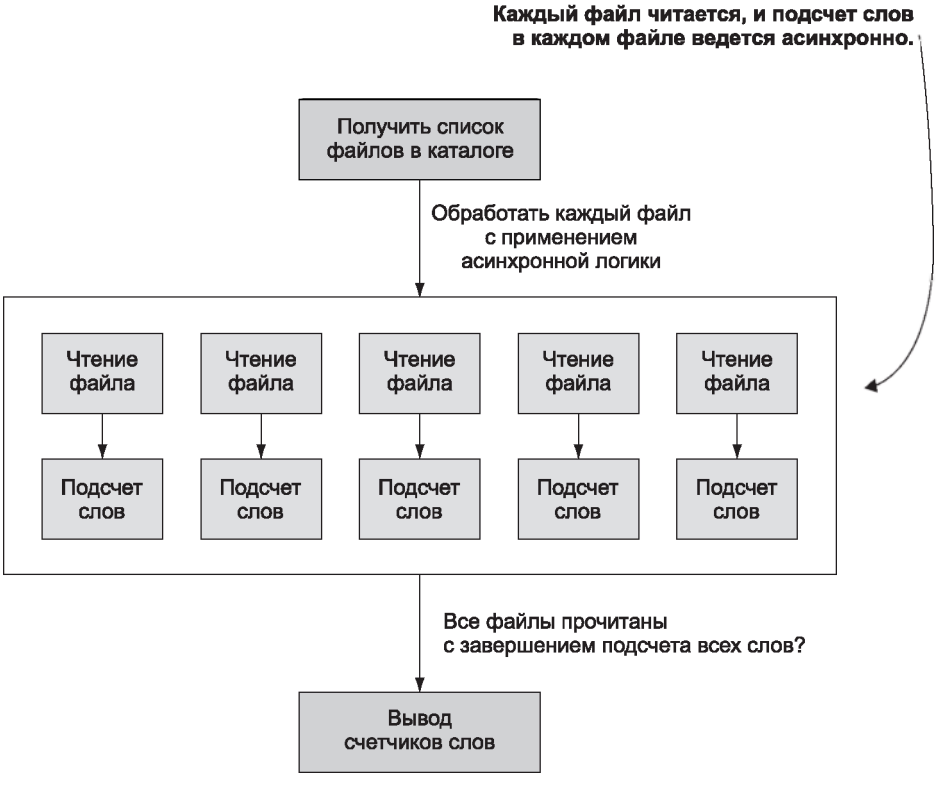


Рис. 2.12. Управление параллельным потоком выполнения

Прежде чем тестировать приложение, создайте несколько текстовых файлов в каталоге, созданном ранее. Откройте окно командной строки и введите следующие команды, чтобы перейти в каталог приложения и выполнить сценарий:

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

Теперь, когда вы знаете, как работает последовательное и параллельное выполнение задач, рассмотрим инструментарий, разработанный в сообществе. Этот инструментарий позволяет легко управлять потоком выполнения, и вам не придется реализовывать его самостоятельно.

2.14. Средства, разработанные в сообществе

Многие дополнения, разработанные в сообществе, предоставляют удобные средства управления потоком выполнения. Наибольшей популярностью пользуются такие

дополнения, как Async, Step и Seq. И хотя каждое из них заслуживает внимания, мы снова используем Async в другом примере.

ДОПОЛНЕНИЯ ДЛЯ УПРАВЛЕНИЯ ПОТОКОМ ВЫПОЛНЕНИЯ

За дополнительной информацией о дополнениях для управления потоком выполнения, разработанных в сообществе, обращайтесь к статье «Virtual Panel: How to Survive Asynchronous Programming in JavaScript» Вернера Шустера (Werner Schuster) и Дио Синодиноса (Dio Synodinos) на InfoQ: http://mng.bz/wKnV

В листинге 2.19 приведен пример использования Async для упорядочения выполнения задач в сценарии, использующем параллельный поток выполнения для одновременной загрузки двух файлов с их последующей архивацией.

СЛЕДУЮЩИЙ ПРИМЕР НЕ БУДЕТ РАБОТАТЬ В MICROSOFT WINDOWS

Так как команды tar и curl не входят в комплект поставки операционной системы Windows, следующий пример не будет работать в этой операционной системе.

Листинг 2.19. Использование средств управления потоком, разработанных в сообществе, в простом приложении

```
const async = require('async');
const exec = require('child_process').exec;
function downloadNodeVersion(version, destination, callback) {
  const url = `http://nodejs.org/dist/v${version}/node-v${version}.tar.gz`;
  const filepath = `${destination}/${version}.tgz`;
  exec(`curl ${url} > ${filepath}`, callback);
}
async.series([
  callback => {
    async.parallel([
      callback => {
        console.log('Downloading Node v4.4.7...');
        downloadNodeVersion('4.4.7', '/tmp', callback);
      },
      callback => {
        console.log('Downloading Node v6.3.0...');
        downloadNodeVersion('6.3.0', '/tmp', callback);
      }
    ], callback);
  },
  callback => {
    console.log('Creating archive of downloaded files...');
    exec(
      'tar cvf node_distros.tar /tmp/4.4.7.tgz /tmp/6.3.0.tgz',
err => {
  if (err) throw err;
```

Загружает исходный код Node для заданной версии.

Последовательно выполняет серию задач.

Выполняет загрузки параллельно.

Создает файл архива.

```
        console.log('All done!');
        callback();
      }
    );
  }
]);
```

В этом примере последовательный поток выполнения используется для того, чтобы загрузка гарантированно была завершена до начала архивации.

Сценарий определяет вспомогательную функцию, которая загружает заданную поставляемую версию исходного кода Node. Затем последовательно выполняются две задачи: параллельная загрузка двух версий Node и упаковка загруженных версий в новый архивный файл.

2.15. Заключение

Код Node может оформляться в модули, предназначенные для повторного использования.

- Функция `require` используется для загрузки модулей.
- Объекты `module.exports` и `exports` предназначены для предоставления внешнего доступа к функциям и переменным из модуля.
- Файл package.json определяет зависимости и файл, экспортируемый как основной.
- Для управления асинхронной логикой можно использовать вложенные обратные вызовы, генераторы событий и средства управления потоком выполнения.

3 Что представляет собой веб-приложение Node?

Эта глава полностью посвящена веб-приложениям Node. Прочитав ее, вы поймете, как выглядят веб-приложения Node и как приступить к их построению. Вы будете делать все, что делает современный веб-разработчик при создании приложений.

Мы создадим веб-приложение по образцу таких популярных сайтов отложенного чтения, как Instapaper (*www.instapaper.com*) и Pocket (*getpocket.com*). В процессе работы нам предстоит создать новый проект Node, управлять зависимостями, создать REST-совместимый API, сохранить информацию в базе данных, а также создать интерфейс на основе шаблонов. На первый взгляд довольно много, но все концепции этой главы будут более подробно исследованы в последующих главах.

На рис. 3.1 показано, как выглядит результат.

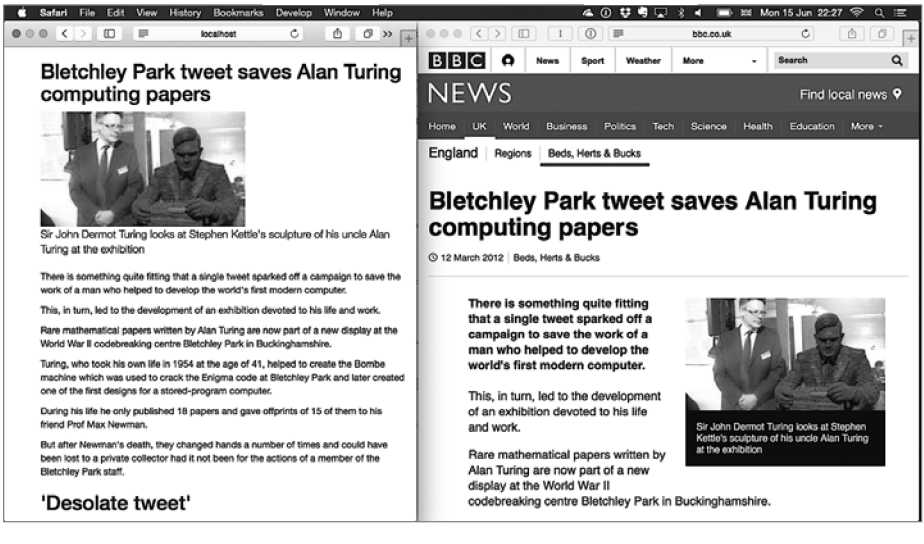


Рис. 3.1. Веб-приложение для отложенного чтения

На странице приложения слева с целевого сайта удалены все элементы навигации, оставлен только основной контент и заголовок. Что еще важнее, статья сохраняется в базе данных; это позволит вам прочитать ее позднее, когда оригинал может оказаться недоступным.

Прежде чем строить веб-приложение, следует сначала создать новый проект. В следующем проекте показано, как создать проект Node с нуля.

3.1. Структура веб-приложения Node

Типичное веб-приложение Node состоит из следующих компонентов:

- package.json — файл со списком зависимостей и командой запуска приложения;
- public/ — папка статических активов (CSS, клиентский код JavaScript);
- node_modules/ — место для установки зависимостей проекта;
- один или несколько файлов JavaScript с кодом приложения.

Код приложения часто подразделяется следующим образом:

- app.js или index.js — код подготовки приложения;
- models/ — модели баз данных;
- views/ — шаблоны, используемые для генерирования страниц приложения;
- controllers/ или routes/ — обработчики запросов HTTP;
- middleware/ — промежуточные компоненты.

Не существует правил, диктующих структуру приложения; обычно веб-фреймворки обладают достаточной гибкостью и требуют настройки. Тем не менее описанная схема встречается во многих проектах.

Всему этому гораздо проще научиться на практике. Давайте создадим заготовку веб-приложения так, как это делают опытные программисты Node.

3.1.1. Создание нового веб-приложения

Чтобы создать новое веб-приложение, необходимо создать новый проект Node. Если вам потребуется освежить память — обращайтесь к главе 2; а мы в двух словах напомним, что для этого нужно создать каталог и выполнить команду `npm init` с настройками по умолчанию:

```
mkdir later
cd later
npm init -fy
```

Новый проект создан; что дальше? Большинство людей добавят в npm модуль, упрощающий веб-разработку. В Node имеется встроенный модуль http с сервером, но проще воспользоваться чем-то таким, что сокращает служебный код, необходимый для типичных задач веб-разработки. А теперь посмотрим, как установить Express.

Добавление зависимости

Для добавления зависимости в проект используйте npm. Следующая команда устанавливает Express:

```
npm install --save express
```

Если теперь просмотреть файл **package.json**, вы увидите, что в него был добавлен модуль Express. Соответствующий фрагмент выглядит так:

```
"dependencies": {
  "express": "^4.14.0"
}
```

Модуль Express находится в папке **node_modules/** проекта. Если вы захотите удалить Express из проекта, выполните команду `npm rm express --save`. Модуль удаляется из **node_modules/** и обновляет файл **package.json**.

Простой сервер

Express ориентируется на построение модели приложения в контексте запросов и ответов HTTP и строится на базе встроенного модуля Node http. Чтобы создать простейшее приложение, следует создать экземпляр приложения вызовом `express()`, добавить обработчик маршрута, а затем связать приложение с портом TCP. Полный код примера:

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

const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(port, () =>
  console.log(`Express web app available at localhost: ${port}`);
};
```

Все не так сложно, как кажется! Сохраните код в файле с именем **index.js** и запустите его командой `node index.js`. Затем откройте страницу *http://localhost:3000* для просмотра результатов. Чтобы не приходилось запоминать, как именно должно запускаться каждое приложение, многие разработчики используют сценарии npm для упрощения процесса.

Сценарии npm

Чтобы сохранить команду запуска сервера (`node index.js`) как сценарий npm, откройте файл package.json и добавьте в раздел `scripts` новое свойство с именем `start`:

```
"scripts": {
  "start": "node index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Теперь приложение можно запустить командой `npm start`. Если вы получите ошибку из-за того, что порт 3000 уже используется на вашей машине, используйте другой порт командой `PORT=3001 npm start`. Сценарии npm используются для самых разных целей: построения пакетов на стороне клиента, запуска тестов и генерирования документации. В них можно разместить все, что угодно; по сути это механизм запуска мини-сценариев.

3.1.2. Сравнение с другими платформами

Для сравнения приведем эквивалентное приложение PHP Hello World:

```
<?php echo '<p>Hello World</p>'; ?>
```

Код помещается в одной строке, он интуитивно понятен — какими же преимуществами обладает более сложный пример Node? Различия проявляются в парадигме программирования: с PHP ваше приложение является страницей, а с Node — сервером. Пример для Node полностью управляет запросом и ответом, поэтому вы можете делать все, что угодно, без настройки сервера. Если вы хотите использовать сжатие HTTP или перенаправление URL, эти возможности можно реализовать как часть логики приложения. Вам не нужно отделять логику HTTP от логики приложения; и то и другое является частью приложения.

Вместо того чтобы создавать отдельную конфигурацию сервера HTTP, вы можете хранить ее в том же месте (то есть в том же каталоге). Это обстоятельство упрощает развертывание приложений Node и управление ими.

Другая особенность, упрощающая развертывание приложений Node, — npm. Поскольку зависимости устанавливаются на уровне проекта, у вас не будет конфликтов между проектами в одной системе.

3.1.3. Что дальше?

Итак, вы получили представление о создании проектов командой `npm init` и установке зависимостей командой `npm install --save`, вы сможете быстро создавать новые проекты. И это очень хорошо, потому что вы сможете опробовать новые идеи без конфликтов с другими проектами. Если появится новый замечательный

веб-фреймворк, который вам захочется опробовать, создайте новый каталог, выполните команду `npm init` и установите модуль из npm.

Когда все будет готово, можно переходить к написанию кода. На этой стадии в проект можно добавить файлы JavaScript и загрузить модули, установленные командой `npm --save`, вызовом `require`. Сосредоточимся на том, что большинство веб-разработчиков делает после этого: на добавлении REST-совместимых маршрутов. Это поможет вам определить API приложения и понять, какие модели базы данных нужны в вашей ситуации.

3.2. Построение REST-совместимой веб-службы

Наше приложение будет представлять собой REST-совместимую веб-службу, которая позволяет создавать и сохранять статьи по аналогии с Instapaper или Pocket. Она будет использовать модуль, который берет за основу исходную службу Readability (*www.readability.com*) для преобразования неряшливых веб-страниц в элегантные статьи, которые вы сможете прочитать позднее.

При проектировании REST-совместимой службы необходимо подумать над тем, какие операции вам потребуются, и связать их с маршрутами в Express. В данном случае необходимо иметь возможность сохранять статьи, организовывать их выборку для чтения, получать список всех статей и удалять те статьи, которые вам уже не нужны. Эти операции соответствуют следующим маршрутам:

- `POST /articles` — создание новой статьи;
- `GET /articles/:id` — получение одной статьи;
- `GET /articles` — получение всех статей;
- `DELETE /articles/:id` — удаление статьи.

Прежде чем переходить к таким вопросам, как базы данных и веб-интерфейсы, остановимся на создании REST-совместимых ресурсов в Express. Вы можете воспользоваться cURL для выдачи запросов к приложению-примеру, чтобы получить некоторое представление о происходящем, а потом перейти к более сложным операциям (таким, как сохранение данных), чтобы приложение было больше похоже на полноценное веб-приложение.

В следующем листинге приведено простое приложение Express, которое реализует эти маршруты с использованием массива JavaScript для хранения статей.

Листинг 3.1. Пример REST-совместимых маршрутов

```
const express = require('express');
const app = express();
const articles = [{ title: 'Example' }];
```

```
app.set('port', process.env.PORT || 3000);

app.get('/articles', (req, res, next) => {
  res.send(articles);
});

app.post('/articles', (req, res, next) => {
  res.send('OK');
});

app.get('/articles/:id', (req, res, next) => {
  const id = req.params.id;
  console.log('Fetching:', id);
  res.send(articles[id]);
});

app.delete('/articles/:id', (req, res, next) => {
  const id = req.params.id;
  console.log('Deleting:', id);
  delete articles[id];
  res.send({ message: 'Deleted' });
});

app.listen(app.get('port'), () => {
  console.log('App started on port', app.get('port'));
});

module.exports = app;
```

(1) Получает все статьи.

(2) Создает статью.

(3) Получает одну статью.

(4) Удаляет статью.

Сохраните этот код в файле index.js; если все сделано правильно, он должен запускаться командой `node index.js`.

Чтобы использовать этот пример, выполните следующие действия:

```
mkdir listing3_1
cd listing3_1
npm init -fy
run npm install --save express@4.12.4
```

Создание новых проектов Node более подробно рассматривается в главе 2.

ЗАПУСК ПРИМЕРОВ И ВНЕСЕНИЕ ИЗМЕНЕНИЙ

Чтобы запустить эти примеры, не забудьте перезапускать сервер после каждой модификации кода. Для этого остановите процесс Node клавишами Ctrl+C и запустите его снова командой node index.js.

Примеры приводятся в виде фрагментов, поэтому вы сможете последовательно объединять их для получения работоспособного приложения. Если они почему-либо не будут работать, попробуйте загрузить исходный код книги по адресу https://github.com/alexyoung/nodejsinaction.

В листинге 3.1 присутствует встроенный массив данных, который используется при выдаче списка всех статей в формате JSON **(1)** методом Express `res.send`. Express автоматически преобразует массив в действительный ответ JSON, что очень удобно при создании простых REST API.

Этот пример также может выдать ответ с одной статьей по тому же принципу **(3)**. Вы даже можете удалить статью **(4)**, использовав ключевое слово JavaScript `delete` с числовым идентификатором, заданным в URL. Чтобы получить значения из URL, включите их в строку маршрута (`/articles/:id`) и прочитайте нужное значение `req.params.id`.

Листинг 3.1 не позволяет создавать статьи **(2)**, потому что для этого нужен парсер тела запроса; эта тема рассматривается в следующем разделе. А пока посмотрим, как использовать этот пример с cURL (*http://curl.haxx.se*).

После запуска примера командой `node index.js` вы можете обращаться к нему с запросами из браузера или cURL. Чтобы получить одну статью, выполните следующую команду:

```
curl http://localhost:3000/articles/0
```

Чтобы получить все статьи, обратитесь с запросом к /articles:

```
curl http://localhost:3000/articles
```

Статьи даже можно удалять:

```
curl -X DELETE http://localhost:3000/articles/0
```

Но почему мы сказали, что вам не удастся создать статью? Главная причина заключается в том, что реализация запроса POST требует *разбора тела* запроса. Ранее в поставку Express входил встроенный парсер тела запроса, но возможных способов реализации было столько, что разработчики решили ввести отдельную зависимость.

Парсер тела запросов знает, как принимать тела запросов POST в кодировке *MIME* (Multipurpose Internet Mail Extensions) и преобразовывать их в данные, которые вы можете использовать в своем коде. Обычно вы получаете данные JSON, с которыми удобно работать. Каждый раз, когда вы отправляете данные формы на сайте, где-то в программном обеспечении на стороне сервера задействуется парсер тела запросов.

Чтобы добавить официально поддерживаемый парсер тела запросов, выполните следующую команду npm:

```
npm install --save body-parser
```

Теперь загрузите парсер тела запросов в своем приложении (поближе к началу файла), как показано в листинге 3.2. Если вы повторяете приводимые примеры, сохраните его в одной папке с листингом 3.1 (listing3_1), но мы также сохранили его в отдельной папке в исходном коде книги (ch03-what-is-a-node-web-app/listing3_2).

Листинг 3.2. Добавление парсера тела запросов

```
const express = require('express');
const app = express();
const articles = [{ title: 'Example' }];
const bodyParser = require('body-parser');

app.set('port', process.env.PORT || 3000);

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.post('/articles', (req, res, next) => {
  const article = { title: req.body.title };
  articles.push(article);
  res.send(article);
});
```

(1) Поддерживает тела запросов, закодированные в формате JSON.

(2) Поддерживает тела запросов в кодировке формы.

В этом листинге добавляются два полезных аспекта: разбор тела запроса в формате JSON **(1)** и тела запроса в кодировке формы **(2)**. Также добавляется базовая реализация создания статей: если вы создадите запрос POST с полем `title`, в массив статей будет добавлена новая статья! Соответствующая команда с URL выглядит так:

```
curl --data "title=Example 2" http://localhost:3000/articles
```

Сейчас мы подошли достаточно близко к построению реального веб-приложения. Остается добавить еще две возможности: механизм постоянного хранения данных в базе данных и механизм генерирования удобочитаемой версии статей, найденных в Интернете.

3.3. Добавление базы данных

Заранее определенного механизма включения баз данных в приложения Node не существует, но процесс обычно состоит из следующих шагов.

1. Определить базу данных, которую вы хотите использовать.
2. Изучить популярные модули npm, реализующие драйвер или объектно-реляционное отображение (ORM, Object-Relational Mapping).
3. Добавить в проект модуль командой `npm -save`.
4. Создать модули, инкапсулирующие обращения к базе данных в JavaScript API.
5. Добавить модели к маршрутам Express.

Прежде чем добавлять поддержку базы данных, продолжим работать с Express и разработаем код обработки маршрутов с шага 5. Обработчики маршрутов HTTP в Express-части приложения выдают простые вызовы к моделям баз данных. Пример:

```
app.get('/articles', (req, res, err) => {
  Article.all(err, articles) => {
    if (err) return next(err);
    res.send(articles);
  });
});
```

В данном случае маршрут HTTP предназначен для получения списка всех статей, поэтому метод модели может иметь вид `Article.all`. Его конкретная форма изменяется в зависимости от API баз данных; типичные примеры — `Article.find({}, cb)`[1] и `Article.fetchAll().then(cb)`[2]. Обратите внимание: `cb` в этих примерах — сокращение от «callback», то есть «обратный вызов».

Существует великое множество баз данных; как же определить, какая вам нужна? Ниже мы изложили причины, по которым для этого примера была выбрана база данных SQLite.

КАКАЯ БАЗА ДАННЫХ?

Для нашего проекта будет использоваться база данных SQLite (www.sqlite.org) с популярным модулем sqlite3 (http://npmjs.com/package/sqlite3). База данных SQLite удобна тем, что она является внутрипроцессной базой данных: вам не нужно устанавливать сервер, выполняемый на заднем плане в вашей системе. Любые добавляемые данные записываются в файл, который сохраняется при остановке и перезапуске приложения, поэтому SQLite хорошо подходит для начала работы с базами данных.

3.3.1. Проектирование собственного API модели

Приложение должно поддерживать создание, выборку и удаление статей. Следовательно, класс модели `Article` должен содержать следующие методы:

- `Article.all(cb)` — возвращает все статьи;
- `Article.find(id, cb)` — находит заданную статью по идентификатору;
- `Article.create({ title, content }, cb)` — создает статью с заголовком и контентом;
- `Article.delete(id, cb)` — удаляет статью по идентификатору.

Все эти методы можно реализовать с помощью модуля sqlite3. Этот модуль позволяет выбирать несколько строк результатов вызовом `db.all` или одну строку — вызовом `db.get`. Но сначала нужно создать подключение к базе данных.

[1] Mongoose: http://mongoosejs.com.

[2] Bookshelf.js http://bookshelfjs.org.

Листинг 3.3 демонстрирует выполнение этих операций с SQLite в Node. Этот код следует сохранить в файле **db.js** в одной папке с кодом из листинга 3.1.

Листинг 3.3. Модель Article

```
const sqlite3 = require('sqlite3').verbose();
const dbName = 'later.sqlite';
const db = new sqlite3.Database(dbName);

db.serialize(() => {
const sql = `
  CREATE TABLE IF NOT EXISTS articles
    (id integer primary key, title, content TEXT)
  `;
  db.run(sql);
});

class Article {
  static all(cb) {
    db.all('SELECT * FROM articles', cb);
  }

  static find(id, cb) {
    db.get('SELECT * FROM articles WHERE id = ?', id, cb);
  }

  static create(data, cb) {
    const sql = 'INSERT INTO articles(title, content) VALUES (?, ?)';
    db.run(sql, data.title, data.content, cb);
  }

  static delete(id, cb) {
    if (!id) return cb(new Error('Please provide an id'));
    db.run('DELETE FROM articles WHERE id = ?', id, cb);
  }
}

module.exports = db;
module.exports.Article = Article;
```

(1) Подключается к файлу базы данных.

(2) Создает таблицу articles, если она еще не существует.

(3) Выбирает все записи.

(4) Выбирает конкретную статью.

(5) Вопросительные знаки задают параметры.

В этом примере создается объект с именем `Article`, который может создавать, читать и удалять данные с использованием стандартного синтаксиса SQL и модуля sqlite3. Сначала файл базы данных открывается вызовом `sqlite3.Database` **(1)**, после чего создается таблица `articles` **(2)**. Синтаксис SQL `IF NOT EXISTS` здесь особенно удобен, потому что он позволяет снова выполнить код без случайного удаления и повторного создания таблицы `articles`.

Когда база данных и таблицы будут подготовлены, приложение готово к выдаче запросов. Для получения всех статей используется метод sqlite3 `all` **(3)**. Для получения конкретной статьи используется синтаксис запросов с вопросительным знаком **(4)**; модуль sqlite3 подставляет в запрос идентификатор. Наконец, вы можете вставлять и удалять данные методом `run` **(5)**.

Чтобы этот пример работал, необходимо установить модуль sqlite3 командой `npm install --save sqlite3`. На момент написания книги последней была версия 3.1.8.

После того как базовая функциональность баз данных будет подготовлена, ее необходимо добавить в маршруты HTTP из листинга 3.2.

Следующий листинг демонстрирует добавление всех методов, кроме POST. (Он рассматривается отдельно, потому что для него понадобится модуль readability, который мы еще не установили.)

Листинг 3.4. Добавление модуля Article к маршрутам HTTP

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const Article = require('./db').Article;  <-- (1) Загружает модуль базы данных.

app.set('port', process.env.PORT || 3000);

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/articles', (req, res, next) => {
  Article.all((err, articles) => {  <-- (2) Получает все статьи.
    if (err) return next(err);
    res.send(articles);
  });
});

app.get('/articles/:id', (req, res, next) => {
  const id = req.params.id;
  Article.find(id, (err, article) => {  <-- (3) Находит конкретную статью.
    if (err) return next(err);
    res.send(article);
  });
});

app.delete('/articles/:id', (req, res, next) => {
  const id = req.params.id;
  Article.delete(id, (err) => {  <-- (4) Удаляет статью.
    if (err) return next(err);
    res.send({ message: 'Deleted' });
  });
});

app.listen(app.get('port'), () => {
  console.log('App started on port', app.get('port'));
});

module.exports = app;
```

Листинг 3.4 предполагает, что вы сохранили листинг 3.3 в файле db.js в том же каталоге. Node загружает этот модуль **(1)** и затем использует его для выборки каждой статьи **(2)**, поиска конкретной статьи **(3)** и удаления **(4)**.

Последнее, что остается сделать, — добавить поддержку создания статей. Для этого необходимо иметь возможность загружать статьи и преобразовывать при помощи «волшебного» алгоритма удобочитаемости. Для этого нам понадобится модуль npm.

3.3.2. Преобразование статей в удобочитаемую форму и их сохранение для чтения в будущем

Итак, вы построили REST-совместимый API, а данные могут сохраняться в базе данных. Теперь следует добавить код для преобразования веб-страниц в их упрощенные версии «для чтения». К счастью, вам не придется делать это самостоятельно; вместо этого можно воспользоваться модулем из npm.

Поиск в npm по слову «readability» дает достаточно много модулей. Попробуем воспользоваться модулем node-readability (существует в версии 1.0.1 на момент написания книги). Установите модуль командой `npm install node-readability --save`. Модуль предоставляет асинхронную функцию для загрузки URL-адреса и преобразования разметки HTML в упрощенное представление.

Следующий фрагмент показывает, как используется модуль node-readability; если вы захотите опробовать его, добавьте следующий фрагмент в файл **index.js** в дополнение к листингу 3.5:

```
const read = require('node-readability');
const url = 'http://www.manning.com/cantelon2/';
read(url, (err, result)=>  {
  // result содержит .title и .content
});
```

Модуль node-readability может использоваться с классом базы данных для сохранения статей методом `Article.create`:

```
read(url, (err, result) => {
  Article.create(
    { title: result.title, content: result.content },
    (err, article) => {
      // Статья сохраняется в базе данных
    }
  );
});
```

Чтобы использовать модуль в приложении, откройте файл **index.js** и добавьте новый обработчик маршрута `app.post` для загрузки и сохранения статей. Объединяя все это с тем, что вы узнали о HTTP POST в Express и парсере тела запроса, мы получаем пример из листинга 3.5.

Листинг 3.5. Генерирование удобочитаемых статей и их сохранение

```
const read = require('node-readability');

// ... Оставшаяся часть файла index.js из листинга 3.4

app.post('/articles', (req, res, next) => {
  const url = req.body.url;  <-------  (1) Получает URL из тела POST.

read(url, (err, result) => {  <-------  (2) Использует режим удобочитаемости от выборки URL.
  if (err || !result) res.status(500).send('Error downloading article');
    Article.create(
      { title: result.title, content: result.content },
      (err, article) => {
        if (err) return next(err);
        res.send('OK');  <-------  (3) После сохранения статьи возвращает код 200.
      }
    );
  });
});
```

Здесь мы сначала получаем URL из тела POST **(1)**, а затем используем модуль node-readability для получения URL **(2)**. Статья сохраняется с использованием класса модели `Article`. Если произойдет ошибка, ее обработка передается промежуточному стеку Express **(3)**; в противном случае представление статьи в формате JSON отправляется клиенту.

Чтобы создать запрос POST, работающий с этим примером, используйте ключ `--data`:

```
curl --data "url=http://manning.com/cantelon2/" http://localhost:3000/articles
```

В предыдущем разделе мы добавили модуль базы данных, создали инкапсулирующий его JavaScript API и связали с REST-совместимым HTTP API. Это значительный объем работы, который составляет основную часть ваших усилий в качестве разработчика на стороне сервера. Базы данных будут более подробно рассмотрены позднее в этой книге, когда мы займемся MongoDB и Redis.

От сохранения статей и их программной выборки мы перейдем к добавлению веб-интерфейса, чтобы пользователи также могли прочитать сохраненные статьи.

3.4. Добавление пользовательского интерфейса

Добавление интерфейса в проект Express состоит из нескольких шагов. Первый шаг — использование ядра шаблонов; вскоре мы покажем, как установить его и выполнить визуализацию шаблонов.

Ваше приложение также должно предоставлять статические файлы (например, разметку CSS). Но прежде чем выполнять визуализацию шаблонов и писать стили CSS, вы должны узнать, как заставить обработчики маршрутов из предыдущих примеров отвечать данными в формате JSON и HTML при необходимости.

3.4.1. Поддержка разных форматов

До настоящего момента мы использовали метод `res.send()` для отправки объектов JavaScript клиенту. Для создания запросов использовался модуль cURL, и в данном случае формат JSON удобен, потому что легко читается с консоли. Но чтобы приложение могло реально использоваться, оно должно также поддерживать HTML. Как обеспечить поддержку обоих форматов?

Проще всего воспользоваться методом `res.format`, предоставляемым Express. Он позволяет приложению выдать ответ в подходящем формате в зависимости от вопроса. Для этого приложению предоставляется список форматов с функциями, которые отвечают нужным образом:

```
res.format({
  html: () => {
    res.render('articles.ejs', { articles: articles });
  },
  json: () => {
    res.send(articles);
  }
});
```

В этом фрагменте `res.render` выполняет визуализацию шаблона **articles.ejs** в папке **views**. Но чтобы этот способ сработал, необходимо установить ядро шаблонов и создать несколько шаблонов.

3.4.2. Визуализация шаблонов

Существует много ядер шаблонов; мы выберем простое ядро, которое достаточно легко изучается, — EJS (Embedded JavaScript). Установите модуль EJS из npm (EJS существует в версии 2.3.1 на момент написания книги):

```
npm install ejs --save
```

Теперь `res.render` может генерировать файлы HTML, отформатированные с использованием EJS. Если заменить `res.send(articles)` в обработчике маршрута `app.get('/articles')` из листинга 3.4, посещение адреса *http://localhost:3000/articles* в браузере приведет к попытке визуализации **articles.ejs**.

Далее необходимо создать шаблон **articles.ejs** в папке **views**. В листинге 3.6 приведен полный шаблон, который вы можете использовать.

Листинг 3.6. Шаблон для списка статей

```
<% include head %>  <------  (1) Включает другой шаблон.
<ul>
  <% articles.forEach((article) => { %>  <------  (2) Перебирает все статьи и проводит их визуализацию.
    <li>
      <a href="/articles/<%= article.id %>">
        <%= article.title %>  <------  (3) Включает заголовок статьи в качестве текста ссылки.
      </a>
    </li>
  <% }) %>
</ul>
<% include foot %>
```

Шаблон для списка статей использует шаблоны для заголовка **(1)** и завершителя, которые включаются в следующие примеры кода. Это делается для того, чтобы избежать дублирования заголовка и завершителя в каждом шаблоне. Для перебора списка статей **(2)** применяется стандартный цикл JavaScript `forEach`, после чего идентификаторы статей и заголовки внедряются в шаблон синтаксической конструкцией EJS `<%= value %>` **(3)**.

Пример шаблона заголовка из файла views/head.ejs:

```
<html>
  <head>
    <title>Later</title>
  </head>
  <body>
    <div class="container">
```

А вот соответствующий завершитель (сохраняется в файле views/foot.ejs):

```
    </div>
  </body>
</html>
```

Метод `res.format` также может использоваться для вывода конкретных статей. Здесь ситуация становится более интересной, потому что для того, чтобы приложение приносило реальную пользу, статьи должны быть аккуратно оформлены и легко читаться.

3.4.3. Использование npm для зависимостей на стороне клиента

Когда шаблоны будут на своих местах, пора сделать следующий шаг — добавить стилевое оформление. Вместо того чтобы создавать таблицу стилей, проще повторно использовать существующие стили, причем это можно сделать даже с npm! Популярный клиентский фреймворк Bootstrap (*http://getbootstrap.com/*) доступен в npm (*www.npmjs.com/package/bootstrap*); добавьте его в проект:

```
npm install bootstrap --save
```

Заглянув в каталог node_modules/bootstrap/, вы найдете в нем исходный код проекта Bootstrap. В папке dist/css хранятся файлы CSS, поставляемые с Bootstrap. Чтобы использовать их в проекте, приложение должно быть способно поставлять статические файлы.

Статические файлы

Для отправки браузеру клиентского кода JavaScript, графики и CSS в Express имеется встроенная прослойка `express.static`. Чтобы использовать ее, передайте каталог со статическими файлами, и эти файлы станут доступными для браузера.

В начале файла приложения Express (index.js) имеются строки, загружающие программные прослойки, необходимые для проекта:

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Чтобы загрузить CSS фреймворка Bootstrap, используйте `express.static` для регистрации файла с нужным URL-адресом:

```
app.use(
  '/css/bootstrap.css',
  express.static('node_modules/bootstrap/dist/css/bootstrap.css')
);
```

После этого можно добавить файл /css/bootstrap.css в шаблоны, чтобы использовать впечатляющие стили Bootstrap. Файл views/head.ejs должен выглядеть примерно так:

```
<html>
  <head>
    <title>later;</title>
    <link rel="stylesheet" href="/css/bootstrap.css">
  </head>
  <body>
    <div class="container">
```

Это только разметка CSS; в поставку Bootstrap также входят другие файлы, включая значки, шрифты и плагины jQuery. Вы можете добавить в свой проект другие файлы Bootstrap или же упаковать их в один файл для удобства загрузки.

Npm и разработка на стороне клиента: новые возможности

В предыдущем примере продемонстрировано простое использование библиотеки, предназначенной для браузеров, с помощью npm. Веб-разработчики обычно загружают файлы Bootstrap и добавляют их в проект вручную (как правило, веб-разработчики, занимающиеся простыми статическими сайтами).

Однако современные разработчики клиентской (интерфейсной) части используют npm как для получения библиотек, так и для загрузки их в клиентском коде JavaScript. Такие инструменты, как Browserify (*http://browserify.org/*) и webpack (*http://webpack.github.io/*), предоставляют в ваше распоряжение всю мощь npm и `require` для загрузки зависимостей. Представьте, что вы можете ввести команду `const React = require('react')` не только в коде Node, но и в коде разработчика клиентской части! Эта тема выходит за рамки настоящей главы, и все же она дает некоторое представление о том, чего можно добиться при объединении механизмов программирования Node с приемами программирования клиентской части.

3.5. Заключение

- Веб-приложение Node можно быстро построить с нуля при помощи команды `npm init` и Express.
- Для установки зависимостей используется команда `npm install`.
- Express позволяет создавать веб-приложения с использованием REST-совместимых API.
- Выбор подходящей базы данных и модуля базы данных требует предварительного анализа и зависит от конкретных требований.
- SQLite хорошо подходит для малых проектов.
- EJS предоставляет простые средства для визуализации шаблонов в Express.
- Express поддерживает разные ядра шаблонов, включая Pug и Mustache.

II Веб-разработка с использованием Node

Итак, вы готовы к более глубокому изучению разработки на стороне сервера. Технология Node нашла важную нишу за пределами серверного кода: системы построения фронтэнда. В этой части вы узнаете, как запускать проекты с использованием webpack и Gulp. Также мы представим самые популярные веб-фреймворки и сравним их с точки зрения разных разработчиков, чтобы помочь вам определиться с выбором идеального фреймворка для ваших проектов.

Если вы захотите больше узнать о Connect и Express, глава 6 полностью посвящена построению веб-приложений с этими модулями. Также имеется глава, посвященная шаблонам и использованию баз данных с Node.

Обзор полностековой веб-разработки на базе Node завершается главами, посвященными тестированию и развертыванию. Они помогут вам подготовить свое первое приложение Node к работе.

4 Системы построения фронтэнда

В современной веб-разработке Node все чаще используется для запуска инструментов и сервисов, от которых зависят фронтэнд-разработчики. Возможно, вам как Node-программисту придется отвечать за настройку и сопровождение этих инструментов. А как полностековому разработчику вам стоит использовать эти инструменты для создания более быстрых и надежных веб-приложений. В этой главе вы научитесь использовать сценарии npm, Gulp и webpack для построения проектов, удобных в сопровождении.

Преимущества от использования систем построения фронтэнда могут быть огромными. Такие системы помогают писать более понятный и устойчивый к будущим изменениям код. Нет необходимости беспокоиться о поддержке браузером ES2015, если приложение обрабатывается транспилятором Babel. Кроме того, благодаря возможности генерирования карт исходного кода, сохраняется возможность отладки на базе браузера.

В следующем разделе приводится краткое введение во фронтэнд-разработку с использованием Node. После этого будут рассмотрены примеры современных технологий — например, React, которые вы сможете использовать в своих проектах.

4.1. Фронтэнд-разработка с использованием Node

В последнее время как разработчики фронтэнда, так и разработчики кода на стороне сервера стали применять npm для передачи JavaScript. Это означает, что npm используется как для модулей фронтэнда (таких, как React), так и для серверного кода (например, Express). Однако некоторые модули трудно четко отнести к той или иной стороне: lodash — пример библиотеки общего назначения, которая может использоваться Node и браузерами. При тщательной упаковке lodash один модуль

может использоваться как Node, так и браузерами, а зависимостями в проекте можно будет управлять при помощи npm.

Возможно, вам уже встречались другие модульные системы, предназначенные для разработки на стороне клиента, — такие, как Bower (*http://bower.io/*). Никто не запрещает вам использовать их, но как разработчику Node вам стоит подумать об использовании npm.

Впрочем, распространение пакетов — не единственная область применения Node. Фронтэнд-разработчики все чаще полагаются на средства создания портируемого кода JavaScript с обратной совместимостью. Транспиляторы — такие, как Babel (*https://babeljs.io/*), — используются для преобразования современного кода ES2015 в более широко поддерживаемый код ES5. Среди других средств можно упомянуть минификаторы (например, UglifyJS; *https://github.com/mishoo/UglifyJS*) и инструменты статического анализа (например, ESLint; i) для проверки правильности кода перед его распространением.

Node также часто управляет системами запуска тестов. Вы можете запускать тесты для UI-кода в процессе Node или же использовать сценарий Node для управления тестами, выполняемым в браузере.

Также эти средства довольно часто используются вместе. Когда вы начинаете жонглировать траспилятором, минификатором, программой статического анализа и системой запуска тестов, вам нужно будет каким-то образом зафиксировать, как работает процесс построения. В одних проектах используются сценарии npm; в других — используют Gulp или webpack. В этой главе мы рассмотрим все эти подходы, а также некоторые практические приемы, связанные с ними.

4.2. Использование npm для запуска сценариев

Node поставляется с npm, а в npm существуют встроенные средства для запуска сценариев. Следовательно, вы можете рассчитывать на то, что коллеги или пользователи смогут выполнять такие команды, как `npm start` и `npm test`. Чтобы добавить собственную команду для `npm start`, включите ее в свойство `scripts` файла `package.json` своего проекта:

```
{
  ...
  "scripts": {
    "start": "node server.js"
  },
  ...
}
```

Даже если не определять `start`, значение `node server.js` используется по умолчанию; строго говоря, вы можете оставить его пустым — только не забудьте создать файл

с именем server.js. Также полезно определить свойство `test`, потому что вы можете включить свой тестовый фреймворк как зависимость и запустить его командой `npm test`. Предположим, вы используете для тестирования фреймворк Mocha (*www.npmjs.com/package/mocha*) и установили его командой `npm install --save-dev`. Чтобы вам не приходилось устанавливать Mocha глобально, вы можете добавить следующую команду в файл package.json:

```
{
  ...
  "scripts": {
    "test": "./node_modules/.bin/mocha test/*.js"
  },
  ...
}
```

Обратите внимание: в предыдущем примере Mocha передаются аргументы. Также при запуске сценариев npm можно передать аргументы через два дефиса:

```
npm test -- test/*.js
```

В табл. 4.1 приведена сводка некоторых команд npm.

Таблица 4.1. Команды npm

Команда	Свойство package.json	Примеры использования
start	scripts.start	Запуск сервера веб-приложения или приложения Electron
stop	scripts.stop	Остановка веб-сервера
restart		Последовательное выполнение команд stop и start
install, postinstall	scripts.install, scripts.postinstall	Выполнение собственных команд построения после установки пакета. Обратите внимание: postinstall может выполняться только в команде npm run postinstall

Поддерживаются многие команды, включая команды очистки пакетов перед публикацией и команды миграции между версиями пакетов. Впрочем, для большинства задач веб-разработки вам хватит команд `start` и `test`.

Многие задачи, которые вы будете определять, не укладываются в поддерживаемые имена команд. Например, предположим, что вы работаете над простым проектом, написанным на ES2015, и хотите транспилировать его на ES5. Это можно сделать командой `npm run`. В следующем разделе рассматривается учебный пример, в котором будет создан новый проект для построения файлов ES2015.

4.2.1. Создание специализированных сценариев npm

Команда `npm run` (синоним для `npm run-script`) используется для определения произвольных сценариев, которые запускаются командой `npm run` *имя-сценария*. Посмотрим, как создать такой сценарий для построения сценария на стороне клиента с использованием Babel.

Создайте новый проект и установите необходимые зависимости:

```
mkdir es2015-example
cd es2015-example
npm init -y
npm install --save-dev babel-cli babel-preset-es2015
echo '{ "presets": ["es2015"] }' > .babelrc
```

В результате выполнения этих команд должен быть создан новый проект Node с базовыми инструментами и плагинами Babel для ES2015. Затем откройте файл package.json и добавьте в раздел `scripts` свойство `babel`.

Оно должно выполнять сценарий, установленный в папке node_modules/.bin проекта:

```
"babel": "./node_modules/.bin/babel browser.js -d build/"
```

Ниже приведен пример файла в синтаксисе ES2015, который вы можете использовать; сохраните его в файле browser.js:

```
class Example {
  render() {
    return '<h1>Example</h1>';
  }
}
const example = new Example();
console.log(example.render());
```

Чтобы протестировать этот пример, введите команду `npm run babel`. Если все было настроено правильно, должна появиться папка построения с файлом browser.js. Откройте browser.js и убедитесь в том, что это действительно файл ES5 (поищите конструкцию вида `var _createClass` в начале файла.

Если ваш проект при построении ничего более не делает, ему можно присвоить имя `build` вместо `babel` в файле package.json. Но можно пойти еще дальше и добавить UglifyJS:

```
npm i --save-dev uglify-es
```

UglifyJS вызывается командой `node_modules/.bin/uglifyjs`; добавьте эту команду в package.json из раздела `scripts` с именем `uglify`:

```
./node_modules/.bin/uglifyjs build/browser.js -o build/browser.min.js
```

Теперь вы сможете выполнить команду `npm run uglify`. Всю функциональность можно связать воедино, объединив оба сценария. Добавьте еще одно свойство `script` с именем `build` для вызова обеих задач:

```
"build": "npm run babel && npm run uglify"
```

Оба сценария запускаются командой `npm run build`. Участники вашей команды могут объединить несколько средств упаковки клиентской части вызовом этой простой команды. Такое решение работает, потому что Babel и UglifyJS могут выполняться как сценарии командной строки, они получают аргументы командной строки и легко добавляются как однострочные команды в файл package.json. Также Babel позволяет определить сложное поведение в файле `.babelrc`, что было сделано ранее в этой главе.

4.2.2. Настройка средств построения фронтэнда

В общем случае возможны три варианта настройки средств построения фронтэнда при использовании со сценариями npm:

- передача аргументов командной строки (например, `./node_modules/.bin/uglify --source-map`);
- создание конфигурационного файла для конкретного проекта. Часто применяется для Babel и ESLint;
- включение параметров конфигурации в файл package.json. Babel также поддерживает эту возможность.

Что, если ваши требования к построению содержат дополнительные этапы с такими операциями, как копирование, конкатенация или перемещение файлов? Можно создать сценарий командного интерпретатора и запустить его из сценария npm, но вашим коллегам, сведущим в JavaScript, будет удобнее, если вы будете использовать JavaScript. Многие системы построения предоставляют JavaScript API для автоматизации построения. В следующем разделе описано одно из таких решений: Gulp.

4.3. Автоматизация с использованием Gulp

Gulp (*http://gulpjs.com/*) — система построения, работающая на основе потоков (streams). Пользователь может сводить потоки воедино для создания процессов построения, не ограничивающихся простой транспиляцией или минификацией кода. Представьте, что у вас имеется проект с административной областью, построенной на базе Angular, и с общедоступной областью, построенной на базе React; оба подпроекта имеют некоторые общие требования к построению. С Gulp вы сможете повторно использовать части процесса построения для каждой стадии. На рис. 4.1 представлены примеры двух процессов построения с общей функциональностью.

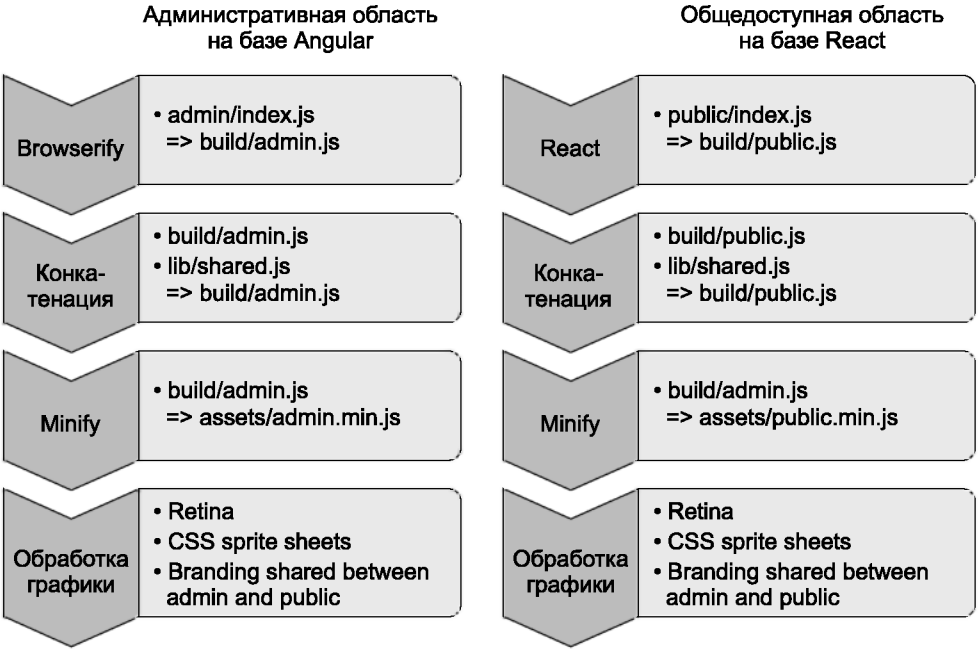


Рис. 4.1. Два процесса построения с общей функциональностью

Gulp позволяет обеспечить высокую степень повторного использования кода с помощью двух приемов: применения плагинов и определения ваших собственных задач построения. Как видно из иллюстрации, процесс построения представляет собой поток, поэтому задачи и плагины можно объединять в цепочку. Например, React-часть приведенного примера можно обработать с использованием Gulp Babel (*www.npmjs.com/package/gulp-babel/*) и встроенных средств gulp.src:

```
gulp.src('public/index.jsx')
  .pipe(babel({
    presets: ['es2015', 'react']
  }))
  .pipe(minify())
  .pipe(gulp.dest('build/public.js'));
```

В эту цепочку можно даже достаточно легко добавить стадию конкатенации. Но, прежде чем описывать синтаксис более подробно, посмотрим, как создать небольшой проект Gulp.

4.3.1. Добавление Gulp в проект

Чтобы добавить Gulp в проект, необходимо установить пакеты gulp-cli и gulp из npm. Многие пользователи устанавливают gulp-cli глобально, поэтому примеры

Gulp можно запустить простым вводом команды `gulp`. Учтите, что, если пакет gulp был ранее установлен глобально, следует выполнить команду `npm rm --global gulp`. Выполните следующий фрагмент, чтобы установить gulp-cli глобально и создать новый проект Node с зависимостью от Gulp:

```
npm i --global gulp-cli
mkdir gulp-example
cd gulp-example
npm init -y
npm i –save-dev gulp
```

Затем создайте файл gulpfile.js:

```
touch gulpfile.js
```

Откройте файл. Далее мы воспользуемся Gulp для построения небольшого проекта React, в котором используются пакеты gulp-babel (*www.npmjs.com/package/gulp-babel*), gulp-sourcemaps и gulp-concat:

```
npm i --save-dev gulp-sourcemaps gulp-babel babel-preset-es2015
npm i --save-dev gulp-concat react react-dom babel-preset-react
```

Не забудьте использовать npm с ключом `--save-dev`, когда вы хотите добавить плагины Gulp в проект. Если вы экспериментируете с новыми плагинами и позднее решите удалить их, используйте команду `npm uninstall --save-dev`, чтобы удалить их из ./node_modules и обновить файл package.json проекта.

4.3.2. Создание и выполнение задач Gulp

Создание задач Gulp требует написания кода Node с Gulp API в файле с именем gulpfile.js. Gulp API содержит методы для таких операций, как поиск файлов и передача их через плагины, которые каким-то образом их преобразовывают.

Попробуйте сами: откройте файл gulpfile.js и создайте задачу построения, которая использует gulp.src для поиска файлов JSX, Babel для обработки ES2015 и React, а затем выполняет конкатенацию для слияния файлов, как показано в листинге 4.1.

Листинг 4.1. Gulp-файл для ES2015 и React с использованием Babel

```
const gulp = require('gulp');
const sourcemaps = require('gulp-sourcemaps');
const babel = require('gulp-babel');
const concat = require('gulp-concat');

gulp.task('default', () => {
  return gulp.src('app/*.jsx')
    .pipe(sourcemaps.init())
    .pipe(babel({
```

Плагины Gulp загружаются так же, как и стандартные модули Node.

Встроенные средства gulp.src используются для поиска всех файлов React с расширением JSX.

Запускает анализ файлов для построения отладочных карт исходного кода.

```
      presets: ['es2015', 'react'] ◄── Настраивает gulp-babel для использования ES2015 и React (jsx).
    }))
    .pipe(concat('all.js'))    ◄──── Объединяет все файлы с исходным кодом в all.js.
    .pipe(sourcemaps.write('.'))  ◄──── Записывает файлы с картами отдельно.
    .pipe(gulp.dest('dist'));    ◄──── Перенаправляет все файлы в dist/folder.
});
```

В листинге 4.1 используются плагины Gulp для получения, обработки и записи файлов. Сначала все входные файлы находятся по шаблону, после чего плагин gulp-sourcemaps используется для сбора метрик карты исходного кода для отладки на стороне клиента. Обратите внимание на то, что для работы с картами исходного кода нужны две фазы: в одной вы указываете, что хотите использовать карты, а в другой записываете файлы карт. Также gulp-babel настраивается для обработки файлов с использованием ES2015 и React.

Эта задача Gulp может быть выполнена командой gulp в терминале.

В этом примере все файлы преобразуются с использованием одного плагина. Так уж вышло, что Babel и транспилирует код React JSX, и преобразует ES2015 и ES5. Когда это будет сделано, выполняется конкатенация файлов с использованием плагина gulp-concat. После завершения транспиляции происходит безопасная запись карт исходного кода, и итоговая сборка помещается в папку **dist**.

Чтобы опробовать этот gulp-файл, создайте файл JSX с именем **app/index.jsx**. Простой файл JSX, который может использоваться для тестирования Gulp, может выглядеть так:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('example')
);
```

Gulp позволяет легко описать стадии построения на JavaScript, а при помощи метода `gulp.task()` вы сможете добавить в этот файл собственные задачи. Задачи обычно строятся по одной схеме:

- ввод — получение исходных файлов;
- траспиляция — прохождение через плагин, который преобразует их;
- конкатенация — объединение файлов для создания монолитной сборки;
- вывод — назначение местонахождения или перемещение выходных файлов.

В предыдущем примере `sourcemaps` является особым случаем, потому что требует двух конвейеров: для конфигурации и для вывода файлов. Это логично, потому

что карты исходного кода зависят от соответствия исходной нумерации строк и нумерации строк в транспилированной сборке.

4.3.3. Отслеживание изменений

Последнее, что потребуется фронтэнд-разработчику, — цикл «построение/обновление». Для упрощения процесса легче всего воспользоваться плагином Gulp для отслеживания изменений в файловой системе. Впрочем, существуют и альтернативные решения. Некоторые библиотеки хорошо работают с «горячей» перезагрузкой, более общие проекты на базе DOM и CSS хорошо работают с проектом LiveReload (*http://livereload.com/*).

Например, к проекту из листинга 4.1 можно добавить gulp-watch (*www.npmjs.com/package/gulp-watch*). Добавьте пакет в проект:

```
npm i --save-dev gulp-watch
```

Не забудьте загрузить пакет в gulpfile.js:

```
const watch = require('gulp-watch');
```

А теперь добавьте задачу `watch`, которая вызывает задачу по умолчанию из предыдущего примера:

```
gulp.task('watch', () => {
  watch('app/**.jsx', () => gulp.start('default'));
});
```

Этот фрагмент определяет задачу с именем `watch`, а затем использует вызов `watch()` для отслеживания изменений в файлах React JSX. При каждом изменении файла выполняется задача построения по умолчанию. С небольшими изменениями этот рецепт может использоваться для построения файлов SASS (Syntactically Awesome Style Sheets), оптимизации графики и вообще практически всего, что только может понадобиться в проектах клиентской части.

4.3.4. Использование отдельных файлов в больших проектах

По мере роста проекта обычно приходится добавлять новые задачи Gulp. Рано или поздно образуется большой файл, в котором трудно разобраться. Впрочем, проблема решается: разбейте свой код на модули.

Как вы уже видели, Gulp использует систему модулей Node для загрузки плагинов. Специальной системы загрузки плагинов не существует; используются стандартные модули. Также система модулей Node может использоваться для разбивки длинных gulp-файлов с целью упрощения сопровождения. Чтобы использовать отдельные файлы, выполните следующие действия.

1. Создайте папку с именем gulp, а в ней — вложенную папку tasks.
2. Определите свои задачи с использованием обычного синтаксиса `gulp.task()` в отдельных файлах. Хорошее практическое правило — создать отдельный файл для каждой задачи.
3. Создайте файл с именем gulp/index.js для включения всех файлов задач Gulp.
4. Включите файл gulp/index.js в gulpfile.js.

Иерархия файлов должна выглядеть так:

```
gulpfile.js
gulp/
gulp/index.js
gulp/tasks/development-build.js
gulp/tasks/production-build.js
```

Описанный прием помогает упорядочить структуру проектов со сложными задачами построения, но он также может работать в сочетании с модулем gulp-help (*www.npmjs.com/package/gulp-help*). Этот модуль позволяет документировать задачи Gulp; команда `gulp help` выводит информацию о каждой задаче. Он удобен при работе в группе или если вам приходится работать со множеством проектов, использующих Gulp. На рис. 4.2 показано, как выглядит результат выполнения команды.

```
~/Projects/world-domination: gulp help
[10:33:38] Using gulpfile ~/Projects/world-domination/gulpfile.js
[10:33:38] Starting 'help'...

Usage
  gulp [TASK] [OPTIONS...]

Available tasks
  help     Display this help text.
  version  prints the version. Aliases: v, V

[10:33:38] Finished 'help' after 1.2 ms
```

Рис. 4.2. Пример вывода gulp-help

Gulp — средство автоматизации проектов общего назначения. Gulp хорошо работает при добавлении в проекты кроссплатформенных сценариев для выполнения служебных операций — например, проведения сложных тестов на стороне клиентов или создания тестов над общим набором объектов для баз данных. Хотя Gulp может использоваться для построения активов на стороне клиента, для этой цели также существуют специальные инструменты — как правило, они требуют меньшего объема кода и настройки, чем Gulp. Одним из таких инструментов является webpack — пакет, предназначенный для упаковки модулей JavaScript и CSS. В следующем разделе продемонстрировано использование webpack в проекте React.

4.4. Построение веб-приложений с использованием webpack

Пакет webpack специально предназначен для построения веб-приложений. Представьте, что вы работаете с дизайнером, который уже создал статический сайт для одностраничного веб-приложения, и теперь хотите адаптировать его для построения более эффективного кода ES2015 JavaScript и CSS. С Gulp вы пишете код JavaScript для управления системой построения, поэтому вам предстоит написать gulp-файл и несколько других задач. С webpack вы пишете конфигурационный файл, а затем подключаете новую функциональность при помощи плагинов и загрузчиков. В некоторых случаях никакая дополнительная настройка вообще не требуется; достаточно ввести команду `webpack` в командной строке, передать в аргументе путь к исходному коду — и она построит проект. Если вас интересует, как это выглядит, обратитесь к разделу 4.4.4.

Одно из преимуществ webpack заключается в том, что эта система позволяет быстро настроить систему построения, поддерживающую инкрементный режим. Если настроить ее для автоматического построения при изменении файлов, ей не обязательно будет перестраивать весь проект при изменении одного файла. В результате процесс построения упрощается и становится более понятным.

В этом разделе показано, как использовать webpack для небольших проектов React. Но сначала определимся с терминологией, принятой в webpack.

4.4.1. Пакеты и плагины

Прежде чем создавать проект webpack, необходимо разобраться с терминологией. Плагины webpack изменяют поведение процесса построения. Например, они могут включать такие операции, как автоматическая отправка активов в Amazon S3 (*https://github.com/MikaAK/s3-plugin-webpack*) или удаление дубликатов файлов из вывода.

В отличие от плагинов, *загрузчики* (loaders) представляют преобразования, применяемые к файлам ресурсов. Если вам потребуется преобразовать SASS в CSS (или ES2015 в ES5), значит, вам нужен загрузчик. Загрузчики представляют собой функции, преобразующие входной текст в выходной; они могут работать асинхронно или синхронно. Плагины представляют собой экземпляры классов, которые могут подключаться к webpack API более низкого уровня.

Если вам нужно преобразовать код React, CoffeeScript, SASS или любого другого транспилируемого языка, вам нужен *загрузчик*. Если же вам нужно обработать JavaScript или выполнить какие-то операции с группами файлов, вам нужен *плагин*.

В следующем разделе вы увидите, как использовать загрузчик Babel для преобразования проекта React ES2015 в пакет, адаптированный для браузера.

4.4.2. Настройка и запуск webpack

В этом разделе мы воссоздадим пример React из листинга 4.1 с использованием webpack. Для начала установите React в новом проекте:

```
mkdir webpack-example
npm init -y
npm install --save react react-dom
npm install --save-dev webpack babel-loader babel-core
npm install --save-dev babel-preset-es2015 babel-preset-react
```

Последняя строка устанавливает плагин Babel для ES2015 и преобразователь React для Babel. Далее необходимо создать файл с именем **webpack.config.js**, который сообщает webpack, где искать входной файл, куда записать результат и какие загрузчики следует использовать. Мы воспользуемся загрузчиком `babel-loader` с дополнительными настройками для React, как показано в следующем листинге.

Листинг 4.2. Файл webpack.config.js

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: './app/index.jsx',    <-- Входной файл
  output: { path: __dirname, filename: 'dist/bundle.js' },    <-- Выходной файл.

  module: {
    loaders: [
      {
        test: /.jsx?$/,    <-- Находит все файлы JSX.
        loader: 'babel-loader',
        exclude: /node_modules/,
        query: {
          presets: ['es2015', 'react']    <-- Использует плагины React и Babel ES2015.
        }
      }
    ]
  },
};
```

Конфигурационный файл содержит все необходимое для успешного построения приложений React с ES2015. Процесс настройки достаточно прост: определите значение `entry` с главным файлом, загружающим приложение. Затем укажите каталог, в который должен записываться вывод; если каталог еще не существует, он будет создан. Затем определите загрузчик и свяжите его с поиском файлов по шаблону при помощи свойства `test`. Наконец, задайте все необходимые параметры загрузчика. В данном примере эти параметры загружают Babel-плагины ES2015 и React.

Включите код React JSX в файл **app/index.jsx**; используйте фрагмент из раздела 4.3.2. Теперь команда `./node_modules/.bin/webpack` откомпилирует ES5-версию файла с зависимостями React.

4.4.3. Использование сервера для разработки webpack

Если вы хотите избежать необходимости перестраивать проект при каждом изменении файла React, используйте сервер для разработки webpack (*http://webpack.github.io/docs/webpack-dev-server.html*). В исходном коде книги он размещается в примере webpack-hotload-example (ch04-front-end/webpack-hotload-example). Этот компактный сервер Express запускает webpack с конфигурационным файлом webpack при изменении файлов, а затем предоставляет изменившиеся активы браузеру. Запустите его с указанием порта, отличного от порта основного веб-сервера; это означает, что теги `script` должны включать другие URL-адреса на стадии разработки. Сервер строит активы и сохраняет их в памяти (вместо выходной папки webpack). Сервер для разработки также может использоваться для горячей загрузки модулей (по аналогии с серверами LiveReload).

Чтобы добавить сервер для разработки webpack, выполните следующие действия.

1. Установите `webpack-dev-server` командой `npm i --save-dev webpack-dev-server@1.14.1`.
2. Добавьте параметр `publicPath` в раздел `output` в файле webpack.config.js.
3. Добавьте в каталог построения файл index.html, который будет управлять загрузкой пакетов JavaScript и CSS. Проследите за тем, чтобы порт совпадал с портом, заданным на следующем шаге.
4. Запустите сервер с параметрами, например `webpack-dev-server --hot --inline --content-base dist/ --port 3001`.
5. Посетите *http://localhost:3001/* и загрузите приложение.

Откройте файл webpack.config.js из листинга 4.2, измените свойство `output` и добавьте в него `publicPath`:

```
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'bundle.js',
  publicPath: '/assets/'
},
```

Создайте новый файл dist/index.html, как показано в листинге 4.3.

Листинг 4.3. Пример шаблона HTML для веб-приложения React

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Warning: Dev server only</title>
</head>
<body>
  <div id="example"></div>
```

```
  <script src="/assets/bundle.js"></script>
</body>
</html>
```

Общедоступный путь пакета, построенного с webpack.

Откройте package.json и добавьте команду запуска сервера webpack в свойство `scripts`:

```
"scripts": {
  "server:dev": "webpack-dev-server --hot -inline
     --content-base dist/ --port 3001"
 },
```

Параметр `--hot` заставляет сервер для разработки использовать режим горячей перезагрузки модулей. Если отредактировать React-файл примера в app/index.jsx, браузер должен обновиться. Механизм обновления задается параметром `--inline`. В режиме `inline` сервер для разработки внедряет код для управления обновлением пакета. Также существует версия `iframe`, которая заключает всю страницу в `iframe`.

Теперь запустите сервер для разработки:

```
npm run server:dev
```

Запуск сервера для разработки webpack инициирует построение и запускает сервер с прослушиванием порта 3001. Чтобы протестировать результат, введите адрес *http://localhost:3001* в браузере.

ГОРЯЧАЯ ПЕРЕЗАГРУЗКА

Из-за React и других фреймворков, включая AngularJS, существуют проекты горячей перезагрузки модулей, предназначенные для конкретных фреймворков. Некоторые из них задействуют фреймворки потоков данных (такие, как Redux и Relay); это означает, что код может обновляться с сохранением текущего состояния. Это идеальный способ выполнения перезагрузки кода, потому что вам не придется заново проделывать все необходимое для воссоздания состояния пользовательского интерфейса, над которым вы работаете. Однако приведенный пример в меньшей степени привязан к React и хорошо подходит для начала работы с серверами для разработки webpack. Обязательно поэкспериментируйте и найдите тот вариант, который лучше подходит для вашего проекта.

4.4.4. Загрузка модулей и активов CommonJS

В этой главе мы использовали React и Babel, но если вы используете webpack с более традиционными проектами CommonJS, webpack может предоставить всю необходимую функциональность без браузерной оболочки совместимости (shim) CommonJS. Он даже может загружать файлы CSS.

WEBPACK и COMMONJS

Чтобы использовать синтаксис модулей CommonJS с webpack, вам не придется ничего настраивать. Допустим, имеется файл, в котором используется `require`:

```
const hello = require('./hello');
hello();
```

И другой файл, определяющий функцию `hello`:

```
module.exports = function() {
  return 'hello';
};
```

В этом случае будет достаточно небольшого конфигурационного файла webpack для определения точки входа (первый фрагмент) и выходного пути построения:

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: './app/index.js',
  output: { path: __dirname, filename: 'dist/bundle.js' },
};
```

Этот пример показывает, насколько сильно различаются Gulp и webpack. webpack полностью специализируется на построении пакетов и в контексте этой задачи может генерировать пакеты с оболочками совместимости CommonJS. Открыв файл **dist/bundle.js**, вы увидите в начале файла оболочку совместимости `webpackBootstrap`, а затем все файлы из исходного дерева кода, включенные в замыкания для моделирования системы модулей. Следующий фрагмент является частью пакета:

```
function(module, exports, __webpack_require__) {

  const hello = __webpack_require__(1);

  hello();

  /***/ },
  /* 1 */
  /***/ function(module, exports) {

  module.exports = function() {
    return 'hello';
  };
```

Комментарии в коде показывают, где определяются модули, а файлы получают доступ к объектам `module` и `exports` как аргументам своих замыканий для моделирования API модуля CommonJS.

Использование пакетов NPM с webpack

Вы можете сделать следующий шаг и включить модули, загруженные из npm. Допустим, вы хотите использовать jQuery. Вместо того чтобы включать соответствующий тег `script` в страницу, вы можете установить jQuery командой `npm i --save-dev jquery`, а потом загрузить как обычный модуль Node:

```
const jquery = require('jquery');
```

Это означает, что webpack предоставляет в ваше распоряжение модули CommonJS и доступ к модулям из npm без какой-либо дополнительной конфигурации!

> **ПОИСК ЗАГРУЗЧИКОВ И ПЛАГИНОВ**
>
> На сайте webpack доступны списки загрузчиков (https://webpack.github.io/docs/list-of-loaders.html) и плагинов (https://webpack.github.io/docs/list-of-plugins.html). Инструменты webpack также можно найти в npm; хорошей отправной точкой станет поиск по ключевому слову webpack (www.npmjs.com/browse/keyword/webpack).

4.5. Заключение

- Если вам понадобится автоматизировать простые задачи или запускать сценарии, сценарии `npm` идеально подходят для этой цели.
- Gulp может использоваться для написания более сложных задач на JavaScript и является кросс-платформенной технологией.
- Если gulp-файлы получаются слишком длинными, код можно разделить на несколько файлов.
- webpack может использоваться для генерирования пакетов на стороне клиента.
- Если вам нужно построить пакет на стороне клиента, решение с webpack может быть менее трудоемким, чем настройка соответствующего сценария с Gulp.
- webpack поддерживает горячую перезагрузку модулей; это означает, что изменения в коде будут видны без обновления браузера.

5 Фреймворки на стороне сервера

Эта глава полностью посвящена веб-разработке на стороне сервера. Вы найдете в ней ответы на разные вопросы: как выбрать идеальный фреймворк для заданного проекта? Какими достоинствами и недостатками обладает каждый фреймворк?

Задача выбора правильного фреймворка сложна, потому что их сложно сравнивать в системе единых критериев. У многих разработчиков нет времени изучать все фреймворки, поэтому мы склонны принимать субъективные решения относительно тех фреймворков, с которыми у нас имеется опыт работы. Иногда разные фреймворки используются совместно. Например, Express может использоваться с более крупными приложениями, а микросервисы, обеспечивающие работу больших приложений, могут быть написаны на hapi.

Представьте, что вы строите систему управления контентом (CMS). Система предназначена для управления юридическими документами, собираемыми исследовательской фирмой. Такая система может быть построена с применением разных фреймворков:

- загрузка, отправка и чтение документов — Express;
- микросервис генератора PDF — hapi;
- компонент электронной коммерции — Sails.js.

Идеальный фреймворк для заданного проекта зависит от потребностей проекта и группы, работающей над этим проектом. В этой главе для выбора фреймворка, подходящего для конкретного типа проекта, используются *персонажи*, то есть вымышленные люди. Вы взглянете на Koa, hapi, Sails.js, DerbyJS, Flatiron и LoopBack глазами этих воображаемых программистов. Персонажи определяются в следующем разделе.

5.1. Персонажи

Мы не собираемся расхваливать на все лады один фреймворк, который может использоваться в каждом проекте. Гораздо лучше действовать многогранно

и использовать набор инструментов, подходящих для каждой задачи. Практика применения персонажей для проведения анализа в ходе проектирования получила широкое распространение еще и потому, что она помогает проектировщикам сопереживать своим пользователям.

В этой главе персонажи помогут вам взглянуть на фреймворки со стороны и понять, как разные классы проектов подходят для разных решений. Персонажи определяются в контексте профессиональной ситуации и средств разработки. Скорее всего, вы найдете что-то общее хотя бы с одним из трех персонажей, придуманных нами для этой главы.

5.1.1. Фил: штатный разработчик

Фил три года проработал полностековым веб-разработчиком. У него имеется некоторый опыт работы на Ruby, Python и клиентском JavaScript:

- *Положение* — штатный работник, полностековый разработчик.
- *Тип работы* — работа с клиентской частью, разработка на стороне сервера.
- *Компьютер* — MacBook Pro.
- *Инструменты* — Sublime Text, Dash, xScope, Pixelmator, Sketch, GitHub.
- *Подготовка* — среднее общее образование; начинал карьеру как программист-любитель.

Типичный рабочий день Фила включает работу с дизайнерами и UX-экспертами на встречах в agile-стиле, где обсуждаются или разрабатываются новые возможности, а также рассматриваются вопросы сопровождения или исправления ошибок.

5.1.2. Надин: разработчик открытого кода

Надин перешла на контрактную систему после успешного начала карьеры на должности корпоративного веб-разработчика:

- *Положение* — внештатный работник, специалист по JavaScript.
- *Тип работы* — программирование на стороне сервера, иногда программирование на Go и Erlang. Также пишет популярное веб-приложение с открытым кодом — базу данных фильмов.
- *Компьютер* — высокопроизводительный PC, Linux.
- *Инструменты* — Vim, tmux, Mercurial, средства командного интерпретатора.
- *Подготовка* — диплом в области компьютерных технологий.

В ходе своего рабочего дня Надин обычно распределяет время между двумя своими крупными клиентами и работой над своим проектом с открытым кодом. В своей

контрактной работе она применяет разработку через тестирование, тогда как ее проекты с открытым кодом ориентированы на разработку функциональности.

5.1.3. Элис: разработчик продукта

Элис работает над успешным приложением для iOS, но также участвует в разработке веб-API своей компании:

- *Положение* — штатный работник, программист.
- *Тип работы* — разработка для iOS; также отвечает за веб-приложения и веб-службы.
- *Компьютер* — MacBook Pro, iPad Pro.
- *Инструменты* — Xcode, Atom, Babel, Perforce.
- *Подготовка* — ученая степень; один из первых пяти сотрудников стартапа, в котором она работает.

Элис вынужденно работает с Xcode, Objective-C и Swift, но втайне предпочитает JavaScript и с большим интересом относится к ES2015 и Babel. С удовольствием занимается разработкой новых веб-служб для поддержки приложений iOS и настольных приложений своей компании и хочет чаще работать над веб-приложениями на базе React.

Определившись с персонажами, перейдем к определению термина «фреймворк».

5.2. Что такое фреймворк?

Некоторые фреймворки на стороне сервера, рассматриваемые в этой главе, с технической точки зрения фреймворками вообще не являются. К сожалению, термин «фреймворк» перегружен изобилием смыслов и несет разные понятия для разных программистов. В сообществе Node большинство этих проектов было бы правильнее называть модулями, но при прямом сравнении этого семейства библиотек желательно иметь более точное определение.

В проекте LoopBack (*http://loopback.io/resources/#compare*) используются следующие определения:

- *API-фреймворк* — библиотека для построений веб-API с поддержкой фреймворка, упрощающего структурирование приложения. Сам проект LoopBack определяется как фреймворк этого типа.
- *Библиотека HTTP-сервера* — к этой категории относятся все разработки на базе Express, включая Koa и Kraken.js. Эти библиотеки помогают строить приложения, основанные на командах и маршрутах HTTP.

- *Фреймворк HTTP-сервера* — фреймворк для построения модульных серверов, использующих протокол HTTP для коммуникаций. Пример фреймворков такого рода — hapi.
- *Веб-фреймворк MVC* — к этой категории относятся фреймворки на базе паттерна «Модель-Представление-Контроллер», включая Sails.js.
- *Полностековый фреймворк* — эти фреймворки используют JavaScript на сервере и в браузере и могут совместно использовать клиентский и серверный код (такой код называется *изоморфным*). В частности, DerbyJS является полностековым фреймворком MVC.

Многие разработчики Node понимают под термином «фреймворк» второй пункт: библиотеку HTTP-сервера. В следующем разделе вы познакомитесь с Koa — библиотекой сервера, которая использует генераторы (новый синтаксис ES2015) как уникальный подход к построению промежуточного ПО (функций промежуточной обработки) HTTP.

5.3. Koa

Koa (*http://koajs.com/*) базируется на Express, но использует синтаксис генераторов ES2015 для определения промежуточного ПО (функций промежуточной обработки). Это означает, что промежуточное ПО можно писать практически в синхронном стиле. Отчасти это решает проблему промежуточного ПО, сильно зависящего от обратных вызовов. С Koa вы можете воспользоваться ключевым словом `yield` для выхода и последующего возврата к промежуточному ПО.

В табл. 5.1 приведена сводка основных возможностей Koa.

Таблица 5.1. Основные возможности Koa

Тип библиотеки	Библиотека HTTP-сервера
Функциональность	Промежуточное ПО с использованием генераторов, модель «запрос/ответ»
Рекомендуемое применение	Облегченные веб-приложения, нежесткие HTTP API, одностраничные веб-приложения
Архитектура плагинов	Промежуточное ПО
Документация	http://koajs.com/
Популярность	10 000 звезд на GitHub
Лицензия	MIT

Листинг 5.1 показывает, как использовать Koa для хронометража запросов. Для этого управление уступается следующему компоненту промежуточного ПО, после завершения которого управление продолжается на стороне вызова.

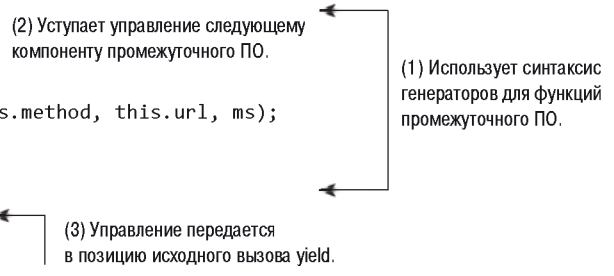
Листинг 5.1. Хронометраж запросов в Koa

```
const koa = require('koa');
const app = koa();

app.use(function*(next) {
  const start = new Date;
  yield next;
  const ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function*() {
  this.body = 'Hello World';
});

app.listen(3000);
```



В листинге 5.1 генераторы **(1)** используются для переключения контекста между двумя компонентами промежуточного ПО. Обратите внимание на использование ключевого слова `function*` — в данном случае стрелочная функция использоваться не может. При использовании ключевого слова `yield` **(2)** выполнение передается вниз по стеку промежуточного ПО, а затем возвращается обратно при возврате из следующего компонента **(3)**. Дополнительное преимущество использования функции-генератора заключается в том, что вы можете просто задать `this.body`, тогда как Express использует функцию для отправки ответов: `res.send(response)`. В терминологии Koa `this` называется *контекстом*. Контекст создается для каждого запроса и используется и для инкапсуляции Node-объектов запроса, и для ответа HTTP (*https://nodejs.org/api/http.html*). Когда вам нужно обратиться к каким-то данным из запроса (например, параметрам GET или cookie), вы используете контекст. Это относится и к ответу: как было показано в листинге 5.1, вы можете управлять тем, какие данные отправляются браузеру, задавая значения в `this.body`.

Если прежде вы использовали промежуточное ПО Express и синтаксис генераторов, изучить Koa будет несложно. Если хотя бы что-то из этого окажется для вас новым, вам будет достаточно трудно понять логику Koa — или по крайней мере увидеть, чем хорош этот стиль. На рис. 5.1 более подробно показано, как `yield` передает выполнение между компонентами промежуточного ПО.

Каждая фаза на рис. 5.1 соответствует номерам в листинге 5.1. Сначала в первом компоненте промежуточного ПО устанавливается таймер **(1)**, а затем управление уступается второму компоненту промежуточного ПО, который строит тело **(2)**. После того как ответ будет отправлен, управление возвращается первому компоненту промежуточного ПО, где вычисляется время выполнения **(3)**. Оно выводится на терминал вызовом `console.log`, и обработка запроса на этом завершается. Обратите внимание: стадия **(4)** в листинге 5.1 не видна; она обрабатывается Koa и HTTP-сервером Node.

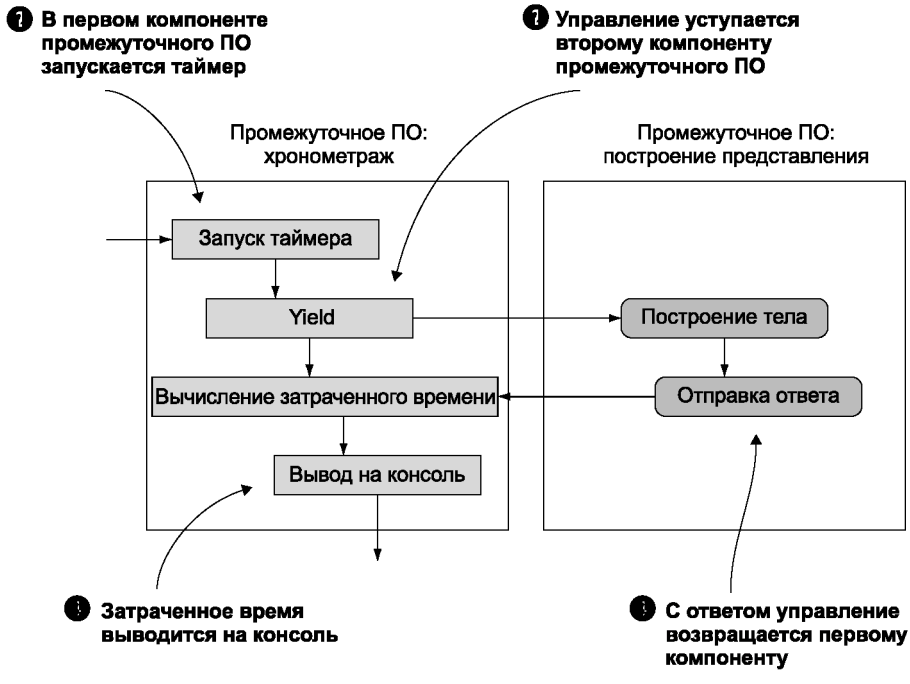


Рис. 5.1. Порядок выполнения компонентов промежуточного ПО в Koa

5.3.1. Настройка

Настройка проекта с Koa требует установки модуля и определения промежуточного ПО. Если вам нужна более широкая функциональность (например, API маршрутизации, упрощающий определение различных типов запросов HTTP и реакцию на них), необходимо установить промежуточное ПО маршрутизации. Таким образом, при типичном рабочем процессе используемое вашим проектом промежуточное ПО должно планироваться заранее, поэтому вы должны заранее собрать информацию о популярных модулях.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Элис: «Мне как разработчику продукта нравится минимализм функциональности Koa, потому что у нашего проекта существуют уникальные требования и мы хотим сформировать весь стек в соответствии с нашими потребностями».

Фил: «Мне как штатному программисту кажется, что фаза изучения промежуточного ПО создает слишком много проблем. Я предпочел бы, чтобы это было сделано за меня, потому что во многих моих проектах действуют похожие требования и я не хочу многократно устанавливать одни и те же модули для выполнения основных операций».

В следующем разделе продемонстрирован сторонний модуль, реализующий мощную библиотеку маршрутизации для Koa.

5.3.2. Определение маршрутов

Одним из популярных компонентов промежуточного ПО для маршрутизации является koa-router (*https://www.npmjs.com/package/koa-router*). Как и Express, он базируется на командах HTTP, но в отличие от Express поддерживает API с возможностью сцепления. Следующий фрагмент показывает, как определяются группы маршрутов:

