www.it-ebooks.info

www.it-ebooks.info

**PROFESSIONAL NODE.JS®**

**INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .**xxvii

**PART I INTRODUCTION AND SETUP**

**CHAPTER 1** Installing Node . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3 **CHAPTER 2** Introducing Node . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .15

**PART II NODE CORE API BASICS**

**CHAPTER 3** Loading Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23 **CHAPTER 4** Using Buff ers to Manipulate, Encode, and Decode Binary Data . . . . . . 29 **CHAPTER 5** Using the Event Emitter Pattern to Simplify Event Binding . . . . . . . . . . 35 **CHAPTER 6** Scheduling the Execution of Functions Using Timers . . . . . . . . . . . . . . . 45

**PART III FILES, PROCESSES, STREAMS, AND NETWORKING CHAPTER 7** Querying, Reading from, and Writing to Files . . . . . . . . . . . . . . . . . . . . . . 53 **CHAPTER 8** Creating and Controlling External Processes . . . . . . . . . . . . . . . . . . . . . . 63 **CHAPTER 9** Reading and Writing Streams of Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75 **CHAPTER 10** Building TCP Servers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83 **CHAPTER 11** Building HTTP Servers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95 **CHAPTER 12** Building a TCP Client . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103 **CHAPTER 13** Making HTTP Requests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113 **CHAPTER 14** Using Datagrams (UDP) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129 **CHAPTER 15** Securing Your TCP Server with TLS/SSL . . . . . . . . . . . . . . . . . . . . . . . . . 139 **CHAPTER 16** Securing Your HTTP Server with HTTPS . . . . . . . . . . . . . . . . . . . . . . . . . 149

**PART IV BUILDING AND DEBUGGING MODULES AND APPLICATIONS CHAPTER 17** Testing Modules and Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157 **CHAPTER 18** Debugging Modules and Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . 167 **CHAPTER 19** Controlling the Callback Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .177

Continues

www.it-ebooks.info

**PART V BUILDING WEB APPLICATIONS**

**CHAPTER 20** Building and Using HTTP Middleware . . . . . . . . . . . . . . . . . . . . . . . . . . . 197 **CHAPTER 21** Making a Web Application Using Express.js . . . . . . . . . . . . . . . . . . . . . . 217 **CHAPTER 22** Making Universal Real-Time Web Applications Using Socket.IO . . . . 241

**PART VI CONNECTING TO DATABASES**

**CHAPTER 23** Connecting to MySQL Using node-mysql . . . . . . . . . . . . . . . . . . . . . . . . 267 **CHAPTER 24** Connecting to CouchDB Using Nano . . . . . . . . . . . . . . . . . . . . . . . . . . . . 277 **CHAPTER 25** Connecting to MongoDB Using Mongoose . . . . . . . . . . . . . . . . . . . . . . . 311

**INDEX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .**351 www.it-ebooks.info

**PROFESSIONAL Node.js**®

www.it-ebooks.info

www.it-ebooks.info

**PROFESSIONAL**

**Node.js**®

**BUILDING JAVASCRIPT-BASED SCALABLE SOFTWARE** Pedro Teixeira

www.it-ebooks.info

**Professional Node.js®: Building JavaScript-Based Scalable Software**

Published by

John Wiley & Sons, Inc.

10475 Crosspoint Boulevard

Indianapolis, IN 46256

www.wiley.com

Copyright © 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-18546-9

ISBN: 978-1-118-22754-1 (ebk)

ISBN: 978-1-118-24056-4 (ebk)

ISBN: 978-1-118-26518-5 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

**Limit of Liability/Disclaimer of Warranty:** The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifi cally disclaim all warranties, including without limitation warranties of fi tness for a particular purpose. No warranty may be created or extended by sales or

promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

**Library of Congress Control Number: 2012940020**

**Trademarks:** Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc., and/or its affi liates, in the United States and other coun tries, and may not be used without written permission. Node.js is a registered trademark of Joyent, Inc. All other trade marks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

www.it-ebooks.info

*This book is dedicated to my wife, Susana. Throughout all these years she has always been an example of strength and persistence.*

www.it-ebooks.info

www.it-ebooks.info

**ABOUT THE AUTHOR**

**PEDRO TEIXEIRA** is a prolifi c open-source programmer and author of many Node.js modules. After graduating with a degree in Software Engineering more than 14 years ago, he has been a consultant, a programmer, and an active and internationally known Node.js community member.

He is a founding partner of The Node Firm and a Senior Programmer at Nodejitsu Inc., the leading Node.js platform-as-a-service provider. He is also the author of the popular Node Tuts screencasts.

When Pedro was 10 years old, his father taught him how to program a ZX Spectrum, and since then he has never wanted to stop. He taught himself how to program his father’s Apple IIc and then entered the PC era. In college he was introduced to the universe of UNIX and open-source, becom ing seriously addicted to it. In his professional life he has developed systems and products built with Visual Basic, C, C++, Java, PHP, Ruby, and JavaScript for big telecommunications companies, banks, hotel chains, and others.

He has been a Node.js enthusiast since its initial development, having authored many applications and many well-known modules like Fugue, Alfred.js, Carrier, Nock, and more.

www.it-ebooks.info

www.it-ebooks.info

**ABOUT THE TECHNICAL EDITOR**

**MANUEL KIESSLING** is a software development and systems administration team lead, using and teaching agile practices in both domains. He runs several open-source projects, is an active blogger, and wrote the freely available Node Beginner Book. He currently lives near Cologne, Germany, with his wife and two children.

He is the co-author of Chapter 22, “Making Universal Real-Time Web Applications Using Socket.IO,” and Chapter 23, “Connecting to MySQL Using node-mysql.”

www.it-ebooks.info

www.it-ebooks.info

**CREDITS**

**ACQUISITIONS EDITOR**

Mary James

**PROJECT EDITOR**

Tora Estep

**TECHNICAL EDITOR**

Manuel Kiessling

**PRODUCTION EDITOR**

Daniel Scribner

**COPY EDITOR**

Kezia Endsley

**EDITORIAL MANAGER**

Mary Beth Wakefi eld

**FREELANCER EDITORIAL MANAGER**

Rosemarie Graham

**ASSOCIATE DIRECTOR OF MARKETING**

David Mayhew

**MARKETING MANAGER**

Ashley Zurcher

**BUSINESS MANAGER**

Amy Knies

www.it-ebooks.info

**PRODUCTION MANAGER**

Tim Tate

**VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER**

Richard Swadley

**VICE PRESIDENT AND EXECUTIVE PUBLISHER**

Neil Edde

**ASSOCIATE PUBLISHER**

Jim Minatel

**PROJECT COORDINATOR, COVER** Katie Crocker

**PROOFREADER**

Nicole Hirschman

**INDEXER**

Ron Strauss

**COVER DESIGNER**

Ryan Sneed

**COVER IMAGE**

© Gregory Olsen/iStockPhoto

www.it-ebooks.info

**ACKNOWLEDGMENTS**

**FIRST, I WANT TO THANK** my wife, Susana, and my kids, Henrique and Beatriz. You have given me courage and persistence by reminding me what is really important in life.

I’d also like to thank my Acquisitions Editor at Wiley, Mary E. James, for having faith in me and signing me up to write this book.

I want to thank my good friend and former colleague, programmer and photographer Pedro Mendes, who persuaded me to start writing a book about Node.js over a few beers in Lisbon.

I also want to thank my good friend Nuno Job, who has been my companion in this open-source and Node.js crusade for these last few years.

A big word of appreciation also goes out to the Node.js community in general. You are the best, most welcoming, appreciative, and fun programming community I have been a part of.

Lastly, I want to thank you, the reader, for buying this book. I hope it helps you in learning about the ins and outs of the wonderful world of programming in Node.js.

www.it-ebooks.info

www.it-ebooks.info

**CONTENTS**

**INTRODUCTION xxvii**

**PART I: INTRODUCTION AND SETUP**

**CHAPTER 1: INSTALLING NODE 3**

**Installing Node on Windows 4 Installing on Mac OS X 5 Installing Node Using the Source Code 6**

Choosing the Node Version 6 Downloading the Node Source Code 6 Building Node 7 Installing Node 7 Running Node 8

**Setting Up and Using Node Package Manager 8** Using NPM to Install, Update, and Uninstall Packages 9 **Summary 13**

**CHAPTER 2: INTRODUCING NODE 15**

**Introducing the Event-Driven Programming Style 16 How Node and JavaScript Make Writing Asynchronous**

**Applications Easier 17** What Are Closures? 18 How Closures Help When Programming Asynchronously 19 **Summary 19**

**PART II: NODE CORE API BASICS**

**CHAPTER 3: LOADING MODULES 23**

**Understanding How Node Loads Modules 24 Exporting a Module 24** Loading a Module 25 **Summary 28**

www.it-ebooks.info

**CONTENTS**

**CHAPTER 4: USING BUFFERS TO MANIPULATE, ENCODE,**

**AND DECODE BINARY DATA 29**

**Creating a Buff er 30 Getting and Setting Bytes in a Buff er 30 Slicing a Buff er 32 Copying a Buff er 32 Decoding a Buff er 32 Summary 33**

**CHAPTER 5: USING THE EVENT EMITTER PATTERN**

**TO SIMPLIFY EVENT BINDING 35**

**Understanding the Standard Callback Pattern 36 Understanding the Event Emitter Pattern 36 Understanding Event Types 37 Using the Event Emitter API 38**

Binding Callbacks Using .addListener() or .on() 38 Binding Multiple Event Listeners 39 Removing an Event Listener from an Event Emitter

Using .removeListener() 40 Getting a Callback Executed at Most Once Using .once() 40 Removing All Event Listeners from an Event Emitter

Using .removeAllListeners() 41 **Creating an Event Emitter 41** Inheriting from Node Event Emitter 42 Emitting Events 42 **Summary 43**

**CHAPTER 6: SCHEDULING THE EXECUTION**

**OF FUNCTIONS USING TIMERS 45**

**Using setTimeout to Defer the Execution of a Function 46 Using clearTimeout to Cancel the Execution of a Function 46 Scheduling and Canceling the Repetitive Execution of a Function 47 Using process.nextTick to Defer the Execution**

**of a Function Until the Next Event Loop Iteration 47 Blocking the Event Loop 48 Escaping the Event Loop 49 Using setTimeout Instead of setInterval**

