George Ornbo

24 Proven One-Hour Lessons

"This is a great book to help you quickly get up and running with Node.js. It covers all aspects of the platform and guides you through producing real, production-ready applications."

—**Andrew Nesbitt**, Developer, Forward Internet Group

Sams **Teach Yourself**

# Node.js

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

George Ornbo

# Sams Teach Yourself Node.js

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

**Sams Teach Yourself Node.js in 24 Hours**

Copyright © 2013 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 9780672335952

ISBN-10: 0672335956

Library of Congress Cataloging-in-Publication Data:

Printed in the United States of America

First Printing September 2012

## Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

### Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**
**1-800-382-3419**
**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**
**international@pearsoned.com**

**Editor-in-Chief**
Mark Taub

**Acquisitions Editor**
Laura Lewin

**Development Editor**
Sheri Cain

**Managing Editor**
Kristy Hart

**Project Editor**
Anne Goebel

**Copy Editor**
Geneil Breeze

**Indexer**
Tim Wright

**Proofreader**
Sarah Kearns

**Technical Editor**
Remy Sharp

**Publishing Coordinator**
Olivia Basegio

**Interior Designer**
Gary Adair

**Cover Designer**
Anne Jones

**Senior Compositor**
Gloria Schurick

# Contents at a Glance

Introduction ........ 1

**Part I: Getting Started**

**HOUR 1** Introducing Node.js ........ 7
**2** npm (Node Package Manager) ........ 15
**3** What Node.js Is Used For ........ 27
**4** Callbacks ........ 41

**Part II: Basic Websites with Node.js**

**HOUR 5** HTTP ........ 59
**6** Introducing Express ........ 73
**7** More on Express ........ 91
**8** Persisting Data ........ 103

**Part III: Debugging, Testing, and Deploying**

**HOUR 9** Debugging Node.js Applications ........ 135
**10** Testing Node.js Applications ........ 151
**11** Deploying Node.js Applications ........ 169

**Part IV: Intermediate Sites with Node.js**

**HOUR 12** Introducting Socket.IO ........ 189
**13** A Socket.IO Chat Server ........ 213
**14** A Streaming Twitter Client ........ 237
**15** JSON APIs ........ 265

## Part V: Exploring the Node.js API

**HOUR 16** The Process Module ... 291
**17** The Child Process Module ... 305
**18** The Events Module ... 317
**19** The Buffer Module ... 333
**20** The Stream Module ... 345

## Part VI: Further Node.js Development

**HOUR 21** CoffeeScript ... 361
**22** Creating Node.js Modules ... 381
**23** Creating Middleware with Connect ... 399
**24** Using Node.js with Backbone.js ... 417
Index ... 435

# Table of Contents

**Introduction** **1**

Who Should Read This Book? ... 1
Why Should I Learn Node.js? ... 2
How This Book Is Organized ... 2
Code Examples ... 2
Conventions Used in This Book ... 3

**Part I: Getting Started**

**HOUR 1: Introducing Node.js** **7**

What Is Node.js? ... 7
What You Can Do with Node.js ... 8
Installing and Creating Your First Node.js Program ... 9
Summary ... 11
Q&A ... 12
Workshop ... 12
Exercises ... 13

**HOUR 2: npm (Node Package Manager)** **15**

What Is npm? ... 15
Installing npm ... 16
Installing Modules ... 17
Using Modules ... 17
How to Find Modules ... 19
Local and Global Installation ... 21
How to Find Module Documentation ... 22
Specifying Dependencies with package.json ... 23
Summary ... 25
Q&A ... 25
Workshop ... 26
Exercises ... 26

**HOUR 3: What Node.js Is Used For** 27

What Node.js Is Designed to Do .... 27
Understanding I/O .... 27
Dealing with Input .... 29
Networked I/O Is Unpredictable .... 33
Humans Are Unpredictable .... 35
Dealing with Unpredictability .... 37
Summary .... 38
Q&A .... 38
Workshop .... 39
Exercises .... 39

**HOUR 4: Callbacks** 41

What Is a Callback? .... 41
The Anatomy of a Callback .... 46
How Node.js Uses Callbacks .... 47
Synchronous and Asynchronous Code .... 50
The Event Loop .... 53
Summary .... 54
Q&A .... 55
Workshop .... 55
Exercises .... 56

**Part II: Basic Websites with Node.js**

**HOUR 5: HTTP** 59

What Is HTTP? .... 59
HTTP Servers with Node.js .... 59
HTTP Clients with Node.js .... 69
Summary .... 70
Q&A .... 71
Workshop .... 71
Exercises .... 72

**HOUR 6: Introducing Express** 73
What Is Express? 73
Why Use Express? 73
Installing Express 74
Creating a Basic Express Site 74
Exploring Express 76
Introducing Jade 77
Summary 89
Q&A 89
Workshop 90
Exercises 90

**HOUR 7: More on Express** 91
Routing in Web Applications 91
How Routing Works in Express 91
Adding a GET Route 92
Adding a POST Route 94
Using Parameters in Routes 95
Keeping Routes Maintainable 96
View Rendering 97
Using Local Variables 99
Summary 101
Q&A 101
Workshop 101
Exercises 102

**HOUR 8: Persisting Data** 103
What Is Persistent Data? 103
Writing Data to a File 104
Reading Data from a File 105
Reading Environment Variables 106
Using Databases 108
Using MongoDB with Node.js 109
Summary 131
Q&A 131

Workshop .......... 132
Exercises .......... 132

**Part III: Debugging, Testing, and Deploying**

**HOUR 9: Debugging Node.js Applications** 135
Debugging .......... 135
The STDIO Module .......... 136
The Node.js Debugger .......... 141
Node Inspector .......... 144
A Note on Testing .......... 147
Summary .......... 148
Q&A .......... 148
Workshop .......... 149
Exercises .......... 149

**HOUR 10: Testing Node.js Applications** 151
Why Test? .......... 151
The Assert Module .......... 152
Third-Party Testing Tools .......... 155
Behavior Driven Development .......... 159
Summary .......... 167
Q&A .......... 167
Workshop .......... 168
Exercises .......... 168

**HOUR 11: Deploying Node.js Applications** 169
Ready to Deploy! .......... 169
Hosting in the Cloud .......... 169
Heroku .......... 171
Cloud Foundry .......... 176
Nodester .......... 180
Other PaaS Providers .......... 184
Summary .......... 184
Q&A .......... 184
Workshop .......... 185
Exercises .......... 186

## Part IV: Intermediate Sites with Node.js

**HOUR 12: Introducting Socket.IO** **189**

Now for Something Completely Different ... 189
Brief History of the Dynamic Web ... 189
Socket.IO ... 191
Basic Socket.IO Example ... 191
Sending Data from the Server to Clients ... 194
Broadcasting Data to Clients ... 199
Bi-Directional Data ... 204
Summary ... 209
Q&A ... 209
Workshop ... 210
Exercises ... 210

**HOUR 13: A Socket.IO Chat Server** **213**

Express and Socket.IO ... 213
Adding Nicknames ... 216
Summary ... 235
Q&A ... 235
Workshop ... 236
Exercises ... 236

**HOUR 14: A Streaming Twitter Client** **237**

Streaming APIs ... 237
Signing Up for Twitter ... 238
Using Twitter's API with Node.js ... 241
Extracting Meaning from the Data ... 244
Pushing Data to the Browser ... 247
Creating a Real-Time Lovehateometer ... 252
Summary ... 262
Q&A ... 263
Workshop ... 263
Exercises ... 264

**HOUR 15: JSON APIs** 265

APIs 265

JSON 266

Sending JSON Data with Node.js 268

Creating JSON from JavaScript Objects 269

Consuming JSON Data with Node.js 271

Creating a JSON API with Node.js 275

Summary 285

Q&A 286

Workshop 286

Exercises 287

**Part V: Exploring the Node.js API**

**HOUR 16: The Process Module** 291

What Processes Are 291

Exiting and Errors in Processes 293

Processes and Signals 293

Sending Signals to Processes 295

Creating Scripts with Node.js 297

Passing Arguments to Scripts 298

Summary 301

Q&A 302

Workshop 302

Exercises 303

**HOUR 17: The Child Process Module** 305

What Is a Child Process? 305

Killing a Child Process 308

Communicating with a Child Process 309

The Cluster Module 311

Summary 314

Q&A 314

Workshop 314

Exercises 315

**HOUR 18: The Events Module** 317

Understanding Events ........ 317
Demonstrating Events Through HTTP ........ 321
Playing Ping-Pong with Events ........ 324
Programming Event Listeners Dynamically ........ 326
Summary ........ 330
Q&A ........ 330
Workshop ........ 331
Exercises ........ 331

**HOUR 19: The Buffer Module** 333

A Primer on Binary Data ........ 333
Binary to Text ........ 334
Binary and Node.js ........ 335
What Are Buffers in Node.js? ........ 338
Writing to Buffers ........ 340
Appending to Buffers ........ 340
Copying Buffers ........ 342
Modifying Strings in Buffers ........ 343
Summary ........ 343
Q&A ........ 343
Workshop ........ 344
Exercises ........ 344

**HOUR 20: The Stream Module** 345

A Primer on Streams ........ 345
Readable Streams ........ 347
Writable Streams ........ 352
Piping Streams ........ 353
Streaming MP3s ........ 354
Summary ........ 356
Q&A ........ 356
Workshop ........ 356
Exercises ........ 357

## Part VI: Further Node.js Development

**HOUR 21: CoffeeScript** 361

What Is CoffeeScript? 361
Installing and Running CoffeeScript 363
Why Use a Pre-Compiler? 365
Features of CoffeeScript 366
Debugging CoffeeScript 376
Reactions to CoffeeScript 377
Summary 378
Q&A 378
Workshop 379
Exercises 379

**HOUR 22: Creating Node.js Modules** 381

Why Create Modules? 381
Popular Node.js Modules 381
The package.json File 383
Folder Structure 384
Developing and Testing Your Module 385
Adding an Executable 388
Using Object-Oriented or Prototype-Based Programming 390
Sharing Code Via GitHub 391
Using Travis CI 392
Publishing to npm 395
Publicizing Your Module 397
Summary 397
Q&A 397
Workshop 398
Exercises 398

**HOUR 23: Creating Middleware with Connect** 399

What Is Middleware? 399
Middleware in Connect 400
Access Control with Middleware 406

Summary ... 414
Q&A ... 414
Workshop ... 415
Exercises ... 415

**HOUR 24: Using Node.js with Backbone.js** **417**

What Is Backbone.js? ... 417
How Backbone.js Works ... 418
A Simple Backbone.js View ... 425
Creating Records with Backbone.js ... 429
Summary ... 432
Q&A ... 432
Workshop ... 433
Exercises ... 433

**Index** **435**

# About the Author

**George Ornbo** is a UK-based JavaScript and Ruby developer. He has been creating web applications for more than eight years, first as a freelancer and more recently working at pebble {code} in London. He blogs at http://shapeshed.com and can be found in most of the usual places around the web as @shapeshed.