```
router
  .post('/pages', function*(next) {
    // Создание страницы
  })
  .get('/pages/:id', function*(next) {
    // Визуализация страницы
  })
  .put('pages-update', '/pages/:id', function*(next) {
    // Обновление страницы
  });
```

Имена маршрутов задаются дополнительным аргументом. Это очень удобно, потому что вы можете генерировать URL-адреса, которые поддерживаются не всеми веб-фреймворками Node. Пример:

```
router.url('pages-update', '99');
```

Этот модуль предоставляет уникальное сочетание возможностей Express и других веб-фреймворков.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Эта библиотека маршрутизации напоминает мне многое из того, что мне понравилось в Ruby on Rails. Пожалуй, к Koa все же стоит присмотреться!»

Надин: «Я вижу возможности для разбиения моих существующих проектов на модули и последующего распространения кода в сообществе».

5.3.3. REST API

Koa не включает в себя инструменты, необходимые для создания REST-совместимых API без реализации некоторого промежуточного ПО обработки маршрутов. Приведенный пример может быть расширен для реализации REST-совместимого API в Koa.

5.3.4. Сильные стороны

Было бы легко сказать, что сильные стороны Koa обусловлены ранним принятием синтаксиса генераторов, но теперь, когда стандарт ES2015 получил широкое распространение в сообществе Node, эта возможность уже перестала быть такой уникальной. В настоящее время главное преимущество Koa — плавность и удобство работы при наличии превосходных сторонних модулей; дополнительную информацию можно найти в вики Koa (*https://github.com/koajs/koa/wiki#middleware*). Разработчики продуктов любят Koa за элегантный синтаксис и возможность адаптации к проектам со специфическими требованиями.

5.3.5. Слабые стороны

Широта возможностей настройки Koa отталкивает некоторых разработчиков. Создание множества мелких проектов с Koa может привести к низкой степени повторного использования кода между проектами, если только у вас нет готовых стратегий.

5.4. Kraken

Kraken базируется на Express, но добавляет новую функциональность через специальные модули, разработанные PayPal. Один из особенно полезных модулей — Lusca (*https://github.com/krakenjs/lusca*) — предоставляет прослойку безопасности приложения. И хотя Lusca может использоваться и без Kraken, одним из преимуществ Kraken является предопределенная структура приложения.

Приложения Express и Koa не требуют никакой конкретной структуры проекта, и если вы пожелаете упростить работу над новым проектом, Kraken поможет вам в этом. В табл. 5.2 приведена сводка основной функциональности Kraken.

Таблица 5.2. Основные возможности Kraken

Тип библиотеки	Библиотека HTTP-сервера
Функциональность	Жесткая структура проекта, модели, шаблоны (Dust), укрепление безопасности (Lusca), настройка конфигурации, интернационализация
Рекомендуемое применение	Корпоративные веб-приложения
Архитектура плагинов	Промежуточное ПО Express
Документация	https://www.kraken.com/help/api
Популярность	4000 звезд на GitHub
Лицензия	Apache 2.0

5.4.1. Настройка

Если у вас уже имеется проект Express, Kraken можно добавить как компонент промежуточного ПО:

```
const express = require('express'),
const kraken = require('kraken-js');

const app = express();
app.use(kraken());
app.listen(3000);
```

Но если вы хотите создать новый проект, попробуйте воспользоваться Yeoman — генератором Kraken, который упрощает генерирование новых проектов. При помощи генераторов Yeoman можно создавать подготовленные проекты для разных фреймворков. Ниже приведена последовательность действий для создания специализированного проекта Kraken с использованием Yeoman, с предпочтительной структурой файловой системы Kraken:

```
$ npm install -g yo generator-kraken bower grunt-cli
$ yo kraken

      ,'""`.
hh   /  _  _ \
     |(@)(@)|   Release the Kraken!
     )  __  (
    /,'))((`.\
   (( ((  )) ))
    `\ `)(' /'

Tell me a bit about your application:

[?] Name: kraken-test
[?] Description: A Kraken application
[?] Author: Alex R. Young
...
```

Генератор создает новый каталог, так что вам не придется делать это самостоятельно. После того как генератор завершит работу, вы сможете запустить сервер, открыть адрес *http://localhost:8000* и убедиться в этом.

5.4.2. Определение маршрутов

В Kraken маршруты определяются наряду с *контроллером*. Вместо того чтобы разделять определения маршрутов и обработчики маршрутов, как это делается в Express, Kraken использует решение, вдохновленное паттерном MVC; оно получается достаточно компактным благодаря использованию стрелочных функций ES6:

```
module.exports = (router) => {
  router.get('/', (req, res) => {
    res.render('index');
  });
};
```

Маршруты могут включать в себя параметры в URL:

```
module.exports = (router) => {
  router.get('/people/:id', (req, res) => {
    const people = { alex: { name: 'Alex' } };
    res.render('people/edit', people[req.param.id]);
  });
};
```

API маршрутизации Kraken — express-enrouten (*https://github.com/krakenjs/express-enrouten*) — частично вычисляет маршрут на основании каталога, в котором находится файл. Предположим, имеется файловая структура следующего вида:

```
controllers
 |-user
     |-create.js
     |-list.js
```

Kraken генерирует маршруты вида `/user/create` и `/user/list`.

5.4.3. REST API

Kraken может использоваться для формирования REST API, но не предоставляет конкретной поддержки для них. Возможности express-enrouten в сочетании с разбором JSON означают, что вы можете использовать Kraken для реализации REST API.

В средствах маршрутизации Kraken имеется поддержка команд HTTP для DELETE, GET, POST, PUT и т. д., благодаря чему реализация REST имеет много общего с Express.

5.4.4. Сильные стороны

Так как в поставку Kraken включается генератор, на высоком уровне проекты Kraken похожи друг на друга. Если проекты Express могут иметь сильно различающуюся структуру, в проектах Kraken обычно используется единая схема размещения файлов и каталогов.

Так как Kraken предоставляет как библиотеку шаблонизации (Dust), так и средства интернационализации (Makara), эти две области идеально интегрированы. Чтобы написать шаблоны Dust с поддержкой интернационализации, необходимо задать соответствующий ключ:

```
<h1>{@pre type="content" key="greeting"/}</h1>
```

Затем добавьте файл `.properties` в locales/language-code/view-name.properties. Файлы свойств представляют собой простые пары «ключ-значение»; если предыдущий пример находится в файле представления с именем public/templates/profile.dust, то файлом .profile будет файл locales/US/en/profile.properties.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Меня радует то, что Kraken использует определенную структуру файловой системы и контроллеры для маршрутов. Некоторые участники моей группы уже знают Django и Ruby on Rails, поэтому переход для них будет несложным. Похоже, у Kraken хорошая документация; и в блоге много полезной информации».

Элис: «Мне нравится идея повышения безопасности за счет использования Lusca, но Kraken предоставляет много функциональности, которая мне на самом деле не нужна. Поэтому пока я попробую просто ограничиться использованием Lusca».

5.4.5. Слабые стороны

Kraken сложнее в изучении, чем Koa или Express. Некоторые задачи, реализуемые в Express на программном уровне, выполняются из конфигурационных файлов JSON, и иногда бывает трудно точно разобраться в том, какие свойства JSON понадобятся для того, чтобы добиться желаемого эффекта.

5.5. hapi

hapi (*http://hapijs.com/*) — серверный фреймворк для создания API веб-приложений. Он имеет собственный API плагинов и не включает никакой поддержки на стороне

Таблица 5.3. Основные возможности hapi

Тип библиотеки	Фреймворк HTTP-сервера
Функциональность	Абстракция высокоуровневого серверного контейнера, заголовки безопасности
Рекомендуемое применение	Одностраничные веб-приложения, HTTP API
Архитектура плагинов	Плагины hapi
Документация	http://hapijs.com/api
Популярность	6000 звезд на GitHub
Лицензия	BSD 3 Clause

клиента или уровня модели базы данных; поддерживает API маршрутизации и имеет собственную обертку для сервера HTTP. В hapi вы проектируете API, ориентируясь на сервер как на основную абстракцию. Благодаря встроенной серверной функциональности подключений и ведения журнала hapi хорошо проявляет себя в области масштабирования и управления с точки зрения DevOps. В табл. 5.3 приведена сводка основной функциональности hapi.

5.5.1. Настройка

Сначала создайте новый проект Node и установите hapi:

```
mkdir listing5_2
cd listing5_2
npm init -y
npm install --save hapi
```

Затем создайте новый файл с именем server.js. Добавьте код из листинга 5.2.

Листинг 5.2. Базовый сервер hapi

```
const Hapi = require('hapi');
const server = new Hapi.Server();

server.connection({
  host: 'localhost',
  port: 8000
});

server.start((err) => {
  if (err) {
    throw err;
  }
  console.log('Server running at:', server.info.uri);
});
```

Пример можно запустить в том виде, в котором он приведен в листинге, но без маршрутов он особой пользы не принесет. В следующем разделе вы узнаете, как в hapi организована обработка маршрутов.

5.5.2. Определение маршрутов

В hapi имеется встроенный API без создания маршрутов. Вы должны предоставить объект, включающий свойства для метода запроса, URL и функцию обратного вызова, называемую *обработчиком* (handler). В листинге 5.3 приведен пример определения маршрута с методом-обработчиком.

Листинг 5.3. Сервер Hello World на hapi

```
const Hapi = require('hapi');
const server = new Hapi.Server();
```

```
server.connection({
  host: 'localhost',
  port: 8000
});

server.route({
  method: 'GET',
  path:'/hello',
  handler: (request, reply) => {
    return reply('hello world');
  }
});

server.start((err) => {
  if (err) {
    throw err;
  }
  console.log('Server running at:', server.info.uri);
});
```

Добавьте этот код в предыдущий листинг для определения маршрута и обработчика, который отвечает текстом «Hello World». Пример можно запустить командой `npm start`. Введите адрес *http.//localhost:8000/hello*, чтобы увидеть ответ.

hapi не включает предопределенной структуры папок или какой-либо функциональности MVC; вся структура полностью базируется на серверах. В этом отношении hapi можно сравнить с Express. Однако обратите внимание на ключевое отличие: сигнатура обработчика маршрута `request, reply` отличается от сигнатуры Express `req, res`. Объекты запроса и ответа hapi также отличаются от своих эквивалентов Express: вы должны вызвать `reply` вместо того, чтобы оперировать с объектом Express `res`. У Express больше общего со встроенным HTTP-сервером Node.

Чтобы выйти за пределы этого простого примера и получить доступ к расширенной функциональности (например, предоставлению статических файлов), вам понадобятся плагины.

5.5.3. Плагины

hapi использует собственную архитектуру плагинов, и большинству проектов для предоставления такой функциональности, как аутентификация и проверка ввода, требуется несколько плагинов. Простой плагин, необходимый большинству проектов, — inert (*https://github.com/hapijs/inert*) — добавляет поддержку статических файлов и обработчики каталогов.

Чтобы включить inert в проект hapi, необходимо сначала зарегистрировать плагин методом `server.register`. При этом добавляются метод `reply.file` для отправки одиночных файлов и встроенный обработчик каталогов. Рассмотрим обработчик каталогов.

Убедитесь в том, что вы создали проект из листинга 5.2. Затем установите inert:

```
npm install --save inert
```

Теперь плагин можно загрузить и зарегистрировать. Откройте файл server.js и добавьте следующие строки:

Листинг 5.4. Добавление плагина в hapi

```
const Inert = require('inert');

server.register(Inert, () => {});

server.route({
  method: 'GET',
  path: '/{param*}',
  handler: {
    directory: {
      path: '.',
      redirectToSlash: true,
      index: true
    }
  }
});
```

Маршруты hapi могут получать не только функции, но и объекты конфигурации для плагинов. В этом листинге объект `directory` включает настройки inert для предоставления файлов из текущего пути и вывода индекса файлов в этом каталоге. В этом отношении hapi отличается от промежуточного ПО Express; пример показывает, как плагины могут расширять поведение сервера в приложениях hapi.

5.5.4. REST API

hapi поддерживает команды HTTP и параметризацию URL, что позволяет реализовать REST API с использованием стандартного API маршрутов hapi. В следующем фрагменте представлен маршрут для обобщенного метода удаления:

```
server.route({
  method: 'DELETE',
  path: '/items/{id}',
  handler: (req, reply) => {
    // Удалить "item" на основании req.params.id
    reply(true);
  }
});
```

Кроме того, плагины упрощают создание REST-совместимых API, например, hapi-sequelize-crud (*https://www.npmjs.com/package/hapi-sequelize-crud*) автоматически генерирует REST-совместимый API на основании моделей Sequelize (*http://docs.sequelizejs.com/en/latest/*).

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Мне определенно хочется опробовать hapi-sequelize-crud, потому что у нас уже есть приложения, использующие PostgreSQL и MySQL, поэтому Sequelize может быть подходящим вариантом. Но hapi не обладает такой функциональностью во встроенном виде; кто знает — вдруг плагин перестанет поддерживаться? Я не уверен, насколько хорошо hapi подходит для штатной работы».

Элис: «Как разработчик продукта я считаю, что hapi представляет интерес. Как и Express, hapi минималистичен, однако API плагинов более формален и выразителен».

Надин: «Я вижу ряд возможностей для создания плагинов с открытым кодом для hapi, а существующие плагины написаны хорошо. Похоже, у hapi технически подкованная аудитория, и мне это нравится».

5.5.5. Сильные стороны

API плагинов — одно из самых больших преимуществ использования hapi. Плагины могут расширять сервер hapi, но при этом они также добавляют другие виды поведения, от проверки данных до шаблонизации. Кроме того, поскольку hapi базируется на серверах HTTP, эта технология хорошо подходит для некоторых типов сценариев развертывания. Если вы запускаете несколько серверов, которые должны быть связаны друг с другом или для которых нужно организовать распределение нагрузки, серверный API hapi может оказаться предпочтительным по сравнению с Express или Koa.

5.5.6. Слабые стороны

Недостатки hapi примерно те же, что у Express: минимализм и вытекающее из него отсутствие правил структуры проекта. Вы никогда не можете быть уверены в том, когда может прекратиться разработка того или иного плагина, поэтому зависимость от слишком большого количества плагинов может создать проблемы с сопровождением в будущем.

5.6. Sails.js

Фреймворки, которые были рассмотрены до настоящего момента, представляли собой минимальные серверные библиотеки. Sails (*http://sailsjs.org/*) — фреймворк MVC (Model-View-Controller), принципиально отличающийся от серверной библиотеки. Sails включает в себя библиотеку объектно-реляционного отображения

(ORM, Object-Relational Mapping) для работы с базами данных и может автоматически генерировать REST API. Также поддерживаются многие современные возможности, включая встроенную поддержку WebSocket. А поклонников React или Angular порадует независимость от клиентской части: Sails не является полностековым фреймворком и поэтому может использоваться практически с любой библиотекой или фреймворком клиентской части. В табл. 5.4 приведена сводка основной функциональности hapi.

Таблица 5.4. Основные возможности Sails

Тип библиотеки	Фреймворк MVC
Функциональность	Поддержка баз данных с ORM, генерирование REST API, WebSocket
Рекомендуемое применение	Приложения MVC в стиле Rails
Архитектура плагинов	Промежуточное ПО Express
Документация	http://sailsjs.org/documentation/concepts/
Популярность	6000 звезд на GitHub
Лицензия	BSD 3 Clause

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Да ведь это именно то, что мне нужно, — в чем подвох?!»

Элис: «Сначала я подумала, что мне это не подойдет, потому что мы уже потратили время разработки на приложение React. Но если это решение ориентировано на сервер, возможно, оно сработает для нашего продукта».

5.6.1. Настройка

В поставку Sails включен генератор проектов, поэтому лучше выполнить глобальную установку, чтобы упростить создание новых проектов. Установите Sails из npm и введите команду `sails new`, чтобы создать проект:

```
npm install -g sails
sails new example-project
```

Команда создает новый каталог с файлом package.json для базовых зависимостей Sails. В новый проект включается собственно Sails, EJS и Grunt. Вы можете запустить сервер командой `npm start` или ввести команду `sails lift`. Когда сервер заработает, по адресу *http://localhost:1337* будет доступна встроенная страница с вводной информацией.

5.6.2. Определение маршрутов

Чтобы добавить маршруты (в Sails они называются *нестандартными маршрутами*), откройте файл config/routes.js и добавьте свойство в экспортируемые маршруты. Свойство состоит из команды HTTP и частичного URL-адреса. Например, некоторые действительные маршруты Sails могут выглядеть так:

```
module.exports.routes = {
  'get /example': { view: 'example' },
  'post /items': 'ItemController.create
};
```

Первый маршрут ожидает получить файл с именем views/example.ejs. Второй маршрут ожидает получить файл с именем api/controllers/ItemController и методом `create`. Чтобы сгенерировать этот контроллер с методом `create`, выполните команду `sails generate controller item create`. Аналогичные команды могут использоваться для быстрого создания REST-совместимых API.

5.6.3. REST API

Sails объединяет модели баз данных и контроллеры в API; чтобы быстро создать заглушки для REST-совместимых API, введите команду `sails generate api` *имя-ресурса*. Чтобы использовать базу данных, сначала необходимо установить адаптер базы данных. Для добавления MySQL необходимо найти имя пакета Waterline MySQL (*https://github.com/balderdashy/waterline*), а затем добавить его в проект:

```
npm install --save waterline sails-mysql
```

Затем откройте файл config/connections.js и введите подробную информацию о подключении для вашего сервера MySQL. Файлы моделей Sails позволяют вам определить подключение к базе данных, так что вы можете использовать разные модели с разными базами данных. Например, это позволяет реализовать такие ситуации, как хранение базы данных пользовательских сеансов в Redis с хранением других, более долгосрочных ресурсов в реляционной базе данных (например, MySQL).

У Waterline — библиотеки баз данных для Sails — имеется собственный репозиторий с документацией (*https://github.com/balderdashy/waterline-docs*). Помимо поддержки разных баз данных Waterline обладает и другими полезными функциями: вы можете определять имена столбцов и таблиц для поддержки унаследованных схем, а API запросов поддерживает *обещания* (promises), чтобы запросы были в большей степени похожи на современный код JavaScript.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Простота создания API и поддержка моделями Waterline существующих схем баз данных? Похоже, Sails идеально подходит для нас. У нас есть клиенты, которые хотят постепенно перейти с MySQL на PostgreSQL; возможно, Waterline позволит нам это сделать. Некоторые разработчики и проектировщики уже работали с Ruby on Rails, и я думаю, что они моментально освоят современный синтаксис ES2015 в Node».

Элис: «Фреймворк предоставляет возможности, которые в нашем проекте не нужны. Я считаю, что Koa или hapi лучше подойдет нам».

5.6.4. Сильные стороны

Наличие встроенных средств создания проектов и генерирования API означает, что операции настройки проектов и добавления типичных REST API выполняются быстро. Также Sails хорошо подходит для создания новых проектов и организации совместной работы, потому что проекты Sails имеют одинаковую структуру файловой системы. Создатели Sails, Майк Макнил (Mike McNeil) и Ирл Натан (Irl Nathan), написали книгу *Sails.js in Action* (Manning Publications, 2017), в которой показано, что Sails помогает и начинающим программистам Node.

5.6.5. Слабые стороны

Sails также обладает рядом недостатков, общих для всех фреймворков MVC на стороне сервера: API маршрутизации означает, что приложение должно проектироваться с учетом средств маршрутизации Sails, и вам может быть трудно адаптировать свою схему для того подхода, который используется в Waterline.

5.7. DerbyJS

DerbyJS — полностековый фреймворк, который поддерживает синхронизацию данных и серверную визуализацию представлений. В этом он зависит от MongoDB и Redis. Уровень синхронизации данных предоставляется ShareJS и поддерживает автоматическое разрешение конфликтов. В табл. 5.5 приведена сводка основной функциональности DerbyJS.

Таблица 5.5. Основные возможности DerbyJS

Тип библиотеки	Полностековый фреймворк
Функциональность	Поддержка баз данных с ORM (Racer), изоморфизм
Рекомендуемое применение	Одностраничные веб-приложения с поддержкой на стороне сервера
Архитектура плагинов	Плагины DerbyJS
Документация	http://derbyjs.com/docs/derby-0.6
Популярность	4000 звезд на GitHub
Лицензия	MIT

5.7.1. Настройка

Если у вас установлен MongoDB или Redis, вам придется установить оба пакета для запуска примеров DerbyJS. Документация DerbyJS объясянет, как это делается в Mac OS, Linux и Windows (*http.//derbyjs.com/started#environment*).

Чтобы быстро создать новый проект DerbyJS, установите derby и derby-starter. Пакет derby-starter используется для базовой инициализации приложения Derby:

```
mkdir example-derby-app
cd example-derby-app
npm init -f
npm install --save derby derby-starter derby-debug
```

Приложения Derby разбиваются на несколько малых приложений; создайте новый каталог приложений с тремя файлами: index.js, server.js и index.html. В листинге 5.5 приведено простое приложение Derby, которое выполняет визуализацию шаблона.

Листинг 5.5. Файл Derby app/index.js

```
const app = module.exports = require('derby')
  .createApp('hello', __filename);
app.loadViews(__dirname);

app.get('/', (page, model) => {
  const message = model.at('hello.message');
  message.subscribe(err => {
    if (err) return next(err);
    message.createNull('');
    page.render();
  });
});
```

Серверный файл должен загрузить только модуль derby-starter, как показано в следующем фрагменте. Сохраните его в файле app/server.js:

```
require('derby-starter').run(__dirname, { port: 8005 });
```

Файл app/index.html строит поле `input` и сообщение, введенное пользователем:

```
<Body:>
  Holler: <input value="{{hello.message}}">
  <h2>{{hello.message}}</h2>
```

Приложение должно запускаться из каталога example-derby-app командой `node derby/server.js`. После того как оно заработает, редактирование файла app/index.html приведет к запуску приложения; редактирование кода и шаблонов сопровождается автоматическими обновлениями в реальном времени.

5.7.2. Определение маршрутов

DerbyJS использует derby-router для маршрутизации. Поскольку в основе DerbyJS лежит Express, API маршрутизации для маршрутов на стороне сервера аналогичен, и в браузере используется тот же модуль маршрутизации. При щелчке на ссылке в приложении DerbyJS делается попытка построить ответ в клиенте.

DerbyJS — полностековый фреймворк, поэтому процедура добавления маршрутов отличается от других библиотек, рассмотренных в этой главе. Самый идиоматический способ добавления базового маршрута основан на добавлении представления. Откройте файл apps/app/index.js и добавьте маршрут следующим вызовом:

```
app.get('hello', '/hello');
```

Откройте файл apps/app/views/hello.pug и добавьте простой шаблон Pug:

```
index:
  h2 Hello
  p Hello world
```

Теперь откройте файл apps/app/views/index.pug и импортируйте шаблон:

```
import:(src="./hello")
```

Если вы выполнили команду `npm start`, проект должен постоянно обновляться, поэтому при открытии адреса *http://localhost:3000/hello* теперь должно отображаться новое представление.

Строка `index:` содержит *пространство имен* для представления. В DerbyJS имена представлений снабжаются пространствами имен, отделенных двоеточиями, поэтому вы фактически создаете имя `hello:index`. Инкапсуляция представлений в пространствах имен нужна для предотвращения конфликтов в больших проектах.

5.7.3. REST API

В проектах DerbyJS REST-совместимые API создаются добавлением маршрутов и обработчиков маршрутов в Express. Ваш проект DerbyJS содержит файл **server.js**, который использует Express для создания сервера. Открыв файл **server/routes.js**, вы найдете в нем пример маршрута, определенного с использованием стандартного API маршрутизации Express.

В серверном файле маршрутов вы можете использовать `app.use` для монтирования другого приложения Express, поэтому REST API можно смоделировать в виде совершенно отдельного приложения Express, которое монтируется основным приложением DerbyJS.

5.7.4. Сильные стороны

В DerbyJS имеется API модели базы данных и API синхронизации данных. DerbyJS может использоваться как для построения одностраничных веб-приложений, так и для современных приложений реального времени. Благодаря встроенной поддержке WebSocket и синхронизации вам не придется беспокоиться о том, какую библиотеку WebSocket стоит использовать или как организовать синхронизацию данных между клиентом и сервером.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Один из наших клиентов интересуется построением проекта визуализации данных на основании информации, получаемой в реальном времени, и я думаю, что DerbyJS хорошо подойдет для этой цели. Но, похоже, освоение этой технологии потребует серьезных усилий. Не уверен, что смогу уговорить наших разработчиков».

Элис: «Как разработчик продукта я не представляю, как приспособить потребности нашего продукта к архитектуре DerbyJS, поэтому вряд ли этот вариант подойдет для моего проекта».

5.7.5. Слабые стороны

Людей, уже имеющих опыт работы с библиотеками на стороне сервера или клиента, довольно трудно убедить использовать DerbyJS. Например, разработчики на стороне клиента, любящие React, обычно не хотят использовать DerbyJS. Разработчики на стороне сервера, которым нравится создавать REST API или проекты MVC и которые уверенно чувствуют себя с WebSocket, также не понимают, зачем им изучать DerbyJS.

5.8. Flatiron.js

Flatiron — веб-фреймворк, включающий в себя функциональность маршрутизации URL, управления данными, промежуточного ПО, плагинов и ведения журнала. В отличие от большинства веб-фреймворков, модули Flatiron проектировались в расчете на максимальную логическую изоляцию, поэтому вам не придется использовать их все. Вы даже можете использовать их в собственных проектах: если, скажем, вам понравится модуль ведения журнала, вы можете подключить его к проекту Express. В отличие от многих фреймворков Node, уровни маршрутизации URL и промежуточного ПО в Flatiron не написаны с использованием Express или Connect, хотя промежуточное ПО обладает обратной совместимостью с Connect. В табл. 5.6 приведена сводка основной функциональности Flatiron.

Таблица 5.6. Основные возможности Flatiron

Тип библиотеки	Модульный фреймворк MVC
Функциональность	Уровень управления базой данных (Resourceful); изолированные модули, пригодные для повторного использования
Рекомендуемое применение	Облегченные приложения MVC, использование модулей Flatiron в других фреймворках
Архитектура плагинов	API плагинов Broadway
Документация	https://github.com/flatiron
Популярность	1500 звезд на GitHub
Лицензия	MIT

5.8.1. Настройка

Установка Flatiron требует глобальной установки средств командной строки для создания новых проектов Flatiron:

```
npm install -g flatiron
flatiron create example-flatiron-app
```

После выполнения этих команд вы найдете новый каталог с файлом package.json и необходимыми зависимостями. Выполните команду `npm install`, чтобы установить зависимости, а затем запустите приложение командой `npm start`.

Основной файл app.js выглядит как типичное приложение Express:

```
const flatiron = require('flatiron');
const path = require('path');
const app = flatiron.app;
```

```
app.config.file({ file: path.join(__dirname, 'config', 'config.json') });

app.use(flatiron.plugins.http);

app.router.get('/', () => {
  this.res.json({ 'hello': 'world' })
});

app.start(3000);
```

Обратите внимание: система маршрутизации отличается как от Express, так и от Koa. Ответы возвращаются через `this.res` вместо аргумента функции обратного вызова. Рассмотрим маршруты Flatiron более подробно.

5.8.2. Определение маршрутов

Библиотека маршрутизации Flatiron называется Director. И хотя она может использоваться для серверных маршрутов, она также поддерживает маршруты в браузерах, поэтому она может применяться и для одностраничных приложений. Director вызывает маршруты с командами HTTP в стиле Express:

```
router.get('/example', example);
router.post('/example', examplePost);
```

Маршруты могут иметь параметры, и параметры могут определяться регулярными выражениями:

```
router.param('id', /([\\w\\-]+)/);
router.on('/pages/:id', pageId => {});
```

Чтобы сгенерировать ответ, используйте `res.writeHead` для отправки заголовков и `res.end` для отправки тела ответа:

```
router.get('/',  () => {
  this.res.writeHead(200, { 'content-type': 'text/plain' });
  this.res.end('Hello, World');
});
```

API маршрутизации также может использоваться в виде класса — с объектом таблицы маршрутизации. Чтобы использовать его, создайте экземпляр маршрутизатора и используйте метод `dispatch` при поступлении запросов HTTP:

```
const http = require('http');
const director = require('director');
const router = new director.http.Router({
  '/example': {
    get: () => {
      this.res.writeHead(200, { 'Content-Type': 'text/plain' })
      this.res.end('hello world');
    }
```

```
  }
});
const server = http.createServer((req, res) =>
  router.dispatch(req, res);
});
```

Использование API маршрутизации в виде класса также означает возможность подключения к потоковому API. Это позволяет организовать быструю и легкую обработку больших запросов, что хорошо подходит для таких задач, как разбор отправленных данных с ранним выходом:

```
const director = require('director');
const router = new director.http.Router();

router.get('/', { stream: true }, () => {
  this.req.on('data', (chunk) => {
    console.log(chunk);
  });
});
```

API маршрутизации Director может быть полезен для создания REST API.

5.8.3. REST API

REST API могут создаваться стандартными методами в стиле HTTP-команд Express или же с использованием средств маршрутизации Director. Это позволяет группировать маршруты на основании фрагментов и параметров URL:

```
router.path(/\/users\/(\w+)/, () => {
  this.get((id) => {});
  this.delete((id) => {});
  this.put((id) => {});
});
```

Flatiron также предоставляет высокоуровневую REST-обертку Resourceful (*https://github.com/flatiron/resourceful*), которая поддерживает CouchDB, MongoDB, Socket.IO и проверку данных.

5.8.4. Сильные стороны

Фреймворкам достаточно трудно завоевать популярность; именно по этой причине логическая изоляция компонентов Flatiron является одной из его сильных сторон. Некоторые из модулей Flatiron могут применяться без использования всего фреймворка. Например, модуль ведения журнала Winston (*https://github.com/winstonjs/winston*) присутствует во многих проектах, которые не используют остальные компоненты Flatiron. Это означает, что некоторые компоненты Flatiron получают достаточно серьезный творческий вклад от сообщества с открытым кодом.

API URL-маршрутизации Director изоморфен; это означает, что он может использоваться в решениях для разработки как на стороне клиента, так и на стороне сервера. API Director также отличается от API маршрутизации в стиле Express: Director использует упрощенный потоковый API, а объект маршрутизации генерирует события до и после выполнения маршрутов.

В отличие от большинства веб-фреймворков Node, Flatiron содержит менеджер плагинов. Плагины, поддерживаемые сообществом, упрощают расширение проектов Flatiron.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Надин: «Мне нравится модульная структура Flatiron, а менеджер плагинов просто замечателен. Я уже представляю себе некоторые плагины, которые я хотела бы создать».

Элис: «Не могу сказать, что все модули Flatiron мне нужны, но мне хотелось бы опробовать Flatiron с другой системой ORM и библиотекой шаблонизации».

5.8.5. Слабые стороны

Фреймворк Flatiron не настолько прост в использовании с большими проектами в стиле MVC, как другие фреймворки. Например, Sails проще настраивается. Если вы создаете несколько традиционных веб-приложений среднего размера, Flatiron может сработать хорошо. Возможность настройки Flatiron может стать дополнительным преимуществом, но проследите за тем, чтобы фреймворк сначала прошел оценку наряду с другими вариантами.

К числу сильных конкурентов принадлежит LoopBack — последний фреймворк, описанный в этой главе.

5.9. LoopBack

Фреймворк LoopBack был создан StrongLoop — компанией, предоставляющей несколько коммерческих сервисов, поддерживающих разработку веб-приложений Node. Формально это фреймворк API, но благодаря некоторым своим возможностям он хорошо подходит для баз данных и приложений MVC. Он даже включает в себя веб-интерфейс для анализа и управления REST API. Если вы ищете решение, которое поможет создавать веб-API для мобильных и настольных клиентов, функциональность LoopBack подойдет идеально. В табл. 5.7 приведена сводка основной функциональности LoopBack.

Таблица 5.7. Основные возможности LoopBack

Тип библиотеки	Фреймворк API
Функциональность	ORM, API пользовательского интерфейса, WebSocket, клиентский SDK (включая iOS)
Рекомендуемое применение	API для поддержки разных клиентов (мобильные, настольные, веб)
Архитектура плагинов	Промежуточное ПО Express
Документация	http://loopback.io/doc/
Популярность	6500 звезд на GitHub
Лицензия	Двойная лицензия: MIT и соглашение о подписке StrongLoop

LoopBack распространяется с открытым кодом, и с момента приобретения StrongLoop компанией IBM фреймворк получил серьезную коммерческую поддержку. В результате он стал уникальным предложением в сообществе Node. В частности, в него включены генераторы Yeoman для быстрой настройки оснастки приложений. В следующем разделе показано, как создать новое приложение LoopBack.

5.9.1. Настройка

Для создания нового проекта LoopBack необходимо воспользоваться программой командной строки StrongLoop (*www.npmjs.com/package/strongloop*). После глобальной установки пакета strongloop инструментарий командной строки запускается командой `slc`. Пакет включает в себя средства управления процессами, но нас интересует прежде всего генератор проектов LoopBack:

```
npm install -g strongloop
slc loopback
```

Программа командной строки StrongLoop помогает вам пройти все действия по созданию нового проекта. Введите имя проекта, а затем выберите заготовку приложения `api-server`. Когда генератор закончит устанавливать зависимости проекта, он выведет полезные рекомендации по работе с новым проектом (рис. 5.2).

Чтобы запустить проект, введите команду `node .`, а чтобы создать модель — команду `slc:loopback:model`. Вы будете регулярно использовать команду `slc` при создании нового проекта LoopBack.

Когда проект заработает, вы сможете обратиться к API Explorer по адресу *http.//0.0.0.0:3000/explorer/*. Щелкните на ссылке **User**, чтобы раскрыть узел. Вы увидите большой список доступных методов API, включая стандартные REST-совместимые маршруты — такие, как `PUT /Users` и `DELETE /Users/{id}`. API Explorer показан на рис. 5.3.

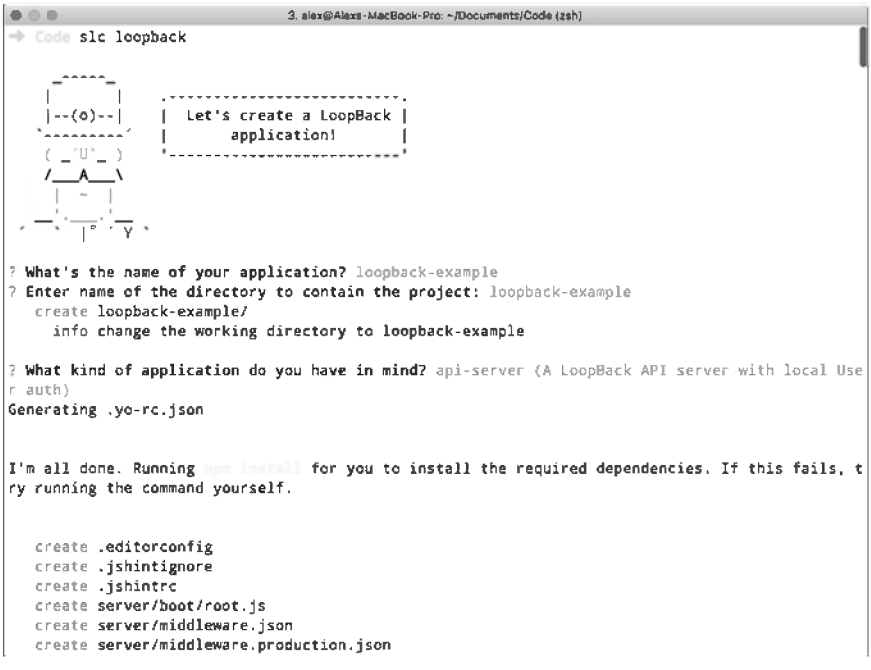


Рис. 5.2. Генератор проектов LoopBack

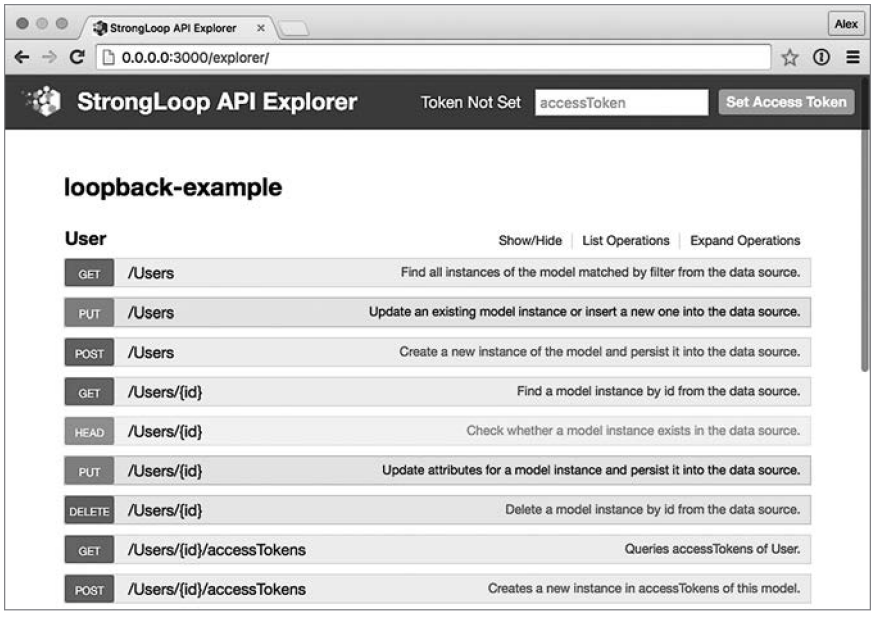


Рис. 5.3. StrongLoop API Explorer с маршрутами User

5.9.2. Определение маршрутов

В LoopBack маршруты можно добавлять на уровне Express. Добавьте новый файл с именем server/boot/routes.js, после чего добавьте маршрут через экземпляр Loopback Router:

```
module.exports = (app) => {
  const router = app.loopback.Router();
  router.get('/hello', (req, res) => {
    res.send('Hello, world');
  });
  app.use(router);
};
```

Теперь при посещении *http://localhost:3000/hello* выдается ответ `Hello, world`. Впрочем, добавление маршрутов таким способом нетипично для проектов LoopBack. Возможно, оно необходимо для некоторых необычных конечных точек API, но, как правило, маршруты добавляются автоматически при генерировании моделей.

5.9.3. REST API

Самый простой способ создания REST API в проекте LoopBack основан на использовании генератора моделей, который является частью функциональности команды `slc`. Например, чтобы добавить новую модель с именем `product`, выполните команду `slc loopback:model`:

```
slc loopback:model product
```

Команда `slc` выполняет всю последовательность действий по созданию модели, позволяя вам выбрать, является ли модель чисто серверной, а также задать некоторые свойства и валидаторы. После того как вы добавите модель, взгляните на соответствующий файл JSON — это должен быть файл common/models/product.json. Этот файл JSON является облегченным способом определения поведения моделей, включающим в себя все свойства, заданные на предыдущем шаге.

Если вы хотите добавить новые свойства, введите команду `slc loopback:property`. Новые свойства могут добавляться в модели в любой момент.

РАЗМЫШЛЕНИЯ ПЕРСОНАЖЕЙ

Фил: «Нашим группам нравится идея LoopBack — в основном из-за способности быстро добавлять REST-совместимые ресурсы и просматривать их в API Explorer. Но мне LoopBack нравится тем, что эта технология достаточно гибка для поддержки наших унаследованных веб-приложений MVC. Можно подключиться к старой базе данных и перевести эти проекты в Node».

Элис: «Это единственный фреймворк, который в действительности ориентирован на iOS и Android, а также полнофункциональных веб-клиентов. У LoopBack имеются клиентские библиотеки для iOS и Android, а это очень важно для нас — разработчиков продуктов, зависящих от мобильных приложений».

5.9.4. Сильные стороны

Даже из этого краткого введения должно быть очевидно, что одна из сильных сторон LoopBack — отсутствие необходимости в написании стереотипного кода. Программа командной строки генерирует практически все необходимое для облегченных REST-совместимых веб-API, даже моделей баз данных и проверки данных. В то же время LoopBack ничего особенно не диктует в отношении кода клиентской части. К тому же LoopBack позволяет вам думать о том, какие модели должны быть доступны для браузера, а какие существуют только на стороне сервера. Некоторые фреймворки действуют неправильно и отправляют браузеру все, что можно.

Если ваши мобильные приложения должны взаимодействовать с веб-API, присмотритесь к клиентскому SDK-пакету LoopBack (*http://loopback.io/doc/en/lb2/Client-SDKs.html*). LoopBack поддерживает интеграцию API и отправку сообщений как для iOS, так и для Android.

5.9.5. Слабые стороны

API модели LoopBack на базе JSON отличается от большинства API баз данных на основе JavaScript. Возможно, вы не сразу поймете, как он соответствует схеме базы данных текущего проекта. И поскольку уровень HTTP базируется на Express, его возможности частично ограничиваются тем, что поддерживает Express. И хотя Express — добротная библиотека HTTP-сервера, существуют более новые библиотеки для Node с более современными API. У LoopBack не существует конкретного API плагинов. Вы можете использовать промежуточное ПО Express, но этот вариант не настолько удобен, как API плагинов Flatiron или hapi.

На этом описание фреймворков в этой главе подходит к концу. Прежде чем переходить к следующей главе, давайте сравним фреймворки, чтобы помочь в выборе наиболее подходящего варианта для вашего следующего проекта.

5.10. Сравнение

Если вы следовали за размышлениями персонажей в этой главе, возможно, вы уже решили, какой фреймворк вам стоит использовать. А если еще не решили — в оставшейся части этой главы сравниваются сильные стороны всех фреймворков. Если же и сравнение не прояснит ситуацию, рис. 5.4 поможет вам выбрать правильный фреймворк, ответив на несколько вопросов.

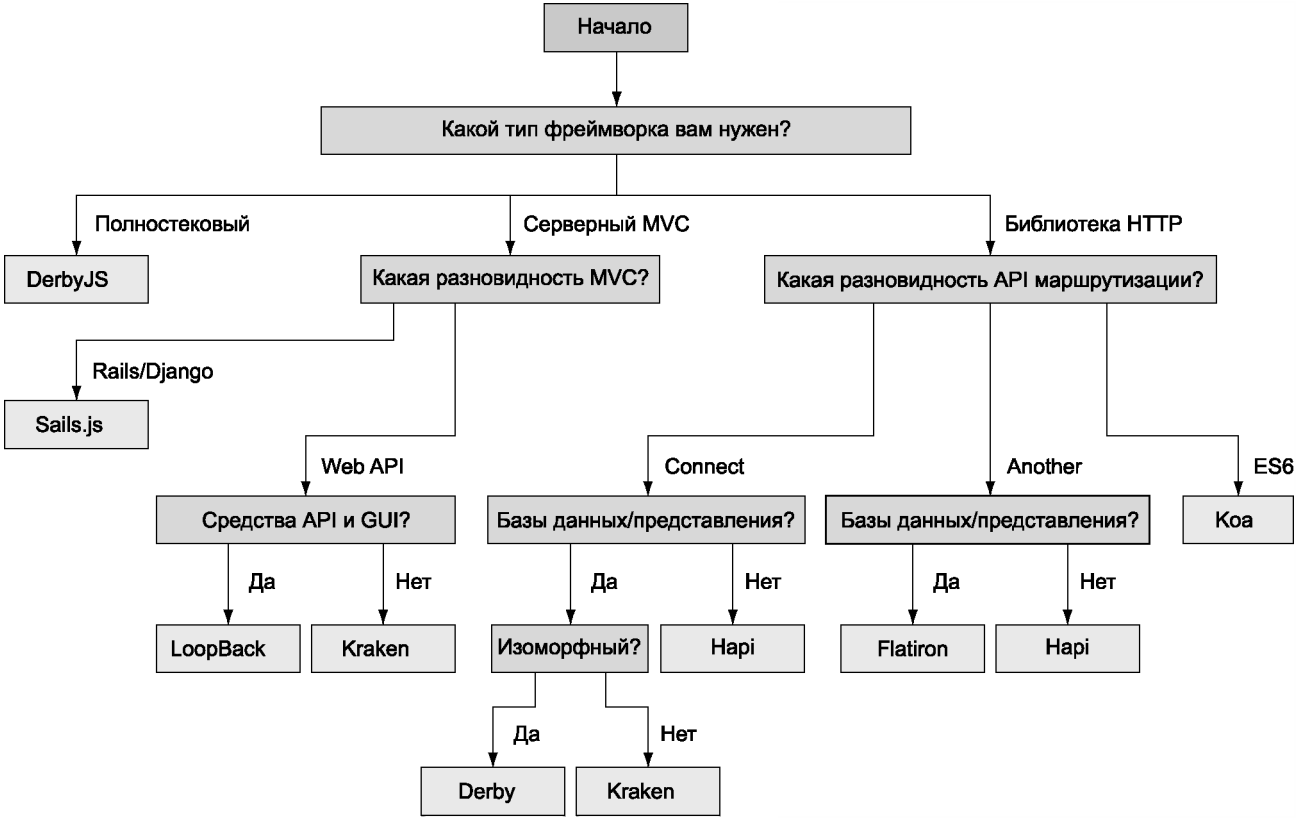


Рис. 5.4. Выбор фреймворка Node

Если взглянуть на описания популярных фреймворков Node на стороне сервера, все они кажутся одинаковыми. Все они предоставляют облегченные HTTP API и используют модель сервера вместо модели страницы, используемой в PHP. Однако различия в их архитектуре имеют значительные последствия для проектов, созданных на их основе, поэтому сравнение таких фреймворков мы начнем на уровне HTTP.

5.10.1. Серверы HTTP и маршруты

Большинство фреймворков Node базируется на Connect или Express. В этой главе были представлены три примера, которые не базируются на Express и представляют собственные решения для HTTP API: Koa, hapi и Flatiron.

Фреймворк Koa был создан тем же автором, что и Express, но предлагает более свежий подход за счет использования более современных средств JavaScript. Если вам нравится Express, но вы хотите использовать синтаксис генераторов ES2015, возможно, Koa вам подойдет.

API сервера hapi и маршрутизации обладают высокой степенью модульности и воспринимаются не так, как Express. Если синтаксис Express кажется вам неуклюжим, опробуйте hapi. hapi упрощает работу с серверами HTTP, так что, если вам нужно выполнять такие операции, как подключение к серверам или кластеризация, возможно, вы предпочтете hapi потомкам Express.

Система маршрутизации Flatiron обладает обратной совместимостью с Express, но также и дополнительной функциональностью. Она генерирует события и использует таблицу маршрутизации. В этом отношении она отличается от стека в стиле Express или компонентов Middleware. Системе маршрутизации Flatiron можно передать объектный литерал. Маршрутизация также работает в браузерах, и если у вас есть серверные разработчики, которые пытаются освоить современную разработку на стороне клиента, они могут более уверенно чувствовать себя с Flatiron, чем пускаться во все тяжкие с React Router.

5.11. Написание модульного кода

Не все фреймворки, рассмотренные здесь, напрямую поддерживают плагины, но все они в той или иной степени поддерживают расширение. Фреймворки на базе Express могут использовать промежуточное ПО Connect, но hapi и Flatiron имеют собственные API плагинов. Хорошо определенный API плагинов полезен, потому что он упрощает расширение фреймворка для его новых пользователей.

Если вы используете большой фреймворк MVC (например, Sails.js или LoopBack), API плагинов упрощает создание нового проекта. LoopBack частично обходит необходимость в API плагинов, предоставляя в распоряжение программиста мощные средства управления проектами. Обратившись к учетной записи StrongLoop в npm

(*www.npmjs.com/~strongloop*), вы увидите множество LoopBack-проектов для добавления поддержки Angular, нескольких баз данных и т. д.

5.12. Выбор персонажей

Персонажи этой главы получили достаточно информации, чтобы принять правильное решение для своего следующего проекта:

Фил: «В итоге я решил остановиться на LoopBack. Это был трудный выбор; и Sails, и Kraken имеют превосходную функциональность, которая понравилась моей команде. Но нам показалось, что LoopBack обладает более сильной долгосрочной поддержкой и избавляет от многих усилий в разработке на стороне сервера».

Надин: «Как разработчик проектов с открытым кодом я выбрала Flatiron. Этот фреймворк хорошо адаптируется к различным проектам, над которыми я работаю. Например, одни проекты ограничиваются использованием Winston и Director, а другие используют весь стек».

Элис: «Я выбрала hapi для своего следующего проекта. Этот фреймворк минимален, поэтому я могу приспособить его к уникальным требованиям проекта. Большая часть кода будет относиться к Node, не полагаясь на какой-то конкретный фреймворк. Я чувствую, что это решение хорошо работает с hapi».

5.13. Заключение

- Koa — облегченный минимальный фреймворк, использующий синтаксис генераторов для промежуточного ПО. Он хорошо подходит для хостинга одностраничных веб-приложений, зависящих от внешних веб-API.
- hapi ориентируется на серверы HTTP и маршруты. hapi хорошо подходит для облегченных внутренних подсистем, состоящих из множества мелких сервисов.
- Flatiron — набор несвязных модулей, которые могут использоваться либо как веб-фреймворк MVC, либо как более облегченная библиотека Express. Flatiron обладает совместимостью с промежуточным ПО Connect.
- Kraken строится на базе Express с добавлением средств безопасности; может использоваться для реализации MVC, содержит поддержку ORM и систему шаблонизации.
- Sails.js — фреймворк MVC, построенный под влиянием Rails/Django; содержит ORM и систему шаблонов.
- DerbyJS — изоморфный фреймворк, хорошо подходящий для приложений реального времени.
- LoopBack снимает необходимость в написании стереотипного кода для быстрого генерирования REST API с полноценной поддержкой баз данных и API Explorer.

6 Connect и Express

Из главы 3 вы узнали, как построить простое приложение Express. В этой главе Express и Connect рассматриваются более подробно. Эти два популярных модуля Node используются многими веб-разработчиками. Эта глава показывает, как строить веб-приложения и REST API с применением наиболее часто применяемых паттернов.

CONNECT И EXPRESS

Концепции, обсуждаемые в следующем разделе, непосредственно применимы к высокоуровневому фреймворку Express, потому что он расширяет и дополняет Connect высокоуровневыми вспомогательными средствами. После чтения этого раздела вы будете твердо понимать, как работают промежуточные компоненты Connect и как объединять их для создания приложения. Другие веб-фреймворки Node работают аналогичным образом, так что изучение Connect даст вам преимущества при изучении новых фреймворков.

Для начала мы покажем, как создать базовое приложение Connect. Далее в этой главе вы увидите, как построить более сложное приложение Express с использованием популярных приемов Express.

6.1. Connect

В этом разделе вы познакомитесь с Connect. Вы увидите, как промежуточные компоненты могут использоваться для построения простых веб-приложений и насколько важную роль играет порядок промежуточных компонентов. Позднее это поможет вам при построении приложений Express с более высокой степенью модульности.

6.1.1. Настройка приложения Connect

Express строится на базе Connect, но знаете ли вы, что полностью функциональное веб-приложение можно построить и просто на базе Connect? Чтобы загрузить и установить Connect из реестра npm, введите следующую команду:

```
$ npm install connect@3.4.0
```

Минимальное приложение Connect выглядит так:

```
const app = require('connect')();
app.use((req, res, next) => {
  res.end('Hello, world!');
});
app.listen(3000);
```

Это простое приложение (находящееся в папке ch06-connect-and-express/hello-world в архиве примеров) отвечает сообщением `Hello, world!`. Функция, передаваемая `app.use`, представляет собой *промежуточный компонент* (middleware component), который завершает запрос отправкой текста `Hello, world!` как ответа. Промежуточные компоненты образуют основу для всех приложений Connect и Express. Рассмотрим их более подробно.

6.1.2. Как работают промежуточные компоненты Connect

В Connect *промежуточный компонент* представляет собой функцию JavaScript, которая по соглашению получает три аргумента: объект запроса, объект ответа и аргумент, которому обычно присваивается имя `next`, — функция обратного вызова, указывающая, что компонент завершил свою работу и выполнение может быть продолжено следующим промежуточным компонентом.

До того как ваши промежуточные компоненты начнут выполняться, Connect использует диспетчера, который получает запросы и передает их первому промежуточному компоненту, добавленному в приложении. На рис. 6.1 показано типичное приложение Connect, состоящее из диспетчера и набора промежуточных компонентов, включающего в себя компонент ведения журнала, парсер тела запроса, сервер статических файлов и специальное промежуточное программное обеспечение.

Как видите, архитектура API промежуточного ПО означает, что более сложное поведение может строиться из меньших структурных блоков. В следующем разделе вы увидите, как это делается посредством объединения компонентов.

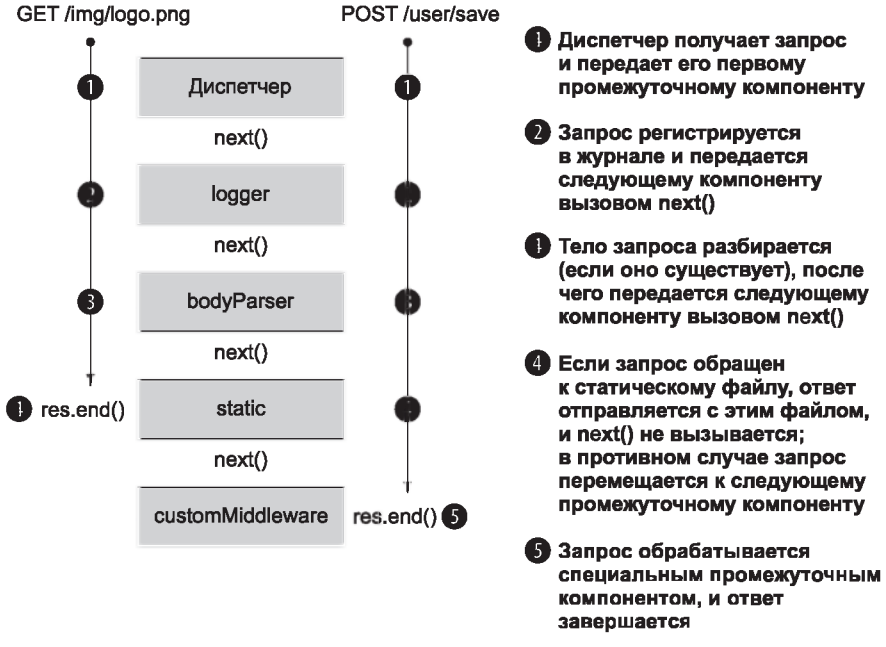


Рис. 6.1. Жизненный цикл двух запросов HTTP в сервере HTTP

6.1.3. Объединение промежуточных компонентов

Connect предоставляет метод с именем `use` для объединения промежуточных компонентов. Определим две функции промежуточных компонентов и добавим их в приложение: первая — уже упоминавшаяся ранее функция «`Hello, World`», а другая — функция `logger`.

Листинг 6.1. Использование нескольких промежуточных компонентов Connect

```
const connect = require('connect');
function logger(req, res, next) {        // Выводит метод HTTP и URL запроса, после чего вызывает next().
  console.log('%s %s', req.method, req.url);
  next();
}
function hello(req, res) {               // Завершает ответ на запрос HTTP текстом «hello world».
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
connect()
  .use(logger)
  .use(hello)
  .listen(3000);
```

Промежуточные компоненты имеют две сигнатуры: с `next` и без. Это объясняется тем, что компонент завершает ответ HTTP и возвращать управление диспетчеру ему не нужно.

Функция `use()` возвращает экземпляр приложения Connect для поддержки сцепления методов, как было показано выше. Обратите внимание: сцепление вызовов `.use()` при этом не требуется:

```
const app = connect();
app.use(logger);
app.use(hello);
app.listen(3000);
```

Итак, у вас заработало простое приложение «Hello World»; теперь можно посмотреть, почему важен порядок вызовов `.use()` промежуточных компонентов и как стратегически использовать этот порядок для внесения изменений в работу приложения.

6.1.4. Упорядочение компонентов

Порядок промежуточных компонентов в вашем приложении может кардинально влиять на его поведение. Опустив `next()`, вы остановите выполнение приложения, а объединение компонентов позволяет реализовать такие функциональные возможности, как аутентификация.

Что произойдет, если промежуточные компоненты не вызовут `next`? Вернемся к предыдущему примеру «Hello World», в котором первым используется компонент `logger`, а за ним следует компонент `hello`. В этом примере Connect направляет вывод в `stdout`, а затем реагирует на запрос HTTP. Но что произойдет, если компоненты будут следовать в другом порядке?

Листинг 6.2. Ошибка: компонент hello предшествует компоненту logger

```
const connect = require('connect');
function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
}
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
const app = connect()
  .use(hello)
  .use(logger)
  .listen(3000);
```

Всегда вызывает next(), поэтому следующий компонент будет вызван.

Не вызывает next(), потому что компонент отвечает на запрос.

Компонент logger никогда не будет вызван, потому что hello не вызывает next().

В этом примере промежуточный компонент `hello` вызывается первым и реагирует на запрос HTTP, как и ожидалось. Однако компонент `logger` вызван никогда не будет, потому что `hello` никогда не вызывает `next()`, поэтому управление никогда не возвращается диспетчеру для вызова следующего промежуточного компонента. Мораль: если компонент не вызывает `next()`, то остальные компоненты в цепочке вызываться не будут.

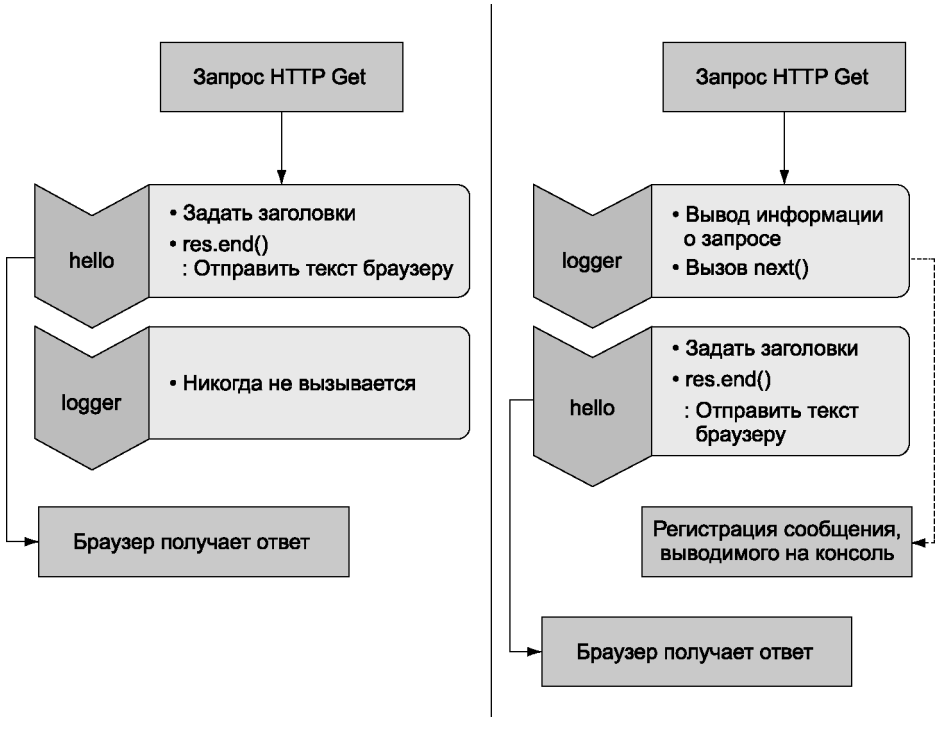


Рис. 6.2. Порядок следования промежуточных компонентов важен

Рисунок 6.2 показывает, как в этом примере обходится компонент `logger` и как исправить эту проблему.

Как видите, размещать `hello` перед `logger` достаточно бессмысленно, но при правильном использовании упорядочение может принести пользу.

6.1.5. Создание настраиваемых промежуточных компонентов

Вы усвоили некоторые основы использования промежуточных компонентов; пора рассмотреть тему более подробно, чтобы понять, как создавать более универсальные и общие промежуточные компоненты.

Обычно при построении промежуточных компонентов используется простое соглашение, которое открывает перед разработчиком возможности для конфигурации: использование функции, которая возвращает другую функцию (замыкание). Базовая структура настраиваемых промежуточных компонентов такого рода выглядит так:

```
function setup(options) {
  // Логика инициализации

  return function(req, res, next) {
    // Логика промежуточного компонента

  }
}
```

Здесь выполняется дополнительная инициализация промежуточного компонента.

Значение options остается доступным, хотя внешняя функция вернула управление.

Пример использования таких компонентов:

```
app.use(setup({ some: 'options' }));
```

Обратите внимание: функция `setup` вызывается в строке `app.use`, тогда как в предыдущих примерах мы просто передавали ссылку на функцию.

В этом разделе данный прием будет применен для построения трех настраиваемых, пригодных для повторного использования промежуточных компонентов:

- компонента `logger` с настраиваемым форматом вывода;
- компонента маршрутизации для вызова функций в зависимости от запрашиваемого URL;
- компонента перезаписи URL, преобразующего альтернативные части URL (slugs) в идентификаторы.

Начнем с доработки компонента `logger`, чтобы расширить возможности его настройки. Промежуточный компонент `logger`, созданный ранее в этой главе, не поддерживал настройку. Он был жестко запрограммирован для вывода значений `req.method` и `req.url` при вызове. Но что, если в какой-то момент вы захотите изменить состав выводимой информации?

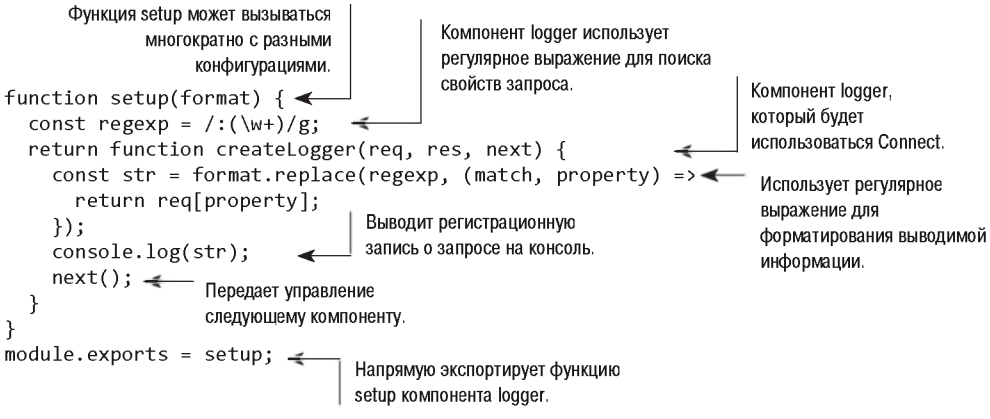
На практике использование настраиваемого промежуточного компонента почти не отличается от использования любого другого промежуточного компонента, который создавался до настоящего момента, если не считать того, что компоненту можно передать дополнительные аргументы для изменения его поведения. Использование настраиваемого компонента в приложении обычно выглядит примерно так, как в следующем примере (где `logger` получает строку с описанием формата вывода):

```
const app = connect()
  .use(logger(':method :url'))
  .use(hello);
```

Для реализации настраиваемого компонента `logger` сначала необходимо определить функцию `setup`, которая получает один строковый аргумент (в этом примере ему присваивается имя `format`). При вызове `setup` возвращается функция, которая будет использоваться промежуточным компонентом Connect. Возвращаемый компонент сохраняет доступ к переменной `format` даже после того, как функция `setup`

вернет управление, потому что он определяется в том же замыкании JavaScript. Затем `logger` заменяет лексемы в форматной строке соответствующими свойствами запроса из объекта `req`, выводит информацию в `stdout` и вызывает `next()`, как показано в листинге 6.3.

Листинг 6.3. Настраиваемый промежуточный компонент logger для Connect




```
function setup(format) {
  const regexp = /:(\w+)/g;
  return function createLogger(req, res, next) {
    const str = format.replace(regexp, (match, property) =>
      return req[property];
    });
    console.log(str);
    next();
  }
}
module.exports = setup;
```

Так как компонент `logger` создавался как настраиваемый, вы можете многократно вызывать `.use()` для `logger` в одном приложении с разными конфигурациями или же использовать этот код в других приложениях, которые вы разработаете в будущем. Простая концепция настраиваемых промежуточных компонентов хорошо известна в сообществе Connect и применяется ко всем базовым промежуточным компонентам Connect для последовательности.

Чтобы использовать компонент `logger` из листинга 6.3, передайте ему строку с включением некоторых свойств объекта `request`. Например, с вызовом `.use(setup(':method :url'))` для каждого запроса будет выводиться метод HTTP (GET, POST и т. д.) и URL-адрес.

Прежде чем переходить к Express, посмотрим, как в Connect поддерживается обработка ошибок.

6.1.6. Использование промежуточных компонентов для обработки ошибок

Во всех приложениях возникают ошибки (как на системном уровне, так и на пользовательском), и вы сильно упростите себе жизнь, если будете готовы к ошибочным ситуациям — даже к непредвиденным. Connect реализует разновидность промежуточных компонентов с обработкой ошибок, которая следует тем же правилам, что и обычная, но наряду с объектами запроса и ответа получает объект ошибки.

Обработка ошибок в Connect намеренно сделана минимальной, чтобы разработчик мог сам задать способ обработки ошибок. Например, через промежуточное ПО могут проходить только системные ошибки и ошибки приложения (например, переменная `foo` не определена), или ошибки пользователя (пароль недействителен), или их некоторая комбинация. Connect позволяет выбрать, что лучше подходит для вашего приложения.

В этом разделе мы используем обе разновидности, и вы узнаете, как работают промежуточные компоненты обработки ошибок. Также будут представлены некоторые полезные паттерны, которые могут применяться в следующих областях:

- использование обработчика ошибок Connect по умолчанию;
- самостоятельная обработка ошибок приложения;
- использование нескольких компонентов обработки ошибок.

А теперь посмотрим, как Connect обрабатывает ошибки без какой-либо настройки.

Обработчик ошибок Connect по умолчанию

Рассмотрим следующий промежуточный компонент, который выдает ошибку `ReferenceError`, потому что функция `foo()` не определена приложением:

```
const connect = require('connect')
connect()
  .use((req, res) => {
    foo();
    res.setHeader('Content-Type', 'text/plain');
    res.end('hello world');
  })
.listen(3000)
```

По умолчанию Connect отвечает кодом статуса 500, телом ответа с текстом «Internal Server Error» и дополнительной информацией о самой ошибке. Это неплохо, но в любом реальном приложении вы, скорее всего, предпочтете реализовать более специализированную обработку ошибок — например, отправлять их демону ведения журнала.

Самостоятельная обработка ошибок приложения

Connect также предоставляет средства для самостоятельной обработки ошибок с использованием промежуточных компонентов. Например, в ходе разработки вы можете отвечать клиенту JSON-представлением ошибки для быстрого и простого получения информации, а в итоговой версии приложение будет отвечать простым сообщением «Server error», чтобы не раскрывать конфиденциальную внутреннюю информацию (трассировку стека, имена файлов, номера строк) потенциальному атакующему.

Функция обработки ошибок промежуточных компонентов должна определяться с четырьмя аргументами, `err`, `req`, `res` и `next` (листинг 6.4), тогда как обычные промежуточные компоненты получают аргументы `req`, `res` и `next`. Следующий листинг демонстрирует пример промежуточного компонента с обработкой ошибок. За полным примером сервера обращайтесь к архиву исходного кода в папке ch06-connect-and-express/listing6_4.

Листинг 6.4. Промежуточный компонент обработки ошибок в Connect

```
const env = process.env.NODE_ENV || 'development';

function errorHandler(err, req, res, next) {
     res.statusCode = 500;
     switch (env) {
       case 'development':
         console.error('Error:');
         console.error(err);
         res.setHeader('Content-Type', 'application/json');
         res.end(JSON.stringify(err));
         break;
       default:
         res.end('Server error');
     }
}

module.exports = errorHandler;
```

Промежуточный компонент обработки ошибок использует четыре аргумента.

Поведение промежуточного компонента errorHandler зависит от значения NODE_ENV.

Использование Node_Env для определения режима работы

Один из распространенных приемов Connect — использование переменной окружения `NODE_ENV` (`process.env.NODE_ENV`) для переключения поведения между вариантами условий работы сервера — например, режимом разработки и режимом реальной эксплуатации.

Обнаружив ошибку, Connect переключается на вызов только промежуточных компонентов, предназначенных для обработки ошибок (рис. 6.3).

Представьте, что ваше приложение предоставляет возможность выполнения аутентификации для работы в административной области блога. Если компонент маршрутизации для пользовательских маршрутов вызывает ошибку, компоненты `blog` и `admin` будут пропущены, потому что они не являются компонентами обработки ошибок — они определяют только три аргумента. Затем Connect видит, что `errorHandler` получает аргумент ошибки, и активизирует его. Промежуточные компоненты выглядят примерно так:

```
connect()
  .use(router(require('./routes/user')))
  .use(router(require('./routes/blog'))) // Пропускается
  .use(router(require('./routes/admin'))) // Пропускается
  .use(errorHandler);
```

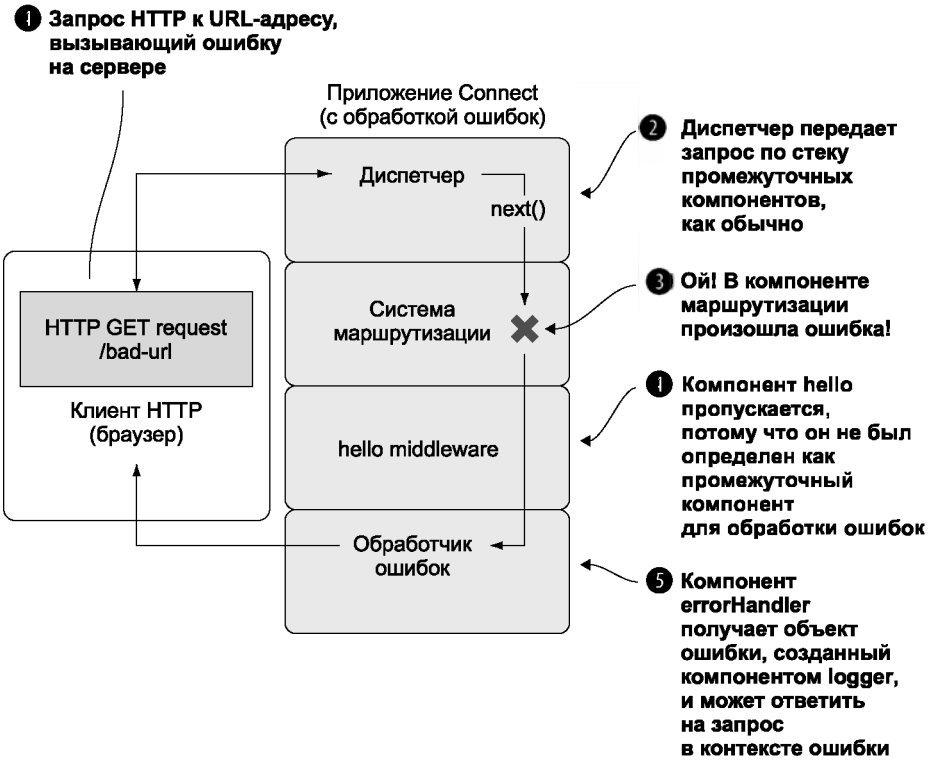


Рис. 6.3. Жизненный цикл запроса HTTP, порождающего ошибку сервера Connect

Функциональность ускоренной обработки в зависимости от хода выполнения — одна из фундаментальных концепций, применяемых для организации приложений Express.

Теперь, когда вы изучили основы Connect, можно перейти к более подробному изучению Express.

6.2. Express

Express — популярный веб-фреймворк, который некогда строился на базе Connect, но и сейчас остается совместимым с промежуточными компонентами Connect. Хотя Express включает в себя базовую функциональность (такую, как предоставление статических файлов, маршрутизация URL и конфигурация приложения), этот фреймворк все равно остается минимальным. Он предоставляет достаточную структуру для создания блоков, подходящих для повторного использования, без излишних ограничений для вашей практики разработки.

В нескольких ближайших разделах мы реализуем приложение Express при помощи генератора приложений Express. Процесс будет описан более подробно, чем краткий обзор в главе 3, поэтому в конце главы вы будете достаточно хорошо понимать Express для самостоятельного построения веб-приложений Express и REST-совместимых API. По ходу главы к заготовке приложения будет добавляться новая функциональность, а к ее концу будет построено полноценное приложение.

6.2.1. Генерирование заготовки приложения

Express не требует от разработчика соблюдения определенной структуры приложения. Вы можете разместить маршруты в любом количестве файлов, разместить открытые активы в любом каталоге на свое усмотрение и т. д. Минимальное приложение Express получается совсем коротким — как в листинге 6.5, реализующем полностью работоспособный сервер HTTP.

Листинг 6.5. Минимальное приложение Express

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

app.get('/', (req, res) => {  <-- Отвечает на любой веб-запрос к /.
  res.send('Hello');          <-- Отправляет строку «Hello» как текст ответа.
});

app.listen(3000);             <-- Прослушивает порт 3000.
```

Программа командной строки `express(1)`, входящая в пакет `express-generator` (*www.npmjs.com/package/express-generator*), создает заготовку приложения за вас. Сгенерированное приложение — хорошая отправная точка для начинающих разработчиков Express, так как в сгенерированное приложение включаются шаблоны, открытые активы, конфигурация и многое другое.

Стандартная заготовка приложения, генерируемая `express(1)`, состоит из нескольких каталогов и файлов (рис. 6.4). Этот набор был разработан для того, чтобы разработчики могли освоиться с Express за считанные минуты, но структура приложения полностью определяется вами и вашей командой.

В примере этой главы используются шаблоны Embedded JavaScript (EJS), которые по своей структуре напоминают разметку HTML. Язык EJS близок к PHP, JSP (для Java) и ERB (для Ruby): серверный код JavaScript встраивается в документ HTML и выполняется перед отправкой клиенту. EJS более подробно рассматривается в главе 7.

В этом разделе будут рассмотрены следующие операции:

- глобальная установка Express с npm;
- генерирование приложения;
- анализ приложения и установка зависимостей.

Рис. 6.4. Структура стандартной заготовки приложения с использованием шаблонов EJS

Установка Express

Выполните глобальную установку `express-generator` с npm:

```
$ npm install -g express-generator
```

Для просмотра всех доступных параметров используется флаг `--help` (рис. 6.5).

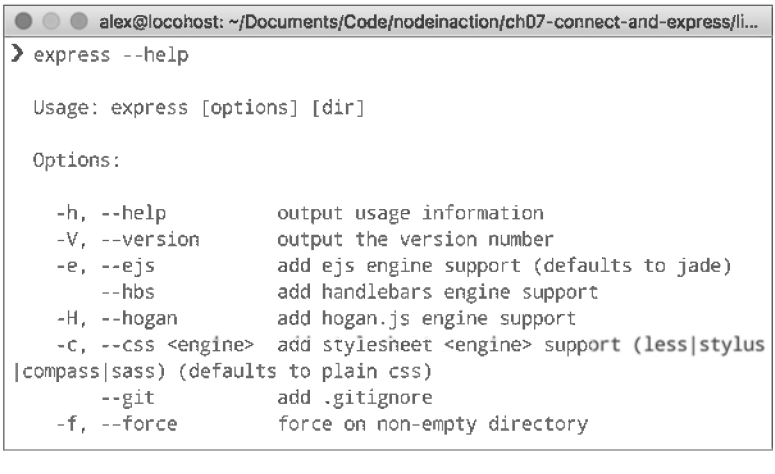


Рис. 6.5. Справочная информация Express

Некоторые из этих параметров генерируют небольшие части приложения за вас. Например, вы можете приказать сгенерировать фиктивный шаблонный файл для выбранного ядра шаблонизации. Аналогичным образом, если задать препроцессор CSS при помощи параметра `--css`, для него будет сгенерирован фиктивный шаблонный файл.

После того как исполняемые файлы будут установлены, посмотрим, как сгенерировать приложение, над которым мы будем работать в этой главе.

Генерирование приложения

В нашем приложении будет использован флаг `-e` (или `--ejs`), выбирающий ядро шаблонизации EJS. Выполните команду `express -e shoutbox`. Чтобы воспроизвести примеры кода из нашего репозитория GitHub, введите команду `express -e listing6_6`.

Приложение будет создано в каталоге **shoutbox**. Оно содержит файл **package.json** для описания проекта и зависимостей, сам файл приложения, каталоги с общедоступными файлами и каталог для обработчиков маршрутов.