**to Force Serialization 49 Summary 50**

**xviii**

www.it-ebooks.info

**CONTENTS**

**PART III: FILES, PROCESSES, STREAMS, AND NETWORKING**

**CHAPTER 7: QUERYING, READING FROM, AND WRITING TO FILES 53**

**Manipulating File Paths 54** Normalizing Paths 54 Joining Paths 54 Resolving Paths 55 Finding the Relative Path Between Two Absolute Paths 55 Extracting Components of a Path 55 Determining the Existence of a Path 56

**Introducing the fs Module 57** Querying File Statistics 57 **Opening a File 58 Reading from a File 59** Writing to a File 60 Closing a File 60 **Summary 62**

**CHAPTER 8: CREATING AND CONTROLLING**

**EXTERNAL PROCESSES 63**

**Executing External Commands 64 Spawning Child Processes 69** Creating the Child Process 69 Listening for Data from the Child Process 69 Sending Data to the Child Process 70 Receiving Notifi cation When the Child Process Exits 72 **Signaling and Killing Processes 73 Summary 74**

**CHAPTER 9: READING AND WRITING STREAMS OF DATA 75**

**Using a Readable Stream 76** Waiting for Data 76 Pausing and Resuming a Stream 77 Knowing When the Stream Ends 77

**Using Writable Streams 77** Writing Data into a Stream 78 Waiting for a Stream to Drain 78

**Considering Some Stream Examples 78** Creating File-System Streams 79 Understanding Networking Streams 80

**xix**

www.it-ebooks.info

**CONTENTS**

**Avoiding the Slow Client Problem and Saving Your Server 80** Understanding the Slow Client Problem 80 Avoiding the Slow Client Problem 81 Using stream.pipe() to Prevent the Slow Client Problem and Assembling

Readable and Writable Streams Using pipe() 82 **Summary 82**

**CHAPTER 10: BUILDING TCP SERVERS 83**

**Creating a TCP Server 83** Using the Socket Object 85 Understanding Idle Sockets 86 Setting Up Keep-Alive 87 Using Delay or No Delay 87 Listening for Client Connections 88 Closing the Server 88 Handling Errors 88

**Building a Simple TCP Chat Server 89** Accepting Connections 89 Reading Data from a Connection 90 Collecting All the Clients 90 Broadcasting Data 91 Removing Closed Connections 92 Using Your TCP Chat Server 93 **Summary 93**

**CHAPTER 11: BUILDING HTTP SERVERS 95**

**Understanding the http.ServerRequest Object 97 Understanding the http.ServerResponse Object 98** Writing a Header 98 Changing or Setting a Header 99 Removing a Header 99 Writing a Piece of the Response Body 99 **Streaming HTTP Chunked Responses 99** Piping a File 100 Piping the Output of Another Process 100 **Shutting Down the Server 101 Example 1: Building a Server that Serves Static Files 101 Example 2: Making Use of HTTP Chunked**

**Responses and Timers 102 Summary 102**

**xx**

www.it-ebooks.info

**CONTENTS**

**CHAPTER 12: BUILDING A TCP CLIENT 103**

**Connecting to a Server 104 Sending and Receiving Data 105 Ending the Connection 105 Handling Errors 106 Building an Example Command-Line TCP Client 106**

Connecting to the Server 107 Sending the Command Line to the Server 107 Printing Server Messages 107 Reconnecting if the Connection Dies 108 Closing the Connection 109 Putting the Client Together 111 **Summary 112**

**CHAPTER 13: MAKING HTTP REQUESTS 113**

**Making GET Requests 113 Using Other HTTP Verbs 114** Inspecting the Response Object 115 Obtaining the Response Body 116 Streaming the Response Body 116 **Pooling Sockets Using http.Agent 116 Using a Third-Party Request Module to Simplify**

**HTTP Requests 118** Installing and Using Request 118 Creating a Testing Server 120 Following Redirects 121 Setting Some Request Options 123 Encoding the Request Body 125 Streaming 127 Using a Cookie Jar 127 **Summary 128**

**CHAPTER 14: USING DATAGRAMS (UDP) 129**

**Understanding UDP 129 Understanding the Uses of UDP 130 Building a Datagram Server 130**

Listening for Messages 130 Testing the Server 131 Inspecting Additional Message Information 132 **Creating a Simple Datagram Echo Server 132**

**xxi**

www.it-ebooks.info

**CONTENTS**

Waiting for Messages 132 Sending Messages Back to Senders 132 Putting the Echo Server Together 133

**Building a Datagram Client 134** Creating the Client 134 Sending Messages 134 Closing the Socket 134

**Creating a Simple Datagram Command-Line Client 135** Reading from the Command Line 135 Sending Data to the Server 135 Receiving Data from the Server 136 Putting the Command-Line UDP Client Together 136

**Understanding and Using Datagram Multicast 136** Receiving Multicast Messages 137 Sending Multicast Messages 138 Understanding Maximum Datagram Size 138 **Summary 138**

**CHAPTER 15: SECURING YOUR TCP SERVER WITH TLS/SSL 139**

**Understanding Private and Public Keys 139** Generating a Private Key 140 Generating a Public Key 140

**Building a TLS Server 141** Initializing the Server 141 Listening for Connections 141 Reading Data from the Client 142 Sending Data to the Client 142 Ending the Connection 142

**Building a TLS Client 143** Initializing the Client 143 Connecting to the Server 143 Verifying the Server Certifi cate 143 Sending Data to the Server 144 Reading Data from the Server 144 Ending the Connection 144

**Building Some Examples 145** Creating a TLS Chat Server 145 Creating a TLS Command-Line Chat Client 146 Verifying the Client Certifi cate 147 **Summary 148**

**xxii**

www.it-ebooks.info

**CONTENTS**

**CHAPTER 16: SECURING YOUR HTTP SERVER WITH HTTPS 149**

**Building a Secure HTTP Server 149** Setting Up the Server Options 150 Listening for Connections 150 Validating the HTTPS Client Certifi cate 151

**Creating an HTTPS Client 152** Initializing the Client 152 Making the Request 152 Validating the HTTPS Server Certifi cate 153 **Summary 154**

**PART IV: BUILDING AND DEBUGGING MODULES AND APPLICATIONS CHAPTER 17: TESTING MODULES AND APPLICATIONS 157**

**Using a Test Runner 157** Writing Tests 158 Running Tests 159

**Using an Assertion Testing Module 159** Using the assert Module 159 Using the Built-in Assertion Functions in Node-Tap 161

**Testing Your Asynchronous Module 163 Summary 166**

**CHAPTER 18: DEBUGGING MODULES AND APPLICATIONS 167**

**Using console.log 167 Using Node’s Built-in Debugger 169 Using Node Inspector 173 Summary 175**

**CHAPTER 19: CONTROLLING THE CALLBACK FLOW 177**

**Understanding the Boomerang Eff ect 177 Avoiding the Boomerang Eff ect**

**by Declaring Functions 179 Using the async Flow Control Library 183** Executing in Series 184 Executing in Parallel 185 Cascading 186 Queuing 187 Iterating 189 Mapping 190

**xxiii**

www.it-ebooks.info

**CONTENTS**

Reducing 191 Filtering 192 Detecting 193 **Summary 194**

**PART V: BUILDING WEB APPLICATIONS**

**CHAPTER 20: BUILDING AND USING HTTP MIDDLEWARE 197**

**Understanding the Connect HTTP Middleware Framework 198 Building Your Own HTTP Middleware 198** Creating Asynchronous Middleware 200 Registering Callbacks Inside Middleware 201 Handling Errors Inside Middleware 203 **Using the HTTP Middleware Bundled in Connect 206** Logging Requests 206 Handling Errors 208 Serving Static Files 209 Parsing the Query String 210 Parsing the Request Body 211 Parsing Cookies 212 Using a Session 213 Other Available Middleware 216 **Summary 216**

**CHAPTER 21: MAKING A WEB APPLICATION USING EXPRESS.JS 217**

**Initializing Your Express.js Application 218 Setting Up Middleware in Your Application 220 Routing Requests 222**

Handling Routes 222 Using Sessions 229 Using Route Middleware 234 **Summary 238**

**CHAPTER 22: MAKING UNIVERSAL REAL-TIME**

**WEB APPLICATIONS USING SOCKET.IO 241**

**Understanding How WebSockets Work 242 Using Socket.IO to Build WebSocket Applications 243** Installing and Running Socket.IO on the Server 243 Building a Real-Time Web Chat with Socket.IO 245 Extending the Chat Application 250 Detecting Disconnections 254

**xxiv**

www.it-ebooks.info

**CONTENTS**

Separating Users into Rooms 255 Using Namespaces 259 Distributing the Server-Side Application Using Redis 259 **Summary 263**

**PART VI: CONNECTING TO DATABASES**

**CHAPTER 23: CONNECTING TO MYSQL USING NODE-MYSQL 267**

**Using a Library to Connect to and Communicate**

**with a MySQL Database 268 Adding Data to the Database with**

**Security Concerns in Mind 270 Reading Data Effi ciently 272 Summary 276**

**CHAPTER 24: CONNECTING TO COUCHDB USING NANO 277**

**Installing Nano 278 Connecting and Creating a Database 281 Storing Documents 285 Creating and Using CouchDB Views 286 Attaching Files to a CouchDB Document 298 Summary 310**

**CHAPTER 25: CONNECTING TO MONGODB USING MONGOOSE 311**

**Installing Mongoose 313 Understanding How Mongoose Uses Models**

**to Encapsulate Database Access 313 Connecting to MongoDB 314 Defi ning a Schema 314 Defi ning a Model 315**

Using Validators 324 Using Modifi ers 330 Using Getters 331 Using Virtual Attributes 332 Using Default Values 338 Defi ning Indexes 340 Referencing Other Documents Using DB Refs 341 Defi ning Instance Methods 347 Defi ning Static Methods 348 **Summary 349**

**INDEX 351 xxv**

www.it-ebooks.info

www.it-ebooks.info

**INTRODUCTION**

**IN 1995, WHEN I WAS IN MY SECOND YEAR IN COLLEGE,** I was introduced to UNIX network program ming. In C, you could create sockets to open TCP connections to servers and code the servers that accepted these connections. I remember the excitement I felt the fi rst time I created a TCP server: I could accept connections and receive and send messages on them.

If I wanted my server to accept many concurrent connections, the common solution was to use threads, and soon I had created my fi rst multi-threaded TCP server. This server accessed a shared data structure, which needed to synchronize the access to all the client threads that had been spawned. Getting the synchronization fi ne-grained (to maximize resources and time) and right (to avoid deadlocks) proved to be more diffi cult than anticipated.