# Dedication

*This book is dedicated to my wife, Kirsten.*
*Without your support, this book would not have been possible.*

# Acknowledgments

Thanks to Trina MacDonald and the team at Pearson for giving me the chance to write this book. Your encouragement and guidance was invaluable.

Thanks to Remy Sharp, the technical editor on the book. You picked up numerous mistakes and oversights over the course of the reviews. I owe you a beer! Any mistakes left in the book are, of course, my own.

Thanks to my colleagues at pebble {code}. From the start, you were right behind me writing the book. I am grateful for the flexibility around big projects that allowed me to finish this book.

# We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: errata@informit.com

Mail: Addison-Wesley/Prentice Hall Publishing
ATTN: Reader Feedback
1330 Avenue of the Americas
35th Floor
New York, New York, 10019

# Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

# Introduction

The ability to use JavaScript on the server allows developers who are familiar with JavaScript to add server-side development to their curriculum vitae. Node.js is much more than that, though. It rethinks network programming in the context of the modern web where an application may rely on reading and writing data from many different places and may have millions of concurrent users.

JavaScript is often seen as a toy language by developers who have traditional computer science degrees. But, JavaScript has survived numerous challenges and is now integral to the direction of the web both in the browser and with Node.js on the server-side too. There has never been a better time to write JavaScript, especially on the server!

Node.js represents a development platform that can respond to creating applications for the modern web. This includes

- Real-time applications
- Multiplayer games
- Single-page applications
- JSON-based APIs

It is focused on speed and scalability and can handle thousands of concurrent users without needing expensive hardware. The Node.js project recently became the most watched project on GitHub and is now being used by companies like eBay, LinkedIn, and Microsoft.

Node.js is much more than JavaScript on the server. It is a fully featured network programming platform for responding to the demands of modern web programming.

## Who Should Read This Book?

This book makes few assumptions about programming experience, but it is helpful to have some basic experience with JavaScript. Because Node.js is primarily run from the terminal, it is helpful to understand what a terminal is and how to run basic commands. Finally, because Node.js is primarily a network programming tool, it helps to understand a little of how the Internet works, although this is not essential.

## Why Should I Learn Node.js?

If you are interested in creating applications that have many users, deal with networked data, or have real-time requirements, then Node.js is a great tool for the job. Furthermore, if you are creating applications for the browser, Node.js allows your server to be JavaScript, making it much simpler to share data between your server and client. Node.js is a modern toolkit for the modern web!

## How This Book Is Organized

This books starts with the basics of Node.js, including running your first Node.js program and using npm (Node's package manager). You are then introduced to network programming and how Node.js uses JavaScript callbacks to support an asynchronous style of programming.

In Part II, you learn how to create basic websites with Node.js first by using the HTTP module and then using Express, a web framework for Node.js. You also learn how to persist data with MongoDB.

Part III introduces tools for debugging and testing Node.js application. You are introduced to a number of debugging tools and testing frameworks to support your development. You learn how to deploy your Node.js applications to a number of third-party services, including Heroku and Nodester.

Part IV showcases the real-time capabilities of Node.js and introduces Socket.IO. You learn how to send messages between the browser and server and build full examples of a chat server and a real-time Twitter client. Finally, you learn how to create JSON APIs with Node.js.

Part V focuses on the Node.js API and explores the building blocks for creating Node.js applications. You learn about processes, child processes, events, buffers, and streams.

Part VI introduces areas that you may want to explore once you get beyond the basics. You learn about CoffeeScript, a JavaScript pre-compiler, how to use Middleware with Node.js, and how to use Backbone.js to create single-page applications with Node.js. Hour 22 also introduces how to write and publish your own Node.js modules with npm.

## Code Examples

Each hour in this book comes with several code examples. These examples help you learn about Node.js as much as the text in this book. You can download this code at http://bit.ly/nodejsbook-examples, and they are also available as a GitHub repository at https://github.com/shapeshed/nodejsbook.io.examples.

# Conventions Used in This Book

Each hour starts with "What You'll Learn in This Hour," which includes a brief list of bulleted points highlighting the hour's contents. A summary concluding each hour provides a bit of insight reflecting on what you as the reader should have learned along the way.

In each hour, any text that you type appears as **`bold monospace`**, whereas text that appears on your screen is presented in `monospace` type.

```
It will look like this to mimic the way text looks on your screen.
```

Finally, the following icons introduce other pertinent information used in the book:

BY THE WAY

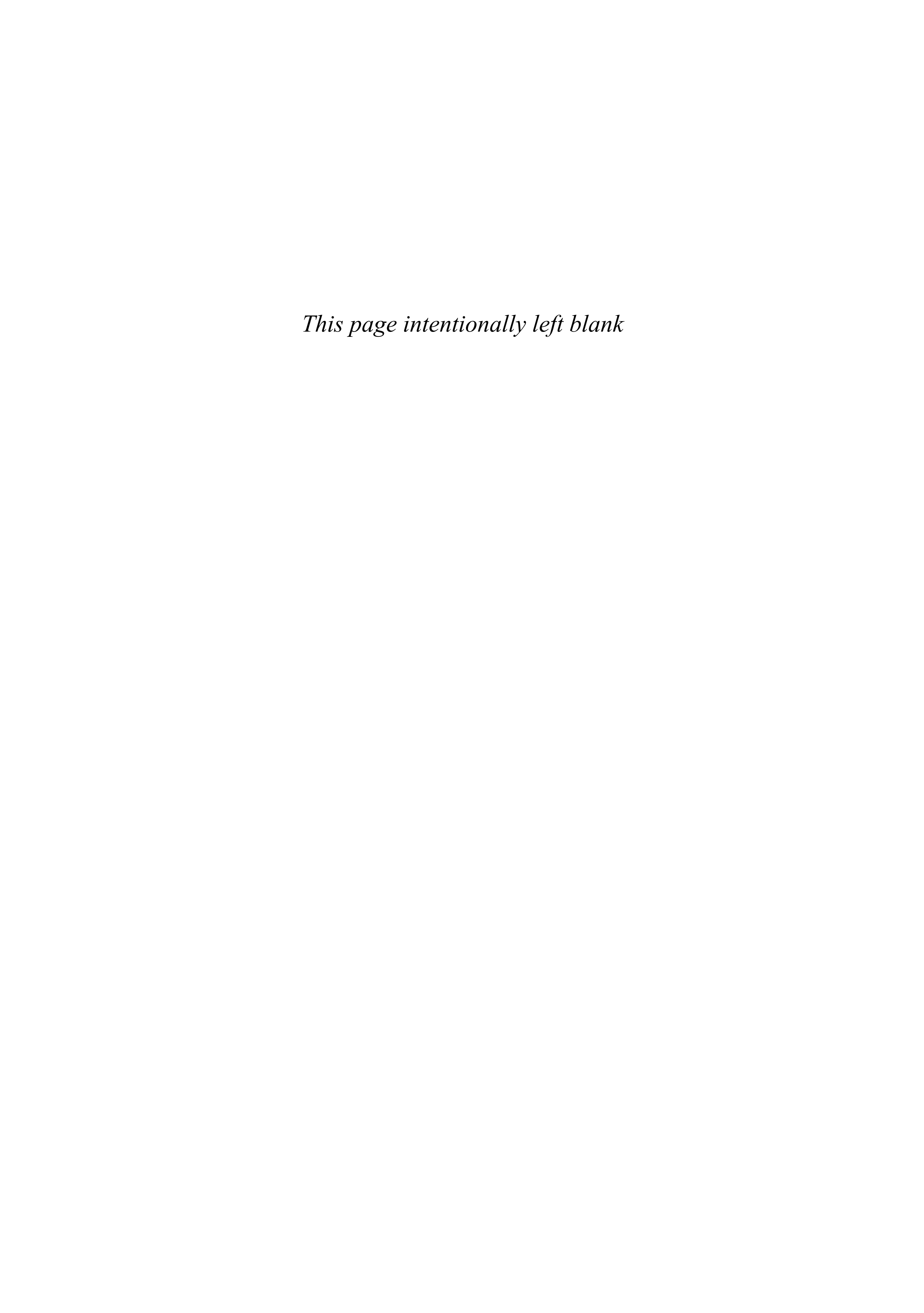
By the Way presents interesting pieces of information related to the surrounding discussion.

DID YOU KNOW?

Did You Know? offers advice or teaches an easier way to do something.

WATCH OUT

Watch Out! advises you about potential problems and helps you steer clear of disaster.


This page intentionally left blank

# HOUR 14
# A Streaming Twitter Client

**What You'll Learn in This Hour:**

- ▶ Receive data from Twitter's streaming API
- ▶ Parse data received from Twitter's streaming API
- ▶ Push third-party data out to clients in real-time
- ▶ Create a real-time graph
- ▶ Discover whether there is more love or hate in the world by using real-time data from Twitter

## Streaming APIs

In Hour 13, "A Socket.IO Chat Server," you learned how to create a chat server with Socket.IO and Express. This involved sending data from clients (or browsers) to the Socket.IO server and then broadcasting it out to other clients. In this hour, you learn how Node.js and Socket.IO can be used to consume data directly from the web and then broadcast the data to connected clients. You will work with Twitter's streaming Application Programming Interface (API) and push data out to the browser in real-time.