Анализ приложения

Давайте поближе присмотримся к тому, что же было сгенерировано. Откройте файл **package.json** в редакторе, чтобы просмотреть зависимости приложения (рис. 6.6). Express не может угадать, какая версия зависимостей вам понадобится, поэтому рекомендуется указать основную/дополнительную версию и версию исправления модуля, чтобы не создать случайные ошибки. Например, выражение `"express":"~4.13.1"` явно задает нужные версии и обеспечивает использование идентичного кода во всех вариантах установки.

```
{
  "name": "listing6_6",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

Рис. 6.6. Содержимое сгенерированного файла package.json

Рассмотрим файл приложения, сгенерированный `express(1)` (листинг 6.6). Пока этот файл останется в исходном виде. Промежуточные компоненты уже должны быть знакомы вам по разделам этой главы, посвященным Connect, но вам стоит взглянуть на то, как задается конфигурация промежуточных компонентов по умолчанию.

Листинг 6.6. Сгенерированная заготовка приложения Express

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var routes = require('./routes/index');
var users = require('./routes/users');
var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);

app.use(function(req, res, next) {
  let err = new Error('Not Found');
  err.status = 404;
  next(err);
});

if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});

module.exports = app;
```

Предоставляет значок приложения по умолчанию.

Выводит журналы в формате, удобном для разработки.

Разбирает тела запросов.

Предоставляет статические файлы из каталога ./public.

Задает маршруты приложения.

Выводит страницы ошибок в формате HTML в режиме разработки.

Файлы package.json и app.js существуют, но приложение еще не работает, потому что зависимости еще не установлены. Каждый раз, когда вы генерируете файл package.json из `express(1)`, для него необходимо установить зависимости. Выполните команду `npm install`, а затем запустите приложение командой `npm start`.

Проверьте работоспособность приложения, открыв адрес *http://localhost:3000* в браузере. Внешний вид приложения по умолчанию показан на рис. 6.7.

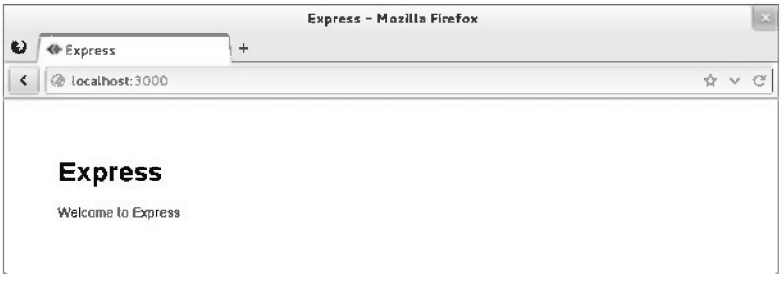


Рис. 6.7. Приложение Express по умолчанию

Ознакомившись со сгенерированной заготовкой, можно переходить к построению реального приложения Express. Приложение будет выполнять функции простейшего чата, в котором пользователи смогут публиковать сообщения. При построении приложений такого рода опытные Express-разработчики обычно начинают с планирования API и, соответственно, необходимых маршрутов и ресурсов.

Планирование приложения Shoutbox

Требования к приложению:

1. Регистрация учетных записей пользователей, процедуры входа и выхода.
2. Возможность публикации сообщений (записей).
3. Страничный просмотр записей.
4. Простой REST API с поддержкой аутентификации.

Для этого необходимо организовать хранение данных и обработку данных при аутентификации. Также следует предусмотреть проверку данных, введенных пользователем. Необходимые маршруты выглядят так:

- маршруты API;
- `GET /api/entries`: получение списка записей;
- `GET /api/entries/page`: получение одной страницы записей;
- `POST /api/entry`: создание новой записи;
- маршруты пользовательского интерфейса;

- `GET /post`: форма для новой записи;
- `POST /post`: публикация новой записи;
- `GET /register`: вывод формы регистрации;
- `POST /register`: создание новой учетной записи;
- `GET /login`: вывод формы входа;
- `POST /login`: вход в приложение;
- `GET /logout`: выход из приложения.

Такая структура типична для большинства веб-приложений. Возможно, вы сможете использовать пример этой главы как образец для построения ваших будущих приложений.

В листинге 6.6 можно заметить несколько вызовов `app.set`:

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

Так выполняется настройка конфигурации приложения Express. В следующем разделе конфигурация Express рассматривается более подробно.

6.2.2. Настройка конфигурации Express и приложения

Требования вашего приложения зависят от условий, в которых оно выполняется. Например, во время разработки может выводиться более подробная журнальная информация, а в условиях реальной эксплуатации — сокращенный вариант журнала. Кроме настройки функциональности для конкретного окружения вы можете определить настройки уровня приложения, чтобы передать Express информацию об используемом ядре шаблонов и о том, где хранятся шаблоны. Express также позволяет определять нестандартные параметры конфигурации в форме пар «ключ-значение».

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Для настройки переменных окружения в системах UNIX используется следующая команда:

```
$ NODE_ENV=production node app
```

В системе Windows код выглядит так:

```
$ set NODE_ENV=production
$ node app
```

Для работы с переменными окружения в приложении используется объект process.env.

В Express реализована минимальная система работы с окружением. Она состоит из нескольких методов, использующих значение переменной окружения `NODE_ENV`:

- `app.set()`;
- `app.get()`;
- `app.enable()`;
- `app.disable()`;
- `app.enabled()`;
- `app.disabled()`.

В этом разделе будет показано, как использовать систему конфигурации для настройки поведения Express и как пользоваться этой системой для ваших собственных целей в ходе разработки.

Разберемся, что же понимается под *конфигурацией на базе окружения*. Хотя переменная окружения `NODE_ENV` появилась в Express, многие другие фреймворки Node приняли ее как механизм оповещения приложений Node о том, в каких условиях они работают (по умолчанию режим разработки).

Метод `app.configure()` получает необязательные строки, представляющие переменные окружения, и функцию. Если переменная окружения соответствует переданной строке, немедленно активизируется функция обратного вызова; если задана только функция, она вызывается для всех переменных окружения. Имена переменных окружения выбираются совершенно произвольно. Например, можно определить переменные `development`, `stage`, `test` и `production` (или сокращенно `prod`):

```
if (app.get('env') === 'development') {
  app.use(express.errorHandler());
}
```

Express использует систему конфигурации в своей внутренней реализации, что позволяет вам настроить поведение Express, но система конфигурации также доступна для вашего личного использования.

Express также предоставляет логические разновидности `app.set()` и `app.get()`. Например, вызов `app.enable(setting)` эквивалентен `app.set(setting, true)`, а `app.enabled(setting)` может использоваться для проверки установки флага (парные методы для `false` — `app.disable(setting)` и `app.disabled(setting)`).

Еще один полезный параметр, часто используемый при разработке API с Express, — `json spaces`. Если добавить его в файл app.js, разметка JSON будет выводиться в более удобочитаемом формате:

```
app.set('json spaces', 2);
```

Теперь вы знаете, как использовать систему конфигурации для ваших целей, и мы можем перейти к визуализации представлений в Express.

6.2.3. Визуализация представлений

В приложении этой главы будут использоваться шаблоны EJS, хотя, как упоминалось ранее, использоваться может практически любое ядро шаблонов, известное в сообществе Node. Если вы не знакомы с EJS, не беспокойтесь. EJS имеет много общего с другими языками шаблонов, встречающимися в других платформах веб-разработки (PHP, JSP, ERB). Основы EJS будут представлены в этой главе, но EJS с несколькими другими ядрами шаблонов будет более подробно рассматриваться в главе 7.

Визуализация представлений, будь то визуализация целой страницы HTML, фрагмента HTML или канала RSS, критична практически для каждого приложения. Концепция проста: вы передаете данные представлению, эти данные преобразуются — обычно в формат HTML для веб-приложений. Вероятно, вы уже знакомы с понятием представления (view), потому что многие фреймворки представляют аналогичную функциональность; на рис. 6.8 показано, как представления формируют новое представление данных.

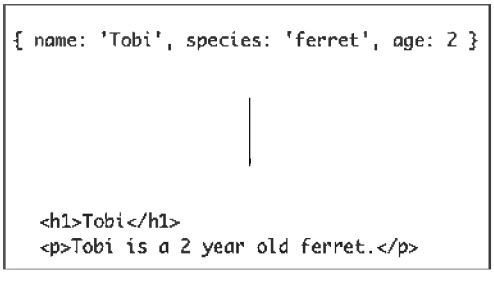


Рис. 6.8. Шаблон HTML + данные = представление данных в формате HTML

Шаблон, генерирующий шаблон на рис. 6.8, выглядит так:

```
<h1><%= name %></h1>
<p><%= name %> is a 2 year old <%= species %>.</p>
```

Express предоставляет два способа визуализации представлений: на уровне приложения вызовом `app.render()` и в ответе вызовом `res.render()`, который использует первый во внутренней реализации. В этой главе используется только `res.render()`. Заглянув в файл ./routes/index.js, вы увидите, что в нем определяется функция, вызывающая `res.render('index')` для визуализации шаблона ./views/index.ejs, как показано в следующем фрагменте (из listing6_8):

```
router.get('/', (req, res, next) => {
  res.render('index', { title: 'Express' });
});
```

Прежде чем рассматривать `res.render()` более подробно, посмотрим, как настроить систему представлений.

Настройка системы представлений

Настройка системы представлений Express выполняется достаточно просто. Но несмотря на то, что `express(1)` генерирует конфигурацию за вас, будет полезно знать, что же происходит «за кулисами», — эта информация пригодится вам для внесения изменений. Мы ограничимся тремя областями:

- настройка поиска представлений;
- настройка ядра шаблонов по умолчанию;
- включение кэширования представлений для сокращения файлового ввода/вывода.

Начнем с параметра `views`.

Изменение каталога поиска

В следующем фрагменте показан параметр `views`, сгенерированный Express:

```
app.set('views', __dirname + '/views');
```

Он задает путь, который будет использоваться Express при поиске представлений. Мы рекомендуем включать в путь переменную `__dirname`, чтобы приложение не зависело от того, что текущий рабочий каталог является корневым каталогом приложения.

__DIRNAME

__dirname (с двумя подчеркиваниями) — глобальная переменная Node, определяющая каталог, в котором находится файл, выполняемый в настоящее время. Часто в процессе разработки этот каталог совпадает с текущим рабочим каталогом, но при реальной эксплуатации приложения исполняемый файл может быть запущен из другого каталога. Переменная __dirname обеспечивает согласование путей между разными рабочими средами.

Следующий параметр конфигурации — `view engine`.

Ядро шаблонов по умолчанию

Когда приложение было сгенерировано `express(1)`, параметру `view engine` было присвоено значение `ejs`, потому что ядро шаблонов EJS было выбрано параметром командной строки `-e`. Эта настройка позволяет использовать `index` вместо `index.`

`ejs`. В противном случае Express требует указать расширение для того, чтобы определить используемое ядро шаблонов.

Почему Express вообще учитывает расширения? Они позволяют использовать несколько ядер шаблонов в одном приложении Express с сохранением ясности API для стандартных сценариев использования (впрочем, в большинстве приложений используется только одно ядро шаблонов).

Предположим, вы обнаружили, что каналы RSS проще программируются с другим ядром шаблонов, или вы переходите с одного ядра на другое. По умолчанию в приложении используется Pug, а для маршрута `/feed` — EJS, на что в следующем коде указывает расширение `.ejs`:

```
app.set('view engine', 'pug');
app.get('/', function(){
  res.render('index');
 });
app.get('/feed', function(){
  res.render('rss.ejs');
});
```

СИНХРОНИЗАЦИЯ PACKAGE.JSON

Помните, что каждое дополнительное ядро шаблонов, которое вы хотите использовать в приложении, должно быть добавлено в объект зависимостей файла package.json. Такие пакеты должны устанавливаться командой `npm install --save` *имя_пакета* и удаляться из node_modules и package.json командой `npm uninstall --save` *имя_пакета*. Такой подход упрощает эксперименты с разными ядрами шаблонов, когда вы еще только подбираете ядро для своего проекта.

Кэширование представлений

Параметр `view cache` включается по умолчанию в среде эксплуатации приложения и предотвращает выполнение дискового ввода/вывода при последующих вызовах `render()`. Содержимое шаблона сохраняется в памяти, что приводит к значительному повышению быстродействия. Побочный эффект заключается в том, что вы не сможете редактировать файлы шаблонов без перезапуска сервера; именно по этой причине данная настройка отключена в процессе разработки. Вероятно, в условиях реальной эксплуатации ее следует держать включенной.

Как показано на рис. 6.9, при отключенном кэшировании представлений шаблон читается с диска при каждом запросе. Именно этот режим позволяет вносить изменения в шаблон без перезапуска приложения. При включенном кэшировании обращение к диску происходит только один раз на шаблон.

Итак, мы показали, как механизм кэширования представлений способствует повышению быстродействия приложения (кроме среды разработки). Теперь посмотрим, как Express находит представления для визуализации.

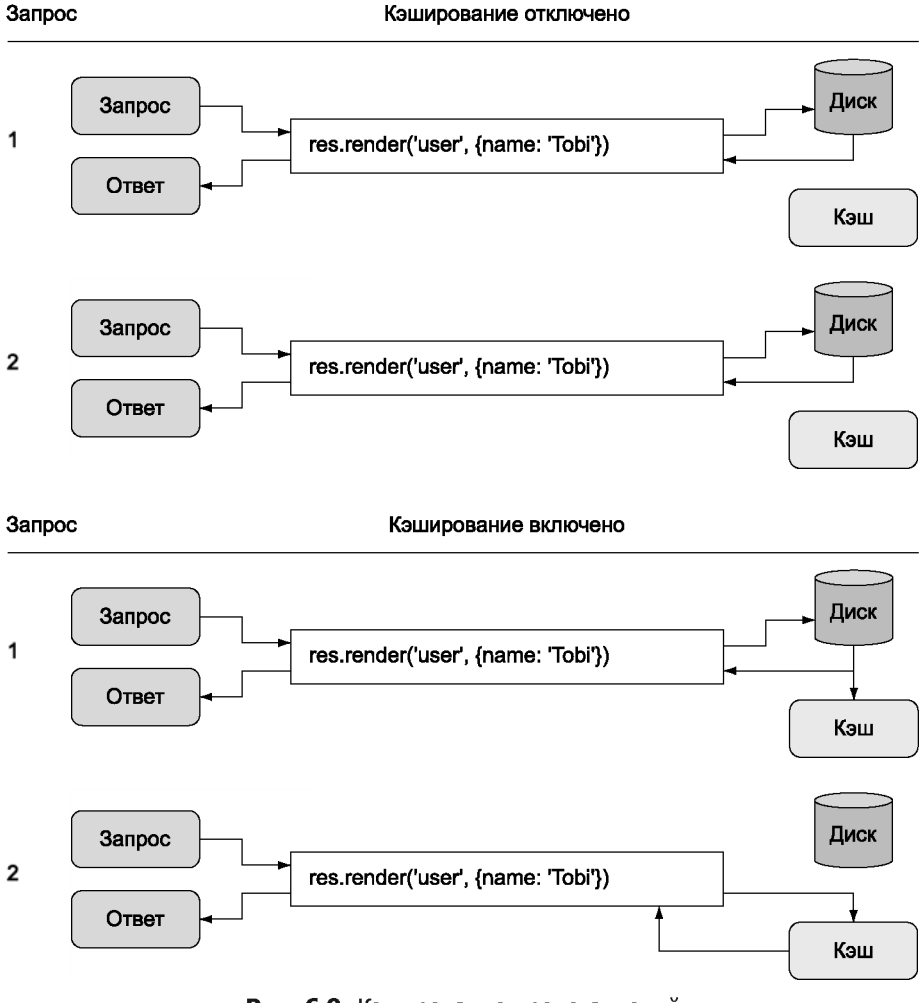


Рис. 6.9. Кэширование представлений

Поиск представлений

Процесс поиска представлений проходит по тому же принципу, что и для функции Node `require()`. При вызове `res.render()` или `app.render()` Express сначала проверяет, существует ли файл по абсолютному пути. Затем Express проводит поиск относительно каталога **views**. Наконец, Express проверяет файл **index**. На рис. 6.10 этот процесс изображен в виде блок-схемы.

Поскольку `ejs` назначается ядром по умолчанию, при вызове `render` расширение **.ejs** будет опущено, и файл шаблона будет разрешен правильно.

По мере эволюции приложения вам понадобятся новые представления, а иногда несколько представлений для одного ресурса. Использование поиска представлений

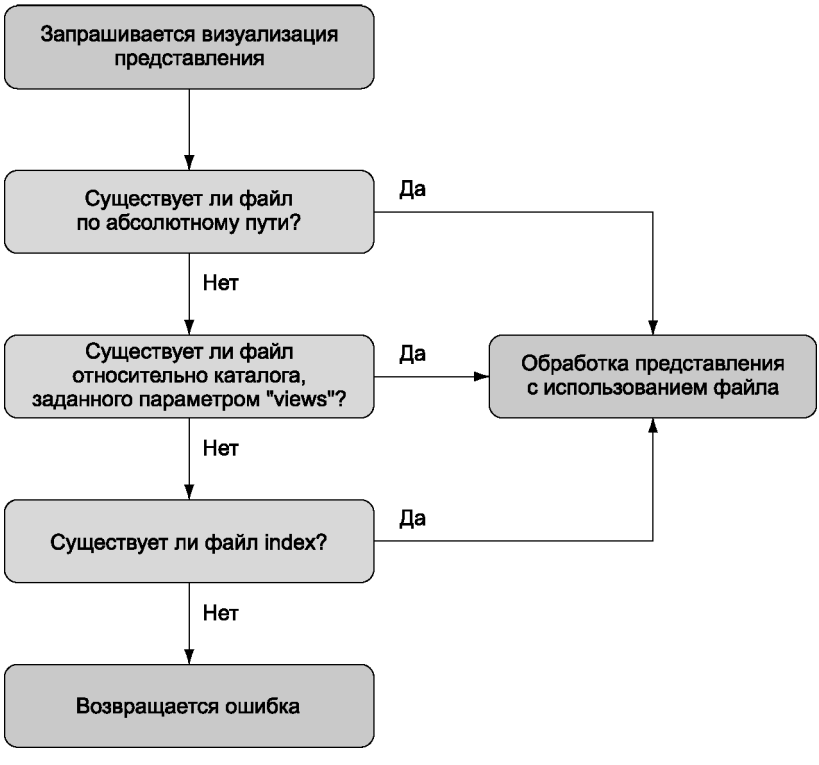


Рис. 6.10. Процесс поиска представлений Express

упрощает организацию: например, вы можете использовать подкаталоги, связанные с ресурсами, и создавать представления в этих подкаталогах.

Добавление подкаталогов позволяет устранить избыточные части имен (например, edit-entry.ejs и show-entry.ejs). Тогда Express добавляет расширение ядра шаблонов и связывает `res.render('entries/edit')` с ./views/entries/edit.ejs.

Express проверяет, существует ли файл с именем *index* в подкаталогах каталога представлений. Присваивание файлам имен ресурсов в множественном числе (например, entries) обычно подразумевает список ресурсов. Это означает, что вы можете использовать вызов `res.render('entries')` для выполнения визуализации файла views/entries/index.ejs.

Методы передачи данных представлениям

Вы уже знаете, как передавать локальные переменные непосредственно вызовам `res.render()`, но для этой цели также можно использовать и другие механизмы. Например, `app.locals` для переменных уровня приложения и `res.locals` для локальных переменных уровня запроса, которые обычно задаются промежуточными

компонентами перед итоговым методом обработки маршрутов, где происходит визуализация представлений.

Значения, переданные непосредственно `res.render()`, обладают более высоким приоритетом, чем значения, заданные в `res.locals` и `app.locals` (рис. 6.11).

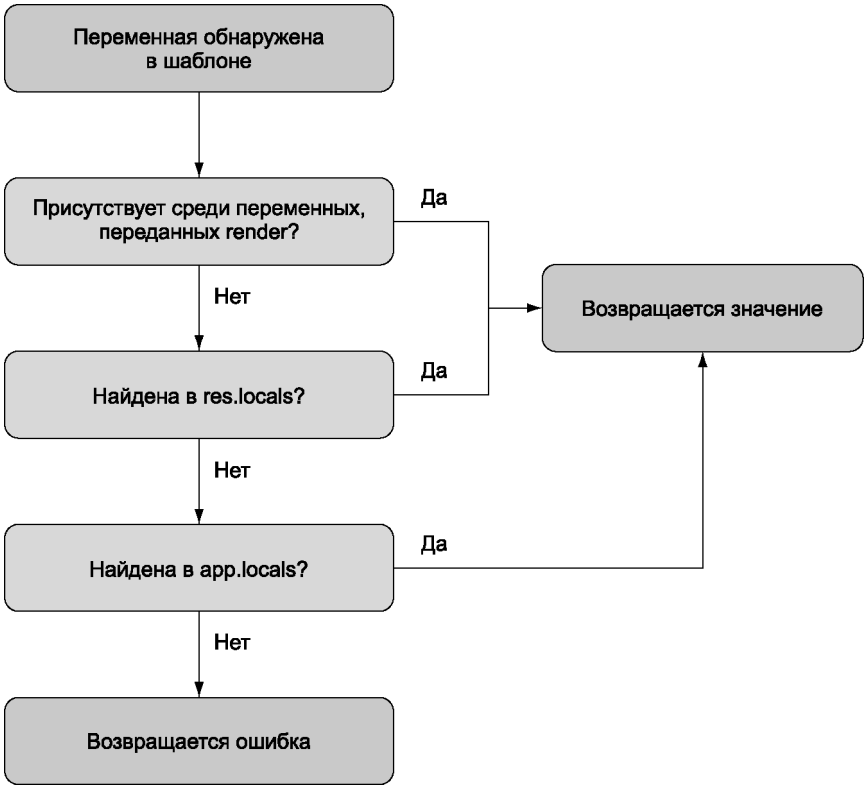


Рис. 6.11. Значения, напрямую передаваемые функции render, имеют более высокий приоритет при визуализации шаблона

По умолчанию Express открывает представлениям доступ только к одной переменной уровня приложения `settings`; она представляет собой объект со всеми значениями, заданными вызовами `app.set()`. Например, при использовании вызова `app.set('title', 'My Application')` шаблону предоставляется значение `settings.title`, как показано в следующем фрагменте EJS:

```
<html>
  <head>
     <title><%= settings.title %></title>
  </head>
  <body>
```

```
    <h1><%= settings.title %></h1>
    <p>Welcome to <%= settings.title %>.</p>
  </body>
```

Во внутренней реализации Express доступ к объекту предоставляется следующим кодом JavaScript:

```
app.locals.settings = app.settings;
```

Вот и все! После знакомства с визуализацией представлений и передачей им данных мы переходим к определению маршрутов и написанию обработчиков маршрутов, которые выполняют визуализацию представлений для приложения. Также в этом разделе будут созданы модели базы данных для долгосрочного хранения информации.

6.2.4. Знакомство с маршрутизацией в Express

Главная задача маршрутов Express — связать схемы URL с логикой ответа. Однако маршруты также могут связать схему URL с промежуточными компонентами. Это позволяет вам использовать промежуточные компоненты для назначения многократно используемой функциональности некоторым маршрутам.

В этом разделе рассматриваются следующие темы:

- проверка введенных данных с использованием промежуточных компонентов, специфических для маршрута;
- реализация проверки данных, специфической для маршрута;
- реализация страничной работы с данными.

Рассмотрим некоторые способы использования специализированных промежуточных компонентов, связанных с конкретным маршрутом.

Проверка данных, введенных пользователем

Чтобы у вас были данные для проверки, мы наконец-то реализуем возможность публикации записей в нашем приложении. Для этого необходимо решить ряд задач:

- создать модель данных;
- добавить маршруты, связанные с записью;
- создать формы для ввода;
- добавить логику создания записей с использованием данных, введенных на форме.

Начнем с создания модели записи.

Создание модели записи

Прежде чем двигаться дальше, необходимо установить в проекте модуль Node redis. Модуль устанавливается командой `npm install --save redis`. Если у вас еще не установлен модуль Redis, посетите сайт *http://redis.io/* и просмотрите инструкции по его установке; если вы работаете в macOS, модуль легко устанавливается из Homebrew (*http://brew.sh/*), а в Windows используется пакет Redis Chocolatey (*https://chocolatey.org/*).

Мы воспользуемся Redis, чтобы немного упростить задачу: функциональность Redis и ES6 упрощает создание облегченных моделей без сложной библиотеки баз данных. Если вы не ищете легких путей, используйте другую библиотеку баз данных (за информацией об использовании баз данных в Node обращайтесь к главе 8).

Посмотрим, как создать облегченную модель для хранения записей чата. Создайте файл **models/entry.js** для определения модели записи. Добавьте в файл код из листинга 6.7. Модель записи будет представлять собой простой класс ES6 для сохранения данных в списке Redis.

Листинг 6.7. Модель для записей

```
const redis = require('redis');
const db = redis.createClient();          <-- Создает экземпляр клиента Redis.
class Entry {
  constructor(obj) {
    for (let key in obj) {                <-- Перебирает ключи в переданном объекте.
      this[key] = obj[key];               <-- Объединяет значения.
    }
  }

  save(cb) {
    const entryJSON = JSON.stringify(this);   <-- Преобразует сохраненные данные записей в строку JSON.
    db.lpush(                             <-- Сохраняет строку JSON в списке Redis.
      'entries',
      entryJSON,
      (err) => {
        if (err) return cb(err);
        cb();
      }
    );
  }
}

module.exports = Entry;
```

После создания базовой модели следует добавить функцию `getRange` (листинг 6.8). Эта функция предназначена для выборки записей.

Листинг 6.8. Логика выборки диапазона записей

```
class Entry {
  static getRange(from, to, cb) {
    db.lrange('entries', from, to, (err, items) => {
      if (err) return cb(err);
      let entries = [];
      items.forEach((item) => {
        entries.push(JSON.parse(item));
      });
      cb(null, entries);
    });
  }
  ...
}
```

Функция Redis lrange используется для выборки записей.

Декодирует записи, ранее сохраненные в формате JSON.

После создания модели можно добавить маршруты для создания и получения списка записей.

Создание формы для ввода записей

Приложение выводит список записей, но пока не умеет добавлять их. Эта возможность будет добавлена сейчас, начиная с включения следующих строк в раздел `routing` файла app.js:

```
app.get('/post', entries.form);
app.post('/post', entries.submit);
```

Затем добавьте следующий маршрут в файл routes/entries.js. Логика маршрута заполняет шаблон, содержащий форму:

```
exports.form = (req, res) => {
  res.render('post', { title: 'Post' });
};
```

Затем шаблон EJS в листинге 6.9 создает шаблон для формы и сохраняет его в views/post.ejs.

Листинг 6.9. Форма для ввода сообщения

```
<!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>
    <form action='/post' method='post'>
      <p>
        <input type='text' name='entry[title]' placeholder='Title' />
```

Текст заголовка записи.

```
      </p>
      <p>
        <textarea name='entry[body]' placeholder='Body'></textarea>
      </p>
      <p>
        <input type='submit' value='Post' />
      </p>
    </form>
  </body>
</html>
```

Текст тела записи.

В форме используются такие имена, как `entry[title]`, поэтому потребуется расширенный разбор тела сообщения. Чтобы сменить парсер тела сообщения, откройте файл app.js и перейдите к строке

```
app.use(bodyParser.urlencoded({ extended: false }));
```

Приведите ее к следующему виду, чтобы использовать расширенный разбор тела сообщения:

```
app.use(bodyParser.urlencoded({ extended: true }));
```

Разобравшись с отображением формы, перейдем к созданию записей по отправленным данным формы.

Реализация создания записи

Чтобы добавить возможность создания записей по отправленным данным формы, добавьте логику из листинга 6.10 в файл routes/entries.js.

Листинг 6.10. Добавление записи по отправленным данным формы

```
exports.submit = (req, res, next) => {
  const data = req.body.entry;
  const user = res.locals.user;
  const username = user ? user.name : null;
  const entry = new Entry({
    username: username,
    title: data.title,
    body: data.body
  });
  entry.save((err) => {
    if (err) return next(err);
    res.redirect('/');
  });
};
```

Берется из name="entry[...]" в форме.

Промежуточный компонент для загрузки пользователей будет добавлен в листинге 6.28.

Теперь при обращении к маршруту `/post` в приложении вы сможете добавлять записи. Проблема обязательного входа в приложение перед созданием сообщений будет решена в листинге 6.21.

Разобравшись с созданием контента, перейдем к следующей задаче — построению списка записей.

Добавление списка записей

Создайте файл **routes/entries.js**. Добавьте код из листинга 6.11 для включения модели записи и экспортирования функции для вывода списка записей.

Листинг 6.11. Вывод списка записей

```
const Entry = require('../models/entry');
exports.list = (req, res, next) => {
  Entry.getRange(0, -1, (err, entries) => {    <-- Получает записи.
    if (err) return next(err);
    res.render('entries', {    <-- Строит ответ HTTP.
      title: 'Entries',
      entries: entries,
    });
  });
};
```

После определения логики маршрутов необходимо добавить шаблон EJS для их отображения. Создайте в каталоге **views** файл с именем **entries.ejs** и поместите в него код EJS, представленный в листинге 6.12.

Листинг 6.12. Представление entries.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>
    <% entries.forEach((entry) => { %>
      <div class='entry'>
        <h3><%= entry.title %></h3>
        <p><%= entry.body %></p>
        <p>Posted by <%= entry.username %></p>
      </div>
    <% }) %>
  </body>
</html>
```

Перед запуском приложения выполните команду `touch views/menu.ejs` для создания временного файла, который будет содержать меню на более поздней стадии. Когда представления и маршруты будут готовы, необходимо сообщить приложению, где искать маршруты.

Добавление маршрутов для работы с записями

Прежде чем добавлять в приложение маршруты, относящиеся к работе с записями, необходимо внести изменения в app.js. Сначала добавьте следующую команду `require` в начало файла app.js:

```
const entries = require('./routes/entries');
```

Затем также в файле app.js приведите строку с текстом `app.get('/'` к следующему виду, чтобы для любых запросов к пути `/` возвращался список записей:

```
app.get('/', entries.list);
```

Теперь при запуске приложения на главной странице выводится список записей. Разобравшись с созданием и выводом списка записей, перейдем к использованию промежуточных компонентов, привязанных к конкретным маршрутам, для проверки данных форм.

Использование специализированных промежуточных компонентов

Предположим, вы хотите, чтобы текстовое поле на форме ввода было обязательным для заполнения. Первое, что приходит в голову, — добавить проверку прямо в обратный вызов маршрута, как в следующем фрагменте. Однако такое решение не идеально, потому что логика проверки тесно связывается с конкретной формой. Во многих случаях логика проверки может быть абстрагирована в компонентах, пригодных для повторного использования; такое решение упростит и ускорит разработку, а код станет более декларативным:

```
...
exports.submit = (req, res, next) => {
  let 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 Express могут получать промежуточные компоненты, применяемые только при сопоставлении этого маршрута, перед завершающим обратным вызовом маршрута. Сами обратные вызовы маршрутов во всех промежуточных компонентах работают одинаково — даже в тех, которые мы собираемся создать для проверки данных!

Начнем с простого, но недостаточно гибкого способа реализации проверки данных в форме специализированных промежуточных компонентов.

Проверка данных формы с использованием специализированных промежуточных компонентов

Первый вариант — написание нескольких простых, но специализированных промежуточных компонентов для выполнения проверки данных. Расширение маршрута `POST /post` с такими компонентами может выглядеть так:

```
app.post('/post',
  requireEntryTitle,
  requireEntryTitleLengthAbove(4),
  entries.submit
);
```

Обратите внимание: это определение маршрута, которое обычно получает в аргументах только путь и логику маршрутизации, имеет два дополнительных аргумента с промежуточными компонентами проверки данных.

Два примера компонентов в листинге 6.13 показывают, как абстрагируется исходная логика проверки данных. Тем не менее такие компоненты все равно нельзя назвать модульными; они работают только для одного поля `entry[title]`.

Листинг 6.13. Две несовершенные попытки создания промежуточных компонентов проверки данных

```
function requireEntryTitle(req, res, next) {
  const title = req.body.entry.title;
  if (title) {
    next();
  } else {
    res.error('Title is required.');
    res.redirect('back');
  }
}
function requireEntryTitleLengthAbove(len) {
  return (req, res, next) => {
    const title = req.body.entry.title;
    if (title.length > len) {
      next();
    } else {
      res.error(`Title must be longer than ${len}.`);
      res.redirect('back');
    }
  };
}
```

Более универсальное решение — абстрагирование логики проверки с передачей имени целевого поля. Посмотрим, как это делается.

Построение гибких промежуточных компонентов проверки данных

Решение с передачей имени поля продемонстрировано в следующем фрагменте. Оно позволяет повторно использовать логику проверки данных и уменьшить объем кода, который вам потребуется написать:

```
app.post('/post',
         validate.required('entry[title]'),
         validate.lengthAbove('entry[title]', 4),
         entries.submit);
```

Замените строку `app.post('/post', entries.submit);` в разделе `routing` файла app.js этим фрагментом. Стоит заметить, что сообщество Express разработало много аналогичных библиотек для всеобщего использования; тем не менее очень полезно понимать, как работают промежуточные компоненты проверки данных и как реализовать их самостоятельно.

Создайте файл с именем ./middleware/validate.js и включите в него код из листинга 6.14. В файл validate.js мы экспортируем несколько промежуточных компонентов — в данном случае `validate.required()` и `validate.lengthAbove()`. Подробности реализации не важны; суть данного примера заключается в том, что небольшие дополнительные усилия значительно расширяют возможность использования кода в разных частях приложения.

Листинг 6.14. Реализация промежуточных компонентов проверки данных

```
function parseField(field) {                    <-- Разбирает синтаксис entry[name].
  return field
    .split(/\[|\]/)
    .filter((s) => s);
}
function getField(req, field) {                 <-- Ищет свойство на основании результатов parseField().
  let val = req.body;
  field.forEach((prop) => {
    val = val[prop];
  });
  return val;
}
exports.required = (field) => {
  field = parseField(field);                    <-- Разбирает поле.
  return (req, res, next) => {
    if (getField(req, field)) {                 <-- При каждом запросе проверяет, содержит ли поле значение.
      next();                                   <-- Если содержит, происходит переход к следующему промежуточному компоненту.
    } else {
      res.error(`${field.join(' ')} is required`);   <-- Если не содержит, выдается ошибка.
      res.redirect('back');
    }
  };
};
exports.lengthAbove = (field, len) => {
  field = parseField(field);
```

```
  return (req, res, next) => {
    if (getField(req, field).length > len) {
      next();
    } else {
      const fields = field.join(' ');
      res.error(`${fields} must have more than ${len} characters`);
      res.redirect('back');
    }
  };
};
```

Чтобы промежуточный компонент стал доступным для приложения, добавьте следующую строку в начало файла app.js:

```
const validate = require('./middleware/validate');
```

Если вы теперь опробуете приложение, то увидите, что проверка данных успешно действует. API проверки данных можно сделать еще более динамичным, но мы оставим вам эту задачу для самостоятельной работы.

6.2.5. Аутентификация пользователей

В этом разделе мы создадим систему аутентификации для приложения с нуля. Процесс будет состоять из следующих шагов:

- реализация логики для хранения данных и аутентификации зарегистрированных пользователей;
- добавление функциональности регистрации учетных записей;
- реализация входа в приложение;
- создание и использование промежуточного компонента для загрузки пользователей.

Для реализации учетных записей пользователей также будет использоваться Redis. Теперь посмотрим, как создать модель пользователя для упрощения работы с Redis в коде Node.

Сохранение и загрузка записей пользователей

В этом разделе мы реализуем загрузку, сохранение и аутентификацию пользователей. Здесь будут решены следующие задачи:

- определение зависимостей приложения с использованием файла package.json;
- создание модели пользователя;
- добавление логики загрузки и сохранения данных пользователей с использованием Redis;

- защита паролей с использованием bcrypt;
- добавление логики аутентификации для попыток входа.

bcrypt — функция хеширования с затравкой (salt), доступная в виде стороннего модуля, спроектированного специально для хеширования паролей. bcrypt особенно хорошо подходит для паролей из-за аргумента со счетчиком итераций, замедляющего попытки подбора.

Прежде чем следовать дальше, добавьте bcrypt в проект shoutbox:

```
npm install --save redis bcrypt
```

Создание модели пользователя

Теперь нужно создать модель пользователя. Создайте файл с именем **user.js** в каталоге **models/**.

В листинге 6.15 приведена модель пользователя. В этом коде включаются зависимости redis и bcrypt, после чего вызов `redis.createClient()` открывает подключение Redis. Функция `User` получает объект и объединяет свойства этого объекта со своими. Например, `new User({ name: 'tobi' })` создает объект и задает свойству `name` этого объекта значение `Tobi`.

Листинг 6.15. Начало работы над созданием модели пользователя

```
const redis = require('redis');
const bcrypt = require('bcrypt');
const db = redis.createClient();    <-- Создает долгосрочное подключение Redis.
class User {
  constructor(obj) {
    for (let key in obj) {    <-- Перебирает свойства переданного объекта.
      this[key] = obj[key];    <-- Задает каждое свойство в текущем классе.
    }
  }
}
module.exports = User;    <-- Экспортирует класс User.
```

На данный момент модель пользователя — не более чем заглушка. Необходимо добавить методы для создания и обновления записей с пользовательскими данными.

Сохранение данных пользователя в Redis

Следующий блок функциональности, который нам понадобится, — сохранение данных пользователя в Redis. Метод `save` из листинга 6.16 проверяет, существует ли идентификатор пользователя, и если да — вызывает метод `update`, индексируя идентификатор пользователя по имени и заполняя хеш Redis свойствами объекта. В противном случае пользователь, не имеющий идентификатора, считается новым

пользователем; значение `user:ids` увеличивается, чтобы пользователь получил уникальный идентификатор, а пароль хешируется перед сохранением в Redis тем же методом `update`.

Добавьте код из листинга 6.16 в файл models/user.js.

Листинг 6.16. Обновление записей пользователей

```
class User {
  // ...
  save(cb) {
    if (this.id) {                                  <-- Если идентификатор определен, то пользователь уже существует.
      this.update(cb);
    } else {
      db.incr('user:ids', (err, id) => {            <-- Создает уникальный идентификатор.
        if (err) return cb(err);
        this.id = id;                               <-- Задает идентификатор для сохранения.
        this.hashPassword((err) => {                <-- Хеширует пароль.
          if (err) return cb(err);
          this.update(cb);                          <-- Сохраняет свойства пользователей.
        });
      });
    }
  }
  update(cb) {
    const id = this.id;
    db.set(`user:id:${this.name}`, id, (err) => {   <-- Индексирует пользователей по имени.
      if (err) return cb(err);
      db.hmset(`user:${id}`, this, (err) => {       <-- Использует Redis для хранения свойств текущего класса.
        cb(err);
      });
    });
  }
```

Защита паролей

При создании пользователя свойству `.pass` присваивается пароль пользователя. Затем логика сохранения пользователей заменяет значение свойства `.pass` хешем, сгенерированным с использованием пароля.

К хешу применяется *затравка* (salt). Применение затравки на уровне пользователей помогает защититься от атак на базе радужных таблиц: затравка играет роль закрытого ключа для механизма хеширования. Вы можете воспользоваться bcrypt для генерирования 12-символьной затравки для хеша методом `genSalt()`.

АТАКИ НА БАЗЕ РАДУЖНЫХ ТАБЛИЦ

Атаки на базе радужных таблиц пытаются взломать хешированные пароли по заранее вычисленным таблицам. С этой темой можно ознакомиться в Википедии: https://ru.wikipedia.org/wiki/Радужная_таблица.

После того как затравка будет сгенерирована, вызывается функция `bcrypt.hash()`, которая хеширует свойство `.pass` и затравку. Полученное значение `hash` заменяет свойство `.pass`, после чего вызов `.update()` сохраняет его в Redis; это делается для того, чтобы в базе данных пароли не сохранялись в текстовом виде — только в виде хеша.

Листинг 6.17, который следует добавить в **models/user.js**, определяет функцию, которая создает хеш с затравкой и сохраняет его в свойстве `.pass` объекта пользователя.

Листинг 6.17. Добавление шифрования bcrypt в модель пользователя

```
class User {
  // ...
  hashPassword(cb) {
    bcrypt.genSalt(12, (err, salt) => {    <-- Генерирует 12-символьную затравку.
      if (err) return cb(err);
      this.salt = salt;    <-- Задает затравку для сохранения.
      bcrypt.hash(this.pass, salt, (err, hash) => {    <-- Генерирует хеш.
        if (err) return cb(err);
        this.pass = hash;    <-- Присваивает хеш для сохранения update().
        cb();
      });
    });
  }
}
```

Вот и всё!

Тестирование логики сохранения пользователя

Чтобы протестировать механизм сохранения пользователей, запустите сервер Redis командой `redis-server` в командной строке. Добавьте код из листинга 6.18 в конец файла **models/user.js**. После этого выполните команду `node models/user.js` в командной строке, чтобы создать тестового пользователя.

Листинг 6.18. Тестирование модели пользователя

```
const User = require('./models/user');
const user = new User({ name: 'Example', pass: 'test' });    <-- Создает нового пользователя.
user.save((err) => {    <-- Сохраняет пользователя.
  if (err) console.error(err);
  console.log('user id %d', user.id);
});
```

Вывод должен сообщать о том, что пользователь был успешно создан: например, `user id 1`. После тестирования модели пользователя удалите код листинга 6.18 из файла **models/user.js**.

В программе redis-cli, входящей в поставку Redis, можно ввести команду `HGETALL` для выборки всех ключей и значений хеша (листинг 6.19).

Листинг 6.19. Вывод информации в интерфейсе командной строки Redis

```
$ redis-cli    <-- Запускает интерфейс командной строки Redis.
redis> get user:ids    <-- Находит идентификатор последнего созданного пользователя.
"1"
redis> hgetall user:1    <-- Читает данные элемента.
 1) "name"    <-- Свойства элемента.
 2) "tobi"
 3) "pass"
 4) "$2a$12$BAOWThTAkNjY7Uht0UdBku4
 5) "age"
 6) "2"
 7) "id"
 8) "4"
 9) "salt"
10) "$2a$12$BAOWThTAkNjY7Uht0UdBku"
redis> quit    <-- Завершает интерфейс командной строки Redis.
```

После определения логики сохранения пользователя необходимо добавить логику загрузки информации.

ДРУГИЕ КОМАНДЫ ПРОГРАММЫ REDIS-CLI

За дополнительной информацией о командах Redis обращайтесь к справочнику команд Redis по адресу http://redis.io/commands.

Чтение данных пользователя

Когда пользователь пытается выполнить вход в веб-приложение, он обычно вводит имя и пароль в форму; введенные данные отправляются приложению для аутентификации. После того как форма входа будет отправлена, необходимо каким-то образом произвести выборку данных пользователя по имени.

Эта логика определяется в листинге 6.20 в методе `User.getByName()`. Функция сначала определяет идентификатор методом `User.getId()`, а затем передает найденный идентификатор методу `User.get()`. Этот метод получает данные хеша этого пользователя от Redis. Добавьте следующие методы в файл models/user.js.

Листинг 6.20. Получение данных пользователя от Redis

```
class User {
  // ...
  static getByName(name, cb) {
    User.getId(name, (err, id) => {    <-- Определяет идентификатор пользователя по имени.
      if (err) return cb(err);
      User.get(id, cb);    <-- Получает данные пользователя по идентификатору.
    });
  }

  static getId(name, cb) {
    db.get(`user:id:${name}`, cb);    <-- Получает идентификатор индексированием по имени.
```

```
  }

  static get(id, cb) {
    db.hgetall(`user:${id}`, (err, user) => {   <-- Получает данные в виде простого объекта.
      if (err) return cb(err);
      cb(null, new User(user));   <-- Преобразует простой объект в новый объект User.
    });
  }
}
```

Если вам потребуется получить данные пользователя, используйте код следующего вида:

```
User.getByName('tobi', (err, user) => {
  console.log(user);
});
```

После получения хешированного пароля можно переходить к аутентификации пользователя.

Аутентификация входа пользователя

Последний компонент, необходимый для аутентификации пользователя, — метод, определенный в листинге 6.21. Он использует определенные ранее функции для получения данных пользователя. Добавьте логику из листинга в файл **models/user.js**.

Листинг 6.21. Аутентификация пользователя

```
static authenticate(name, pass, cb) {
  User.getByName(name, (err, user) => {   <-- Проводит поиск пользователя по имени.
    if (err) return cb(err);
    if (!user.id) return cb();   <-- Пользователь не существует.
    bcrypt.hash(pass, user.salt, (err, hash) => {   <-- Хеширует введенный пароль.
      if (err) return cb(err);
      if (hash == user.pass) return cb(null, user);   <-- Обнаружено совпадение.
      cb();   <-- Неверный пароль.
    });
  });
}
```

Логика аутентификации начинается с выборки данных пользователя по имени. Если пользователь не найден, немедленно активизируется функция обратного вызова. В противном случае сохраненная затравка и введенный пароль хешируются для получения результата, который должен совпадать с хранимым хешем `user.pass`. Если вычисленный хеш не совпадает с хранимым, значит, пользователь ввел неверные данные. При поиске по несуществующему ключу Redis возвращает пустой хеш; вот почему вместо `!user` используется `!user.id`.

Итак, аутентификация заработала; теперь необходимо позаботиться о том, чтобы пользователи могли регистрироваться в приложении.

6.2.6. Регистрация новых пользователей

Чтобы пользователь мог создать учетную запись, а затем выполнить вход, приложение должно реализовать функциональность регистрации и входа.

Для реализации регистрации необходимо решить следующие задачи:

- связать маршруты регистрации и входа с путями URL;
- добавить логику для отображения формы регистрации;
- добавить логику для хранения пользовательских данных, отправленных с формой.

Форма изображена на рис. 6.12.

Эта форма отображается при посещении маршрута `/register` в браузере. Позднее мы создадим аналогичную форму для выполнения входа.

Рис. 6.12. Форма регистрации пользователя

Добавление маршрутов регистрации

Чтобы форма регистрации появилась на экране, необходимо создать маршрут для визуализации формы и вернуть ее браузеру для вывода.

В листинге 6.22 показано, как следует изменить файл **app.js**. Система модулей Node используется для импортирования модуля, определяющего поведение маршрута из каталога маршрутов, а методы HTTP и пути URL связываются с функциями маршрутизации. Так формируется некое подобие контроллера. Как вы увидите, маршруты регистрации существуют как для метода GET, так и для POST.

Листинг 6.22. Добавление маршрутов регистрации

```
...
const register = require('./routes/register');   <-- Включает логику маршрутизации.
...
app.get('/register', register.form);   <-- Добавляет маршруты.
app.post('/register', register.submit);
```

Чтобы определить логику маршрута, создайте в каталоге **routes** пустой файл с именем **register.js**. Начните определение поведения маршрута регистрации с экспортирования следующей функции из **routes/register.js** — маршрута, который выполняет визуализацию шаблона регистрации:

```
exports.form = (req, res) => {
  res.render('register', { title: 'Register' });
};
```

Для определения HTML-разметки формы регистрации маршрут использует шаблон EJS, который будет создан на следующем шаге.

Создание формы регистрации

Чтобы определить HTML-разметку формы регистрации, создайте в каталоге **views** файл с именем **register.ejs**. Разметка HTML/EJS для определения формы приведена в листинге 6.23.

Листинг 6.23. Шаблон представления с формой регистрации

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <% include menu %>    <-- Навигационные ссылки будут добавлены позднее.
    <h1><%= title %></h1>
    <p>Fill in the form below to sign up!</p>
    <% include messages %>    <-- Вывод сообщений будет добавлен позднее.
    <form action='/register' method='post'>
      <p>
        <input type='text' name='user[name]' placeholder='Username' />    <-- Пользователь должен ввести имя пользователя.
      </p>
      <p>
        <input type='password' name='user[pass]'
            placeholder='Password' />    <-- Пользователь должен ввести пароль.
      </p>
      <p>
        <input type='submit' value='Sign Up' />
      </p>
    </form>
  </body>
</html>
```

Обратите внимание на директиву `include messages`, которая включает в себя другой шаблон: **messages.ejs**. Этот шаблон, который будет определен на следующем шаге, используется для взаимодействия с пользователем.

Организация обратной связи с пользователем

Во время регистрации пользователей, как и во многих других частях типичного приложения, возникает необходимость в передаче обратной связи пользователю. Например, пользователь может попытаться зарегистрироваться с именем, которое уже занято кем-то другим. В этом случае необходимо предложить пользователю выбрать другое имя.

В приложении для вывода ошибок будет использоваться шаблон messages.ejs. Многие шаблоны нашего приложения будут включать messages.ejs.

Чтобы создать шаблон сообщений, создайте в каталоге views файл с именем messages.ejs и поместите в него код из приведенного ниже фрагмента. Логика шаблона проверяет, задано ли значение переменной `locals.messages`. Если оно задано, то шаблон перебирает ее содержимое и выводит объекты сообщений. У каждого объекта сообщения имеется свойство `type` (позволяющее при необходимости использовать сообщения для оповещений, не являющихся ошибками) и свойство `string` (текст сообщения). Логика приложения ставит ошибку в очередь для вывода, добавляя ее в массив `res.locals.messages`. После того как сообщения будут выведены, вызывается метод `removeMessages` для очистки очереди сообщений:

```
<% if (locals.messages) { %>
  <% messages.forEach((message) => { %>
    <p class='<%= message.type %>'><%= message.string %></p>
  <% }) %>
  <% removeMessages() %>
<% } %>
```

На рис. 6.13 изображена форма регистрации при выводе сообщения об ошибке.

Рис. 6.13. Вывод сообщений об ошибках на форме

Добавление сообщений в `res.locals.messages` создает простой механизм взаимодействия с пользователем. Однако `res.locals` не сохраняется между перенаправлениями, поэтому необходимо продлить срок их существования, использовав сеансы для их сохранения между запросами.

Сохранение временных сообщений в сеансах

В веб-приложениях применяется стандартный паттерн проектирования PRG (Post/Redirect/Get). В этом паттерне пользователь запрашивает форму, данные формы отправляются в виде запроса HTTP POST, а пользователь перенаправляется на другую веб-страницу. Куда именно перенаправляется пользователь — зависит от результата проверки данных приложением. Если данные формы признаются действительными, то пользователь перенаправляется на новую веб-страницу. Паттерн PRG прежде всего используется для предотвращения повторной отправки форм.

В Express при перенаправлении пользователя содержимое `res.locals` сбрасывается. Если вы сохраняете сообщения для пользователя в `res.locals`, сообщения будут потеряны до того, как они появятся на экране. Проблему можно обойти сохранением сообщений в сеансовой переменной. После этого сообщения могут быть выведены на итоговой странице перенаправления.

Чтобы адаптировать возможность постановки сообщений в очередь к сеансовой переменной, необходимо добавить в приложение модуль. Создайте файл с именем ./middleware/messages.js и добавьте в него следующий код:

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

function message(req) {
  return (msg, type) => {
    type = type || 'info';
    let sess = req.session;
    sess.messages = sess.messages || [];
    sess.messages.push({ type: type, string: msg });
  };
};
```

Функция `res.message` предоставляет возможность добавления в сеансовую переменную сообщений от любых запросов Express. Объект `express.response` представляет собой прототип, который используется Express для объектов ответов. Добавление свойств в объект означает, что они станут доступными для всех промежуточных компонентов и маршрутов. В предшествующем фрагменте `express.response` присваивается переменной с именем `res`, чтобы упростить добавление свойств к объекту и сделать код более понятным.

Для реализации этой возможности потребуется поддержка сеансов. Для этого мы воспользуемся Express-совместимым модулем: официально поддерживаемым пакетом express-session. Установите его командой `npm install --save express-session`, а затем добавьте в app.js:

```
const session = require('express-session');
...
app.use(session({
  secret: 'secret',
  resave: false, saveUninitialized: true
}));
```

Промежуточный компонент лучше разместить после вставки промежуточного компонента для работы с cookie (приблизительно около строки 26).

Чтобы еще больше упростить добавление сообщений, используйте код из следующего фрагмента. Функция `res.error` позволяет легко добавить в очередь сообщений сообщение типа `error`. Используйте функцию `res.message`, которая ранее была определена в модуле:

```
res.error =  msg => this.message(msg, 'error');
```

Остается сделать последний шаг: предоставить шаблонам доступ к этим сообщениям для вывода на экран. Если этого не сделать, придется передавать массив `req.session.messages` каждому вызову `res.render()` в приложении; конечно, такое решение вряд ли можно назвать идеальным.

Для решения проблемы мы создадим промежуточный компонент, который для каждого запроса будет заполнять массив `res.locals.messages` данными из массива `res.session.messages`, фактически открывая доступ к сообщениям для каждого визуализируемого шаблона. Пока что файл ./lib/messages.js расширяет прототип ответа, но ничего не экспортирует. Добавление в файл следующего фрагмента экспортирует необходимый вам промежуточный компонент:

```
module.exports = (req, res, next) => {
  res.message = message(req);
  res.error = (msg) => {
    return res.message(msg, 'error');
  };
  res.locals.messages = req.session.messages || [];
  res.locals.removeMessages = () => {
    req.session.messages = [];
  };
  next();
};
```

Сначала определяется переменная шаблона `messages` для хранения сообщений сеанса; это массив, который может существовать, а может и не существовать после выполнения предыдущего запроса (не забывайте, что эти сообщения сохраняются между сеансами). Затем нужно придумать, как удалять сообщения из сеанса; в противном случае они будут неограниченно накапливаться.

Остается лишь интегрировать этот новый механизм функцией `require()` в файл app.js. Этот компонент должен монтироваться после компонента сеанса, поскольку для него должно быть определено свойство `req.session`. Обратите внимание: поскольку этот промежуточный компонент спроектирован так, что он не получает параметры и не возвращает вторую функцию, можно использовать вызов `app.use(messages)` вместо `app.use(messages())`. Для надежности в промежуточных компонентах от сторонних разработчиков лучше использовать `app.use(messages())` вне зависимости от того, получает компонент параметры или нет:

```
...
const register = require('./routes/register');
const messages = require('./middleware/messages');
...
app.use(express.methodOverride());
app.use(express.cookieParser());
   app.use(session({
     secret: 'secret',
     resave: false,
     saveUninitialized: true
   }));
app.use(messages);
...
```

Теперь к переменной `messages` и функции `removeMessages()` можно обратиться из любого представления, поэтому файл messages.ejs при включении в любой шаблон должен работать правильно.

Разобравшись с выводом на экран формы регистрации и механизмом передачи обратной связи пользователю, перейдем к обработке регистрационной информации, отправленной формой.

Регистрация пользователя

Мы должны создать маршрутную функцию для обработки HTTP-запросов POST к /register. Эта функция будет называться `submit`.

После отправки данных формы промежуточный компонент `bodyParser()` заполняет отправленными данными свойство `req.body`. В форме регистрации используется объектный синтаксис `user[name]`, который после разбора преобразуется в свойство `req.body.user.name`. Аналогичным образом для поля ввода пароля используется свойство `req.body.user.pass`.

Осталось добавить небольшой фрагмент кода к маршруту отправки, который должен обеспечивать проверку данных (например, проверку того, что имя пользователя еще не занято), и сохранить нового пользователя, как продемонстрировано в листинге 6.24.

После завершения регистрации значение `user.id` сохраняется в сеансе пользователя; позднее вы можете проверить его и убедиться в том, что пользователь прошел аутентификацию. Если проверка проходит неудачно, сообщение предоставляется шаблонам в виде переменной `messages` через массив `res.locals.messages`, а пользователь перенаправляется обратно к форме регистрации.

Чтобы реализовать эту функциональность, добавьте код из листинга 6.24 в файл routes/register.js.

Листинг 6.24. Создание пользователя по отправленным данным

```
const User = require('../models/user');
...
exports.submit = (req, res, next) => {
```

```
  const data = req.body.user;
  User.getByName(data.name, (err, user) => {   <-- Проверяет имя пользователя на уникальность.
    if (err) return next(err);   <-- Передает ошибки подключения к базе данных и другие ошибки.
    // redis использует значение по умолчанию
    if (user.id) {   <-- Имя пользователя уже занято.
      res.error('Username already taken!');
      res.redirect('back');
    } else {
      user = new User({   <-- Создает пользователя на основе данных POST.
        name: data.name,
        pass: data.pass
      });
      user.save((err) => {   <-- Сохраняет нового пользователя.
        if (err) return next(err);
        req.session.uid = user.id;   <-- Сохраняет uid для аутентификации.
        res.redirect('/');   <-- Перенаправляет на страницу со списком.
      });
    }
  });
};
```

Теперь вы сможете запустить приложение, перейти по маршруту /register и зарегистрировать пользователя. Следующее, что потребуется, — механизм возвращения зарегистрированных пользователей для аутентификации через форму `/login`.

6.2.7. Вход для зарегистрированных пользователей

Добавить функциональность входа еще 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 будут добавлены:

- логика маршрута для вывода на экран формы входа;
- логика аутентификации отправленных из формы пользовательских данных.

Форма входа выглядит так, как показано на рис. 6.14.

Рис. 6.14. Форма входа

Для начала изменим файл app.js с включением маршрутов входа и назначением маршрутных путей:

```
...
const login = require('./routes/login');
...
app.get('/login', login.form);
app.post('/login', login.submit);
app.get('/logout', login.logout);
...
```

Затем добавляется функциональность для отображения формы входа на экране.

Вывод на экран формы входа

Реализация формы входа начинается с создания файла для хранения маршрутов входа и выхода из приложения: routes/login.js. Маршрутная логика вывода на экран формы входа практически не отличается от логики вывода формы регистрации; различаются только имя шаблона и заголовок страницы:

```
exports.form = (req, res) => {
  res.render('login', { title: 'Login' });
};
```

Форма входа EJS-шаблона, которая определена в файле ./views/login.ejs (листинг 6.25), также очень похожа на форму из файла register.ejs. Различаются только текст инструкций и маршрут, по которому отправляются данные.

Листинг 6.25. 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>
        <input type='text' name='user[name]' placeholder='Username' />
      </p>
      <p>
        <input type='password' name='user[pass]'
               placeholder='Password' />
      </p>
      <p>
        <input type='submit' value='Login' />
```

Пользователь должен ввести имя пользователя.

Пользователь должен ввести пароль.

```
      </p>
    </form>
  </body>
</html>
```

После того как вы добавили маршрут и шаблон для вывода формы входа на экран, в приложение нужно добавить логику обработки попыток входа.

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 верны, идентификатор пользователя присваивается сеансовой переменной, после чего пользователь перенаправляется на домашнюю страницу. Добавьте эту логику (листинг 6.26) в файл routes/login.js.

Листинг 6.26. Маршрут для обработки попыток входа

```
const User = require('../models/user');
...
exports.submit = (req, res, next) => {
  const data = req.body.user;
  User.authenticate(data.name, data.pass, (err, user) => {   <-- Проверяет учетные данные.
    if (err) return next(err);   <-- Делегирует обработку ошибок.
    if (user) {   <-- Обрабатывает пользователя с действительными учетными данными.
      req.session.uid = user.id;   <-- Сохраняет идентификатор пользователя для аутентификации.
      res.redirect('/');   <-- Перенаправляет к списку.
    } else {
      res.error('Sorry! invalid credentials. ');   <-- Предоставляет сообщение об ошибке.
      res.redirect('back');   <-- Перенаправляет обратно к форме входа.
    }
  });
};
```

Если пользователь проходит аутентификацию методом `User.authenticate()`, то значение свойства `req.session.uid` присваивается по тому же принципу, как и в случае с POST-маршрутом `/register`: это значение сохраняется в сеансе и может применяться в дальнейшем для выборки объекта `User` или других данных, связанных с пользователем. Если соответствие не обнаруживается, устанавливается признак ошибки, а форма снова выводится на экран.

Некоторые пользователи предпочитают явно завершать работу с приложением, поэтому нужно создать соответствующую ссылку в приложении. В файле app.js маршрут создается следующим образом:

```
const login = require('./routes/login');
…
app.get('/logout', login.logout);
```

Затем в файле ./routes/login.js следующая функция удалит сеанс, обнаруженный промежуточным компонентом `session()`, что приведет к установке сеанса для последующих запросов:

```
exports.logout = (req, res) => {
  req.session.destroy((err) => {
    if (err) throw err;
    res.redirect('/');
  })
};
```

После создания страниц регистрации и входа 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. Через это меню пользователь сможет входить в систему, регистрироваться, отправлять записи из формы и выходить из системы. На рис. 6.15 представлено меню для анонимного пользователя.

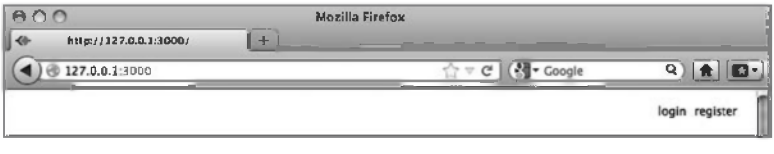


Рис. 6.15. Меню входа и регистрации обеспечивает доступ к созданным нами формам

Когда пользователь проходит аутентификацию, 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. 6.16.

Рис. 6.16. Меню для пользователя, прошедшего аутентификацию

Каждый созданный вами EJS-шаблон, представляющий страницу приложения, содержит код `<% include menu %>`, расположенный после тега `<body>`. Этот код подключает шаблон ./views/menu.ejs (листинг 6.27).

Листинг 6.27. Шаблон меню для анонимных пользователей и пользователей, прошедших аутентификацию

```
<% if (locals.user) { %>
  <div id='menu'>  <-- Меню для пользователей, выполнивших вход.
    <span class='name'><%= user.name %></span>
    <a href='/post'>post</a>
    <a href='/logout'>logout</a>
  </div>
<% } else { %>
  <div id='menu'>  <-- Меню для анонимных пользователей.
    <a href='/login'>login</a>
    <a href='/register'>register</a>
  </div>
<% } %>
```

Предполагается, что, раз переменная `user` доступна для шаблона, пользователь прошел аутентификацию, поскольку в противном случае эта переменная не была бы определена. Если переменная присутствует, имя пользователя можно вывести на экран вместе со ссылками для ввода данных и выхода из приложения. Если же сайт посещает анонимный пользователь, выводятся ссылки входа и регистрации.

Возникает резонный вопрос — откуда берется локальная переменная `user`, если мы ее еще не создали? В следующем разделе будет написан код, реализующий для каждого запроса загрузку данных пользователя, вошедшего в систему, и открывающий шаблонам доступ к этим данным.

6.2.8. Промежуточный компонент для загрузки пользовательских данных

Одна из типичных задач веб-приложений — загрузка из базы данных информации о пользователе, которая обычно предоставляется в виде JavaScript-объекта. Наличие подобных данных облегчает взаимодействие с пользователем. В приложении этой главы для каждого запроса будут загружаться пользовательские данные с использованием промежуточного компонента.

Сценарий размещается в файле **./lib/middleware/user.js**, то есть модель `User` должна находиться в папке более высокого уровня (**./lib**). Сначала экспортируется функция промежуточного компонента, затем сеанс проверяется на предмет наличия идентификатора пользователя. Если идентификатор присутствует, значит, пользователь прошел аутентификацию, поэтому приложение может безопасно выполнить выборку данных из Redis.

Поскольку Node является однопоточной платформой, локальное хранилище программных потоков здесь отсутствует. В случае с HTTP-сервером переменные запроса и ответа являются единственными доступными контекстными объектами. Высокоуровневые фреймворки могут пользоваться средствами Node и предоставлять

дополнительные объекты для хранения данных аутентифицированных пользователей, однако в Express было принято решение придерживаться исходных объектов, предлагаемых платформой Node. В результате контекстные данные обычно хранятся в объекте запроса, как показано в листинге 6.28, где пользовательские данные сохраняются в свойстве `req.user`; последующие программные промежуточные компоненты и маршруты могут получать доступ к пользовательским данным через это свойство.

Возможно, вас интересует, для чего нужно присваивание `res.locals.user`. `res.locals` — объект уровня запроса, который Express предоставляет для получения доступа к данным из шаблонов, что очень напоминает `app.locals`. Это тоже некая функция, которая может применяться для объединения существующих объектов.

Листинг 6.28. Промежуточный компонент для загрузки данных пользователя, выполнившего вход

```
const User = require('../models/user');
module.exports = (req, res, next) => {
  const uid = req.session.uid;
  if (!uid) return next();
  User.get(uid, (err, user) => {
    if (err) return next(err);
    req.user = res.locals.user = user;
    next();
  });
};
```

Получает идентификатор пользователя, выполнившего вход, из сеансовых данных.

Получает данные пользователя, выполнившего вход, от Redis.

Предоставляет доступ к данным пользователя объекту запроса.

Чтобы воспользоваться новым промежуточным компонентом, сначала удалите из файла app.js все строки с текстом «user». Затем, как обычно, включите этот модуль вызовом `require()` и передайте его методу `app.use()`. В нашем приложении переменная `user` используется перед маршрутизатором, поэтому свойство `req.user` будет доступно только для маршрутов и промежуточных компонентов, следующих в коде после `user`. Если вы используете промежуточный компонент, который загружает данные (как в нашем случае), компонент `express.static` следует разместить перед ним, иначе каждый раз при предоставлении статического файла будет происходить лишнее обращение к базе данных для выборки пользовательских данных.

В листинге 6.29 показано, как включить этот компонент в файл app.js.

Листинг 6.29. Включение компонента, загружающего пользовательские данные

```
const user = require('./middleware/user');
...
app.use(express.session());
app.use(express.static(__dirname + '/public'));
app.use(user);
app.use(messages);
app.use(app.router);
...
```

Добавляет компонент в приложение.

Если вы снова запустите приложение и откроете в браузере страницу `/login` или `/register`, на экране появится меню. Чтобы применить к меню стилевое оформление, добавьте в файл public/stylesheets/style.css разметку CSS из листинга 6.30.

Листинг 6.30. Разметка CSS, добавляемая в файл style.css для оформления меню

```
#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 зарегистрироваться в качестве пользователя. После этого на экране должно появиться меню для пользователя, прошедшего аутентификацию, в котором есть ссылка `Post`.

В следующем разделе вы узнаете, как создать открытый REST API для приложения.

6.2.9. Создание открытого REST API

В этом разделе для приложения будет реализован REST-совместимый открытый API, чтобы сторонние приложения могли обращаться к данным и добавлять новые записи. Архитектурный стиль REST обеспечивает возможность чтения и изменения данных приложения с использованием «глаголов» и «существительных», представленных методами HTTP и URL-адресами соответственно. Запрос REST обычно возвращает данные в формате, удобном для машинной обработки, например JSON или XML.

Реализация API состоит из следующих этапов:

- проектирование API, с помощью которого пользователи смогут отображать, выводить список, удалять и публиковать записи;
- добавление базовой аутентификации;
- реализация маршрутизации;
- предоставление ответов в формате JSON и XML.

Для аутентификации и сертификации запросов API могут применяться разные методы, но реализация более сложных решений выходит за рамки темы книги.

Чтобы продемонстрировать, как аутентификация интегрируется в приложение, мы воспользуемся пакетом basic-auth.

Проектирование API

Прежде чем браться за реализацию, желательно спланировать необходимые маршруты. В нашем приложении REST-совместимый API будет снабжаться префиксом `/api`, но это всего лишь решение из области проектирования, которое вы можете изменить — например, использовать поддомен вида *http://api.myapplication.com*.

Следующий фрагмент демонстрирует, почему функции обратного вызова стоит выделить в отдельные модули Node (вместо определения их во встроенном формате с вызовами `app.`*`метод`*`()`). Простой список маршрутов дает четкое представление о том, что вы и ваша группа реализовали и где находятся реализации обратных вызовов:

```
app.get('/api/user/:id', api.user);
app.get('/api/entries/:page?', api.entries);
app.post('/api/entry', api.add);
```

Добавление базовой аутентификации

Как упоминалось ранее, существует много возможных подходов к защите API и ограничений, выходящих за рамки темы книги. В этом разделе мы продемонстрируем процесс на примере базовой аутентификации.

Процесс будет абстрагироваться промежуточным компонентом `api.auth`, потому что реализация будет находиться в модуле ./routes/api.js (вскоре мы создадим этот модуль). Метод `app.use()` может получать путь, который в Express называется *точкой монтирования*. С этой точкой монтирования для любых путей, начинающихся с `/api`, и любой команды HTTP, будет активизироваться промежуточный компонент.

Строка `app.use('/api', api.auth)`, как показано в следующем фрагменте, должна располагаться до промежуточного компонента, загружающего данные пользователя. Это нужно для того, чтобы позднее вы могли изменить компонент загрузки данных, чтобы загружать данные пользователей, прошедших аутентификацию:

```
...
const api = require('./routes/api');
...
app.use('/api', api.auth);
app.use(user);
...
```

Для выполнения базовой аутентификации установите модуль basic-auth командой `npm install --save basic-auth`. Затем создайте файл ./routes/api.js и включите как Express, так и модель пользователя, как показано в следующем фрагменте. Пакет

basic-auth получает функцию для выполнения аутентификации с сигнатурой *(имя_пользователя, пароль, обратный_вызов)*. Метод `User.authenticate` идеально подходит для этой цели:

```
const auth = require('basic-auth');
const express = require('express');
const User = require('../models/user');

exports.auth = (req, res, next) => {
  const { name, pass } = auth(req);
  User.authenticate(name, pass, (err, user) => {
    if (user) req.remoteUser = user;
    next(err);
  });
};
```

Аутентификация готова к работе. Перейдем к реализации маршрутов API.

Реализация маршрутизации

Первый маршрут, который мы реализуем, — `GET /api/user/:id`. Логика этого маршрута должна сначала получить пользователя по идентификатору *id* и ответить кодом 404 Not Found, если пользователь не существует. Если пользователь существует, то данные пользователя будут переданы `res.send()` для сериализации, а приложение ответит представлением этих данных в формате JSON. Добавьте логику из следующего фрагмента в файл routes/api.js:

```
exports.user = (req, res, next) => {
  User.get(req.params.id, (err, user) => {
    if (err) return next(err);
    if (!user.id) return res.sendStatus(404);
    res.json(user);
  });
};
```

Затем добавьте следующий путь маршрута в файл app.js:

```
app.get('/api/user/:id', api.user);
```

Все готово к тестированию.

Тестирование выборки данных пользователя

Запустите приложение и протестируйте его в программе командной строки cURL. Тестирование REST-аутентификации приложения продемонстрировано в следующем фрагменте. Учетные данные передаются в URL *tobi:ferret*, по которым cURL строит поле заголовка Authorization:

```
$ curl http://tobi:ferret@127.0.0.1:3000/api/user/1 -v
```

В листинге 6.31 представлен результат успешного тестирования. Для выполнения аналогичного теста необходимо знать идентификатор пользователя. Если значение 1 не работает, а пользователь зарегистрирован, попробуйте использовать redis-cli и команду `GET user:ids`.

Листинг 6.31. Тестовый вывод

```
* 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'
> GET /api/user/1 HTTP/1.1    <-------  Отправленные заголовки HTTP.
> 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: */*
>
< HTTP/1.1 200 OK    <-------  Полученные заголовки HTTP.
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8    <-------  Полученные данные JSON.
< Content-Length: 150
< Connection: keep-alive
<
{"id":"1","name":"tobi"}
```

Удаление конфиденциальной информации пользователя

Как видно из ответа JSON, в ответе передается как пароль пользователя, так и затравка. Чтобы изменить ситуацию, реализуйте метод `.toJSON()` в объекте `User` из файла models/user.js:

```
class User {
  // ...
  toJSON() {
    return {
      id: this.id,
      name: this.name
    };
  }
```

Если метод `.toJSON` существует в объекте, он будет использоваться вызовами `JSON.stringify` для получения формата JSON. Если бы приведенный выше запрос cURL был выдан повторно, на этот раз были бы получены только свойства `id` и `name`:

```
{
  "id": "1",
  "name": "tobi"
}
```

На следующем шаге в API необходимо добавить возможность создания записей.

Добавление записей

Процессы добавления записи с формы HTML и через API практически идентичны, поэтому в данном случае стоит снова воспользоваться уже реализованной логикой маршрута `entries.submit()`.

Впрочем, при добавлении записей в логике маршрута хранится имя пользователя, а запись добавляется к прочей информации. По этой причине необходимо модифицировать компонент загрузки данных пользователя для заполнения информации, загруженной промежуточным компонентом `basic-auth`. Промежуточный компонент `basic-auth` возвращает эту информацию, которую можно присвоить `req.remoteUser`. Соответствующая проверка в компоненте загрузки данных пользователя выполняется достаточно прямолинейно; измените определение `module.exports` в файле **middleware/user.js**, чтобы компонент загрузки пользователей работал с API:

```
...
module.exports = (req, res, next) => {
  if (req.remoteUser) {
    res.locals.user = req.remoteUser;
  }
  const uid = req.session.uid;
  if (!uid) return next();
  User.get(uid, (err, user) => {
    if (err) return next(err);
    req.user = res.locals.user = user;
    next();
  });
};
```

После этого вы сможете добавлять записи через API.

Впрочем, необходимо внести еще одно изменение — ответ, совместимый с API, вместо перенаправления на домашнюю страницу приложения. Для добавления этой функциональности приведите вызов `entry.save` в файле **routes/entries.js** к следующему виду:

```
...
  entry.save(err => {
    if (err) return next(err);
    if (req.remoteUser) {
      res.json({ message: 'Entry added.' });
    } else {
      res.redirect('/');
    }
  });
...
```

Наконец, чтобы активировать API добавления записей в вашем приложении, добавьте содержимое следующего фрагмента в раздел `routing` файла **app.js**:

```
app.post('/api/entry', entries.submit);
```

Следующая команда cURL позволяет протестировать добавление записи через API. В этом случае данные заголовка и тела сообщения передаются с именами полей, которые используются в форме HTML:

```
$ curl -X POST -d "entry[title]='Ho ho ho'&entry[body]='Santa loves you'"
      http://tobi:ferret@127.0.0.1:3000/api/entry
```

После способности создания записей необходимо добавить возможность получения данных записей.

Добавление поддержки списка записей

Следующим будет реализован маршрут API `GET /api/entries/:page?`. Реализация маршрута почти не отличается от маршрута вывода в файле **./routes/entries.js**. Также потребуется промежуточный компонент страничного вывода — `page()` в следующих фрагментах. Реализация `page()` будет добавлена ниже.

Так как маршрутная логика будет работать с записями, в начало файла **routes/api.js** следует включить модель `Entry`:

```
const Entry = require('../models/entry');
```

Затем добавьте следующий фрагмент в **app.js**:

```
const Entry = require('./models/entry');
...
app.get('/api/entries/:page?', page(Entry.count), api.entries);
```

После чего добавьте маршрутную логику из следующего фрагмента в файл **routes/api.js**. Различия между этой логикой и аналогичной логикой из **routes/entries.js** отражают тот факт, что вместо визуализации шаблона строится разметка JSON:

```
exports.entries = (req, res, next) => {
  const page = req.page;
  Entry.getRange(page.from, page.to, (err, entries) => {
    if (err) return next(err);
    res.json(entries);
  });
};
```

Реализация промежуточного компонента страничного вывода

Для разбиения на страницы будет использоваться строка запроса `?page=`*N* (где N — текущая страница). Добавьте функцию (листинг 6.32) промежуточного компонента в файл **./middleware/page.js**.

Листинг 6.32. Промежуточный компонент для разбиения на страницы

```
module.exports = (cb, perpage) => {
  perpage = perpage || 10;    <-- По умолчанию 10 записей на страницу.
  return (req, res, next) => {    <-- Возвращает функцию промежуточного компонента.
    let page = Math.max(
```

```
      parseInt(req.params.page || '1', 10)
      1
    ) - 1;    <-------  Разбирает параметр page как десятичное целое число.
    cb((err, total) => {    <-------  Активизирует переданную функцию.
      if (err) return next(err);    <-------  Делегирует обработку ошибок.
      req.page = res.locals.page = {    <-------  Сохраняет свойства page для будущих обращений.
        number: page,
        perpage: perpage,
        from: page * perpage,
        to: page * perpage + perpage - 1,
        total: total,
        count: Math.ceil(total / perpage)
      };
      next();    <-------  Передает управление следующему промежуточного компоненту.
    });
  }
};
```

Компонент получает значение, присвоенное *?page=N* (например, `?page=1`). Затем он получает общее количество результатов и предоставляет объект `page` с заранее вычисленными значениями любым представлениям, которые позднее могут визуализироваться. Эти значения вычисляются вне шаблона, чтобы шаблон был более четким и содержал меньше логики.

Тестирование маршрута entries

Следующая команда cURL запрашивает данные записей из API:

```
$ curl http://tobi:ferret@127.0.0.1:3000/api/entries
```

Эта команда cURL должна вывести данные JSON следующего вида:

```
[
  {
    "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."
  },
...
```

Разобравшись с базовой реализацией API, перейдем к тому, как API может поддерживать множественные форматы ответов.

6.2.10. Согласование контента

Согласование контента позволяет клиенту указать форматы, которые он готов принять и которые он считает предпочтительными. В этом разделе мы предоставим

контент API в форматах JSON и XML, чтобы пользователи API могли решить, что им нужно.

HTTP поддерживает механизм согласования контента через поле заголовка `Accept`. Например, клиент, предпочитающий HTML, но готовый принять простой текст, может задать следующий заголовок запроса:

```
Accept: text/plain; q=0.5, text/html
```

Значение *qvalue* (`q=0.5` в данном примере) указывает, что, хотя формат `text/html` указан на втором месте, он на 50% предпочтительнее формата `text/plain`. Express разбирает эту информацию и предоставляет нормализованный массив `req.accepted`:

```
[{ value: 'text/html', quality: 1 },
 { value: 'text/plain', quality: 0.5 }]
```

Express также предоставляет метод `res.format()`, который получает массив типов MIME и обратных вызовов. Express определяет, какую информацию хочет получить клиент и какую информацию вы готовы предоставить, после чего вызывает соответствующую функцию обратного вызова.

Реализация согласования контента

Реализация согласования контента для маршрута `GET /api/entries` из файла routes/api.js может выглядеть приблизительно так, как показано в листинге 6.33. JSON поддерживается так же, как и прежде, — записи сериализуются в формат JSON вызовом `res.send()`. Обратный вызов XML перебирает записи и записывает данные в сокет в процессе перебора. Учтите, что явно задавать значение `Content-Type` не нужно; `res.format()` присваивает нужное значение автоматически.

Листинг 6.33. Реализация согласования контента

```
exports.entries = (req, res, next) => {
  const page = req.page;
  Entry.getRange(page.from, page.to, (err, entries) => {   <-- Получает данные записей.
    if (err) return next(err);
    res.format({   <-- Отвечает по-разному в зависимости от значения заголовка Accept.
      'application/json': () => {   Ответ JSON.
        res.send(entries);
      },
      'application/xml': () => {   <-- Ответ XML.
        res.write('<entries>\n');
        entries.forEach((entry) => {
          res.write(```
            <entry>
              <title>${entry.title}</title>
              <body>${entry.body}</body>
              <username>${entry.username}</username>
            </entry>
            ```
          );
```
```

```
        });
        res.end('</entries>');
      }
    })
  });
};
```

Если вы задали обратный вызов для формата ответа по умолчанию, он будет выполнен в том случае, если пользователь не запросил формат, который вы явно предоставляете.

Метод `res.format()` также получает имя, которое соответствует заданному типу MIME. Например, вместо `application/json` и `application/xml` могут использоваться `json` и `xml`:

```
...
res.format({
  json: () => {
    res.send(entries);
  },
  xml: () => {
    res.write('<entries>\n');
    entries.forEach((entry) => {
      res.write(```
        <entry>
             <title>${entry.title}</title>
          <body>${entry.body}</body>
          <username>${entry.username}</username>
        </entry>
        ```
      );
    });
    res.end('</entries>');
  }
})
...
```

### Ответ в формате XML

Пожалуй, написание специализированной логики в маршруте для выдачи ответа в формате XML — не самое элегантное решение. Посмотрим, как воспользоваться системой представлений для решения этой проблемы.

Создайте шаблон с именем **./views/entries/xml.ejs** со следующим кодом EJS, перебирающим записи для генерирования тегов `&lt;entry&gt;` (листинг 6.34).

**Листинг 6.34.** Использование шаблона EJS для генерирования разметки XML

```
<entries>
<% entries.forEach(entry => { %>   <------- Перебирает все записи.
  <entry>
    <title><%= entry.title %></title>   <------- Выводит поля.
```
```

```
    <body><%= entry.body %></body>
    <username><%= entry.username %></username>
  </entry>
<% }) %>
</entries>
```

Обратный вызов XML теперь можно заменить одним вызовом `res.render()` с передачей массива `entries`:

```
...
  xml: () => {
    res.render('entries/xml', { entries: entries });
  }
})
...
```

Теперь все готово для тестирования XML-версии API. Введите следующую команду в командной строке, чтобы просмотреть вывод в формате XML:

```
curl -i -H 'Accept: application/xml'
   http://tobi:ferret@127.0.0.1:3000/api/entries
```

6.3. Заключение

- Connect — фреймворк HTTP, который позволяет строить цепочки промежуточных компонентов до и после обработки запросов.
- Промежуточные компоненты Connect представляют собой функции, которые получают объекты запроса и ответа Node, а также функцию, которая вызывает следующий компонент, и необязательный объект ошибки.
- Веб-приложения Express также строятся из промежуточных компонентов.
- В Express можно строить REST API, используя команды HTTP для определения маршрутов.
- Маршруты Express могут отвечать данными JSON, HTML или в других форматах данных.
- Express содержит простой API шаблонизации с поддержкой многих разных ядер.