A couple of years later, I entered the working world to become a consultant, programming and leading teams of programmers to implement various client projects. At fi rst I continued to work within the UNIX world, but soon I was diverted to Java and all its enterprise fl avors and fi nally

landed on the fertile plains of web development, using scripting languages like PHP and Ruby. Doing web development, I slowly became familiar with JavaScript and the event-driven program ming model, never realizing it would later connect me back to the world of UNIX.

Fast-forwarding to early 2010, a good friend of mine talked to me about Node.js. It was fast, he said, and you can program it in JavaScript. It transported the event-driven browser programming into the UNIX network programming world.

Curious, I went to take a look at the API documents and was immediately hooked. The ease with which you could create highly scalable servers without using threads and mix-and-match client and server code made me take a deep dive into Node’s source code and surrounding modules. Node.js connected the ease of a scripting language with all the power of UNIX network programming, and I felt like I was fi nally home.

**WHO THIS BOOK IS FOR**

This book was written for the developer who is familiar with JavaScript, either browser or server side programming. The reader should be familiar with some introductory concepts of how TCP and HTTP works. For the later chapters on web application development, it also helps if the reader is familiar with classic development for the web (HTML, CSS, and JavaScript).

If you already have Node.js installed, you may skip to Chapter 2, “Introducing Node.”

If you already know the basics of how Node.js works internally and know about server-side event programming in JavaScript, you may skip to Chapter 3, “Loading Modules.”

www.it-ebooks.info

**INTRODUCTION**

After introducing core Node.js concepts and API subsets, I go into application building starting in Chapter 17 (“Testing Modules and Applications”) and cover debugging (Chapter 18, “Debugging Modules and Applications”), and I present some tips on controlling asynchronous fl ow (Chapter 19, “Controlling the Callback Flow”).

Next, I address the necessary parts of building web applications, starting in Chapter 20, “Building and Using HTTP Middleware,” passing through Express.js (Chapter 21), and creating real-time web allocations using Socket.IO (Chapter 22).

Finally, I also address how to access and use databases from Node.js, including MySQL (Chapter 23), CouchDB (Chapter 24), and MongoDB (Chapter 25).

**WHAT THIS BOOK COVERS**

This book covers Node.js v0.8, Express.js v2.5, Socket.io 0.9, Node-mysql v0.9, Nano v3.1, and Mongoose v2.7.

**HOW THIS BOOK IS STRUCTURED**

The book starts with setting up and introducing Node.js.

It then explains the Node core fundamentals, which include modules, buffers, the Event Emitter pattern, and timers, after which the fundamental Node core specifi c fi le and networking APIs are introduced and explained, all complemented with practical examples.

After covering core Node concepts, the book continues with some best practices for developing applications with Node.js, such as testing modules, debugging applications, and maintaining control of the asynchronous callback fl ow.

Building real-time web applications is one of the main use cases of Node, and this book shows you how to do it using Connect, Express.js, and Socket.IO.

Because most applications need to connect to a database, the book explains how to connect to and use MySQL, CouchDB, and MongoDB from your Node.js application.

**WHAT YOU NEED TO USE THIS BOOK**

To install and run Node.js applications, you need a PC or Macintosh computer running a recent version of either Windows, Linux, or MacOS.

The source code for the samples is available for download from the Wrox website at: www.wrox .com/remtitle.cgi?isbn=P010093766.

**xxviii**

www.it-ebooks.info

**INTRODUCTION**

**CONVENTIONS**

To help you get the most from the text and keep track of what’s happening, we’ve used a number of conventions throughout the book.

**WARNING** *Warnings hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.*

**NOTE** *Notes indicates notes, tips, hints, tricks, and asides to the current*

*discussion.*

➤ We *italicize* new terms and important words when we introduce them.

➤ We show keyboard strokes like this: Ctrl+A.

➤ We show fi le names, URLs, and code within the text like so: persistence.properties. We present code in two different ways:

We use a monofont type with no highlighting for most code examples.

**We use bold to emphasize code that is particularly important in the present context or to show changes from a previous code snippet.**

**SOURCE CODE**

As you work through the examples in this book, you may choose either to type in the code manu ally, or to use the source code fi les that accompany the book. All the source code used in this book is available for download at www.wrox.com. Specifi cally for this book, the code download is on the Download Code tab at: http://www.wrox.com/WileyCDA/WroxTitle/Professional-Node-js Building-Javascript-Based-Scalable-Software.productCd-1118185463.html.

You can also search for the book at www.wrox.com by ISBN to fi nd the code. And a complete list of code downloads for all current Wrox books is available at www.wrox.com/dynamic/books/ download.aspx.

Throughout each chapter, you’ll fi nd references to the names of code fi les as needed in listing titles and text.

Most of the code on www.wrox.com is compressed in a .ZIP, .RAR archive, or similar archive for mat appropriate to the platform. Once you download the code, just decompress it with an appropri ate compression tool.

**xxix**

www.it-ebooks.info

**INTRODUCTION**

**NOTE** *Because many books have similar titles, you may fi nd it easiest to search by ISBN; this book’s ISBN is 978-1-118-18546-9.*

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download .aspx to see the code available for this book and all other Wrox books.

**ERRATA**

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you fi nd an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher quality information.

To fi nd the errata page for this book, go to http://www.wrox.com/WileyCDA/WroxTitle/ Professional-Node-js-Building-Javascript-Based-Scalable-Software .productCd-1118185463.html and click the Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

If you don’t spot “your” error on the Book Errata page, go to www.wrox.com/contact/techsupport .shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s errata page and fi x the problem in subsequent editions of the book.

**P2P.WROX.COM**

For author and peer discussion, join the P2P forums at http://p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, edi tors, other industry experts, and your fellow readers are present on these forums.

At http://p2p.wrox.com, you will fi nd a number of different forums that will help you, not only as you read this book but also as you develop your own applications. To join the forums, just follow these steps:

**1.** Go to http://p2p.wrox.com and click the Register link.

**2.** Read the terms of use and click Agree.

**3.** Complete the required information to join, as well as any optional information you wish to provide, and click Submit.

**xxx**

www.it-ebooks.info

**INTRODUCTION**

**4.** You will receive an e-mail with information describing how to verify your account and com plete the joining process.

**NOTE** *You can read messages in the forums without joining P2P, but to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to This Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specifi c to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

**xxxi**

www.it-ebooks.info

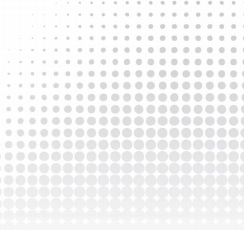
www.it-ebooks.info

PART I

**Introduction and Setup**

**CHAPTER 1:** Installing Node

**CHAPTER 2:** Introducing Node

www.it-ebooks.info

www.it-ebooks.info

**1**

**Installing Node**

**WHAT’S IN THIS CHAPTER?**

➤ Getting Node up and running

➤ Installing Node Package Manager (NPM)

➤ Using NPM to install, uninstall, and update packages

At the European JSConf in 2009, Ryan Dahl, a young programmer, presented a project he had been working on. This project was a platform that combined Google’s V8 JavaScript engine, an event loop, and a low-level I/O API. This project was not like other server-side JavaScript platforms where all the I/O primitives were event-driven and there was no way around it. By leveraging the power and simplicity of JavaScript, this project turned the diffi cult task of writing event-driven server-side applications into an easy one. The project received a standing ovation and has since then been met with unprecedented growth, popularity, and adoption.

The project was named Node.js and is now known to developers simply as Node. Node provides a purely event-driven, non-blocking infrastructure for building highly concurrent software.

**NOTE** *Node allows you to easily construct fast and scalable network services.*

Ever since its introduction, Node has received attention from some of the biggest players in the industry. They have used Node to deploy networked services that are fast and scalable. Node is so attractive for several reasons.

www.it-ebooks.info

4 ❘ **CHAPTER 1** INSTALLING NODE

One reason is JavaScript. JavaScript is the most widely used programming language on the planet. Most web programmers are used to writing JavaScript in the browser, and the server is a natural extension of that.

The other reason is Node’s simplicity. Node’s core functionalities are kept to a minimum and all the existing APIs are quite elegant, exposing the minimum amount of complexity to the programmers. When you want to build something more complex, you can easily pick, install, and use several of the available third-party modules.

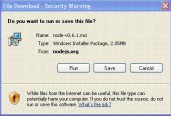
Another reason Node is attractive is because of how easy it is to get started using it. You can download and install it very easily and then get it up and running in a matter of minutes.

The typical way to install Node on your development machine is by following the steps on the http://nodejs.org website. Node installs out of the box on Windows, Linux, Macintosh, and Solaris.

**INSTALLING NODE ON WINDOWS**

Node supports the Windows operating system since version 0.6.0. To install Node on Windows, point your browser to http://nodejs.org/#download and download the node-v\*.msi Windows installer by clicking on the link. You should then be prompted with a security dialog box, as shown in Figure 1-1.

Click on the Run button, and you will be prompted with another security dialog box asking for confi rmation. If you agree, the Node install wizard begins (see Figure 1-2).

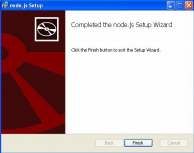
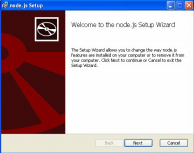
**FIGURE 1-1 FIGURE 1-2**

When you click Run, the Installation Wizard starts (see Figure 1-3).

Click on the Next button and Node will start installing. A few moments later you will get the confi rmation that Node was installed (see Figure 1-4).

www.it-ebooks.info

**Installing on Mac OS X** ❘ 5

**FIGURE 1-3 FIGURE 1-4**

**INSTALLING ON MAC OS X**

If you use Mac OS X you can install Node using an Install Wizard. To start, head to http://nodejs.org/#download and download the node-v\*.pkg Macintosh installer by clicking on the link. Once the download is fi nished, click on the downloaded fi le to run it. You will then get the fi rst wizard dialog box, as seen in Figure 1-5.

Choose to continue and install. The wizard will then ask you for the system user password, after which the installation will start. A few seconds later you’ll get the confi rmation window stating that Node is installed on your system (see Figure 1-6).

**FIGURE 1-5 FIGURE 1-6**

www.it-ebooks.info

6 ❘ **CHAPTER 1** INSTALLING NODE

**INSTALLING NODE USING THE SOURCE CODE**

If you have a UNIX system, you can install Node by compiling the source code. First you need to select which version of Node you will be installing, then you will download the source code and build, install, and run Node.