With Twitter's standard API, the process for getting data is as follows:

1. You open a connection to the API server.
2. You send a request for some data.
3. You receive the data that you requested from the API.
4. The connection is closed.

With Twitter's streaming API, the process is different:

1. You open a connection to the API server.
2. You send a request for some data.
3. Data is pushed to you from the API.
4. The connection remains open.
5. More data is pushed to you when it becomes available.

Streaming APIs allow data to be pushed from the service provider whenever new data is available. In the case of Twitter, this data can be extremely frequent and high volume. Node.js is a great fit for this type of scenario, where large numbers of events are happening frequently as data is received. This hour represents another excellent use case for Node.js and highlights some of the features that make Node.js different from other languages and frameworks.

## Signing Up for Twitter

Twitter provides a huge amount of data to developers via a free, publically available API. Many Twitter desktop and mobile clients are built on top of this API, but this is also open to developers to use however they want.

If you do not already have a Twitter account, you need one for this hour. You can sign up for an account for free at https://twitter.com/. It takes less than a minute! Once you have a Twitter account, you need to sign into the Twitter Developers website with your details at http://dev.twitter.com/. This site provides documentation and forums for anything to do with the Twitter API. The documentation is thorough, so if you want, you can get a good understanding of what types of data you can request from the API here.

Within the Twitter Developers website, you can also register applications that you create with the Twitter API. You create a Twitter application in this hour, so to register your application, do the following:

1. Click the link Create an App.
2. Pick a name for your application and fill out the form (see Figure 14.1). Application names on Twitter must be unique, so if you find that the name has already been taken, choose another one.

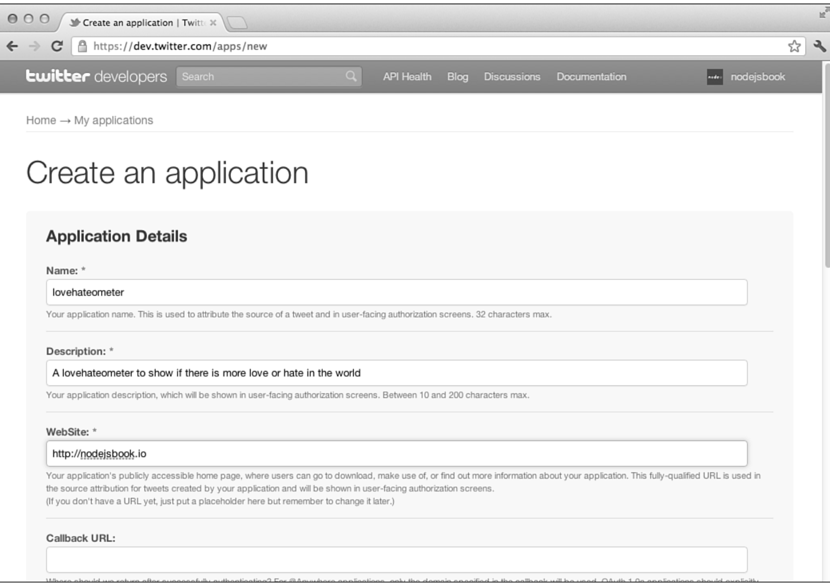



**FIGURE 14.1**
Creating a Twitter application

Once you create your application, you need to generate an access token and an access-token secret to gain access to the API from your application.

3. At the bottom of the Details tab is a Create My Access Token button (see Figure 14.2). Click this button to create an access token and an access token secret.

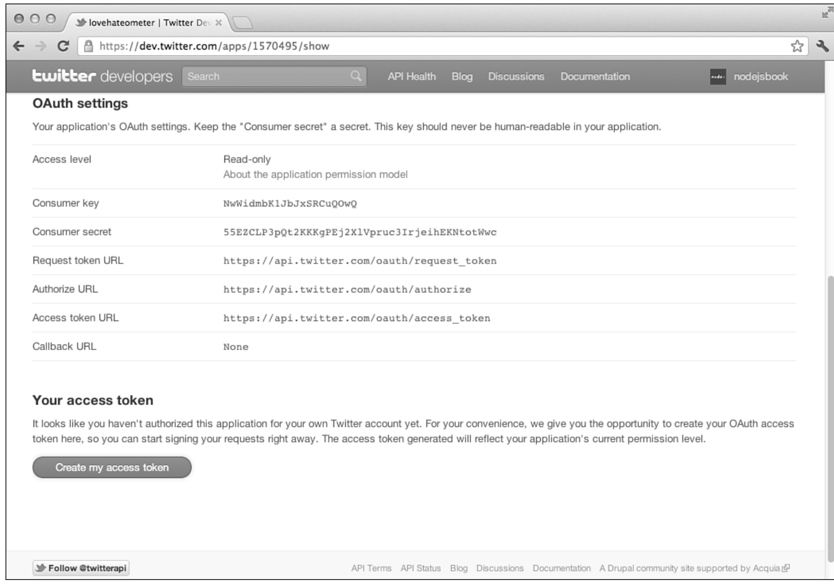



**FIGURE 14.2**
Requesting an access token

4. When the page refreshes, you see that values have been added for access token and access token secret (see Figure 14.3). Now, you are ready to start using the API!

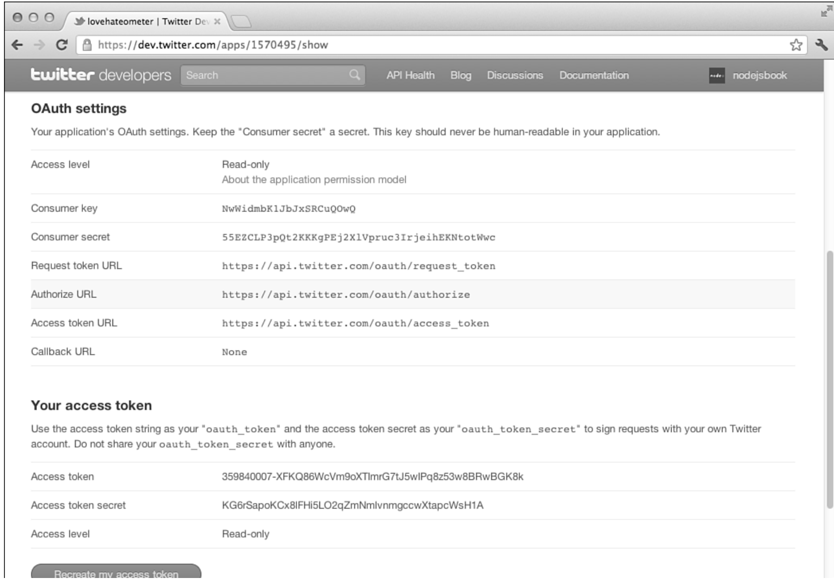



**FIGURE 14.3**
A successful creation of an access token

BY THE WAY

**OAuth Is a Way of Allowing Access to Online Accounts**

OAuth is an open standard for authentication, typically used within the context of web applications. It allows users to grant access to all or parts of an account without handing over a username or password. When a user grants an application access to their account, a unique token is generated. This can be used by a third-party services to access all or parts of a user's account. At any time, the user can revoke access and the token will no longer be valid so an application would no longer have access to the account.

# Using Twitter's API with Node.js

Once you create your application within the Twitter Developers website and request an OAuth access token, you are ready to start using the Twitter API. An excellent Node.js module is available for interacting with the Twitter API called ntwitter. This module was initially developed by technoweenie (Rick Olson), then jdub (Jeff Waugh), and is now maintained AvianFlu (Charlie McConnell). All the authors have done an amazing job of abstracting the complexity of interacting with Twitter's API to make it simple to get data and do things with it. You continue to use Express in this hour, so the package.json file for the application will include the Express and ntwitter modules.

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10"
  }
}
```

The ntwitter module uses OAuth to authenticate you, so you must provide four pieces of information:

- Consumer key
- Consumer secret
- Access token key
- Access token secret

If you requested these when you were setting up the application in the Twitter Developers website, these will be available on the Details page for your application. If you did not request them when you set up the application, you need to do so now under the Details tab. Once you have the keys and secrets, you can create a small Express server to connect to Twitter's streaming API:

```
var app = require('express').createServer(),
    twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
  consumer_secret: 'YOUR_CONSUMER_SECRET',
  access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
  access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});
```

Of course, you need to remember to replace the values in the example with your actual values. This is all you need to start interacting with Twitter's API! In this example, you answer the question, "Is there more love or hate in the world?" by using real-time data from Twitter. You request tweets from Twitter's streaming API that mention the words "love" or "hate" and perform a small amount of analysis on the data to answer the question. The ntwitter module makes it easy to request this data:

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    console.log(data);
  });
});
```

This requests data from the `'statuses/filter'` endpoint that allows developers to track tweets by keyword, location, or specific users. In this case, we are interested in the keywords `'love'` and `'hate'`. The Express server opens a connection to the API server and listens for new data being received. Whenever a new data item is received, it writes the data to the console. In other words, you can see the stream live for the keywords "love" and "hate" in the terminal.

**TRY IT YOURSELF ▼**

If you have downloaded the code examples for this book, this code is hour14/example01.