7 Шаблонизация веб-приложений

В главах 3 и 6 рассматривались основы использования шаблонов для создания представлений в приложениях Express. В этой главе, полностью посвященной шаблонам, мы рассмотрим три популярных шаблонизатора, а также узнаем, как шаблоны помогают «очистить» код веб-приложений за счет отделения программной логики от разметки визуализации.

Если вы знакомы с шаблонизацией и паттерном MVC (Model-View-Controller — Модель-Представление-Контроллер), можете сразу переходить к разделу 7.2, с которого начинается подробное рассмотрение шаблонизаторов, включая EJS, Hogan и Pug. Если же шаблонизация вам в новинку, продолжайте читать — в следующих нескольких разделах изложены концептуальные основы работы с шаблонами.

7.1. Поддержка чистоты кода путем шаблонизации

С паттерном MVC можно разрабатывать традиционные приложения как на базе Node, так и практически любой другой веб-технологии. 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 их данные *представлению* (view), которое осуществляет окончательное форматирование данных для конечного пользователя. MVC-представления часто реализуются с помощью одного из языков шаблонизации. Если в приложении используется шаблонизация, представление передает *шаблонизатору* (template engine) значения, возвращенные моделью, и указывает файл шаблона, определяющий способ отображения этих значений.

На рис. 7.1 показано, как логика шаблонизации вписывается в общую архитектуру MVC-приложения. Файлы шаблонов обычно содержат заполнители для значений

Рис. 7.1. Последовательность операций при работе MVC-приложения и его взаимодействие с уровнем шаблона

приложений, а также фрагменты HTML-, CSS- и иногда клиентского JavaScript-кода, предназначенного для реализации динамического поведения (вывода на экран сторонних виджетов вроде кнопки 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 с программной логикой, разработчикам клиентских и серверных приложений эти файлы помогают организовать разделение труда.

В этом разделе мы займемся визуализацией разметки HTML, выполняемой как с помощью шаблонов, так и без них, чтобы вы смогли почувствовать разницу. Однако сначала давайте разберем практический пример использования шаблонов.

7.1.1. Шаблонизация в действии

В качестве простого примера применения шаблонов мы рассмотрим проблему элегантного HTML-вывода из простого приложения для блога. Каждая запись блога включает в себя заголовок, дату создания и текст. В окне браузера блог будет выглядеть так, как показано на рис. 7.2.

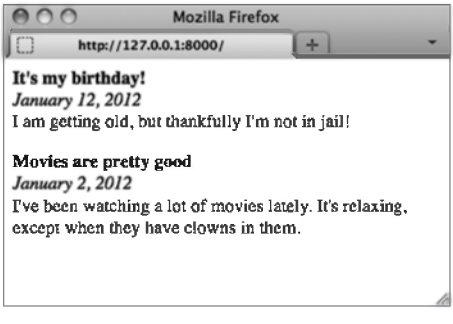


Рис. 7.2. Пример вывода приложения блога в браузере

Записи блога читаются из текстового файла, отформатированного так, как фрагмент из файла entries.txt (листинг 7.1). Символы --- показывают, где заканчивается одна запись и начинается другая.

Листинг 7.1. Текстовый файл с записями в блоге

```
title: It's my birthday!
date: January 12, 2016
I am getting old, but thankfully I'm not in jail!
---
title: Movies are pretty good
date: January 2, 2016
I've been watching a lot of movies lately. It's relaxing,
except when they have clowns in them.
```

Код приложения в файле **blog.js** начинается с включения необходимых модулей и чтения записей блога, как показано в листинге 7.2.

Листинг 7.2. Логика разбора записей блога из текстового файла

```
const fs = require('fs');
const http = require('http');
function getEntries() {
  const entries = [];
  let entriesRaw = fs.readFileSync('./entries.txt', 'utf8');
  entriesRaw = entriesRaw.split('---');
  entriesRaw.map((entryRaw) => {
    const entry = {};
    const lines = entryRaw.split('\n');
    lines.map((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;
}
const entries = getEntries();
console.log(entries);
```

Функция для чтения и разбора текста записи.

Читает данные записи из файла.

Разбирает текст на отдельные записи блога.

Разбирает текст записи на строки.

Разбирает строки на свойства entry.

Следующий код, добавленный в приложение, определяет сервер HTTP. Когда сервер получает запрос HTTP, он возвращает страницу, содержащую все записи блога. Эта страница визуализируется функцией `blogPage`, которую мы определим чуть позже:

```
const server = http.createServer((req, res) => {
  const output = blogPage(entries);
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(output);
});
server.listen(8000);
```

Теперь нужно определить функцию `blogPage`, которая преобразует записи блога в HTML-страницу, отправляемую 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.

7.1.2. Визуализация HTML без шаблона

Приложение для блога могло бы непосредственно выводить HTML-код на экран, но смешивание HTML-разметки с программной логикой приложения привело бы к хаосу. В листинге 7.3 функция `blogPage` демонстрирует вывод на экран записей блога без помощи шаблона.

Листинг 7.3 Шаблонизатор отделяет подробности визуализации от программной логики

```
function blogPage(entries) {
  let 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(entry => {
    output += `    <-- Разметка HTML слишком сильно смешивается с программной логикой.
      <div class="entry_title">${entry.title}</div>
      <div class="entry_date">${entry.date}</div>
      <div class="entry_body">${entry.body}</div>
    `;
  });
  output += '</body></html>';
  return output;
}
```

Обратите внимание: весь контент, относящийся к оформлению вывода, CSS-определения и HTML-разметка значительно увеличивают число строк в коде приложения.

Визуализация HTML с помощью шаблона

Визуализация разметки HTML с применением шаблона позволяет убрать из прикладной логики HTML-разметку, что делает код приложения существенно чище.

Для тестирования примеров, приведенных в этом разделе, нужно установить в папку приложения модуль для работы с EJS (Embedded JavaScript). Для этого в командой строке введите следующую команду:

```
npm install ejs
```

Следующий фрагмент загружает шаблон из файла, а затем определяет новую версию функции `blogPage`, которая теперь использует шаблонизатор EJS (о нем мы поговорим в разделе 7.2):

```
const fs = require('fs');
const ejs = require('ejs');
const template = fs.readFileSync('./templatess/blog_page.ejs', 'utf8');
function blogPage(entries) {
  const values = { entries };
  return ejs.render(template, values);
}
```

Полный листинг находится в архиве кода, прилагаемом к книге, в каталоге ch07-templates/listing7_4/. Файл EJS-шаблона содержит HTML-разметку (отделенную от прикладной логики), а также заполнители, которые указывают, где должны располагаться переданные шаблонизатору данные. В листинге 7.4 приведен файл EJS-шаблона, применяемый для вывода записей блога на экран, включающий HTML-разметку и заполнители.

Листинг 7.4. Шаблон EJS для вывода 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>
    <% entries.map(entry => { %>
      <div class="entry_title"><%= entry.title %></div>
      <div class="entry_date"><%= entry.date %></div>
      <div class="entry_body"><%= entry.body %></div>
    <% }); %>
  </body>
</html>
```

Заполнитель, перебирающий записи блога.

Заполнители для фрагментов данных в каждой записи.

В модулях, разработанных сообществом Node, также реализованы шаблонизаторы, причем достаточно разнообразные. Если вы полагаете, что HTML- и (или) CSS-разметке не хватает элегантности, так как HTML требует закрывающих тегов, а CSS — открывающих и закрывающих скобок, обратите внимание на шаблонизаторы. Они позволяют файлам шаблонов использовать специальные языки (такие, как язык Pug, о котором мы поговорим в этой главе позже), предлагающие способы сокращенной записи HTML и (или) CSS.

Хотя шаблонизаторы могут сделать шаблоны более «чистыми», возможно, вам не захочется тратить время на изучение альтернативных способов записи HTML и CSS. В конце концов, окончательное решение зависит от ваших личных предпочтений.

В оставшейся части главы мы поговорим о том, как встроить шаблонизацию в ваши Node-приложения, на примере трех популярных шаблонизаторов:

- шаблонизатор EJS (Embedded JavaScript);
- минималистский шаблонизатор Hogan;
- шаблонизатор Pug.

Каждый из этих шаблонизаторов предлагает альтернативный способ записи разметки HTML. Начнем с EJS.

7.2. Шаблонизация с EJS

EJS (Embedded JavaScript, *https://github.com/visionmedia/ejs*) предлагает достаточно прямолинейный подход к шаблонизации, который будет близок разработчикам, использующим шаблоны в других языках, таких как JSP (Java Server Pages), Smarty (PHP), ERB (Embedded Ruby) и подобных. Теги EJS внедряются в разметку HTML-кода в качестве заполнителей для данных. Кроме того, EJS позволяет выполнять в шаблонах простейшую логику JavaScript для таких операций, как условное ветвление и итерация, как это делается в PHP.

В этом разделе вы узнаете:

- как создавать EJS-шаблоны;
- как использовать EJS-фильтры для поддержки обычной функциональности представлений, такой как манипулирование текстом, сортировка и итерация;
- как интегрировать EJS с приложениями Node;
- как использовать EJS для клиентских приложений.

А теперь давайте глубже погрузимся в мир шаблонов EJS.

7.2.1. 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 использованием контекста:

```
const ejs = require('ejs');
const template = '<%= message %>';
const context = { message: 'Hello template!' };
console.log(ejs.render(template, context));
```

Обратите внимание на использование `locals` во втором аргументе, передаваемом `render`. Второй аргумент может включать параметры визуализации, а также контекстные данные. Параметр `locals` гарантирует, что отдельные фрагменты 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 ключевые слова: `cache`, `client`, `close`, `compileDebug`, `debug`, `filename`, `open` или `scope`. Они зарезервированы с целью изменения настроек шаблонизатора.

Экранирование символов

В процессе визуализации EJS *экранирует* (escapes) все специальные символы в контекстных значениях, заменяя их кодами HTML-сущностей. Это позволяет предотвратить атаки межсайтового выполнения сценариев (Cross-Site Scripting, XSS), в ходе которых злоумышленники пытаются в качестве данных отправить вредоносный код JavaScript 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:

```
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 следующий текст:

```
&lt;script&gt;alert('XSS attack!');&lt;/script&gt;
```

Если вы доверяете используемым в шаблоне данным и не хотите экранировать контекстные значения в EJS, используйте в теге шаблона символы `<%-` вместо `<%=`, как в следующем примере:

```
const ejs = require('ejs');
const template = '<%- message %>';
const 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:

```
const ejs = require('ejs');
ejs.delimiter = '$'
const template = '<$= message $>';
const context = { message: 'Hello template!' };
console.log(ejs.render(template, context));
```

Итак, вы ознакомились с основами EJS; давайте посмотрим, как EJS позволяет упростить управление представлением данных.

7.2.2. Интеграция шаблонов EJS в приложение

Хранить шаблоны в файлах вместе с кодом приложения неудобно, вдобавок это приводит к загромождению кода. Мы воспользуемся API файловой системы Node для загрузки шаблонов из отдельных файлов.

Перейдите в рабочую папку и создайте файл **app.js** с кодом из листинга 7.5.

Листинг 7.5. Хранение шаблонного кода в файлах

```
const ejs = require('ejs');
const fs = require('fs');
const http = require('http');
const filename = './templates/students.ejs';   <-- Обозначает местонахождение файла шаблона.
const students = [   <-- Данные для передачи шаблонизатора.
  { name: 'Rick LaRue', age: 23 },
  { name: 'Sarah Cathands', age: 25 },
  { name: 'Bob Dobbs', age: 37 }
];

const server = http.createServer((req, res) => {   <-- Создает сервер HTTP.
  if (req.url === '/') {
    fs.readFile(filename, (err, data) => {   <-- Читает шаблон из файла.
      const template = data.toString();
      const context = { students: students };
      const output = ejs.render(template, context);   <-- Выполняет визуализацию шаблона.
      res.setHeader('Content-type', 'text/html');
      res.end(output);   <-- Отправляет ответ HTTP.
    });
  } else {
    res.statusCode = 404;
    res.end('Not found');
  }
});

server.listen(8000);
```

Затем создайте дочерний каталог с именем **templates**. В этом каталоге будут храниться шаблоны. Создайте файл **students.ejs** в каталоге **templates**. Включите в файл **templates/students.ejs** код из листинга 7.6.

Листинг 7.6. Шаблон EJS для построения массива

```
<% if (students.length) { %>
  <ul>
    <% students.forEach((student) => { %>
      <li><%= student.name %> (<%= student.age %>)</li>
    <% }) %>
  </ul>
<% } %>
```

Кэширование шаблонов EJS

EJS поддерживает кэширование функций шаблона в памяти. Это означает, что после однократного разбора файла шаблона созданная в результате функция сохраняется в памяти. Визуализация кэшированного шаблона осуществляется быстрее, поскольку этап разбора пропускается.

Если вы занимаетесь разработкой веб-приложения в Node и хотите, чтобы изменения, внесенные в файлы шаблона, немедленно отражались в приложении, 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:

```
const cache = process.env.NODE_ENV === 'production';
const output = ejs.render(
  template,
  { students, cache, filename }
);
```

Обратите внимание на то, что в параметре `filename` не обязательно указывать файл. В данном случае может применяться уникальное значение, определяющее, какой шаблон вы визуализируете.

Теперь, когда вы знаете, как интегрировать шаблоны EJS в Node-приложение, давайте рассмотрим другую возможность применения EJS в браузерах.

7.2.3. Использование EJS в клиентских приложениях

Чтобы воспользоваться EJS на стороне клиента, сначала нужно загрузить шаблонизатор EJS в рабочий каталог с помощью следующих команд:

```
cd /your/working/directory
curl -O https://raw.githubusercontent.com/tj/ejs/master/lib/ejs.js
```

После загрузки файла **ejs.js** EJS можно использовать в клиентском коде. В листинге 7.7 представлено простое клиентское приложение с шаблоном EJS. Сохранив код в файле с именем **index.html**, вы сможете открыть его в браузере и просмотреть результаты.

Листинг 7.7. Добавление шаблонизации на стороне клиента средствами EJS

```
<html>
  <head>
  <title>EJS example</title>
    <script src="ejs.js"></script>
```

```
    <script
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.8/jquery.js">
    </script>
  </head>
  <body>
    <div id="output"></div>
    <script>
      const template = "<%= message %>";
      const context = { message: 'Hello template!' };
      $(document).ready(() => {
        $('#output').html(
          ejs.render(template, context)
        );
      });
    </script>
  </body>
</html>
```

Включает библиотеку jQuery для операций с DOM.

Заполнитель для результатов визуализации шаблона.

Шаблон, используемый для визуализации контента.

Данные для шаблона.

Ожидает загрузки страницы в браузере.

Выполняет визуализацию шаблона в div с идентификатором «output».

Теперь вы знаете, как использовать в Node полнофункциональный шаблонизатор. Пришло время познакомиться с шаблонизатором Hogan, в котором функциональность, доступная в коде шаблона, намеренно ограничена.

7.3. Использование языка Mustache с шаблонизатором Hogan

Шаблонизатор hogan.js (*https://github.com/twitter/hogan.js*) был создан в компании Twitter для собственных потребностей в работе с шаблонами. Hogan представляет собой реализацию популярного стандарта языка шаблонов Mustache (*http://mustache.github.com/*), разработанного Крисом Ванстратом (Chris Wanstrath) для проекта GitHub.

В Mustache выбран минималистский подход к шаблонизации. В отличие от EJS стандарт Mustache сознательно не включает ни условную логику, ни встроенные средства фильтрации контента (кроме экранирования контента для предотвращения XSS-атак). Разработчики Mustache постарались сделать код шаблона максимально простым.

В этом разделе мы узнаем:

- как создать и реализовать в приложении шаблоны Mustache;
- 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.

7.3.1. Создание шаблона

Чтобы использовать Hogan в приложении или выполнить любой из примеров, предлагаемых в этом разделе, нужно установить Hogan в каталог приложения (ch07-templates/hogan-snippet). Для этого в командной строке введите следующую команду:

```
npm i --save hogan.js
```

Далее представлен простейший пример использования Hogan в приложении Node для визуализации простого шаблона на основе контекста. При его выполнении выводится текст «Hello template!».

```
const hogan = require('hogan.js');
const templateSource = '{{message}}';
const context = { message: 'Hello template!' };
const template = hogan.compile(templateSource);
console.log(template.render(context));
```

Теперь, когда вы знаете, как с помощью Hogan обрабатывать шаблоны Mustache, перейдем к рассмотрению тегов, предлагаемых стандартом Mustache.

7.3.2. Теги Mustache

Концептуально теги Mustache не отличаются от тегов EJS. Теги Mustache резервируют место для значений переменных, а также указывают на необходимость перебора, что позволяет расширить функциональность Mustache или добавить в шаблон комментарий.

Вывод простых значений

Чтобы вывести контекстное значение в шаблоне Mustache, заключите имя значения в двойные фигурные скобки. Подобные скобки среди Mustache-разработчиков получили жаргонное название «mustaches» (дословно — усы). Если, например, нужно вывести значение для контекстного элемента `name`, воспользуйтесь тегом Hogan `{{name}}`.

Подобно большинству других шаблонизаторов, Hogan по умолчанию экранирует контент, чтобы предотвратить возможные XSS-атаки. Для вывода неэкранированного значения в Hogan можно либо добавить третью фигурную скобку, либо предварить контекстный элемент символом `&`. Так, в предыдущем примере для вывода неэкранированного контекстного значения будет использоваться тег `{{{name}}}` или `{{&name}}`.

Для добавления в шаблон Mustache комментария применяется формат вида

```
{{! This is a comment }}
```

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 }
   ]
};
```

Допустим, 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>
```

С шаблонами Hogan эта задача решается тривиально:

```
{{#students}}
  <p>Name: {{name}}, Age: {{age}} years old</p>
{{/students}}
```

Инвертированные секции — HTML по умолчанию при отсутствии значений

Что произойдет, если элемент `students` в контекстных данных не является массивом? Если, например, это один объект, то шаблон просто выведет его. 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 предоставляет 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 разработчики могли расширять функциональность Mustache, стандарт допускает определение тегов секций, обрабатывающих контент шаблона путем

вызова функций, а не перебором элементов массива. Такие секции называются *лямбда-секциями* (section lambda).

В листинге 7.8 приведен пример того, 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 --dev
```

Если вы используете архив исходного кода книги, выполните команду `npm install` из каталога ch07-templates/listing7_8.

В листинге 7.8 конструкция `**Name**` в шаблоне визуализируется в разметку `<strong>Name</strong>` при передаче через парсер Markdown, который вызывается кодом лямбда-секции.

Листинг 7.8. Использование лямбда-секции в Hogan

```
const hogan = require('hogan.js');
const md = require('github-flavored-markdown');
const templateSource = `
  {{#markdown}}**Name**: {{name}}{{/markdown}}
`;
const context = {
  name: 'Rick LaRue',
  markdown: () => text => md.parse(text)
};
const template = hogan.compile(templateSource);
console.log(template.render(context));
```

Включает парсер Markdown.

Шаблон Mustache также содержит форматирование Markdown.

Контекст шаблона включает лямбда-секцию для разбора разметки Markdown в шаблоне.

С помощью лямбда-секций вы сможете легко реализовать в своих шаблонах такие вещи, как кэширование и преобразования.

Компоненты — 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 7.9 с помощью компонента код шаблона, применяемый для вывода на экран данных о студентах, выделяется из главного шаблона.

Листинг 7.9. Использование компонентов в Hogan

```
const hogan = require('hogan.js');
const studentTemplate = `
  <p>
    Name: {{name}},
```

Шаблонный код, используемый для компонента.

```
    Age: {{age}} years old
  </p>
`;
const mainTemplate = `
  {{#students}}
    {{>student}}
  {{/students}}
`;
const context = {
  students: [{
     name: 'Jane Narwhal',
     age: 21
  }, {
     name: 'Rick LaRue',
     age: 26
  }]
};
const template = hogan.compile(mainTemplate);
const partial = hogan.compile(studentTemplate);
const html = template.render(context, {student: partial });
console.log(html);
```

Код основного шаблона.

Компиляция основного и частичного шаблона.

Визуализация основного и частичного шаблона.

7.3.3. Тонкая настройка Hogan

Использовать Hogan очень просто — достаточно изучить набор тегов, и можно приступать к работе. Возможно, в процессе работы вам потребуется изменить всего несколько 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 разделителем в стиле EJS:

```
hogan.compile(text, { delimiters: '<% %>' });
```

Кроме Mustache, существуют и другие языки шаблонов. Одним из проектов, который старается по возможности избавить разработчика от рутины HTML, является Pug.

7.4. Шаблоны Pug

Шаблонизатор Pug (*http://pugjs.org*), ранее известный под названием Jade, предоставляет альтернативный механизм определения HTML. Ключевое различие между Pug и большинством других систем шаблонизации заключается в том, что в Pug применяются содержательные пробельные символы (whitespace). При создании шаблонов Pug задаются отступы, определяющие уровень вложенности тегов HTML. Кроме того, теги HTML не обязательно явно закрывать, что позволяет исключить проблему случайной преждевременной вставки закрывающих тегов или их полного

отсутствия. Благодаря отступам шаблоны становятся визуально более свободными, что облегчает их сопровождение.

Чтобы вы лучше поняли, как это работает, посмотрим, как представляется следующий фрагмент HTML:

```
<html>
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <div id="main" class="content">
      <strong>"Hello world!"</strong>
    </div>
  </body>
</html>
```

Для моделирования этой разметки HTML можно задействовать такой шаблон Pug:

```
html
   head
      title Welcome
   body
      div.content#main
         strong "Hello world!"
```

В Pug, как и в EJS, допускается внедрение кода JavaScript как на стороне сервера, так и на стороне клиента. Pug предлагает также дополнительные механизмы, такие как наследование шаблонов и примеси (mixins). Благодаря примесям можно определять простые многократно используемые мини-шаблоны, предназначенные для представления в разметке HTML наиболее востребованных визуальных элементов, таких как списки и поля. Примеси (mixins) напоминают применяемые в Hogan.js компоненты, упоминавшиеся в предыдущем разделе. Наследование шаблонов упрощает организацию шаблонов Pug, необходимых для визуализации одной страницы HTML в нескольких файлах. Далее мы подробно рассмотрим эти механизмы. Чтобы установить Pug в папку Node-приложения, в командной строке введите следующую команду:

```
npm install pug --save
```

В этом разделе мы узнаем:

- основные принципы Pug: как задаются имена классов, атрибуты и блочное расширение;
- как добавить программную логику в шаблоны Pug с помощью встроенных ключевых слов;
- как организовать шаблоны с применением наследования, блоков и примесей.

Для начала рассмотрим основы использования и синтаксиса Pug.

7.4.1. Основные сведения о Pug

В Pug используются те же имена тегов, что и в HTML, но в Pug можно не применять открывающие и закрывающие символы < и >, а уровень вложенности тегов может выражаться отступами. Тег может включать в себя 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
```

Идентификаторы CSS назначаются тегам с помощью префикса `#<ID>`. Для назначения идентификатора CSS `featured_content` в предыдущем примере может использоваться следующая конструкция Pug:

```
div.content.sidebar#featured_content
```

Сокращенная запись тега div

Поскольку тег `div` очень часто используется в разметке HTML, в Pug предусмотрена сокращенная форма его записи. Следующий пример кода генерирует ту же разметку HTML, что и предыдущий:

```
.content.sidebar#featured_content
```

Теперь, когда вы научились записывать теги HTML и соответствующие им классы CSS и идентификаторы, перейдем к назначению атрибутов тегов HTML.

Запись атрибутов тегов

Атрибуты тегов записываются в круглых скобках и разделяются запятыми. Чтобы определить гиперссылку, которая откроется в другой вкладке, воспользуйтесь следующей записью Pug:

```
a(href='http://nodejs.org', target='_blank')
```

Поскольку перечисление атрибутов тегов в Pug может привести к появлению длинных строк, шаблонизатор предлагает более гибкий подход. Следующая запись Pug вполне корректна и эквивалентна предыдущей:

```
a(href='http://nodejs.org'
  target='_blank')
```

Можно также указывать атрибуты, которые не требуют значений. Следующий пример кода Pug демонстрирует определение формы HTML, которая включает в себя элемент `select` вместе с предварительно выбранным параметром:

```
strong Select your favorite food:
form
```

```
  select
    option(value='Cheese') Cheese
    option(value='Tofu', selected) Tofu
```

Запись контента тегов

В предыдущем фрагменте кода вы уже видели примеры контента тегов: `Select your favorite food` после тега `strong`, `Cheese` после первого тега `option` и `Tofu` после второго тега `option`.

Это стандартный, но не единственный способ определения контента тегов Pug. Хотя подобный стиль прекрасно подходит для записи небольших фрагментов контента, если контента окажется много, это может привести к появлению шаблонов Pug со слишком длинными строками. Как видно из следующего примера, для предотвращения подобных проблем при записи контента в Pug можно использовать символ |:

```
textarea
  | This is some default text
  | that the user should be
  | provided with.
```

Если тег HTML (такой, как `style` или `script`) принимает только текст (то есть вложение элементов HTML не допускается), символы | можно опустить, как в следующем примере:

```
style
  h1 {
    font-size: 6em;
    color: #9DFF0C;
  }
```

Наличие двух разных способов выражения длинного и короткого контента тегов упрощает создание элегантных шаблонов Pug. Кроме того, в Pug поддерживается альтернативный механизм описания вложенности, который называется *блочным расширением* (block expansion).

Упорядочение кода путем блочного расширения

В Pug вложенные теги обычно выделяются отступами, но иногда это может привести к слишком большому количеству пробельных символов. Например, в этом шаблоне Pug 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
```

Pug позволяет записать этот пример в более компактной форме с использованием блочного расширения. 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 остается 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 в Pug представляется разметка, давайте выясним, как интегрировать шаблон Pug в веб-приложение.

Включение данных в шаблоны Pug

Данные передаются шаблонам Pug так же, как и шаблонам EJS. Сначала шаблон компилируется в функцию, которая затем вызывается вместе с контекстом для визуализации вывода HTML. Пример:

```
const pug = require('pug');
const template = 'strong #{message}';
const context = { message: 'Hello template!' };
const fn = pug.compile(template);
console.log(fn(context));
```

В этом примере кода конструкция `#{message}` в шаблоне определяет заполнитель, который заменяется контекстным значением.

С помощью контекстных значений можно также предоставлять значения для атрибутов. Например, следующий шаблон визуализирует разметку `<a href="http://google.com"></a>`:

```
const pug = require('pug');
const template = 'a(href = url)';
const context = { url: 'http://google.com' };
const fn = pug.compile(template);
console.log(fn(context));
```

Теперь, когда мы узнали, как в Pug представляется разметка HTML и как данные приложений включаются в шаблоны Pug, пришло время узнать, как встроить в шаблон Pug программную логику.

7.4.2. Программная логика в шаблонах Pug

Когда вы передаете шаблонам Pug данные приложения, вам требуется логика, чтобы как-то обрабатывать эти данные. Pug позволяет непосредственно внедрять в шаблоны строки кода JavaScript — именно с помощью JavaScript вы определяете программную логику в своих шаблонах. Чаще всего используются инструкции `if`,

циклы `for` и объявления `var`. Но прежде чем углубиться в детали, давайте рассмотрим пример шаблона, визуализирующего список контактов. Этот пример хорошо иллюстрирует то, как логика Pug 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 рассмотрим различные способы обработки выводимых данных шаблонизатором Pug при внедрении кода JavaScript.

Использование JavaScript-кода в шаблонах Pug

Если поставить перед строкой кода JavaScript префикс `-`, код JavaScript будет выполнен без возвращения какого-либо значения в выводимых шаблоном данных. Если же использовать префикс `=`, возвращаемое этим кодом значение попадет в выводимые данные, причем оно будет экранировано для предотвращения XSS-атак. Если же код, генерируемый из JavaScript, экранировать не нужно, можно снабдить его префиксом `!=`. Перечень доступных префиксов приведен в табл. 7.1.

Таблица 7.1. Префиксы, используемые для встраивания JavaScript в Pug

Префикс	Вывод
=	Выходные данные экранируются (для не заслуживающих доверия или непредсказуемых значений, для защиты от XSS-атак)
!=	Выходные данные не экранируются (для заслуживающих доверия или предсказуемых значений)
-	Выходные данные отсутствуют

В Pug включено несколько наиболее востребованных условных и итеративных инструкций, которые можно записывать без префиксов: `if`, `else`, `case`, `when`, `default`, `until`, `while`, `each` и `unless`.

Кроме того, в Pug можно определять переменные. В следующем примере кода показаны два эквивалентных способа присваивания значений переменным в Pug:

```
- count = 0
count = 0
```

Как и упомянутые ранее инструкции с префиксом -, инструкции без префиксов не порождают никаких данных.

Перебор объектов и массивов

В Pug значения, переданные в контексте, доступны для кода JavaScript. В следующем примере кода мы считываем шаблон Pug из файла, а затем передаем шаблону контекст с парой сообщений, которые мы собираемся показывать в массиве:

```
const pug = require('pug');
const fs = require('fs');
const template = fs.readFileSync('./template.pug');
const context = { messages: [
  'You have logged in successfully.',
  'Welcome back!'
]};
const fn = pug.compile(template);
console.log(fn(context));
```

Шаблон Pug содержит следующий код:

```
- messages.forEach(message => {
  p= message
- })
```

Финальная разметка HTML должна выглядеть следующим образом:

```
<p>You have logged in successfully.</p><p>Welcome back!</p>
```

В Pug также поддерживается отличная от JavaScript форма итераций, реализуемая инструкцией `each`. С ее помощью можно легко 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 перебор свойств объекта:

```
each value, key in post
   div
      strong #{key}
      p value
```

Условная визуализация в шаблоне

Иногда шаблон должен решить, как именно должны выводиться те или иные данные, в зависимости от значения этих данных. В следующем примере кода показано условие, в котором примерно в половине случаев тег `script` выводится в формате HTML:

```
- n = Math.round(Math.random() * 1) + 1
- if (n == 1) {
  script
    alert('You win!');
- }
```

Условные выражения Pug также могут записываться в более чистой альтернативной форме:

```
- n = Math.round(Math.random() * 1) + 1
  if n == 1
    script
      alert('You win!');
```

Если в условиях используются отрицания (например, `if (n != 1)`), можно воспользоваться ключевым словом `unless`:

```
- n = Math.round(Math.random() * 1) + 1
  unless n == 1
    script
      alert('You win!');
```

Инструкция case

В Pug также поддерживается условная конструкция `case`, напоминающая `switch`: она позволяет выбрать в шаблоне нужный вариант выводимых данных для одного из нескольких возможных сценариев.

Следующий пример шаблона демонстрирует использование инструкции `case` для вывода результатов поиска в блоге тремя разными способами. Если в результате поиска ничего не найдено, выводится соответствующая информация. Если найдено единственное сообщение, оно показывается полностью. Если найдено несколько сообщений, то инструкция `each` перебирает сообщения 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
```

7.4.3. Организация шаблонов Pug

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 структурному элементу приложения, например странице, врезке или контенту сообщения в блоге.

В этом разделе мы рассмотрим несколько механизмов, с помощью которых разные файлы шаблонов для визуализации контента объединяются вместе:

- структурирование шаблонов путем наследования;
- реализация макетов путем добавления блока в начало и в конец шаблона;
- включение шаблона;
- многократное использование программной логики шаблона с помощью примесей.

Начнем с наследования шаблонов в Pug.

Структурирование нескольких шаблонов путем наследования

Наследование шаблонов — один из механизмов структурирования. В рамках этой концепции шаблоны рассматриваются по аналогии с классами в парадигме объектно-ориентированного программирования. Один шаблон может расширять другой шаблон, который в свою очередь может расширять еще один шаблон. Количество уровней наследования ограничивается лишь соображениями целесообразности.

В качестве простого примера давайте выясним, как путем наследования шаблонов создать базовую HTML-обертку для контента страницы. В рабочем каталоге создайте папку **template**, в которой будет находиться Pug-файл примера. Для шаблона страницы создается файл **layout.pug** со следующей разметкой:

```
html
  head
    block title
  body
    block content
```

Шаблон **layout.pug** содержит минимальное определение страницы HTML в виде двух *блоков* (blocks). С помощью блоков при наследовании шаблонов задается место, в котором будет находиться контент, предоставляемый производным шаблоном. В файле **layout.pug** для этого служат два блока: блок `title` позволяет производному шаблону задать заголовок, а блок `content` — то, что должно выводиться на странице.

Затем в папке шаблона, находящейся в рабочем каталоге, создайте файл page.pug. Этот файл шаблона заполняет блоки `title` и `content`:

```
extends layout
block title
  title Messages
block content
  each message in messages
    p= message
```

Наконец, добавьте программную логику из листинга 7.10 (это модифицированная версия предыдущего примера данного раздела), которая выводит результаты применения шаблона и показывает наследование в действии.

Листинг 7.10. Наследование шаблонов в действии

```
const pug = require('pug');
const fs = require('fs');
const templateFile = './templates/page.pug';
const iterTemplate = fs.readFileSync(templateFile);
const context = { messages: [
  'You have logged in successfully.',
  'Welcome back!'
]};
const iterFn = pug.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.pug, не содержали контента, что делало задачу включения контента в шаблон page.pug 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 без замены существующего контента.

Следующий шаблон, layout.pug, содержит дополнительный блок `scripts` с контентом (тегом `script`, который предназначен для загрузки библиотеки jQuery):

```
html
  head
    - const baseUrl = "http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/"
    block title
    block style
```

```
    block scripts
  body
    block content
```

Если вы хотите, чтобы шаблон page.pug дополнительно загружал библиотеку jQuery, можете воспользоваться шаблоном из листинга 7.11.

Листинг 7.11. Использование механизма добавления блока в конец шаблона для загрузки дополнительного JavaScript-файла

```
extends layout  <-------  Расширяет шаблон layout.
block title
  title Messages
block style  <-------  Определяет блок style.
  link(rel="stylesheet", href=baseUrl+"themes/flick/jquery-ui.
block scripts  <-------  Определяет блок scripts.
  script(src=baseUrl+"jquery-ui.js")
block content
  - count = 0
  each message in messages
    - count = count + 1
    script
      $(() => {
        $("#message_#{count}").dialog({
          height: 140,
          modal: true
        });
      });
    != '<div id="message_' + count + '">' + message + '</div>'
```

Наследование — не единственный путь объединения нескольких шаблонов. Также можно воспользоваться командой Pug `include`.

Включение шаблонов

Еще одним инструментом организации шаблонов является команда Pug `include`. Эта команда встраивает в шаблон контент другого шаблона. Если в шаблон layout.pug из предыдущего примера добавить строку `include footer`, получится 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.pug в разметку, визуализируемую шаблоном layout.pug, как показано на рис. 7.3.

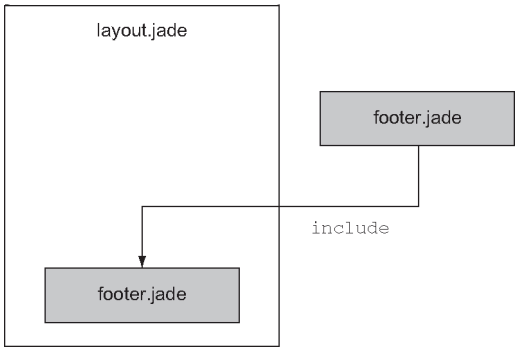


Рис. 7.3. Механизм include в Pug предоставляет простой способ включения контента одного шаблона в другой шаблон во время визуализации

Этот механизм может применяться, например, для добавления в файл layout.pug информации о веб-сайте или элементов дизайна. Указав расширение, можно также включать в шаблон файлы, не относящиеся к Pug, например:

```
include twitter_widget.html
```

Примеси и многократное использование программной логики шаблона

Хотя для использования готовых фрагментов кода может применяться команда Pug `include`, с ее помощью вряд ли удастся создать библиотеку многократно используемой функциональности, которая могла бы использоваться как страницами, так и приложениями. Для решения этой задачи в Pug служит команда `mixin`, которая позволяет определять *примеси* — многократно используемые фрагменты кода Pug.

Примеси Pug напоминают функции JavaScript. Как и функция, примесь может получать аргументы, которые учитываются при генерировании кода Pug.

Предположим, например, что наше приложение обрабатывает следующую структуру данных:

```
const students = [
  { name: 'Rick LaRue', age: 23 },
  { name: 'Sarah Cathands', age: 25 },
  { name: 'Bob Dobbs', age: 37 }
];
```

Чтобы определить способ вывода списка HTML, получаемого из значений заданного свойства каждого объекта, можно задействовать следующую примесь:

```
mixin list_object_property(objects, property)
  ul
    each object in objects
      li= object[property]
```

Затем можно воспользоваться примесью для вывода данных следующей строкой Pug:

```
mixin list_object_property(students, 'name')
```

Пользуясь наследованием шаблонов, инструкциями `include` и примесями, вы можете без проблем многократно использовать визуализирующую разметку без чрезмерного «разбухания» файлов шаблона.

7.5. Заключение

- Шаблонизаторы помогают упорядочить прикладную логику и средства визуализации.
- В Node поддерживаются некоторые популярные шаблонизаторы, включая EJS, Hogan.js и Pug.
- EJS поддерживает простые управляющие конструкции и экранирование при незащищенной интерполяции.
- Hogan.js — простой шаблонизатор без управляющих конструкций, но с поддержкой стандарта Mustache.
- Pug — более сложный язык шаблонов, который может выводить разметку HTML без использования угловых скобок.
- Работа Pug зависит от пробельных символов при внедрении тегов.

8 Хранение данных в приложениях

Node.js ориентируется на невероятно широкий спектр разработчиков со столь же разнообразными потребностями. Ни одна база данных или технология хранения данных не способна удовлетворить большинство сценариев использования, обслуживаемых Node. В этой главе предоставлен широкий обзор возможностей хранения данных наряду с некоторыми важными высокоуровневыми концепциями и терминологией.

8.1. Реляционные базы данных

На протяжении почти всего времени существования веб-технологий реляционные базы данных занимали ведущее место в области хранения данных. Эта тема уже рассматривается во многих других книгах и образовательных программах, поэтому мы не будем тратить много времени на ее рассмотрение.

Реляционные базы данных, базирующиеся на математических концепциях реляционной алгебры и теории множеств, известны с 1970-х годов. *Схема* (schema) определяет формат различных типов данных и отношения, существующие между этими типами. Например, при построении социальной сети можно создать типы данных `User` и `Post` и определить отношения «один ко многим» между `User` и `Post`. Далее на языке SQL (Structured Query Language) формулируются запросы к данным типа «Получить все сообщения, принадлежащие пользователю с идентификатором 123», или на SQL: `SELECT * FROM post WHERE user_id=123`.

8.2. PostgreSQL

MySQL и PostgreSQL (Postgres) остаются самыми популярными реляционными базами данных для приложений Node. Различия между реляционными базами данных в основном эстетические, поэтому этот раздел в равной степени относится

и к другим реляционным базам данных — например, MySQL в Node. Но сначала разберемся, как установить Postgres на машине разработки.

8.2.1. Установка и настройка

Сначала нужно установить Postgres в вашей системе. Простой команды `npm install` для этого недостаточно. Инструкции по установке зависят от платформы. В macOS установка сводится к простой последовательности команд:

```
brew update
brew install postgres
```

Если в вашей системе поддержка Postgres уже установлена, вы можете столкнуться с проблемами при попытке обновления. Выполните инструкции для своей платформы, чтобы произвести миграцию существующих баз данных, либо полностью сотрите каталог базы данных:

```
# WARNING: will delete existing postgres configuration & data
rm -rf /usr/local/var/postgres
```

Затем инициализируйте и запустите Postgres:

```
initdb -D /usr/local/var/postgres
pg_ctl -D /usr/local/var/postgres -l logfile start
```

Эти команды запускают демона Postgres. Демон должен запускаться каждый раз, когда вы перезагружаете компьютер. Возможно, вам строит настроить автоматическую загрузку демона Postgres при запуске; во многих сетевых руководствах можно найти описание этого процесса для вашей операционной системы.

Во многих системах семейства Linux имеется пакет для установки Postgres. Для системы Windows следует загрузить программу установки на сайте postgresql.org (*www.postgresql.org/download/windows/*).

В составе Postgres устанавливаются некоторые административные программы командной строки. Ознакомьтесь с ними; необходимую информацию можно найти в электронной документации.

8.2.2. Создание базы данных

После того как демон Postgres заработает, необходимо создать базу данных. Эту процедуру достаточно выполнить всего один раз. Проще всего воспользоваться программой createdb из режима командной строки. Следующая команда создает базу данных с именем `articles`:

```
createdb articles
```

Если операция завершается успешно, команда ничего не выводит. Если база данных с указанным именем уже существует, команда ничего не делает и сообщает об ошибке.

Как правило, приложения в любой момент времени подключены только к одной базе данных, хотя можно настроить сразу несколько баз данных в зависимости от режима, в котором работает база данных. Во многих приложениях различаются по крайней мере два режима: режим разработки и режим реальной эксплуатации.

Чтобы удалить все данные из существующей базы данных, выполните команду `dropdb` с терминала, передав ей имя базы данных в аргументе:

```
dropdb articles
```

Чтобы снова использовать базу данных, необходимо выполнить команду `createdb`.

8.2.3. Подключение к Postgres из Node

Самый популярный пакет для взаимодействия с Postgres из Node называется pg. Его можно установить при помощи npm:

```
npm install pg --save
```

Когда сервер Postgres заработает, база данных будет создана, а пакет pg установлен, вы сможете переходить к использованию базы данных из Node. Прежде чем вводить какие-либо команды к серверу, необходимо создать подключение к нему, как показано в листинге 8.1.

Листинг 8.1. Подключение к базе данных

```
const pg = require('pg');
const db = new pg.Client({ database: 'articles' });
db.connect((err, client) => {
  if (err) throw err;
  console.log('Connected to database', db.database);
  db.end();
});
```

Параметры конфигурации подключения.

Закрывает подключение к базе данных, позволяя процессу node завершиться.

Подробную документацию по `pg.Client` и другим методам можно найти на вики-странице пакета pg на GitHub: *https://github.com/brianc/node-postgres/wiki*.

8.2.4. Определение таблиц

Чтобы хранить данные в PostgreSQL, сначала необходимо определить таблицы и формат данных, которые в них будут храниться. Пример такого рода приведен в листинге 8.2 (ch08-databases/listing8_3 в архиве исходного кода книги).

Листинг 8.2. Определение схемы

```
db.query(`
  CREATE TABLE IF NOT EXISTS snippets (
    id SERIAL,
    PRIMARY KEY(id),
    body text
  );
`, (err, result) => {
  if (err) throw err;
  console.log('Created table "snippets"');
  db.end();
});
```

8.2.5. Вставка данных

После того как таблица будет определена, в нее можно вставить данные запросами `INSERT` (листинг 8.3). Если значение `id` не указано, то PostgreSQL выберет его за вас. Чтобы узнать, какой идентификатор был выбран для конкретной записи, присоедините условие `RETURNING id` к запросу; идентификатор будет выведен в строках результата, переданного функции обратного вызова.

Листинг 8.3. Вставка данных

```
const body = 'hello world';
db.query(`
  INSERT INTO snippets (body) VALUES (
    '${body}'
  )
  RETURNING id
`, (err, result) => {
  if (err) throw err;
  const id = result.rows[0].id;
  console.log('Inserted row with id %s', id);
  db.query(`
    INSERT INTO snippets (body) VALUES (
      '${body}'
    )
    RETURNING id
  `, () => {
    if (err) throw err;
    const id = result.rows[0].id;
    console.log('Inserted row with id %s', id);
  });
});
```

8.2.6. Обновление данных

После того как данные будут вставлены, их можно будет обновить запросом `UPDATE` (листинг 8.4). Количество записей, задействованных в обновлении, будет доступно в свойстве `rowCount` результата запроса. Полный пример для этого листинга содержится в каталоге ch08-databases/listing8_4.

Листинг 8.4. Обновление данных

```
const id = 1;
const body = 'greetings, world';
db.query(`
  UPDATE snippets SET (body) = (
    '${body}'
  ) WHERE id=${id};
`, (err, result) => {
  if (err) throw err;
  console.log('Updated %s rows.', result.rowCount);
});
```

8.2.7. Запросы на выборку данных

Одна из самых замечательных особенностей реляционных баз данных — возможность выполнения сложных произвольных запросов к данным. Запросы выполняются командой `SELECT`, а простейший пример такого рода представлен в листинге 8.5.

Листинг 8.5. Запрос данных

```
db.query(`
  SELECT * FROM snippets ORDER BY id
`, (err, result) => {
  if (err) throw err;
  console.log(result.rows);
});
```

8.3. Knex

Многие разработчики предпочитают работать с командами SQL в своих приложениях не напрямую, а через абстрактную надстройку. Это желание вполне понятно: конкатенация строк в команды SQL может быть громоздким процессом, который усложняет понимание и сопровождение запросов. Сказанное особенно справедливо по отношению к языку JavaScript, в котором не было синтаксиса представления многострочных строк до появления в ES2015 шаблонных литералов (см. *https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals*). На рис. 8.1 показана статистика Knex с количеством загрузок, доказывающих популярность.

Рис. 8.1. Статистика использования Knex

Knex — пакет Node, реализующий облегченную абстракцию для SQL, известную как *построитель запросов*. Построитель запросов формирует строки SQL через

декларативный API, который имеет много общего с генерируемыми командами SQL. Knex API интуитивен и предсказуем:

```
knex({ client: 'mysql' })
  .select()
  .from('users')
  .where({ id: '123' })
  .toSQL();
```

Этот вызов создает параметризованный запрос SQL на диалекте MySQL:

```
select * from `users` where `id` = ?
```

8.3.1. jQuery для баз данных

Хотя стандарты ANSI и ISO SQL появились еще в середине 1980-х годов, большинство баз данных продолжает использовать собственные диалекты SQL. PostgreSQL является заметным исключением: эта база данных может похвастать соблюдением стандарта SQL:2008. Построитель запросов способен нормализировать различия между диалектами SQL, предоставляя единый унифицированный интерфейс для генерирования SQL в разных технологиях. Такой подход обладает очевидными преимуществами для групп, регулярно переключающихся между разными технологиями баз данных.

В настоящее время Knex.js поддерживает следующие базы данных:

- PostgreSQL;
- MSSQL;
- MySQL;
- MariaDB;
- SQLite3;
- Oracle.

В табл. 8.1 сравниваются способы генерирования команды `INSERT` в зависимости от выбранной базы данных.

Таблица 8.1. Сравнение команд SQL, сгенерированных Knex, для разных баз данных

База данных	SQL
PostgreSQL, SQLite и Oracle	insert into "users" ("name", "age") values (?, ?)
MySQL и MariaDB	insert into `users` (`name`, `age`) values (?, ?)
Microsoft SQL Server	insert into [users] ([name], [age]) values (?, ?)

Knex поддерживает *обещания* (promises) и обратные вызовы в стиле Node.

8.3.2. Подключение и выполнение запросов в Knex

В отличие от многих других построителей запросов, Knex также может подключаться и выполнять запросы к выбранному драйверу базы данных за вас.

```
db('articles')
  .select('title')
  .where({ title: 'Today's News' })
  .then(articles => {
    console.log(articles);
  });
```

По умолчанию запросы Knex возвращают обещания, но они также поддерживают соглашения обратного вызова Node с использованием `.asCallback`:

```
db('articles')
  .select('title')
  .where({ title: 'Today's News' })
  .asCallback((err, articles) => {
    if (err) throw err;
    console.log(articles);
  });
```

В главе 3 мы взаимодействовали с базой данных SQLite непосредственно при помощи пакета sqlite3. Этот API можно переписать с использованием Knex. Прежде чем запускать этот пример, сначала проверьте из npm, что пакеты knex и sqlite3 установлены:

```
npm install knex@~0.12.0 sqlite3@~3.1.0 --save
```

В листинге 8.6 sqlite используется для реализации простой модели `Article`. Сохраните файл под именем **db.js**; он будет использоваться в листинге 8.7 для взаимодействия с базой данных.

Листинг 8.6. Использование Knex для подключения и выдачи запросов к sqlite3

```
const knex = require('knex');

const db = knex({
  client: 'sqlite3',
  connection: {
    filename: 'tldr.sqlite'
  },
  useNullAsDefault: true
});

module.exports = () => {
  return db.schema.createTableIfNotExists('articles', table => {
    table.increments('id').primary();
    table.string('title');
    table.text('content');
  });
};
```

Выбор этого режима по умолчанию лучше работает при смене подсистемы баз данных.

Определяет первичный ключ с именем «id», значение которого автоматически увеличивается при вставке.

```
module.exports.Article = {
  all() {
    return db('articles').orderBy('title');
  },
  find(id) {
    return db('articles').where({ id }).first();
  },
  create(data) {
    return db('articles').insert(data);
  },
  delete(id) {
    return db('articles').del().where({ id });
  }
};
```

Листинг 8.7. Взаимодействие с API на базе Knex

```
db().then(() => {
  db.Article.create({
    title: 'my article',
    content: 'article content'
  }).then(() => {
    db.Article.all().then(articles => {
      console.log(articles);
      process.exit();
    });
  });
})
.catch(err => { throw err });
```

SQLite требует минимальной настройки: вам не нужно загружать демон сервера или создавать базы данных за пределами приложения. SQLite записывает все данные в один файл. Выполнив предыдущий код, вы увидите, что в текущем каталоге появился файл articles.sqlite. Чтобы уничтожить базу данных SQLite, достаточно удалить всего один файл:

```
rm articles.sqlite
```

SQLite также поддерживает режим работы в памяти, при котором запись на диск вообще не осуществляется. Этот режим обычно используется для ускорения выполнения автоматизированных тестов. Для настройки режима работы в памяти используется специальное имя файла `:memory:`. При открытии нескольких подключений к файлу `:memory:` каждое подключение получает собственную изолированную базу данных:

```
const db = knex({
  client: 'sqlite3',
  connection: {
    filename: ':memory:'
  },
  useNullAsDefault: true
});
```

8.3.3. Переход на другую базу данных

Благодаря использованию Knex, листинги 8.6 и 8.7 позволяют легко переключаться с sqlite3 на PostgreSQL. Для взаимодействия Knex с сервером PostgreSQL необходима установка и запуск пакета pg. Установите пакет pg в папку листинга 8.7 (ch08-databases/listing8_7 в коде книги) и не забудьте создать соответствующую базу данных программой командной строки PostgreSQL `createdb`:

```
npm install pg --save
createdb articles
```

Все изменения кода, необходимые для использования новой базы данных, вносятся в конфигурации Knex; в остальном API и схема использования идентичны:

```
const db = knex({
  client: 'pg',
  connection: {
    database: 'articles'
  }
})
```

Стоит заметить, что в реальной ситуации вам также придется выполнить миграцию всех существующих данных.

8.3.4. Остерегайтесь ненадежных абстракций

Построители запросов способны нормализовать синтаксис SQL, но не справятся с нормализацией поведения. Некоторые возможности поддерживаются только конкретными базами данных, и некоторые базы данных могут проявлять совершенно иное поведение при идентичных запросах. Например, ниже приведены два способа определения первичного ключа с использованием Knex:

- `table.increments('id').primary();`
- `table.integer('id').primary();`

Оба варианта работают в SQLite3 так, как ожидается, но второй вариант вызывает ошибку в PostgreSQL при вставке новой записи:

```
"null value in column "id" violates not-null constraint"
```

Значениям, вставленным в SQLite с неопределенным первичным ключом, будет присвоен автоматически увеличиваемый идентификатор — независимо от того, было ли явно настроено автоматическое увеличение для столбца первичного ключа. С другой стороны, PostgreSQL требует, чтобы столбцы с автоматическим увеличением определялись явно. Между базами данных существует много подобных поведенческих различий, и некоторые различия могут оставаться незаметными,

не приводя к видимым ошибкам. Если вы решили перейти на другую базу данных, обязательно проведите тщательное тестирование.

8.4. MySQL и PostgreSQL

Как MySQL, так и PostgreSQL — мощные, проверенные временем базы данных, и во многих проектах выбор одного или другого варианта практически ни на что не влияет. Многие различия, которые не играют роли, пока не возникнет необходимость в масштабировании проекта, существуют на границе интерфейса, предоставляемого разработчику приложений, или ниже нее.

Подробный сравнительный анализ реляционных баз данных в основном выходит за рамки книги, поскольку данная тема достаточно сложна. Некоторые важные различия перечислены ниже:

- PostgreSQL поддерживает более выразительные типы данных, включая массивы, JSON и типы, определяемые пользователем.
- PostgreSQL поддерживает встроенный механизм полнотекстового поиска.
- PostgreSQL в полной мере поддерживает стандарт ANSI SQL:2008.
- Поддержка репликации в PostgreSQL не настолько мощна и не так проверена на практике, как в MySQL.
- MySQL старше и обладает более многочисленным сообществом. Для MySQL существует больше совместимых инструментов и ресурсов.
- Сообщество MySQL в большей степени фрагментировано из-за нетривиальных различий между ветвями (например, MariaDB и WebScaleSQL от Facebook, Google, Twitter и т. д.).
- Ядро хранения данных MySQL с подключаемыми модулями создает больше проблем с пониманием, администрированием и настройкой. Тем не менее его можно рассматривать как полезную возможность для более точной настройки быстродействия.

MySQL и PostgreSQL проявляют разные характеристики быстродействия при масштабировании в зависимости от типа рабочей нагрузки. Особенности вашей нагрузки могут проявиться только в процессе становления проекта.

На многих интернет-ресурсах представлены гораздо более глубокие сравнения реляционных баз данных:

- *www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems*;
- *https://blog.udemy.com/mysql-vs-postgresql/*;
- *https://eng.uber.com/mysql-migration/*.

Исходный выбор базы данных вряд ли значительно повлияет на успех проекта, поэтому не стоит слишком сильно беспокоиться по поводу этого решения. Позднее вы сможете переключиться на другую базу данных, но, скорее всего, возможностей Postgres будет достаточно для любых потребностей в функциональности и масштабируемости. Однако если вы выбираете между несколькими базами данных, вам стоит познакомиться с идеей гарантий ACID.

8.5. Гарантии ACID

Сокращение «ACID» описывает набор желательных свойств, которыми должны обладать транзакции баз данных: атомарность (Atomicity), согласованность (Consistency), изолированность (Isolation) и устойчивость (Durability). Точные определения этих терминов могут различаться. Как правило, чем жестче система гарантирует свойства ACID, тем значительнее потери по быстродействию. Классификация ACID — распространенный способ, которым разработчик может быстро описать достоинства и недостатки конкретного решения (например, присущие системам NoSQL).

8.5.1. Атомарность

Атомарная транзакция не может быть выполнена частично: либо вся операция завершается, либо база данных остается без изменений. Например, если транзакция заключается в удалении всех комментариев некоторого пользователя, то либо все комментарии будут удалены, либо не будет удален ни один. Ситуация, при которой часть комментариев удаляется, а другая часть остается, невозможна.

Принцип атомарности должен действовать даже в случае системной ошибки или сбоя питания. В данном случае «атомарность» следует понимать как «неделимость».

8.5.2. Согласованность

При завершении успешной транзакции должны соблюдаться все ограничения *согласованности* (целостности данных), определенные в системе. Примеры таких ограничений — уникальность первичных ключей, соответствие данных некоторой схеме и т. д. Транзакции, которые приводят к некорректному состоянию данных, обычно приводят к откату транзакций, хотя незначительные проблемы могут разрешаться автоматически (например, приведение данных к правильной форме). Этот вид согласованности не следует путать с согласованностью (C) в теореме CAP, которая относится к единству представления данных для всех пользователей распределенного хранилища.

8.5.3. Изоляция

Изолированные транзакции должны приводить к одинаковому результату независимо от их параллельного или последовательного применения. Уровень изоляции, предоставляемый системой, напрямую влияет на ее способность выполнять параллельные операции. Наивная схема изоляции основана на применении *глобальной блокировки*; вся база данных блокируется на время транзакции, в результате чего все транзакции фактически выполняются последовательно. Такая схема предоставляет сильные гарантии изоляции, но она патологически неэффективна: транзакции, работающие с полностью несвязанными наборами данных, блокируются без необходимости (например, добавление пользователем комментария в идеале не должно мешать обновлению профиля другим пользователем). На практике системы предоставляют разные уровни изоляции с применением схем блокировки с разной избирательностью (на уровне таблиц, строк или полей). Более сложные системы могут даже оптимистически пытаться проводить все транзакции с минимальной блокировкой, а затем заново повторять транзакции со снижением детализации в случае выявления конфликтов.

8.5.4. Устойчивость

Устойчивость транзакции определяет степень, в которой ее эффект гарантированно сохранится даже при перезапуске, сбое питания, системных ошибках и даже сбоях оборудования. Например, приложение, использующее SQLite в режиме работы в памяти, не обладает устойчивостью транзакций; все данные теряются при выходе из процесса. С другой стороны, SQLite в режиме записи данных на диск будет обладать хорошей транзакционной устойчивостью, потому что данные сохраняются даже после перезапуска машины.

Казалось бы, проще некуда: запишите данные на диск — вуаля, у вас появляются устойчивые транзакции. Но дисковый ввод/вывод является одной из самых медленных операций, выполняемых вашим приложением, и может быстро стать серьезным «узким местом» вашего приложения даже при умеренных уровнях масштабирования. Некоторые базы данных поддерживают разные степени компромисса по устойчивости, которые могут обеспечивать приемлемое быстродействие системы.

8.6. NoSQL

Хранилища данных, которые не вписываются в реляционную модель, объединяются под термином *NoSQL*. Так как в наши дни некоторые базы данных NoSQL поддерживают SQL, смысл термина NoSQL ближе к определению «нереляционный» или придуманному задним числом сокращению «Not Only SQL» («не только SQL»).

Подмножество парадигм и примеры баз данных, которые можно отнести к NoSQL:

- пары «ключ-значение»/кортежи — DynamoDB, LevelDB, Redis, etcd, Riak, Aerospike, Berkeley DB;
- графовые базы данных — Neo4J, OrientDB;
- документные базы данных — CouchDB, MongoDB, Elastic (formerly Elasticsearch);
- столбцовые базы данных — Cassandra, HBase;
- базы данных временных рядов — Graphite, InfluxDB, RRDtool;
- многопарадигменные базы данных — Couchbase (документная база данных, хранилище пар «ключ-значение», распределенный кэш).

За более подробным списком баз данных NoSQL обращайтесь по адресу *http://nosql-database.org/*.

Концепции NoSQL бывает трудно усвоить, если вы работали только с реляционными базами данных, потому что применение NoSQL часто противоречит установившейся практике: отсутствие определенных схем. Дублирование данных. Слабое соблюдение ограничений. Системы NoSQL берут на себя обязанности, обычно закрепляемые за базами данных, и выводят их в область ответственности приложения. На первый взгляд все это выглядит сомнительно.

Обычно небольшое количество схем доступа создает основную нагрузку на базу данных — например, запросы, генерирующие начальный экран приложения, для которого необходимо получить несколько объектов предметной области. Стандартным приемом повышения производительности чтения в реляционных базах данных является *нормализация* — запросы обрабатываются и приводятся к форме, сокращающей количество операций чтения, необходимых для потребления данных клиентом.

Данные NoSQL чаще денормализируются по умолчанию, и весь этап моделирования предметной области может быть полностью опущен. Такой подход предотвращает чрезмерное техническое усложнение модели данных, позволяет быстрее отрабатывать изменения и в целом приводит к более простой и производительной архитектуре.

8.7. Распределенные базы данных

Приложение может масштабироваться вертикально (за счет повышения производительности машин) или горизонтально (за счет добавления новых машин). Вертикальное масштабирование обычно проще реализуется, но возможности оборудования ограничивают возможности масштабирования одной машины. Кроме того, вертикальное масштабирование также быстро дорожает. Напротив,

горизонтальное масштабирование обладает намного большим потенциалом для роста при добавлении новых процессов и новых машин. За все это приходится платить сложностью управления большим количеством «подвижных частей». Все растущие системы со временем достигают точки, в которой приходится проводить горизонтальное масштабирование.

Распределенные базы данных с самого начала проектируются с расчетом на горизонтальное масштабирование. Хранение данных на нескольких машинах повышает устойчивость данных за счет устранения «единых точек сбоя». Многие реляционные системы в некоторой степени способны выполнять горизонтальное масштабирование в форме сегментации, репликации «главный/подчиненный» или «главный/главный», хотя даже с этими функциями реляционные системы не рассчитаны на масштабирование более чем для нескольких сотен узлов. Например, верхний предел кластера MySQL составляет 255 узлов. Распределенные базы данных, напротив, могут масштабироваться на тысячи узлов по самой архитектуре.

8.8. MongoDB

MongoDB — документно-ориентированная распределенная база данных, исключительно популярная среди разработчиков Node. Она стоит на первом месте в модном технологическом стеке MEAN (MongoDB, Express, Angular, Node) и часто становится одной из первых баз данных, с которой сталкиваются люди в начале работы с Node. На рис. 8.2 показано, насколько популярен модуль mongodb в npm.

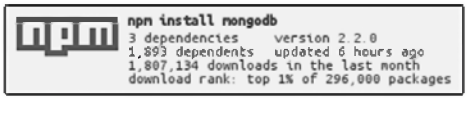


Рис. 8.2. Статистика использования MongoDB

MongoDB привлекает более чем справедливую долю критики; несмотря на это, MongoDB остается надежным хранилищем данных для многих разработчиков. Поддержка MongoDB развернута во многих видных компаниях, включая Adobe, LinkedIn и eBay, и даже используется в компоненте Большого адронного коллайдера в Европейском центре ядерных исследований (CERN).

В базе данных MongoDB документы хранятся в бессхемных коллекциях. Документ не обязан строиться по заранее определенной схеме, а документы одной коллекции не обязаны совместно использовать одну схему. Таким образом, MongoDB наделяется значительной гибкостью, хотя на приложение ложится бремя по поддержанию прогнозируемой структуры документа (гарантированная согласованность — «C» в ACID).

8.8.1. Установка и настройка

Пакет MongoDB должен быть установлен в вашей системе. Процесс установки отличается в зависимости от платформы.

В macOS установка сводится к простой команде:

```
brew install mongodb
```

Сервер MongoDB запускается исполняемым файлом mongod:

```
mongod --config /usr/local/etc/mongod.conf
```

Самым популярным драйвером MongoDB является официальный пакет mongodb, созданный Кристианом Амором Квалхеймом (Christian Amor Kvalheim):

```
npm install mongodb@^2.1.0 --save
```

Пользователям Windows следует учитывать, что для установки драйвера необходим файл msbuild.exe, устанавливаемый Microsoft Visual Studio.

8.8.2. Подключение к MongoDB

После установки пакета mongodb и запуска сервера mongod вы сможете подключиться в качестве клиента из Node:

Листинг 8.8. Подключение к MongoDB

```
const { MongoClient } = require('mongodb');
MongoClient.connect('mongodb://localhost:27017/articles')
  .then(db  => {
    console.log('Client ready');
    db.close();
  }, console.error);
```

Обработчику успешного завершения передается экземпляр клиента базы данных, из которого выполняются все команды базы данных.

Большинство взаимодействий с базой данных осуществляется через API коллекций:

- `collection.insert(doc)` — вставка одного или нескольких документов;
- `collection.find(query)` — поиск документов, соответствующих запросу;
- `collection.remove(query)` — удаление документов, соответствующих запросу;
- `collection.drop()` — удаление всей коллекции;
- `collection.update(query)` — обновление документов, соответствующих запросу;
- `collection.count(query)` — подсчет документов, соответствующих запросу.

Такие операции, как поиск, вставка и удаление, обычно существуют в нескольких разновидностях — в зависимости от того, работаете ли вы с одним или многими значениями. Примеры:

- `collection.insertOne(doc)` — вставка одного документа;
- `collection.insertMany([doc1, doc2])` — вставка нескольких документов;
- `collection.findOne(query)` — поиск одного документа, соответствующего запросу;
- `collection.updateMany(query)` — обновление всех документов, соответствующих запросу.

8.8.3. Вставка документов

`collection.insertOne` вставляет один объект в коллекцию как документ (листинг 8.9). Обработчику успешного завершения передается объект с метаданными, относящимися к состоянию операции.

Листинг 8.9. Вставка документа

```
const article = {
  title: 'I like cake',
  content: 'It is quite good.'
};
db.collection('articles')
  .insertOne(article)
  .then(result => {
    console.log(result.insertedId);
    console.log(article._id);  В
  });
```

Если у документа нет свойства _id, создается новый идентификатор. Сгенерированное значение хранится в insertId.

Исходный объект, определяющий документ, изменяется: в него добавляется поле _id.

Вызов `insertMany` работает аналогично, если не считать того, что он получает массив из нескольких документов. Ответ `insertMany` будет содержать массив `insertedIds` в порядке передачи документов вместо одного значения `insertedId`.

8.8.4. Получение информации

Методы, читающие документы из коллекции (такие, как `find`, `update` и `remove`), получают аргумент, используемый для идентификации документов. Простейшая форма аргумента запроса представляет собой объект, который используется MongoDB для подбора документов с такой же структурой и теми же значениями. Например, следующий фрагмент находит все статьи с заголовком «I like cake»:

```
db.collection('articles')
  .find({ title: 'I like cake' })
```

```
  .toArray().then(results => {
    console.log(results);
  });
```

Массив документов, соответствующих запросу.

Запросы используются для поиска объектов по их уникальному значению `_id`:

```
collection.findOne({ _id: someID })
```

Или для поиска с применением *оператора запроса*:

```
db.collection('articles')
  .find({title: { $regex:  /cake$/I })
```

Заголовок заканчивается словом 'cake' (без учета регистра символов).

В языке запросов MongoDB поддерживаются разные операторы запросов. Например:

- `$eq` — равно заданному значению;
- `$neq` — не равно заданному значению;
- `$in` — присутствует в массиве;
- `$nin` — не присутствует в массиве;
- `$lt`, `$lte`, `$gt`, `$gte` — больше/меньше или равно;
- `$near` — геопространственное значение расположено вблизи заданных координат;
- `$not`, `$and`, `$or`, `$nor` — логические операторы.

Эти значения могут объединяться для поиска практически по любым условиям. Так создается в высшей степени понятный, сложный и выразительный язык запросов. За дополнительной информацией о запросах и операторах запросов обращайтесь по адресу *https://docs.mongodb.com/manual/reference/operator/query/*.

В листинге 8.10 приведен пример реализации приведенного выше API `Articles` с MongoDB; при этом внешний интерфейс остается практически идентичным. Сохраните файл под именем db.js (файл listing8_10/db.js в исходном коде книги).

Листинг 8.10. Реализация API Article с MongoDB

```
const { MongoClient, ObjectID } = require('mongodb');

let db;

module.exports = () => {
  return MongoClient
    .connect('mongodb://localhost:27017/articles')
    .then((client) => {
      db = client;
    });
};

module.exports.Article = {
  all() {
    return db.collection('articles2').find().sort({ title: 1 }).toArray();
```

```
  },

  find(_id) {
    if (typeof _id !== 'object') _id = ObjectID(_id);
    return db.collection('articles2').findOne({ _id });
  },

  create(data) {
    return db.collection('articles2').insertOne(data, { w: 1 });
  },

  delete(_id) {
    if (typeof _id !== 'object') _id = ObjectID(_id);
    return db.collection('articles2').deleteOne({ _id }, { w: 1 });
  }
};
```

Добавляет поддержку передачи _id в формате String и ObjectID.

Следующий фрагмент показывает, как использовать листинг 8.10 (listing 8_10/index.js в примере кода):

```
const db = require('./db');
db().then(() => {
  db.Article.create({ title: 'An article!' }).then(() => {
    db.Article.all().then(articles => {
      console.log(articles);
      process.exit();
    });
  });
});
```

В этом фрагменте обещание из листинга 8.10 используется для подключения к базе данных, после чего объект создается методом `create` класса `Article`. После этого фрагмент загружает все статьи и выводит их на консоль.

8.8.5. Идентификаторы MongoDB

Идентификаторы MongoDB кодируются в формате BSON (Binary JSON). Свойство `_id` документа представляет собой объект JavaScript `Object`, инкапсулирующий значение `ObjectID` в формате BSON. MongoDB использует BSON для внутреннего представления документов и как формат передачи данных. Формат BSON более компактен, чем JSON, и быстрее разбирается; это означает ускорение операций с базой данных с меньшими затратами ресурсов.

Значение BSON `ObjectID` — не простая последовательность байтов; в нем закодирована информация о том, где и как был сгенерирован идентификатор. Например, первые четыре байта `ObjectID` содержат временную метку, и вам не придется включать свойство `createdAt` в свои документы:

```
const id = new ObjectID(61bd7f57bf1532835dd6174b);
id.getTimestamp();
```

getTimestamp возвращает JavaScript / Date: 2016-07-08T14:49:05.000Z.

За дополнительной информацией о формате `ObjectID` обращайтесь по адресу *https://docs.mongodb.com/manual/reference/method/ObjectId/*.

Может показаться, что объекты `ObjectID` похожи на строки из-за особенностей их вывода на терминал; тем не менее они являются объектами. Объекты `ObjectID` подвержены классической проблеме сравнения объектов: полностью эквивалентные на первый взгляд значения считаются неэквивалентными, потому что они относятся к разным объектам.

В следующем фрагменте одно значение извлекается дважды. Затем при помощи встроенного модуля Node assert мы пытаемся проверить на эквивалентность объекты и их идентификаторы, но в обоих случаях проверка дает отрицательный результат:

```
const Articles = db.collection('articles');
Articles.find().then(articles => {
  const article1 = articles[0];
  return Articles
    .findOne({_id: article1._id})
    .then(article2 => {
      assert.equal(article2._id, article1._id);
    });
});
```

Для этих проверок выдаются сообщения об ошибке, которые сначала кажутся непонятными, так как фактические значения на первый взгляд совпадают с ожидаемыми:

```
operator: equal
expected: 577f6b45549a3b991e1c3c18
actual:   577f6b45549a3b991e1c3c18
operator: equal
expected:
  { _id: 577f6b45549a3b991e1c3c18, title: 'attractive-money' ... }
actual:
  { _id: 577f6b45549a3b991e1c3c18, title: 'attractive-money' ... }
```

Для правильной проверки эквивалентности следует использовать метод `equal` объектов `ObjectID`, доступный для всех `_id`. Также можно выполнить приведение типа идентификаторов и сравнить их как строки или же воспользоваться методом `deepEquals` — вроде того, который доступен во встроенном модуле Node assert:

```
article1._id.equals(article2._id);
String(article1._id) === String(article2._id);
assert.deepEqual(article1._id, article2._id);
```

Выдает исключение, если условие не выполняется.

Идентификаторы, переданные драйверу Node mongodb, должны представлять собой `ObjectID` в формате BSON. Строка преобразуется в `ObjectID` при помощи конструктора `ObjectID`:

```
const { ObjectID } = require('mongodb');
const stringID = '577f6b45549a3b991e1c3c18';
const bsonID = new ObjectID(stringID);
```

Там, где это возможно, следует поддерживать формат BSON; затраты на маршализацию в строковую форму и обратно снижают выигрыш по быстродействию, которого MongoDB стремится достичь за счет работы с клиентскими идентификаторами в формате BSON. За дополнительной информацией о формате BSON обращайтесь по адресу *http://bsonspec.org/*.

8.8.6. Реплицированные наборы

Распределенная функциональность MongoDB в основном выходит за рамки книги, но в этом разделе будут кратко представлены основы работы с реплицированными наборами. Многие процессы mongod могут выполняться как узлы/участники *реплицированного набора* (replica set). Реплицированный набор состоит из одного первичного узла и множества вторичных узлов. Каждому узлу реплицированного набора должен быть назначен уникальный порт и каталог для хранения данных. Экземпляры не могут совместно использовать порты или каталоги, а каталоги должны существовать до запуска.

В листинге 8.11 для каждого участника создается уникальный каталог, а участники запускаются на портах с последовательно увеличивающимися номерами, начиная с 27017. Возможно, вам стоит выполнять каждую команду mongod в новой вкладке терминала без выполнения в фоновом режиме (без завершающего символа `&`).

Листинг 8.11. Запуск реплицированного набора

```
mkdir -p ./mongodata/db0 ./mongodata/db1 ./mongodata/db2

pkill mongod  <-------  Гарантирует отсутствие других выполняемых экземпляров mongod.
sleep 3  <-------  Дает существующим экземплярам время на завершение.

mongod --port 27017 --dbpath ./rs0-data/db0 --replSet rs0 &
mongod --port 27018 --dbpath ./rs0-data/db1 --replSet rs0 &
mongod --port 27019 --dbpath ./rs0-data/db2 --replSet rs0 &
```

После того как реплицированный набор заработает, MongoDB необходимо провести некоторую инициализацию. Необходимо подключиться к порту экземпляра, который станет первым *первичным* узлом (27017 по умолчанию), и вызвать метод `rs.initiate()`, как показано в листинге 8.12. Затем необходимо добавить каждый экземпляр как участника реплицированного набора. Обратите внимание на необходимость передачи имени хоста той машины, к которой вы подключаетесь.

Листинг 8.12. Инициализация реплицированного набора

```
mongo --eval "rs.initiate()"
mongo --eval "rs.add(``hostname`:27017')"
mongo --eval "rs.add(``hostname`:27018')"
mongo --eval "rs.add(``hostname`:27019')"
```

Команда UNIX hostname выводит имя хоста текущей машины.

Клиенты MongoDB должны располагать информацией обо всех возможных участниках реплицированного набора при подключении, хотя не все участники должны быть подключены в текущий момент. После подключения вы можете использовать клиента MongoDB, как обычно. В листинге 8.13 показано, как создать реплицированный набор с тремя участниками.

Листинг 8.13. Подключение к реплицированному набору

```
const os = require('os');
const { MongoClient } = require('mongodb');
const hostname = os.hostname();

const members = [
  `${hostname}:27018`,
  `${hostname}:27017`,
  `${hostname}:27019`
];

MongoClient.connect(`mongodb://${members.join(',')}/test?replSet=rs0`)  ///
.then(db => {
  db.admin().replSetGetStatus().then(status => {  ///
    console.log(status);
    db.close();
  });
});
```

Если на любом из узлов mongod произойдет сбой, система продолжит работать (при условии выполнения по крайней мере двух экземпляров). Если сбой произойдет на первичном узле, то вторичный узел будет автоматически повышен до первичного.

8.8.7. Уровень записи

MongoDB предоставляет в распоряжение разработчика средства точного управления компромиссами по быстродействию и безопасности, доступные для разных частей вашего приложения. Чтобы использовать MongoDB без неприятных сюрпризов, важно понимать концепции уровней записи и чтения — особенно с увеличением количества узлов в реплицированном наборе. В этом разделе рассматривается только концепция значимости записи, так как она является более важной.

Уровень записи (write concern) определяет количество экземпляров mongod, в которые должна быть успешно выполнена запись, чтобы вся операция была признана

успешной. Если значение не задано явно, по умолчанию уровень записи равен 1; он гарантирует, что данные были записаны по крайней мере в один узел. Может оказаться, что это значение не обеспечивает необходимого уровня защиты критических данных; если узел отключится до того, как данные будут реплицированы на других узлах, данные могут быть потеряны.

Можно (и даже желательно) установить нулевой уровень записи, при котором приложение вообще не ожидает никакого ответа:

```
db.collection('data').insertOne(data, { w: 0 });
```

Нулевой уровень записи обеспечивает наивысшее быстродействие при минимальных гарантиях устойчивости. Обычно он применяется только для временных или некритичных данных (например, для записи журналов или кэширования).

Если вы подключены к реплицированному набору, можно установить уровень записи выше 1. Репликация по большему количеству узлов снижает вероятность потери данных за счет появления дополнительной задержки при выполнении операций:

```
db.collection('data').insertOne(data, { w: 2 });
db.collection('data').insertOne(data, { w: 5 });
```

Уровень записи желательно масштабировать при изменении количества узлов в кластере. Эта задача может решаться динамически MongoDB, если выбрать для уровня записи значение `majority`. Оно гарантирует, что данные будут записаны более чем в 50% доступных узлов:

```
db.collection('data').insertOne(data, { w: 'majority' });
```

Уровень записи по умолчанию, равный 1, может не обеспечивать адекватного уровня защиты для критических данных. Данные могут быть потеряны, если узел отключится перед репликацией на других узлах.

При назначении уровня записи выше 1 выполнение продолжится только после проверки того, что данные существуют на нескольких экземплярах mongod. Выполнение нескольких экземпляров на одной машине повышает уровень защиты, но не помогает в случае общесистемных сбоев вроде нехватки дискового пространства или ОЗУ. Чтобы защититься от машинных сбоев, следует запустить экземпляры на нескольких машинах и следить за тем, чтобы операции записи распространились по этим узлам; однако выполнение записи при этом замедлится, и этот режим не защитит от сбоев на уровне центров обработки данных. Распределение узлов по нескольким центрам обработки данных обеспечивает защиту в случае выхода центров из строя, но обеспечение репликации этих данных между центрами обработки данных серьезно повлияет на быстродействие.

Как обычно, чем больше защиты вы добавляете, тем медленнее и сложнее становится ваша система. Эта проблема не является специфической только для MongoDB; она присуща всем хранилищам данных без исключения. Идеального решения не существует, и вам придется выбрать приемлемый уровень риска для различных частей вашего приложения.

За дополнительной информацией о том, как работает репликация в MongoDB, обращайтесь к следующим ресурсам:

- *https://docs.mongodb.com/manual/faq/replica-sets/*;
- *https://docs.mongodb.com/manual/faq/concurrency/*.

8.9. Хранилища «ключ-значение»

Каждая запись в хранилище «ключ-значение» состоит из одного ключа и одного значения. Во многих системах «ключ-значение» значение может относиться к любому типу данных, иметь любую длину или структуру. С точки зрения базы данных значения являются непрозрачными и атомарными: базе данных неизвестен тип данных, и к этим данным можно обращаться только как к единому целому без разбиения на части. Сравните с хранением значений в реляционных базах данных: данные хранятся в таблицах, которые содержат строки данных, разбитые на заранее определенные столбцы. В хранилищах «ключ-значение» ответственность за управление форматом данных возлагается на приложение.