**NOTE** *Node depends on several third-party code libraries, but fortunately most of them are already distributed along with the program. If you are building from source code, you should need only two things:*

➤ *python (version 2.4 or higher) — The build tools distributed with Node run on python.*

➤ *libssl-dev — If you plan to use SSL/TLS encryption in your networking,*

*you’ll need this. libssl is the library used in the openssl tool. On Linux*

*and UNIX systems it can usually be installed with your favorite package*

*manager. The libssl library comes pre-installed on Mac OS X.*

**Choosing the Node Version**

Two different versions of Node are usually available for download on the nodejs.org website: the latest stable and the latest unstable.

In Node, the minor version numbering denotes the stability of the version. Stable versions have an even minor version (0.2, 0.4, 0.6), and unstable versions have an odd minor version (0.1, 0.3, 0.5, 0.7).

Not only might an unstable version be functionally unstable, but the API might also be mutating. The stable versions should not change the public API. For each stable branch, a new patch should include only bug fi xes, whereas APIs sometimes change in the unstable branch.

Unless you want to test a new feature that is only available in the latest unstable release, you should always choose the latest stable version. The unstable versions are a battleground for the Node Core Team to test new developments in the platform.

More and more projects and companies successfully use Node in production (some of the most relevant are on the nodejs.org home page), but you might have to put some effort into keeping up with the API changes on each new minor stable release. That’s the price of using a new technology.

**Downloading the Node Source Code**

After you choose a version to download, copy the source code *tarball* URL from the http://nodejs.org website and download it. If you’re running in a UNIX system, you probably have *wget* installed, which means that you can download it by using a shell prompt and typing the following:

$ wget http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz

www.it-ebooks.info

**Installing Node Using the Source Code** ❘ 7

If you don’t have *wget* installed, you may also use the *curl* utility:

$ curl -O http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz

If you don’t have either of these tools installed, you have to fi nd another way to download the *tarball* fi le into your local directory — such as by using a browser or transferring it via the local network.

(The examples in this book use the latest stable version at the time of writing, which is 0.6.12.) **Building Node**

Now that you have the source code in your computer, you can build the Node executable. First you need to unpack the source fi les like this:

$ tar xfz node-v0.6.12.tar.gz

Then step into the source directory:

$ cd node-v0.6.12

Confi gure it:

$ ./configure

You should get a successful output like this:

'configure' finished successfully (9.278s)

Then you are ready to compile it:

$ make

You should get a successful output like this:

'build' finished successfully (0.734s)

**Installing Node**

When you have built Node, you can install it by running the following command: $ make install

This will copy the Node executable into /usr/local/bin/node.

If you have a permissions problem when issuing this command, run it as the root user or using sudo: $ sudo make install

www.it-ebooks.info

8 ❘ **CHAPTER 1** INSTALLING NODE

**Running Node**

Now you are ready to start using Node. First you can simply experiment running Node as a *command-line interface* (CLI). For that you need only to call the Node executable with no argu ments like this:

$ node

This will start the CLI, which will then wait for you to input an expression. Just to test the installa tion and see Node actually doing something, you can type:

> console.log('Hello World!');

Hello World!

> undefined

You can also run a JavaScript script from a fi le. For instance, if you create a fi le with this content: console.log('Hello World!');

Name the fi le hello\_world.js. Then run the fi le by passing the fi le path as fi rst argument to the *Node* executable while inside a shell prompt. For example:

$ node hello\_world.js

Hello World!

You can quit the CLI by typing Ctrl+D or Ctrl+C.

**SETTING UP AND USING NODE PACKAGE MANAGER**

You can only get so far using the language features and the core functions. That’s why most programming platforms have a system in place that allows you to download, install, and manage third-party modules. In Node, you have *Node Package Manager* (NPM).

NPM is three things — a third-party package repository, a way to manage packages installed in your computer, and a standard to defi ne dependencies on other packages. NPM provides a public registry service that contains all the packages that programmers publish in NPM. NPM also provides a command-line tool to download, install, and manage these packages. You can also use the standard package descriptor format to specify which third-party modules your module or application depends on.

You don’t need to know about NPM to start using Node, but it will become necessary once you want to use third-party modules. Because Node provides only low-level APIs, including third-party modules is almost always necessary to fulfi ll any complex application without having to do it all yourself. As you will see, NPM allows you to download and play with modules without installing packages globally, which makes it ideal for playing around and trying things.

NPM and Node once required separate installs, but since Node version 0.6.0, NPM is already included.

www.it-ebooks.info

**Setting Up and Using Node Package Manager** ❘ 9

**Using NPM to Install, Update, and Uninstall Packages**

NPM is a powerful package manager and can be used in many ways. NPM maintains a centralized repository of public modules, which you can browse at http://search.npmjs.org. A Node opensource module author may choose, as most do, to publish the module to NPM, and in the installation instructions should be the NPM module name you can use to remotely download and install it.

This section covers the most typical uses of NPM, which are installing and removing packages. This should be just enough for you to start managing your application dependencies on third-party modules published on NPM. However, fi rst you need to understand the differences between global and local modes of operation and how these affect module lookups.

**Using the Global versus the Local Mode**

NPM has two main modes of operation: global and local. These two modes change target directories for storing packages and have deep implications for how Node loads modules.

The local mode is the default mode of operation in NPM. In this mode, NPM works on the local directory level, never making system-wide changes. This mode is ideal for installing the modules your application depends on without affecting other applications you might also have installed locally.

The global mode is more suited for installing modules that should always be available globally, like the ones that provide command-line utilities and that are not directly used by applications.

Always use the default local mode if you are in doubt. If module authors intend for one specifi c module to be installed globally, generally they will tell you so in the installation instructions.

The Global Mode

If you installed Node using the default directory, while in the global mode, NPM installs packages into /usr/local/lib/node\_modules. If you type the following in the shell, NPM will search for, download, and install the latest version of the package named sax inside the directory /usr/local/ lib/node\_modules/sax:

$ npm install -g sax

**NOTE** *If your current shell user doesn’t have enough permissions, you will need to login as the root user or run this command within sudo:*

$ sudo npm install -g sax

If you then have the requirement for this package in any Node script:

var sax = require('sax');

www.it-ebooks.info

10 ❘ **CHAPTER 1** INSTALLING NODE

Node will pick up this module in this directory (unless you have it installed locally, in which case Node prefers the local version).

The local mode is the default mode, and you have to explicitly activate the global mode in NPM by using the –g fl ag.

The Local Mode

The local mode is the default mode of operation in NPM and the recommended dependency resolution mechanism. In this mode NPM installs everything inside the current directory — which can be your application root directory — and never touches any global settings. This way you can choose, application by application, which modules and which versions to include without polluting global module space. This means that you can have two applications that depend on different ver sions of the same module without them confl icting.

In this mode, NPM works with the node\_modules directory under the current working directory. If you are inside /home/user/apps/my\_app, NPM will use /home/user/apps/my\_app/node\_modules as fi le storage for the modules.

If you run a Node application installed in the directory /home/user/apps/my\_app, Node will search this /home/user/apps/my\_app/node\_modules directory fi rst (before trying the parent directories and fi nally searching for it inside the global one). This means that, when Node is resolving a module dependency, a module installed using the local mode will always take precedence over a module installed globally.

**Installing a Module**

Using the following command, you can download and install the latest version of a package: $ npm install <package name>

For example, to download and install the latest version of the sax package, you should fi rst change your current directory to your application root directory and then type:

$ npm install sax

This will create the node\_modules directory if it doesn’t already exist and install the sax module under it.

You can also choose which version of a specifi c module to install by specifying it like this: $ npm install <package name>@<version spec>

You can use a specifi c version number in the <version spec> placeholder. For instance, if you want to install version 0.2.5 of the sax module, you need to type:

$ npm install sax@0.2.5

Under <version spec> you can also specify a version range. For instance, if you want to install the latest release of the 0.2 branch, you can type:

www.it-ebooks.info

**Setting Up and Using Node Package Manager** ❘ 11

$ npm install sax@0.2.x

If you want to install the latest version before 0.3 you should type:

$ npm install sax@"<0.3"

You can get more sophisticated and combine version specifi cations like this:

$ npm install sax@">=0.1.0 <0.3.1"

**Uninstalling a Module**

When you use the following command, NPM will try to fi nd and uninstall the package with the specifi ed name.

$ npm uninstall <package name>

If you want to remove a globally installed package, you should use:

$ npm uninstall -g <package name>

**Updating a Module**

You can also update an installed module by using the following command:

$ npm update <package name>

This command will fetch the latest version of the package and update it. If the package does not exist, it will install it.

You can also use the global switch (-g) to update a globally installed module like this: $ npm update –g <package name>

**Using the Executable Files**

It’s possible that a module includes one or more executable fi les. If you choose to install a module globally and you used the default installation directory settings, NPM installs the executables inside /usr/local/bin. This path is usually included in the default executable PATH environment variable.

If you installed a package locally, NPM installs any executables inside the ./node\_modules/.bin directory.

**Resolving Dependencies**

NPM not only installs the packages you request but also installs the packages that those packages depend on. For instance, if you request to install package A and this package depends on package B and C, Node will fetch packages B and C and install them inside ./node\_modules/A/ node\_modules.

www.it-ebooks.info

12 ❘ **CHAPTER 1** INSTALLING NODE

For instance, if you locally installed the *nano* package like this:

$ npm install nano

NPM outputs something like:

nano@0.9.3 ./node\_modules/nano

├── underscore@1.1.7

└── request@2.1.1

This shows that the *nano* package depends on the *underscore* and *request* packages and indicates which versions of these packages were installed. If you peek inside the ./node\_modules/nano/ node\_modules directory, you’ll see these packages installed there:

$ ls node\_modules/nano/node\_modules

request underscore

**Using package.json to Defi ne Dependencies**

When coding a Node application, you can also include a package.json fi le at the root. The package.json fi le is where you can defi ne some of your application metadata, such as the name, authors, repository, contacts, and so on. This is also where you should specify extraneous dependencies.

You don’t need to use it to publish the application to NPM — you may want to keep that application private — but you can still use package.json to specify the application dependencies.