To stream data from Twitter, follow these steps:

1. Create a new folder called express_twitter.
2. Within the express_twitter folder, create a new file called package.json and add the following content to declare ntwitter and Express as dependencies:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10"
  }
}
```

3. Within the express_twitter folder, create a new file called app.js with the following content. Remember to replace the keys and secrets with your own:

```
var app = require('express').createServer(),
    twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
  consumer_secret: 'YOUR_CONSUMER_SECRET',
  access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
  access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
```

```
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    console.log(data);
  });
});
```

4. Install the dependencies by running the following from a terminal:

```
npm install
```

5. Start the server by running the following from a terminal:

```
node app.js
```

6. Watch the terminal; you should see data being received from Twitter’s streaming API (see Figure 14.4). There is a lot of data, so expect it to move fast!
7. Kill the server pressing Ctrl+C in the terminal.

Terminal — node — 80×24


```
    notifications: null,
    profile_background_tile: false,
    follow_request_sent: null,
    profile_sidebar_fill_color: '252429',
    created_at: 'Sat Dec 11 15:04:33 +0000 2010',
    protected: false,
    default_profile_image: false,
    contributors_enabled: false,
    profile_sidebar_border_color: '181A1E',
    followers_count: 21,
    profile_image_url: 'http://a0.twimg.com/profile_images/1789738168/____norma
l.jpg',
    name: 'Noura',
    id_str: '225434395',
    favourites_count: 0,
    id: 225434395,
    lang: 'en',
    profile_use_background_image: true,
    utc_offset: -18000,
    url: null },
  in_reply_to_screen_name: null,
  id: 164010055543435260,
  entities: { user_mentions: [], urls: [], hashtags: [ [Object] ] } }
```


**FIGURE 14.4**
Streaming data to the terminal

# Extracting Meaning from the Data

So far, you created a way to retrieve data in real-time from Twitter, and you saw a terminal window move very fast with a lot of data. This is good, but in terms of being able to understand the data, you are not able to answer the question set. To work toward this, you need to parse the tweets received and extract information. Twitter provides data in JSON, a subset of JavaScript, and this is great news for using it with Node.js. For each response, you can simply use dot

notation to retrieve the data that you are interested in. So, if you wanted to view the screen name of the user along with the tweet, this can be easily achieved:

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    console.log(data.user.screen_name + ': ' + data.text);
  });
});
```

Full documentation on the structure of the data received from Twitter is available on the documentation for the status element. This can be viewed online at https://dev.twitter.com/docs/api/1/get/statuses/show/%3Aid. Under the section, "Example Request," you can see the data structure for a status response. Using dot notation on the data object returned from Twitter, you are able to access any of these data points. For example, if you want the URL for the user, you can use data.user.url. Here is the full data available for the user who posted the tweet:


```
"user": {
    "profile_sidebar_border_color": "eeeeee",
    "profile_background_tile": true,
    "profile_sidebar_fill_color": "efefef",
    "name": "Eoin McMillan ",
    "profile_image_url": "http://a1.twimg.com/profile_images/1380912173/Screen_
➥ shot_2011-06-03_at_7.35.36_PM_normal.png",
    "created_at": "Mon May 16 20:07:59 +0000 2011",
    "location": "Twitter",
    "profile_link_color": "009999",
    "follow_request_sent": null,
    "is_translator": false,
    "id_str": "299862462",
    "favourites_count": 0,
    "default_profile": false,
    "url": "http://www.eoin.me",
    "contributors_enabled": false,
    "id": 299862462,
    "utc_offset": null,
    "profile_image_url_https": "https://si0.twimg.com/profile_images/1380912173/
➥ Screen_shot_2011-06-03_at_7.35.36_PM_normal.png",
    "profile_use_background_image": true,
    "listed_count": 0,
    "followers_count": 9,
    "lang": "en",
    "profile_text_color": "333333",
    "protected": false,
    "profile_background_image_url_https": "https://si0.twimg.com/images/themes/
➥ theme14/bg.gif",
    "description": "Eoin's photography account. See @mceoin for tweets.",
    "geo_enabled": false,
    "verified": false,
```

```
    "profile_background_color": "131516",
    "time_zone": null,
    "notifications": null,
    "statuses_count": 255,
    "friends_count": 0,
    "default_profile_image": false,
    "profile_background_image_url": "http://a1.twimg.com/images/themes/theme14/
➥ bg.gif",
    "screen_name": "imeoin",
    "following": null,
    "show_all_inline_media": false
  }
```


There is much more information available with each response, including geographic coordinates, whether the tweet was retweeted, and more.

## ▼ TRY IT YOURSELF

If you have downloaded the code examples for this book, this code is hour14/example02.

To parse data from Twitter, follow these steps:

1. Create a new folder called parsing_twitter_data.
2. Within the parsing_twitter_data folder, create a new file called package.json and add the following content to declare ntwitter and Express as dependencies:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10"
  }
}
```

3. Within the express_twitter folder, create a new file called app.js with the following content. Remember to replace the keys and secrets with your own:

```
var app = require('express').createServer(),
    twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
```

```
    consumer_secret: 'YOUR_CONSUMER_SECRET',
    access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
    access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
  });

  twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
    stream.on('data', function (data) {
      console.log(data.user.screen_name + ': ' + data.text);
    });
  });
```

**4.** Install the dependencies by running the following from a terminal:

```
npm install
```

**5.** Start the server by running the following from a terminal:

```
node app.js
```

**6.** Watch the terminal; you should see that now only the screen name of the user and the tweet are displayed (see Figure 14.5).

**7.** Kill the server by pressing Ctrl+C in the terminal.

```
Terminal — node — 80×24
accioabbie: RT @justsaypenny: I love @accioabbie and @underapapermo0n they are m
y life XxxxxxxxxxxxxxxxxxxXxxxxcc
JacobIrsan: RT"@monicaaakim: pacaran : love you beb, miss you &lt;3 .. putus : f
uck you! damn you..."
NSUCoachJones: No ESPN love, 16-6, 8-0 Conference, preseason player of year and
they show the 5th and 6th place teams..
jockinmyfresh24: Call me a jerk one more time ctfu lmao RT @TassieBaddAss: @jock
inmyfresh24 you freakin love me you lil jerk!!!
Huangaholic: @TheManuMania This, a thousand times this. See, this is why I love
you guys. @foreword55 @jeannette_sites
imaginebojan: Jefferson Airplane Somebody To Love   http://t.co/1bl9RH76  兄さん
バレるな！ｗｗｗ  空耳？
raulorlandoJ: "Never love someone easily" real man
iWant_GUMMiES: i hate when #Twitter try to tell me i already tweeted something a
nd i didn't
ChoczBB: Lowkey Wizkid RT @DONJAYYYY: Kalamoko Skally  RT @ChoczBB: Oti yan yan
Slim joe RT @DONJAYYYY: Dadubule Skally RT @ChoczBB: No Love Eminem
_YummyKellogs: I hate Mondays, and I'm tired |:
kcruzz2: But jealousy is just love n hate at the same time
hishh: Among the all actress i know @divyaspandana is a sweet heart. Real gem ,
brave , dedicated and the best.. Respect.. Loads of love
LUcKyAcE_103: RT @Mr_Gjasmin5: I love google
JakePena7: RT @BW0RD: You're tacky and I hate you
```

**FIGURE 14.5**
Parsing data received from Twitter

# Pushing Data to the Browser

Now that data from Twitter is in a more digestible format, you can push this data out to connected browsers using Socket.IO and use some client-side JavaScript to display the tweets. This is similar to the patterns you saw in Hours 12 and 13, where data is received by a Socket.

IO server and then broadcast to connected clients. To use Socket.IO, it must first be added as a dependency in the package.json file:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10",
    "socket.io":"0.8.7"
  }
}
```

Then, Socket.IO must be required in the main server file and instructed to listen to the Express server. This is exactly the same as the examples you worked through in Hours 12 and 13:

```
var app = require('express').createServer(),
    twitter = require('ntwitter'),
    io = require('socket.IO').listen(app);
```

The streaming API request can now be augmented to push the data out to any connected Socket.IO clients whenever a new data event is received:

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    io.sockets.volatile.emit('tweet', {
      user: data.user.screen_name,
      text: data.text
    });
  });
});
```

Instead of logging the data to the console, you are now doing something useful with the data by pushing it out to connected clients. A simple JSON structure is created to hold the name of the user and the tweet. If you want to send more information to the browser, you could simply extend the JSON object to hold other attributes.

You may have noticed that, instead of using `io.sockets.emit` as you did in Hours 12 and 13, you are now using `io.sockets.volatile.emit`. This is an additional method provided by Socket.IO for scenarios where certain messages can be dropped. This may be down to network issues or a user being in the middle of a request-response cycle. This is particularly the case where high volumes of messages are being sent to clients. By using the `volatile` method, you can ensure that your application will not suffer if a certain client does not receive a message. In other words, it does not matter whether a client does not receive a message.

The Express server is also instructed to serve a single HTML page so that the data can be viewed in a browser.