Хранилища «ключ-значение» часто встречаются в частях приложения, критичных по быстродействию. В идеале значения располагаются в такую структуру, чтобы для выполнения задачи требовалось минимальное количество операций чтения. Хранилища «ключ-значение» обладают более простой функциональностью запроса, чем другие типы баз данных. В идеале сложные запросы должны обрабатываться заранее; в противном случае они должны выполняться в приложении, а не в базе данных. Это ограничение должно обеспечивать понятные и предсказуемые характеристики быстродействия.

Самые популярные хранилища «ключ-значение» — такие, как Redis и Memcached, — часто используются для временного хранения данных (при выходе из процесса данные теряются). Отказ от записи на диск — один из лучших способов повышения быстродействия. Это решение может стать приемлемым для функций, позволяющих восстановить или потерять данные без особых проблем (например, для кэширования и пользовательских сеансов).

Обычно считается, что хранилища «ключ-значение» не могут использоваться для первичного хранения данных, но это не всегда так. Многие хранилища «ключ-значение» обеспечивают такую же устойчивость, как и «настоящие» базы данных.

8.10. Redis

Redis — популярное хранилище структур данных в памяти. И хотя многие разработчики считают Redis хранилищем «ключ-значение», ключи и значения составляют лишь небольшое подмножество функций Redis среди разнообразных полезных базовых структур данных. На рис. 8.3 приведена статистика использования redis в npm.

Рис. 8.3. Статистика использования Redis

К числу структур данных, встроенных в Redis, относятся:

- строки;
- хеши;
- списки;
- множества;
- сортированные множества.

Redis также включает в себя много других полезных возможностей:

- *растровые данные* — прямые манипуляции с битами в значениях;
- *геопространственные индексы* — хранение геопространственных данных с радиальными запросами;
- *каналы* — механизм поставки данных «публикация/подписка»;
- *срок жизни (TTL)* — значения могут настраиваться со сроком жизни, по истечении которого они автоматически уничтожаются;
- *вытеснение LRU* — (необязательное) вытеснение давно не использовавшихся значений для максимально эффективного использования памяти;
- *HyperLogLog* — высокопроизводительная аппроксимация для вычисления мощности множества при небольших затратах памяти (без необходимости хранения каждого элемента);
- *репликация, кластеризация и сегментация* — горизонтальное масштабирование и устойчивость данных;
- *сценарии Lua* — расширение Redis нестандартными командами.

В этом разделе приводятся списки команд Redis. Не стоит относиться к ним как к справочникам; они дают лишь некоторое представление о том, что можно делать

в Redis. Redis — невероятно мощный и гибкий инструмент; за подробностями обращайтесь по адресу *http://redis.io/commands*.

8.10.1. Установка и настройка

Redis может устанавливаться при помощи системного менеджера пакетов. В macOS Redis легко устанавливается в Homebrew:

```
brew install redis
```

Для запуска сервера используется исполняемый файл redis-server:

```
redis-server /usr/local/etc/redis.conf
```

По умолчанию сервер выполняет прослушивание на порту 6379.

8.10.2. Выполнение инициализации

Экземпляр клиента Redis создается функцией `createClient` из npm-пакета redis:

```
const redis = require('redis');
const db = redis.createClient(6379, '127.0.0.1');
```

В аргументах функции передаются порт и хост. Но если вы запускаете сервер Redis на порту по умолчанию локальной машины, никаких аргументов передавать вообще не придется:

```
const db = redis.createClient();
```

Экземпляр клиента Redis расширяет `EventEmitter`, поэтому вы можете присоединять слушателей для различных статусных событий Redis, как показано в листинге 8.14. Вы можете сразу начинать вводить команды для клиента; введенные команды буферизуются до готовности подключения.

Листинг 8.14. Подключение к Redis и прослушивание статусных событий

```
const redis = require('redis');
const db = redis.createClient();
db.on('connect', () => console.log('Redis client connected to server.'));
db.on('ready', () => console.log('Redis server is ready.'));
db.on('error', err => console.error('Redis error', err));
```

Обработчик `error` срабатывает при возникновении проблемы с подключением или клиентом. Если при срабатывании события `error` обработчик не присоединен, процесс приложения выдает ошибку и аварийно завершается; это свойство присуще всем объектам `EventEmitter` в Node. Если при подключении происходит сбой, а обработчик `error` определен, то клиент Redis пытается восстановить подключение.

8.10.3. Работа с парами «ключ-значение»

Redis может использоваться в качестве обобщенного хранилища «ключ-значение» для строк и произвольных двоичных данных. Для чтения и записи пар «ключ-значение» используются методы `set` и `get` соответственно:

```
db.set('color', 'red', err => {
  if (err) throw err;
});
db.get('color', (err, value) => {
  if (err) throw err;
  console.log('Got:', value);
});
```

Если указать существующий ключ, значение будет заменено. Если вы попытаетесь прочитать значение с несуществующим ключом, значение будет равно `null`; такое обращение не считается ошибкой.

Следующие команды могут использоваться для чтения и изменения значений:

- `append`;
- `decr`;
- `decrby`;
- `get`;
- `getrange`;
- `getset`;
- `incr`;
- `incrby`;
- `incrbyfloat`;
- `mget`;
- `mset`;
- `msetnx`;
- `psetex`;
- `set`;
- `setex`;
- `setnx`;
- `setrange`;
- `strlen`.

8.10.4. Работа с ключами

Чтобы проверить, существует ли ключ, используйте метод `exists`. Он работает с любым типом данных:

```
db.exists('users', (err, doesExist) => {
  if (err) throw err;
      console.log('users exists:', doesExist);
});
```

Кроме `exists`, для работы с ключами могут использоваться следующие команды (независимо от типа значения — команды работают со строками, множествами, списками и т. д.):

- `del`;
- `exists`;
- `rename`;
- `renamenx`;
- `sort`;
- `scan`;
- `type`.

8.10.5. Кодирование и типы данных

Сервер Redis хранит ключи и значения в виде двоичных объектов; это представление не зависит от способа кодирования значений, передаваемых клиенту. Любая допустимая строка JavaScript (UCS2/UTF16) может использоваться как действительный ключ или значение:

```
db.set('greeting', '你好', redis.print);
db.get('greeting', redis.print);
db.set('icon', '?', redis.print);
db.get('icon', redis.print);
```

По умолчанию ключи и значения преобразуются в строки при записи. Например, если задать ключ в числовом виде, при попытке получения того же ключа вы получите строку:

```
db.set('colors', 1, (err) => {
  if (err) throw err;
});
db.get('colors', (err, value) => {
  if (err) throw err;
  console.log('Got: %s as %s', value, typeof value);
});
```

Клиент Redis незаметно преобразует числа, логические значения и даты в строки; кроме того, он спокойно принимает объекты буферов. Попытка задать в качестве значения любой другой тип JavaScript (например, `Object`, `Array`, `RegExp`) приводит к выдаче предупреждения, к которому стоит прислушаться:

```
db.set('users', {}, redis.print);
Deprecated: The SET command contains a argument of type Object.
This is converted to "[object Object]" by using .toString() now
  and will return an error from v.3.0 on.
Please handle this in your code to make sure everything works
  as you intended it to.
```

В будущем это будет рассматриваться как ошибка. Приложение должно проследить за тем, чтобы клиенту Redis передавались допустимые типы.

Массивы с одним и несколькими значениями

При попытке присваивания массива значений клиент выдает загадочную ошибку «ReplyError: ERR syntax error»:

```
db.set('users', ['Alice', 'Bob'], redis.print);
```

Однако для массива, содержащего только одно значение, ошибки не будет:

```
db.set('user', ['Alice'], redis.print);
db.get('user', redis.print);
```

Такие ошибки нередко проявляются только в условиях реальной эксплуатации. Они остаются незамеченными, если тестовый пакет генерирует только массивы с одним значением, типичные для усеченных тестовых данных. Будьте внимательны!

Буферы с двоичными данными

Redis позволяет хранить произвольные байтовые данные; фактически это означает возможность сохранения любых типов данных. Клиент Node поддерживает эту возможность за счет специальной обработки типа Node `Buffer`. При передаче буфера клиенту Redis в качестве ключа или значения байты передаются в неизменном виде серверу Redis. Таким образом предотвращается случайное повреждение данных и снижение быстродействия из-за лишних преобразований между строками и буферами. Например, если вы захотите записать данные с диска или из сети прямо в Redis, непосредственная запись буферов в Redis выполняется более эффективно, чем с предварительным преобразованием их в строки.

В Redis недавно были добавлены команды для манипуляций с отдельными битами строковых значений, которые могут пригодиться при работе с буферами:

- `bitcount`;
- `bitfield`;

- `bitop`;
- `setbit`;
- `bitpos`.

БУФЕРЫ

Буферы — данные, которые возвращаются по умолчанию базовыми API Node для работы с файлами и сетями. Они представляют собой контейнеры для смежных блоков двоичных данных. Поддержка буферов появилась в Node еще до того, как в JavaScript появились встроенные типы для работы с двоичными данными (Uint8Array, Float32Array и т. д). В наши дни буферы реализуются в Node специализированным субклассом Uint8Array. API для работы с буферами доступен в Node глобально; чтобы пользоваться им, вам не придется что-то делать дополнительно.

См. https://github.com/nodejs/node/blob/master/lib/buffer.js.

8.10.6. Работа с хешами

Хеш представляет собой коллекцию пар «ключ-значение». Команда `hmset` получает ключ и объект, представляющий набор пары «ключ-значение» в хеше. Для получения пар «ключ-значение» в виде объекта используется команда `hmget` (листинг 8.15).

Листинг 8.15. Хранение данных в элементах хешей Redis

```
db.hmset('camping', {                          <-- Задает пары «ключ-значение», входящие в хеш.
  shelter: '2-person tent',
  cooking: 'campstove'
}, redis.print);

db.hget('camping', 'cooking', (err, value) => {   <-- Получает значение для «camping.cooking».
  if (err) throw err;
  console.log('Will be cooking with:', value);
});

db.hkeys('camping', (err, keys) => {           <-- Получает ключи хеша в виде массива.
  if (err) throw err;
  keys.forEach(key => console.log(`  ${key}`));
});
```

Не допускается хранение вложенных объектов в хеше Redis — поддерживается только один уровень ключей и значений.

Для работы с хешами используются следующие команды:

- `hdel`;
- `hexists`;

- `hget`;
- `hgetall`;
- `hincrby`;
- `hincrbyfloat`;
- `hkeys`;
- `hlen`;
- `hmget`;
- `hmset`;
- `hset`;
- `hsetnx`;
- `hstrlen`;
- `hvals`;
- `hscan`.

8.10.7. Работа со списками

Список представляет собой упорядоченную коллекцию строковых значений. Список может содержать несколько копий одного значения. На концептуальном уровне списки близки к массивам. Чаще всего списки используются за их способность моделировать поведение структур данных стека (LIFO: «последним пришел, первым вышел») или очереди (FIFO: «первым пришел, первым вышел»).

Следующий фрагмент демонстрирует сохранение и выборку значений из списка. Команда `lpush` добавляет значение в список. Команда `lrange` извлекает диапазон значений по начальному и конечному индексу. Аргумент `-1` в следующем фрагменте обозначает последний элемент списка, поэтому в этом варианте использования `lrange` извлекаются 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, (err, items) => {
  if (err) throw err;
  items.forEach(item => console.log(`  ${item}`));
});
```

Списки не содержат встроенных средств проверки присутствия значения в списке или определения индекса конкретного значения в списке. Вы можете вручную перебрать элементы списка для получения нужной информации, но это крайне неэффективное решение, которого следует избегать. Если вам нужна функциональность такого рода, лучше выбрать другую структуру данных (например, множество) — а возможно, даже использовать ее в дополнение к списку. Дублирование данных

между несколькими структурами часто желательно для использования различных характеристик быстродействия.

Для работы со списками предназначены следующие команды:

- `blpop`;
- `brpop`;
- `lindex`;
- `linsert`;
- `llen`;
- `lpop`;
- `lpush`;
- `lpushx`;
- `lrange`;
- `lrem`;
- `lset`;
- `ltrim`;
- `rpop`;
- `rpush`;
- `rpushx`.

8.10.8. Работа со множествами

Множество представляет собой неупорядоченную коллекцию уникальных значений. Проверка принадлежности, а также операции добавления и удаления элементов из множества выполняются за время O(1); это означает, что множество является высокопроизводительной структурой, подходящей для многих задач:

```
db.sadd('admins', 'Alice', redis.print);
db.sadd('admins', 'Bob', redis.print);
db.sadd('admins', 'Alice', redis.print);
db.smembers('admins', (err, members) => {
  if (err) throw err;
  console.log(members);
});
```

Для работы с множествами Redis предназначены следующие команды:

- `sadd`;
- `scard`;
- `sdiff`;

- `sdiffstore`;
- `sinter`;
- `sinterstore`;
- `sismember`;
- `smembers`;
- `spop`;
- `srandmember`;
- `srem`;
- `sunion`;
- `sunionstore`;
- `sscan`.

8.10.9. Реализация паттерна «публикация/подписка» на базе каналов

Redis выходит за рамки традиционной роли хранилища данных, предоставляя в распоряжение разработчика *каналы* (channels). Этот механизм передачи данных реализует функциональность публикации/подписки; его концептуальная схема изображена на рис. 8.4. Каналы хорошо подходят для приложений реального времени — таких, как чаты и игры.

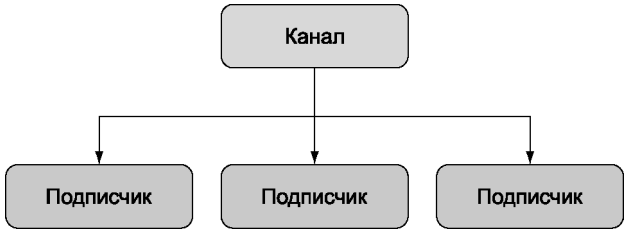


Рис. 8.4. Каналы Redis предоставляют простое решение для стандартного сценария передачи данных

Клиент Redis может подписываться на каналы или публиковать в них сообщения. Сообщение, опубликованное в канале, будет доставлено всем подписчикам. Публикатору не обязательно располагать информацией о подписчиках, как и подписчикам — о публикаторах. Именно логическая изоляция между подписчиками и публикаторами делает этот паттерн столь мощным и элегантным.

В листинге 8.16 приведен пример использования функциональности публикации/подписки в Redis для реализации чат-сервера TCP/IP.

Листинг 8.16. Реализация простого чат-сервера на базе функциональности публикации/подписки в Redis

```
const net = require('net');
const redis = require('redis');

const server = net.createServer(socket => {
  const subscriber = redis.createClient();
  subscriber.subscribe('main');
  subscriber.on('message', (channel, message) => {
    socket.write(`Channel ${channel}: ${message}`);
  });

  const publisher = redis.createClient();
  socket.on('data', data => {
    publisher.publish('main', data);
  });

  socket.on('end', () => {
    subscriber.unsubscribe('main');
    subscriber.end(true);
    publisher.end(true);
  });
});

server.listen(3000);
```

Определяет логику настройки для каждого пользователя, подключающегося к чат-серверу.

Создает клиента подписки для каждого пользователя.

Подписывается на канал.

Когда сообщение принимается из канала, оно выводится для пользователя.

Создает клиента публикации для каждого пользователя.

Публикует сообщение, введенное пользователем.

Если пользователь отключается, то подключение клиента завершается.

8.10.10. Повышение быстродействия Redis

Npm-пакет hiredis связывает JavaScript с парсером протокола из официальной библиотеки C Hiredis. Hiredis может значительно повысить быстродействие приложений Redis на базе Node, особенно если вы используете операции `sunion`, `sinter`, `lrange` и `zrange` с большими базами данных.

Чтобы использовать hiredis, просто установите пакет вместе с пакетом redis в вашем приложении; пакет Node redis обнаружит его и автоматически использует его при следующем запуске:

```
npm install hiredis --save
```

У hiredis есть свои недостатки. Так как пакет компилируется из кода C, при построении hiredis для некоторых платформ вы можете столкнуться с осложнениями или ограничениями. Как и со всеми встроенными дополнениями, возможно, вам придется заново построить hiredis командой `npm rebuild` после обновления Node.

8.11. Встроенные базы данных

Встроенная база данных не требует установки или администрирования внешних серверов. Она работает прямо *внутри* процесса приложения. Взаимодействие со

встроенной базой данных обычно выполняется через прямые вызовы процедур в вашем приложении, а не через каналы IPC или по сети.

Во многих ситуациях приложение должно быть автономным, поэтому встроенная база данных становится единственно возможным выходом (например, для мобильных или настольных приложений). Встроенные базы данных также могут использоваться с веб-серверами; они часто обеспечивают реализацию таких функций, как сеансы пользователей или кэширование, а иногда даже используются в качестве первичного хранилища данных.

Некоторые встроенные базы данных, часто используемые в приложениях Node и Electron:

- SQLite;
- LevelDB;
- RocksDB;
- Aerospike;
- EJDB;
- NeDB;
- LokiJS;
- Lowdb.

NeDB, LokiJS и Lowdb написаны на «чистом» JavaScript и поэтому естественным образом встраиваются в приложения Node/Electron. Многие встроенные базы данных представляют собой простые хранилища пар «ключ-значение» или документов, хотя встроенное *реляционное* хранилище SQLite является заметным исключением.

8.12. LevelDB

LevelDB — встраиваемое хранилище «ключ-значение», разработанное в начале 2011 года компанией Google и изначально предназначенное для использования в качестве вспомогательного хранилища реализации IndexedDB в Chrome. Архитектура LevelDB строится на концепциях базы данных Bigtable компании Google. LevelDB можно сравнить с такими базами данных, как Berkeley DB, Tokyo/Kyoto Cabinet и Aerospike, но в контексте этой книги LevelDB можно рассматривать как встроенную версию Redis с минимальным набором функций. Как и многие встроенные базы данных, LevelDB не является многопоточной системой и не поддерживает использование несколькими экземплярами общего файлового хранилища, поэтому эта система не будет работать в распределенной среде без приложения-обертки.

LevelDB хранит произвольные байтовые массивы, отсортированные в лексикографическом порядке по ключу. Значения сжимаются с использованием алгоритма

Snappy компании Google. Данные всегда сохраняются на диске; общая емкость данных не ограничивается объемом оперативной памяти на компьютере (в отличие от систем, хранящих данные в памяти, таких как Redis).

LevelDB поддерживает небольшой набор понятных операторов: Get, Put, Del и Batch. LevelDB также умеет сохранять «снимки» текущего состояния базы данных и создавать двусторонние итераторы для перебора данных в прямом и обратном направлении. При создании итератора неявно создается снимок данных; данные, которые видны итератору, не могут изменяться последующими операциями записи.

LevelDB закладывает фундамент для других баз данных. Количество заслуживающих внимания ответвлений LevelDB объясняется простотой системы LevelDB:

- RocksDB (Facebook);
- HyperLevelDB (Hyperdex);
- Riak (Basho);
- leveldb-mcpe (Mojang, создатели Minecraft);
- bitcoin/leveldb (для проекта bitcoind).

За дополнительной информацией о LevelDB обращайтесь по адресу *http://leveldb.org/*.

8.12.1. LevelUP и LevelDOWN

Поддержка LevelDB в Node предоставляется пакетами LevelUP и LevelDOWN, которые написал один из основателей Node, активный разработчик из Австралии Род Вэгг (Rod Vagg). LevelDOWN — простая, минимальная прослойка C++ для LevelDB в Node; вряд ли вам придется взаимодействовать с ней напрямую. LevelUP инкапсулирует LevelDOWN API в более удобном и идиоматичном интерфейсе Node, добавляя поддержку кодирования ключей и значений, JSON, буферизации записи до открытия базы данных и инкапсуляции интерфейса итераторов LevelDB в потоках Node. Популярность LevelUP в npm продемонстрирована на рис. 8.5.

Рис. 8.5. Статистика использования LevelUP в npm

8.12.2. Установка

Главное достоинство использования LevelDB в приложениях Node — встроенная природа базы данных: все необходимое устанавливается исключительно из npm.

Вам не придется устанавливать дополнительное ПО; просто введите следующую команду, и все будет готово к использованию LevelDB:

```
npm install level --save
```

Пакет level представляет собой простую вспомогательную обертку для пакетов LevelUP и LevelDOWN, предоставляющую API LevelUP для использования внутренней подсистемы LevelDown. Документация API LevelUP, предоставляемого пакетом level, находится в Readme-файле LevelUP:

- *www.npmjs.com/package/levelup*;
- *www.npmjs.com/package/leveldown*.

8.12.3. Обзор API

Основные методы клиента LevelDB для хранения и чтения значений:

- `db.put(key, value, callback)` — сохраняет значение с заданным ключом;
- `db.get(key, callback)` — читает значение с заданным ключом;
- `db.del(key, callback)` — удаляет значение с заданным ключом;
- `db.batch().write()` — выполняет пакетные операции;
- `db.createKeyStream(options)` — создает поток ключей из базы данных;
- `db.createValueStream(options)` — создает поток значений из базы данных.

8.12.4. Инициализация

При инициализации level необходимо предоставить путь к каталогу, в котором будут храниться данные, как показано ниже; если каталог не существует, то он будет создан. В сообществе существует неформальное правило: присваивать такому каталогу расширение .db (например, ./app.db).

Листинг 8.17. Инициализация базы данных level

```
const level = require('level');
const db = level('./app.db', {
  valueEncoding: 'json'
});
```

После вызова `level()` возвращаемый экземпляр LevelUP немедленно готов к синхронному получению команд. Команды, введенные перед открытием хранилища LevelDB, буферизуются до его открытия.

8.12.5. Кодирование ключей и значений

Так как LevelDB может использовать произвольные данные любого типа как в ключах, так и в значениях, за сериализацию и десериализацию вызывающих данных отвечает приложение. LevelUp можно настроить для автоматического кодирования ключей и значений следующих типов данных:

- `utf8`;
- `json`;
- `binary`;
- `id`;
- `hex`;
- `ascii`;
- `base64`;
- `ucs2`;
- `utf16le`.

По умолчанию ключи и значения кодируются в строках UTF-8. В листинге 8.17 ключи остаются строками UTF-8, а значения кодируются/декодируются в формате JSON. Кодирование JSON позволяет хранить и читать структурированные значения (например, объекты или массивы) по аналогии с хранилищами документов (такими, как MongoDB). Но учтите, что в отличие от реальных хранилищ документов в базовой версии LevelDB не существует возможности обратиться к ключам внутри значений; значения непрозрачны. Пользователи также могут определять собственные кодировки — например, для поддержки другого структурированного формата (например, `MessagePack`).

8.12.6. Чтение и запись пар «ключ-значение»

Базовый API прост: вызов `put(key, value)` используется для записи значения, `get(key)` — для чтения и `del(key)` — для удаления значения (листинг 8.18). Код в листинге 8.18 следует присоединить к коду из листинга 8.17; полный пример содержится в файле ch08-data-bases/listing8_18/index.js в архиве исходного кода книги.

Листинг 8.18. Чтение и запись значений

```
const key = 'user';
const value = {
  name: 'Alice'
};

db.put(key, value, err => {
```

```
  if (err) throw err;
  db.get(key, (err, result) => {
    if (err) throw err;
    console.log('got value:', result);
    db.del(key, (err) => {
      if (err) throw err;
      console.log('value was deleted');
    });
  });
});
```

Если сохранить значение с уже существующим ключом, старое значение будет перезаписано. Попытка чтения значения с несуществующим ключом приведет к ошибке. Объект ошибки будет относиться к конкретному типу `NotFoundError` со специальным свойством `err.notFound`, по которому его можно отличить от других типов ошибок. На первый взгляд это необычно, но поскольку в LevelDB нет встроенного метода для проверки существования, LevelUP необходимо как-то отличать несуществующие значения от неопределенных. В отличие от `get`, попытка вызвать `del` с несуществующим ключом не приводит к ошибке.

Листинг 8.19. Получение значений с несуществующими ключами

```
db.get('this-key-does-not-exist', (err, value) => {
  if (err && !err.notFound) throw err;
  if (err && err.notFound) return console.log('Value was not found.');
  console.log('Value was found:', value);
});
```

Все операции чтения и записи данных получают необязательный аргумент `options` для переопределения параметров кодирования текущей операции (листинг 8.20).

Листинг 8.20. Переопределение параметров кодирования для конкретных операций

```
const options = {
  keyEncoding: 'binary',
  valueEncoding: 'hex'
};

db.put(new Uint8Array([1, 2, 3]), '0xFF0099', options, (err) => {
  if (err) throw err;
  db.get(new Uint8Array([1, 2, 3]), options, (err, value) => {
    if (err) throw err;
    console.log(value);
  });
});
```

8.12.7. Заменяемые подсистемы базы данных

Положительный побочный эффект разделения LevelUP/LevelDOWN заключается в том, что LevelUP не ограничивается использованием LevelDB в качестве внутренней подсистемы базы данных. Любая база данных, которая может быть упакована

в MemDown API, может использоваться в качестве подсистемы хранения данных для LevelUP. Это позволяет использовать один API для взаимодействия со многими хранилищами данных.

Некоторые примеры альтернативных подсистем баз данных:

- MySQL;
- Redis;
- MongoDB;
- файлы JSON;
- электронные таблицы Google;
- AWS DynamoDB;
- табличное хранилище Windows Azure;
- веб-хранилище браузера (IndexedDB/localStorage).

Возможность простой замены среды хранения данных и даже написания собственной подсистемы баз данных означает, что вы можете использовать один последовательный набор API баз данных и инструментов во многих ситуациях и условиях. Один API баз данных на все случаи!

Среди альтернативных подсистем баз данных часто используется система memdown, которая хранит значения исключительно в памяти, а не на диске (по аналогии с SQLite в режиме работы в памяти). Данная возможность особенно полезна в тестовой среде для снижения затрат на организацию тестирования и переналадку.

Чтобы выполнить листинг 8.21, убедитесь в том, что у вас установлены пакеты LevelUP и memdown:

```
npm install --save levelup memdown
```

Листинг 8.21. Использование memdown с LevelUP

```
const level = require('levelup')
const memdown = require('memdown')

const db = level('./level-articles.db', {
  keyEncoding: 'json',
  valueEncoding: 'json',
  db: memdown
});
```

Для memdown «путем» может быть любая строка, так как диск не используется.

Единственное реальное отличие – передача memdown в параметре db.

В этом примере можно было использовать тот же пакет level, использовавшийся ранее, потому что он представляет собой обертку для LevelUP. Но если вы не используете пакет LevelDOWN на базе LevelDB, входящий в поставку level, вы можете просто использовать LevelUP и избежать двоичной зависимости от LevelDB через LevelDOWN.

8.12.8. Модульная база данных

Быстродействие и минимализм LevelDB нашли отклик у многих разработчиков Node и породили движение модульных баз данных в сообществе Node. Концепция заключалась в том, чтобы разработчик мог точно выбрать, какие возможности нужны в его приложении, и адаптировать базу данных под ваш конкретный сценарий использования.

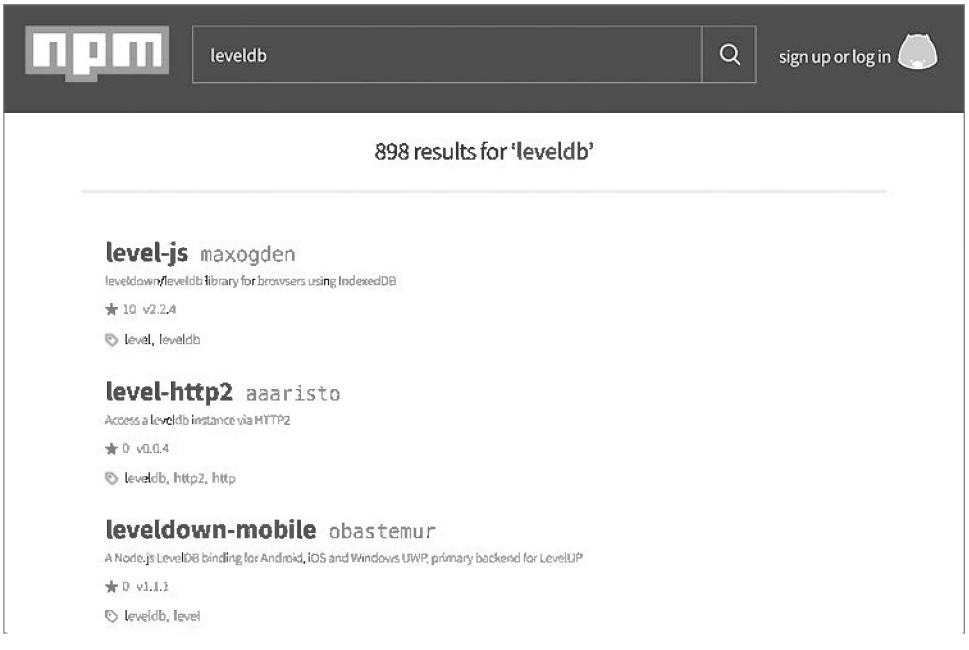


Рис. 8.6. Примеры сторонних пакетов LevelDB в npm

Несколько примеров модульной функциональности LevelDB, доступной в пакетах npm:

- атомарные обновления;
- автоматическое увеличение ключей;
- геопространственные запросы;
- оперативно обновляемые потоки;
- вытеснение LRU;
- задания отображения/свертки;
- репликация «главный/главный»;
- репликация «главный/подчиненный»;

- запросы SQL;
- вторичные индексы;
- триггеры;
- управление версиями данных.

В вики LevelUP ведется достаточно подробная сводка экосистемы LevelDB: *https://github.com/Level/levelup/wiki/Modules*. Вы также можете поискать информацию о leveldb в npm; на момент написания книги там было 898 пакетов. На рис. 8.6 показана информация о популярности LevelDB в npm.

8.13. Затратные операции сериализации и десериализации

Важно помнить, что встроенные операции JSON обходятся дорого и выполняются в блокирующем режиме; ваш процесс не может делать ничего другого в то время, пока приложение преобразует данные в JSON и обратно. Это относится к большинству других форматов сериализации. Сериализация часто становится важнейшим «узким местом» веб-сервера. Лучший способ сократить ее влияние — выполнять ее как можно реже и свести к минимуму объем обрабатываемых данных.

Возможно, вам удастся добиться некоторого повышения скорости за счет перехода на другой формат сериализации (например, MessagePack или Protocol Buffers), но рассматривать альтернативные форматы стоит только после того, как вы выжмете всю возможную экономию за счет сокращения размеров передаваемых данных и устранения избыточных шагов сериализации/десериализации.

Встроенные функции `JSON.stringify` и `JSON.parse` подверглись тщательной оптимизации, но при обработке мегабайтов данных они начинают пасовать. Для демонстрации в листинге 8.22 выполняется сериализация и десериализация приблизительно 10 Мбайт данных.

Листинг 8.22. Хронометражное тестирование сериализации

```
const bytes = require('pretty-bytes');
const obj = {};
for (let i = 0; i < 200000; i++) {
  obj[i] = {
    [Math.random()]: Math.random()
  };
}

console.time('serialise');
const jsonString = JSON.stringify(obj);
console.timeEnd('serialise');
console.log('Serialised Size', bytes(Buffer.byteLength(jsonString)));
```

```
console.time('deserialise');
const obj2 = JSON.parse(jsonString);
console.timeEnd('deserialise');
```

В 2015 году на компьютере Intel Core i7 MacBook Pro 3,1 ГГц с Node 6.2.2 на сериализацию приблизительно 10 Мбайт данных требовалось приблизительно 140 мс, а на десериализацию — 335 мс. На веб-сервере такая нагрузка была бы катастрофической, потому что эти операции выполняются в блокирующем режиме, и выполняются последовательно. Такой сервер сможет обрабатывать около ничтожных семи запросов в секунду при сериализации и около трех запросов в секунду при десериализации.

8.14. Хранение данных в браузере

Асинхронная модель программирования, используемая в Node, хорошо работает во многих практических сценариях, потому что у большинства веб-приложений самым значительным «узким местом» является ввод/вывод. Главное, что можно сделать для одновременного сокращения нагрузки на сервер и повышения качества взаимодействий с пользователем, — организация хранения данных на стороне клиента. Доволен будет тот пользователь, которому не приходится ожидать завершения циклической передачи по сети для получения результатов. Хранение данных на стороне клиента также улучшает доступность приложения: ваше приложение может сохранить по крайней мере частичную работоспособность, пока пользователь или ваш сервис не активен.

8.14.1. Веб-хранилище: localStorage и sessionStorage

Веб-хранилище определяет простое хранилище «ключ-значение» и обладает отличной поддержкой среди настольных и мобильных браузеров. При использовании веб-хранилища домен может сохранить небольшой объем данных в браузере и прочитать их позднее — даже после обновления сайта, закрытия вкладки или завершения браузера. Веб-хранилище — первый шаг на пути хранения данных на стороне клиента. Главной сильной стороной этой технологии является ее простота.

Существует два API веб-хранилища: localStorage и sessionStorage. sessionStorage реализует API, идентичный API localStorage, хотя и несколько отличается в поведении. Как и в случае с localStorage, данные, хранящиеся в sessionStorage, сохраняются между перезагрузками страниц, но в отличие от localStorage срок действия всех данных sessionStorage истекает при завершении сеанса страницы (при закрытии вкладки или браузера). Данные sessionStorage недоступны из разных окон браузеров.

API веб-хранилища были разработаны для преодоления ограничений cookie в браузерах. Если говорить конкретнее, cookie плохо подходят для обмена данными между

несколькими активными вкладками одного домена. Если пользователь выполняет некоторые действия между вкладками, хранилище sessionStorage может использоваться для обмена данными состояния между вкладками без использования сети.

Cookie также плохо подходят для долгосрочного хранения данных, которые должны «пережить» границы сеансов, вкладок и окон (например, созданных пользователем документов или электронной почты). Для таких сценариев было спроектировано хранилище localStorage. Объем данных, которые могут храниться в веб-хранилище, ограничивается в зависимости от конкретного браузера. Для мобильных браузеров этот порог составляет всего 5 Мбайт.

Сводка API

API localStorage предоставляет следующие методы для работы с ключами и значениями:

- `localStorage.setItem(key, value)` — сохраняет значение с заданным ключом;
- `localStorage.getItem(key)` — получает значение для заданного ключа;
- `localStorage.removeItem(key)` — удаляет значение для заданного ключа;
- `localStorage.clear()` — удаляет все ключи и значения;
- `localStorage.key(index)` — получает значение с заданным индексом;
- `localStorage.length` — возвращает общее количество ключей в localStorage.

8.14.2. Чтение и запись значений

Как ключи, так и значения должны быть строками. Если передать значение, которое не является строкой, оно будет автоматически преобразовано в строку. Такое преобразование не генерирует строки JSON; это обычное наивное преобразование с использованием `.toString`. Объекты в конечном итоге сериализуются в строку `[object Object]`. Чтобы разместить в веб-хранилище более сложные типы данных, приложение должно само сериализовать значения в строки и обратно. В листинге 8.23 показано, как хранить JSON в localStorage.

Листинг 8.23. Хранение JSON в веб-хранилище

```
const examplePreferences = {
  temperature: 'Celcius'
};

// Сериализация при записи
localStorage.setItem('preferences', JSON.stringify(examplePreferences));

// Десериализация при чтении
const preferences = JSON.parse(localStorage.getItem('preferences'));
console.log('Loaded preferences:', preferences);
```

Обращение к данным веб-хранилища осуществляется достаточно быстро, хотя и оно также выполняется синхронно. Веб-хранилище блокирует UI-поток на время выполнения чтения и записи. Для малых нагрузок эти затраты останутся незаметными, но вы должны позаботиться о том, чтобы избежать лишних операций чтения или записи (особенно при больших объемах данных). К сожалению, веб-хранилище также недоступно для веб-работников (web workers), поэтому все операции чтения и записи должны выполняться в одном UI-потоке. Подробный анализ влияния на быстродействие различных технологий хранения на стороне клиента приведен в посте Нолана Лоусона (Nolan Lawson), автора PouchDB: *http://nolanlawson.com/2015/09/29/indexeddb-websql-localstorage-what-blocks-the-dom/*.

API веб-хранилища не предоставляют встроенных средств для выполнения запросов, выборки ключей по диапазонам или поиска по значениям. Разработчик ограничивается возможностью обращения с перебором ключей. Чтобы провести поиск, вам придется создавать и поддерживать собственные индексы; или, если ваш набор данных достаточно мал, можно полностью перебрать его элементы. В листинге 8.24 перебираются все ключи localStorage.

Листинг 8.24. Перебор всего набора данных в localStorage

```
function getAllKeys() {
  return Object.keys(localStorage);
}

function getAllKeysAndValues() {
  return getAllKeys()
    .reduce((obj, str) => {
      obj[str] = localStorage.getItem(str);
      return obj;
    }, {});
}

// Получение всех значений
const allValues = getAllKeys().map(key => localStorage.getItem(key));

// В виде объекта
console.log(getAllKeysAndValues());
```

Как и у большинства хранилищ «ключ-значение», у ключей существует только одно пространство имен. Например, если в базе данных хранятся сообщения и комментарии, невозможно создать два разных хранилища для сообщений и для комментариев. Впрочем, достаточно легко реализовать собственные «пространства имен», снабдив каждый ключ префиксом для обозначения пространства имен (листинг 8.25).

Листинг 8.25. Ключи, реализующие пространства имен

```
localStorage.setItem(`/posts/${post.id}`, post);
localStorage.setItem(`/comments/${comment.id}`, comment);
```

Чтобы получить все элементы из пространства имен, отфильтруйте набор элементов при помощи функции `getAllKeys` (листинг 8.26).

Листинг 8.26. Получение всех элементов в пространстве имен

```
function getNamespaceItems(namespace) {
  return getAllKeys().filter(key => key.startsWith(namespace));
}
console.log(getNamespaceItems('/exampleNamespace'));
```

Учтите, что этот фрагмент перебирает все ключи из localStorage; помните о возможном снижении быстродействия при переборе большого количества элементов.

Из-за синхронности API localStorage существуют некоторые ограничения на то, где и как этот API может использоваться. Например, localStorage может использоваться для мемоизации результатов любой функции, которая получает и возвращает данные, сериализуемые в формат JSON (листинг 8.27).

Листинг 8.27. Использование localStorage для мемоизации

```
// При последующих вызовах с тем же аргументом
// будет извлекаться сохраненный результат.
function memoizedExpensiveOperation(data) {
  const key = `/memoized/${JSON.stringify(data)}`;
  const memoizedResult = localStorage.getItem(key);
  if (memoizedResult != null) return memoizedResult;
  // Затратные операции
  const result = expensiveWork(data);
  // Сохранение результатов в localStorage,
  // чтобы их не приходилось вычислять заново
  localStorage.setItem(key, result);
  return result;
}
```

Учтите, что операция должна быть достаточно медленной, чтобы преимущества мемоизации перевесили затраты на процесс сериализации/десериализации (например, криптографический алгоритм). Соответственно, localStorage лучше всего работает в том случае, если мемоизация экономит время, расходуемое на передачу данных по сети.

Технология веб-хранилищ имеет свои ограничения, но для правильно выбранных задач она становится мощным и простым инструментом. Также заслуживают внимания некоторые другие темы, относящиеся к хранению данных браузером:

- IndexedDB;
- Service workers;
- Offline-first.

8.14.3. localForage

Основные недостатки веб-хранилища — синхронный API с блокировкой и ограниченная емкость памяти в некоторых браузерах. Кроме веб-хранилища многие современные браузеры поддерживают WebSQL и (или) IndexedDB. Оба хранилища данных работают в неблокирующем режиме и позволяют хранить гораздо больше данных, чем технологии веб-хранилища.

Однако использовать любую из этих баз данных напрямую, как это делалось с API веб-хранилища, не рекомендуется. Технология WebSQL считается устаревшей, а ее наследник IndexDB имеет неудобный и громоздкий API, не говоря уже об исправлениях, необходимых для поддержки в браузерах. Чтобы удобно и надежно хранить данные в браузере без блокировки, мы рекомендуем использовать нестандартный инструмент для «нормализации» интерфейса. Одним из таких инструментов нормализации является библиотека localForage от Mozilla (*http.//mozilla.github.io/localForage/*).

Обзор API

Интерфейс localForage довольно близко воспроизводит интерфейс веб-хранилища, хотя и в асинхронной (не блокирующей) форме:

- `localforage.setItem(key, value, callback)` — сохраняет значение с заданным ключом;
- `localforage.getItem(key, callback)` — получает значение для заданного ключа;
- `localforage.removeItem(key, callback)` — удаляет значение для заданного ключа;
- `localforage.clear(callback)` — удаляет все ключи и значения;
- `localforage.key(index, callback)` — получает значение с заданным индексом;
- `localforage.length(callback)` — возвращает количество ключей в localForage.

API localForage также включает полезные дополнения, не имеющие аналогов в веб-хранилище:

- `localforage.keys(callback)` — удаляет все ключи и значения;
- `localforage.iterate(iterator, callback)` — перебирает ключи и значения.

8.14.4. Чтение и запись

API localForage поддерживает как обещания (promises), так и схему обратного вызова Node с объектом ошибки на первом месте.

Листинг 8.28. Сравнение получения данных в localStorage и localForage

```
const value = localStorage.getItem(key);
console.log(value);

localforage.getItem(key)
  .then(value => console.log(value));

localforage.getItem(key, (err, value) => {
  console.log(value);
});
```

localStorage: блокировка, синхронное выполнение.

localForage: без блокировки, асинхронное выполнение с использованием обещаний.

localForage: без блокировки, асинхронный вызов с использованием обратных вызовов в стиле Node.

Во внутренней реализации localForage использует лучший механизм хранения, доступный в текущей среде браузера. Если поддержка IndexedDB доступна, localForage использует ее. В противном случае будет сделана попытка переключиться на WebSQL и даже использовать веб-хранилище при необходимости. Вы можете настроить порядок, в котором будут опробованы механизмы хранения, и даже исключить некоторые варианты:

```
// Не будет использовать localStorage
localforage.setDriver([localforage.INDEXEDDB, localforage.WEBSQL]);
```

Никогда не возвращается к использованию localStorage.

В отличие от localStorage, localForage не ограничивается хранением одних лишь строк. Поддерживается большинство примитивов JavaScript (таких, как массивы и объекты), а также двоичные типы данных: `TypedArray`, `ArrayBuffer` и `Blob`. Учтите, что IndexedDB — единственная подсистема баз данных, изначально способная хранить двоичные данные: при использовании WebSQL и localStorage появляются затраты, связанные с преобразованием данных:

```
Promise.all([
  localforage.setItem('number', 3),
  localforage.setItem('object', { key: 'value' }),
  localforage.setItem('typedarray', new Uint32Array([1,2,3]))
]);
```

Повторение API веб-хранилища делает работу с localForage более интуитивной, а также исключает многие недостатки и проблемы совместимости при организации хранения данных в браузере.

8.15. Виртуальное хранение

Виртуальное хранение (hosted storage) — другая тактика, которая позволяет избежать необходимости самостоятельной организации хранения на стороне сервера. Сервисы виртуальной инфраструктуры — например, предоставляемые AWS (Amazon Web Services) — часто рассматриваются исключительно как средство оптимизации быстродействия и масштабирования, но умное использование виртуальных

сервисов с ранней стадии работы может сэкономить много времени, которое было бы потрачено на неудачную и излишнюю реализацию инфраструктуры.

Многие (если не все) базы данных, перечисленные в этой главе, предлагают возможность виртуального хранения. Виртуальные сервисы позволяют быстро опробовать инструментарий и даже развернуть общедоступные приложения без хлопот, связанных с настройкой самостоятельного хостинга базы данных. Тем не менее развернуть собственную базу данных становится все проще. Многие облачные сервисы предоставляют готовые образы серверов со всем необходимым программным обеспечением и конфигурацией для запуска машины с выбранной вами базой данных.

8.15.1. S3

Amazon Simple Storage Service (S3) — сервис удаленного размещения файлов, предоставляемый как часть популярного комплекса сервисов AWS. S3 представляет собой эффективный по затратам механизм хранения и размещения файлов, доступных по сети. По сути это файловая система в облаке. С использованием REST-совместимых вызовов HTTP файлы могут загружаться в *гнезда* (buckets) наряду с 2 Кбайт метаданных. Далее к содержимому гнезд можно обращаться методом HTTP GET или через протокол BitTorrent.

Для гнезд и их содержимого можно настроить разные уровни разрешений, включая доступ, контролируемый по времени. Также можно задать срок жизни для содержимого гнезд; по истечении этого срока содержимое становится недоступным и удаляется из гнезда. Данные S3 легко преобразуются в сеть доставки контента (CDN, Content Delivery Network). AWS предоставляет сервис CloudFront CDN, который можно легко связать с вашими файлами для обращения к ним с низкой задержкой из любой точки мира.

Не все данные необходимо (или хотя бы желательно) хранить в базе данных. Есть ли в ваших данных компоненты, которые могут рассматриваться как файлы? После того как вы сгенерировали результаты сложных вычислений для пользователя, возможно, вам стоит направить эти результаты в S3, а затем навсегда устраниться от обеспечения доступа к ним.

Распространенный и наиболее очевидный вариант использования S3 — размещение ресурсов, отправленных пользователем (например, графики). Ресурсы хранятся во временном каталоге на машине приложения, обрабатываются таким инструментом, как ImageMagick, для сокращения размера файла, и в дальнейшем отправляются в S3 для хостинга. Этот процесс можно еще сильнее упростить, отправляя данные прямо в S3, где они инициируют дальнейшую обработку. Клиентские приложения также могут отправлять данные прямо в S3. Некоторые сервисы, ориентированные на разработчиков, даже могут не предоставлять собственное хранилище; пользователь

должен предоставить маркеры доступа, чтобы приложение пользовалось собственными гнездами S3.

S3 не ограничивается хранением графики

Сервис S3 может использоваться для хранения любых типов файлов в любом формате размером до 5 терабайт. S3 лучше всего подходит для больших объемов данных, которые изменяются относительно редко, к которым обращаются как к единому целому.

Хранение данных в S3 обходит все сложности с настройкой и сопровождением сервера для размещения и хранения файлов. Этот вариант отлично подходит для сценариев, в которых относительно редко записываются большие блоки данных, к которым происходят атомарные обращения (то есть обращения как к единому целому), с большим количеством чтений из многих потенциальных мест.

8.16. Какую базу данных выбрать?

В этой главе рассмотрены лишь некоторые из баз данных, часто используемых в приложениях Node. Успешные приложения могут строиться на основе любых из этих баз данных. В одном приложении не всегда находится идеальное решение хранения данных; панацеи не существует. У каждой базы данных есть свои плюсы и минусы, и разработчик должен оценить, какие стороны лучше подходят для текущего состояния проекта. Часто наиболее уместным решением оказывается комбинация технологий. Вместо того чтобы спрашивать: «Какую базу данных следует использовать?», стоит задаться вопросом: «Насколько далеко можно зайти вообще без использования базы данных?» Какую часть проекта можно построить с минимумом долгосрочных решений? Часто решения лучше отложить на будущее; вы всегда сможете принять лучшее решение позднее, когда у вас будет больше информации.

8.17. Заключение

- В Node могут использоваться как реляционные базы данных, так и базы данных NoSQL.
- Простой модуль Node pg отлично подходит для работы с языком SQL.
- Модуль Knex позволяет работать с несколькими базами данных в Node.
- ACID — набор характеристик транзакций баз данных, обеспечивающий безопасность данных.
- MongoDB — база данных NoSQL, использующая JavaScript.

- Redis — хранилище структур данных, которое может использоваться как база данных и кэш.
- LevelDB — быстрое хранилище пар «ключ-значение» компании Google, ассоциирующее строки со значениями.
- LevelDB является модульной базой данных.
- Технологии веб-хранилища, включая localForage и localStorage, могут использоваться для хранения данных в браузере.
- Такие сервисы, как Amazon S3, могут использоваться для сохранения данных у провайдеров облачных сервисов.

9 Тестирование приложений Node

По мере расширения функциональности приложения возрастает риск появления ошибок. Без тестирования приложение считается незавершенным, а поскольку ручное тестирование утомительно и чревато новыми ошибками, обусловленными человеческим фактором, у разработчиков все более популярным становится автоматизированное тестирование. В этом случае вместо того, чтобы проверять функциональность приложения вручную, разработчик пишет логику автоматизированного тестирования кода.

Если вы ранее не сталкивались с концепцией автоматизированного тестирования, вообразите себе робота, который вместо вас выполняет всю рутинную работу, предоставляя вам возможность заняться более интересными вещами. Тогда при внесении изменений в код робот автоматически проверяет, 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 в процессе разработки приложения.

В этой главе рассматриваются два типа автоматизированного тестирования: модульное и приемочное. *Модульное тестирование* направлено на непосредственную проверку 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 через тестирование (Test-Driven Development, TDD) и разработка через реализацию поведения (Behavior-Driven Development, BDD). С практической точки зрения тесты в стиле TDD и BDD почти во всем одинаковы, несмотря на стилистические различия (которые могут быть важны в зависимости от того, кому придется читать ваши тесты). Между тестами в стиле TDD и BDD существуют и некоторые другие различия, но их описание выходит за рамки темы книги. *Приемочное тестирование* представляет собой дополнительный уровень тестирования, применяемый преимущественно при отладке веб-приложений. Этот вид тестирования подразумевает

сценарное управление браузером и проверку функциональности веб-приложений с его помощью.

В этой главе представлены готовые решения, иллюстрирующие модульное и приемочное тестирование. В рамках модульного тестирования мы познакомимся с Node-модулем assert и со средами Mocha, Vows и should.js, а также Chai. В рамках приемочного рассматривается среда Selenium. Инструменты тестирования вместе с соответствующими методологиями и подходами представлены на рис. 9.1.

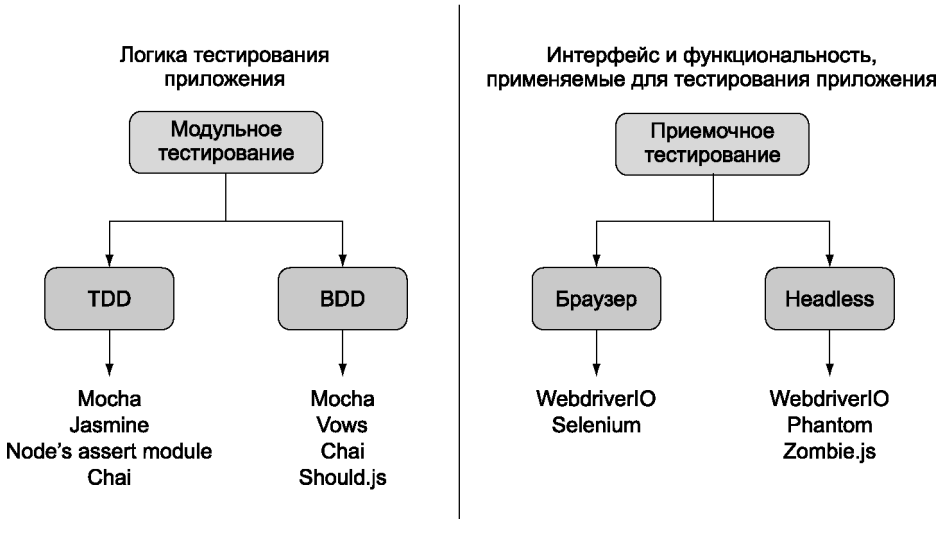


Рис. 9.1. Обзор тестовых фреймворков

Начнем с модульного тестирования.

9.1. Модульное тестирование

Модульное тестирование — это такая разновидность автоматического тестирования, при которой вы пишете логику для тестирования отдельных частей приложения. Благодаря тестам вы сможете более критично оценить архитектуру приложения и избежать ловушек на ранних этапах разработки. Тесты также придадут уверенность в том, что последние изменения не привели 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 внимательны при удалении этих файлов после завершения каждого теста, чтобы не удалить рабочие файлы другого теста, который еще не завершился. С учетом подобных особенностей, многие среды модульного тестирования предусматривают средства управления порядком выполнения тестов.

В этом разделе мы рассмотрим:

- *встроенный в Node модуль assert* — неплохой вариант для пользователей, осваивающих автоматизированное тестирование в стиле TDD;
- *Mocha* — относительно новый тестовый фреймворк, который может использоваться для тестирования в стиле TDD или BDD;
- *Vows* — широко используемый фреймворк тестирования в стиле BDD;
- *should.js* — модуль, который строится на основе модуля Node assert для проведения тестирования в стиле BDD.

В следующем разделе продемонстрировано тестирование бизнес-логики с использованием модуля assert, входящего в поставку Node.

9.1.1. Модуль assert

В большинстве случаев модульное тестирование приложений Node выполняется с помощью встроенного модуля assert. При этом проводится проверка некоего условия, и, если условие не выполняется, генерируется ошибка. Модуль assert используется многими сторонними фреймворками тестирования. Впрочем, даже без фреймворка он может быть весьма полезен для тестирования.

Простой пример

Предположим, у вас имеется простое приложение для ведения списка запланированных дел. Это приложение нужно протестировать и убедиться в том, что все работает как положено.

В листинге 9.1 определяется модуль, содержащий основную функциональность приложения. Логика модуля поддерживает создание, извлечение и удаление элементов списка запланированных дел. Модуль также содержит простой метод `doAsync`, что позволяет проследить за тестированием асинхронных методов. Сохраните этот файл под именем `todo.js`.

Листинг 9.1. Модель списка запланированных дел

```
class Todo {
  constructor() {
    this.todos = [];  <-------  Определяет базу данных запланированных дел.
```

```
  }
  add(item) {   <-- Добавляет элемент списка.
    if (!item) throw new Error('Todo.prototype.add requires an item');
    this.todos.push(item);
  }
  deleteAll() {   <-- Удаляет все элементы из списка.
    this.todos = [];
  }
  get length() {   <-- Определяет количество элементов списка.
    return this.todos.length;
  }
  doAsync(cb) {   <-- Выполняет обратный вызов через 2 секунды.
    setTimeout(cb, 2000, true);
  }
}
module.exports = Todo;   <-- Экспортирует функцию Todo.
```

Теперь можно воспользоваться модулем Node assert для тестирования кода. Введите в файле **test.js** код из листинга 9.2 для загрузки необходимых модулей, создайте новый список дел и присвойте значение переменной, в которой хранится количество завершенных тестов.

Листинг 9.2. Настройка требуемых модулей

```
const assert = require('assert');
const Todo = require('./todo');
const todo = new Todo();
let testsCompleted = 0;
```

Тестирование значения переменной с помощью утверждения equal

Теперь можно добавить тест, проверяющий функциональность удаления элементов из списка запланированных дел. Добавьте функцию из листинга 9.3 в конец файла **test.js**.

Листинг 9.3. Тест, проверяющий, что элементы не остаются в списке после удаления

```
function deleteTest() {
  todo.add('Delete Me');   <-- Добавляет данные для проверки удаления.
  assert.equal(todo.length, 1, '1 item should exist');   <-- Убеждается в том, что данные были добавлены правильно.
  todo.deleteAll();   <-- Удаляет все записи.
  assert.equal(todo.length, 0, 'No items should exist');   <-- Убеждается в том, что запись была удалена.
  testsCompleted++;   <-- Отмечает, что тест прошел успешно.
}
```

Тест добавляет элемент `todo`, а затем удаляет его. Если логика приложения работает правильно, в конце теста в списке не должно остаться ни одного запланированного дела — а следовательно, значение `todo.length` должно быть равно 0. Если же возникнут проблемы, генерируется исключение. Если же значение, возвращаемое `todo.length`, отлично от 0, в результате проверки утверждения на консоль выводится

трассировка стека с сообщением об ошибке «No items should exist». После успешной проверки значение переменной `testsCompleted` увеличивается, показывая, что тест пройден успешно.

Выявление возможных проблем с помощью утверждения notEaqual

В качестве следующего шага добавьте в файл test.js код из листинга 9.4. С помощью этого кода проверяются функциональность приложения, реализующая добавление элементов в список запланированных дел.

Листинг 9.4. Тестирование добавления элементов в список запланированных дел

```
function addTest() {
  todo.deleteAll();  <-- Удаляет все существующие элементы.
  todo.add('Added');  <-- Добавляет элемент.
  assert.notEqual(todo.getCount(), 0, '1 item should exist');  <-- Убеждается в том, что элементы существуют.
  testsCompleted++;  <-- Отмечает, что тест прошел успешно.
}
```

Как видите, модуль assert также поддерживает утверждения `notEqual`. Этот тип утверждений применяется в тех случаях, когда код приложения генерирует определенное значение, свидетельствующее о наличии проблем в логике приложения. В листинге 9.4 демонстрируется использование утверждения `notEqual`. Все элементы списка удаляются, элемент добавляется, затем логика приложения получает все элементы. Если количество элементов равно 0, значит, утверждение несостоятельно и в результате генерируется исключение.

Использование дополнительной функциональности: 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.

Проверка асинхронных значений на истинность с помощью утверждения ok

Пришло время протестировать метод `doAsync` в приложении списка запланированных дел (листинг 9.5). Поскольку тест является асинхронным, передается функция обратного вызова (`cb`), которая сигнализирует исполнителю о завершении теста. В данном случае вы не можете положиться на то, что функция вернет результат,

как в случае с синхронными тестами. Для проверки значения `doAsync` применяется утверждение `ok` — оно позволяет легко удостовериться в том, что это значение равно `true`.

Листинг 9.5. Тестирование передачи значения true при обратном вызове doAsync

```
function doAsyncTest(cb) {
  todo.doAsync(value => {   <-- Обратный вызов срабатывает через 2 секунды.
    assert.ok(value, 'Callback should be passed true');   <-- Проверяет значение на истинность.
    testsCompleted++;   <-- Отмечает, что тест прошел успешно.
    cb();   <-- Инициирует обратные вызовы при завершении.
  });
}
```

Тестирование корректности механизма генерирования ошибок

С помощью модуля assert можно также проверить корректность генерирования сообщений об ошибках, как показано в листинге 9.6. Во втором аргументе вызова `throws` передается регулярное выражение, которое ищет в сообщении об ошибке текст «requires».

Листинг 9.6. Тестирование генерирования ошибки при отсутствии параметра

```
function throwsTest(cb) {
  assert.throws(todo.add, /requires/);   <-- todo.add вызывается без аргументов.
  testsCompleted++;   <-- Отмечает, что тест прошел успешно.
}
```

Добавление логики запуска тестов

После определения тестов можно добавить в файл логику запуска каждого теста. Код, приведенный в листинге 9.7, запускает каждый тест, а затем выводит количество запущенных и завершенных тестов.

Листинг 9.7. Выполнение тестов и оповещение о завершении тестов

```
deleteTest();
addTest();
throwsTest();
doAsyncTest(() => {
  console.log(`Completed ${testsCompleted} tests`);   <-- Оповещает о завершении.
});
```

Для запуска тестов можно воспользоваться следующей командой:

```
$ node chapter09-testing/listing_09_1-7/test.js
```

Если тесты пройдут, сценарий сообщит 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 содержать большое количество шаблонного кода для подготовки теста (например, удалить все элементы) и получить трассу его выполнения (счетчик `completed`). Рутина отвлекает внимание разработчика от главной задачи — написания модульных тестов, поэтому лучше воспользоваться специальным фреймворком, который выполнит за вас всю тяжелую работу, чтобы вы могли заняться собственно тестированием бизнес-логики. Давайте посмотрим, как упростить свою задачу с помощью фреймворка модульного тестирования Mocha от независимого разработчика.

9.1.2. Mocha

Mocha — самый популярный фреймворк тестирования — прост в освоении. Хотя по умолчанию этот фреймворк использует стиль BDD, с его помощью также можно тестировать приложения в стиле TDD. Mocha включает в себя широкий набор средств, позволяющих, в частности, обнаруживать утечки глобальных переменных и тестирование на стороне клиента приложений.

ВЫЯВЛЕНИЕ УТЕЧЕК ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ

Необходимость в глобальных переменных, которые доступны во всем приложении, возникает нечасто, к тому же согласно передовой практике программирования их количество нужно стремиться минимизировать. Однако в ES5 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, включите в командную строку mocha параметр --ignored-leaks. Если же вы хотите указать допустимое число используемых глобальных переменных, перечислите их через запятую после параметра командной строки --globals.

По умолчанию логика тестов Mocha определяется BDD-функциями `describe`, `it`, `before`, `after`, `beforeEach` и `afterEach`. В качестве альтернативы можно использовать TDD-интерфейс Mocha, в котором `describe` заменяется на `suite`, `it` на `test`, `before` на `setup` и `after` на `teardown`. В нашем примере будет использоваться BDD-интерфейс по умолчанию.

Тестирование Node-приложений с помощью Mocha

Давайте создадим небольшой проект memdb — 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
$ npm init -y
$ npm install --save-dev mocha
```

Откройте файл package.json и добавьте свойство `scripts`, определяющее способ выполнения тестов:

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

Тесты располагаются в каталоге `test`. По умолчанию Mocha использует BDD-интерфейс. В листинге 9.8 показано, как он выглядит (chapter09-testing/memdb в исходном коде книги).

Листинг 9.8. Базовая структура тестов Mocha

```
const memdb = require('..');
describe('memdb', () => {
  describe('.saveSync(doc)', () => {
    it('should save the document', () => {
    });
  });
});
```

Mocha поддерживает также интерфейсы в стиле TDD, `qunit` и `exports`, которые подробно описаны на веб-сайте проекта (*https://mochajs.org/*). Чтобы проиллюстрировать концепцию использования различных интерфейсов, приведем интерфейс `exports`:

```
module.exports = {
  'memdb': {
    '.saveSync(doc)': {
      'should save the document': () => {
      }
    }
  }
}
```

Все эти интерфейсы предлагают одинаковую функциональность, но мы ограничимся BDD-интерфейсом и напишем первый тест, код которого приведен в листинге 9.9. Поместите его в файл test/memdb.js. В этом тесте для проверки утверждений используется модуль Node assert.

Листинг 9.9. Описание функциональности memdb .save

```
const memdb = require('..');
const assert = require('assert');
describe('memdb', () => {      <-- Описывает функциональность memdb.
  describe('.saveSync(doc)', () => {      <-- Описывает функциональность метода .save().
```

```
    it('should save the document', () => {  <-- Описывает ожидание.
      const pet = { name: 'Tobi' };
      memdb.saveSync(pet);
      const ret = memdb.first({ name: 'Tobi' });
      assert(ret == pet);  <-- Проверяет, что питомец был найден.
    });
  });
});
```

Чтобы выполнить тесты, достаточно выполнить команду npm test. По умолчанию фреймворк ищет выполняемые файлы JavaScript в папке ./test. Однако поскольку метод `.save()` не реализован, единственный определенный к настоящему времени тест провалится (рис. 9.2).

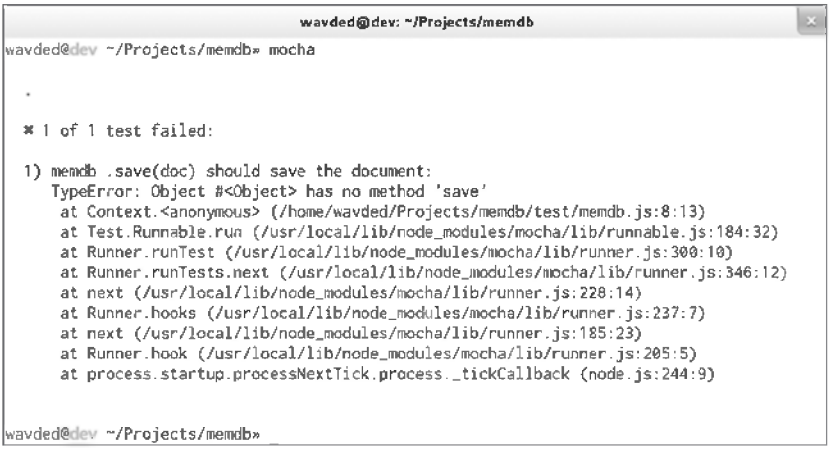


Рис. 9.2. Проваленный тест Mocha

А теперь обеспечим успешное прохождение теста. Добавьте код из листинга 9.10 в файл index.js.

Листинг 9.10. Добавление функциональности сохранения

```
const db = [];
exports.saveSync = (doc) => {
  db.push(doc);  <-- Добавляет документ в массив базы данных.
};
exports.first = (obj) => {
  return db.filter((doc) => {  <-- Выбирает документы, соответствующие каждому свойству в obj.
    for (let key in obj) {
      if (doc[key] != obj[key]) {  <-- Совпадение отсутствует; возвращает false и не выбирает документ.
        return false;
      }
    }
    return true;  <-- Все совпало; документ выбирается.
  }).shift();  <-- Только первый документ или null.
};
```

Снова запустите тесты командой npm. Результат должен выглядеть примерно так, как показано на рис. 9.3.

Рис. 9.3. Успешный тест в Mocha

Определение логики инициализации и завершения с использованием перехватчиков Mocha

В варианте теста из листинга 9.10 предполагается, что `memdb.first()` работает правильно, поэтому можно добавить еще несколько вариантов тестов. В измененном файле теста, код которого приведен в листинге 9.11, используется новая для Mocha концепция *перехватчиков* (hooks). Например, BDD-интерфейс предоставляет перехватчики `beforeEach()`, `afterEach()`, `before()` и `after()`, принимающие обратные вызовы для определения логики инициализации и завершения.

Листинг 9.11. Добавление перехватчика beforeEach

```
const memdb = require('..');
const assert = require('assert');
describe('memdb', () => {
  beforeEach(() => {
    memdb.clear();
  });
  describe('synchronous .saveSync(doc)', () => {
    it('should save the document', () => {
      const pet = { name: 'Tobi' };
      memdb.saveSync(pet);
      const ret = memdb.first({ name: 'Tobi' });
      assert(ret == pet);
    });
  });
  describe('.first(obj)', () => {
    it('should return the first matching doc', () => {
      const tobi = { name: 'Tobi' };
      const loki = { name: 'Loki' };
      memdb.saveSync(tobi);
      memdb.saveSync(loki);
      let ret = memdb.first({ name: 'Tobi' });
      assert(ret == tobi);
      ret = memdb.first({ name: 'Loki' });
      assert(ret == loki);
    });
```

Очищает базу данных перед каждым тестовым сценарием, чтобы тесты не имели состояния.

Первое ожидание для .first().

Сохраняет два документа.

Проверяет правильность возвращения каждого документа.

```
    it('should return null when no doc matches', () => {
      const ret = memdb.first({ name: 'Manny' });
      assert(ret == null);
    });
  });
});
```

Второе ожидание для .first().

В идеальном случае тестовые сценарии вообще не должны иметь состояния. Чтобы добиться этого с базой данных memdb, нужно просто удалить все документы, реализовав метод `.clear()` в файле index.js:

```
exports.clear = () => {
  db.length = 0;
};
```

Запустив Mocha снова, вы увидите, что все три теста пройдены.

Тестирование асинхронной логики

До сих пор мы еще не рассматривали тестирование асинхронной логики 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` так, как показано ниже, можно передать функцию обратного вызова, которая будет выполняться после небольшой задержки (имитирующей выполнение асинхронной операции):

```
exports.save = (doc, cb) => {
  db.push(doc);
  if (cb) {
    setTimeout(() => {
      cb();
    }, 1000);
  }
};
```

Тестовые сценарии Mocha определяются как асинхронные простым добавлением аргумента в функцию, задающую логику тестирования. Этот аргумент обычно называется `done`. Листинг 9.12 показывает, как написать тест для асинхронного метода `save`.

Листинг 9.12. Тестирование асинхронной логики

```
describe('asyncronous .save(doc)', () => {
  it('should save the document', (done) => {
    const pet = { name: 'Tobi' };
    memdb.save(pet, () => {
      const ret = memdb.first({ name: 'Tobi' });
      assert(ret == pet);
      done();
    });
  });
});
```

Сохраняет документ.

Активизирует обратный вызов с первым документом.

Проверяет, что документ сохранен правильно.

Сообщает Mocha об окончании тестового сценария.

Аналогичное правило применяется ко всем перехватчикам. Например, перехватчик `beforeEach()`, предназначенный для очистки базы данных, мог бы добавить обратный вызов, и тогда фреймворк Mocha будет ожидать его завершения для продолжения работы. Если `done()` вызывается с объектом ошибки в первом аргументе, Mocha сообщит об ошибке и пометит перехватчик или тестовый сценарий как непрошедший:

```
beforeEach((done) => {
  memdb.clear(done);
});
```

За дополнительной информацией о Mocha обращайтесь к электронной документации по адресу *http://mochajs.org*. Фреймворк Mocha также может работать с JavaScript на стороне клиента.

НЕПАРАЛЛЕЛЬНОЕ ТЕСТИРОВАНИЕ В MOCHA

В Mocha тесты выполняются не параллельно, а последовательно; хотя этот режим упрощает написание тестов, выполнение групп тестов происходит медленнее. Тем не менее Mocha не допустит, чтобы тест продолжался слишком долго. По умолчанию Mocha разрешает каждому тесту выполняться не более 2000 миллисекунд, после чего он считается проваленным. Если вам требуются более длительные тесты, запустите Mocha с параметром командной строки --timeout и укажите большее время тестирования.

В большинстве случаев последовательный режим выполнения тестов приемлем. Если вам этот способ не подходит, воспользуйтесь другими фреймворками тестирования, работающими в параллельном режиме, — такими как фреймворк Vows, рассматриваемый в следующем разделе.

9.1.3. Vows

Тесты, создаваемые с помощью фреймворка модульного тестирования Vows, получаются более структурированными, чем с другими фреймворками, поэтому Vows-тесты проще читать и сопровождать.

Vows использует собственную терминологию в стиле BDD, 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 протестировать. Пакеты и контексты запускаются параллельно. Контекст может содержать *раздел* (topic), одно или больше *обязательств* (vows) и (или) один или больше связанных контекстов (вложенные контексты также могут выполняться

параллельно). Раздел — это логика тестирования, связанная с контекстом. Обязательство представляет собой тест результата раздела. Структура Vows-тестов представлена на рис. 9.4.

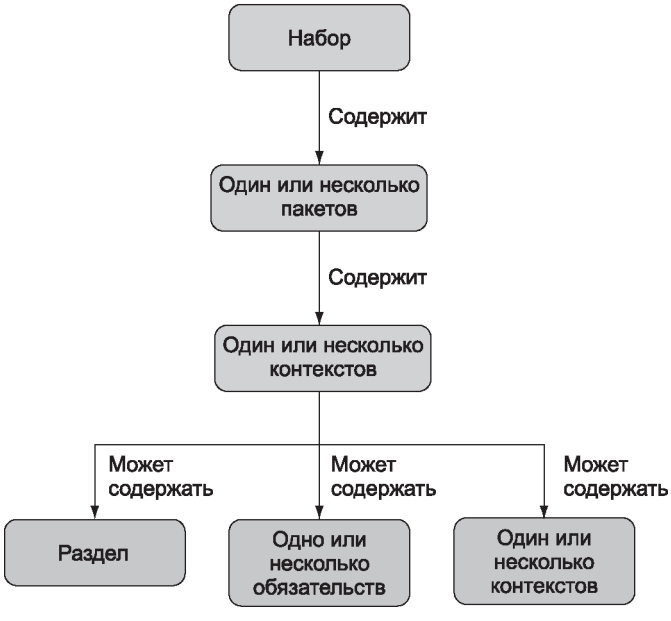


Рис. 9.4. В Vows для структурирования тестов используются пакеты, контексты, разделы и обязательства

Фреймворк Vows, как и Mocha, ориентирован на автоматизированное тестирование 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.

Добавьте фреймворк Vows в проект, установив его с использованием npm:

```
mkdir -p vows-todo/test
cd vows-todo
touch todo.js
touch test/todo-test.js
npm init -y
npm install --save-dev -g vows
```

Vows следует добавить в свойство `test` файла package.json, чтобы для проведения тестов было достаточно ввести команду `npm test`:

```
"scripts": {
  "test": "vows test/*.js"
},
```

Тестирование логики приложения с помощью Vows

Чтобы запустить процесс тестирования 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 сценарий Node) проводится один из тестов основной логики приложения для списка запланированных дел.

В листинге 9.13 создается пакет тестов. Внутри пакета можно определить контекст; внутри контекста определяются раздел и обязательство. Обратите внимание на использование обратного вызова для обработки асинхронной логики в разделе. Если раздел не является асинхронным, то вместо передачи значения через обратный вызов осуществляется возврат этого значения. Сохраните файл под именем test/todo-test.js.

Листинг 9.13. Использование Vows для тестирования

```
const vows = require('vows');
const assert = require('assert');
const Todo = require('./../todo');
vows.describe('Todo').addBatch({  <-- Пакет
  'when adding an item': {  <-- Контекст
    topic: () => {  <-- Раздел
      const todo = new Todo();
      todo.add('Feed my cat');
      return todo;
    },
    'it should exist in my todos': (er, todo) => {  <-- Обязательство
      assert.equal(todo.length, 1);
    }
  }
}).export(module);
```

Если все было сделано правильно, тест можно будет запустить командой `npm test`. Если вы установили Vows глобально командой `npm i -g vows`, все тесты из папки test выполняются следующей командой:

```
$ vows test/*
```

За дополнительной информацией о Vows обращайтесь к электронной документации проекта (*http://vowsjs.org/*) — рис. 9.5.

Vows предоставляет всеобъемлющее решение для тестирования, причем вы можете заменять функциональность библиотеки тестирования за счет использования другой библиотеки проверки тестовых утверждений. Допустим, вам нравится Mocha, но не нравится библиотека утверждений Node. В следующем разделе рассматривается Chai — библиотека тестовых утверждений, которая может использоваться вместо модуля Node assert.

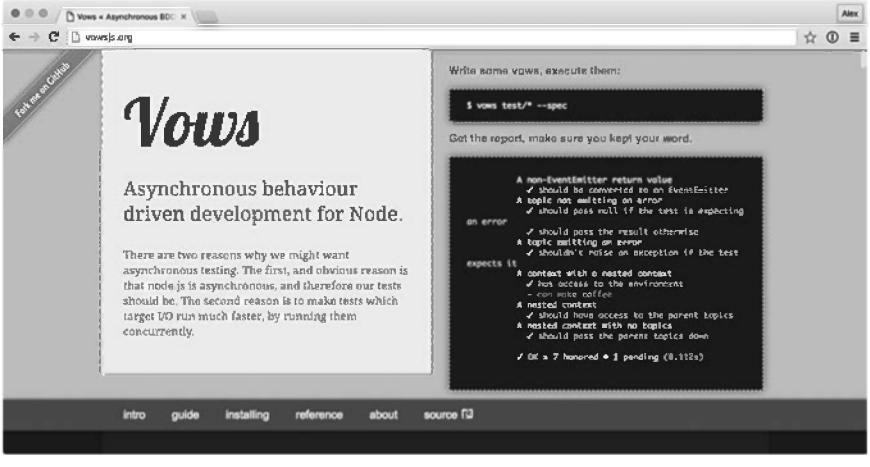


Рис. 9.5. Vows сочетает полнофункциональное BDD-тестирование с макросами и управлением логикой выполнения

9.1.4. Chai

Chai (*http://chaijs.com/*) — популярная библиотека тестовых утверждений, включающая в себя три интерфейса: `should`, `expect` и `assert`. Интерфейс `assert`, представленный в листинге 9.14, внешне напоминает встроенный модуль тестовых утверждений Node, но включает полезные средства для сравнения объектов, массивов и их свойств. Например, метод `typeOf` может использоваться для сравнения типов, а `property` проверяет, обладает ли объект нужным свойством.

Листинг 9.14. Интерфейс assert в Chai

```
const chai = require('chai');
const assert = chai.assert;  <-- Выбирает интерфейс assert.
const foo = 'bar';
const tea = { flavors: ['chai', 'earl grey', 'pg tips'] };

assert.typeOf(foo, 'string');

assert.equal(foo, 'bar');
assert.lengthOf(foo, 3);

assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Впрочем, главной причиной для использования Chai все же являются интерфейсы `should` и `expect`. Они предоставляют динамичный API, напоминающий специализированные библиотеки в стиле BDD. Вот как выглядит интерфейс `expect`:

```
const chai = require('chai');
const expect = chai.expect;
const foo = 'bar';
```

```
expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
```

Такой API выглядит как предложение на английском языке — декларативный стиль не такой компактный, но проще читается. В интерфейсе `should` все наоборот: объекты декорируются дополнительными свойствами, так что вам не придется заключать проверку в вызов метода, как в случае `expect`:

```
const chai = require('chai');
chai.should();
const foo = 'bar';
foo.should.be.a('string');
foo.should.equal('bar');
```

Выбор используемого интерфейса зависит от проекта. Если вы начинаете с написания тестов, которые используются для документирования проекта, детализированные интерфейсы `expect` и `should` подойдут хорошо. Блюстители чистоты JavaScript предпочитают `expect`, потому что этот вариант не изменяет прототипы, но разработчикам с опытом Ruby могут быть более знакомы такие API, как `should`.

Главным преимуществом Chai является широкий ассортимент плагинов. К их числу относятся такие расширения, как chai-as-promised (*http.//chaijs.com/plugins/chai-as-promised/*), упрощающее написание тестового кода с использованием обещаний, и chai-stats (*http://chaijs.com/plugins/chai-stats/*) — библиотеки для сравнения чисел в соответствии со статистическими методами. Обратите внимание: Chai является библиотекой тестовых утверждений, которая может использоваться наряду с такими системами выполнения тестов, как Mocha.

Еще одна библиотека тестовых утверждений BDD — Should.js. В следующем разделе мы рассмотрим библиотеку Should.js и покажем, как писать тесты с использованием этой библиотеки.

9.1.5. Библиотека should.js

Библиотека should.js — это библиотека тестовых утверждений, которая сделает ваши тесты более понятными, так как вы сможете выражать утверждения в стиле BDD. Поскольку она предназначена для использования вместе с другими фреймворками тестирования, вам не придется отказываться от своего любимого фреймворка. В этом разделе мы разберемся, 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")`.

Допустим, вы пишете в Node калькулятор, запускаемый из командной строки; он поможет вам понять, кто сколько должен заплатить чаевых, когда вы с друзьями решите пропустить по стаканчику. Естественно, вам придется написать тесты для проверки логики вычислений, причем они должны быть понятны даже вашим друзьям-гуманитариям, чтобы они не решили, что вы пытаетесь их надуть.

Чтобы создать приложение-калькулятор, 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 следующими командами:

```
npm init -y
npm install --save-dev should
```

Затем отредактируйте содержимое файла **index.js**, который будет содержать логику, определяющую базовую функциональность приложения. Точнее, логика калькуляции чаевых будет содержать четыре вспомогательные функции:

- `addPercentageToEach` — увеличивает каждое число в массиве на заданную процентную величину;
- `sum` — вычисляет сумму всех элементов массива;
- `percentFormat` — форматирует выводимую процентную величину;
- `dollarFormat` — форматирует выводимую величину в долларах.

Добавьте описанную логику, включив в файл **index.js** код из листинга 9.15.

Листинг 9.15. Логика вычисления чаевых

```
exports.addPercentageToEach = (prices, percentage) => {
  return prices.map((total) => {
    total = parseFloat(total);
    return total + (total * percentage);
  });
};
exports.sum = (prices) => {
  return prices.reduce((currentSum, currentValue) => {
    return parseFloat(currentSum) + parseFloat(currentValue);
  });
};
exports.percentFormat = (percentage) => {
  return parseFloat(percentage) * 100 + '%';
};
exports.dollarFormat = (number) => {
  return `$${parseFloat(number).toFixed(2)}`;
};
```

Прибавляет процентную величину к элементам массива.

Вычисляет сумму элементов массива.

Форматирует процентную величину для вывода.

Форматирует денежную сумму в долларах для вывода.

Затем отредактируйте тестовый сценарий в файле test/tips.js в соответствии с листингом 9.16. Этот сценарий загружает модуль с логикой вычисления чаевых, определяет величину налога, процент чаевых и тестируемые позиции счета, а также дополнительно тестирует добавление процента для каждого элемента массива и общую сумму счета.

Листинг 9.16. Логика тестирования кода вычисления чаевых

```
const tips = require('..');  <-- Использует модуль с логикой вычисления чаевых.
const should = require('should');
const tax = 0.12;  <-- Определяет ставку налога и процент чаевых.
const tip = 0.15;
const prices = [10, 20];  <-- Определяет тестируемые позиции счета.

const pricesWithTipAndTax = tips.addPercentageToEach(prices, tip + tax);
pricesWithTipAndTax[0].should.equal(12.7);  <-- Тестирует добавление налога и чаевых.
pricesWithTipAndTax[1].should.equal(25.4);

const totalAmount = tips.sum(pricesWithTipAndTax).toFixed(2);
totalAmount.should.equal('38.10');  <-- Тестирует суммирование позиций счета.

const totalAmountAsCurrency = tips.dollarFormat(totalAmount);
totalAmountAsCurrency.should.equal('$38.10');

const tipAsPercent = tips.percentFormat(tip);
tipAsPercent.should.equal('15%');
```

Запустите сценарий с помощью следующей команды. Если все сделано правильно, 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
```

Чтобы упростить запуск, добавьте команду в свойство `test` из раздела `scripts` в файле package.json:

```
"scripts": {
  "test": "node test/tips.js"
}
```

Библиотека Should.js поддерживает многие типы тестовых 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/shouldjs/should.js*) есть полная документация, описывающая функциональность библиотеки Should.js.