The package.json is a JSON-formatted fi le that can contain a series of attributes, but for the pur poses of declaring the dependencies you only need one: dependencies. An application that depends on the *sax*, *nano*, and *request* packages could have a package.json fi le like this:

{

"name" : "MyApp",

"version" : "1.0.0",

"dependencies" : {

"sax" : "0.3.x",

"nano" : "\*",

"request" : ">0.2.0"

}

}

Here you are specifying that your application or module depends on *sax* version 0.3, on any version of *nano*, and on any version of *request* greater than 0.2.0.

**NOTE** *You may also fi nd that if you omit the* name *and* version *fi elds, NPM will not work. This happens because NPM was initially conceived to describe public packages, not private applications.*

www.it-ebooks.info

**Summary** ❘ 13

Then, on the application root, type:

$ npm install

NPM will then analyze the dependencies and your node\_modules directory and automatically download and install any missing packages.

You can also update all the locally installed packages to the latest version that meets your depen dency specifi cations, like this:

$ npm update

In fact you can always use this last command, because it will also make NPM fetch any missing packages.

**SUMMARY**

You’ve learned how to install *Node* and the *Node Package Manager*. You can now use NPM to install, uninstall, and remove third-party packages. You can also use the package.json fi le together with NPM to manage the third-party packages you depend on.

Now that you have Node and NPM installed, you are ready to experiment with them. But fi rst you need some background on Node and the event-driven programming style.

www.it-ebooks.info

www.it-ebooks.info

**2**

**Introducing Node**

**WHAT’S IN THIS CHAPTER?**

➤ What is the event-driven programming style and what are the

advantages of using it

➤ How Node.js and JavaScript make event-driven programming easy

Traditional programming does I/O the same way as it does local function calls: Processing cannot continue until an operation fi nishes. This programming model of blocking when doing I/O operations derives from the early days of time-sharing systems in which each process corresponded to one human user. The purpose was to isolate users from one another. In those systems, a user would typically need to fi nish one operation before deciding what the next operation would be. But with widespread use of computer networks and the Internet, this model of “one user, one process” did not scale well. Managing many processes places a big burden on the operating system — in memory and context switching costs — and the performance of these tasks starts to decay after a certain number is reached.

*Multi-threading* is one alternative to this programming model. A thread is a kind of light weight process that shares memory with every other thread within the same process. Threads were created as an ad hoc extension of the previous model to accommodate several concurrent threads of execution. When one thread is waiting for an I/O operation, another thread can take over the CPU. When the I/O operation fi nishes, that thread can wake up, which means the thread that was running can be interrupted and eventually be resumed later. Furthermore, some systems allow threads to execute in parallel in different CPU cores.

This means that programmers do not know what set of threads is executing at any given time, so they must be careful with concurrent access to the shared memory state. They have to use synchronization primitives like locks and semaphores to synchronize access to some data

www.it-ebooks.info

16 ❘ **CHAPTER 2** INTRODUCING NODE

structures, forcing them to foresee every possible way threads can be scheduled to execute to try to prevent problems. If the application relies heavily on a shared state between threads, this type of programming can easily lead to strange bugs that happen at random and are usually diffi cult to fi nd.

An alternative to having the operating system scheduling the thread execution for you is to use cooperative multi-threading. In this scheme you are responsible for explicitly relinquishing the CPU to give time for another thread to execute. Because you are now responsible for thread scheduling, this can relax the synchronization requirements. However, this approach can become complex and error-prone for the same reasons as regular multi-threading.

**INTRODUCING THE EVENT-DRIVEN PROGRAMMING STYLE**

Event-driven programming is a programming style whereby the fl ow of execution is determined by events. Events are handled by event handlers or event callbacks. An *event callback* is a function that is invoked when something signifi cant happens — such as when the result of a database query is available or when the user clicks on a button.

Consider how a query to a database is completed in typical blocking I/O programming:

result = query('SELECT \* FROM posts WHERE id = 1');

do\_something\_with(result);

This query requires that the current thread or process wait until the database layer fi nishes process ing it.

In event-driven systems, this query would be performed in this way:

query\_finished = function(result) {

do\_something\_with(result);

}

query('SELECT \* FROM posts WHERE id = 1', query\_finished);

Here you are fi rst defi ning what will happen when the query is fi nished and storing that in a function named query\_finished. Then you are passing that function as an argument to the query. When it’s fi nished, the query will invoke the query\_finished function, instead of simply returning the result.

This style of programming — whereby instead of using a return value you defi ne functions that are called by the system when interesting events occur — is called *event-driven* or *asynchronous* programming. This is one of the defi ning features of Node. This style of programming means the current process will not block when it is doing I/O. Therefore, several I/O operations can occur in parallel, and each respective callback function will be invoked when the operation fi nishes.

The event-driven programming style is accompanied by an *event loop*. An event loop is a construct that mainly performs two functions in a continuous loop — event detection and event handler triggering. In any run of the loop, it has to detect which events just happened. Then, when an event happens, the event loop must determine the event callback and invoke it.

www.it-ebooks.info

**How Node and JavaScript Make Writing Asynchronous Applications Easier** ❘ 17

This event loop is just one thread running inside one process, which means that, when an event hap pens, the event handler can run without interruption. This means the following:

➤ There is at most one event handler running at any given time.

➤ Any event handler will run to completion without being interrupted.

This allows the programmer to relax the synchronization requirements and not have to worry about concurrent threads of execution changing the shared memory state.

**A WELL-KNOWN SECRET** 

For quite some time, the systems-programming community has known that

event-driven programming is the best way to create a service that can handle many concurrent connections. It has been known to be more effi cient regarding memory because there is less context to store, and more effi cient regarding time because

there is less context switching.

This knowledge has infi ltrated other platforms and communities; some of the

most well-known event loop implementations are Ruby’s *Event Machine,* Perl’s

*AnyEvent*, and Python’s *Twisted*. There are also others for these and other

languages.

Implementing an application using one of these frameworks requires framework specifi c knowledge and framework-specifi c libraries. For example, when using Event Machine, you should avoid using synchronous libraries. To gain the benefi t of not blocking, you are limited to using asynchronous libraries built specifi cally for Event Machine. If you use any blocking library (like most of the ones in the Ruby standard library), your server will not be able to scale optimally because the event loop will be constantly blocking, which prevents timely processing of I/O events.

Node was devised as a non-blocking I/O server platform from day one, so generally you should expect everything built on top of it to be non-blocking. Because 

JavaScript itself is very minimal and does not impose any way of doing I/O (it does not have a standard I/O library), Node has a clean slate to build upon.

**HOW NODE AND JAVASCRIPT MAKE WRITING ASYNCHRONOUS APPLICATIONS EASIER**

Ryan Dahl, the author of Node, began his project building a C platform, but maintaining the con text between function calls was too complicated and led to complex code. He then turned to Lua, but Lua already had several blocking I/O libraries. This mix of blocking and non-blocking could

confuse developers and prevent many of them from building scalable applications, thus Lua was not ideal either.

www.it-ebooks.info

18 ❘ **CHAPTER 2** INTRODUCING NODE

Dahl then turned to JavaScript. JavaScript has closures and fi rst-class functions, which makes it a powerful match for event-driven programming. The power of JavaScript is one of the main reasons Node has become so popular.

**What Are Closures?**

Closures are functions that inherit variables from their enclosing environment. When you pass a function callback as an argument to another function that will do I/O, this callback function will be invoked later, and this function will — almost magically — remember the context in which it was declared, along with all the variables available in that context and any parent contexts. This powerful feature is at the heart of Node’s success.

The following example shows how a closure works when programming JavaScript in the web browser. For instance, if you want to listen for an event — a button click, for instance — you can do something like:

var clickCount = 0;

document.getElementById('myButton').onclick = function() {

clickCount += 1;

alert("clicked " + clickCount + " times.");

};

Or, using jQuery:

var clickCount = 0;

$('button#mybutton').click(function() {

clickedCount ++;

alert('Clicked ' + clickCount + ' times.');

});

In JavaScript, functions are fi rst-class objects, which means you can pass functions as arguments to other functions. In both examples you assigned or passed a function as an argument to be executed later. The click handling function — your callback function — has every variable in scope at the point where the function was declared, which means that, in this case, it has access to the clickCount variable declared in the parent closure.

The variable clickCount stores the number of times that the user has clicked on the button. This variable is stored in the global scope (the outermost scope you can get to in JavaScript). Storing variables in the global scope is generally bad practice because they can easily collide with other code; you should keep variables in a scope that is local to the functions that need to use them. Most of the time you can easily avoid global scope pollution by wrapping all of your code inside another function to create an additional closure, as in the next example:

(function() {

var clickCount = 0;

$('button#mybutton').click(function() {

clickCount ++;

alert('Clicked ' + clickCount + ' times.');

});

}());

www.it-ebooks.info

**Summary** ❘ 19

**NOTE** *Line 7 invokes a function immediately after defi ning it. This is a*

*common JavaScript pattern where you create a function just to create a new*

*scope.*

**How Closures Help When Programming Asynchronously**

In event-driven programming you start by defi ning the code that will be executed when an event occurs, then put that code inside a function, and fi nally pass that function as an argument to be called later.

In JavaScript, a function operates not by working in isolation but by remembering the context where it was declared, which enables manipulation of any variable present in that or any parent context.

When you pass a callback function as an argument, that function will be invoked some time later, but it can manipulate any variable present in that or in the parent scope, even if these scopes have already returned. In the last example, the callback function was able to access and manipulate the clickCount variable, even though it was invoked from within the jQuery click() function.

This shows that by using the closure pattern, you can have the best of both worlds: You can do event-driven programming without having to maintain the state by passing it around to functions. A JavaScript closure keeps the state for you.

**SUMMARY**

Event-driven programming is a programming style whereby the fl ow is determined by the occurrence of events. Programmers register callbacks to be used as event handlers for events they are interested in, and the system invokes these handlers when those events occur. This model of programming has some advantages over the traditional blocking paradigm where, to scale, you have to use multiple processes or threads.

JavaScript is a powerful language, which is well suited for this style of programming, mainly because it has fi rst-class functions and closures.

www.it-ebooks.info

www.it-ebooks.info

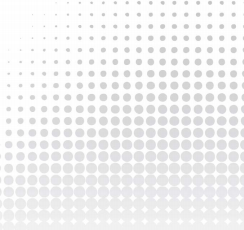
PART II 

**Node Core API Basics**

**CHAPTER 3:** Loading Modules 

**CHAPTER 4:** Using Buff ers to Manipulate, Encode, and Decode Binary Data

**CHAPTER 5:** Using the Event Emitter Pattern to Simplify Event Binding

**CHAPTER 6:** Scheduling the Execution of Functions Using Timers www.it-ebooks.info

www.it-ebooks.info

**3**

**Loading Modules**

**WHAT’S IN THIS CHAPTER?**

➤ Loading modules

➤ Creating modules

➤ Using the node\_modules folder

JavaScript is one of the most frequently deployed programming languages in the world; it’s the web’s *lingua franca*, used by all the browsers. The core of the language was created quickly back in the Netscape days, in a rush to beat Microsoft during the heat of the browser wars. The language was released prematurely, which inevitably meant it came out with some bad features.