```
app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});
```

On the client side (or browser), some simple client-side JavaScript is added to the index.html file to listen for new tweets being sent to the browser and display them to the user. The full HTML file is available in the following example:

```
<ul class="tweets"></ul>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
➥ script>
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect();
  jQuery(function ($) {
    var tweetList = $('ul.tweets');
    socket.on('tweet', function (data) {
      tweetList
        .prepend('<li>' + data.user + ': ' + data.text + '</li>');
    });
  });
</script>
```

An empty unordered list is added to the DOM (Document Object Model), and this is filled with a new list item containing the screen name of the user and the tweet each time a new tweet is received. This uses jQuery's `prepend()` method to insert data received into a list item within the unordered list. This has the effect of creating a stream on the page.

Now, whenever Socket.IO pushes a new tweet event out, the browser receives it and writes it to the page immediately. Instead of viewing the stream of tweets in a terminal, it can now be viewed in the browser.

**TRY IT YOURSELF ▼**

If you have downloaded the code examples for this book, this code is hour14/example03.

Here's how to stream Twitter data to a browser:

1. Create a new folder called socket.io-twitter-example.
2. Within the socket.io-twitter-example folder, create a new file called package.json and add the following content to declare ntwitter, Express, and Socket.IO as dependencies:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
```

```
    "express":"2.5.4",
    "ntwitter":"0.2.10",
    "socket.io":"0.8.7"
  }
}
```

3. Within the socket.io-twitter-example folder, create a new file called app.js with the following content. Remember to replace the keys and secrets with your own:

```
var app = require('express').createServer(),
    twitter = require('ntwitter'),
    io = require('socket.io').listen(app);

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
  consumer_secret: 'YOUR_CONSUMER_SECRET',
  access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
  access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    io.sockets.volatile.emit('tweet', {
      user: data.user.screen_name,
      text: data.text
    });
  });
});

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});
```

4. Within the Socket.IO-twitter-example, create a file called index.html and add the following content:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Socket.IO Twitter Example</title>
  </head>
  <body>
    <h1>Socket.IO Twitter Example</h1>
    <ul class="tweets"></ul>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
➥min.js"></script>
```

```
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var socket = io.connect();
      jQuery(function ($) {
        var tweetList = $('ul.tweets');
        socket.on('tweet', function (data) {
          tweetList
            .prepend('<li>' + data.user + ': ' + data.text + '</li>');
        });
      });
    </script>
  </body>
</html>
```

5. Install the dependencies by running the following from a terminal:

   ```
   npm install
   ```

6. Start the server by running the following from a terminal:

   ```
   node app.js
   ```

7. Open a browser window at http://127.0.0.1:3000.
8. You should see a stream of tweets in your browser (see Figure 14.6).
9. Kill the server by pressing Ctrl+C in the terminal.

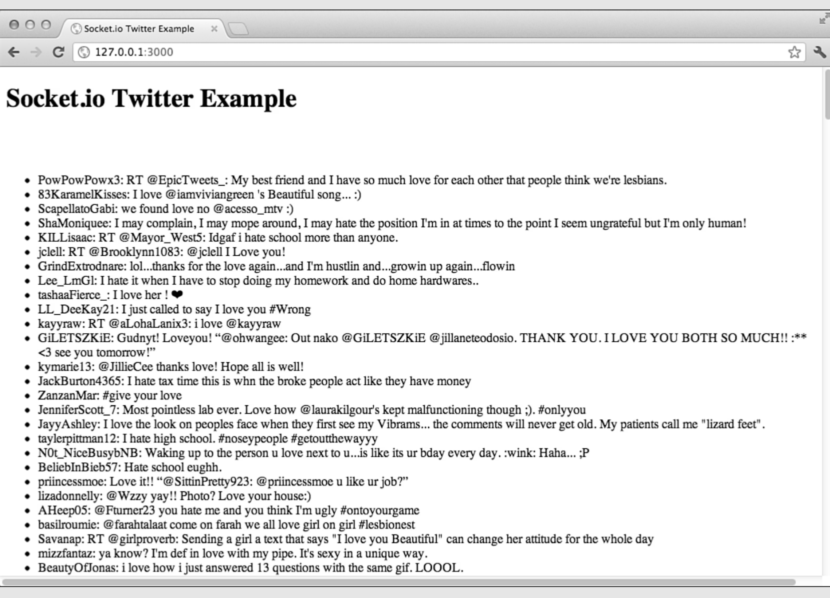



**FIGURE 14.6**
Streaming tweets to the browser

## Creating a Real-Time Lovehateometer

Although the application can now stream tweets to a browser window, it is still not very useful. It is still impossible to answer the question of whether there is more love or hate in the world. To answer the question, you need a way to visualize the data. Assuming that the tweets received from the API are indicative of human sentiment, you set up several counters on the server that increment when the words "love" and "hate" are mentioned in the streaming data that is received. Furthermore, by maintaining another counter for the total number of tweets with either love or hate in them, you can calculate whether love or hate is mentioned more often. With this approach, it is possible to say—in unscientific terms—that there is *x*% of love and *y*% of hate in the world.

To be able to show data in the browser, you need counters on the server to hold:

- Total number of tweets containing "love" or "hate"
- Total number of tweets containing "love"
- Total number of tweets containing "hate"

This can be achieved by initializing variables and setting these counters to zero on the Node.js server:

```
var app = require('express').createServer(),
    twitter = require('ntwitter'),
    io = require('socket.io').listen(app),
    love = 0,
    hate = 0,
    total = 0;
```

Whenever new data is received from the API, the love counter will be incremented if the word "love" is found and so on. JavaScript's `indexOf()` string function can be used to look for words within a tweet and provides a simple way to analyze the content of tweets:

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {

    var text = data.text.toLowerCase();
    if (text.indexOf('love') !== -1) {
      love++
      total++
    }
    if (text.indexOf('hate') !== -1) {
      hate++
      total++
    }
  });
});
```

Because some tweets may contain both “love” and “hate,” the total is incremented each time a word is found. This means that the total counter represents the total number of times “love” or “hate” was mentioned in a tweet rather than the total number of tweets.

Now that the application is maintaining a count of the occurrences of words, this data can be added to the tweet emitter and pushed to connected clients in real-time. Some simple calculation is also used to send the values as a percentage of the total number of tweets:

```
io.sockets.volatile.emit('tweet', {
  user: data.user.screen_name,
  text: data.text,
  love: (love/total)*100,
  hate: (hate/total)*100
});
```

On the client side, by using an unordered list and some client-side JavaScript, the browser can receive the data and show it to users. Before any data is received from the server, the values are set to zero:

```
<ul class="percentage">
  <li class="love">0</li>
  <li class="hate">0</li>
</ul>
```

Finally, a client-side listener can be added to receive the tweet event and replace the percentage values with the ones received from the server. By starting the server and opening the browser, you can now answer the question!

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
➥ script>
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect();
  jQuery(function ($) {
    var tweetList = $('ul.tweets'),
        loveCounter = $('li.love'),
        hateCounter = $('li.hate');
    socket.on('tweet', function (data) {
      tweetList
        .prepend('<li>' + data.user + ': ' + data.text + '</li>');
      loveCounter
        .text(data.love + '%');
      hateCounter
       .text(data.hate + '%');
    });
  });
</script>
```

▼ TRY IT YOURSELF

If you have downloaded the code examples for this book, this code is hour14/example04.

To analyze data from Twitter's streaming API, follow these steps:

1. Create a new folder called percentages.
2. Within the percentages folder, create a new file called package.json and add the following content to declare ntwitter, Express, and Socket.IO as dependencies:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10",
    "socket.io":"0.8.7"
  }
}
```

3. Within the percentages folder, create a new file called app.js with the following content. Remember to replace the keys and secrets with your own:

```
var app = require('express').createServer(),
    twitter = require('ntwitter'),
    io = require('socket.io').listen(app),
    love = 0,
    hate = 0,
    total = 0;

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
  consumer_secret: 'YOUR_CONSUMER_SECRET',
  access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
  access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {
    var text = data.text.toLowerCase();
    if (text.indexOf('love') !== -1) {
      love++
      total++
    }
    if (text.indexOf('hate') !== -1) {
```

```
      hate++
      total++
    }
    io.sockets.volatile.emit('tweet', {
      user: data.user.screen_name,
      text: data.text,
      love: (love/total)*100,
      hate: (hate/total)*100
    });
  });
});

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});
```

4. Within the percentages folder, create a file called index.html and add the following content:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Socket.IO Twitter Example</title>
  </head>
  <body>
    <h1>Socket.IO Twitter Example</h1>
    <ul class="percentage">
      <li class="love">0</li>
      <li class="hate">0</li>
    </ul>
    <ul class="tweets"></ul>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
➥ min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var socket = io.connect();
      jQuery(function ($) {
        var tweetList = $('ul.tweets'),
            loveCounter = $('li.love'),
            hateCounter = $('li.hate');
        socket.on('tweet', function (data) {
          tweetList
            .prepend('<li>' + data.user + ': ' + data.text + '</li>');
          loveCounter
            .text(data.love + '%');
          hateCounter
```

```
            .text(data.hate + '%');
        });
      });
    </script>
  </body>
</html>
```

5. Install the dependencies by running the following from a terminal:

```
npm install
```

6. Start the server by running the following from a terminal:

```
node app.js
```

7. Open a browser window at http://127.0.0.1:3000.
8. You should see a stream of tweets in your browser, along with the percentages being dynamically updated (see Figure 14.7).
9. Kill the server by pressing Ctrl+C in the terminal.

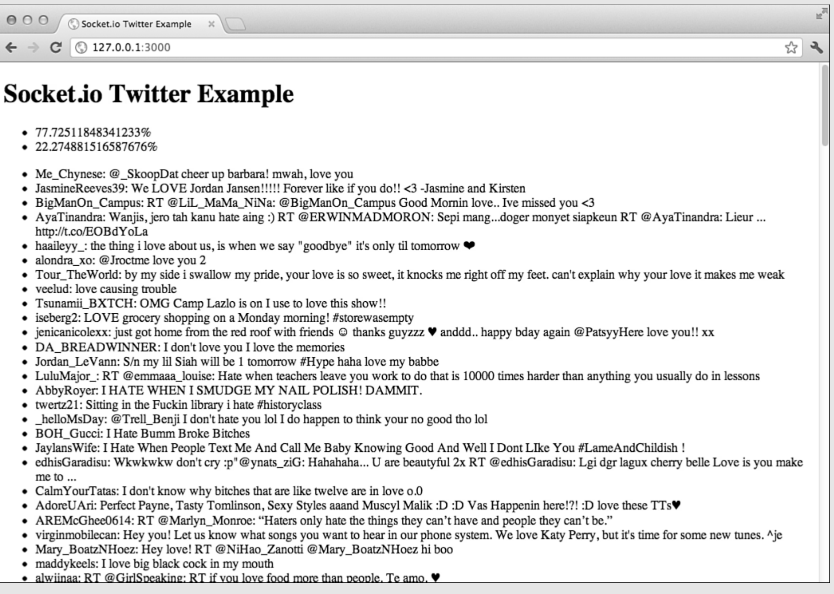



**FIGURE 14.7**
Dynamically updating percentage values

## Adding a Real-Time Graph

The application is now able to answer the question. Hurray! In terms of visualization, though, it is still just data. It would be great if the application could generate a small bar graph that moved dynamically based on the data received. The server is already sending this data to the browser, so this can be implemented entirely using client-side JavaScript and some CSS. The application has an unordered list containing the percentages, and this is perfect to create a simple bar graph. The unordered list will be amended slightly so that it is easier to style. The only addition here is to wrap the number in a `span` tag:

```
<ul class="percentage">
  <li class="love">
    <span>0</span>
  </li>
  <li class="hate">
    <span>0</span>
  </li>
</ul>
```

Some CSS can then be added to the head of the HTML document that makes the unordered list look like a bar graph. The list items represent the bars with colors of pink to represent love and black to represent hate:

```
<style>
  ul.percentage { width: 100% }
  ul.percentage li { display: block; width: 0 }
  ul.percentage li span { float: right; display: block}
  ul.percentage li.love { background: #ff0066; color: #fff}
  ul.percentage li.hate { background: #000; color: #fff}
</style>
```

Finally, some client-side JavaScript allows the bars (the list items) to be resized dynamically based on the percentage values received from the server:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
➥ script>
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect();
  jQuery(function ($) {
    var tweetList = $('ul.tweets'),
        loveCounter = $('li.love'),
        hateCounter = $('li.hate'),
        loveCounterPercentage = $('li.love span'),
        hateCounterPercentage = $('li.hate span');
    socket.on('tweet', function (data) {
      loveCounter
```

```
        .css("width", data.love + '%');
      loveCounterPercentage
        .text(Math.round(data.love * 10) / 10 + '%');
      hateCounter
        .css("width", data.hate + '%');
      hateCounterPercentage
        .text(Math.round(data.hate * 10) / 10 + '%');
      tweetList
        .prepend('<li>' + data.user + ': ' + data.text + '</li>');
    });
  });
</script>
```

Whenever a new tweet event is received from Socket.IO, the bar graph is updated by dynamically setting the CSS width of the list items with the percentage values received from the server. This has the effect of adjusting the graph each time a new tweet event is received. You have created a real-time graph!

### ▼ TRY IT YOURSELF

If you have downloaded the code examples for this book, this code is hour14/example05.

Follow these steps to visualize real-time data:

1. Create a new folder called realtime_graph.
2. Within the realtime_graph folder, create a new file called package.json and add the following content to declare ntwitter, Express, and Socket.IO as dependencies:

```
{
  "name":"socket.io-twitter-example",
  "version":"0.0.1",
  "private":true,
  "dependencies":{
    "express":"2.5.4",
    "ntwitter":"0.2.10",
    "socket.io":"0.8.7"
  }
}
```

3. Within the realtime_graph folder, create a new file called app.js with the following content. Remember to replace the keys and secrets with your own:

```
var app = require('express').createServer(),
    twitter = require('ntwitter'),
    io = require('socket.io').listen(app),
    love = 0,
    hate = 0,
```

```
    total = 0;

app.listen(3000);

var twit = new twitter({
  consumer_key: 'YOUR_CONSUMER_KEY',
  consumer_secret: 'YOUR_CONSUMER_SECRET',
  access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
  access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
  stream.on('data', function (data) {

    var text = data.text.toLowerCase();
    if (text.indexOf('love') !== -1) {
      love++
      total++
    }
    if (text.indexOf('hate') !== -1) {
      hate++
      total++
    }

    io.sockets.volatile.emit('tweet', {
      user: data.user.screen_name,
      text: data.text,
      love: (love/total)*100,
      hate: (hate/total)*100
    });
  });
});

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});
```

**4.** Within the realtime_graph folder, create a file called index.html and add the following content:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Socket.IO Twitter Example</title>
    <style>
      ul.percentage { width: 100% }
```

```
      ul.percentage li { display: block; width: 0 }
      ul.percentage li span { float: right; display: block}
      ul.percentage li.love { background: #ff0066; color: #fff}
      ul.percentage li.hate { background: #000; color: #fff}
    </style>
  </head>
  <body>
    <h1>Socket.IO Twitter Example</h1>
    <ul class="percentage">
      <li class="love">
        Love <span>0</span>
      </li>
      <li class="hate">
        Hate <span>0</span>
      </li>
    </ul>
    <ul class="tweets"></ul>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
➥min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var socket = io.connect();
      jQuery(function ($) {
        var tweetList = $('ul.tweets'),
            loveCounter = $('li.love'),
            hateCounter = $('li.hate'),
            loveCounterPercentage = $('li.love span'),
            hateCounterPercentage = $('li.hate span');
        socket.on('tweet', function (data) {
          loveCounter
            .css("width", data.love + '%');
          loveCounterPercentage
            .text(Math.round(data.love * 10) / 10 + '%');
          hateCounter
            .css("width", data.hate + '%');
          hateCounterPercentage
            .text(Math.round(data.hate * 10) / 10 + '%');
          tweetList
            .prepend('<li>' + data.user + ': ' + data.text + '</li>');
        });
      });
    </script>
  </body>
</html>
```

5. Install the dependencies by running the following from a terminal:

```
npm install
```

6. Start the server by running the following from a terminal:

```
node app.js
```

7. Open a browser window at http://127.0.0.1:3000.
8. You should see a stream of tweets in your browser, along with a real-time graph resizing based on data received (see Figure 14.8).
9. Kill the server by pressing Ctrl+C in the terminal.

**FIGURE 14.8**
A real-time graph

The application that you created provides a visual representation of whether there is more love than hate in the world based on real-time data from Twitter. Granted, this is totally unscientific, but it showcases the capabilities of Node.js and Socket.IO to receive large amounts of data and push it out to the browser. With a little more CSS work, the application can be styled to look better (see Figure 14.9).

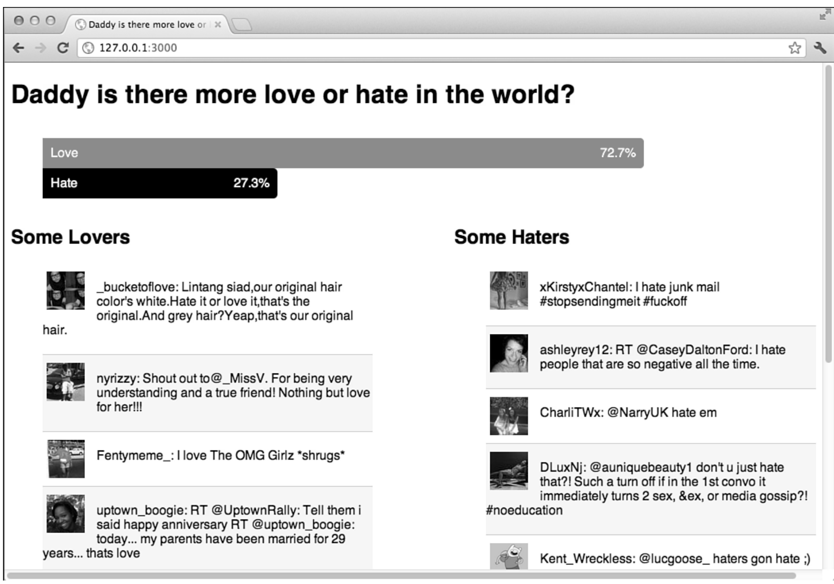



**FIGURE 14.9**
The finished application with additional styling

If you want to run this example yourself, this version is available in the code for this book as hour14/example06.

# Summary

In this hour, you answered a fundamental question about human nature using Node.js, Twitter, and Socket.IO. Not bad for an hour's work! At the time of writing, there is more love in the world, so if you take nothing else from this hour, rejoice! You learned how a Node.js server can receive large amounts of data from a third-party service and push it out to the browser in real-time using Socket.IO. You saw how to manipulate the data to extract meaning from it and perform simple calculations on the data to extract percentage values. Finally, you added some client-side JavaScript to receive the data and create a real-time graph. This hour showcased many of the strengths of Node.js, including the ease that data can be sent between the server and browser, the ability to process large amounts of data, and the strong support for networking.

# Q&A

**Q. Are there other streaming APIs that I can use to create applications like this?**

**A.** Yes. An increasing number of streaming APIs is becoming available to developers. At the time of writing, some APIs of interest include Campfire, Salesforce, Datasift, and Apigee, with many more expected to be created.

**Q. How accurate is this data?**

**A.** Not very. This data is based on the “statuses/filter” method from Twitter’s streaming API. More information about what goes into this feed is available at https://dev.twitter.com/docs/streaming-api/methods. In short, do not base any anthropological studies on it.

**Q. Can I save this data somewhere?**

**A.** The application created in this hour does not persist data anywhere, so if the server is stopped, the counters and percentages are reset. Clearly, the longer that data can be collected, the more accurate the results. The application could be extended to store the counters with a data store that can handle high volumes of writes, like Redis. This is outside the scope of this hour, though!

# Workshop

This workshop contains quiz questions and exercises to help cement your learning in this hour.

## Quiz

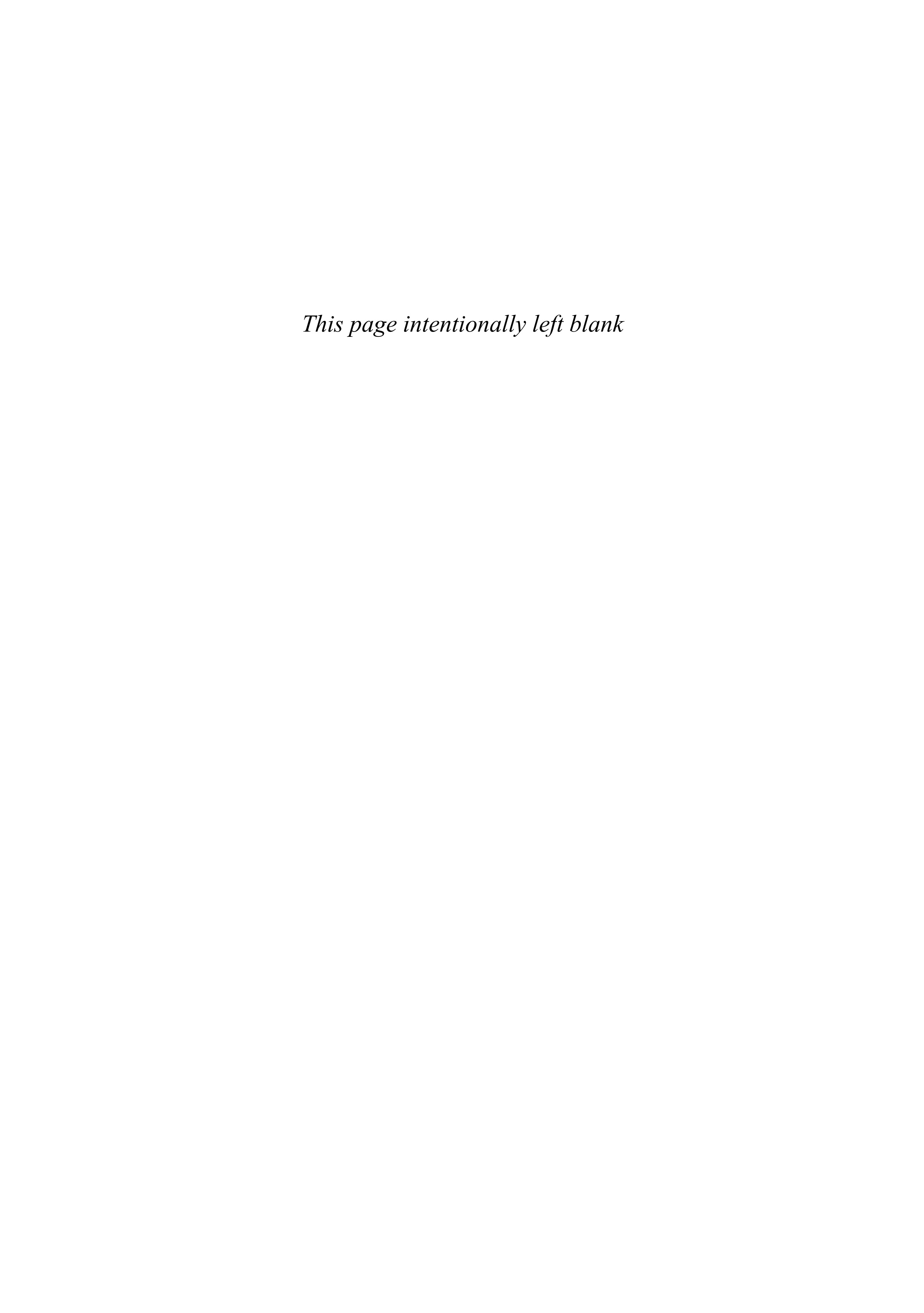
1. What is different about a streaming API?
2. What is OAuth?
3. Why is Node.js a good fit for working with streaming APIs?

## Quiz Answers

1. A streaming API keeps the connection between client and server open and is able to push new data to the client when it becomes available. This enables applications to become real-time as data is pushed to the client as soon as it is available.
2. OAuth is a way for applications to grant access to data without exposing user credentials. Authorization is granted on a per-application basis and can be revoked at any time. If you have connected your Twitter account with any other services, you may be familiar with allowing other services to access your data. OAuth is used to achieve this.
3. As Node.js is designed around evented I/O, it can respond very well to new data being received from a streaming API. It can handle large amounts of data without needing huge amounts of memory. Because Node.js is JavaScript, it is easy to communicate with clients like browsers that understand JSON. Node.js is able to receive, process, and transmit large numbers of data events without needing many machines to process it on.

# Exercises

1. Amend the example available in this book's code examples, as hour14/example02, to display the user's real name and URL. Consult the data structure earlier in this hour to understand which attributes you need for this.
2. Amend the server to receive data from Twitter's streaming API based on some keywords that you are interested in. If there are more than two keywords, update the application to show more than two bars on the graph.
3. Think about how you could create an application to provide visualizations of different streaming Twitter datasets. Remember that you can limit your query by location, certain users, and keywords. Some examples to get you started:
    - Do people talk more about beer or wine in London?
    - How often do famous people use the words "me" or "you"?
    - Are the Beatles more popular than the Rolling Stones?


*This page intentionally left blank*

# Index

## A

**access control with Middleware, 406-414**
access, limiting by IP address, 407-409
forcing users onto single domain, 410-414
**adding**
Backbone.js views, 425-429
executables to modules, 387-389
HTTP headers, 60-62
**Ajax (Asynchronous JavaScript and XML), 190**
**Allamaraju, Subbu, 37**
**analyzing Twitter data, 252-262**
**APIs, 265-266**
streaming APIs, 237-238
**appending to buffers, 340-342**
**arguments, passing to scripts, 298-300**
**ASCII (American Standard Code for Information Exchange), 334-335**
**Ashkenas, Jeremy, 361**
**assert module, 152-155**
**asynchronous code, 50-53**
testing, 157
**authentication, OAuth, 241**

## B

**Backbone.js, 417-418**
operability with Node.js, 418-422
records, creating, 428-430
views, 425-429
**BDD (Behavior Driven Development), 159-167**
**bi-directional data, 202-208**
**binary data, 333-334**
converting to text, 334-335
**bits, 106, 334**
**Blagovest, Dachev, 20**
**blocking code, 50-53**
**Brandzeg, Eirik, 21**

**broadcasting data to cients (Socket.IO), 197-203**
**Buffer module, 333-337**
- objects, 335-334

**buffers, 338-339**
- appending to, 340-342
- copying, 341
- strings, modifying, 343
- writing to, 340

**bugs, 135**

## C

**callbacks, 41-49**
- in chat server application, 221-225

**characters, encoding, 335-334**
**chat server application**
- callbacks, 221-225
- messaging, 229-231
- nicknames, adding, 214-231

**child processes, 305-307**
- communicating with, 308-310
- killing, 308-309

**classes, CoffeeScript, 371-376**
**clients**
- HTTP, 68

**client-side JavaScript, 8**
**Cloud Foundry, 175-179**
- JSON API, creating, 275-285

**Cluster module, 311-312**
**code execution, 34**
- callbacks, 41-45
- synchronous code, 50-53

**CoffeeScript, 361-363**
- classes, 371-376
- comparisons, 368
- conditions, 368
- debugging, 376-377
- features of, 366-376
- Heredocs, 371-372
- inheritance, 371-376
- installing, 362-364
- loops, 368-370
- numbers, cubing, 366
- objects, 371
- reactions to, 377
- strings, 370

**Comet, 190**
**CommonJS, 384**
**communicating with child processes, 308-310**
**companies using Node.js, 8**
**comparisons (CoffeeScript), 368**
**compiling CoffeeScript to JavaScript, 365-364**
**concatenating data from streams, 350**
**concurrency, 37**
**conditions**
- CoffeeScript, 368
- Jade, 84-86

**Connect, Middleware, 400-405**
**connecting**
- to MongoDB, 113-114
- Socket.IO client and server, 191-194

**consuming JSON data, 270-274**
**Continuous Integration servers, Travis CI, 392-395**
**converting**
- binary data to text, 334-335
- JavaScript objects to JSON, 268-271

**copying buffers, 341**
**create view (MongoDB), 118-121**
**creating**
- Backbone.js records, 428-430
- child processes, 305-307
- Express sites, 74-76
- HTML clients, 68
- JSON API, 275-285
- JSON from JavaScript objects, 268-271
- lovehateometer, 252-262
- modules, 381
- scripts, 297

**Crockford, Douglas, 266**
**cubing numbers**
- in CoffeeScript, 366
- in JavaScript, 367

**Cucumber, 29**
**cURL, 62-65**
**custom headers, adding with Middleware, 404**

## D

**Dahl, Ryan, 7**
**databases, 107-109**
- MongoDB, 109-131
  - connecting to, 113-114
  - create view, 118-121
  - documents, defining,
  - edit view, 120-125

index view, 117-119
installing, 110-112
Twitter Bootstrap, 116-117
NoSQL, 109
relational databases, 107-109
**debugging, 135, 173-174**
in CoffeeScript, 376-377
Node Inspector, 144-147
Node.js debugger, 141-143
*See also* testing
STDIO module, 136-140
**declaring classes in CoffeeScript, 373**
**defining**
mixins, 87-89
MongoDB documents,
page structure in Jade, 79-82
**deleting tasks (MongoDB), 126-127**
**dependencies, specifying, 23-24**
**deployment**
Cloud Foundry, 175-179
Heroku, 175
preparing applications for, 173-174
hosting in the cloud, 169-171
Nodester, 180-183
**developing modules, 385-387**
**DHTML (Dynamic HyperText Markup Language), 190**
**discovering process id, 291-293**
**documentation for modules, 22-23**

## E

**edit view (MongoDB), 120-125**
**emitting events, 321-320**
**encoding, 335-334**
buffers, 338-339
**environment variables, reading, 106-108**
**event loops, 53-54**
**event-driven programming, 34**
**events**
emitting, 321-320
firing, 318-320
HTTP, 320-322
listeners, 325-327
pingpong, playing, 324-325
**Events module, 317**
**examples**
of Middleware, 400
of Socket.IO, 191-194