Помимо библиотек тестовых утверждений, для управления выполнением тестируемого кода часто применяются *шпионы*, *заглушки* и *макеты*. В следующем разделе показано, как это делается при использовании Sinon.js.

9.1.6. Шпионы и заглушки в Sinon.JS

Последний инструмент в вашем инструментарии тестирования — библиотека макетов и заглушек. Мы пишем модульные тесты для изоляции частей тестируемой системы, но иногда добиться этой цели бывает трудно. Представьте, что вы тестируете код для масштабирования изображений. Записывать данные в реальные графические файлы не хочется, но как писать тесты? В коде не должно быть специальных ветвей, обходящихся без записи в файловую систему, потому что в таком случае вы фактически не будете тестировать свой код. В таких случаях необходимо определять *заглушки* (stubs) для функциональности файловой системы. Практика написания заглушек также помогает реализовать принцип TDD, потому что вы можете определять заглушки для зависимостей, которые еще не готовы.

В этом разделе вы узнаете, как использовать Sinon.JS (*http://sinonjs.org/*) для написания тестовых шпионов, заглушек и макетов. Прежде чем браться за дело, создайте новый проект и установите Sinon:

```
mkdir sinon-js-examples
cd sinon-js-examples
npm init -y
mkdir test
npm i --save-dev sinon
```

Затем создайте файл с тестовыми данными. В нашем примере будет использоваться простая база данных «ключ-значение» в формате JSON. Наша цель — создать заглушку для API файловой системы, чтобы избежать создания реальных файлов в файловой системе. Это позволит нам протестировать код базы данных — в отличие от кода файловых операций (листинг 9.17).

Листинг 9.17. Класс Database

```
const fs = require('fs');

class Database {
  constructor(filename) {
    this.filename = filename;
    this.data = {};
  }

  save(cb) {
    fs.writeFile(this.filename, JSON.stringify(this.data), cb);
  }

  insert(key, value) {
    this.data[key] = value;
  }
}

module.exports = Database;
```

Сохраните файл под именем db.js. А теперь попробуем протестировать его с использованием шпионов Sinon.

Шпионы

Иногда при тестировании бывает достаточно знать, что метод просто был вызван. *Шпионы* (spies) идеально подходят для этой цели. API позволяет заменить метод объектом, к которому можно применять тестовые утверждения. Чтобы смоделировать вызов `fs.writeFile` в db.js, воспользуйтесь заменой метода в Sinon — шпионом:

```
sinon.spy(fs, 'writeFile');
```

При завершении теста исходный метод восстанавливается методом `restore`:

```
fs.writeFile.restore();
```

В тестовой библиотеке (такой, как Mocha) эти вызовы размещаются в блоках `beforeEach` и `afterEach`. В листинге 9.18 приведен полный пример использования шпионов. Сохраните файл под именем spies.js.

Листинг 9.18. Использование шпионов

```
const sinon = require('sinon');
const Database = require('./db');
const fs = require('fs');
const database = new Database('./sample.json');

const fsWriteFileSpy = sinon.spy(fs, 'writeFile');   <-- (1) Заменяет исходный метод fs.
const saveDone = sinon.spy();

database.insert('name', 'Charles Dickens');
database.save(saveDone);

sinon.assert.calledOnce(fsWriteFileSpy);   <-- (2) Проверяет, что writeFile вызывается только один раз.

fs.writeFile.restore();   <-- (3) Восстанавливает исходный метод.
```

После назначения шпиона **(1)** выполняется тестируемый код. Затем конструкция sinon.assert проверяет, что нужный метод был вызван **(2)**, после чего восстанавливается исходный метод **(3)**. В этом тесте восстановление не является строго обязательным, но на практике рекомендуется всегда восстанавливать измененные методы.

Заглушки

В других ситуациях требуется управлять логикой выполнения. Например, можно инициировать выполнение ошибочной ветви, чтобы протестировать обработку ошибок в коде. Предыдущий пример можно переписать с использованием заглушки

(stub) вместо шпиона, чтобы метод `writeFile` выполнил свой обратный вызов. Обратите внимание: при этом следует избегать вызова исходного метода, а вместо этого заставить тестируемый код выполнить переданную функцию обратного вызова. В листинге 9.19 продемонстрировано использование заглушек для замены функций. Сохраните файл под именем stub.js.

Листинг 9.19. Использование заглушек

```
const sinon = require('sinon');
const Database = require('./db');
const fs = require('fs');
const database = new Database('./sample.json');

const stub = sinon.stub(fs, 'writeFile', (file, data, cb) => {
  cb();
});
const saveDone = sinon.spy();

database.insert('name', 'Charles Dickens');
database.save(saveDone);

sinon.assert.calledOnce(stub);
sinon.assert.calledOnce(saveDone);

fs.writeFile.restore();
```

Заменяет writeFile собственной функцией.

Проверяет, что метод writeFile был вызван.

Проверяет, что метод обратного вызова database, save был выполнен.

Сочетание шпионов и заглушек идеально подходит для тестирования кода Node, в котором интенсивно используются функции, заданные пользователем, обратные вызовы и обещания. От рассмотрения инструментов, предназначенных для модульного тестирования, мы перейдем к совершенно другому стилю тестирования: *функциональному* тестированию.

9.2. Функциональное тестирование

В большинстве проектов веб-разработки *функциональные тесты* основаны на управлении браузером и проверке различных преобразований DOM для списка требований, относящихся к конкретному пользователю. Представьте, что вы строите систему управления контентом. Функциональный тест для функции отправки графики в библиотеку должен переслать изображение, проверить, что добавление прошло успешно, и убедиться в том, что оно было добавлено в соответствующий список.

Выбор средств для реализации функционального тестирования в Node чрезвычайно широк. Однако на высоком уровне их можно разделить на две крупные группы: *терминальные* (headless) и *браузерные* тесты. Терминальные тесты обычно используют некий аналог PhantomJS для формирования браузерной среды, ориентированной на

терминальную среду; при этом более простые решения используют такие библиотеки, как Cheerio и JSDOM. Браузерные тесты используют средства автоматизации браузера — такие, как Selenium (*www.seleniumhq.org*), — чтобы вы могли писать сценарии, управляющие реальным браузером. Оба подхода используют одни и те же тестовые средства Node, так что вы сможете использовать Mocha, Jasmine и даже Cucumber для управления взаимодействием Selenium с вашим приложением. Пример среды тестирования показан на рис. 9.6.

Рис. 9.6. Тестирование средствами автоматизации браузера

В этом разделе рассматриваются средства функционального тестирования в Node, которые позволят вам сформировать тестовую среду на основании ваших специфических требований.

9.2.1. Selenium

Selenium — популярная библиотека автоматизации браузера на базе Java. Использовав драйвер для конкретного языка, вы сможете подключиться к серверу Selenium и провести тесты для реального браузера. В этом разделе вы научитесь пользоваться WebdriverIO (*http://webdriver.io/*), драйвером Selenium для Node.

Работать с Selenium сложнее, чем с простыми тестовыми библиотеками Node, потому что вам придется установить Java и загрузить JAR-файл Selenium. Загрузите Java для своей операционной системы, зайдите на сайт загрузки Selenium (*http://docs.seleniumhq.org/download/*) и загрузите JAR-файл. После этого сервер Selenium можно будет запустить командой следующего вида:

```
java -jar selenium-server-standalone-2.53.0.jar
```

В вашем случае точная версия Selenium может быть другой. Возможно, вам также придется указать путь к двоичному файлу браузера. Например, в Windows 10 с браузером Firefox, заданным в свойстве `browserName`, полный путь для Firefox может быть указан следующим образом:

```
java -jar -Dwebdriver.firefox.driver="C:\path\to\firefox.exe" selenium-
server-standalone-3.0.1.jar
```

Точный путь зависит от конкретного способа установки Firefox на вашей машине. За дополнительной информацией о драйвере Firefox обращайтесь к документации SeleniumHQ (*https://github.com/SeleniumHQ/selenium/wiki/FirefoxDriver*). Драйверы для Chrome и Microsoft Edge настраиваются аналогичным образом.

Теперь создайте новый проект Node и установите WebdriverIO:

```
mkdir -p selenium/test/specs
cd selenium
npm init -y
npm install --save-dev webdriverio
npm install --save express
```

В комплект поставки WebdriverIO включен удобный генератор конфигурационных файлов. Чтобы запустить его, выполните команду `wdio config`:

```
./node_modules/.bin/wdio config
```

Ответьте на вопросы и подтвердите значения по умолчанию. На рис. 9.7 показан пример сеанса.

```
=========================
WDIO Configuration Helper
=========================

? Where do you want to execute your tests? On my local machine
? Which framework do you want to use? mocha
? Shall I install the framework adapter for you? Yes
? Where are your test specs located? ./test/specs/**/*.js
? Which reporter do you want to use?
? Do you want to add a service to your test setup?
? Level of logging verbosity: verbose
? In which directory should screenshots gets saved if a command fails? ./errorShots/
? What is the base url? http://localhost:4000

Installing wdio packages:
pkg: wdio-mocha-framework

Packages installed successfully, creating configuration file...

Configuration file was created successfully!
To run your tests, execute:

$ wdio wdio.conf.js
```

Рис. 9.7. Настройка тестов Selenium в wdio

Дополните файл package.json командой wdio, чтобы тесты можно было запускать командой npm test:

```
"scripts": {
  "test": "wdio wdio.conf.js"
},
```

Теперь добавьте что-нибудь в тест. Базового сервера Express будет достаточно. Пример в листинге 9.20 может использоваться в последующих листингах для тестирования. Сохраните код из листинга в файле index.js (файл c09-testing/selenium/index.js в архиве кода книги).

Листинг 9.20. Пример проекта Express

```
const express = require('express');
const app = express();
const port = process.env.PORT || 4000;

app.get('/', (req, res) => {
  res.send(`
<html>
  <head>
    <title>My to-do list</title>
  </head>
  <body>
    <h1>Welcome to my awesome to-do list</h1>
  </body>
</html>
  `);
});

app.listen(port, () => {
  console.log('Running on port', port);
});
```

К преимуществам WebdriverIO можно отнести простой, динамичный API для написания тестов Selenium. Синтаксис логичен и понятен — вы даже можете использовать селекторы CSS при написании тестов. В листинге 9.21 (файл test/specs/todo-test.js в архиве кода) приведен простой тест, который создает клиента WebdriverIO, а затем проверяет заголовок на странице.

Листинг 9.21. Тест WebdriverIO

```
const assert = require('assert');
const webdriverio = require('webdriverio');
describe('todo tests', () => {
  let client;
  before(() => {
    client = webdriverio.remote();    <-- (1) Создает клиента WebdriverIO.
    return client.init();
  });
  it('todo list test', () => {
```

```
    return client
      .url('/')
      .getTitle()
        .then(title => assert.equal(title, 'My to-do list'));
  });
});
```

(2) Получает домашнюю страницу.

Получает заголовок.

Проверяет заголовок.

После подключения WebdriverIO **(1)** вы сможете использовать экземпляр клиента для получения страниц от приложения **(2)**. После этого вы сможете запросить текущее состояние документа в браузере — в примере используется метод `getTitle` для получения элемента `title` из заголовка документа. Если вы хотите запросить элементы CSS из документа, используйте метод `.elements` (*http://webdriver.io/api/protocol/elements.html*). Существуют разные методы для манипуляций с документом, формами и даже cookie.

Этот тест, похожий на другие тесты Mocha этой главы, заставляет реальный браузер обратиться к веб-приложению Node. Чтобы выполнить тест, запустите сервер на порту 4000:

```
PORT=4000 node index.js
```

Затем введите команду `npm test`. Вы увидите, как откроется Firefox, а в командной строке выполняются тесты. Если вы предпочитаете Chrome, откройте файл `wdio.conf.js` и измените свойство `browserName`.

НЕТРИВИАЛЬНОЕ ТЕСТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ SELENIUM

Если вы используете WebdriverIO и Selenium для тестирования более сложного веб-приложения, использующего React или Angular и т. д., вам стоит поближе познакомиться со вспомогательными методами. Некоторые из методов приостанавливают тест до того момента, когда станут доступными некоторые элементы; эта возможность отлично подходит для приложений React, которые асинхронно строят документ, многократно обновляя его до появления удаленных данных.

За дополнительной информацией обращайтесь к описанию методов waitFor* — таких, как waitForVisible (http://webdriver.io/api/utility/waitForVisible.html).

9.3. Ошибки при прохождении тестов

Когда вы работаете над серьезным проектом, неизбежно наступит момент, когда тесты перестанут проходить. Node предоставляет ряд инструментов для получения более подробной информации о сбойных тестах. В этом разделе вы узнаете, как расширить выходную информацию, генерируемую при отладке сбойных тестов.

Если тест не прошел, прежде всего следует сгенерировать более подробный отладочный вывод. В следующем разделе показано, как сделать это при помощи переменной NODE_DEBUG.

9.3.1. Получение более подробных журналов

Если тест не прошел, полезно получить информацию о том, что при этом происходило в программе. В Node существует два способа получения такой информации: один предназначен для внутренней функциональности Node, другой — для модулей npm. Для отладки базовых модулей Node используется переменная NODE_DEBUG.

Переменная NODE_DEBUG

Представьте, что в приложении имеется вызов функции файловой системы с большим уровнем вложенности, для которого вы забыли использовать обратный вызов. Например, для следующего примера будет выдано исключение:

```
const fs = require('fs');

function deeplyNested() {
fs.readFile('/');
}

deeplyNested();
```

В трассировке стека выводится ограниченная информация об исключении. В частности, в нее не включается полная информация о точке вызова, в которой произошло исключение:

```
fs.js:60
      throw err;  // Forgot a callback but don't know where? Use
     NODE_DEBUG=fs
      ^
Error: EISDIR: illegal operation on a directory, read
    at Error (native)
```

Без содержательных комментариев многие программисты, столкнувшись с такой ошибкой, начинают винить Node. Но, как видно из комментария, значение NODE_DEBUG=fs позволяет получить более подробную информацию о модуле fs. Запустите сценарий следующей командой:

```
NODE_DEBUG=fs node node-debug-example.js
```

Теперь вы получите более подробную трассировку, упрощающую процесс отладки:

```
fs.js:53
        throw backtrace;
        ^
```

```
Error: EISDIR: illegal operation on a directory, read
    at rethrow (fs.js:48:21)
    at maybeCallback (fs.js:66:42)
    at Object.fs.readFile (fs.js:227:18)
    at deeplyNested (node-debug-example.js:4:6)
    at Object.<anonymous> (node-debug-example.js:7:1)
    at Module._compile (module.js:435:26)
    at Object.Module._extensions..js (module.js:442:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:311:12)
    at Function.Module.runMain (module.js:467:10)
```

Из трассировки видно, что проблема кроется в нашем файле — в функции из строки 4, которая была изначально вызвана из строки 7. Такая информация существенно упрощает отладку любого кода, использующего базовые модули, причем этот способ включает в себя не только файловую систему, но и сетевые библиотеки — такие, как Node-модули клиента и сервера HTTP.

Переменная DEBUG

Общедоступная альтернатива для `NODE_DEBUG` — переменная `DEBUG`. Многие пакеты в npm проверяют переменную окружения `DEBUG`. Она имитирует стиль параметров, используемый `NODE_DEBUG`, поэтому вы можете указать список модулей или просмотреть их все в режиме `DEBUG='*'`. На рис. 9.8 показан проект из главы 4 в режиме `DEBUG='*'`.

```
➜ tldr git:(master) ✗ DEBUG="*" npm start

> tldr@1.0.0 start /Users/alex/Documents/Code/nodeinaction/ch04-what-is-a-node-web-app/tldr
> node index.js

  express:application set "x-powered-by" to true +0ms
  express:application set "etag" to 'weak' +3ms
  express:application set "etag fn" to [Function: wetag] +2ms
  express:application set "env" to 'development' +1ms
  express:application set "query parser" to 'extended' +0ms
  express:application set "query parser fn" to [Function: parseExtendedQueryString] +0ms
  express:application set "subdomain offset" to 2 +0ms
  express:application set "trust proxy" to false +0ms
  express:application set "trust proxy fn" to [Function: trustNone] +1ms
  express:application booting in development mode +0ms
  express:application set "view" to [Function: View] +0ms
  express:application set "views" to '/Users/alex/Documents/Code/nodeinaction/ch04-what-is-a-node-web-app/tldr/views' +0ms
  express:application set "jsonp callback name" to 'callback' +0ms
  express:router use / query +429ms
```

Рис. 9.8. Запуск приложения Express в режиме DEBUG='*'

Если вы хотите встроить функциональность `NODE_DEBUG` в свои собственные проекты, используйте встроенный метод `util.debuglog`:

```
const debuglog = require('util').debuglog('example');
debuglog('You can only see these messages by setting NODE_DEBUG=example!');
```

Чтобы создать нестандартные средства отладочного вывода, настроенные с переменной `DEBUG`, необходимо использовать пакет debug из npm (*www.npmjs.com/*

package/debug). Вы можете создать столько вариантов отладочного вывода, сколько потребуется. Представьте, что вы строите веб-приложение MVC. Вы можете создать отдельные подсистемы ведения журнала для моделей, представлений и контроллеров. Затем, когда тесты не пройдут, вы сможете указать журналы, необходимые для отладки конкретной части приложения. Листинг 9.22 (файл ch09-testing/debug-example/index.js в архиве кода) демонстрирует использование модуля debug.

Листинг 9.22. Использование пакета debug

```
const debugViews = require('debug')('debug-example:views');
const debugModels = require('debug')('debug-example:models');

debugViews('Example view message');
debugModels('Example model message');
```

Чтобы запустить этот пример и просмотреть отладочный журнал представлений, присвойте `DEBUG` значение `debug-example:views`:

```
DEBUG=debug-example:views node index.js
```

Последняя интересная возможность отладочных журналов — вы можете снабдить раздел отладочной информации префиксом `-`, чтобы исключить его из журналов:

```
DEBUG='* -debug-example:views' node index.js
```

Исключение определенных модулей означает, что вы по-прежнему можете использовать режим `*`, но убрать из вывода ненужные или слишком длинные разделы.

9.3.2. Получение расширенной трассировки стека

Если вы используете асинхронные операции (а их использует каждый, кто когда-либо писал код с асинхронными обратными вызовами или обещаниями), недостаточно подробная трассировка стека может создать проблемы. В таких случаях вам могут помочь пакеты npm. Например, при асинхронном выполнении обратных вызовов Node не сохраняет стек вызовов, из которого была поставлена в очередь операция. Чтобы убедиться в этом, создайте два файла: один, с именем async.js, определяет асинхронную функцию, а другой, с именем index.js, включает async.js. Следующий фрагмент хранится в файле async.js (файл ch09-testing/debug-stacktraces/async.js в архиве кода):

```
module.exports = () => {
  setTimeout(() => {
    throw new Error();
  })
};
```

А в файле index.js достаточно включить async.js:

```
require('./async.js')();
```

Если теперь запустить index.js командой `node index.js`, вы получите краткую трассировку стека, в которой не указана точка вызова ошибочной функции — только точка, в которой было сгенерировано исключение:

```
    throw new Error();
    ^

Error
    at null._onTimeout (async.js:3:11)
    at Timer.listOnTimeout (timers.js:92:15)
```

Чтобы получить отчет с более подробной информацией, установите пакет trace (*www.npmjs.com/package/trace*) и запустите его командой `node -r trace index.js`. Флаг `-r` приказывает Node включить модуль trace, прежде чем загружать что-либо еще.

У трассировки стека есть и другая проблема: она может быть чрезмерно подробной — например, если трассировка включает в себя слишком много внутренней информации Node. Для очистки трассировки стека используется пакет clarify (www.npmjs.com/package/clarify). Его также можно запустить с флагом `-r`:

```
$ node -r clarify index.js
    throw new Error();
    ^

Error
    at null._onTimeout (async.js:3:11)
```

Пакет clarify особенно полезен, если вы хотите включить трассировку стека в сигнальные сообщения об ошибках веб-приложений, пересылаемые по электронной почте.

Если вы выполняете в Node код, предназначенный для браузеров (например, как часть изоморфных веб-приложений), то для улучшения трассировки стека можно воспользоваться пакетом source-map-support (*www.npmjs.com/package/source-map-support*). Пакет может запускаться с флагом `-r`, но он также работает с некоторыми тестовыми фреймворками:

```
$ node -r source-map-support/register index.js
$ mocha --require source-map-support/register index.js
```

Когда вы в следующий раз будете мучиться с трассировкой стека, сгенерированной для асинхронного кода, воспользуйтесь такими инструментами, как trace и clarify. С их помощью вы сможете в полной мере использовать всю информацию, которую вам предоставляют V8 и Node.

9.4. Заключение

- Для написания модульных тестов требуется система исполнения тестов — такая, как Mocha.
- Node содержит встроенную библиотеку тестовых утверждений assert.
- Также существуют другие библиотеки тестовых утверждений — например, Chai и Should.js.
- Если вы хотите обойти выполнение некоторого кода (например, сетевых запросов), используйте Sinon.js.
- Sinon.js также позволяет следить за выполнением кода и контролировать выполнение некоторых функций и методов.
- Selenium используется для написания браузерных тестов, основанных на сценарном управлении реальными браузерами.

10 Развертывание и обеспечение доступности приложений Node

Разработка веб-приложения — одно дело, а запуск его в эксплуатацию — совсем другое. Для каждой веб-технологии существуют приемы, направленные на повышение надежности и производительности, и Node не является исключением из этого правила. В этой главе мы расскажем, как выбрать подходящую среду разработки для вашего приложения и как обеспечить доступность приложения.

В следующем разделе описаны различные типы сред, в которых может развертываться приложение. Затем мы займемся обеспечением высокой доступности приложений.

10.1. Хостинг Node-приложений

В веб-приложениях, которые разрабатываются в этой книге, используется сервер HTTP на базе Node. Браузер может взаимодействовать с приложением без выделенного сервера HTTP (такого, как Apache или Nginx). Впрочем, вы можете разместить сервер (например, Nginx) перед своим приложением, так что приложения Node часто могут размещаться в любой среде, где можно было запустить веб-сервер.

Облачные провайдеры, включая Heroku и Amazon, также поддерживают Node. Как следствие, существует три надежных и масштабируемых способа запуска приложений:

- платформа как сервис — приложение запускается на платформе Amazon, Azure или Heroku;
- сервер или виртуальная машина — приложение запускается на сервере UNIX или Windows в облаке, на машинах компании, предоставляющих услуги приватного хостинга, или на внутренних мощностях вашей компании;
- контейнер — приложение и любые другие ассоциированные сервисы запускаются с использованием программного контейнера (например, Docker).

Выбрать нужный вариант бывает тяжело — особенно если учесть, что сначала опробовать их не всегда просто. Варианты не привязаны к конкретным провайдерам: например, Amazon и Azure могут предоставить все стратегии развертывания. В этом разделе описаны требования к каждому варианту, их достоинства и недостатки. К счастью, для каждого варианта доступны бесплатные или экономрешения, поэтому все варианты доступны как разработчикам-любителям, так и профессионалам.

10.1.1. Платформа как сервис

В схеме «платформа как сервис» (PaaS, Platform as Service) приложение обычно готовится к развертыванию: вы должны зарегистрироваться для использования сервиса, создать новое приложение, а затем добавить удаленный репозиторий Git в проект. При отправке в этот удаленный репозиторий происходит развертывание приложения. По умолчанию приложение выполняется в одном контейнере (точное определение контейнера зависит от провайдера), а в случае сбоя сервис пытается перезапустить приложение. Вы получаете ограниченный доступ к журналам, а также веб-интерфейс и интерфейс командной строки для управления приложением. Чтобы провести масштабирование, следует запустить дополнительные экземпляры приложения, что требует дополнительных затрат. В табл. 10.1 представлена сводка характеристик типичных предложений PaaS.

Таблица 10.1. Характеристики PaaS

Простота использования	Высокая
Возможности	Развертывание посредством отправки в GiT, простое горизонтальное масштабирование
Инфраструктура	Абстрагированная/черный ящик
Пригодность для коммерческого использования	Хорошая: приложения обычно изолированы по сети
Стоимость*	Низкий трафик: $$, популярный сайт: $$$$
Провайдеры	Heroku, Azure, AWS Elastic Beanstalk

* $: дешево, $$$$$: дорого

У провайдеров PaaS есть свои предпочтительные базы данных. Для Heroku это PostgreSQL, а для Azure — SQL Database. Подробности подключения к базе данных определяются переменными окружения, поэтому вы можете подключаться без добавления учетных данных в исходный код проекта. Вариант PaaS отлично подходит для разработчиков-любителей, потому что он обходится недорого (а иногда даже бесплатно) для небольших проектов с незначительным трафиком.

Некоторые провайдеры проще в использовании, чем другие: платформа Heroku в высшей степени проста для программистов, знакомых с Git, даже если они не обладают навыками системного администрирования или DevOps. Системы PaaS обычно знают, как следует запускать проекты, созданные такими популярными инструментами, как Node, Rails и Django, так что все происходит почти автоматически.

Пример: Node в Heroku за 10 минут

В этом разделе мы развернем приложение в Heroku. Использовав настройки Heroku по умолчанию, мы развернем приложение в одном облегченном контейнере Linux. Для развертывания базового приложения Node в Heroku потребуются следующие предварительные условия:

- развертываемое приложение;
- учетная запись Heroku: *https://signup.heroku.com/*;
- Heroku CLI: *https://devcenter.heroku.com/articles/heroku-cli*.

Когда все эти элементы будут готовы, войдите в Heroku в командной строке:

```
heroku login
```

Heroku предлагает ввести адрес электронной почты и пароль Heroku. Затем создайте простое приложение Express:

```
mkdir heroku-example
npm i -g express-generator
express
npm i
```

Выполните команду `npm start`, введите адрес *http://localhost:3000* и убедитесь в том, что все работает правильно. На следующем шаге создайте репозиторий Git и приложение Heroku:

```
git init
git add .
git commit –m 'Initial commit'
heroku create
git push heroku master
```

Команда выводит случайно сгенерированный URL-адрес вашего приложения и репозиторий Git. Каждый раз, когда вы хотите развернуть приложение, зафиксируйте изменения в Git и выполните команду `git push heroku master`. Вы можете изменить URL и имя приложения командой `heroku rename`.

Теперь откройте URL *herokuapp.com* с предыдущего шага, чтобы увидеть приложение Express. Чтобы просмотреть журналы приложения, выполните команду `heroku logs`, а для получения доступа к командному интерпретатору в контейнере приложения выполните команду `heroku run bash`.

Heroku предоставляет простой и быстрый механизм запуска приложений Node. Обратите внимание: никакая настройка, относящаяся к Node, не потребуется — Heroku запускает базовые приложения Node без дополнительных изменений конфигурации. Впрочем, в некоторых случаях требуется более высокая степень контроля над средой выполнения, так что в следующем разделе рассматривается второй вариант: использование серверов для хостинга приложений Node.

10.1.2. Серверы

Использование собственного сервера обладает рядом преимуществ перед PaaS. Вместо того чтобы беспокоиться, где будет работать база данных, вы можете установить PostgreSQL, MySQL или даже Redis на том же сервере. Установить можно все, что угодно: нестандартное программное обеспечение ведения журнала, серверы HTTP, прослойки кэширования — все зависит от вас. В табл. 10.2 представлена сводка характеристик решений с собственным сервером.

Таблица 10.2. Характеристики решений с собственным сервером

Простота использования	Низкая
Возможности	Полный контроль над всем стеком, запуск собственной базы данных и уровня кэширования
Инфраструктура	Открыта для разработчика (или системного администратора/DevOps)
Пригодность для коммерческого использования	Хорошая, если у вас имеется персонал, способный заниматься сопровождением сервера
Стоимость	Малая виртуальная машина: $, большой сервер: $$$$$
Провайдеры	Azure, Amazon, компании, предоставляющие сервис хостинга

Существует несколько вариантов сопровождения сервера. Вы можете получить недорогую виртуальную машину у такой компании, как Linode или Digital Ocean; это будет полноценный сервер, который вы сможете настроить по своему усмотрению, но он будет совместно использовать ресурсы одного комплекта оборудования с другими виртуальными машинами. Также можно купить собственное оборудование или взять сервер в аренду. Некоторые компании-хостеры предоставляют сервис управляемого хостинга, оказывая поддержку в сопровождении операционной системы сервера.

Необходимо решить, какую операционную систему вы собираетесь использовать. Debian существует в нескольких разновидностях, Node также хорошо работает с Windows и Solaris, поэтому выбор оказывается более сложным, чем кажется на первый взгляд.

Другое критическое решение — организация доступа к вашему приложению: трафик может перенаправляться с портов 80 и 443 вашему приложению, но вы также можете разместить Nginx перед приложением для прокси-обработки запросов и (возможно) предоставления статических файлов.

Перемещение кода из репозитория на сервер также может осуществляться разными способами. Вы можете вручную скопировать файлы при помощи scp, sftp или rsync, или же воспользоваться Chef для управления несколькими серверами и управления версиями. Некоторые специалисты создают Git-перехватчик в стиле Heroku, который автоматически обновляет приложение на сервере в зависимости от отправки данных в определенные ветви Git.

Важно понимать, что управление собственным сервером — достаточно непростое дело. Настройка требует значительных усилий, а на сервере приходится устанавливать новейшие исправления ошибок ОС и обновления безопасности. Разработчика-любителя это может отпугнуть — однако в процессе настройки вы узнаете много нового, а возможно, проявите интерес к области DevOps.

Для запуска приложений Node на виртуальной машине или полном сервере не нужно ничего особенного. Если вы хотите ознакомиться с некоторыми приемами, используемыми для запуска приложений Node на сервере и обеспечения их работы в течение длительного времени, переходите к разделу 10.2. В противном случае продолжайте читать — в следующем разделе приводится дополнительная информация о Node и Docker.

10.1.3. Контейнеры

Использование программных контейнеров представляет собой некий механизм виртуализации ОС, который автоматизирует процесс развертывания приложений. Самый известный из таких проектов — Docker — распространяется с открытым кодом, но также содержит коммерческие функции, упрощающие развертывание приложений для реальной эксплуатации. В табл. 10.3 представлена сводка характеристик решений с контейнером.

Таблица 10.3. Характеристики решений с контейнерами

Простота использования	Средняя
Возможности	Полный контроль над всем стеком, запуск собственной базы данных и уровня кэширования, возможность перехода на других провайдеров и локальные машины
Инфраструктура	Открыта для разработчика (или системного администратора/DevOps)
Пригодность для коммерческого использования	Превосходная: развертывание на управляемом хосте, хосте Docker или собственном центре обработки данных

Стоимость	$$$
Провайдеры	Azure, Amazon, Docker Cloud, Google Cloud Platform (cKubernetes), компании, предоставляющие сервис хостинга для управления контейнерами Docker

Docker позволяет определять приложения в формате *образов* (images). Если вы построили типичную систему управления контентом (CMS), которая имеет микросервис для обработки графики, основной сервис для хранения данных приложений, и базу данных, ее можно развернуть с четырьмя разными образами Docker:

- образ 1 — микросервис масштабирования графики, отправляемой CMS;
- образ 2 — PostgreSQL;
- образ 3 — основное веб-приложение CMS с административным интерфейсом;
- образ 4 — открытый интерфейс веб-приложения.

Так как Docker распространяется с открытым кодом, вы не ограничиваетесь одним провайдером для развертывания приложений на базе Docker. Также для развертывания образов можно воспользоваться Amazon Elastic Beanstalk, Docker Cloud и даже Microsoft Azure. Amazon также предоставляет сервисы ECS (EC2 Container Service) и AWS CodeCommit для облачных репозиториев Git, которые могут быть развернуты в Elastic Beanstalk по аналогии с Heroku.

Самый замечательный факт при использовании контейнеров заключается в том, что после контейнеризации приложения вы можете вызвать его новый экземпляр всего одной командой. Если у вас появился новый компьютер, вы просто обращаетесь к репозиторию приложения, устанавливаете Docker локально, а затем выполняете сценарий для запуска приложения. Так как приложение имеет четко определенную процедуру развертывания, вам и вашим коллегам будет проще понять, как ваше приложение должно выполняться за пределами локальной среды разработки.

Пример: запуск приложений Node с Docker

Пример: *https.//nodejs.org/en/docs/guides/nodejs-docker-webapp/*.

Чтобы запустить приложение Node в контейнере Docker, сначала следует выполнить ряд подготовительных действий.

1. Установите Docker: *https.//docs.docker.com/engine/installation/*.
2. Создайте приложение Node. За информацией о том, как быстро создать приложение Express, обращайтесь к разделу 10.1.1.
3. Добавьте в проект новый файл с именем Dockerfile.

Файл Dockerfile сообщает Docker, как построить образ вашего приложения, как установить приложение и запустить его. Мы используем официальный Docker-образ

Node (*https://hub.docker.com/_/node/*), включив в Dockerfile команду `FROM node:boron` и затем выполнив команды `npm install` и `npm start` с инструкциями RUN и CMD. Следующий фрагмент содержит полный файл Dockerfile для простых приложений Node:

```
FROM node:argon

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY package.json /usr/src/app/
RUN npm install

COPY . /usr/src/app

EXPOSE 3000
CMD ["npm", "start"]
```

После того как вы создали файл Dockerfile, выполните команду `docker build` (*https://docs.docker.com/engine/reference/commandline/build/*) в терминале, чтобы создать образ приложения. Для построения достаточно указать только каталог, так что, если вы находитесь в каталоге приложения Express, введите команду `docker build`. Команда строит образ и отправляет его демону Docker.

Выполните команду `docker images`, чтобы просмотреть список изображений. Получите идентификатор изображения, а затем запустите приложение командой `docker run -p 8080:3000 -d <image ID>`. Мы привязали внутренний порт (3000) к 8080 на локальном хосте, поэтому для обращения к приложению в браузере вводится адрес *http://localhost:8080*.

10.2. Основы развертывания

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 из репозитория Git, а также выясним, как с помощью программы Forever сохранять приложение работоспособным и работающим. Временно развернутое приложение не сохраняется после перезагрузки сервера, но зато быстро создается.

10.2.1. Развертывание из репозитория Git

В этом разделе мы познакомимся с базовым развертыванием из репозитория Git, 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 репозитория контроля версий на сервер файлов приложения, включая Node-сценарии, изображения и таблицы стилей CSS.
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
```

Как и PHP, Node не может выполняться в фоновом режиме. Поэтому базовое развертывание, про которое мы упомянули, требует открытия SSH-соединения. Как только подключение SSH будет закрыто, выполнение приложения завершится. К счастью, поддерживать выполнение приложения очень легко, если прибегнуть к помощи одного простого инструмента.

АВТОМАТИЗИРОВАННОЕ РАЗВЕРТЫВАНИЕ

Существует много способов автоматизации развертывания 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 (https://blog.evantahler.com/deploying-node-js-applications-with-capistrano-af675cdaa7c6#.8r9v0kz3l).

10.2.2. Поддержание работы Node-приложения

Предположим, вы создали персональный блог с помощью специального приложения для разработки блогов Ghost (*https://ghost.org/*) и собираетесь развернуть его. При этом нужно гарантировать продолжение выполнения приложения даже в случае разрыва подключения SSH.

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 соединения SSH и дополнительно обеспечивает его перезапуск после сбоя. На рис. 10.1 концептуально показано, как работает программа Forever.

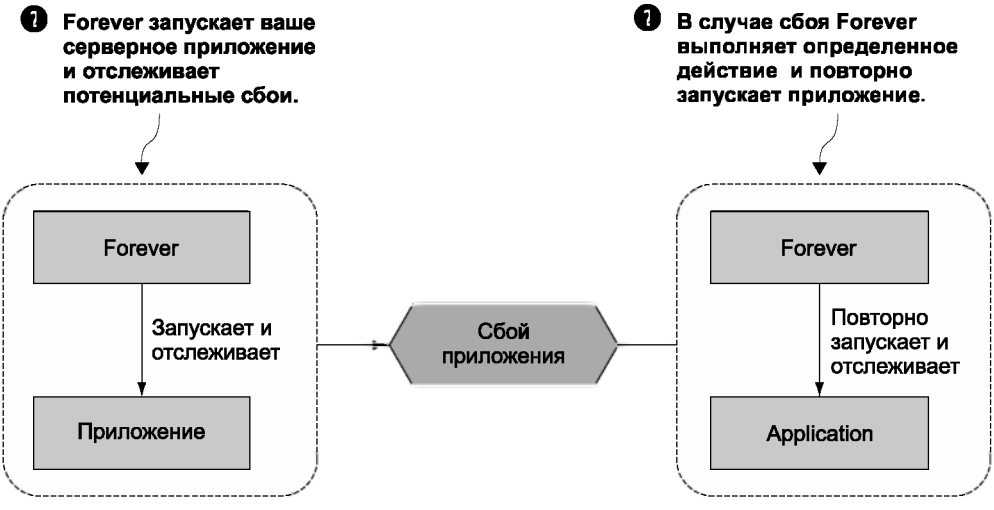


Рис. 10.1. Forever обеспечивает выполнение приложения даже в случае сбоя

Для выполнения глобальной установки Forever используется команда `sudo`.

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. При первом выполнении команды sudo понадобится ввести пароль. Только тогда команда, указанная после пароля, будет выполнена.

Чтобы установить Forever, выполните следующую команду:

```
npm install -g forever
```

После завершения установки воспользуйтесь программой Forever для запуска блога и поддержания его в работоспособном состоянии:

```
forever start server.js
```

Если нужно остановить работу блога, введите команду Forever `stop`:

```
forever stop server.js
```

Для получения списка 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.

Для запуска 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.

10.3. Максимизация времени доступности и производительности приложений

Когда приложение Node готово к публикации, нужно убедиться в том, что оно запускается и останавливается при запуске и остановке сервера, 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 необходимых шагах по максимизации производительности приложения. Например, если приложение выполняется на сервере с четырехъядерным процессором, не стоит загружать только одно ядро. Если при использовании одного ядра резко возрастет трафик веб-приложения, вычислительных возможностей одного ядра может не хватить для обслуживания трафика, в результате ваше приложение не сможет стабильно отвечать на запросы.

Старайтесь не использовать Node с целью хостинга статических файлов для крупномасштабных сайтов. Платформа Node «заточена» под интерактивные приложения, такие как веб-приложения и протоколы TCP/IP, поэтому не может предоставлять статические файлы так же эффективно, как специально предназначенное для этого программное обеспечение. Для предоставления статических файлов используются такие решения, как система Nginx (*http://nginx.org/en/*), которая оптимизирована для решения подобных задач. Также можно выгрузить все статические файлы в сеть доставки контента (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 доступности и работоспособности приложения при перезапусках и сбоях;
- использование кластерного API-интерфейса в Node для многоядерных процессоров;
- предоставление статических файлов приложения Node с помощью Nginx.

Начнем мы с рассмотрения очень мощного и удобного инструмента Upstart, предназначенного для поддержания доступности приложений.

10.3.1. Поддержание доступности приложения с Upstart

Допустим, вы в восторге от своего приложения и собираетесь выпустить его на рынок. Вы хотите иметь железную гарантию того, что при перезапуске сервера вы потом не забудете перезапустить приложение. Кроме того, нужно принять меры к тому, чтобы в случае сбоя приложения оно не только автоматически перезапускалось, но и информация о сбое была записана, а вы были оповещены об этом, чтобы диагностировать причину сбоя.

Проект Upstart (*http://upstart.ubuntu.com*) предлагает элегантный способ управления запуском и остановкой любого Linux-приложения, включая приложение Node. Поддержку программы Upstart обеспечивают, в частности, современные версии платформ Ubuntu и CentOS. Альтернативное решение для macOS основано на создании файлов launchd (node-launchd в npm позволяет это сделать), а в системе Windows для этого используется подсистема Windows Services, поддержка которой обеспечивается пакетом node-windows в npm.

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
```

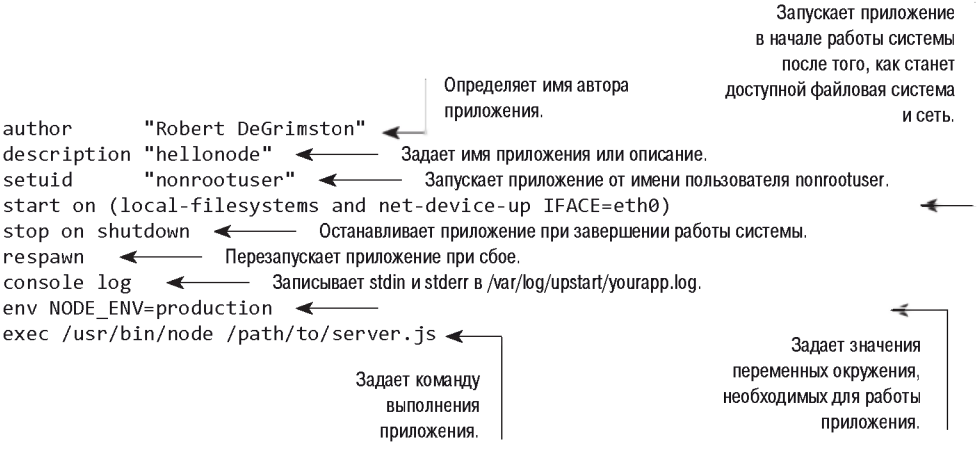
После установки Upstart для каждого из ваших приложений нужно добавить конфигурационный файл программы Upstart. Указанные файлы создаются в каталоге /etc/init и имеют названия вида *имя_приложения*.conf. Конфигурационные файлы не обязательно должны помечаться как исполняемые.

С помощью следующей команды создается пустой конфигурационный файл программы Upstart для учебного приложения, рассматриваемого в этой главе:

```
sudo touch /etc/init/hellonode.conf
```

Далее добавьте в конфигурационный файл код из листинга 10.1. Этот код запустит приложение после запуска сервера и остановит его после остановки сервера. То, что должно запускаться программой Upstart, находится в разделе `exec`.

Листинг 10.1. Типичный конфигурационный файл Upstart



С этим конфигурационным файлом ваш процесс будет работать 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 будет управлять своевременной архивацией и обновлением журнала.

После создания конфигурационного файла программы Upstart запустите приложение следующей командой:

```
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 электронной документации (*http://upstart.ubuntu.com/cookbook/*).

Upstart и Respawn

Если указан параметр `respawn`, Upstart в случае сбоя по умолчанию будет постоянно пытаться перезагрузить приложение, пока количество попыток перезагрузки не достигнет 10 в течение 5 секунд. Чтобы изменить это ограничение, воспользуйтесь параметром `respawn limit` *КОЛИЧЕСТВО ИНТЕРВАЛ*. Здесь *КОЛИЧЕСТВО* — это

количество попыток внутри заданного (в секундах) *ИНТЕРВАЛА* 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 (режим 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 работоспособности 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, логично направить усилия на следующий аспект — быстродействие. И в этом нам поможет кластерный API платформы Node.

10.3.2. Кластерный API

Несмотря на то что у большинства современных процессоров несколько ядер, процесс Node использует только одно из них. Если вы размещаете Node-приложение на сервере и хотите максимизировать загрузку сервера, можно вручную запустить несколько экземпляров приложения на разных TCP/IP-портах, а затем с помощью системы балансировки загрузки распределить веб-трафик по этим экземплярам. Тем не менее такое решение потребует значительных усилий.

Чтобы упростить использование нескольких ядер одним приложением, в Node был включен кластерный API. С его помощью приложение может легко запустить несколько рабочих процессов, которые одновременно будут выполнять одни и те же действия на разных ядрах и использовать один и тот же TCP/IP-порт. На рис. 10.2 показано, как на сервере с четырехъядерным процессором организуется работа приложения с применением кластерного API.

Код из листинга 10.2 автоматически порождает главный процесс и дополнительно рабочие процессы (работники) для каждого ядра процессора.

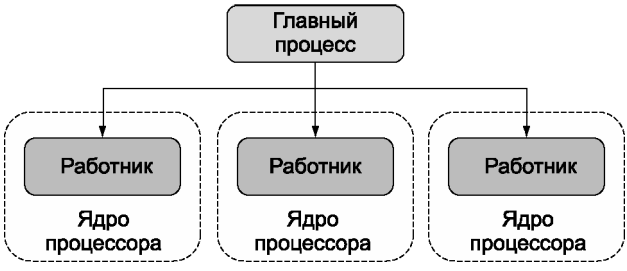


Рис. 10.2. Для выполнения на четырехъядерном процессоре главный процесс порождает трех работников

Листинг 10.2 Демонстрация кластерного API платформы Node

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;  <-- Определяет количество ядер процессора.
if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {  <-- Создает ответвление для каждого ядра.
    cluster.fork();
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log('Worker %s died.', worker.process.pid);
  });
} else {
  http.Server((req, res) => {  <-- Определяет работу, которая должна выполняться каждым работником.
    res.writeHead(200);
    res.end('I am a worker running in process: ' + process.pid);
  }).listen(8000);
}
```

Поскольку главный процесс и работники выполняются в разных системных процессах (это необходимо для того, чтобы они выполнялись на разных ядрах), они не могут обмениваться данными состояния через глобальные переменные. Тем не менее кластерный API предоставляет мастеру и работникам механизм передачи данных.

В листинге 10.3 приведен пример обмена сообщениями между главным процессом и работниками. Счетчик всех запросов хранится у мастера, и когда какой-то работник сообщает о завершении обработки запроса, эта информация передается каждому работнику.

Листинг 10.3. Демонстрация кластерного API платформы Node

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
const workers = {};
let requests = 0;
```

```
if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    workers[i] = cluster.fork();
    ((i) => {
      workers[i].on('message', (message) => {   <-- Прослушивает сообщения от работника.
        if (message.cmd == 'incrementRequestTotal') {
          requests++;   <-- Увеличивает счетчик запросов.
          for (var j = 0; j < numCPUs; j++) {
            workers[j].send({   <-- Передает новый счетчик запросов каждому работнику.
              cmd: 'updateOfRequestTotal',
              requests: requests
            });
          }
        }
      });
    })(i);   <-- Использует замыкание для сохранения значения счетчика.
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log('Worker %s died.', worker.process.pid);
  });
} else {
  process.on('message', (message) => {   <-- Прослушивает сообщения от главного процесса.
    if (message.cmd === 'updateOfRequestTotal') {
      requests = message.requests;   <-- Обновляет счетчик запросов по данным сообщения.
    }
  });
  http.Server((req, res) => {
    res.writeHead(200);
    res.end(`Worker ${process.pid}: ${requests} requests.`);
    process.send({ cmd: 'incrementRequestTotal' });   <-- Сообщает главному процессу о необходимости увеличения счетчика запросов.
  }).listen(8000);
}
```

С помощью кластерного API-интерфейса платформы Node очень просто создавать приложения, способные использовать возможности современного оборудования.

10.3.3. Хостинг статических файлов и представительство

Платформа Node в первую очередь предназначена для предоставления динамического веб-контента, поэтому она не столь эффективна, когда речь идет о статическом контенте, таком как изображения, таблицы CSS-стилей или клиентский код JavaScript. Предоставление статических файлов по протоколу 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 предоставления статических файлов, можно легко установить вместе с Node именно с целью предоставления статических файлов. В типичной

конфигурации Nginx/Node сервер Nginx сначала обрабатывает каждый веб-запрос и возвращает Node те запросы, которые не относятся к статическим файлам. Соответствующая конфигурация представлена на рис. 10.3.

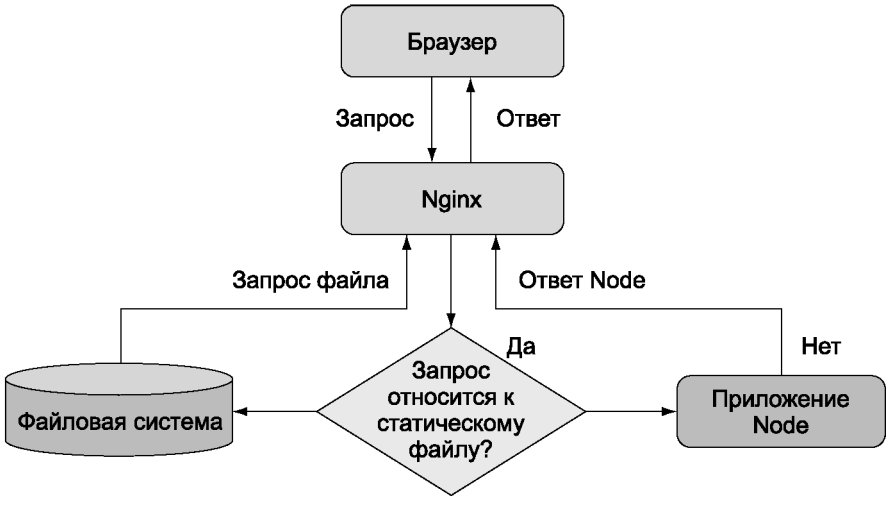


Рис. 10.3. Nginx-сервер можно использовать как посредника, который быстро возвращает статические ресурсы веб-клиентам

Реализует эту конфигурацию код из листинга 10.4, который помещается в раздел `http` конфигурационного файла сервера Nginx. Конфигурационный файл обычно хранится в каталоге /etc Linux-сервера и называется /etc/nginx/nginx.conf.

Листинг 10.4. Конфигурационный файл для использования сервера Nginx в качестве посредника для Node.js и предоставления статических файлов

```
http {
  upstream my_node_app {
    server 127.0.0.1:8000;  <-- IP-адрес и порт приложения Node.
   }
  server {
    listen 80;  <-- Порт, на котором посредник принимает запросы.
    server_name localhost domain.com;
    access_log /var/log/nginx/my_node_app.log;
    location ~ /static/ {  <-- Обрабатывает запросы файлов для URL-путей, начинающихся с /static/.
      root /home/node/my_node_app;
      if (!-f $request_filename) {
        return 404;
      }
    }
    location / {  <-- Определяет URL-путь, по которому будет отвечать посредник.
      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;
    }
  }
}
```

Использование Nginx для предоставления статических веб-ресурсов гарантирует, что Node будет делать только то, что лучше всего умеет.

10.4. Заключение

- Для хостинга приложений Node могут использоваться провайдеры PaaS, выделенные серверы, виртуальные приватные серверы и облачный хостинг.
- Для быстрого развертывания приложений Node на серверах Linux используются программы Forever и Upstart.
- Для повышения быстродействия приложений модуль Node cluster позволяет порождать дополнительные процессы.

III За пределами веб-разработки

Миллионы людей пользуются приложениями, построенными на базе Node. Если вы когда-либо использовали Slack или Visual Studio Node, значит, вы уже работали с приложениями Node. В этой части рассматривается Electron и модули для написания программ командной строки на базе Node. Если вы когда-либо хотели создать приложение, которое работало бы в Linux, macOS или Windows, — теперь у вас есть такая возможность.

11 Написание приложений командной строки

Программы командной строки Node используются во множестве областей, от средств автоматизации проектов (таких, как Gulp или Yeoman) до парсеров XML и JSON. Если вас когда-либо интересовало, как строить программы командной строки на базе Node, вы найдете в этой главе все необходимое для начала работы. Вы узнаете, как программы Node получают аргументы командной строки и как реализуется ввод/вывод с использованием каналов. Мы также привели полезные советы, которые помогут вам пользоваться режимом командной строки более эффективно.

Хотя написать программу командной строки с Node несложно, очень важно соблюдать некоторые соглашения, принятые в сообществе. В этой главе описаны многие из этих соглашений, поэтому другие пользователи смогут работать с вашими программами без излишней документации.

11.1. Соглашения и философия

Важной частью разработки программ командной строки является понимание соглашений, встречающихся в проверенных временем программах. В качестве реального примера возьмем Babel:

```
  Usage: babel [options] <files ...>

  Options:

    -h, --help                         output usage information
    -f, --filename [filename]          filename to use when reading from
stdin
  [ ... ]
    -q, --quiet                        Don't log anything
    -V, --version                      output the version number
```

Здесь стоит обратить внимание на несколько фактов. Первое — использование ключей `-h` и `--help` для вывода справочной информации: флаг — используется во

многих программах. Второй флаг, `-f`, предназначен для имени файла; эта мнемоника легко запоминается. Многие флаги основаны на мнемонических сокращениях. Также часто встречается флаг `-q` для скрытого (quiet) вывода, как и флаг `-v` для вывода версии программы. Все эти флаги должны поддерживаться вашими приложениями.

Впрочем, пользовательский интерфейс — не просто соглашение. Дефис и двойной дефис (`--`) входят в спецификацию интерфейса Open Group Utility Conventions[1]. В документе даже указано, как они должны использоваться:

- рекомендация 4 — все ключи должны начинаться с префикса `-`;
- рекомендация 10 — первый аргумент `--`, не являющийся префиксом ключа, должен интерпретироваться как признак конца списка ключей. Все последующие аргументы должны рассматриваться как операнды, даже если они начинаются с символа `-`.

Другой аспект проектирования приложений командной строки — философия. Она уходит корнями к создателям UNIX, которые хотели создавать «компактные, мощные инструменты» для использования в простом текстовом интерфейсе.

> Философия UNIX такова: писать программы, которые делают что-то одно — и делают это хорошо. Писать программы, которые работают в сочетании друг с другом. Писать программы для работы с текстовыми потоками, потому что это универсальный интерфейс.
>
> *Дуг Макилрой*[2]

В этой главе представлен широкий обзор средств командной строки и общепринятых соглашений UNIX, чтобы вы могли разрабатывать инструменты командной строки, которыми могут пользоваться другие люди. Также будут приведены некоторые рекомендации, относящиеся к системе Windows, но в основном ваши программы Node по умолчанию будут кросс-платформенными.

ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: ВЫВОД СПРАВКИ

Если у вас возникнут проблемы с использованием командной строки, попробуйте ввести команду man <команда>. Эта команда загружает страницу с документацией команды.

Если вы не помните название команды, введите команду apropos <условие> для проведения поиска по базе данных системных команд.

[1] «The Open Group Base Specifications Issue 7», http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap11.html.

[2] «Basics of the Unix Philosophy», www.catb.org/~esr/writings/taoup/html/ch01s06.html.

11.2. Знакомство с parse-json

Для программистов JavaScript одно из самых полезных приложений читает разметку JSON и сообщает, является ли эта разметка действительной. В этой главе мы воссоздадим эту программу.

Сначала определимся, как должна выглядеть командная строка такого приложения. В следующем фрагменте приведен пример запуска такой программы:

```
node parse-json.js -f my.json
```

Прежде всего необходимо понять, как извлечь аргументы программы `-f my.json` из командной строки. Входные данные должны читаться из потока `stdin`. Ниже рассказано, как решаются эти задачи.

11.3. Аргументы командной строки

Многие — хотя и не все — программы командной строки получают аргументы. В Node существует встроенный механизм для обработки аргументов, но сторонние модули npm предоставляют дополнительные возможности. Эти возможности пригодятся для реализации некоторых распространенных соглашений.

11.3.1. Разбор аргументов командной строки

Для работы с аргументами командной строки можно использовать массив `process.argv`. Элементы массива содержат строки, передаваемые командному интерпретатору при выполнении команды. Таким образом, разбив команду на части, вы можете определить, какая подстрока будет храниться в каждом элементе массива. Элемент `process.argv[0]` содержит подстроку `node`, элемент `process.argv[1]` — подстроку `parse-json.js`, элемент `[2]` — подстроку `-f`, и т. д.

Читателям, работавшим с приложениями командной строки, уже встречались аргументы с префиксами `-` или `--`. Эти префиксы определяют специальные соглашения для передачи значений приложениям: `--` обозначает полное имя ключа, а префикс `-` – односимвольное сокращение. Хорошим примером служит исполняемый файл командной строки npm с ключами `-h` и `--help`.

СОГЛАШЕНИЯ АРГУМЕНТОВ

Также на практике широко применяются следующие соглашения по поводу аргументов:

- `--version` — вывод версии приложения;
- `-y` или `--yes` — для пропущенных ключей используются значения по умолчанию.

Альтернативные имена аргументов (такие, как `-h` и `--help`) затрудняют разбор строки при поддержке нескольких ключей. К счастью, для разбора аргументов существует специальный модуль, который называется yargs. Следующий фрагмент показывает, как yargs работает в простейшем случае. От вас потребуется совсем немного: включите `yargs` вызовом `require` и обратитесь к свойству `argv` для анализа аргументов, переданных сценарию:

```
const argv = require('yargs').argv;
console.log({ f: argv.f });
```

На рис. 11.1 показано, чем встроенная версия аргументов командной строки Node отличается от объекта, генерируемого модулем yargs.

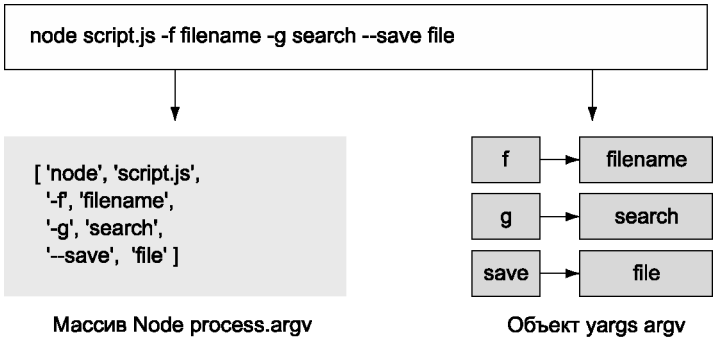


Рис. 11.1. Массив Node argv и объект yargs argv

Хотя объект с информацией параметров полезен, он не предоставляет особой структуры для проверки аргументов и генерирования сообщений об использовании. В следующем разделе показано, как вывести описания аргументов и проверить их значения.

11.3.2. Проверка аргументов

Модуль yargs включает в себя методы для проверки аргументов. Листинг 11.1 демонстрирует использование yargs для разбора аргумента `-f`, который понадобится вашему парсеру JSON, а также используются методы `describe` и `nargs` для контроля за ожидаемым форматом аргументов.

Листинг 11.1. Использование yargs для разбора аргументов командной строки

```
const readFile = require('fs').readFile;
const yargs = require('yargs');
const argv = yargs
  .demand('f')  <-------  Требует передачи ключа -f для выполнения.
```

```
  .nargs('f', 1)  <-------  Сообщает yargs, что за -f должен следовать один аргумент.
  .describe('f', 'JSON file to parse')
  .argv;
const file = argv.f;
readFile(file, (err, dataBuffer) => {
  const value = JSON.parse(dataBuffer.toString());
  console.log(JSON.stringify(value));
});
```

Использовать yargs удобнее, чем работать с массивом `process.argv`, — прежде всего из-за возможности установления правил. В листинге 11.1 метод `demand` требует обязательной передачи аргумента, после чего метод `nargs` объявляет, что единственным аргументом будет файл JSON. Чтобы программой было проще пользоваться, yargs также можно передать текст с описанием. По соглашению текст с описанием выводится при вызове программы с ключом `-h` или `--help`. Вы можете добавить эти ключи с помощью yargs, как в следующем фрагменте:

```
yargs
  // ...
  .usage('parse-json [options]')
  .help('h')
  .alias('h', 'help')
  // ...
```

Теперь парсер JSON может получить аргумент с именем файла и обработать файл. Впрочем, работа с файлами в этом проекте еще не завершена, потому что программа также должна уметь получать данные из `stdin`. В следующем разделе вы узнаете, как реализуется это стандартное соглашение UNIX.

ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: ИСТОРИЯ

Командный интерпретатор хранит журнал команд, которые вводились ранее. Введите команду history, чтобы просмотреть историю команд; часто этот ключ сокращается до h.

11.3.3. Передача stdin в виде файла

Если в качестве аргумента файла передается дефис (`-f -`), это означает, что данные должны быть получены из `stdin`. Это еще одно распространенное соглашение командной строки. Используйте пакет mississippi, чтобы легко реализовать эту возможность. При этом перед вызовом `JSON.parse` необходимо выполнить конкатенацию всех данных, передаваемых приложению, потому что метод ожидает получить для разбора полную строку JSON. С модулем mississippi пример выглядит так, как показано в листинге 11.2.

Листинг 11.2. Чтение файла из stdin

```
#!/usr/bin/env node
const concat = require('mississippi').concat;
const readFile = require('fs').readFile;
const yargs = require('yargs');
const argv = yargs
  .usage('parse-json [options]')
  .help('h')
  .alias('h', 'help')
  .demand('f') // Ключ -f необходим для выполнения
  .nargs('f', 1) // Сообщает yargs, что после -f следует 1 аргумент
  .describe('f', 'JSON file to parse')
  .argv;
const file = argv.f;
function parse(str) {
  const value = JSON.parse(str);
  console.log(JSON.stringify(value));
}
if (file === '-') {
  process.stdin.pipe(concat(parse));
} else {
  readFile(file, (err, dataBuffer) => {
    if (err) {
      throw err;
    } else {
      parse(dataBuffer.toString());
    }
  });
}
```

Этот код загружает модуль mississippi и назначает псевдоним `concat`. После этого `concat` используется с потоком `stdin`. Так как mississippi получает функцию с полным набором данных, мы можем использовать исходную функцию `parse` из листинга 11.1. Это происходит только в том случае, если вместо имени файла указан символ `-`.

11.4. Использование программ командной строки с npm

Любое приложение, которое вы хотите сделать доступным для других пользователей, легко устанавливается через npm. Чтобы передать npm информацию о приложении командной строки, проще всего воспользоваться полем `bin` в файле package.json. Это поле заставляет npm установить исполняемый файл, доступный для любых сценариев в текущем проекте. Поле `bin` также приказывает npm установить исполняемый файл глобально при использовании команды `npm install --global`. Данная возможность полезна не только для разработчиков Node, но и для всех остальных, кто захочет использовать ваши сценарии.

Этот фрагмент и строка `#!/usr/bin/env node` в листинге 11.2 — все, что необходимо для примера с парсером JSON этой главы:

```
...
    "name": "parse-json",
    "bin": {
      "parse-json": "index.js"
    },
...
```

При установке пакета командой `npm install -global` команда `parse-json` становится доступной на общесистемном уровне. Чтобы опробовать ее, откройте окно терминала (или окно командной строки в Windows) и введите команду `parse-json`. Обратите внимание: этот способ работает даже в Windows, потому что npm автоматически устанавливает обертку, обеспечивающую ее прозрачную работу в Windows.

11.5. Связывание сценариев с каналами

Программа parse-json проста — она получает текст и проверяет его на правильность формата. А если у вас имеются другие инструменты командной строки, с которыми должна использоваться эта программа? Представьте, что у вас имеется программа, которая может включать в себя цветовое выделение синтаксиса в файлы JSON. Было бы замечательно, если бы разметку JSON можно было сначала разобрать, а уже потом применять цветовое выделение. В этом разделе вы узнаете о каналах, которые могут делать это — и многое другое.

Программа parse-json и другие программы будут использоваться для организации необычной последовательности операций с применением каналов. Командные интерпретаторы Windows и UNIX различаются, но основные моменты (к счастью) остаются неизменными. Некоторые различия проявляются при отладке, но они не должны повлиять на вашу работу при написании приложений командной строки.

11.5.1. Передача данных parse-json

Для связывания приложений командной строки чаще всего применяется механизм *каналов* (pipes). Каналы берут поток `stdout` приложения и присоединяют его к потоку `stdin` другого процесса. Каналы являются центральным элементом межпроцессных коммуникаций: организации взаимодействия между программами. Вы можете обратиться к содержимому `stdin` в Node, используя обозначение `process.stdin`, потому что этот поток доступен для чтения. Следующий фрагмент разбирает данные JSON, поступающие из `stdin`:

```
echo "[1,2,3]" | parse-json -f -
```

Обратите внимание на символ `|`. Он сообщает командному интерпретатору, что команда `echo '{}'` должна направить свой вывод в поток `stdin` программы parse-json.

ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: КОМБИНАЦИИ КЛАВИШ

Для выполнения поиска в истории команд можно связать команду history с grep:

history | grep node

Еще удобнее для обращения к предыдущим командам использовать клавиши со стрелками ↑ и ↓ на клавиатуре. Пользователи пользуются этими клавишами постоянно — но есть еще лучший способ! Комбинация клавиш Ctrl+R, выполняющая рекурсивный поиск по истории команд, позволяет находить длинные команды по частичному совпадению текста.

Еще несколько полезных комбинаций: Ctrl+S выполняет поиск в прямом направлении, а Ctrl+G отменяет поиск. Также предусмотрены комбинации для эффективного редактирования текста: Ctrl+W удаляет слова, ALT+F/B перемещается вперед или назад на одно слово, а Ctrl+A/E перемещается в начало или в конец строки.

11.5.2. Ошибки и коды завершения

В текущем варианте программа ничего не выводит. Но если ей были переданы некорректные данные, как узнать, что она успешно завершилась, если вы даже не знаете, что она должна выводить? Правильный ответ — код завершения. Вы можете проверить код завершения последней выполненной команды, но учтите, что команды `echo` и `node` рассматриваются как единое целое из-за каналов.

В системе Windows код завершения проверяется следующим образом:

```
echo %errorlevel%
```

В UNIX для проверки используется следующая команда:

```
echo $?
```

Если команда выполнена успешно, код завершения равен 0. Таким образом, если сценарию были переданы некорректные данные JSON, он завершается с ненулевым кодом:

```
parse-json -f invalid.json
```

При выполнении этой команды приложение завершается с ненулевым кодом и выводит сообщение с описанием причины. Если ошибка была инициирована, но осталась необработанной, Node автоматически завершает работу и выводит сообщение об ошибке.

Потоки ошибок

Вывод на консоль может быть полезен, но еще лучше сохранить его в файле для последующего чтения и для отладки. К счастью, в командном интерпретаторе для этого достаточно перенаправить поток `stdout`:

```
echo 'you can overwrite files!' > out.log
echo 'you can even append to files!' >> out.log
```

Казалось бы, при попытке выполнить эти команды с некорректной разметкой JSON вывод parse-json с сообщением об ошибке стоит сохранить в файле:

```
parse-json -f invalid.json >out.log
```

Однако никаких ошибок в файле нет. И это поведение становится вполне понятным, если понять различия между `stderr` и `stdout`:

- содержимое потока `stdout` используется другими приложениями командной строки;
- содержимое `stderr` используется разработчиками.

Node выводит данные в `stderr` при вызове `console.error` или генерировании ошибки. В этом отношении Node отличается от команды `echo`, которая выводит данные в `stdout` (как и `console.log`). Вероятно, с этой информацией вы захотите направить `stderr` в файл (вместо `stdout`). К счастью, это делается очень просто.

С потоками `stdin`, `stdout` и `stderr` связываются числа от 0 до 2; `stderr` соответствует число 2. Перенаправление осуществляется конструкцией `2>out.log`; она сообщает командному интерпретатору номер перенаправляемого потока и файл для сохранения вывода:

```
parse-json -f invalid.json 2> out.log
```

Перенаправление вывода — то, что делают каналы, но только с процессами вместо файлов. Возьмем следующий фрагмент:

```
node -e "console.log(null)" | parse-json
```

Мы выводим `null` и передаем parse-json. Значение `null` не будет выведено на консоль, потому что оно передается только следующей команде. Предположим, нечто подобное делается с `console.error`:

```
node -e "console.error(null)" | parse-json
```

Вы увидите ошибку, потому что никакой текст не передается parse-json для потребления. Значение `null`, направленное в `stderr`, появится на консоли. Данные должны направляться в `stdout`, а не в `stderr`.

На рис. 11.2 показано, как каналы и номерные выходные потоки могут использоваться для связывания программ и направления вывода в отдельные файлы.

У Node также имеется API для работы с каналами. Он основан на потоках Node, что позволяет использовать его с чем угодно, что реализует классы потоков Node. Использование каналов в Node рассматривается в следующем разделе.

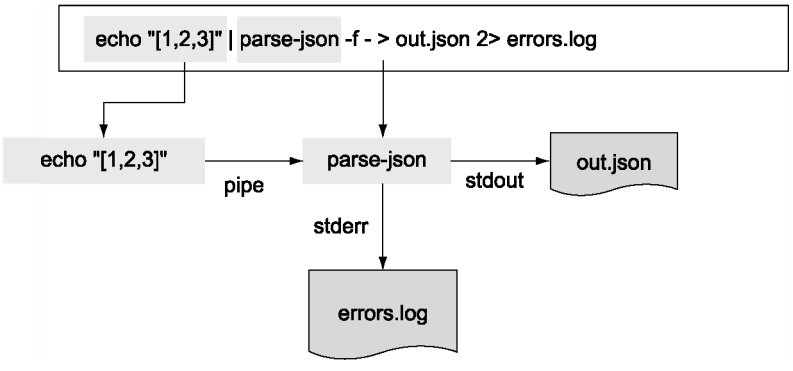


Рис. 11.2. Объединение каналов и потоков вывода

ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: ОЧИСТКА СТРОКИ

Некоторые команды получаются довольно длинными; что делать, если вы хотите удалить длинную команду без выполнения? Полезная комбинация клавиш Ctrl+U удаляет текущую строку. При нажатии клавиш Ctrl+Y строка появится обратно, так что вы можете использовать эти комбинации клавиш при копировании и вставке.

11.5.3. Использование каналов в Node

В этом разделе рассматривается работа с каналами через Node API. Для этого будет написан короткий сценарий, который выводит время выполнения программы без прерывания конвейерной передачи.

Программа может отслеживать состояние канала без прерывания его работы; для этого она ожидает закрытия `stdin` и последующего направления результатов в `stdout`. Так как программы Node завершают работу, когда у них не остается ввода для потребления, при завершении программы может выводиться сообщение. Сохраните следующий пример в файле time.js, чтобы опробовать его:

```
process.stdin.pipe(process.stdout);
const start = Date.now();
process.on('exit', () => {
  const timeTaken = Date.now() - start;
  console.error(`Time (s): ${timeTaken / 1000}`);
});
```

Из-за повторного связывания с `stdout` можно разместить time.js между сцепленными командами, и это не помешает их работе! И parse-json, и time.js могут использоваться с каналами. Например, следующая цепочка команд сообщает, сколько времени тратится на разбор JSON и отправку данных:

```
parse-json -f test.json | node time.js
```

Итак, вы получили представление о том, как выводить данные и как получать ввод от других приложений; теперь можно приступать к построению более сложных процессов. Но сначала стоит поговорить о последовательности выполнения при объединении процессов каналами.

ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: ЗАВЕРШЕНИЕ

Кроме истории команд, многие командные интерпретаторы могут автоматически подставлять команды и файлы при нажатии клавиши Tab. Некоторые даже позволяют просмотреть список возможных подстановок комбинацией клавиш Alt+?.

11.5.4. Каналы и последовательность выполнения команд

При связывании команд каналами каждая команда запускается немедленно. Команды никоим образом не ожидают друг друга. Это означает, что данные не будут дожидаться завершения какой-либо команды и программы могут потреблять только те данные, которые были получены. Так как команды ничего не ожидают, вы не знаете, как завершилась предыдущая команда.

Представьте, что сообщение должно выводиться только в том случае, если разбор JSON прошел успешно. Для этого нам понадобятся новые операторы. Операторы `&&` и `||` в командном интерпретаторе работают примерно так же, как и в JavaScript с числами. Оператор `&&` выполняет следующую команду, если предыдущий код завершения равен нулю, а `||` выполняет следующую команду, если код завершения отличен от нуля.

Посмотрим, как создать небольшой сценарий, который выводит сообщение в `stderr` при завершении процесса. Заметим, что ситуация отличается от `echo`, потому что вывод направляется в `stderr` — он предназначен для разработчиков, а не для других программ. Для этого необходимо прослушивать событие `exit` процесса, а затем записать аргументы в `stderr`:

```
process.stdin.pipe(process.stdout);
process.on('exit', () => {
  const args = process.argv.slice(2);
  console.error(args.join(' '));
});
```

Использовав оператор &&, можно вызвать exit-message.js в том случае, если разбор JSON прошел успешно:

```
parse-json -f test.json && node exit-message.js "parsed JSON successfully"
```

Однако exit-message.js не получит вывода parse-json. Оператор && должен дождаться завершения parse-json.js, чтобы узнать, нужно ли выполнять следующую команду. При использовании && не существует автоматического перенаправления, действующего при использовании каналов.

Перенаправление ввода

Вы уже видели, как выполняется перенаправление вывода, но и ввод может перенаправляться аналогичным образом. Такая необходимость возникает достаточно редко, но она может быть полезной, если исполняемый файл не получает имя файла в аргументе. Если вы хотите, чтобы команда прочитала файл в `stdin`, используйте конструкцию `<`*имя_файла*:

```
parse-json -f - <invalid.json
```

Объединяя обе формы перенаправления, вы можете использовать временный файл для восстановления вывода parse-json:

```
parse-json -f test.json >tmp.out &&
  node exit-message.js "parsed JSON successfully" <tmp.out
```

Научившись обращаться с потоками, кодами завершения и порядком команд, вы сможете писать сценарии с командами Node для ваших собственных пакетов. В следующем разделе показано, как использовать Browserify и UglifyJS с использованием каналов.

> **ПОЛЕЗНАЯ ИНФОРМАЦИЯ О КОМАНДНОЙ СТРОКЕ: ОЧИСТКА ЭКРАНА**
>
> В некоторых случаях на терминал могут быть направлены двоичные данные. Словно в сцене из «Матрицы», экран заполняется непонятными символами. В таких случаях вы можете либо нажать Ctrl+L, чтобы обновить экран, либо ввести команду reset для сброса терминала.

11.6. Интерпретация реальных сценариев

Вы готовы к тому, чтобы начать писать собственные поля `script` в файлах package.json. Для примера посмотрим, как объединить пакеты browserify и uglifyjs из npm. Приложение Browserify (*http://browserify.org/*) получает модули Node и объединяет их для использования в браузере. Приложение UglifyJS (*https://github.com/mishoo/UglifyJS2*) проводит минификацию файла JavaScript, чтобы при передаче его браузеру расходовалось меньше ресурсов канала связи и времени. Ваш сценарий

будет получать файл main.js (из каталога *ch11-command-line/snippets/uglify-example* в архиве кода), выполнять его конкатенацию для использования в браузере, а затем минифицировать полученный сценарий:

```
{
  "devDependencies": {
    "browserify": "13.3.0",
    "uglify-js": "2.7.5"
  },
  "scripts": {
    "build": "browserify -e main.js > bundle.js && uglifyjs bundle.js >
bundle.min.js"
  }
}
```

Сценарий построения запускается командой `npm run build`. В этом примере он создает файл bundle.js. Если создание bundle.js прошло успешно, сценарий создает файл bundle.min.js. При помощи оператора && вы можете гарантировать, что вторая фаза будет выполнена только при успешном выполнении первой фазы.

Использовав приемы, представленные в этой главе, вы сможете создавать и использовать приложения командной строки. Помните, что вы всегда можете воспользоваться командной строкой для объединения сценариев из других языков — если у вас имеется полезная программа командной строки, написанная на языке Python, Ruby или Haskell, вы сможете легко использовать ее в своих программах Node.

11.7. Заключение

Аргументы командной строки могут читаться из массива `process.argv`.

- Такие модули, как yargs, упрощают разбор и проверку аргументов.
- Удобный способ добавления сценариев в проекты Node — определение сценариев npm в файле package.json.
- Для чтения и записи данных в программы командной строки используются стандартные каналы ввода/вывода.
- Стандартный ввод, вывод и ошибки могут перенаправляться разным процессам и файлам.
- Программы выдают коды завершения, по которым можно проверить успешность их завершения.
- Программы командной строки соблюдают общепринятые соглашения, знакомые другим пользователям.

12 Разработка настольных приложений с использованием Electron

Предыдущая глава была посвящена построению программ командной строки в Node. Тем не менее Node начинает обращать на себя внимание в другой области: настольных приложениях. Программисты все чаще привлекают веб-технологии для решения проблем кроссплатформенной разработки. В этой главе вы узнаете, как построить настольное веб-приложение на базе встроенных средств, Node и клиентской веб-технологии. Вы можете разработать и запустить это приложение в Linux, macOS и Windows. Также модули Node будут использоваться в модели, не так далеко ушедшей от разработки веб-приложений «клиент-сервер».

12.1. Знакомство с Electron

Фреймворк Electron, изначально известный под названием Atom Shell, позволяет строить настольные приложения на базе веб-технологий. Программный и пользовательский интерфейс создаются разработчиком на базе HTML, CSS и JavaScript, но некоторые части настольных программ предоставляются фреймворком. К их числу относятся:

- автоматические обновления;
- сообщения о сбоях;
- установка для Microsoft Windows;
- отладка;
- платформенные меню и уведомления.

На базе Electron были созданы некоторые известные приложения. Первое из них — Atom, текстовый редактор GitHub; среди более современных приложений заслуживают упоминания Slack, популярный чат-сервис, и Visual Studio Code компании Microsoft (рис. 12.1).

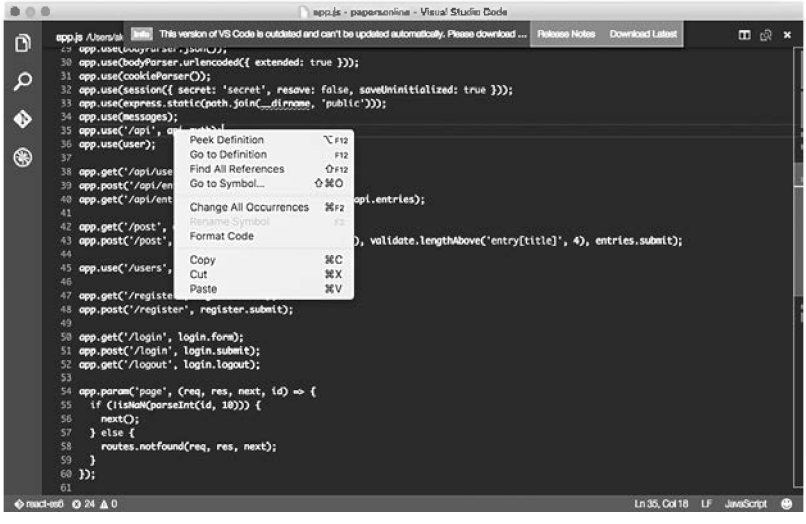


Рис. 12.1. Окно приложения и встроенное контекстное меню в Visual Studio Code

Опробуйте некоторые из этих приложений — они дают представление о том, что можно сделать с помощью Electron. Вооружившись знаниями Node и JavaScript, вы сможете строить конкурентоспособные настольные приложения.

12.1.1. Технологический стек Electron

Прежде чем браться за Electron, вам стоит ознакомиться с тем, какое место занимает Electron в Node, HTML и CSS. Приложение Electron содержит следующие компоненты:

- главный процесс — сценарий Node, который загружает приложение и открывает доступ к модулям Node;
- процесс визуализации — веб-страница, находящаяся под управлением Chromium.

Однако у реального приложения имеются и другие зависимости. Приведенный выше список можно дополнить следующими функциями:

- включение главного процесса;
- подключение к платформенной базе данных (например, SQLite);
- взаимодействие с веб-API;
- чтение и запись локальных файлов (например, конфигурационных файлов);
- доступ к платформенной функциональности (например, контекстные меню);
- включение процесса визуализации;

- реализация современного полнофункционального веб-приложения с использованием выбранной технологии клиентской стороны (например, React или Angular);
- использование встроенной функциональности (например, контекстные меню и уведомления);
- встроенные сценарии;
- генерирование клиентского кода JavaScript в выбранной системе построения (Grunt, Gulp, сценарии npm);
- подготовка к распространению.

На рис. 12.2 приведена схема трех основных частей типичного приложения Electron. Как видно из схемы, Node используется для запуска главного процесса и взаимодействия с операционной системой для использования сервисных функций: открытия файлов, чтения и записи в базу данных, взаимодействия с веб-службами. И хотя особое внимание уделяется пользовательскому интерфейсу процесса визуализации, Node остается важнейшей частью архитектуры приложения.

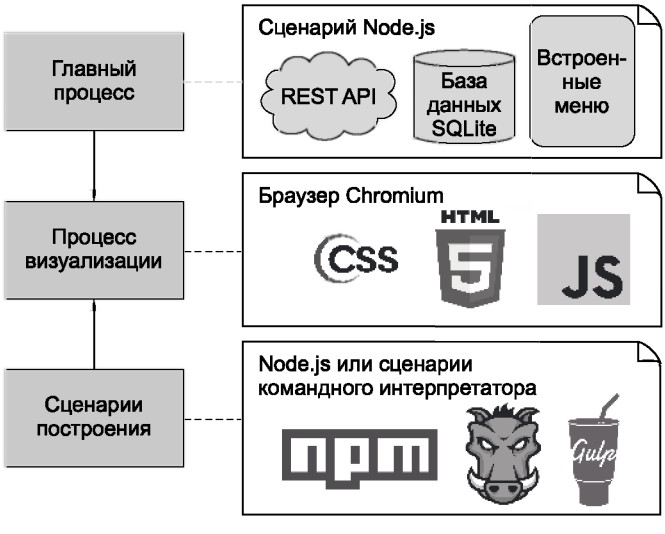


Рис. 12.2. Основные части типичного приложения Electron

12.1.2. Проектирование интерфейса

Познакомившись с основными компонентами приложений Electron, посмотрим, как строятся современные интерфейсы. Приложения Electron строятся на базе HTML, CSS и JavaScript, так что вы не сможете задействовать платформенные виджеты. Представьте, что вы хотите создать интерфейс в стиле Mac. Панель инструментов

macOS можно имитировать градиентами CSS. Также средствами CSS можно использовать платформенные шрифты, предоставляемые macOS и Windows; вы даже можете включить сглаживание, чтобы ваше приложение больше походило на платформенные приложения. Можно снять текстовое выделение для некоторых UI-компонентов и реализовать поддержку перетаскивания мышью. В настоящее время многие приложения Electron используют CSS с такими же цветами, стилями, значками и градиентами, как в macOS и Windows.

Некоторые приложения прилагают особые усилия к имитации платформенной функциональности; примером такого рода служит почтовое приложение N1 (*https://github.com/nylas/N1*). Другие приложения — такие, как Slack (*https://slack.com/*) — обладают достаточно четко выраженной индивидуальностью и работают без особых модификаций на всех платформах.

Когда вы начнете строить собственные приложения Electron, вам придется выбирать, какой подход лучше соответствует вашему проекту. Если вы хотите создать приложение, которое выглядит так, словно оно использует платформенные виджеты, вам придется создать стили, подходящие для каждой платформы. Проектирование всех вариантов пользовательского интерфейса потребует дополнительного времени. Возможно, этот вариант будет более привлекательным для пользователей вашего приложения, но он также может повысить затраты на развертывание новой функциональности.

В следующем разделе заготовка приложения Electron используется для создания нового приложения. Это стандартный метод построения новых проектов в Electron.

12.2. Создание приложения Electron

Начать работу с Electron проще всего с проекта electron-quick-start, доступного на GitHub (*https://github.com/atom/electron-quick-start*). В этом маленьком репозитории содержатся зависимости, необходимые для запуска базового приложения Electron.

Чтобы использовать этот проект, скопируйте проект из репозитория и установите его зависимости в npm:

```
git clone https://github.com/atom/electron-quick-start
cd electron-quick-start
npm install
```

После завершения всех загрузок вы можете запустить главный процесс командой `npm start`. Вы можете спокойно брать этот проект за основу для других приложений Electron; вам совершенно не обязательно создавать свои проекты с нуля.

При запуске приложения появляется окно с веб-страницей и инструментами Chromium Developer Tools. Если вы — веб-разработчик, использующий Chrome, результат не слишком впечатляет: приложение напоминает веб-страницу без CSS-оформления в Chrome. Но для того, чтобы все это заработало, «за кулисами» происходит много всего. На рис. 12.3 показано, как результат выглядит в macOS.

Рис. 12.3. Проект electron-quick-start в macOS

Это автономный пакет приложения macOS: он включает в себя версию Node, отличную от той, которая работает в моей системе, использует собственные команды меню и окно `About`.

К этому моменту вы можете начать строить собственное веб-приложение в **index.html** с использованием HTML, JavaScript и CSS. Но вам как Node-программисту, вероятно, не терпится использовать Node на практике; давайте сначала посмотрим, как это делается.

Electron включает в себя модуль remote, использующий механизм межпроцессных коммуникаций (IPC, InterProcess Communication) между процессом визуализации и главным процессом Node. Модуль remote даже может предоставить доступ к модулям Node. Чтобы опробовать его на практике, добавьте в свой проект Electron файл с именем **readfile.js** и включите в него код из листинга 12.1.

Листинг 12.1. Простой модуль Node

```
const fs = require('fs');
module.exports = (cb) => {
  fs.readFile('./main.js', { encoding: 'utf8' }, cb);
};
```

Теперь откройте файл **index.html** и внесите изменения: добавьте элемент с идентификатором `source` и сценарий, который загружает **readfile.js** (листинг 12.2).

Листинг 12.2. Загрузка модулей Node из процесса визуализации

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    <pre id="source"></pre>
    <script>
var readfile = require('remote').require('./readfile');
readfile(function(err, text) {
  console.log('readfile:', err, text);
  document.getElementById('source').innerHTML = text;
});
    </script>
  </body>
</html>
```

В листинге 12.2 модуль remote используется для загрузки файла readfile.js и его выполнения в главном процессе. Взаимодействие между двумя процессами проходит без проблем, поэтому внешне все мало отличается от использования стандартных модулей Node. Единственное серьезное различие — использование `require('remote').require(file)`.