Despite its short development time, JavaScript also shipped with some really powerful features. The sharing of a global namespace among scripts is not one of them, though.

Once you load JavaScript code into a web page, it is injected into the global namespace, which is a common addressing space shared by all other scripts that have been loaded. This can lead to security issues, confl icts, and general bugs that are hard to trace and solve.

Thankfully, Node brings some order in this regard to server-side JavaScript and implements the CommonJS modules standard. In this standard each module has its own context, separated from the other modules. This means that modules cannot pollute a global scope — because there is none — and cannot interfere with other modules.

In this chapter, you will learn about different types of modules and how to load them into your application.

Dividing your code into a series of well-defi ned modules can help you keep your code under control. You will also see how you can create and use your own modules.

www.it-ebooks.info

24 ❘ **CHAPTER 3** LOADING MODULES

**UNDERSTANDING HOW NODE LOADS MODULES**

In Node, modules are referenced either by fi le path or by name. A module that is referenced by a name will eventually map into a fi le path unless the module is a core module. Node’s core modules expose some Node core functions to the programmer, and they are preloaded when a Node process starts.

Other modules include third-party modules that you installed using NPM (Node Package Manager) or local modules that you or your colleagues have created.

Each module of any type exposes a public API that the programmer can use after the module is imported into the current script. To use a module of any type, you have to use the require function like this:

var module = require('module\_name');

This will import a core module or a module installed by NPM. The *require* function returns an object that represents the JavaScript API exposed by the module. Depending on the module, that object can be any JavaScript value — a function, an object with some properties that can be functions, an array, or any other type of JavaScript object.

**EXPORTING A MODULE**

The CommonJS module system is the only way you can share objects or functions among fi les in Node. For a suffi ciently complex application you should divide some of the classes, objects, or functions into reusable well-defi ned modules. To the module user, a module exposes exactly what you specify it to.

In Node, fi les and modules are in one-to-one correspondence, which you can see in the following example. Start by creating a fi le named circle.js, which just exports the Circle constructor:

function Circle(x, y, r) {

function r\_squared() {

return Math.pow(r, 2);

}

function area() {

return Math.PI \* r\_squared();

}

return {

area: area

};

}

module.exports = Circle;

The important bit here lies on the last line, where you defi ne what is to be exported by the module. module is a variable that represents the module you are currently in. module.exports is the object

www.it-ebooks.info

**Exporting a Module** ❘ 25

that the module will export to other scripts that require this module. You can export any object. In this case, you are just exporting the Circle constructor function, which a module user can use to create fully functional Circle instances.

You can also export more complex objects. module.exports is initialized with an empty object, which you can populate with the attributes you want to export. For instance, you can devise a module that exports a set of functions:

function printA() {

console.log('A');

}

function printB() {

console.log('B');

}

function printC() {

console.log('C');

}

module.exports.printA = printA;

module.exports.printB = printB;

module.exports.pi = Math.PI;

This module exports two functions (printA and printB) and a number (pi). A client of this module would look something like this:

var myModule2 = require('./myModule2');

myModule2.printA(); // -> A

myModule2.printB(); // -> B

console.log(myModule2.pi); // -> 3.141592653589793

**Loading a Module**

As explained previously, you can use the require function to load a module. Having a require function call in your code does not change the state of the global namespaces, because there is no such thing in Node. If the module is found and doesn’t contain a syntax or initialization error, calling require()simply returns the module object, which you can then assign to any local variable you choose.

There are several ways to reference modules, depending on which kind of module it is — a core module, a third-party module installed via NPM, or a local module. Let’s take a look at the different methods.

**Loading a Core Module**

Node has several modules compiled into its binary distribution. These are called the *core modules*, are referred to solely by the module name — not the path — and are preferentially loaded even if a third-party module exists with the same name.

www.it-ebooks.info

26 ❘ **CHAPTER 3** LOADING MODULES

For instance, if you wanted to load and use the http core module, you would do the following: var http = require('http');

This returns the http module object that implements the API described in the Node API documentation.

**Loading a File Module**

You can also load a non-core module from the fi le system by providing the absolute path like this: var myModule = require('/home/pedro/my\_modules/my\_module');

Or you can provide a path relative to the current fi le:

var myModule = require('../my\_modules/my\_module');

var myModule2 = require('./lib/my\_module\_2');

Notice here that you can omit the .js fi le termination. When it fails to fi nd such a fi le, Node will look for the path by adding the .js extension. So, if the fi le my\_module.js exists inside the current directory, the two following lines are equivalent:

var myModule = require('./my\_module');

var myModule = require('./my\_module.js');

**Loading a Folder Module**

You can use the path for a folder to load a module like this:

var myModule = require('./myModuleDir');

If you do so, Node will search inside that folder. Node will presume this folder is a package and will try to look for a package defi nition. That package defi nition should be a fi le named package.json.

If that folder does not contain a package defi nition fi le named package.json, the package entry point will assume the default value of index.js, and Node will look, in this case, for a fi le under the path ./myModuleDir/index.js.

However, if you place a fi le named package.json inside the module directory, Node will try to parse that fi le and look for and use the main attribute as a relative path for the entry point. For instance, if your ./myModuleDir/package.json fi le looks something like the following, Node will try to load the fi le with the path./myModuleDir/lib/myModule.js:

{

"name" : "myModule",

"main" : "./lib/myModule.js"

}

www.it-ebooks.info

**Exporting a Module** ❘ 27

**Loading from the node\_modules Folder**

If the module name is not relative and is not a core module, Node will try to fi nd it inside the node\_modules folder in the current directory.

For instance, if you do the following, Node will try to look for the fi le ./node\_modules/myModule.js: var myModule = require('myModule.js');

If Node fails to fi nd the fi le, it will look inside the parent folder called ../node\_modules/myModule .js. If it fails again, it will try the parent folder and keep descending until it reaches the root or fi nds the required module.

You can use this feature to manage the contents of the node\_modules directory or, preferentially, you can leave it up to NPM (see Chapter 1) to manage the modules for you. This local node\_modules directory is the default place where NPM installs modules, and this functionality is what ties Node and NPM together. Typically, as a programmer, you won’t care much about this feature. You can simply use NPM to install, update, and remove packages, and it will manage the node\_modules directory for you.

**Caching Modules**

Modules are cached the fi rst time they are loaded, which means that every call to require('myModule') returns exactly the same module if the module name resolves to the exact same fi lename.

For instance, say you have the following module inside the fi le called my\_module.js:

console.log('module my\_module initializing...');

module.exports = function() {

console.log('Hi!');

};

console.log('my\_module initialized.');

If the following script loads the script once:

var myModuleInstance1 = require('./my\_module');

It will print the following:

module my\_module initializing...

my\_module initialized

If you require the module two times:

var myModuleInstance1 = require('./my\_module');

var myModuleInstance2 = require('./my\_module');

www.it-ebooks.info

28 ❘ **CHAPTER 3** LOADING MODULES

You will notice that the output is still the same:

module my\_module initializing...

my\_module initialized

This means that the module initialization runs only once, which may be important to know when you are building a module that produces some side effects when being initialized.

**SUMMARY**

Node sets aside JavaScript’s default global namespace and uses CommonJS modules instead. This enables you to better organize your code, thus avoiding security issues and bugs. You can use require() to load a core module, a third-party module, or your own module from a fi le or a folder.

You can load non-core modules using relative or absolute fi le paths. You can also load your modules by name if you place them inside the node\_modules folder or install them using NPM.

You can create your own modules by authoring a JavaScript fi le that exports the objects that repre sent the module API.

www.it-ebooks.info

**4**

**Using Buff ers to Manipulate, Encode, and Decode**

**Binary Data**

**WHAT’S IN THIS CHAPTER?**

➤ Understanding why you need buff ers in Node

➤ Creating a buff er from a string

➤ Converting a buff er to a string

➤ Manipulating the bytes in a buff er

➤ Slicing and copying a buff er

JavaScript is good at handling strings, but because it was initially designed to manipulate HTML documents, it is not very good at handling binary data. JavaScript doesn’t have a *byte* type — it just has numbers — or structured types, or even byte arrays: It just has strings.

Because Node is based on JavaScript, Node can handle text protocols like HTTP, but you can also use it to talk to databases, manipulate images, and handle fi le uploads. As you can imagine, doing this using only strings is very diffi cult. In the early days, Node handled binary data by encoding each byte inside a text character, but that proved to be wasteful, slow, unreliable, and hard to manipulate.

To make these types of binary-handling tasks easier, Node includes a binary buffer implementation, which is exposed as a JavaScript API under the Buffer pseudo-class. A buffer length is specifi ed in bytes, and you can randomly set and get bytes from a buffer.

www.it-ebooks.info

30 ❘ **CHAPTER 4** USING BUFFERS TO MANIPULATE, ENCODE, AND DECODE BINARY DATA

**NOTE** *Another thing that is special about this buffer class is that the memory where the data sits is allocated outside of the JavaScript VM memory heap. This means that these objects will not be moved around by the garbage-collection*

*algorithm: It will sit in this permanent memory address without changing, which saves CPU cycles that would be wasted making memory copies of the buffer*

*contents.*

**CREATING A BUFFER**

You can create a buffer from a UTF-8-encoded string like this:

var buf = new Buffer('Hello World!');

You can also create a buffer from strings with other encodings as long as you specify the encoding on the second argument of the constructor:

var buf = new Buffer('8b76fde713ce', 'base64');

Accepted encodings and their identifi ers are:

➤ ascii— ASCII. This encoding is limited to the ASCII character set.

➤ utf8—UTF-8. This is a variable width encoding that can represent every character in the Unicode character set. It has become the dominant encoding on the web. This is the default encoding if you don’t specify one.

➤ base64—Base64. This encoding is used to represent binary data in an ASCII string format by translating it into a radix-64 representation. Base64 is commonly used to embed binary data into textual documents in a way that ensures that the data remains intact during transport.