**executables, adding to modules, 387-389**
**exiting Process module, 293**
**Express, 73**
GET requests, specifying, 92-93
installing, 74
Jade, 77-89
conditions, 84-86
includes, 87-88
inline JavaScript, 84
loops, 83-85
mixins, 87-89
page structure, defining, 79-82
variables, 82
JSON data, serving, 277-278
local variables, 99-101
parameters, 94-96
POST requests, specifying, 94-95
reasons for using, 73-74
routing, 91
organization, 96-97
Sinatra, 74
sites, creating, 74-76
and Socket.IO, 213-214
structure, 75-77
view rendering, 97-98
**extracting meaning from Twitter data, 243-247**

## F

**features of CoffeeScript, 366-376**
**files**
reading data from, 105-106
reading with streams, 349
writing data to, 104
**finding modules, 19-21**
**Firefox, Live HTTP Headers add-on, 62-64**
**firing events, 318-320**
**flash messages, 126-130**
**folders**
Express, 75-77
in modules, 384-385

**forcing users onto single domain, 410-414**

**fork() method, 308**

**functions**

- callbacks, 41-49
- event loops, 53-54

## G

**-g flag, 74**

**GET requests, 91**

- specifying, 92-93

**GitHub, 391-392**

**global module installation, 22**

**Gmail, 190**

**Google Chrome, 7**

- HTTP Headers Extension, 62-63

## H

**handling unpredictability, 37**

**headers (HTTP), adding, 60-62**

**Hello World program, 10**

**Heredocs, 371-372**

**Heroku**

- deploying applications to, 174-175
- preparing applications for, 173-174

**hosting in the cloud, 169-171**

**HTML clients, creating, 68**

**HTTP (HyperText Transfer Protocol), 59**

- clients, 68
- events, 320-322
- GET requests, 92-93
- headers, adding, 60-62
- POST requests, 94-95
- requests, responding to, 66-69
- response headers, 62-65
- routing, 91
  - organization, 96-97
  - parameters, 94-96
- verbs, 91

**HTTP Headers Extension, 62-63**

## I

**includes, Jade, 87-88**

**indentation, 78-79**

**index view (MongoDB), 117-119**

**inheritance, 371-376**

**inline JavaScript, 84**

**input, 29-32**

**input data, validating, 130-131**

**installing**

- CoffeeScript, 362-364
- Express, 74
- modules, 17
  - global installation, 22
  - local installation, 21-22
- MongoDB, 110-112
- Node.js, 9-10

**I/O (input/output), 27-28**

- input, 29-32
- unpredictability of, 33-34

## J-K

**Jade, 77-89**

- conditions, 84-86
- includes, 87-88
- inline JavaScript, 84
- loops, 83-85
- mixins, 87-89
- outputting data, 82-89
- page structure, defining, 79-82
- variables, 82

**JavaScript**

- numbers, cubing, 367
- objects, converting to JSON, 268-271

**jsconf.eu, 7**

**JSON (JavaScript Object Notation), 266-268**

- creating from JavaScript objects, 268-271
- data
  - consuming, 270-274
  - sending, 268

## L


**layout files (Express), 97-98**

**lightweight frameworks, 73**

**limiting access by IP address, 407-409**

**listening for events, 321-320**

**Live HTTP Headers add-on, 62-64**

**local module installation, 21-22**

**local variables (Express), 99-101**

**locating modules, 19-21**

**loops**

- CoffeeScript, 368-370
- in Jade, 83-85

**lovehateometer, creating, 252-262**


## M


**managing events dynamically, 325-327**

**McConnell, Charlie, 241**

**messaging**

- in chat server application, 229-231
- in Socket.IO, 202-208

**Middleware, 399-400**

- access control, 406-414
  - forcing users onto single domain, 410-414
  - limiting access by IP address, 407-409
- custom headers, adding, 404

**mixins, Jade, 87-89**

**Mocha, 163-166**

**modifying strings in buffers, 343**

**modules, 15-16, 381-383**

- assert module, 152-155
- Buffer module, 333-337
- Cluster module, 311-312
- creating, 381
- developing, 385-387
- documentation, 22-23
  - locating, 22-23
- Events module, 317
- executables, adding, 387-389
- folder structure, 384-385
- HTTP, 59-69
- installing, 17
  - global installation, 22
  - local installation, 21-22
- locating, 19-21
- Process module, 291-293
  - exiting, 293
- publicizing, 397
- requiring, 17-18
- Socket.IO, 189-191
  - bi-directional data, 202-208
  - data, broadcasting to clients, 197-203
  - data, sending from server to clients, 192-198
  - example of, 191-194
  - and Express, 213-214
- STDIO, 136-140
- Stream module, 345-346
  - MP3s, streaming, 354-355
- testing, 385-387
- third-party, 18
- URL, 67
- websites, 381-383

**MongoDB, 109-131**

- connecting to, 113-114
- create view, 118-121
- edit view, 120-125
- flash messages, 126-130
- index view, 117-119
- input data, validating, 130-131
- installing, 110-112
- tasks, deleting, 126-127
- Twitter Bootstrap, 116-117

**MP3s, streaming, 354-355**


## N


**network programming frameworks, 54**

**networked I/O, unpredictability of, 33-34**

**nibbles, 334**

**nicknames, adding (chat server application), 214-231**

**Node Inspector, 144-147**

**Node.js, installing, 9-10**

**Node.js debugger, 141-143**

**Nodester, 180-183**
**Nodeunit, 156-157**
**non-blocking code, 50-53**
**NoSQL databases, 109**
**npm (Node Package Manager), 15-16**
Express, installing, 74
installing, 16
publishing to, 395-396
**numbers, cubing, 367**

# O

**OAuth, 241**
**object-oriented programming, 390-391**
**objects, CoffeeScript, 371**
**official module sources, 19-20**
**Olson, Rick, 241**
**organization of routes, 96-97**
**output, 28**

# P

**PaaS (Platform as a Service), 170, 184**
Cloud Foundry, 175-179
Heroku, 173-175
Nodester, 180-183
**package managers, npm, 15-16**
**package.json, 23-24, 383-384**
**page structure**
Jade, 79-82
**parameters (Express), 94-96**
**parsing**
JSON data, 270-274
Twitter data, 243-247
**passing arguments to scripts, 298-300**
**persistent data, 103**
files
reading data from, 105-106
writing data to, 104
**piping streams, 353-354**
**playing pingpong with events, 324-325**
**popular Node.js modules, 381-383**
**POST requests, 91**
specifying, 94-95
**pre-compilers, 364**
**preparing applications for Heroku, 173-174**
**printing stack traces, 140**
**process id, discovering, 291-293**
**Process module, 291-293**
exiting, 293
**processes, 291-293**
child processes, 305-307
communicating with, 308-310
killing, 308-309
signals, 293-295
**programming**
event listeners, 325-327
event-driven, 34
object-oriented, 390-391
prototype-based, 390-391
**prototype-based programming, 390-391**
**publicizing modules, 397**
**publishing to npm, 395-396**
**pushing**
Twitter data to browser, 246-251

# Q-R

**ql.io, 37**
**reactions to CoffeeScript, 377**
**readable streams, 347-351**
piping data from writable streams, 353-354
**reading**
data from files, 105-106
environment variables, 106-108
files, 349
**records (Backbone.js), creating, 428-430**
**redirects, 62**
**relational databases, 107-109**
**requests (HTTP), responding to, 66-69**
**requiring modules, 17-18**
**responding to HTTP requests, 66-69**
**response headers (HTTP), 62-65**
**routes, 77**
**routing, 66, 91**
in Express, 91
organization, 96-97
parameters, 94-96

## S

**scripts**
- creating, 297
- passing arguments to, 298-300

**sending**
- data with JSON, 268
- signals to processess, 295-297

**server-side JavaScript, 8**

**sharing code with GitHub, 391-392**

**shebangs, 297**

**signals, 293-295**
- sending to processes, 295-297

**signing up for Twitter, 238-241**

**Sinatra, 74**

**sites (Express), creating, 74-76**

**Socket.IO, 191**
- bi-directional data, 202-208
- chat server application, 213-214
  - callbacks, 221-225
  - nicknames, adding, 214-231
- data
  - broadcasting to clients, 197-203
  - sending from server to clients, 192-198
- example of, 191-194
- and Express, 213-214

**specifying**
- dependencies, 23-24
- GET requests, 92-93
- POST requests, 94-95

**stack traces, 140**

**STDIO module, 136-140**

**stepping through code, 141-143**

**Stream module, 345-346**
- MP3s, streaming, 354-355

**streaming APIs, 237-238**
- Twitter, 241-244
  - data, extracting meaning from, 243-247
  - data, pushing to browser, 246-251
  - lovehateometer, creating, 252-262

**streams**
- concatenating data from, 350
- piping, 353-354
- readable, 347-351
- writable, 351

**strings**
- CoffeeScript, 370
- modifying in buffers, 343

**structure**
- of Express, 75-77
- of module folders, 384-385

**subclassing (CoffeeScript), 373-375**

**synchronous code, 50-53**

## T

**tasks, MongoDB, 118-121**
- deleting, 126-127

**TDD (Test Driven Development), 151-152**

**template engines, Jade, 77-89**

**terminating child processes, 308-309**

**testing, 147-148, 151-152**
- assert module, 152-155
- BDD, 159-167
- JSON API, 282-285
- Mocha, 163-166
- modules, 385-387
- TDD (Test Driven Development), 151-152
- third-party testing tools, 155-157
- Vows, 159-162

**text, converting binary data to, 334-335**

**third-party modules, 18**

**third-party testing tools, 155-157**

**Travis CI, 392-395**

**Twitter**
- data, extracting meaning from, 243-247
- data, pushing to browser, 246-251
- lovehateometer, creating, 252-262
- signing up for, 238-241
- streaming API, 241-244

**Twitter Bootstrap, 116-117**

## U

Underscore, 382
unofficial module sources, 20-21
unpredictability, handling, 37
unpredictability of networked I/O, 33-34
URL module, 67
UTF-8, 106

## V

V8, 7
validating input data, 130-131
variables, Jade, 82
verbs (HTTP), 91
verifying
  Node.js installation, 9-10
  npm installation, 16
view rendering, 97-98
views (Backbone.js), 425-429
views folder (Express), 77
Vows, 159-162

## W-Z

Waugh, Jeff, 241
websites, popular Node.js modules, 381-383
WebSockets, 190-191
writable streams, 351
  piping data from readable streams, 353-354
writing
  data to buffers, 340
  data to files, 104