12.3. Построение полнофункционального настольного приложения

Итак, вы узнали, как создать базовое приложение Electron и как использовать модули Node. Пора сделать следующий шаг и построить полноценное приложение. Приложение, которое мы создадим, представляет собой инструмент разработчика для выдачи и просмотра запросов HTTP. Его можно рассматривать как графический интерфейс для модуля request (*www.npmjs.com/package/request*).

Хотя приложения Electron можно строить непосредственно из HTML, JavaScript, CSS и Node, в этой главе мы воспользуемся современными инструментами разработчика клиентской части; это упростит сопровождение и расширение приложения. При этом будут использоваться следующие средства:

- electron-quick-start как основа для проекта;
- модуль request для запросов HTTP;
- React для кода пользовательского интерфейса;
- Babel для преобразования современного синтаксиса ES6 в синтаксис ES5, совместимый с браузером;
- webpack для построения клиентского приложения.

На рис. 12.4 показано, как будет выглядеть готовое приложение.

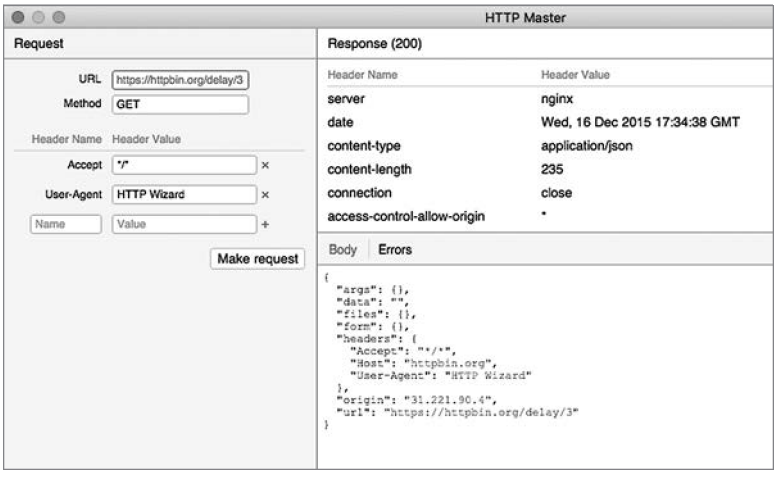


Рис. 12.4. Electron-приложение HTTP Master

Затем вы узнаете, как настроить проект на базе React с webpack и Babel.

12.3.1. Исходная настройка React и Babel

Одна из основных проблем при построении нового приложения со сложной клиентской частью — настройка таких библиотек, как React и Babel, с системой построения, без особых проблем с сопровождением. Есть много вариантов, включая Grunt, Gulp и Webpack. Ситуация дополнительно усложняется тем, что библиотеки изменяются со временем, так что книги и учебные описания быстро устаревают.

Чтобы избежать проблем со стремительным развитием средств разработки клиентской части, мы указываем точные версии всех зависимостей, так что вы сможете воспроизвести описания и получить сходные результаты. Если вы все же запутаетесь, воспользуйтесь такими инструментами, как Yeoman (*http://yeoman.io/*), для генерирования заготовки приложения. Затем измените сгенерированную заготовку и приведите ее в соответствие с приложением, описанным в этой главе.

12.3.2. Установка зависимостей

Создайте новый проект electron-quick-start. Стоит напомнить, что проект следует клонировать с GitHub:

```
git clone https://github.com/atom/electron-quick-start
cd electron-quick-start
npm install
```

Затем установите react, react-dom и babel-core:

```
npm install --save-dev react@0.14.3 react-dom@0.14.3 babel-core@6.3.17
```

Далее необходимо установить плагины Babel. Главный из них — babel-preset-es2015 — избыточен для проекта, ограниченного Chromium, но его включение упростит эксперименты с функциями ES2015, которые Chromium еще не поддерживает. Используйте для установки следующие команды:

```
npm install --save-dev babel-preset-es2015@6.3.13
npm install --save-dev babel-plugin-transform-class-properties@6.3.13
```

Плагин добавляет поддержку JSX в Babel:

```
npm install --save-dev babel-plugin-transform-react-jsx@6.3.13
```

Затем установите webpack:

```
npm install --save-dev webpack@1.12.9
```

Для работы с Babel вам также понадобится пакет babel-loader для webpack:

```
npm install --save-dev babel-loader@6.2.0
```

Когда зависимости будут готовы, добавьте в проект файл **.babelrc**. Он приказывает Babel использовать плагины ES2015 и React:

```
{
  "plugins": [
    "transform-react-jsx"
  ],
 "presets": ["es2015"]
}
```

Наконец, откройте файл **package.json** и обновите свойство `scripts`, включив в него вызов webpack:

```
  "scripts": {
    "start": "electron main.js",
    "build": "node_modules/.bin/webpack --progress --colors"
  },
```

Это позволяет строить приложение командой `npm run build`. Также доступны плагины webpack для горячей загрузки React, но здесь эта возможность не рассматривается. Если вы хотите, чтобы код на стороне клиента автоматически строился при изменении файлов, воспользуйтесь такими инструментами, как fswatch и nodemon.

12.3.3. Настройка webpack

Для использования webpack вам понадобится файл **webpack.config.js**. Добавьте его в корневой каталог проекта.

Файл содержит код JavaScript с использованием модуля CommonJS в стиле Node:

```
const webpack = require('webpack');
module.exports = {
  setting: 'value'
};
```

В нашем проекте понадобятся настройки для поиска файлов React (.jsx), загрузки точки входа (/app/index.jsx) и размещения вывода в месте, в котором пользовательский интерфейс Electron сможет его найти (js/app.js). Файлы React также должны обрабатываться Babel. В результате объединения всех этих требований будет получен файл, приведенный в листинге 12.3.

Листинг 12.3. webpack.config.js

```
const webpack = require('webpack');
module.exports = {
  module: {
    loaders: [
     { test: /\.jsx?$/, loaders: ['babel-loader'] }
    ]
  },
  entry: [
    './app/index.jsx'
  ],
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  output: {
    path: __dirname + '/js',
    filename: 'app.js'
  }
};
```

В этом листинге мы приказываем webpack преобразовывать файлы .jsx (React) программой Babel при помощи свойства `module.loaders`. Программа Babel уже была настроена для обработки файлов React записью `transform-react-jsx` в .babelrc. Затем свойство `entry` используется для определения главной точки входа для кода React. Этот способ работает, потому что компоненты React базируются на элементах HTML. Поскольку элементы HTML должны иметь один родительский узел, одна точка входа может охватывать все приложение.

Свойство `resolve.extensions` сообщает webpack, что файлы .jsx следует рассматривать как модули. Если использовать команду вида `import {Class} from 'class'`, она проверит файлы class.js и class.jsx.

Наконец, свойство `output` сообщает webpack, куда следует записать выходной файл. В данном примере используется значение js/, но вы можете выбрать любой путь, доступный для пользовательского интерфейса Electron.

Пора наполнить приложение React реальным содержанием. Начнем с главной точки входа и того, как она будет обращаться к UI-элементам запроса и ответа.

12.4. Приложение React

На рис. 12.4 показан внешний вид приложения. Оно содержит две группы UI-компонентов, которые можно разделить на 7 элементов:

- запрос (Request);
  - URL: String;
  - метод: String;
  - заголовки: объект с парами строк;
- ответ (Response);
  - код статуса HTTP;
  - заголовки: объект с парами строк;
  - тело: String;
  - ошибки: String.

Но в React нельзя просто вывести два графических объекта по соседству: они должны содержаться в одном родителе. Потребуется объект приложения верхнего уровня, содержащий UI-элементы запроса и ответа.

С классами запроса (`Request`) и ответа (`Response`), которые будут реализованы позднее, сам класс `App` должен выглядеть так, как показано в листинге 12.4.

Листинг 12.4. Класс App

```
import React from 'react';
import ReactDOM from 'react-dom';
import Request from './request';
import Response from './response';
class App extends React.Component {
  render() {
    return (
      <div className="container">
        <Request />
        <Response />
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('app'));
```

Сохраните код в файле app/index.jsx. Он сначала загружает классы `Request` и `Response`, а затем выводит их данные в элементе `div`. В последней строке ReactDOM

используется для визуализации узлов модели DOM класса `App`. Тег `<App />` используется для обращения к классу `App`.

Чтобы это решение работало, необходимо определить компоненты `Request` и `Response`.

12.4.1. Определение компонента Request

Класс `Request` получает на входе URL-адрес и метод HTTP, после чего генерирует запрос, который отправляется с использованием модуля Node request. Построение интерфейса осуществляется с использованием JSX, но, в отличие от предыдущего примера, элемент не прорисовывается напрямую ReactDOM; это происходит при включении в главный класс приложения в файле app/index.jsx.

В листинге 12.5 (файл app/request.js) содержится полный код класса. Мы удалили функцию редактирования заголовков для сокращения объема примера; если вам захочется увидеть пример с расширенной функциональностью, включая редактирование заголовков, обращайтесь к репозиторию HTTP Wizard на GitHub (*https://github.com/alexyoung/http-wizard*).

Листинг 12.5. Класс Request

```
import React from 'react';
import Events from './events';

const request = remote.require('request');

class Request extends React.Component {
  constructor(props) {
    super(props);
    this.state = { url: null, method: 'GET' };
  }

  handleChange = (e) => {
    const state = {};
    state[e.target.name] = e.target.value;
    this.setState(state);
  }

  makeRequest = () => {
    request(this.state, (err, res, body) => {
      const statusCode = res ? res.statusCode : 'No response';
      const result = {
        response: `(${statusCode})`,
        raw: body ? body : '',
        headers: res ? res.headers : [],
        error: err ? JSON.stringify(err, null, 2) : ''
      };

      Events.emit('result', result);
      new Notification(`HTTP response finished: ${statusCode}`)
```

```
    });
  }

  render() {
    return (
      <div className="request">
        <h1>Request</h1>
        <div className="request-options">
          <div className="form-row">
            <label>URL</label>
            <input
              name="url"
              type="url"
              value={this.state.url}
              onChange={this.handleChange} />
          </div>
          <div className="form-row">
            <label>Method</label>
            <input
              name="method"
              type="text"
              value={this.state.method}
              placeholder="GET, POST, PATCH, PUT, DELETE"
              onChange={this.handleChange} />
          </div>
          <div className="form-row">
            <a className="btn" onClick={this.makeRequest}>Make request</a>
          </div>
        </div>
      </div>
    );
  }
}

export default Request;
```

Основную часть листинга занимает HTML-разметка метода `render`. Сосредоточимся на остальном, прежде чем заниматься построением пользовательского интерфейса. Сначала потомок класса Node `EventEmitter` в файле app/events.jsx используется для взаимодействия между этим компонентом и компонентом `response`. Следующий фрагмент взят из файла app/events.jsx:

```
import { EventEmitter } from 'events';
const Events = new EventEmitter();
export default Events;
```

Обратите внимание на то, что класс `Request` является потомком `React.Component`. Он определяет конструктор, который устанавливает состояние по умолчанию; свойство `state` занимает особое место в React и может задаваться таким образом только в конструкторе. Во всех остальных местах необходимо использовать `this.setState`.

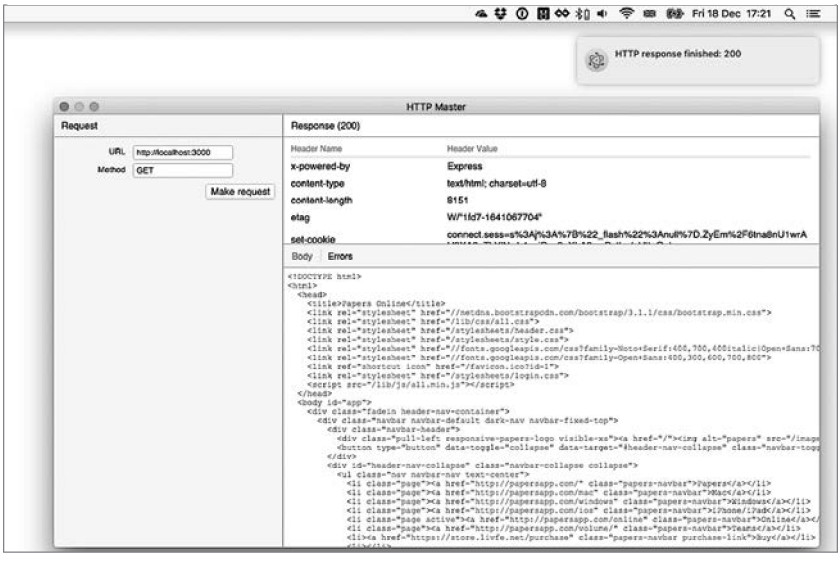


Рис. 12.5. Уведомление

Метод `handleChange` задает состояние на основании атрибута `name` элемента HTML. Чтобы понять, как работает этот механизм, обратитесь к элементу `<input>` для URL в методе `render`:

```
<input
  name="url"
  type="url"
  value={this.state.url}
  onChange={this.handleChange} />
```

Заданное значение `name` используется для задания URL при редактировании. Назначение `state` также инициирует выполнение `render`, а React обновляет атрибут `value` измененным состоянием. А теперь посмотрим, как модуль request будет использоваться этим классом.

Класс содержит клиентский код, который выполняется в веб-представлении, поэтому вам понадобится доступ к модулю request для создания запросов HTTP. Electron предоставляет возможность загрузки удаленных модулей без лишнего шаблонного кода. В начале класса глобальный объект `remote` включает модуль Node request:

```
const request = remote.require('request');
```

Далее в методе `makeRequest` запрос HTTP выдается простым вызовом `request()`. Аргументы `request` были заданы в состоянии класса, поэтому от вас потребуется лишь обработать обратный вызов, который отрабатывает при завершении запроса. Объем императивного кода очень мал: состояние класса задается на основании результата

запроса, после чего результат передается для использования компонентом `Response`. Также отображается визуальное уведомление; если запрос слишком медленный, пользователь оповещается об этом визуально с использованием всплывающего окна операционной системы:

```
new Notification(`HTTP response finished: ${statusCode}`)
```

На рис. 12.5 показано типичное уведомление.

Посмотрим, как компонент `Response` отображает ответ HTTP.

12.4.2. Определение компонента Response

Компонент `Response` прослушивает события `result` и задает свое состояние в соответствии с результатами последнего запроса. Для вывода результатов используется таблица для заголовков и элементы `div` для тела запроса и ошибок.

В листинге 12.6 приведен полный код компонента `Response` из файла app/response.jsx.

Листинг 12.6. Компонент Response

```
import React from 'react';
import Events from './events';
import Headers from './headers';

class Response extends React.Component {
  constructor(props) {
    super(props);
    this.state = { result: {}, tab: 'body' };
  }

  componentWillUnmount() {
    Events.removeListener('result', this.handleResult.bind(this));
  }

  componentDidMount() {
    Events.addListener('result', this.handleResult.bind(this));
  }

  handleResult(result) {
    this.setState({ result: result });
  }

  handleSelectTab = (e) => {
    const tab = e.target.dataset.tab;
    this.setState({ tab: tab });
  }

  render() {
    const result = this.state.result;
    const tabClasses = {
```

```
      body: this.state.tab === 'body' ? 'active' : null,
      errors: this.state.tab === 'errors' ? 'active' : null,
    };
    const rawStyle = this.state.tab === 'body'
      ? null
      : { display: 'none' }
    const errorsStyle = this.state.tab === 'errors'
      ? null
      : { display: 'none' };

    return (
      <div className="response">
        <h1>Response <span id="response">{result.response}</span></h1>
        <div className="content-container">
          <div className="content">
            <div id="headers">
              <table className="headers">
                <thead>
                  <tr>
                    <th className="name">Header Name</th>
                    <th className="value">Header Value</th>
                  </tr>
                </thead>
                <Headers headers={result.headers} />
              </table>
            </div>
            <div className="results">
              <ul className="nav">
                <li className={tabClasses.body}>
                  <a data-tab='body' onClick={this.handleSelectTab}>Body</a>
                </li>
                <li className={tabClasses.errors}>
                  <a data-tab='errors' href="#"
onClick={this.handleSelectTab}>Errors</a>
                </li>
              </ul>
              <div
                className="raw"
                id="raw"
                style={rawStyle}>{result.raw}</div>
              <div
                className="raw"
                id="error"
                style={errorsStyle}>{result.error}</div>
            </div>
          </div>
        </div>
      </div>
    );
  }
}

export default Response;
```

Компонент `Response` не содержит кода, непосредственно относящегося к обработке ответов HTTP; он выводит свое состояние в разных элементах HTML. Для переключения вкладок обработчик `onclick` связывается с методом `handleSelectTab`, который переключается между телом запроса и ошибками при помощи атрибута `(data-tab)`.

Компонент `Response` использует другой компонент, `Headers`, для отображения заголовков ответа HTTP. Разбиение компонента на еще меньшие компоненты — стандартная практика в React. Значения заголовков передаются субкомпонентам через атрибуты; в React они называются *свойствами* (props):

```
<Headers headers={result.headers} />
```

В листинге 12.7 представлен код компонента `Headers` из файла app/headers.jsx.

Листинг 12.7. Компонент Headers

```
import React from 'react';

class Headers extends React.Component {
  render() {
    const headers = this.props.headers || {};
    const headerRows = Object.keys(headers).map((key, i) => {
      return (
        <tr key={i}>
          <td className="name">{key}</td>
          <td className="value">{headers[key]}</td>
        </tr>
      );
    });

    return (
      <tbody className="header-body">
        {headerRows}
      </tbody>
    );
  }
}

export default Headers;
```

Обратите внимание на обращение к свойствам в начале метода `render()` в выражении `this.props.headers`.

12.4.3. Взаимодействие между компонентами React

Классы `Request` и `Response` достаточно хорошо изолированы друг от друга; они сосредоточены на решении своих конкретных задач без того, чтобы напрямую вызывать друг друга. В React существуют другие, более сложные средства управления

состоянием, но они выходят за рамки темы этой главы. В нашем примере сложные механизмы передачи данных не нужны, потому что в нем используется всего два основных компонента, поэтому для передачи данных используется класс Node `EventEmitter`.

Чтобы использовать `EventEmitter` таким образом, создайте экземпляр в отдельном файле (app/events.jsx), а затем экспортируйте экземпляр:

```
import { EventEmitter } from 'events';
const Events = new EventEmitter();
export default Events;
```

Теперь компоненты могут инициировать события и подключать слушателей для передачи данных. Компонент `Require` использует эту возможность в методе `makeRequest` с результатом запроса HTTP:

```
Events.emit('result', result);
```

Затем в классе `Response` мы сохраняем результаты, назначая слушателя на ранней стадии жизненного цикла компонента:

```
componentWillUnmount() {
  Events.removeListener('result', this.handleResult.bind(this));
}
```

С ростом приложения сопровождать этот паттерн становится все сложнее; в частности, возникает проблема с отслеживанием имен событий. Так как данные хранятся в строковом виде, их легко забыть или записать неправильно. В усовершенствованной версии этого паттерна используется список констант, представляющих собой имена событий. Если вы снова расширите этот паттерн, чтобы разделить обязанности диспетчеризации событий и сохранения данных, у вас получится нечто похожее на контейнер состояния Redux от Facebook (*http://redux.js.org/*); именно поэтому многие программисты используют его для проектирования и построения более крупных приложений.

12.5. Построение и распространение

Итак, вы построили функциональное настольное приложение, и теперь его можно упаковать для macOS, Linux и Windows. Распространение приложений Electron состоит из трех этапов:

1. Измените стандартное оформление приложения Electron, назначив ему имя и значок.
2. Упакуйте приложение в файл.
3. Создайте двоичный файл для каждой платформы.

Проект electron-quick-start уже почти готов к распространению. От вас потребуется лишь скопировать свой код в папку Electron Contents/Resources/app в macOS или electron/resources/app в Windows и Linux.

Тем не менее ручное копирование — не лучший способ построения распространяемого двоичного файла. Другой, более надежный способ основан на использовании пакета electron-packager (*www.npmjs.com/package/electron-packager*) Макса Огдена (Max Ogden). Пакет предоставляет интерфейс командной строки для построения исполняемых файлов для Windows, Linux и macOS.

12.5.1. Построение приложений с использованием Electron Packager

Чтобы установить electron-packager, установите его глобально. После этого вы сможете построить любой проект, для которого потребуется создать двоичные файлы для конкретных платформ:

```
npm install electron-packager -g
```

После того как установка будет завершена, вы сможете запустить пакет из каталога приложения. При вызове следует указать путь к приложению, имя приложения, платформу, архитектуру (32- или 64-разрядную) и версию Electron:

```
electron-packager . HttpWizard --version=1.4.5
```

Команда загружает Electron версии 1.4.5 и генерирует двоичные файлы для всех поддерживаемых платформ и архитектур. Это потребует времени (размер Electron — около 40 Мбайт), но когда все будет сделано, вы получите двоичные файлы, которые будут работать во всех основных операционных системах.

СКРЫВАЕМ ИНСТРУМЕНТЫ РАЗРАБОТЧИКА

Прежде чем распространять построенную версию, удалите или измените строку main.js, которая открывает инструменты разработчика Chromium:

```
mainWindow.webContents.openDevTools();
```

Также можно заключить эту команду в условную конструкцию с проверкой флага, чтобы инструменты разработчика открывались только при установленном флаге:

```
if (process.env.NODE_ENV === 'debug') {
  mainWindow.webContents.openDevTools();
}
```

12.5.2. Упаковка

Чтобы дополнительно повысить быстродействие приложения, вы можете упаковать файлы JavaScript (клиентские и Node) с использованием архивов Atom Shell Archives (*https://github.com/atom/asar*). Эти архивы, называемые *файлами asar*, похожи на *tar*-архивы UNIX. Они скрывают код JavaScript, но не маскируют его настолько, чтобы его было невозможно декодировать. Тем не менее они решают проблему с искажением длинных имен файлов в Windows, которая может возникнуть в зависимостях с очень большим уровнем вложенности.

В Electron Chromium может читать файлы `asar`, как и Node, так что вам не придется делать ничего особенного для их поддержки. Кроме того, electron-packager может создавать пакеты asar за вас с ключом командной строки `--asar`.

На рис. 12.6 показано, как выглядит приложение, упакованное без использования asar.

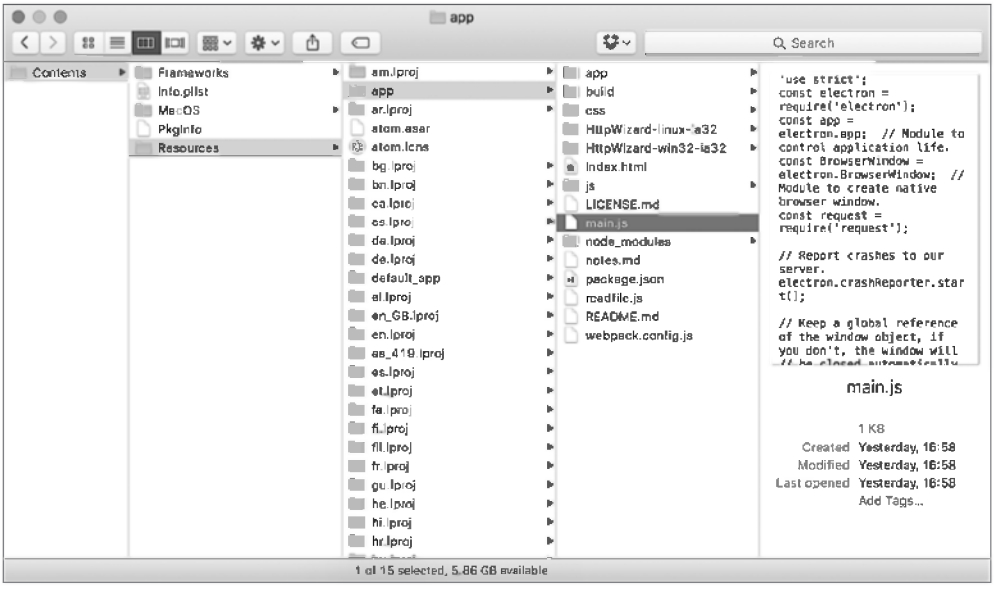


Рис. 12.6. Содержимое типичного пакета приложения Electron

Обратите внимание: файлы JavaScript можно открывать для просмотра исходного кода. В приложении Electron двоичный формат имеют только файлы ресурсов (например, графика) и двоичные модули Node.

Чтобы построить приложение с файлами asar, используйте electron-packager с флагом `--asar`:

```
electron-packager . HttpWizard --version=0.36.0 --asar=true
```

Это самый простой способ, потому что electron-packager выполнит все необходимые команды. Чтобы проделать то же вручную, необходимо установить asar, а затем запустить программу командной строки для создания пакета:

```
npm install -g asar
asar pack path-to-your-app/ app.asar
```

Получив архив asar, загрузите двоичный файл Electron (*https://github.com/atom/electron/releases*) для той платформы, которую вы хотите поддерживать, и добавьте архив в каталог Resources (см. рис. 12.6). Запуск исполняемого файла или пакета приложения должен привести к запуску вашего приложения.

Также приложения Electron могут изменяться посредством редактирования двоичных файлов. Так можно изменить имя и значки приложения. Если запустить двоичный файл Electron без изменений, на экране появится окно для запуска приложений Electron, созданных на базе репозитория electron-quick-start.

12.6. Заключение

- Фреймворк Electron позволяет создавать настольные приложения с использованием Node, JavaScript, HTML и CSS.
- Вы можете генерировать платформенные меню и уведомления без использования C++, C# или Objective-C.
- Если у вас имеются полезные модули Node, вы можете использовать их из кода JavaScript на стороне клиента в пользовательском интерфейсе приложения Electron.
- Electron использует полнофункциональный браузер, так что вы можете строить пользовательские интерфейсы с применением новейших технологий JavaScript (таких, как React или Angular).

Приложения

Приложение А. Установка Node

Приложение Б. Автоматизированное извлечение веб-данных

Приложение В. Официально поддерживаемые промежуточные компоненты

A Установка Node

В этом приложении приведена более подробная информация об установке Node.js. Если вы начали работать с Node недавно, мы рекомендуем использовать для установки заранее построенный пакет. Ниже процесс установки будет рассмотрен для всех остальных операционных систем.

В зависимости от требований Node также может устанавливаться другими способами. Если вы принадлежите к числу опытных пользователей Node или если у вас действуют особые требования в области DevOps, переходите к обзору других способов установки Node.

A.1. Установка Node с использованием программы установки

У Node есть две программы установки и несколько заранее построенных двоичных пакетов. Если вы работаете в macOS или Windows, можно использовать как двоичные пакеты, так и программы установки. Двоичные пакеты содержат исполняемые файлы, но у программ установки есть специальные мастеры (wizards), которые помогут установить Node в каталоге системы, который будет легко находиться при выполнении в терминале таких команд, как node и npm.

Если вы еще не обладаете достаточным опытом работы с Node, используйте программу установки. Все версии можно найти на сайте Node в разделе Downloads (*https://nodejs.org/en/download/*).

A.1.1. Программа установки для macOS

Для macOS загрузите 64-разрядный файл .pkg с сайта Node (*https://nodejs.org/en/download/*). Используйте либо LTS, либо текущую версию. После загрузки вы получите файл пакета, изображенный на рис. A.1.

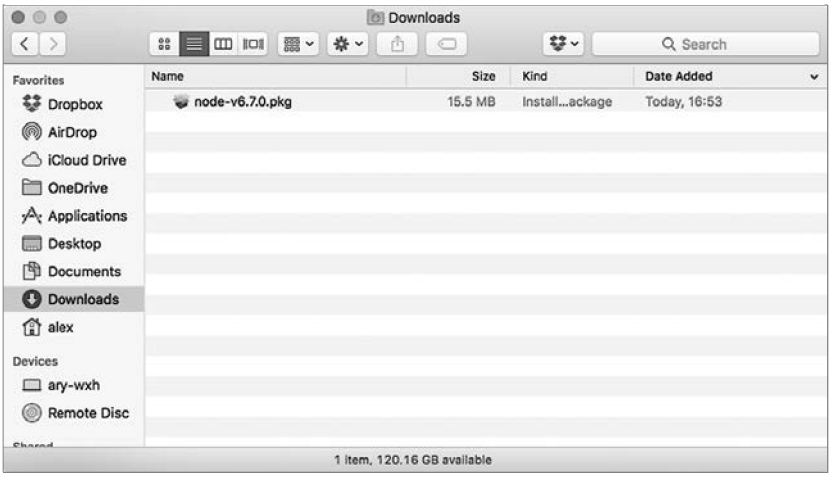


Рис. А.1. Установочный файл .pkg

После того как программа установки будет загружена, сделайте на ней двойной щелчок; откроется мастер установки (рис. А.2).

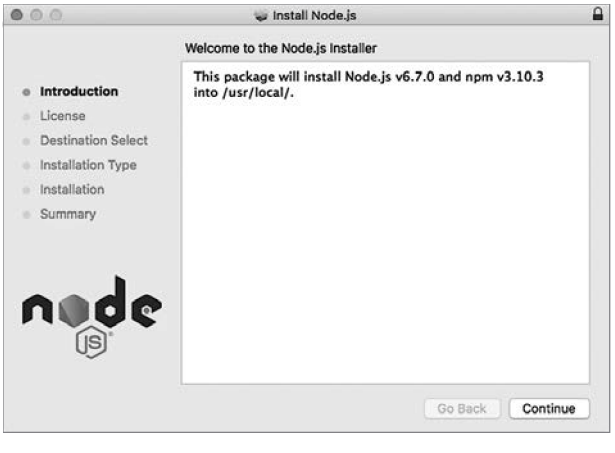


Рис. А.2. Мастер установки

Щелкните на кнопке Continue и выполните инструкции; со значениями по умолчанию Node устанавливается правильно. После того как процесс установки будет завершен, вы сможете открыть терминал и ввести команду `node` для запуска терминала Node REPL (рис. А.3).

В следующем разделе рассматривается процесс установки для пользователей Windows.

Рис. А.3. Node REPL

А.1.2. Программа установки для Windows

На странице Node Downloads (*https://nodejs.org/en/download/*) щелкните на значке Windows Installer или на ссылке Windows Installer .msi. Доступны 32- и 64-разрядные версии; вероятно, вы выберете 64-разрядную. После того как файл будет загружен, сделайте на нем двойной щелчок, чтобы запустить мастер установки (рис. А.4).

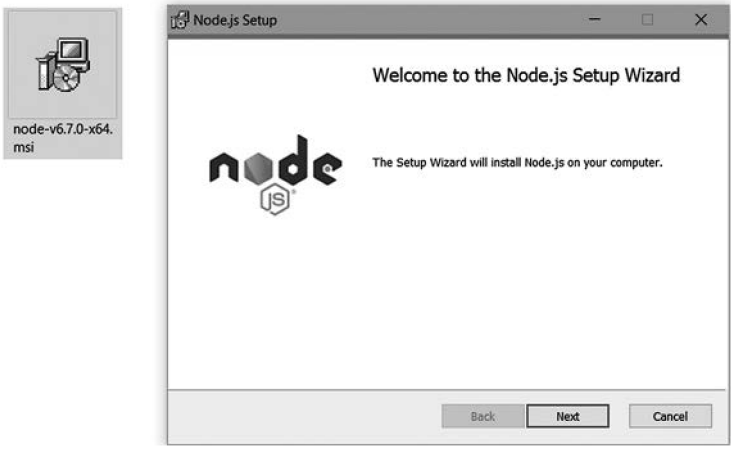


Рис. А.4. Программа установки Windows .msi

Подтвердите значения по умолчанию, затем откройте cmd.exe и проверьте, как работает Node REPL.

На рис. А.5 изображено окно Node REPL в Windows.

Рис. А.5. Node REPL в Windows

Если вы обычно не устанавливаете программное обеспечение этим способом или не хотите устанавливать Node на общесистемном уровне, в следующем разделе рассматриваются другие варианты установки Node.

А.2. Другие способы установки Node

Node также можно построить из исходных кодов, воспользоваться менеджером пакетов операционной системы или менеджером версий Node. При построении из исходного кода вам понадобится рабочая система и работоспособный экземпляр Python.

А.2.1. Установка Node из исходного кода

Исходный код Node можно загрузить на странице загрузки сайта *nodejs.org*, но он также доступен в репозитории GitHub (*https://github.com/nodejs/node*). Полное руководство по построению также доступно на GitHub (node/Building.md (*https://github.com/nodejs/node/blob/master/BUILDING.md*). Для построения Node вам понадобится:

- Linux — Python 2.6 или 2.7, gcc и g++ 4.8 и выше или clang и clang++ 3.4 или выше. Чтобы получить все эти программы, проще всего установить пакет build-essentials в дистрибутивах семейства Debian или его эквивалент в других дистрибутивах;
- macOS — Xcode и инструменты командной строки, которые могут устанавливаться с Xcode;
- Windows — Python 2.6 или 2.7, Visual C++ Build Tools, Visual Studio 2015 Update 3.

Когда инструментарий построения будет готов, выполните `./configure` и `make` в операционных системах семейства UNIX. В Windows выполните команду `.\vcbuild nosign`.

А.2.2. Установка Node из менеджера пакетов

Если вы работаете в Linux или macOS, возможно, вы предпочтете установить Node при помощи менеджера пакетов. Это решение упрощает обновление Node. Например, если вы используете веб-сервер на базе Linux, вы можете установить Node для автоматического получения обновлений безопасности.

На сайте Node размещен обширный список инструкций по установке для операционных систем, которые предоставляют Node в формате пакета (*https://nodejs.org/en/download/package-manager/*).

Например, в Debian и системах на базе Ubuntu вы можете получить Node из репозитория Node-Source. Он имеет собственный репозиторий на GitHub с дополнительной информацией (*https://github.com/nodesource/distributions*).

В macOS для установки Node можно воспользоваться менеджером Homebrew (*http://brew.sh/*). Если менеджер Homebrew уже установлен в вашей системе, достаточно выполнить команду `brew install node`.

Пакет Node также доступен в Docker Hub. Если добавить `FROM node:argon` в Dockerfile, в образе будет установлена LTS-версия Node.

Б Автоматизированное извлечение веб-данных

Извлечение веб-данных (web scraping), требующее сочетания навыков серверного и клиентского программирования, ориентировано на применение программных средств для анализа веб-страниц и преобразования их в структурированные данные. Представьте, что вам поручено создать новую версию сайта, который в настоящее время представляет собой набор старомодных статических HTML-страниц. Вы хотите загрузить страницы, проанализировать их и извлечь названия, описания, авторов и цены для всех книг. Делать это вручную не хочется, поэтому вы пишете для выполнения этой работы программу Node. Это и есть извлечение веб-данных.

Node отлично подходит для этой цели благодаря идеальному балансу между браузерной технологией и мощью сценарных языков общего назначения. В этой главе вы узнаете, как использовать библиотеки разбора HTML для извлечения полезных данных по селекторам CSS и даже выполнения динамических веб-страниц в процессе Node.

Б.1. Извлечение веб-данных

Извлечением веб-данных называется процесс извлечения полезной информации с веб-сайтов. Обычно для этого программа загружает необходимые страницы, разбирает их, а затем запрашивает низкоуровневую разметку HTML с использованием селекторов CSS или XPath. Результаты запросов экспортируются в файлы CSV или сохраняются в базе данных. На рис. Б.1 показано, как работает процесс извлечения веб-данных от начала до конца.

Извлечение веб-данных может противоречить условиям использования некоторых сайтов из-за ценного контента или ограничений по ресурсам. Если тысячи программ извлечения данных обращаются к одному сайту, который работает на старом и медленном сервере, сайт может выйти из строя. Прежде чем извлекать контент, необходимо убедиться в том, что у вас есть разрешение на чтение и дублирование

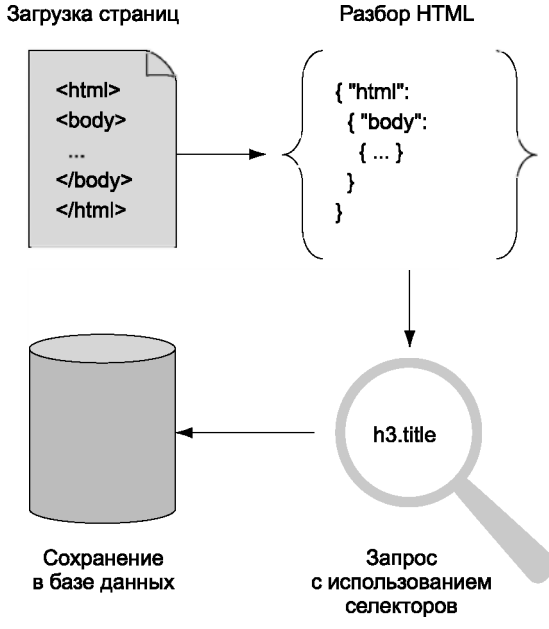


Рис. Б.1. Последовательность действий по извлечению и сохранению контента

контента. Формально за этой информацией можно обратиться к файлу robots.txt сайта (*www.robotstxt.org*), но сначала следует связаться с владельцами сайта. А может быть, владельцы сайта сами предложили вам проиндексировать информацию — например, в контексте более крупного контракта веб-разработки.

В этом разделе вы узнаете, как разработчики применяют извлечение данных на реальных сайтах. Затем мы рассмотрим необходимые инструменты, обеспечивающие эффективное применение Node для извлечения веб-данных.

Б.1.1. Применение извлечения веб-данных

Отличным примером извлечения веб-данных служит вертикальная поисковая система Octopart (*https://octopart.com/*). Octopart (рис. Б.2) индексирует данные производителей и продавцов электроники, чтобы пользователям было проще найти нужные компоненты. Например, вы можете провести поиск резисторов на основании величины сопротивления, допуска на номинал, класса мощности и типа корпуса. Такие сайты используют веб-боты для загрузки контента, скрейперы для анализа контента и извлечения интересных значений (например, величины допуска), и внутреннюю базу данных для хранения обработанной информации.

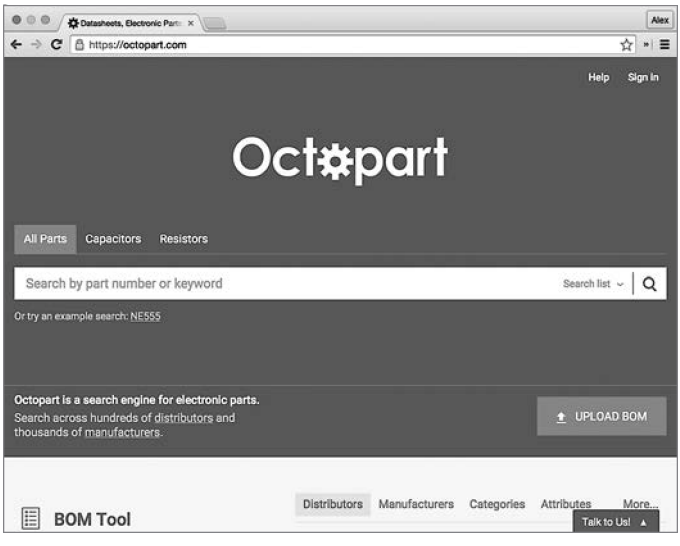


Рис. Б.2. Octopart предоставляет средства поиска электронных компонентов

Впрочем, извлечение веб-данных используется не только для поисковых систем. Оно также применяется в развивающихся областях теории обработки данных и в журналистике данных. Специалисты по журналистике данных используют базы данных для создания историй, но поскольку огромные объемы данных не хранятся в легкодоступных форматах, они могут использовать такие инструменты, как извлечение веб-данных, для автоматизации сбора и обработки данных. Все это дает журналистам возможность представления информации новыми способами, с применением средств визуализации данных, включая инфографику и интерактивную графику.

Б.1.2. Необходимые инструменты

Для выполнения работы по извлечению веб-данных вам понадобится пара легкодоступных инструментов: браузер и Node. Браузер — один из самых полезных инструментов извлечения данных; щелкнув правой кнопкой мыши и выбрав команду Исследовать элемент (Inspect Element), вы уже сделали первые шаги на пути анализа сайта и преобразования его в необработанные данные. Следующим шагом должен стать разбор страниц в Node. В этой главе мы рассмотрим два типа парсеров:

- облегченный и неприхотливый: cheerio;
- поддерживающий веб-стандарты и эмулирующий модель DOM: jsdom.

Обе библиотеки устанавливаются из npm. Возможно, вам потребуется разбирать форматы данных с нечеткой структурой, например даты. Мы кратко рассмотрим модули JavaScript `Date.parse` и Moment.js.

В первом примере используется cheerio — простой и удобный способ разбора статических веб-страниц.

Б.2. Простейшее извлечение веб-данных с использованием cheerio

Библиотека cheerio (*www.npmjs.com/package/cheerio*), написанная Феликсом Бемом (Felix B hm), идеально подходит для извлечения веб-данных, потому что она объединяет два ключевых аспекта: быстрый разбор HTML и API в стиле jQuery для запроса и выполнения операций с разметкой HTML.

Представьте, что вам нужно извлечь информацию о книгах на сайте издательства. У издателя еще нет API для получения информации о книгах, поэтому вам придется загрузить страницы с сайта и преобразовать их в результат в формате JSON, включающий в себя имя автора и название книги. На рис. Б.3 показано, как работает извлечение веб-данных в cheerio.

В листинге Б.1 приведена небольшая программа извлечения веб-данных на базе cheerio. В нее включен фрагмент разметки HTML, так что вам пока не нужно беспокоиться о загрузке самой страницы.

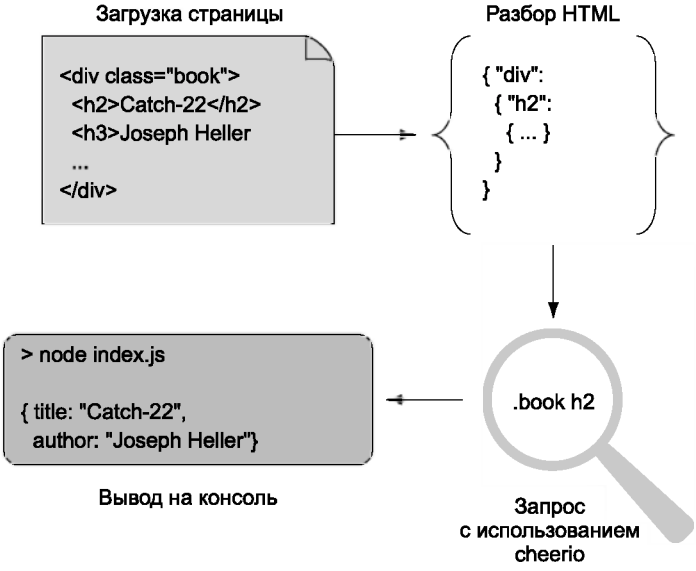


Рис. Б.3. Извлечение веб-данных с использованием cheerio

Листинг Б.1. Извлечение информации о книге

```
const html = `  <-- Определяет разметку HTML для разбора.
<html>
<body>
  <div class="book">
    <h2>Catch-22</h2>
    <h3>Joseph Heller</h3>
    <p>A satirical indictment of military madness.</p>
  </div>
</body>
</html>`;
const cheerio = require('cheerio');
const $ = cheerio.load(html);  <-- Разбирает весь документ.

const book = {
  title: $('.book h2').text(),  <-- Извлекает поля с использованием селекторов CSS.
  author: $('.book h3').text(),
  description: $('.book p').text()
};

console.log(book);
```

Листинг Б.1 использует cheerio для разбора жестко запрограммированного документа HTML с использованием метода `.load()` и селекторов CSS. В этом простом примере селекторы CSS просты и понятны, но реальная разметка HTML намного сложнее. К сожалению, вам неизбежно придется столкнуться с плохо структурированной разметкой HTML, и ваша квалификация аналитика веб-данных будет определяться вашим умением извлечь нужные значения.

Анализ плохой разметки HTML проходит в два этапа: сначала вы визуализируете документ, а затем определяете селекторы интересующих вас элементов. Для определения селекторов используется функциональность cheerio.

К счастью, современные браузеры предлагают простое решение для поиска селекторов: если ваш браузер содержит средства разработчика, обычно вы можете щелкнуть правой кнопкой мыши и выбрать команду Исследовать элемент (Inspect Element). Вы увидите соответствующую разметку HTML, а браузер также должен вывести представление селектора для этого элемента.

Предположим, вы пытаетесь извлечь информацию о книге с неряшливо написанного сайта, использующего таблицы без классов CSS. Разметка HTML может выглядеть так:

```
<html>
  <body>
    <h1>Alex's Dated Book Website</h1>
    <table>
      <tr>
        <td><a href="/book1">Catch-22</a></td>
        <td>Joseph Heller</td>
```

```
      </tr>
    </table>
  </body>
</html>
```

Если вы откроете этот фрагмент в Chrome и щелкнете правой кнопкой мыши на названии, вы увидите нечто похожее на рис. Б.4.

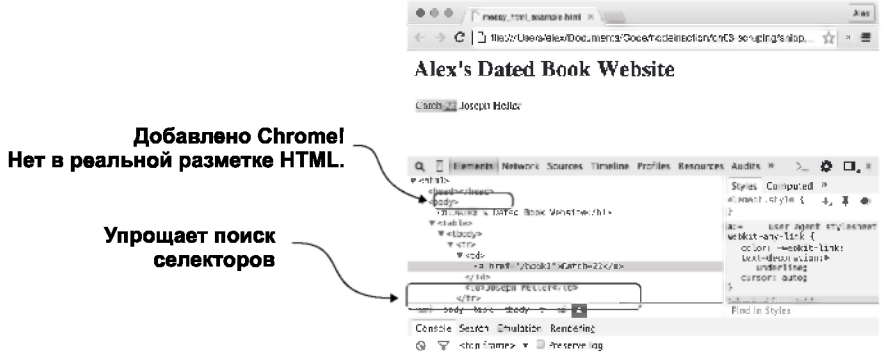


Рис. Б.4. Просмотр HTML в Chrome

В белой полосе под HTML выводится текст «html body table tbody tr td a» — довольно близкий к нужному селектору. Но это не совсем правильно, потому что в реальной разметке HTML нет `tbody` — этот элемент был вставлен Chrome. Когда вы используете браузер для визуализации элементов, будьте готовы скорректировать свои результаты на основании настоящей используемой разметки HTML. Этот пример показывает, что для получения названия следует найти ссылку внутри ячейки таблицы, а для получения соответствующего автора взять следующую ячейку.

Допустим, приведенная выше разметка HTML хранится в файле **messy_html_example.html**. Код из листинга Б.2 извлекает название, ссылку и автора.

Листинг Б.2. Обработка разметки HTML

```
const fs = require('fs');
const html = fs.readFileSync('./messy_html_example.html', 'utf8');
const cheerio = require('cheerio');
const $ = cheerio.load(html);

const book = {
  title: $('table tr td a').first().text(),
  href: $('table tr td a').first().attr('href'),
  author: $('table tr td').eq(1).text()
};

console.log(book);
```

Загружает HTML из файла.

Использует метод cheerio first() для получения конкретной ссылки.

Использует метод cheerio attr() для получения URL.

Использует метод cheerio eq() для перехода ко второму элементу.

Для загрузки HTML используется модуль fs; это делается для того, чтобы вам не приходилось вводить разметку HTML в примере. На практике источником данных может служить живой веб-сайт, но данные также могут загружаться из файла или базы данных. После того как разбор документа будет завершен, вызов `first()` используется для получения первой ячейки таблицы со ссылкой. Для получения URL-адреса ссылки используется метод cheerio `attr()`; он возвращает конкретный атрибут элемента, как и jQuery. Также заслуживает внимания метод `eq()`; в этом листинге он пропускает первый элемент `td`, потому что имя автора содержится во втором элементе.

ОПАСНОСТИ РАЗБОРА ВЕБ-ДАННЫХ

С помощью такого модуля, как cheerio, вы сможете легко и быстро интерпретировать веб-документы. Но будьте осторожны с типом контента, который вы пытаетесь обрабатывать. Например, с двоичными данными может быть выдано исключение, так что использование модуля в веб-приложении может привести к аварийному завершению процесса Node. Может быть рискованно, если программа извлечения данных встроена в процесс, обслуживающий ваше веб-приложение.

Лучше проверить тип контента перед тем, как передавать его парсеру; также рассмотрите возможность выполнения программы извлечения веб-данных в отдельных процессах Node, чтобы снизить последствия любых серьезных сбоев.

Одно из ограничений cheerio заключается в том, что работать можно только со статической версией документа; cheerio используется для работы с «чистыми» документами HTML, но не с динамическими страницами, использующими JavaScript на стороне клиента. В следующем разделе вы узнаете, как использовать jsdom для моделирования среды браузера в своих приложениях Node для выполнения кода JavaScript на стороне клиента.

Б.3. Обработка динамического контента с jsdom

Библиотека jsdom — настоящая мечта специалиста по извлечению веб-данных; она загружает разметку HTML, интерпретирует ее в соответствии с моделью DOM типичного браузера и выполняет клиентский код JavaScript. Вы можете указать клиентский код JavaScript, который необходимо выполнить; обычно это означает включение jQuery. Это означает, что вы можете внедрять jQuery (или ваши собственные сценарии отладки) в любые страницы. На рис. Б.5 показано, как jsdom сочетает HTML и JavaScript, чтобы получить доступ к контенту, обычно недоступному для извлечения.

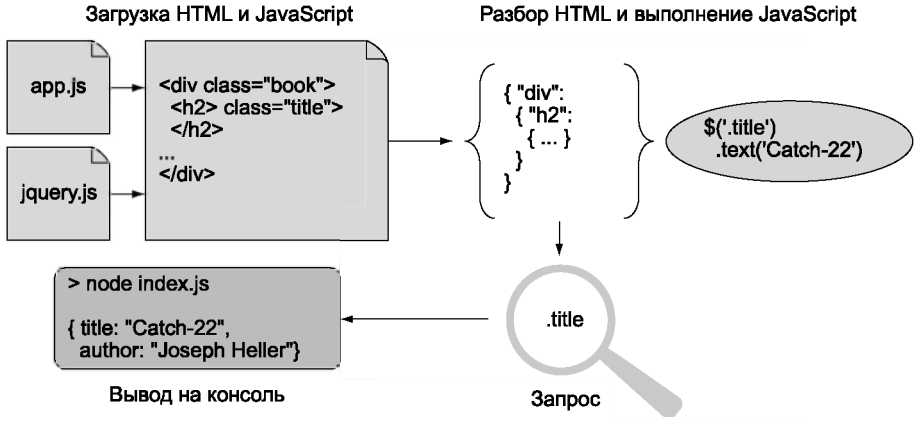


Рис. Б.5. Извлечение веб-данных с использованием jsdom

У jsdom также есть свои недостатки. Библиотека не является идеальной моделью браузера, она работает медленнее cheerio, а парсер HTML работает в жестком режиме, так что на страницах с плохо написанной разметкой могут происходить сбои. Впрочем, некоторые сайты не имеют смысла без поддержки JavaScript на стороне клиента, поэтому jsdom остается неоценимым инструментом для некоторых задач извлечения веб-данных.

Обычно jsdom используется через метод `jsdom.env`. В листинге Б.3 продемонстрировано применение jsdom для извлечения данных со страницы посредством анализа jQuery и извлечения полезных значений.

Листинг Б.3. Извлечение данных с jsdom

```
const jsdom = require('jsdom');
const html = `                          <-- Включает подходящий фрагмент HTML.
<div class="book">
  <h2>Catch-22</h2>
  <h3>Joseph Heller</h3>
  <p>A satirical indictment of military madness.</p>
</div>
`;

jsdom.env(html, ['./node_modules/jquery/dist/jquery.js'], scrape);
                                        <-- Разбирает документ и загружает jQuery.

function scrape(err, window) {
  var $ = window.$;                     <-- Определяет удобный псевдоним для объекта jQuery.
  $('.book').each(function() {          <-- Перебирает книги методом jQuery $.each.
    var $el = $(this);
    console.log({
      title: $el.find('h2').text(),     <-- Использует методы jQuery для извлечения значений, относящихся к книге.
      author: $el.find('h3').text(),
      description: $el.find('p').text()
    });
  });
}
```

Чтобы выполнить листинг Б.3, необходимо сохранить jQuery локально и установить jsdom[1]. И то и другое можно сделать в npm; модули называются jsdom (*www.npmjs.com/package/jsdom*) и jQuery (*www.npmjs.com/package/jquery*) соответственно. А когда все будет сделано, этот код выведет название книги, автора и описание из фрагмента HTML.

Метод `jsdom.env` используется для разбора документа и внедрения jQuery. Внедрение jQuery осуществляется загрузкой из npm, но вы также можете указать URL-адрес jQuery в сети CDN или вашей файловой системе; jsdom будет знать, что делать. Метод `jsdom.env` работает асинхронно; для работы ему необходима функция обратного вызова. Эта функция получает объекты ошибки и окна; для обращения к документу используется объект окна. В этом примере для объекта окна определяется псевдоним, чтобы к нему можно было легко обращаться в записи `$`.

Селектор используется с методом jQuery `.each` для перебора книг. В этом примере книга всего одна, но она демонстрирует, что методы обхода jQuery действительно работают. Для обращения ко всем значениям из книги также используются методы обхода jQuery.

Листинг Б.3 похож на пример для cheerio из листинга Б.1. Принципиальное различие заключается в том, что jQuery разбирается и выполняется Node в текущем процессе. В листинге Б.1 cheerio используется для предоставления аналогичной функциональности, но cheerio предоставляет собственную прослойку, похожую на jQuery. На этот раз вы выполняете код, предназначенный для браузера, так, словно он действительно выполняется в браузере.

Метод `jsdom.env` полезен только при работе со статическими страницами. Для разбора страниц, использующих клиентский код JavaScript, вместо него следует использовать `jsdom.jsdom`. Этот синхронный метод возвращает объект окна, с которым можно работать другими средствами jsdom. В листинге Б.4 jsdom используется для разбора документа с тегом `script`, а метод `jsdom.jQueryify` упрощает извлечение данных.

Листинг Б.4. Разбор динамической разметки HTML с использование jsdom

```
const jsdom = require('jsdom');
const jqueryPath = './node_modules/jquery/dist/jquery.js';   <-- Задает путь jQuery.
const html = `
<div class="book">
  <h2></h2>     <-- HTML без статических значений.
  <h3></h3>
  <script>                                                   <-- Сценарий, который динамически вставляет значение.
document.querySelector('h2').innerHTML = 'Catch-22';
document.querySelector('h3').innerHTML = 'Joseph Heller';
  </script>
</div>
`;
```

[1] На момент написания книги текущей была версия jsdom 6.3.0.

```
const doc = jsdom.jsdom(html);  <-- Создает объект, представляющий документ.
const window = doc.defaultView;
jsdom.jQueryify(window, jqueryPath, function() {  <-- Вставляет jQuery в документ.
  var $ = window.$;
  $('.book').each(function() {
    var $el = $(this);
    console.log({
      title: $el.find('h2').text(),  <-- Извлекает значения, связанные с книгой.
      author: $el.find('h3').text()
    });
  });
});
```

Для работы листинга кода Б.4 необходима установка jQuery, и, если код создается вручную, вы должны создать новый проект командами `npm init` и `npm install --save jquery jsdom`. Он использует простой документ HTML, в котором интересующие нас значения вставляются динамически. Вставка осуществляется клиентским кодом JavaScript из тега `script`.

На этот раз вместо `jsdom.env` используется `jsdom.jsdom`. Он выполняется синхронно, потому что объект документа создается в памяти, но ничего не делает до тех пор, пока вы не попытаетесь обратиться к нему с запросом или выполнить какую-нибудь операцию. Для этого вызов `jsdom.jQueryify` используется для вставки конкретной версии jQuery в документ. После того как библиотека jQuery будет загружена и заработает, выполняется обратный вызов, который запрашивает у документа интересующие вас значения и выводит их на консоль. Результат выглядит так:

```
{ title: 'Catch-22', author: 'Joseph Heller' }
```

Это доказывает, что библиотека jsdom вызвала необходимый код JavaScript на стороне клиента. Теперь представьте, что это реальная веб-страница — и вам станет ясно, почему библиотека jsdom настолько мощна: она позволяет извлекать данные даже с сайтов, построенных с минимумом разметки HTML на базе таких динамических технологий, как Angular и React.

Б.4. Обработка «сырых» данных

После того как полезные данные будут наконец-то прочитаны со страницы, их необходимо обработать для сохранения в базе данных или для экспортирования в таком формате, как CSV. Извлеченные данные либо состоят из неструктурированного обычного текста, либо кодируются в микроформатах.

Микроформаты — облегченные форматы данных на основе разметки, используемые для таких данных, как адреса, календари и события, с тегами или ключевыми словами. Информацию о выработанных микроформатах можно найти на сайте *microformats.org*. Пример имени, представленного в микроформате:

```
<a class="h-card" href="http://example.com">Joseph Heller</a>
```

Микроформаты относительно просто разбираются; с cheerio или jsdom для извлечения имени `Joseph Heller` достаточно простого выражения вида `$('.h-card').text()`. Простой текст потребует большей работы. В этом разделе вы увидите, как разобрать данные и как преобразовать их в форматы, более подходящие для баз данных.

На большинстве веб-страниц микроформаты не используются. Одна из областей, в которых ситуация может создать проблемы, но остается потенциально управляемой, — значения данных. Данные могут встречаться во многих форматах, но в пределах одного сайта они обычно остаются последовательными. После того как формат будет определен, вы сможете разобрать и отформатировать данные.

В JavaScript имеется встроенный парсер данных: при выполнении команды `new Date('2016 01 01')` возвращается новый экземпляр `Date`, соответствующий 1 января 2016 года. Поддерживаемые входные форматы определяются методом `Date.parse`, который базируется на стандартах RFC 2822 (*http://tools.ietf.org/html/rfc2822#page-14*) или ISO 8601 (*www.w3.org/TR/NOTE-datetime*). Также могут сработать другие форматы; нередко стоит опробовать их с исходными данными и посмотреть, что получится.

Другое решение основано на поиске значений в исходных данных с использованием регулярного выражения, с последующим использованием конструктора `Date` для создания новых объектов `Date`. Конструктор имеет следующую сигнатуру:

```
new Date(year, month[,day[,hour[,minutes[,seconds[,millis]]]]]);
```

Возможностей разбора данных в JavaScript обычно хватает для разных случаев, но при реформатировании данных его недостаточно. Превосходным решением проблемы может стать Moment.js (*http://momentjs.com*) — библиотека разбора, проверки и форматирования дат. Moment.js обладает динамичным API, что позволяет использовать сцепленные вызовы следующего вида:

```
moment().format("MMM Do YY"); // Sep 7th 15
```

Это удобно для преобразования извлеченных данных в CSV-файлы, хорошо работающие во многих программах (таких, как Microsoft Excel). Представьте, что у вас имеется веб-страница с книгами, включающая в себя название и дату публикации. Значения нужно сохранить в базе данных, но ваша база данных требует, чтобы данные были отформатированы по схеме ГГГГ-ММ-ДД. Листинг Б.5 показывает, как использовать Moment с cheerio для этой цели.

Листинг Б.5. Разбор данных и генерирование CSV

```
'use strict';
const cheerio = require('cheerio');
const fs = require('fs');
const html = fs.readFileSync('./input.html');   <-- Загружает входной файл.
const moment = require('moment');   <-- Включает moment.
```

```
const $ = cheerio.load(html);
const books = $('.book')
  .map((i, el) => {                    <-- Связывает каждую книгу с автором, названием и датой публикации.
    return {
      author: $(el).find('h2').text(),
      title: $(el).find('h3').text(),
      published: $(el).find('h4').text()
    };
  })
  .get();

console.log('title, author, sourceDate, dbDate');   <-- Заголовки файла CSV.

books.forEach((book) => {
  let date = moment(new Date(book.published));   <-- Разбирает дату.
  console.log(
    '%s, %s, %s, %s',
    book.author,
    book.title,
    book.published,
    date.format('YYYY-MM-DD')
  );
});
```

Для выполнения листинга Б.5 необходимо установить cheerio, Moment и информацию книг. Программа получает разметку HTML (из input.html) и выводит данные CSV. В разметке HTML даты должны храниться в элементах `h4`:

```
<div>
  <div class="book">
    <h2>Catch-22</h2>
    <h3>Joseph Heller</h3>
    <h4>11 November 1961</h4>
  </div>
  <div class="book">
    <h2>A Handful of Dust</h2>
    <h3>Evelyn Waugh</h3>
    <h4>1934</h4>
  </div>
</div>
```

После того как программа загрузит входной файл, она загружает Moment, а затем связывает каждую книгу с простым объектом JavaScript при помощи методов cheerio `.map` и `.get`. Метод `.map` перебирает книги, а обратный вызов извлекает каждый интересующий вас элемент при помощи метода обхода селекторов `.find`. Для получения итогового текста в виде массива используется метод `.get`.

Листинг Б.5 выводит данные в формате CSV методом `console.log`. Сначала выводится заголовок, а затем каждая строка в цикле, перебирающем все книги. Даты преобразуются в формат, совместимый с MySQL; дата сначала разбирается выражением `new Date`, а затем форматируется средствами Moment.

Освоившись с разбором и форматированием дат, вы сможете применить аналогичные методы к другим форматам данных. Например, денежные суммы и расстояния можно извлекать с применением регулярных выражений, а затем форматировать с применением более общих библиотек числового форматирования — таких, как Numeral (*www.npmjs.com/package/numeral*).

Б.5. Заключение

- Извлечение веб-данных представляет собой автоматизированное преобразование веб-страниц (иногда плохо структурированных) в форматы, удобные для компьютерной обработки — CSV, базы данных и т. д.
- Извлечение веб-данных применяется вертикальными поисковыми системами и в области журналистики данных.
- Если вы собираетесь извлечь данные с сайта, сначала получите разрешение. Для этого можно проверить содержимое файла robots.txt и связаться с владельцем сайта.
- Основные инструменты — парсеры статической разметки HTML (cheerio) и парсеры, способные выполнять JavaScript (jsdom); также пригодятся средства разработки браузера для нахождения подходящего селектора CSS для интересующих вас элементов.
- Иногда сами данные плохо отформатированы, и вам приходится разбирать такие величины, как даты или денежные суммы, чтобы их можно было сохранить в базе данных.

B Официально поддерживаемые промежуточные компоненты

Connect представляет собой минимальную обертку для встроенных модулей клиента и сервера HTTP в Node. Авторы и участники проекта Connect также предоставляют официально поддерживаемые промежуточные компоненты (middleware components) для реализации низкоуровневых функций, используемых многими веб-фреймворками, включая такие операции, как обработка cookie, разбор тела, сеансы, базовая аутентификация и межсайтовая фальсификация запросов (CSRF, Cross-Site Request Forgery). В этом приложении описаны все официально поддерживаемые модули, чтобы вы могли использовать их для построения облегченных веб-приложений без более крупного фреймворка.

B.1. Разбор cookie, тел запросов и строк информационных запросов

Ядро Node не предоставляет модулей, реализующих высокоуровневую функциональность веб-приложений, такую как синтаксический разбор cookie, буферизация тел обычных запросов или синтаксический разбор строк информационных запросов, — все эти возможности реализованы в модулях Connect. В этом разделе рассматриваются четыре модуля для разбора данных запросов:

- cookie-parser — разбор cookie из веб-браузеров и сохранение результатов в `req.cookies`;
- qs — разбор строки URL запроса в `req.query`;
- bodyParser — использование и разбор тела запроса в `req.body`.

Начнем с модуля cookie-parser. Он упрощает получение данных, хранимых браузером посетителя сайта, чтобы вы могли прочитать статус авторизации, настройки веб-сайтов и т. д.

B.1.1. cookie-parser: разбор HTTP-cookie

Модуль cookie-parser поддерживает разбор обычных cookie, подписанных cookie и специальных cookie формата JSON (*www.npmjs.com/package/cookie-parser*). По умолчанию используются обычные неподписанные cookie, при этом заполняется значениями объект `req.cookies`. Если же вам нужна поддержка подписанных cookie, затрудняющая манипуляции с cookie, то при создании экземпляра `cookie-parser` следует передать секретную строку.

НАСТРОЙКА COOKIE НА СЕРВЕРЕ

Модуль cookie-parser() не предоставляет никаких вспомогательных функций для настройки исходящих cookie. Для этого нужно использовать функцию res.setHeader() с Set-Cookie в качестве заголовка. Среда Connect исправляет заданную по умолчанию функцию Node res.setHeader() таким образом, чтобы использовать заголовки Set-Cookie специального вида. В результате функция работает так, как надо.

Обычные cookie

Чтобы прочитать cookie, необходимо загрузить модуль, добавить его в стек промежуточного ПО, а затем прочитать cookie в запросе. В листинге B.1 продемонстрирован каждый из этих шагов.

Листинг B.1. Чтение cookie, отправленных в запросе

```
const connect = require('connect');
const cookieParser = require('cookie-parser');

connect()
  .use(cookieParser())
  .use((req, res, next) => {
    res.end(JSON.stringify(req.cookies));
  })
  .listen(3000);
```

(1) Загружает промежуточный компонент cookie-parser.

(2) Добавляет его в стек промежуточных компонентов приложения.

(3) Отвечает строковой версией cookie.

В этом фрагменте загружается промежуточный компонент **(1)**. Помните, что для этого необходимо установить компонент командой `npm install cookie-parser`. Затем экземпляр `cookie-parser` добавляется в стек промежуточных компонентов этого приложения **(2)**. Остается сделать последний шаг — вернуть cookie браузеру в строке **(3)**, чтобы вы могли убедиться в правильности работы.

При запуске этого примера необходимо назначить cookie с запросом. Открыв адрес *http://localhost:3000* в браузере, скорее всего, вы ничего не увидите; возвращается пустой объект (`{}`). Для назначения cookie можно воспользоваться cURL:

```
curl http://localhost:3000/ -H "Cookie: foo=bar, bar=baz"
```

Подписанные cookie

Подписанные cookie лучше всего использовать при передаче важных данных, поскольку их целостность можно проверить для предотвращения атак через посредника (man-in-the-middle attack). Если проверка будет пройдена, подписанные cookie помещаются в объект `req.signedCookies`. Два отдельных объекта используются для прояснения намерений разработчика. Если бы подписанные и неподписанные cookie хранились в одном объекте, обычные cookie можно было бы изменить таким образом, чтобы имитировать подписанные cookie.

Содержимое подписанного cookie может выглядеть примерно так: `tobi.DDm3AcVxE9oneYnbmpqxoy[…]`[1]; слева от точки (.) находится значение cookie, а справа — хеш-код, сгенерированный сервером по алгоритму SHA-256 в формате HMAC (Hash-based Message Authentication Code). Если Connect предпримет попытку убрать подпись cookie-файла, ничего не получится, поскольку окажется измененным значение или HMAC-код.

Предположим, например, что в вашем распоряжении имеется подписанный объект cookie с ключом `name` и значением `luna`. Вызов `cookieParser` закодирует cookie таким образом, что получится значение `s:luna.PQLM0wNvqOQEObZX[…]`. Хеш-часть проверяется для каждого запроса, и, если cookie отправляется в неизменном виде, к данным можно обратиться в виде `req.signedCookies.name`:

```
$ curl http://localhost:3000/ -H "Cookie:
  name=s:luna.PQLM0wNvqOQEObZXU[…]"
{}
{ name: 'luna' }
GET / 200 4ms
```

Если изменить значение cookie, как в следующей команде `curl`, cookie `name` будет доступно в записи `req.cookies.name`, потому что оно не прошло проверку. При этом оно может пригодиться для отладки или специфических целей приложения:

```
$ curl http://localhost:3000/ -H "Cookie:
  name=manny.PQLM0wNvqOQEOb[…]"
{ name: 'manny.PQLM0wNvqOQEOb[…]' }
{}
GET / 200 1ms
```

В первом аргументе `cookieParser` передается секретная строка, используемая для подписывания cookie. В листинге B.2 используется строка «tobi is a cool ferret».

Листинг B.2. Разбор подписанных cookie

```
const connect = require('connect');
const cookieParser = require('cookie-parser');
const secret = 'tobi is a cool ferret';
```

[1] Значение сокращено для экономии места.

```
const connect = require('connect');
const cookieParser = require('cookie-parser');
const secret = 'tobi is a cool ferret';

connect()
  .use(cookieParser(secret))
  .use((req, res) => {
    console.log('Cookies:', req.cookies);
    console.log('Signed cookies:', req.signedCookies);
    res.end('hello\n');
  }).listen(3000);
```

(1) Подписанные cookie автоматически добавляются в объект запроса.

(2) Обращается к подписанным cookie из объекта запроса.

В этом примере подписанные cookie разбираются автоматически, потому что промежуточному компоненту `cookieParser` был передан аргумент `secret` **(1)**. Обращение к значениям происходит через объект `request` **(2)**. Модуль cookie-parser также открывает доступ к функциональности разбора cookie, предоставляемой через методы `signedCookie` и `signedCookies`.

Прежде чем двигаться дальше, посмотрим, как использовать этот пример. Как и в случае с листингом B.1, для отправки cookie можно воспользоваться командой `curl` с ключом `-H`. Но чтобы объект cookie можно было считать подписанным, он должен быть закодирован определенным образом.

Модуль Node crypto используется для отмены подписи cookie методом `signedCookie`. Если вы хотите подписать cookie для тестирования листинга B.2, установите cookie-signature и подпишите строку с той же секретной строкой:

```
const signature = require('cookie-signature');
const message = 'luna';
const secret = 'tobi is a cool ferret';
console.log(signature.sign(message, secret));
```

Теперь в случае изменения подписи или сообщения сервер узнает об этом. Кроме подписанных cookie модуль поддерживает cookie в кодировке JSON. В следующем разделе показано, как работает этот вариант.

Cookie в формате JSON

Специальные cookie в формате JSON снабжаются префиксом `j:`, который распознается Connect как сериализуемый формат JSON. Cookie в формате JSON могут быть как подписанными, так и неподписанными.

Такие фреймворки, как Express, могут использовать этот формат для создания интуитивно понятного интерфейса для работы с cookie — вместо того, чтобы выполнять сериализацию вручную и разбирать значения cookie в формате JSON. Следующий пример демонстрирует, как Connect разбирает cookie в формате JSON:

```
$ curl http://localhost:3000/ -H 'Cookie: foo=bar,
bar=j:{"foo":"bar"}'
{ foo: 'bar', bar: { foo: 'bar' } }
{}
GET / 200 1ms
```

Как упоминалось ранее, JSON-cookie также могут быть подписанными, как показано в следующем примере:

```
$ curl http://localhost:3000/ -H "Cookie:
  cart=j:{\"items\":[1]}.sD5p6xFFBO/4ketA1OP43bcjS3Y"
{}
{ cart: { items: [ 1 ] } }
GET / 200 1ms
```

Настройка исходящих cookie

Как уже отмечалось, модуль cookie-parser не предлагает никакой функциональности для записи исходящих заголовков для HTTP-клиента с помощью заголовка `Set-Cookie`. В Connect предусмотрена явная поддержка нескольких заголовков `Set-Cookie`, обеспечиваемая функцией `res.set-Header()`.