If you don’t have the initial content for a buffer and you need to create a buffer with a certain capacity to hold data in the future, you can create a new buffer by specifying its length like this:

var buf = new Buffer(1024); // creating a 1024 byte buffer

**GETTING AND SETTING BYTES IN A BUFFER**

After you create or receive a buffer, you might want to inspect and change its contents. You can access the byte value on any position of a buffer by using the [] operator like this:

var buf = new Buffer('my buffer content');

// accessing the 10th position of buf

console.log(buf[10]); // -> 99

www.it-ebooks.info

**Getting and Setting Bytes in a Buff er** ❘ 31

**NOTE** *When you create an initialized buffer, keep in mind that it will contain random bytes, not zeros.*

var buf = new Buffer(1024);

console.log(buf[100]); // -> 5 (some random number)

You can also manipulate the content of any position:

buf[99] = 125; // set the 100th byte to 125

**NOTE** *In certain cases, some buffer operations will not yield an error. For*

*instance:*

➤ *If you set any position of the buffer to a number greater than 255, that position will be assigned the 256 modulo value.*

➤ *If you assign the number 256 to a buffer position, you will actually be assigning the value 0.*

➤ *If you assign a fractional number like 100.7 to a buffer position, the buffer position will store the integer part — 100 in this case.*

➤ *If you try to assign a position that is out of bounds, the assignment will fail and the buffer will remain unaltered.*

You can obtain a buffer length by inquiring the length property like this:

var buf = new Buffer(100);

console.log(buf.length); // -> 100

You can then use a buffer length to iterate over the buffer content and set or get each individual byte value:

var buf = new Buffer(100);

for(var i = 0; i < buf.length; i++) {

buf[i] = i;

}

Here, you create a new buffer with the capacity of 100 bytes and then set each byte with a value starting from 0 to 99.

www.it-ebooks.info

32 ❘ **CHAPTER 4** USING BUFFERS TO MANIPULATE, ENCODE, AND DECODE BINARY DATA

**SLICING A BUFFER**

Once you have created or received a buffer, you may want to extract a part from it. You can slice a buffer to create another smaller buffer, specifying the starting and ending positions like this:

var buffer = new Buffer("this is the content of my buffer");

var smallerBuffer = buffer.slice(8, 19);

console.log(smallerBuffer.toString()); // -> "the content"

Notice that when you slice a buffer no new memory is allocated and nothing is copied. The new buffer uses the parent buffer memory and just references different start and/or end positions. This has some implications.

First, if your program changes the parent buffer and any of that change touches any of the bytes of the child buffer, the child buffer will change. Because the parent and the child buffer are different JavaScript objects, this may not be obvious to you and can introduce some bugs.

Second, when you create a smaller buffer from a parent buffer this way, the parent buffer has to be kept around after the operation and not be reclaimed by the garbage collector, which can easily introduce a memory if you’re not careful.

**NOTE** *If you are concerned that you will leak memory by keeping an old buffer around, you should use the* copy *method instead, covered in the next section.*

**COPYING A BUFFER**

You can also copy part of a buffer into another buffer by using the copy method like this:

var buffer1 = new Buffer("this is the content of my buffer");

var buffer2 = new Buffer(11);

var targetStart = 0;

var sourceStart = 8;

var sourceEnd = 19;

buffer1.copy(buffer2, targetStart, sourceStart, sourceEnd);

console.log(buffer2.toString()); // -> "the content"

Here, you are copying position 8 to 19 of the source buffer into position 0 of the target buffer.

**DECODING A BUFFER**

A buffer can be converted into a UTF-8-encoded string like this:

var str = buf.toString();

www.it-ebooks.info

**Summary** ❘ 33

If you specify an encoding you can convert the buffer to another encoding. For instance, if you want to convert a buffer to a base64-encoded string, you can do so this way:

var b64Str = buf.toString("base64");

Using the toString method you can, for instance, transcode a UTF-8 string into base64 like this:

var utf8String = 'my string';

var buf = new Buffer(utf8String);

var base64String = buf.toString('base64')

**SUMMARY**

Sometimes you have to deal with binary data, but native JavaScript does not provide a clean way to do that.

The Node Buffer class encapsulates access to a continuous memory chunk. You can manipulate the bytes in that memory, obtain slices of it, and then copy that memory between two buffers.

You can also transform a buffer into an encoded string representation or go the other way around: transform a string into a buffer to access or manipulate the individual bits.

www.it-ebooks.info

www.it-ebooks.info

**5**

**Using the Event Emitter Pattern to Simplify Event Binding**

**WHAT’S IN THIS CHAPTER?**

➤ Introducing the event emitter pattern

➤ Binding and unbinding event listeners

➤ Creating your own event emitter

In Node many objects emit events. For instance, a TCP server can emit a “connect” event every time a new client connects, or a fi le stream can emit a “data” event every time a new chunk of data is read. These objects are, in Node nomenclature, *event emitters*. Event emitters allow programmers to subscribe to events they are interested in. The programmer attaches a callback function that will be invoked every time a relevant event in that event emitter occurs. This publisher/subscriber pattern is very similar to the typical GUI pattern, whereby a program gets notifi ed that a certain button was clicked. By using this pattern, a server-side program can react when, for instance, a client connects to the server, data is available on a socket, or a fi le gets closed.

You can also create your own event emitters. In fact, Node supplies an EventEmitter pseudo class that can work as a base for creating your own event emitters.

www.it-ebooks.info

36 ❘ **CHAPTER 5** USING THE EVENT EMITTER PATTERN TO SIMPLIFY EVENT BINDING

**UNDERSTANDING THE STANDARD CALLBACK PATTERN**

Asynchronous programming does not use function return values to denote that a function is fi nished. Instead it uses the *continuation-passing style* (CPS):

*Continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. (…)*

*A function written in continuation-passing style takes as an extra argument an explicit “continuation,” that is, a function of one argument. When the CPS function has computed its result value, it “returns” it by calling the continuation function with this value as the argument.*

Wikipedia — http://en.wikipedia.org/wiki/Continuation-passing\_style

This is a style in which a function invokes a callback after the operation is complete so that your program can continue. As you will see, JavaScript lends itself to this type of programming. Here is an example in Node that involves loading a fi le into memory:

var fs = require('fs');

fs.readFile('/etc/passwd', function(err, fileContent) {

if (err) {

throw err;

}

console.log('file content', fileContent.toString());

});

Here, you are passing an anonymous inline function as the second argument of the fs.readFile function, and you’re making use of the CPS, because you are continuing the execution of the program inside that function.

As you can see here, the fi rst argument to the callback function is an error object, which will have an instance of the Error class if an error occurs. This is a common pattern in Node when using CPS.

**UNDERSTANDING THE EVENT EMITTER PATTERN**

The standard callback pattern — whereby you pass a callback function as an argument of the function you are executing — works well when you want the client to be notifi ed when a function completes. But if several events take place during execution, or if they happen several times, this style doesn’t work as well. For instance, if you are interested in being notifi ed every time data is available on a socket, the standard callback pattern is not very helpful. This is when the event emitter pattern can help. You can use a standard interface to clearly separate the event emitter and the event listener.

When you use an event emitter pattern, two or more objects are involved — the event emitter and one or more event listeners.

An event emitter is an object that — as the name says — emits events. An event listener is a part of the code that binds to the event emitter and listens for certain types of events, like in this example:

www.it-ebooks.info

**Understanding Event Types** ❘ 37

var req = http.request(options, function(response) {

response.on("data", function(data) {

console.log("some data from the response", data);

});

response.on("end", function() {

console.log("response ended");

});

});

req.end();

Here, you are looking at some of the steps required to make an HTTP request to a remote HTTP server using the Node http.request API (which is covered later). Line 1 uses the continuation-passing style, passing in an inline function that will be executed once the response is available. The HTTP request API uses the CPS here because the program continues to execute after the http.request function completes.

When complete, the http.request function invokes the callback, passing a response object. This response object is an event emitter and, according to the Node documentation, can emit, among others, the data and end events. You are then registering callback functions that will be invoked every time any of these events happen.

As a rule of thumb, use CPS when you want to regain control after the requested operation completes and use the event emitter pattern when an event can happen multiple times.

**UNDERSTANDING EVENT TYPES**

Notice that emitted events always have a type, which is represented by a string. In this example you have the “data” and “end” event types. These are arbitrary strings dictated by the event emitter; by convention, event types are usually lowercase words with no spaces.

You cannot infer programmatically what types of events a given event emitter emits: The event emitter API provides no such introspection mechanism. The API you are using should document which event types it is emitting.

The event emitter will invoke the listener once a relevant event occurs, and it will pass in any relevant data. In the previous http.request example, the “data” event callback function received the data object as its fi rst and sole argument, whereas the “end” event didn’t receive any. These are also arbitrary arguments that are part of the specifi c API contract. The callback argument signature should be documented in the API specifi cation of each event emitter.

Event emitter is a generic interface that serves any type of event, but there is a special case in the Node implementation, and that’s the “error” event. Most event emitter implementations in Node will emit an “error” event every time there is an error. If the programmer chooses to not listen to

that event type and an event with the “error” type occurs, the event emitter will notice it and raise that error as an uncaught exception.

You can test that effect by running the following code in the Node REPL, which simulates an event emitter emitting two types of events:

var em = new (require('events').EventEmitter)();

em.emit('event1');

em.emit('error', new Error('My mistake'));

www.it-ebooks.info

38 ❘ **CHAPTER 5** USING THE EVENT EMITTER PATTERN TO SIMPLIFY EVENT BINDING

You will see the following output:

> var em = new (require('events').EventEmitter)();

undefined

> em.emit('event1');

false

> em.emit('error', new Error('My mistake'));

Error: My mistake

at repl:1:18

at REPLServer.eval (repl.js:80:21)

at repl.js:190:20

at REPLServer.eval (repl.js:87:5)

at Interface.<anonymous> (repl.js:182:12)

at Interface.emit (events.js:67:17)

at Interface.\_onLine (readline.js:162:10)

at Interface.\_line (readline.js:426:8)

at Interface.\_ttyWrite (readline.js:603:14)

at ReadStream.<anonymous> (readline.js:82:12)

>

Emitting an arbitrary event1 event did not have any effect, but when you emitted the “error” event, that error was thrown down the stack. If this program were running outside of a REPL, it would have halted because of the uncaught error.

As a rule of thumb, you should always listen to error events and handle them appropriately.