Предположим, что нужно задать для cookie с именем `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 (например, срок действия), как показано во втором вызове `set-Header()`:

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

connect()
  .use((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 запрос HTTP. 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
```

Вот и все, что относится к способам передачи cookie с ответом HTTP. В cookie можно хранить произвольные текстовые данные, но чаще всего на стороне сеанса хранится

один объект cookie для каждого сеанса, чтобы сервер мог получить полную информацию о состоянии пользователя. Требуемый механизм инкапсулирован в модуле express-session, который рассматривается далее в этом приложении.

Теперь, когда вы научились работать с cookie, вам, вероятно, не терпится взяться за другие стандартные методы получения пользовательского ввода. В следующих двух разделах рассматривается разбор строк и тел запросов; вы узнаете, что, хотя Connect работает на достаточно низком уровне, можно воспроизвести функциональность более сложных веб-фреймворков без написания больших объемов кода.

B.1.2. Разбор строк запросов

Один из способов получения ввода основан на использовании параметров GET. После URL-адреса ставится вопросительный знак, за которым следует список аргументов, разделенных символами &:

```
http://localhost:3000/page?name=tobi&species=ferret
```

URL-адреса такого типа могут предоставляться вашему приложению формой, настроенной на использование метода GET, или якорными элементами в шаблонах приложения. Вероятно, вы уже видели, как этот способ передачи данных применяется для разбивки вывода на страницы.

Объект запроса, передаваемый каждому промежуточному компоненту в приложениях Connect, включает в себя свойство `url`, но в данном случае вас интересует последняя часть URL: только символы, следующие за вопросительным знаком. В поставку Node входит модуль разбора URL, так что формально для получения нужной строки можно воспользоваться методом `url.parse`. Но Connect также приходится разбирать URL, и разобранная версия сохраняется во внутреннем свойстве.

Для разбора строк запросов рекомендуется использовать модуль qs (*www.npmjs.com/package/qs*). Этот модуль официально не поддерживается Connect; альтернативные решения доступны через npm. Чтобы использовать модуль qs и другие похожие модули, вызовите его метод `.parse()` из своего промежуточного компонента.

Простой пример использования

В листинге B.3 метод `qs.parse` используется для создания объекта, хранящегося в свойстве `req.query`, для использования последующими промежуточными компонентами.

Листинг B.3. Разбор строк запросов

```
const connect = require('connect');
const qs = require('qs');
connect()
  .use((req, res, next) => {
```

```
    console.log(req._parsedUrl.query);
    req.query = qs.parse(req._parsedUrl.query); ◄── (1) Использует qs для разбора строки запроса.
    next();
  })
  .use((req, res) => {
    console.log('query string:', req.query); ◄──── Выводит разобранную строку.
    res.end('\n');
  })
  .listen(3000);
```

В этом примере нестандартный промежуточный компонент используется для получения разобранного URL-адреса, его разбора с использованием `qs.parse` **(1)** и его последующего вывода в следующем компоненте.

Предположим, вы проектируете приложение для построения музыкальной фонотеки. В приложение можно включить поисковую систему и использовать строку запроса для формирования параметров поиска:

```
/songSearch?artist=Bob%20Marley&track=Jammin.
```

В этом примере будет получен объект `res.query` следующего вида:

```
{ artist: 'Bob Marley', track: 'Jammin' }
```

Метод `qs.parse` поддерживает вложенные массивы, так что сложные строки запросов, такие как `?images[]=foo.png&images[]=bar.png`, будут создавать объекты следующего вида:

```
{ images: [ 'foo.png', 'bar.png' ] }
```

Если в запросе HTTP параметры строки запроса не заданы (например, */song-Search*), `req.query` по умолчанию содержит пустой объект:

```
{}
```

Фреймворки более высокого уровня, такие как Express, обычно содержат встроенные средства разбора строк запросов, потому что эта задача очень часто встречается в веб-разработке. Еще одна стандартная функция веб-фреймворков — разбор тел запросов для получения данных, отправленных формами. Следующий раздел объясняет, как разбирать тела запросов, работать с формами и отправленными файлами и как убедиться в безопасности таких запросов.

B.1.3. body-parser: разбор тел запросов

Большинство веб-приложений получает и обрабатывает пользовательский ввод. Данные могут поступать от форм или даже от других программ (в случае REST-совместимых API). Запросы и ответы HTTP называются *сообщениями HTTP*. Формат сообщения состоит из списка заголовков и тела сообщения. В веб-приложениях Node тело обычно представляет собой поток данных и может кодироваться разными

способами: запрос POST от формы обычно кодируется в формате `application/x-www-form-urlencoded`, а REST-совместимый запрос JSON может быть закодирован в формате `application/json`.

Это означает, что приложениям Connect необходимы промежуточные компоненты, способные декодировать потоки данных в кодировке формы, JSON или даже в формате сжатых данных с применением gzip или deflate. В этом разделе мы покажем, как решаются следующие задачи:

- обработка ввода от форм;
- разбор запросов JSON;
- проверка тел запросов на основании контента и размера;
- получение отправленных файлов.

Формы

Предположим, нужно принять регистрационную информацию для вашего приложения от формы. Для решения этой задачи достаточно поместить компонент body-parser (*www.npmjs.com/package/body-parser*) перед любым другим промежуточным компонентом, который будет обращаться к объекту `req.body`. На рис. B.1 показано, как это происходит.

Рис. B.1. Обработка формы модулем body-parser

Листинг B.4 демонстрирует использование модуля body-parser с POST-запросами HTTP от форм.

Листинг B.4. Разбор запросов форм

```
const connect = require('connect');
const bodyParser = require('body-parser');

connect()
  .use(bodyParser.urlencoded({ extended: false }))
  .use((req, res, next) => {
    res.setHeader('Content-Type', 'text/plain');
    res.end('You sent: ' + JSON.stringify(req.body) + '\n');
  })
  .listen(3000);
```

(1) Добавляет body-parser в стек промежуточных компонентов.

(2) Возвращает тело запроса в виде строки.

Чтобы использовать этот пример, необходимо установить модуль body-parser[1], а затем выдать простой запрос HTTP с телом в URL-кодировке. Самый простой способ — использование `curl` с ключом `-d`:

```
curl -d name=tobi http://localhost:3000
```

В результате сервер должен вывести текст `You sent: {"name":"tobi"}`. Чтобы этот способ работал, необходимо добавить body-parser в стек промежуточных компонентов **(1)**, а затем разобранное тело в `req.body` преобразуется в строку **(2)** для удобства вывода. Парсер `urlencoded` получает строку в кодировке UTF-8 и автоматически распаковывает тела запросов, закодированные gzip или deflate.

В этом примере парсеру тела передаются параметры `extended: false`. Когда этот параметр равен `true`, парсер использует другую библиотеку для разбора формата строки. Это позволяет использовать более сложные вложенные JSON-подобные объекты в формах. Другие параметры представлены в следующем разделе, посвященном проверке запросов.

Проверка запросов

Каждый парсер, поставляемый с модулем body-parser, поддерживает два режима проверки запросов: `limit` и `verify`. Режим `limit` позволяет блокировать запросы с размером больше некоторого порога: по умолчанию этот размер равен 100 Кбайт, однако его можно увеличить, если вы захотите получать формы большего размера. Если вы создаете блог, систему управления контентом или что-нибудь в этом роде, где пользователь теоретически может вводить действительные, но очень длинные данные, такая возможность будет очень полезной.

Режим `verify` позволяет использовать функцию для проверки запросов. Например, она пригодится, если вы хотите получить низкоуровневое тело запроса и убедиться

[1] Мы использовали версию 1.11.0.

в том, что оно имеет правильный формат. Скажем, вы можете воспользоваться ею и убедиться в том, что методы API, получающие XML, всегда начинали с правильного заголовка XML. В листинге B.5 продемонстрировано использование обоих параметров.

Листинг B.5. Проверка запросов форм

```
const connect = require('connect');
const bodyParser = require('body-parser');

function verifyRequest(req, res, buf, encoding) {
  if (!buf.toString().match(/^name=/)) {
    throw new Error('Bad format');  <-- (1) Выдает ошибку в случае неправильного формата.
  }
}

connect()
  .use(bodyParser.urlencoded({
    extended: false,
    limit: 10,  <-- (2) Задает лимит запроса.
    verify: verifyRequest  <-- (3) Добавляет функцию проверки.
  }))
  .use(function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('You sent: ' + JSON.stringify(req.body) + '\n');
  })
  .listen(3000);
```

Обратите внимание: объект `Error` инициируется ключевым словом `throw` **(1)**. Модуль body-parser настраивается для перехвата этих ошибок перед обработкой запроса, поэтому ошибка будет возвращена Connect. После того как функция проверки запроса будет создана, вы должны передать ее промежуточному компоненту body-parser посредством использования режима `verify` **(3)**.

Лимит размера тела задается в байтах; здесь он достаточно мал — всего 10 байт **(2)**. Чтобы увидеть, что происходит при слишком большом запросе, используйте команду `curl` с большим значением `name`. Кроме того, если вы хотите узнать, что произойдет при выдаче ошибки проверки, используйте `curl` для отправки другого значения вместо `name`.

Для чего нужно ограничение?

А теперь посмотрим, как злоумышленник может вывести из строя уязвимый сервер. Сначала создадим небольшое приложение Connect с именем server.js. Это приложение делает очень мало: оно разбирает тело запроса с помощью промежуточного компонента `bodyParser()`:

```
const connect = require('connect');
const bodyParser = require('body-parser');

connect()
```

```
  .use(bodyParser.json({ limit: 99999999, extended: false }))
  .use((req, res, next) => {
    res.end('OK\n');
  })
  .listen(3000);
```

А теперь создайте файл **dos.js**, представленный в следующем листинге. Как видите, хакер может воспользоваться HTTP-клиентом в Node для атаки приведенного выше приложения Connect. Для организации подобной атаки достаточно записать несколько мегабайтов данных JSON.

```
const http = require('http');
let req = http.request({
  method: 'POST',
  port: 3000,
  headers: {
    'Content-Type': 'application/json'
  }
});
req.write('[');
let n = 300000;
while (n--) {
  req.write('"foo",');
}
req.write('"bar"]');
req.end();
```

Сообщает серверу о том, что вы отправляете данные JSON.

Начинает отправку большого объекта массива.

Массив содержит 300 000 строк «foo».

Запустите сервер и выполните сценарий атаки:

```
$ node server.js &
$ node dos.js
```

Понаблюдав за процессом `node` в `top(1)`, вы увидите, что во время выполнения **dos.js** он начинает использовать больше ресурсов процессора и оперативной памяти. Конечно, это плохо, но зато теперь вы понимаете, почему промежуточные компоненты разбора тела поддерживают режим `limit`.

Разбор данных JSON

Каждому разработчику, который создает веб-приложения с Node, приходится часто иметь дело с JSON. Парсер JSON модуля body-parser поддерживает ряд полезных параметров, которые вы видели в предыдущих примерах. Листинг B.6 показывает, как происходит разбор разметки JSON и обработка полученных значений.

Листинг B.6. Проверка запросов форм

```
const connect = require('connect');
const bodyParser = require('body-parser');

connect()
  .use(bodyParser.json())
  .use((req, res, next) => {
```

(1) Добавляет парсер JSON.

```
  res.setHeader('Content-Type', 'application/json');
  res.end(`Name: ${req.body.name}\n`);   <-- (2) Получает значение из объекта body.
})
.listen(3000);
```

После того как парсер JSON будет загружен **(1)**, ваши обработчики запросов могут интерпретировать значение `req.body` как объект JavaScript вместо строки. Данный пример предполагает, что был отправлен объект JSON со свойством `name`, и возвращает в ответе значение **(2)**. Это означает, что у запроса заголовок `Content-Type` должен содержать `application/json`, а вы должны отправить действительную разметку JSON. По умолчанию промежуточный компонент `json` использует строгий режим разбора, но вы можете смягчить требования к кодировке, присвоив параметру значение `false`.

> **НАЗНАЧЕНИЕ ЗАГОЛОВКА CONTENT-TYPE ДЛЯ JSON**
>
> Один из параметров, о котором вам необходимо знать, — type. Он позволяет изменить тип разбираемого контента на JSON. В следующем примере используется значение по умолчанию application/json. Но в некоторых случаях приложению приходится взаимодействовать с клиентами HTTP, которые не отправляют этот заголовок; будьте внимательны.

Следующий запрос `curl` может использоваться для отправки данных приложению, которое отправляет объект JSON со свойством `username`, которому присвоено значение `tobi`:

```
curl -d '{"name":"tobi"}' -H "Content-Type: application/json"
  http://localhost:3000
Name: tobi
```

Разбор многосекционных данных <form>

Модуль body-parser не поддерживает многосекционные (multipart) тела запросов. Для поддержки отправки файлов необходимо обрабатывать многосекционные сообщения, поэтому для таких операций, как отправка аватара пользователя, понадобится многосекционная поддержка.

Официально поддерживаемого многосекционного парсера для Connect не существует, но некоторые популярные решения качественно сопровождаются. Два примера — busboy (*www.npmjs.com/package/busboy*) и multiparty (*www.npmjs.com/package/multiparty*). У обоих модулей имеются соответствующие модули Connect: connect-busboy и connect-multiparty. Это происходит из-за того, что сами многосекционные парсеры зависят от низкоуровневых HTTP-модулей Node, что позволяет использовать их с разными фреймворками, не ограничиваясь исключительно Connect.

Листинг B.7 использует multiparty и выводит информацию об отправленном файле на консоль.

Листинг B.7. Обработка отправленных файлов

```
const connect = require('connect');
const multipart = require('connect-multiparty');

connect()
  .use(multipart())  <-------  (1) Добавляет промежуточный компонент multiparty.
  .use((req, res, next) => {
    console.log(req.files);  <-------  (2) Выводит информацию об отправленных файлах.
    res.end('Upload received\n');
  })
  .listen(3000);
```

Этот короткий пример добавляет промежуточный компонент multiparty **(1)**, а затем выводит полученные файлы **(2)**. Файлы будут загружены в каталог для временных файлов, поэтому вам придется воспользоваться модулем fs для удаления файлов, когда приложение закончит работать с ними.

Чтобы использовать этот пример, убедитесь в том, что у вас установлен модуль connect-multiparty[1]. Затем запустите сервер и отправьте ему файл командой `curl` с ключом `-F`:

```
curl -F file=@index.js http://localhost:3000
```

Имя файла размещается после символа `@` и снабжается префиксом — именем поля. Имя поля используется в файле `req.files`, чтобы вы могли отличить один загруженный файл от другого.

Примерный вывод приложения показан ниже. Как видите, значение `req.files.file.path` будет доступно для вашего приложения, и вы сможете переименовать файл на диске, передать данные для обработки, отправить их в сеть доставки контента или выполнить любые другие операции, нужные вашему приложению:

```
{ fieldName: 'file',
  originalFilename: 'index.js',
  path: '/var/folders/d0/_jqj3lf96g37s5wrf79v_g4c0000gn/T/60201-p4pohc.js',
  headers:
   { 'content-disposition': 'form-data; name="file"; filename="index.js"',
     'content-type': 'application/octet-stream' },
```

Возможно, вас заинтересует, как организовать сжатие ответов. Ниже рассказано о том, как промежуточный компонент сжатия поможет снизить нагрузку на канал и создать впечатление повышенной скорости отклика приложения.

[1] Для тестирования в этом примере использовалась версия 1.2.5.

B.1.4. Сжатие ответов

Вероятно, в предыдущем разделе вы обратили внимание на то, что парсеры могут распаковывать запросы, использующие gzip или deflate. В поставку Node входит базовый модуль сжатия данных, который называется zlib; он используется для реализации методов как сжатия, так и распаковки. Промежуточный компонент сжатия (*www.npmjs.com/package/compression*) может использоваться для сжатия исходящих ответов; это означает, что данные, отправляемые вашим сервером, могут сжиматься.

Сервис Google PageSpeed Insights рекомендует использовать сжатие gzip[1]; а если вы присмотритесь к запросам, выдаваемым вашим браузером, в средствах разработчика, вы увидите, что многие сайты возвращают сжатые ответы. Сжатие повышает нагрузку на процессор, но такие форматы, как обычный текст и HTML, хорошо сжимаются; сжатие может улучшить быстродействие вашего сайта и сократить нагрузку на канал.

DEFLATE ИЛИ GZIP?

Наличие двух вариантов сжатия может вызвать недоумение. Какой из двух вариантов лучше, и вообще зачем нужны два варианта? В соответствии со стандартами (RFC 1950 и RFC 2616), оба варианта используют один алгоритм, но различаются по способу обработки заголовка и контрольной суммы.

К сожалению, некоторые браузеры некорректно обрабатывают deflate, поэтому в общем случае лучше использовать gzip. При разборе желательно поддерживать оба варианта, но если вы сжимаете вывод своего сервера — используйте gzip для большей надежности.

Модуль сжатия обнаруживает принимаемые кодировки по полю заголовка `Accept-Encoding`. Если поле отсутствует, используется тождественная кодировка (это означает, что ответ не изменяется). Если же поле содержит `gzip` и (или) `deflate`, ответ сжимается.

Простой пример использования

Обычно компонент сжатия следует размещать высоко в стеке Connect, потому что он служит оберткой для методов `res.write()` и `res.end()`.

В следующем примере продемонстрировано сжатие контента:

```
const connect = require('connect');
const compression = require('compression');
```

[1] См. https://developers.google.com/speed/docs/insights/EnableCompression.

```
connect()
  .use(compression({ threshold: 0 }))
  .use((req, res) => {
    res.setHeader('Content-Type', 'text/plain');
    res.end('This response is compressed!\n');
  })
  .listen(3000);
```

Для запуска этого примера необходимо установить модуль compression из npm. Затем запустите сервер и попробуйте выдать запрос `curl`, в котором заголовку `Accept-Encoding` присваивается значение `gzip`:

```
$ curl http://localhost:3000 -i -H "Accept-Encoding: gzip"
```

Аргумент `-i` приказывает cURL вывести заголовки, чтобы вы могли убедиться в том, что `Content-Encoding` присвоено значение `gzip`. Вывод выглядит невразумительно, потому что сжатые данные не состоят из стандартных символов. Попробуйте обработать их gunzip с ключом `-i`, чтобы просмотреть вывод:

```
$ curl http://localhost:3000 -H "Accept-Encoding: gzip" | gunzip
```

Это мощный и относительно несложный в реализации механизм, но не стоит полагать, что вам всегда следует сжимать все, что отправляет ваш сервер. Для отключения сжатия можно воспользоваться нестандартной функцией фильтрации.

Использование нестандартной функции фильтрации

По умолчанию compression включает типы MIME `text/*`, `*/json` и `*/javascript` в стандартную функцию `filter`, чтобы избежать сжатия этих типов данных:

```
exports.filter = function(req, res){
  const type = res.getHeader('Content-Type') || '';
  return type.match(/json|text|javascript/);
};
```

Чтобы изменить это поведение, можно передать `filter` в объекте параметров, как показано в следующем фрагменте кода, который сжимает только обычный текст:

```
function filter(req) {
  const type = req.getHeader('Content-Type') || '';
  return 0 === type.indexOf('text/plain');
}
connect()
  .use(compression({ filter: filter }));
```

Задание уровней сжатия и расходования памяти

Привязки Node-модуля zlib поддерживают параметры быстродействия и сжатия, которые также могут передаваться функции `compression`.

В следующем примере кода переменной `level` присваивается значение 3, что соответствует невысокой степени сжатия при высоком быстродействии, 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 модулю Node zlib):

```
connect()
  .use(compression({ level: 3, memLevel: 8 }));
```

А теперь обратимся к промежуточным компонентам, обеспечивающим такие базовые потребности веб-приложений, как ведение журнала и сеансовые данные.

B.2. Реализация базовых функций веб-приложения

Connect старается реализовать и предоставить встроенные промежуточные компоненты для большинства потребностей стандартных веб-приложений, чтобы их не приходилось реализовывать снова и снова каждому разработчику. Такие функции веб-приложений, как ведение журнала, сеансы и виртуальный хостинг, предоставляются Connect во встроенном виде.

В этом разделе рассматриваются пять полезных промежуточных компонентов, которые с большой вероятностью понадобятся вам в приложениях:

- morgan — предоставляет гибкую систему протоколирования запросов;
- serve-favicon — обслуживает запросы /favicon.ico, не требуя никаких специальных мер с вашей стороны;
- method-override — прозрачная замена `req.method` для клиентов с ограниченной функциональностью;
- vhost — создание нескольких сайтов на одном сервере (виртуальный хостинг);
- express-session — управление сеансовыми данными.

До настоящего момента вы сами создавали нестандартные промежуточные компоненты ведения журнала, но разработчики, ответственные за сопровождение Connect, предоставили гибкое решение, которое называется morgan. Для начала рассмотрим его.

B.2.1. morgan: ведение журнала запросов

Модуль morgan (*www.npmjs.com/package/morgan*) — гибкий промежуточный компонент для протоколирования запросов с возможностью настройки формата

вывода. Он также поддерживает ключи для буферизации вывода, чтобы сократить количество операций записи на диск, и для назначения журнального потока, если данные вместо консоли должны выводиться в другой приемник (например, файл или сокет).

Простейший вариант использования

Чтобы использовать morgan в своем приложении, вызовите его как функцию для получения промежуточной функции, как показано в листинге B.8.

Листинг B.8. Использование модуля morgan для ведения журнала

```
const connect = require('connect');
const morgan = require('morgan');

connect()
  .use(morgan('combined'))  <-------  (1) Режим combined используется для каждого запроса.
  .use((req, res) => {
    res.setHeader('Content-Type', 'application/json');
    res.end('Logging\n');  <-------  (2) Отвечает на запрос сообщением.
  })
  .listen(3000);
```

Чтобы использовать этот пример, необходимо установить модуль morgan из npm[1]. Модуль добавляется в начало стека промежуточных компонентов **(1)**, после чего выводится простой текстовый ответ **(2)**. При использовании аргумента `combined` **(1)** это приложение Connect будет выводить данные в формате журналов Apache. Это гибкий формат, который может разбираться многими приложениями командной строки, что позволит вам обработать свои журналы аналитическими приложениями, генерирующими полезную статистику. При выдаче запросов от разных клиентов (таких, как `curl`, `wget` и браузер) вы увидите в журнале строку, идентифицирующую агента.

Формат журнала `combined` определяется следующим образом:

```
:remote-addr - :remote-user [:date[clf]] ":method :url
  HTTP/:http-version" :status :res[content-length] ":referrer" ":user-agent"
```

Каждый из фрагментов, предваряемый двоеточием, представляет собой *маркер* (token), который в журнальной записи заменяется реальными значениями из запроса HTTP. Например, в результате простого запроса `curl(1)` будет сгенерирована примерно такая строка журнала:

```
127.0.0.1 - - [Thu, 05 Feb 2015 04:27:07 GMT]
                      "GET / HTTP/1.1" 200 - "-"
                      "curl/7.37.1"
```

[1] Мы использовали версию 1.5.1.

Настройка форматов журнала

Вы также можете создавать собственные форматы журнала. Для этого нужно передать специальную строку маркеров. Например, следующий формат будет создавать записи вида `GET /users 15 ms`:

```
connect()
  .use(morgan(':method :url :response-time ms'))
  .use(hello)
  .listen(3000);
```

По умолчанию доступны следующие маркеры (обратите внимание, что в названиях заголовков регистр символов не важен):

- `:req[header]` `example: :req[Accept]`;
- `:res[header]` `example: :res[Content-Length]`;
- `:http-version`;
- `:response-time`;
- `:remote-addr`;
- `:date`;
- `:method`;
- `:url`;
- `:referrer`;
- `:user-agent`;
- `:status`.

Вы также можете определять нестандартные маркеры. Для этого достаточно указать название маркера и функцию обратного вызова для функции `connect.logger.token`. Например, для регистрации каждой строки запроса можно использовать следующий код:

```
var url = require('url');
morgan.token('query-string', function(req, res){
  return url.parse(req.url).query;
});
```

Кроме формата по умолчанию модуль morgan также включает другие предопределенные форматы (такие, как `short` и `tiny` для вывода сокращенной информации). Еще один предопределенный формат, `dev`, позволяет выводить краткие сведения для разработчиков веб-приложений. Этот формат может потребоваться в тех случаях, когда, например, нужно выяснить, находится ли пользователь на веб-сайте, и вам не хотелось бы возиться с подробностями запросов HTTP. 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. Подобная цветовая схема существенно облегчает разработку.

Чтобы задействовать предопределенный формат, просто передайте его название `logger()`:

```
connect()
  .use(morgan('dev'))
  .use(hello);
  .listen(3000);
```

Теперь, когда вы знаете, как форматировать данные, выводимые журналом, давайте поговорим о его параметрах.

Режимы вывода журнала: stream и immediate

Как уже упоминалось, с помощью параметров поведение модуля morgan можно изменить.

Один из этих параметров, `stream`, позволяет передать экземпляр Node-объекта `Stream`, который будет использоваться вместо `stdout` для вывода данных. С помощью этого параметра можно перенаправить вывод 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 `append` файл не будет усекаться при перезагрузке приложения:

```
const fs = require('fs');
const morgan = require('morgan');
const log = fs.createWriteStream('/var/log/myapp.log', { flags: 'a' })
connect()
  .use(morgan({ format: ':method :url', stream: log }))
  .use('/error', error)
  .use(hello)
  .listen(3000);
```

Другой полезный параметр, `immediate`, обеспечивает вывод строки журнала, если запрос получен в первый раз (вместо ожидания ответа). Этот параметр рекомендуется использовать в том случае, если сервер держит запросы открытыми на протяжении длительного времени и вы хотите узнать, когда начинается 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:

```
const app = connect()
  .use(connect.logger({ immediate: true }))
  .use('/error', error)
  .use(hello);
```

Вот и все, что касается ведения журнала! А теперь рассмотрим промежуточный компонент для предоставления значка сайта.

B.2.2. serve-favicon: значки адресной строки и закладки

Маленький *значок сайта* (favicon) отображается в адресной строке и на ярлычках вкладок браузера. Чтобы вывести такой значок, браузер запрашивает файл /favicon.ico. Обычно лучше предоставлять файлы значков как можно раньше, чтобы остальные части приложения попросту игнорировали их. Модуль serve-favicon (*www.npmjs.com/package/serve-favicon*) по умолчанию предоставляет значок Connect. Чтобы сменить значок, передайте соответствующие аргументы. Значок сайта показан на рис. B.2.

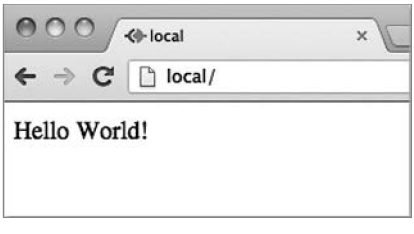


Рис. B.2. Значок сайта

Применение

Обычно промежуточный компонент serve-favicon размещается в самой верхней части стека, поэтому запросы значков сайта игнорируются всеми последующими компонентами ведения журнала. Значок кэшируется в памяти для ускорения последующих ответов.

В следующем примере кода показан компонент serve-favicon, отправляющий файл .ico путем передачи в качестве единственного аргумента пути к файлу:

```
const connect = require('connect');
const favicon = require('serve-favicon');
connect()
  .use(favicon(__dirname + '/favicon.ico'))
  .use((req, res) => {
    res.end('Hello World!\n');
  });
```

Для тестирования вам понадобится файл с именем favicon.ico. 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 промежуточный компонент — method-override. Он позволяет имитировать метод запроса HTTP в том случае, если функциональность клиента ограничена.

B.2.3. method-override — имитация методов HTTP

Иногда бывает полезно использовать команды HTTP, выходящие за рамки стандартных методов GET и POST. Представьте, что вы создаете систему ведения блогов и хотите предоставить пользователям возможность создания, обновления и удаления статей. Вместо GET и POST более естественно использовать запись `DELETE /`*статья*. К сожалению, не все браузеры поддерживают метод DELETE.

Стандартное обходное решение — разрешить серверу получить информацию о том, какой метод HTTP следует использовать, по параметрам запроса, значениям формы, а иногда даже заголовкам HTTP. Один из способов основан на включении тега `<input type=hidden>` со значением, соответствующим нужному имени метода. Сервер проверяет значение и имитирует метод, соответствующий запросу.

Этот прием поддерживается большинством веб-фреймворков; в Connect для его реализации рекомендуется использовать модуль method-override (*www.npmjs.com/package/method-override*).

Применение

По умолчанию полю HTML присваивается имя `_method`, но вы можете передать собственное значение методу `methodOverride`, как показано в следующем примере:

```
connect()
const connect = require('connect');
const methodOverride = require('method-override');
connect()
  .use(methodOverride('__method__'))
  .listen(3000)
```

Чтобы показать, как реализуется `methodOverride()`, посмотрим, как создать крошечное приложение для обновления информации пользователей. Приложение состоит из одной формы, которая отвечает одним сообщением об успехе, когда форма отправляется браузером и обрабатывается сервером (рис. B.3).

Приложение обновляет данные о пользователе с помощью двух разных промежуточных компонентов. В функции `update` функция `next()` вызывается в том случае, если метод запроса отличен от PUT. Как упоминалось ранее, большинство браузеров не поддерживают атрибут `method="put"` формы, поэтому приложение из листинга B.9 работает некорректно.

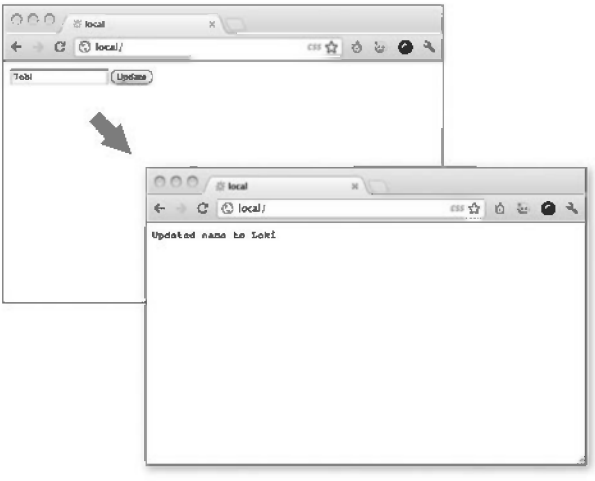


Рис. В.3. Использование method-override для имитации запроса PUT, чтобы обновить форму

Листинг В.9. Некорректно работающее приложение, обновляющее сведения о пользователе

```
const connect = require('connect');
const morgan = require('morgan');
const bodyParser = require('body-parser');

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);
}

connect()
  .use(morgan('combined'))
  .use(bodyParser.urlencoded({ extended: false }))
  .use(edit)
  .use(update)
     .listen(3000);
```

(1) Форма отправляет запрос PUT вместо GET или POST.

(2) Проверяет, был ли запрос отправлен методом PUT.

Форма из этого примера отправляет серверу запрос PUT **(1)**. Форма должна отправить данные функции `update`, но только в том случае, если при отправке использован

метод PUT **(2)**. Опробуйте этот пример с разными браузерами и клиентами HTTP; для отправки запроса PUT из `curl` используется ключ `-X`.

Для решения проблемы с поддержкой методов браузером добавляется модуль method-override. В листинге B.10 на форму добавляется дополнительный элемент с именем `_method`, а метод `methodOverride()` добавляется после метода `bodyParser()`, потому что он обращается к `req.body` для получения доступа к данным формы.

Листинг B.10. Использование method-overrider для поддержки метода HTTP PUT

```
const connect = require('connect');
const morgan = require('morgan');
const bodyParser = require('body-parser');
const methodOverride = require('method-override');

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}');;
}

connect()
  .use(morgan('dev'))
  .use(bodyParser.urlencoded({ extended: false }))
  .use(methodOverride('_method'))
  .use(edit)
  .use(update)
  .listen(3000);
```

Отправляет подсказку с методом HTTP в виде переменной формы _method.

Использует промежуточный компонент methodOverride для отслеживания переменной формы.

Запустив этот пример, вы увидите, что теперь у вас появилась возможность отправлять запросы PUT практически из любого браузера.

Получение доступа к исходному значению req.method

Метод `methodOverride()` заменяет исходное значение свойства `req.method`, однако Connect копирует исходный метод, и вы всегда можете получить доступ к нему через свойство `req.originalMethod`. Таким образом, предыдущая форма могла бы выводить значения так:

```
console.log(req.method);
  // "PUT"
console.log(req.originalMethod);
  // "POST"
```

Чтобы избежать включения лишних переменных формы, также поддерживаются заголовки HTTP. Разные фирмы-разработчики используют разные заголовки, так что вы можете создавать серверы, которые поддерживают несколько разных имен полей заголовков. Это будет полезно в том случае, если вы хотите поддерживать клиентские инструменты и библиотеки, рассчитанные на конкретный заголовок. В следующем примере поддерживаются три имени поля заголовка:

```
app.use(methodOverride('X-HTTP-Method'))          <-- Microsoft
app.use(methodOverride('X-HTTP-Method-Override'))  <-- Google/GData
app.use(methodOverride('X-Method-Override'))       <-- IBM
```

Маршрутизация на основании заголовков — достаточно стандартная задача. Хорошим примером служит поддержка виртуальных хостов. Возможно, вы видели серверы Apache, которые используют эту возможность при размещении нескольких сайтов на меньшем количестве IP-адресов. Apache и Nginx могут определить, к какому именно сайту относится обращение, на основании заголовка Host.

Connect тоже предоставляет такую возможность, причем реализуется она проще, чем можно было бы предположить. В следующем разделе рассматриваются виртуальные хосты и модуль vhost.

B.2.4. vhost: виртуальный хостинг

Модуль vhost (виртуальный хост) (*www.npmjs.com/package/vhost*) представляет собой простое облегченное средство маршрутизации запросов на основании 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

Как и для большинства промежуточных компонентов, 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 соответствующим именем хоста. Поскольку все приложения Connect являются субклассами класса `http.Server`, экземпляр приложения также подойдет:

```
const connect = require('connect');
const server = connect();
const vhost = require('vhost');
const app = require('./sites/expressjs.dev');
server.use(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:

```
const http = require('http')
module.exports = http.createServer((req, res) => {
  res.end('hello from expressjs.com\n');
});
```

Использование нескольких экземпляров vhost()

Подобно другим промежуточным компонентам, компонент `vhost` можно использовать многократно в приложении для связывания нескольких хостов с соответствующими приложениями:

```
const app = require('./sites/expressjs.dev');
server.use(vhost('expressjs.dev', app));
const app = require('./sites/learnboost.dev');
server.use(vhost('learnboost.dev', app));
```

Вместо ручной настройки промежуточного компонента `vhost` можно сгенерировать список хостов в файловой системе. Соответствующая методика продемонстрирована в следующем примере, в котором метод `fs.readdirSync()` возвращает массив записей каталога:

```
const connect = require('connect')
const fs = require('fs');
cons app = connect()
const sites = fs.readdirSync('source/sites');
sites.forEach((site) => {
  console.log('  ... %s', site);
  app.use(vhost(site, require('./sites/' + site)));
});
app.listen(3000);
```

Главное преимущество компонента `vhost` перед реверсным прокси-сервером — его простота. Он позволяет управлять всеми приложениями как одним целым. Подобная методика идеально подходит для обслуживания нескольких небольших сайтов или сайтов, содержащих преимущественно статический контент. Впрочем, есть и недостаток: если на одном сайте произойдет сбой, 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 промежуточных компонентов, представляемых Connect. Это компонент управления сеансами, который называется express-session.

B.2.5. express-session: управление сеансами

Механизм реализации сеансов в веб-приложениях зависит от требований. Например, одним из важных решений становится подсистема хранения данных. Одни

приложения пользуются такими высокопроизводительными базами данных, как Redis; для других предпочтительна простота, и они используют ту же базу данных, что и главное приложение. Модуль express-session (*www.npmjs.com/package/express-session*) предоставляет API, который может расширяться для разных баз данных. Это мощный и легко расширяемый модуль, поэтому у него существует множество расширений, поддерживаемых сообществом. В этом разделе вы узнаете, как пользоваться версией с хранением данных в памяти и Redis.

А для начала мы поговорим о настройке промежуточных компонентов и доступных параметрах.

Применение

В листинге B.11 представлен пример небольшого приложения, ведущего подсчет просмотров страницы. Информация хранится в сеансовых данных пользователя. По умолчанию cookie присваивается имя `connect.sid` со значением `httpOnly` (это означает, что клиентские сценарии не могут получить доступ к этому значению). Данные самого сеанса хранятся в памяти на сервере. В листинге B.11 продемонстрировано базовое использование express-session в Connect[1].

Листинг B.11. Использование сеансов в Connect

```
const connect = require('connect');
const session = require('express-session');

connect()
  .use(session({
    secret: 'example secret',  <-- (1) Базовые параметры, необходимые для использования сеансов.
    resave: false,
      saveUninitialized: true
    }))
  .use((req, res) => {
    req.session.views = req.session.views || 0;
    req.session.views++;  <-- Создает сеансовую переменную views и увеличивает ее.
    res.end('Views:' + req.session.views);  <-- Возвращает значение браузеру.
  })
  .listen(3000);
```

Этот короткий пример создает сеансы, а затем оперирует с сеансовой переменной с именем `views`. Сначала сеансовый промежуточный компонент инициализируется с необходимыми параметрами: `secret`, `resave` и `saveUninitialized` **(1)**. Обязательный параметр `secret` определяет, подписан ли объект cookie, используемый для идентификации сеанса. Параметр `resave` обеспечивает принудительное сохранение сеанса при каждом запросе, даже если он не изменился. Это необходимо для некоторых подсистем хранения сеансовых данных, поэтому перед включением лучше уточнить

[1] Для тестирования использовалась версия 1.10.2.

нужное значение по документации. Последний параметр, `saveUninitialized`, инициирует создание сеанса даже при отсутствии значений для сохранения. Сбросьте этот флаг, если хотите выполнять правило, требующее подтверждения пользователя перед сохранением cookie.

Установка времени истечения сеанса

Предположим, что нужно установить время истечения сеанса в 24 часа, разрешить передачу cookie сеанса только при использовании протокола HTTPS и задать имя cookie. Для управления сроком жизни сеанса можно задать свойства `expires` или `maxAge` объекта cookie:

```
const hour = 3600000
req.session.cookie.expires = new Date(Date.now() + hour * 24);
req.session.cookie.maxAge = hour * 24;
```

При использовании Connect часто устанавливается значение переменной `maxAge`, определяющей количество миллисекунд, прошедших от заданного времени. С помощью этой переменной выражение будущих дат зачастую записывается в интуитивно более понятном виде:

```
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

API управления данными в express-session очень прост. Основной принцип заключается в том, что любые свойства, присвоенные объекту `req.session`, сохраняются после выполнения запроса. Затем эти свойства загружаются при поступлении последующих запросов от того же пользователя (браузера). Например, для сохранения информации о корзине покупок достаточно задать объект свойству `cart`:

```
req.session.cart = { items: [1,2,3] };
```

Когда вы обращаетесь к свойству `req.session.cart` в последующих запросах, вам будет доступен массив `.items`. А поскольку это обычный объект JavaScript, в последующих запросах можно вызывать методы вложенных объектов, причем он сохраняется именно так, как ожидается:

```
req.session.cart.items.push(4);
```

Следует иметь в виду, что, если этот объект сеанса между запросами сериализуется в формате JSON, на объект `req.session` накладываются те же ограничения, что и на формат JSON: циклические свойства недопустимы, не могут применяться объекты `function`, объекты `Date` сериализуются некорректно и т. п. Учитывайте эти ограничения при использовании объекта сеанса.

В Connect данные сеанса сохраняются автоматически, но во внутренней реализации 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()`. При построении приложений на базе Express эти методы используются для аутентификации.

А теперь рассмотрим операции с сеансовыми cookie.

Манипулирование сеансовыми cookie

Connect позволяет задать глобальные параметры cookie для сеансов, но можно также манипулировать конкретными cookie 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 сеансовых cookie путем записи каждого свойства в отдельные теги `<p>` в HTML-разметке 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 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 с методом доступа `.maxAge`, который позволяет получить и установить значение в миллисекундах относительно текущего времени. Следующий код завершает активный сеанс через 5 секунд:

```
req.session.cookie.maxAge = 5000;
```

Остальные свойства (такие, как `domain`, `path` и `secure`) ограничивают *область видимости* (scope) cookie, распространяя ее только на домен, путь или защищенные соединения, в то время как `httpOnly` предотвращает доступ к данным cookie 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. Давайте разберемся, как подключать альтернативные хранилища данных.

Хранилища данных сеанса

В предыдущих примерах мы использовали 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 хранение данных сеанса; в противном случае вы будете терять сеансовые данные при перезапуске сервера.

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 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, для работы с которой используется модуль connect-redis (*https://www.npmjs.com/package/connect-redis*). База данных Redis представляет собой хорошее резервное хранилище, поскольку поддерживает политику истечения срока действия ключей, обеспечивает высокую производительность и проста в установке.

Чтобы убедиться в том, что база данных Redis установлена в вашей системе, 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 --save connect-redis`[1]. Модуль connect-redis экспортирует функцию, которая должна быть передана переменной connect.

Листинг B.12. Использование Redis для хранения сеансовых данных

```
const connect = require('connect');
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const favicon = require('serve-favicon');
```

Передает RedisStore экземпляр express-session.

[1] При написании книги использовалась версия 2.2.0.

```
const options = {
  host: 'localhost'
};

connect()
  .use(favicon(__dirname + '/favicon.ico'))
  .use(session({
    store: new RedisStore(options),
    secret: 'keyboard cat',
    resave: false,
    saveUninitialized: true
  }))
  .use((req, res) => {
    req.session.views = req.session.views || 0;
    req.session.views++;
    res.end('Views: ' + req.session.views);
  })
  .listen(3000);
```

Настраивает сеанс рекомендуемыми значениями по умолчанию и RedisStore.

Изменяет сеансовые значения обычным способом.

В этом примере создается хранилище сеансовой информации, использующее Redis. Передача connect-redis ссылки на переменную `connect` позволит выполнить наследование из прототипа `connect.session.Store.prototype`. Это важно, поскольку в Node один процесс может одновременно использовать несколько версий модуля. Передавая конкретную версию Connect, можно гарантировать, что connect-redis задействует правильную копию.

Экземпляр класса `RedisStore` передается `session()` как значение `store`, а конструктору `RedisStore` могут передаваться произвольные параметры, которые вы собираетесь использовать, — например, префикс для сеансов. После того как оба шага будут выполнены, вы сможете обращаться к сеансовым переменным так же, как к `MemoryStore`. В этом примере стоит обратить внимание на одну подробность: мы включаем промежуточный компонент значка сайта, чтобы предотвратить повторное увеличение сеансовой переменной; в противном случае значение `views` будет увеличиваться на 2 при каждом запросе, так как браузер запрашивает страницу и /favicon.ico.

Раздел получился довольно длинным, но на этом описание промежуточных компонентов для основных функций веб-приложений завершается. В следующем разделе рассматриваются встроенные промежуточные компоненты, обеспечивающие безопасность веб-приложений. Эта область чрезвычайно важна для приложений, которые должны защищать свои данные.

B.3. Безопасность веб-приложений

Как уже неоднократно отмечалось, API-интерфейс ядра Node преднамеренно реализован на низком уровне. Это означает, что он не поддерживает встроенные механизмы защиты или оптимальные методики в том, что касается создания

веб-приложений. К счастью, эти средства безопасности реализуются промежуточными компонентами Connect.

В этом разделе рассматриваются три модуля, относящихся к безопасности, которые можно установить из npm:

- *basic-auth* — поддерживает базовую HTTP-аутентификацию защищенных данных;
- *csurf* — обеспечивает защиту против атак фальсификации межсайтовых запросов (Cross-Site Request Forgery, CSRF);
- *errorHandler* — помощь в отладке на этапе разработки.

Начнем с компонента basic-auth, реализующего базовую HTTP-аутентификацию.

B.3.1. basic-auth: базовая HTTP-аутентификация

В главе 4 мы создали примитивный промежуточный компонент, реализующий базовую аутентификацию. Оказывается, существуют несколько модулей Connect, предназначенных для выполнения той же операции. Как упоминалось ранее, базовая аутентификация — это простейшая HTTP-аутентификация, и применять ее следует осторожно, поскольку учетные данные пользователя могут быть легко перехвачены (если при передаче данных не используется 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 модуль basic-auth, при первой попытке подключения пользователя к приложению веб-браузер запросит учетные данные, как показано на рис. B.4.

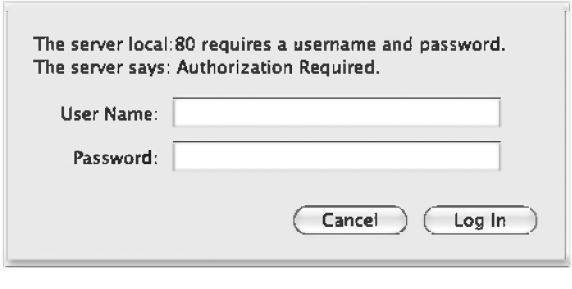


Рис. B.4. Запрос учетных данных при базовой аутентификации

Применение

Модуль basic-auth (*www.npmjs.com/package/basic-auth*) позволяет получить учетные данные из поля заголовка HTTP `Authorization`. В листинге B.13 показано, как использовать его с вашей собственной функцией проверки пароля.

Листинг В.13. Использование модуля basic-auth

```
const auth = require('basic-auth');
const connect = require('connect');

function passwordValid(credentials) {
  return credentials
    && credentials.name === 'tj'
    && credentials.pass === 'tobi';
}

connect()
  .use((req, res, next) => {
    const credentials = auth(req);

    if (passwordValid(credentials)) {
      next();
    } else {
      res.writeHead(401, {
        'WWW-Authenticate': 'Basic realm="example"'
      });
      res.end();
    }
  })
  .use((req, res) => {
    res.end('This is the secret area\n');
  })
  .listen(3000);
```

Проверяет пароль для жестко запрограммированного имени пользователя.

Получает разобранные учетные данные.

Возвращает заголовок WWW-Authenticate, если пароль указан неправильно.

В противном случае next() передает управление в «защищенную область».

Модуль basic-auth обеспечивает только часть процесса аутентификации, связанную с разбором поля заголовка Authorization. Вы должны самостоятельно проверить пароль, обратившись с вызовом к промежуточному компоненту; модуль basic-auth отправляет нужные заголовки в том случае, если попытка аутентификации завершилась неудачей. В этом примере в случае удачной аутентификации вызывается `next()`, чтобы управление было передано в защищенные части приложения.

Пример с cURL

Теперь попробуйте выдать запрос HTTP к серверу при помощи `curl`; вы увидите, что пользователь не авторизован:

```
$ curl http://localhost:3000 -i
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Authorization Required"
Connection: keep-alive
Transfer-Encoding: chunked
Unauthorized
```

При выдаче того же запроса с учетными данными базовой аутентификации HTTP (обратите внимание на начало URL) доступ будет успешно получен:

```
$ curl --user tj:tobi http://localhost:3000 -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 модуль csurf, предназначенный для защиты от атак фальсификации межсайтовых запросов.

B.3.2. csurf: защита от атак CSRF

При атаке фальсификации межсайтовых запросов (Cross-Site Request Forgery, CSRF) фактически эксплуатируется доверие к сайту со стороны браузера. В ходе такой атаки аутентифицированный пользователь вашего приложения посещает другой сайт, созданный или взломанный организатором атаки. После этого от имени пользователя выполняются подложные запросы, а он даже не подозревает об этом.

Эту атаку лучше пояснить на примере. Предположим, что в вашем приложении запрос `DELETE /account` вызывает удаление учетной записи пользователя (только во время подключения пользователя к серверу). Также предположим, что пользователь посетил форум, уязвимый для CSRF-атаки. Организатор атаки может передать сценарий, который генерирует запрос `DELETE /account` и удаляет учетную запись пользователя. Конечно, такая ситуация крайне нежелательна; избежать ее можно с помощью модуля csurf.

Модуль csurf (*https://www.npmjs.com/package/csurf*) 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 проверить маркер при подписании. Этот процесс повторяется для каждого взаимодействия.

Применение

Чтобы убедиться в том, что модуль csurf может получить доступ к `req.body._csrf` (скрытое значение вводимых данных) и `req.session._csrf`, следует поместить csurf после модулей body-parser и express-session, как показано в листинге B.14[1]:

[1] При тестировании примера использовалась версия 1.6.6.

Листинг В.14. Защита от атак CSRF

```
const bodyParser = require('body-parser');
const connect = require('connect');
const csurf = require('csurf');
const session = require('express-session');
const sesionOptions = {
  resave: false,
  saveUninitialized: false,
  secret: '1234'
};

connect()
  .use(bodyParser.urlencoded({ extended: false }))
  .use(session(sesionOptions))
  .use(csurf())
  .use((req, res, next) => {
    if ('/' != req.url) return next();

    const token = req.csrfToken();
    const html = `
      <form method="post" action="/save">
        <input type="text" name="_csrf" value="${token}">
        <button type="submit">Submit</button>
      </form>`;

    res.setHeader('Content-Type', 'text/html');
    res.end(html);
  })
  .use((req, res) => {
    const html = `
      <p>Body: ${req.body._csrf}</p>
      <p>Session secret: ${req.session.csrfSecret}</p>
    `;
    res.end(html);
  })
  .use((err, req, res, next) => {
    console.error(err);
    res.end('Did you get the csrf token wrong?');
  })
  .listen(3000);
```

Загружает промежуточный компонент csurf после парсера и обработчика сеансов.

Отображает форму для маршрута /.

Получает текущий маркер CSRF при помощи этого метода, добавленного csurf.

Эта функция выполняется после запроса POST с правильным маркером.

Обработчик ошибок для неправильного маркера.

Чтобы использовать csurf, необходимо сначала загрузить промежуточные компоненты body-parser и session. Затем этот пример отображает форму, которая включает в себя текстовое поле для маркера CSRF. С этим маркером все запросы с определенным типом метода будут проверяться на основании секретной строки в сеансе. Для получения текущего маркера используется `req.csrfToken` — метод, добавленный csurf. Запросы с недействительными маркерами будут автоматически помечаться csurf, поэтому мы включили обработчик для правильного маркера и обработчик ошибки. В этом примере используется текстовое поле, чтобы вы видели, что происходит при попытке изменения маркера.

Пример показывает, что csurf автоматически активизируется для определенных типов запросов. Эти типы определяются параметром `ignoreMethods`, который

может передаваться csurf. По умолчанию HTTP-методы GET, HEAD и OPTIONS игнорируются, но при необходимости можно добавить и другие.

Другой аспект веб-разработки — обеспечение доступа к подробным журналам и детализированным отчетам об ошибках в среде разработки и в рабочей среде. Рассмотрим модуль errorhandler, предназначенный для решения этих задач.

B.3.3. errorhandler: — обработка ошибок при разработке

Модуль errorhandler (*www.npmjs.com/package/errorhandler*) идеально подходит для разработки, предлагая подробные ответы с информацией об ошибках в форматах HTML, JSON и простого текста на базе поля заголовка Accept. Он предназначен только для использования в ходе разработки и не должен являться частью рабочей конфигурации.

Применение

Обычно этот компонент должен располагаться последним, поскольку предназначен для перехвата всех ошибок:

```
connect()
  .use((req, res, next) => {
    setTimeout(function () {
       next(new Error('something broke!'));
     }, 500);
  })
  .use(errorhandler());
```

Получение HTML-ответа об ошибке

Если просмотреть в браузере любую страницу, для которой был выполнен показанный здесь код, вы увидите Connect-страницу ошибки, подобную показанной на рис. B.5. На ней выводится сообщение об ошибке, состояние ответа и полная трассировка стека.

Получение ответа об ошибке в формате простого текста

Предположим, что мы тестируем API, построенный на базе Connect. Ответы, генерируемые в виде больших фрагментов HTML-разметки, далеки от идеала, поэтому по умолчанию `errorHandler()` отвечает в формате `text/plain`, который хорошо подходит для HTTP-клиентов командной строки (таких, как `curl(1)`). Следующий вывод демонстрирует эту концепцию:

```
$ curl localhost:3000 -H "Accept: text/plain"
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)
```

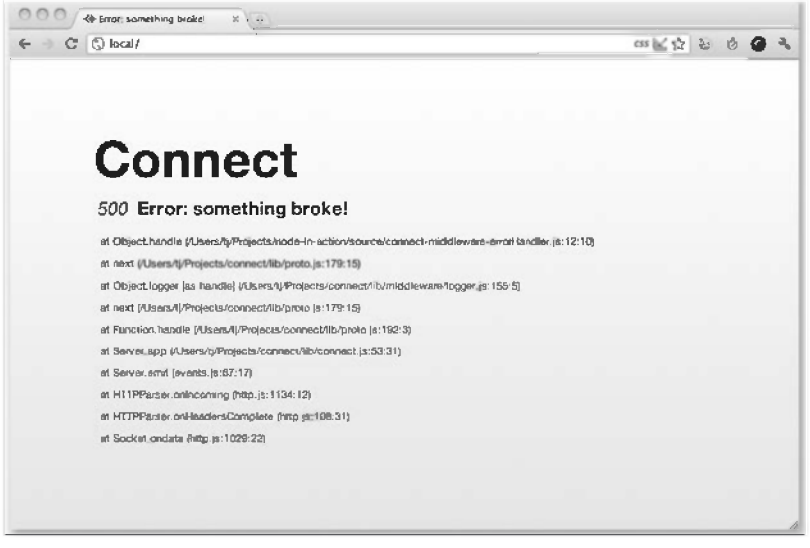


Рис. B.5. Разметка HTML для errorhandler в окне браузера

```
    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)
```

Получение JSON-ответа об ошибке

Если вы отправляете HTTP-запрос, который включает в себя заголовок HTTP `Accept: application/json`, то получите следующий ответ в формате JSON:

```
$ curl http://localhost:3000 -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 ответе JSON, чтобы упростить его чтение на странице, но, когда Connect отправляет ответ JSON, он уплотняется с помощью функции `JSON.stringify()`.

Чувствуете ли вы теперь себя профессионалом в вопросах безопасности Connect? Вероятно, нет, тем не менее вы знаете уже достаточно, чтобы защитить свои приложения хотя бы на начальном уровне. А теперь мы рассмотрим функцию, используемую во всех веб-приложениях: предоставление статических файлов.

B.4. Предоставление статических файлов

Предоставление статических файлов — еще одна функция, которая не реализована в ядре Node, но используется во многих веб-приложениях. К счастью, с некоторыми простыми модулями нужная функциональность реализована в Connect.

В этом разделе мы рассмотрим еще два официально поддерживаемых модуля Connect — на этот раз предоставляющих файлы из файловой системы. Обычно эти функции предоставляются серверами HTTP (такими, как Apache и Nginx), но при небольших усилиях вы можете добавить их в свои проекты Connect:

- *serve-static* — предоставляет файлы из файловой системы с заданным корневым каталогом;
- *serve-index* — предоставляет аккуратно отформатированное содержимое каталога при запросе каталога.

Сначала давайте рассмотрим, каким образом можно предоставлять статические файлы с помощью единственной строки кода, в которой используется модуль serve-static.

B.4.1. serve-static — автоматическое предоставление статических файлов браузеру

Модуль serve-static (*www.npmjs.com/package/serve-static*) реализует высокопроизводительный, гибкий и многофункциональный статический файловый сервер с поддержкой механизмов кэширования HTTP, запросов `Range` и т. п. Модуль также обеспечивает проверку безопасности для опасных путей, по умолчанию запрещает доступ к скрытым файлам (начинающимся с `).` и отклоняет вредоносные нулевые байты. В сущности, serve-static является очень надежным компонентом предоставления статических файлов, совместимым с различными HTTP-клиентами.

Применение

Предположим, что приложение следует типичному сценарию обслуживания статических ресурсов, находящихся в каталоге ./public. Задача решается с помощью единственной строки кода:

```
app.use(connect.static('public'));
```

В этой конфигурации модуль serve-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 происходит пересылка данных. Если запрошенный путь не представляет файл, активизируется функция обратного вызова `next()`, которая реализует переход к следующему (если таковой имеется) промежуточному компоненту.

Чтобы протестировать приложение, 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 предоставляются так, как и следовало ожидать. 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 модуля serve-static с монтированием

Иногда название приложения предваряется путем вида /public, /assets, /static и т. д. Реализованная в Connect концепция монтирования упрощает представление статических файлов, находящихся в разных папках. Для этого просто смонтируйте приложение в нужном месте. Как упоминалось в главе 5, промежуточные компоненты «не знают», где они смонтированы, поскольку префикс пути удаляется.

Например, запрос `GET /app/files/js/jquery.js` с модулем serve-static, смонтированным в точке /app/files, передается промежуточному компоненту в виде `GET /js/jquery`. Такое решение хорошо работает с функциональностью префиксов, поскольку точка /app/files перестает быть частью процесса разрешения файла:

```
app.use('/app/files', connect.static('public'));
```

Исходный запрос `GET /foo.js` в данном случае не сработает, поскольку промежуточный компонент не вызывается до тех пор, пока присутствует точка монтирования,

зато с префиксной версией запроса `GET /app/files/foo.js` файл будет передан успешно:

```
$ curl http://localhost/foo.js
Cannot get /foo.js
$ curl http://localhost/app/files/foo.js
console.log('tobi');
```

Абсолютный и относительный пути

Имейте в виду, что путь, переданный модулю serve-static, задается относительно текущего рабочего каталога. Это означает, что передача значения `"public"` в качестве пути разрешается, по сути, в виде `process.cwd() + "public"`.

Тем не менее при задании базового каталога иногда требуется использовать абсолютный путь. Для этого служит переменная `__dirname`, которая используется следующим образом:

```
app.use('/app/files', connect.static(__dirname + '/public'));
```

Предоставление файла index.html при запросе каталога

Еще одна полезная возможность модуля serve-static — возможность предоставления файлов index.html. Если запрашивается каталог, в котором находится файл index.html, будет предоставлен этот файл.

Возможность предоставления статических файлов полезна для ресурсов веб-приложений (CSS, JavaScript, графика и т. д.). Но что, если вы хотите разрешить пользователям загружать списки произвольных файлов из списка каталогов? На помощь приходит модуль serve-index.

B.4.2. serve-index: генерирование списков содержимого каталогов

Компактный модуль serve-index (*www.npmjs.com/package/serve-index*) предоставляет пользователям возможность просмотра удаленных файлов. На рис. B.6 изображен интерфейс этого компонента с полем для ввода критерия поиска, значками файлов и активных цепочечных ссылок (breadcrumbs).

Применение

Этот компонент спроектирован для работы с модулем serve-static, который предоставляет файлы; serve-index просто предоставляет списки содержимого. Конфигурация сводится к следующему фрагменту, в котором запрос `GET /` предоставляет каталог ./public:

```
const connect = require('connect');
const serveStatic = require('serve-static');
```

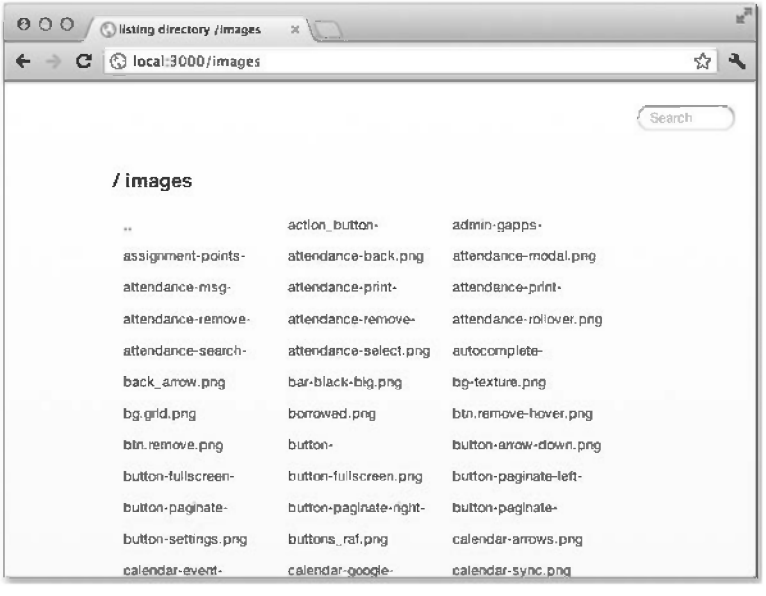


Рис. B.6. Предоставление списка содержимого каталога с использованием промежуточного компонента Connect directory()

```
const serveIndex = require('serve-index');

connect()
  .use(serveIndex('public'))
  .use(serveStatic('public'))
  .listen(3000);
```

Использование directory() с монтированием

При монтировании можно задать префикс, позволяющий направить оба компонента, serve-static и serve-index, по любому пути на ваше усмотрение — например, 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 предоставления скрытых файлов:

```
connect()
  .use('/files', serveIndex('public', { icons: true, hidden: true }))
  .use('/files', serveStatic('public', { hidden: true }))
  .listen(3000);
```

Теперь вы можете легко перемещаться по файлам и папкам.

Глоссарий

Глава 1

JSON (JavaScript Object Notation) — облегченный формат передачи данных, удобный для чтения и записи и основанный на подмножестве JavaScript.

libuv — многоплатформенная библиотека асинхронного ввода/вывода; используется в Node, а также в других библиотеках и языках (например, Julia).

npm — менеджер пакетов Node, позволяет устанавливать пакеты Node из большого центрального репозитория и управления зависимостями в проектах Node.

Promise — объект `Promise`, используется в стандартизированном API ECMAScript 2015 для представления значений, которые могут быть доступны прямо сейчас, в будущем или никогда.

REPL (read-eval-print loop) — интерфейс командной строки, позволяющий выполнить код и просмотреть результаты.

Абстрактный интерфейс — программное описание API, не включающее реализацию. Хороший пример абстрактного интерфейса в Node.js — API Streams.

Асинхронный — код, который не обязательно выполняется в порядке следования. В Node.js этот термин используется для обозначения API, получающих функции обратного вызова, которые будут выполнены в некоторый момент будущего. Например, `fs.readFile` получает функцию обратного вызова, которая получает содержимое файла после завершения его чтения.

Базовые модули — библиотеки, встроенные в Node.

Деструктуризация — в ECMAScript 2015 была введена концепция деструктуризации, которая позволяет разбивать объекты и массивы на переменные и константы. Например, `const { name } = { name:'Alex' }` создает константу с именем `name` и значением `alex`.

Неблокирующий ввод/вывод — блокирующие операции ввода/вывода прерывают выполнение программы до завершения операции. В Node используется неблокирующий ввод/вывод, при котором чтение данных из сети или файлового ресурса не приостанавливает выполнение программы.

Оставшиеся параметры — синтаксис оставшихся параметров в ECMAScript 2015 позволяет представить неизвестное количество аргументов функции в виде массива. Чтобы задать имена двух аргументов, а остальные поместить в массив, используйте запись `function (a, b, ...rest)`. Синтаксис также может использоваться при деструктуризации для копирования объектов: `const newObject = { ...oldObject }`.

Семантическое управление версиями — схема определения совместимости библиотеки с тремя числами: основным, дополнительным и оперативным номерами версии. Версия записывается в виде 1.0.2 (основная: 1, дополнительная: 0, оперативная: 2). Приложение, зависящее от версии 1.0.2, должно быть совместимо с версией 1.1.1, но не с версией 2.0.0.

Событие — строка, обеспечивающая вызов функции. Эта функция называется *слушателем* события. Событие с заданным именем отправляется *генератором* (emitter). Базовым классом для создания генераторов в Node является класс `EventEmitter`.

Стандарт ECMAScript — спецификация языка сценариев ECMAScript была стандартизирована ассоциацией Ecma International. Существует несколько стандартов ECMAScript; книга ориентирована на стандарт ECMAScript 2015 (ECMAScript 6th Edition). Создатели реализации JavaScript используют стандарт ECMAScript, чтобы обеспечить совместимость их интерпретатора с кодом JavaScript, написанным для других реализаций.

Стрелочная функция — функция, записанная в сокращенном синтаксисе. При передаче функций в аргументах других функций вместо записи `function() {}` используется запись `() => {}`. Если функция получает только один аргумент, круглые скобки можно опустить.

Цикл событий — цикл событий Node, ожидает появления внешних событий и преобразует их в активизацию функций обратного вызова. В других системах используются аналогичные механизмы (диспетчеры сообщений, исполнительные циклы) для быстрой маршрутизации событий по соответствующим обработчикам событий.

Глава 2

CommonJS — спецификация модуля для определения значений, которые должны экспортироваться из текущего файла JavaScript. См. *модуль*.

package.json — файл, определяющий имя, автора, лицензию и зависимости проекта Node. У каждой программы Node и библиотеки, созданной вами, должен быть файл package.json.

Свойство — объекты JavaScript представляют собой коллекции ключей и значений; ключи и значения называются свойствами объекта.

Вложенный обратный вызов — обратный вызов внутри обратного вызова; при передаче функции обратного вызова другой функции иногда бывает нужно определить другую функцию обратного вызова внутри первой.

Глобальная область видимости — под областью видимости (scope) понимаются части программы, которые могут обращаться к значению. Таким образом, под глобальной областью видимости понимается значение, доступное в любой точке программы.

Замыкание — функции JavaScript получают доступ к переменным, определенным в их внешней области видимости. Если функция `B` определяется внутри функции `A`, функция `B` получит доступ ко всем значениям `A`.

Модуль — модули Node представляют собой отдельные файлы, содержащие код JavaScript. Значения (как правило, функции и константы) могут экспортироваться для использования в других файлах.

Программная логика (или логика управления потоком) — порядок выполнения команд. Так как среда Node является асинхронной, программная логика представляет интерес, так как в JavaScript существует много способов управления программной логикой, включая обратные вызовы, обещания, базовые примитивы циклов и итераторы. В Node под *программной логикой* понимается способ группировки серий асинхронных задач.

Система управления контентом (CMS) — веб-приложение для редактирования текста и графики, которые должны отображаться на общедоступном сайте.

Состояние — совокупность значений всех переменных в программе на заданный момент времени.

Трассировка стека — список программных инструкций, выполненных до момента возникновения ошибки.

Функция обратного вызова — функция, которая передается другой функции для вызова в будущем.

Глава 3

cURL — программа командной строки и программная библиотека для выдачи запросов HTTP. Часто используется как средство отладки для быстрой проверки реакции веб-серверов на запросы.

MIME (Multipurpose Internet Mail Extensions) — интернет-стандарт для включения нетекстовых данных в сообщения электронной почты и многосекционные

тела сообщений. Это позволяет почтовым клиентам отображать разметку HTML, изображения и текст из других кодировок, кроме ASCII.

ORM (object-relational mapping) — библиотека, связывающая структуры данных, удобные для программиста (например, объекты JavaScript), со структурами баз данных (например, таблицами и внешними ключами).

REST (Representational State Transfer), **REST-совместимые API** — веб-API без состояния, использующий набор заранее определенных операций в HTTP. Операции базируются на командах HTTP; самые распространенные команды — GET, POST, PUT и DELETE.

Клиентский пакет — предварительно обработанный код JavaScript из нескольких исходных файлов, который обычно подвергается минификации и сжатию, а затем отправляется клиентам.

Кодирование данных формы — при отправке веб-серверу запроса HTTP POST, включая простую отправку формы, содержимое формы кодируется в теле запроса. Самый распространенный формат, `application/x-www-form-urlencoded`, аналогичен кодированию URL-адресов: небезопасные ASCII-символы заменяются комбинациями со знаком процента.

Маршрут — фрагмент URL-адреса и команда HTTP, которые должны быть обработаны обработчиком маршрута.

Модель базы данных — модель данных, удобная для программиста, которая упрощает выполнение операций с таблицами баз данных или документами на «родном» языке базы данных.

Обработчик маршрута — определяемая пользователем функция обратного вызова, которая выполняется при передаче запроса HTTP к веб-приложению. Обработчик маршрута обычно генерирует контент (возможно, загружая его из базы данных) или вносит изменения в базу данных, а затем генерирует ответ по шаблону или в определенном формате (например, JSON).

Статический файл — файл, предоставляемый веб-сервером без дополнительной обработки. Обычно графика, файлы CSS и файлы с клиентским кодом JavaScript представляют собой статические файлы.

Шаблон — текстовый формат, который может включать в себя встроенные данные или код JavaScript и используется для генерирования разметки HTML.

Глава 4

webpack loader — выполняет преобразование или транспиляцию исходного кода.

Канал — передача выходных данных на вход другой программы. В UNIX каналы между процессами обозначаются символом вертикальной черты (|); в Node потоки данных соединяются посредством цепочечного вызова методов.

Карта исходного кода — файл, при помощи которого средства отладки браузера могут связать строку транспилированного файла с исходным кодом.

Плагин webpack — изменяет поведение самого процесса построения (вместо выходных файлов).

Поток — эффективный механизм ввода и (или) вывода, который может содержать текстовые или двоичные данные. Node поддерживает потоки для чтения, потоки для записи, а также другие типы потоков, и эти потоки можно связывать друг с другом при помощи каналов.

Система построения — набор инструментов и конфигурационных файлов для генерирования кода JavaScript, эффективно выполняемого в браузере.

Статический анализатор — программа, проверяющая правильность формата исходного файла. Статические анализаторы могут использоваться для соблюдения требований конкретного стиля программирования в проекте, для чего исходный код проверяется по набору статических правил.

Транспилятор — транспиляторы JavaScript преобразуют одну разновидность ECMAScript в другую. Чаще всего транспиляторы используются для преобразования современного кода ES2015 в старый код ECMAScript 5, который может работать в большем количестве браузеров. Другой пример — TypeScript — представляет собой надмножество JavaScript, транспилируемое в ES5 или ES2015.

Цепочечный вызов методов — вызов метода для возвращаемого значения ранее вызванного метода.

Глава 5

GET, параметры — параметры URL-адреса, следующие за вопросительным знаком и разделенные символом &.

HTTP, команда — метод HTTP (GET, POST, PUT, PATCH, DELETE), представляющий действие, которое должно быть выполнено с удаленным ресурсом.

MVC (Model-View-Controller) — паттерн проектирования для разбиения программной системы на компоненты: модель (model) управляет данными и логикой, представление (view) преобразует данные в пользовательский интерфейс, а контроллер (controller) преобразует взаимодействия в операции модели или представления.

Адаптер базы данных — некоторые библиотеки баз данных пишутся в обобщенном виде и могут расширяться специализированными адаптерами, реализующими функциональность для нужной базы данных.

Логическая изоляция — возможность простого изменения функции, класса или модуля в проекте или их повторного использования в другом проекте.

Полностековый фреймворк — фреймворк, включающий в себя средства для работы как с клиентским, так и с серверным кодом. Обычно это означает наличие библиотек для работы с запросами HTTP, маршрутизации запросов, создания моделей баз данных и взаимодействия с кодом, выполняемым в браузере.

Изоморфные приложения — приложения JavaScript, способные выполняться как на стороне клиента, так и на стороне сервера, с использованием одного кода.

Промежуточные компоненты — функции, которые могут вызываться последовательно для изменения запроса и ответа HTTP.

Реляционная база данных — структура базы данных, основанная на отношениях между хранимыми сущностями.

Одностраничное веб-приложение — приложение, которое предоставляется браузеру однократно и не требует перезагрузки всей страницы. Если приложению потребуется по какой-то причине изменить URL-адрес в браузере, при помощи HTML History API создается иллюзия изменения URL и загрузки новой страницы с сервера.

Веб-фреймворк — набор библиотек для разработки веб-приложения с возможностью расширения с использованием плагинов или промежуточных компонентов.

Глава 6

bcrypt — функция хеширования паролей. Функция связывает данные произвольного объема со строкой фиксированного размера, которая может безопасно храниться в базе данных (чтобы избежать хранения пароля в текстовом виде).

База данных Redis — база данных в памяти, которая также выполняет функции кэша и брокера сообщений. Может использоваться для хранения сеансовых данных и обработки push-сообщений в веб-приложениях.

Затравка — случайные данные, добавляемые к входным данным хеш-функции для усложнения словарных атак.

Объект ответа — объект, определяющий реакцию вашего сервера на запрос HTTP. Включает в себя тело ответа (обычно это веб-страница) и заголовки.

Однопоточность — выполняемая программа (процесс) может состоять из параллельно выполняемых программных потоков. В модели JavaScript используется один программный поток, но предусмотрена возможность сохранения переключения контекста и выполнения другого кода при возникновении событий. События в браузере представляют собой взаимодействия (например, нажатие кнопки пользователем); в Node они обычно относятся к вводу/выводу (например, сетевые операции или чтение данных с диска).

Препроцессор CSS — программа, преобразующая надмножество CSS в разметку CSS, которая может интерпретироваться браузером. Языки таблиц стилей Sass

и LESS включают в себя препроцессоры CSS; в этих языках добавляются такие возможности, как переменные, вложение и примеси.

Согласование контента — часть стандарта HTTP, относящаяся к предоставлению разных версий документа с одинаковым URI. Пользовательские агенты (браузеры) могут запрашивать альтернативные форматы данных, если они поддерживаются сервером.

Стороннее промежуточное ПО — промежуточные компоненты, не распространяемые авторами исходной веб-библиотеки или фреймворка.

Хеш Redis — связь строкового поля со значением, используемая для представления объектов в базе данных Redis.

Язык шаблонов — облегченный язык разметки, который преобразуется в HTML и расширяется возможностями, упрощающими внедрение значений из кода, перебор содержимого массивов и объектов.

Глава 7

XSS (межсайтовые сценарные) атаки — если веб-приложение получает входные данные от форм или параметров URL и эти значения отображаются в шаблонах, появляется возможность внедрения вредоносного кода. Значения необходимо сначала экранировать для предотвращения атак этого вида.

Значимые пробельные символы — в JavaScript фигурные скобки, символы «точка с запятой» и символы новой строки используются для разделения команд. Если потребуется создать новый лексический блок, используется функция или управляющая команда. В языках со значимыми пробельными символами (например, Pug) строки кода группируются по количеству пробелов, используемых для формирования отступа в каждой строке.

Лексическая область видимости — возможность обращения к переменной определяется ее областью видимости. В JavaScript добавление функции вводит новый уровень области видимости. Любые переменные, определенные в функции, становятся видимыми для всех функций, определяемых внутри этой функции.

Лямбда-секция — лямбда-функцией называется анонимная функция. В Hogan лямбда-секцией называется способ связывания функции с тегом в шаблоне.

Примесь — обычно так называется класс, содержащий методы для использования в других классах. В Sass примеси представляют собой группы объявлений CSS, которые могут повторно использоваться в разных местах; в Pug примеси предназначаются для определения шаблонных фрагментов, пригодных для повторного использования.

Частичный шаблон — небольшой шаблон, пригодный для повторного использования.

Глава 8

ACID — набор характеристик, которыми должна обладать база данных. Операции с такой базой данных должны быть атомарными (операция либо завершается успешно, либо терпит полную неудачу, и база данных остается в неизменном состоянии), согласованными (данные могут изменяться только разрешенным образом), изолированными (с гарантиями параллельного выполнения) и устойчивыми (внесенные изменения остаются даже в случае сбоя или перезагрузки системы).

BSON — двоичный формат, используемый MongoDB для представления объектов. Объект состоит из упорядоченного набора элементов; элементы состоят из имени поля, типа и значения. К множеству типов, поддерживаемых BSON, относятся строки, целые числа, даты и код JavaScript.

NoSQL — база данных, не основанная на табличных отношениях между сущностями (в отличие от реляционных баз данных).

Веб-работник — механизм, обеспечивающий возможность выполнения браузерного кода JavaScript в фоновых потоках.

Демон — программа, выполняемая в фоновом режиме (обычно запускается автоматически при загрузке системы).

Документо-ориентированная база данных — база данных, содержащая частично структурированные данные и не имеющая предопределенной схемы (иногда в формате JSON или XML). К этой категории относятся базы данных MongoDB и CouchDB.

Мемоизация — прием оптимизации, основанный на сохранении результата вызова функции, чтобы ее не приходилось вызывать заново.

Ненадежная абстракция — попытка сокрытия сложности, которая раскрывает слишком много подробностей реализации.

Первичный ключ — столбец базы данных, используемый для однозначной идентификации каждой строки.

Построитель запросов — программный интерфейс, более удобный для программистов по сравнению с написанием запросов SQL вручную.

Публикация-подписка — паттерн, реализующий возможность рассылки сообщений нескольким получателям.

Распределенная база данных — база данных, хранящаяся на нескольких компьютерах — возможно (хотя и не обязательно), расположенных в разных географических местах.

Реляционная алгебра — теоретическая основа для моделирования хранимых данных и запросов, которые могут применяться к этим данным.

Реплицированный набор — группа процессов MongoDB, обеспечивающих хранение одного набора данных.

Схема базы данных — формальное определение данных и отношений между элементами базы данных; по сути, архитектура базы данных.

Транзакция базы данных — одна или несколько операций с базами данных, объединенных в группу и выполняемых в соответствии со свойствами ACID.

Глава 9

BDD (разработка через реализацию поведения) — расширение TDD с другим стилем API, который уделяет основное внимание тому, где в процессе происходит тестирование, что должно или не должно тестироваться и какой объем тестирования должен проводиться за один заход. BDD также старается более наглядно продемонстрировать, почему не проходят тесты и как назначаются имена тестовых модулей.

TDD (разработка через тестирование) — написание тестов перед написанием тестирования кода.

typeof — оператор JavaScript, возвращающий строку для заданного объекта или значения.

Макеты — объекты или значения, которые выглядят как реальные аналоги, но обычно обладают поведением, достаточным только для выполнения тестов. Вместо того чтобы обращаться к реальным файлам или сетевым ресурсам в тестах (такие операции могут быть медленными или опасными при деструктивном поведении), макеты могут использоваться для безопасной имитации их поведения.

Модульное тестирование — методология изолированного тестирования отдельных частей модуля (например, функций или методов класса) и проверки результатов по тестовым утверждениям в малых тестовых сценариях (модулях).

Система исполнения тестов — программа, управляющая нагрузочным тестированием, их выполнением и сбором результатов для вывода (например, Mocha).

Тестовое утверждение — проверка результата выражения на соответствие ожиданиям. Таким выражением может быть простая логическая конструкция, условие равенства и вообще практически все, что угодно. В Node тестовые утверждения выдают исключения, если условие не выполняется. Система исполнения тестов может перехватывать и накапливать эти исключения для построения отчетов по тестированию.

Функциональное тестирование — тестирование фрагмента функциональности в системе. В контексте веб-разработки это подразумевает полностековое тестирование: браузер и сервер тестируются одновременно.

Глава 10

Amazon EC2 (Amazon Elastic Compute Cloud) — сервис виртуализации Amazon.

CDN (сеть доставки контента) — распределенные серверы, поставляющие статический контент.

Docker, образ — образ файловой системы, который используется Docker для создания контейнера.

Elastic Beanstalk — сервис Amazon для сценарного управления развертыванием на других сервисах Amazon (например, EC2).

SSH (Secure Shell) — интерфейс командной строки (или X11) для выполнения команд на удаленном компьютере с поддержкой шифрования. Может формировать рабочий процесс веб-разработчика при исходной настройке нового сервера или применяться к серверам для выполнения операций сопровождения или команд отладки.

sudo — программа для запуска программ с привилегиями другого пользователя. Обычно используется для выполнения команд, нуждающихся в специальных привилегиях (например, редактирования системных конфигурационных файлов).

Контейнер — разновидность технологии виртуализации. Контейнеры представляют собой экземпляры операционной системы с изолированными пользовательскими пространствами, работающие на базе ведущей операционной системы. Контейнеры предоставляют дополнительные средства управления использованием ресурсов и средства безопасности, они быстро создаются и уничтожаются.

Ротация журнала — периодически выполняемая команда, которая переименовывает файлы журналов с включением даты и (не обязательно) сжимает их для экономии пространства.

Глава 11

Open Group — комитет, публикующий Single UNIX Specification — семейство стандартов, позволяющее создателям операционных систем использовать товарный знак UNIX.

stderr — поток для вывода сообщений об ошибках выполняемой программы.

stdin — входной поток выполняемой программы.

stdout — выходной поток для сообщений, выводимых программой.

Ключи командной строки — флаги, передаваемые в командной строке для включения или отключения некоторых функций приложения.

Командный интерпретатор — пользовательский интерфейс командной строки, предназначенный для ввода команд и просмотра результатов. Формирует своего рода «обертку» для операционной системы.

Межпроцессные коммуникации — механизм, предоставляемый операционной системой для обмена данными между программами. Одним из примеров являются каналы, использующие выходные данные одной программы как входные данные другой программы, но даже файлы могут рассматриваться как разновидность межпроцессных коммуникаций.

Перенаправление — сохранение вывода одной программы и передача его на вход другой программы или файла.

Статус завершения — значение, возвращаемое программой при завершении. Ненулевые значения являются признаком ошибки.

Алекс Янг, Брэдли Мек, Майк Кантелон

Node.js в действии

2-е издание

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*И. Карпова*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, И. Тимофеева*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.
Дата изготовления: 12.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 11.12.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*