**USING THE EVENT EMITTER API**

Any object that implements the event emitter pattern (like a TCP Socket, an HTTP request, and many others) implements a set of methods:

➤ .addListener and .on — To add an event listener to an event type

➤ .once — To attach an event listener to a given event type that will be called at most once  ➤ .removeEventListener — To remove a specifi c event listener of a given event  ➤ .removeAllEventListeners — To remove all event listeners of a given event type

Let’s take a closer look at them.

**Binding Callbacks Using .addListener() or .on()**

By specifying an event type and a callback function, you can register to be called when one of these events occur. For instance, a fi le read stream may emit a “data” event when some data chunk is available. Here is how you can be informed of that by passing in a callback function:

function receiveData(data) {

console.log("got data from file read stream: %j", data);

}

**readStream.addListener(“data”, receiveData);**

www.it-ebooks.info

**Using the Event Emitter API** ❘ 39

Instead of using the .addListener function, you can use .on, which is simply a shortcut. The following code is equivalent:

function receiveData(data) {

console.log("got data from file read stream: %j", data);

}

**readStream.on(“data”, receiveData);**

Here, you are using a named function that you declare in advance, but instead you can use an inline anonymous function to be more succinct:

readStream.on("data", function(data) {

console.log("got data from file read stream: %j", data);

});

As pointed out before, the arguments that are passed into the callback depend on the specifi c event emitter object and event type and are not standardized. A “data” event might emit the data buffer, an “error” event might emit the error object, and the stream “end” event might not emit any value to the event listener.

**Binding Multiple Event Listeners**

The event emitter pattern allows multiple event listeners to listen to the same event type on the same event emitter. For instance:

readStream.on("data", function(data) {

console.log('I have some data here.');

});

readStream.on("data", function(data) {

console.log('I have some data here too.');

});

In this example you bind two functions to the readStream “data” event type. Every time the readStream object emits the “data” event, you will see printed out:

I have some data here.

I have some data here too.

The event emitter is then responsible for calling all the registered listeners for a given event type, and it will call them in the order in which they were registered. This means two things:

➤ An event listener might not be called immediately after the event is emitted. There might be other event listeners called before it.

➤ Throwing exceptions into the stack is not normal behavior, but it might be caused by a bug in your code. When an event is emitted, if one of the event listeners throws an error when invoked, some event listeners might never be called. In that case, the event emitter will catch the error and may handle it.

www.it-ebooks.info

40 ❘ **CHAPTER 5** USING THE EVENT EMITTER PATTERN TO SIMPLIFY EVENT BINDING

As an example, consider this:

readStream.on("data", function(data) {

throw new Error("Something wrong has happened");

});

readStream.on("data", function(data) {

console.log('I have some data here too.');

});

In this example, the second listener will not be invoked because the fi rst one throws an error.

**Removing an Event Listener from an Event Emitter Using .removeListener()**

If and when you no longer want to be informed of a specifi c event on a specifi c object, you can unregister by specifying the event type and the callback function like this:

function receiveData(data) {

console.log("got data from file read stream: %j", data);

}

readStream.on("data", receiveData);

// ...

readStream.removeListener("data", receiveData);

In this last example, the last line where you are removing the event listener may happen at any time in the future, as a reaction to another event.

Removing a specifi c event listener forces you to name your callback function because it will be used in at least two places — when adding the listener and when removing it.

**Getting a Callback Executed at Most Once Using .once()**

If you are listening for an event that will happen at most once, or even if you are only interested in listening to the fi rst occurrence of any given event type, you can use .once().

This method adds the event listener and removes it right after the fi rst event occurs.

function receiveData(data) {

console.log("got data from file read stream: %j", data);

}

readStream.once("data", receiveData);

Here, the receiveData function will be called only once. If a data event is emitted on the readStream object, the receiveData callback will be triggered only once.

This is a convenience method, because you could easily implement it like this:

var EventEmitter = require("events").EventEmitter;

EventEmitter.prototype.once = function(type, callback) {

var that = this;

www.it-ebooks.info

**Creating an Event Emitter** ❘ 41

this.on(type, function listener() {

that.removeListener(type, listener);

callback.apply(that, arguments);

});

};

Here, you are redefi ning the EventEmitter.prototype.once function, which redefi nes the once method for every object that inherits from EventEmitter. You are simply using the .on() method and, once you get the event, you use .removeEventListener() to unregister the callback and call the original callback.

**NOTE** *Here you use the* function.apply() *method, which takes an object to be used as the implicit variable* this *and an array of arguments. In this case, you are passing in the unmodifi ed arguments array, allowing for transparent hando ver of all arguments originally passed to the callback by the event emitter.*

**Removing All Event Listeners from an Event Emitter Using .removeAllListeners()**

You can remove all registered listeners for a particular event type from an event emitter like this: emitter.removeAllListeners(type);

For instance, if you want to remove all event listeners for the process interruption signal, you could do this:

process.removeAllListeners("SIGTERM");

**NOTE** *As a rule of thumb, I recommend you use this function only if you know exactly what you are removing. Otherwise, you might remove event listeners set by other parts of the application. Instead, those parts of the application should themselves be responsible for removing them. However, there could be some rare cases when this function would be useful, such as when you’re orderly shutting down an event emitter or even the whole process.*

**CREATING AN EVENT EMITTER**

The event emitter provides a great way of making a programming interface more generic. When you use a common understood pattern, clients bind to events instead of invoking functions, making your program more fl exible.

www.it-ebooks.info

42 ❘ **CHAPTER 5** USING THE EVENT EMITTER PATTERN TO SIMPLIFY EVENT BINDING

Also, by using the event emitter, you get some features for free, like having multiple independent listeners for the same events.

**Inheriting from Node Event Emitter**

If you are interested in using Node’s event emitter pattern throughout your application, you can. You can create a pseudo-class and make it inherit from EventEmitter like this:

util = require('util');

var EventEmitter = require('events').EventEmitter;

// Here is the MyClass constructor:

var MyClass = function() {

}

util.inherits(MyClass, EventEmitter);

**NOTE** util.inherits *sets up the prototype chain so that you get the*

EventEmitter *prototype methods available to your* MyClass *instances.*

**Emitting Events**

By creating a class that inherits from EventEmitter, instances of MyClass can emit events:

MyClass.prototype.someMethod = function() {

this.emit("custom event", "argument 1", "argument 2");

};

Here, when the someMethod method is called on an instance of MyClass, the example emits an event named custom event. The event also emits some data, in this case two strings: "argument 1" and "argument 2". This data will be passed along as arguments to the event listeners.

Clients of MyClass instances can listen to the event named custom event like this:

var myInstance = new MyClass();

myInstance.on('custom event', function(str1, str2) {

console.log('got a custom event with the str1 %s and str2 %s!', str1, str2); });

For example, you could build a pseudo-class named Ticker that emits a “tick” event every second:

var util = require('util'),

EventEmitter = require('events').EventEmitter;

var Ticker = function() {

var self = this;

www.it-ebooks.info

**Summary** ❘ 43

setInterval(function() {

self.emit('tick');

}, 1000);

};

util.inherits(Ticker, EventEmitter);

Clients of this class could instantiate this Ticker class and listen for the “tick” events like so:

var ticker = new Ticker();

ticker.on("tick", function() {

console.log("tick");

});

**SUMMARY**

The event emitter pattern is a recurrent pattern in Node. You can use it to decouple event emitter objects from the code interested in certain events.

You can use event\_emitter.on() to register listeners for certain types of events and event\_emitter.removeListener() to unregister them.

You can create your own event emitters by inheriting from the EventEmitter class and simply using the .emit() function.

www.it-ebooks.info

www.it-ebooks.info

**6**

**Scheduling the Execution of Functions Using Timers**

**WHAT’S IN THIS CHAPTER?**

➤ Deferring the execution of a function

➤ Canceling a scheduled execution

➤ Scheduling a periodic execution of a function

➤ Deferring the execution of a function until the next event loop tick

If you are accustomed to programming in browser JavaScript, you probably use the setTimeout and setInterval functions. These functions allow for the deferred execution of a function after a given period of time. For example, the following snippet of code, once loaded into a web page, appends the string "Hello there" after one second has elapsed:

var oneSecond = 1000 \* 1; // one second = 1000 x 1 ms

setTimeout(function() {

document.write('<p>Hello there.</p>');

}, oneSecond);

setInterval allows for the repetitive execution of a function spaced by a given time interval. If you inject the following snippet into a web page, it will also append the phrase "Hello there" to the document and keep doing so every second:

var oneSecond = 1000 \* 1; // one second = 1000 x 1 ms

setInterval(function() {

document.write('<p>Hello there.</p>');

}, oneSecond);

www.it-ebooks.info

46 ❘ **CHAPTER 6** SCHEDULING THE EXECUTION OF FUNCTIONS USING TIMERS

The need for these functions has arisen because the web has become a platform for building applications instead of static web pages. These scheduling functions help developers build periodic form validation, deferred remote data syncing, and all sorts of user interface interactions that require a delayed reaction. Node implements this exact set of functions. On the server side they can be used for assisting in repetitive or deferred execution of many different procedures that include cache expiration, connection pool cleanup, session timeout, polling, and others.

I will now go over these functions and then discuss some of their limitations.

**USING SETTIMEOUT TO DEFER THE EXECUTION**

**OF A FUNCTION**

The setTimeout function lets you schedule any other function to be executed once in the future. Here is an example:

var timeout\_ms = 2000; // 2 seconds

var timeout = setTimeout(function() {

console.log("timed out!");

}, timeout\_ms);

Exactly as in browser JavaScript, setTimeout accepts the deferred function as a fi rst argument and the time after which the function is executed — in milliseconds — as the second argument.

The call to setTimeout returns a timeout handler, which is an internal object that cannot be used for anything except canceling the execution using clearTimeout.

**USING CLEARTIMEOUT TO CANCEL THE EXECUTION OF A FUNCTION**

Once you have obtained a timeout handler, you can use it to cancel the scheduled function call using the clearTimeout function like this:

var timeoutTime = 1000; // one second

var timeout = setTimeout(function() {

console.log("timed out!");

}, timeoutTime);

clearTimeout(timeout);

In this case the timeout will never be fi red and "timed out!" will never be printed to the console. You can also cancel the scheduled execution at some time in the future. The following code snippet shows an example where this happens:

var timeout = setTimeout(function A() {

console.log("timed out!");

}, 2000);

setTimeout(function B() {

clearTimeout(timeout);

}, 1000);

www.it-ebooks.info