

Node.js for .NET Developers



David Gaynes



Node.js for .NET Developers

David Gaynes

PUBLISHED BY Microsoft Press A division of Microsoft Corporation One Microsoft Way Redmond, Washington 98052-6399

Copyright © 2015 by David Gaynes. All rights reserved.

No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2015930568 ISBN: 978-0-7356-6298-8

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at http://www.microsoft.com on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Acquisitions and Developmental Editor: Devon Musgrave Project Editor: Devon Musgrave Editorial Production: Waypoint Press (www.waypointpress.com) Technical Reviewer: Marc Young Copyeditor: Roger LeBlanc Indexer: Cristina Yeager Cover: Twist Creative • Seattle

Contents at a Glance

CHAPTER 1	Setup	1
CHAPTER 2	JavaScript and asynchronous code	11
CHAPTER 3	Coding Node.js	21
CHAPTER 4	Rendering with Node.js	31
CHAPTER 5	Working with site data in Node.js	39
CHAPTER 6	Working with external data in Node.js	47
CHAPTER 7	Working with file data in Node.js	57
CHAPTER 8	External authentication in Node.js	63
CHAPTER 9	Putting it all together in Node.js	71

Table of Contents

	Introduction	vii
Chapter 1	Setup	1
Chapter 2	JavaScript and asynchronous code	11
	Working with JavaScript	12
	Object-Oriented JavaScript	
Chapter 3	Coding Node.js	21
	Using the MVVM pattern	
	Writing the code	23
Chapter 4	Rendering with Node.js	31
	Before you start	
	Using real data	
	Adding images and styling	
Chapter 5	Working with site data in Node.js	39
	Data from URLs	
	Data from users	42
Chapter 6	Working with external data in Node.js	47

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Chapter 7	Working with file data in Node.js	57
Chapter 8	External authentication in Node.js	63
Chapter 9	Putting it all together in Node.js	71
	Sockets	
	Conclusion	77
	Index	79

What do you think of this book? We want to hear from you! Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

N ode.js is a JavaScript-based, highly scalable, open-source collection of tools used for sophisticated web development. Using your own chosen set of npm modules woven together under the Node.js paradigm, you can build websites of every imaginable type, from e-commerce to interactive games with multiple simultaneous users. In certain types of web applications, Node.js provides solutions to otherwise challenging technical issues.

Node.js is at its best in real-time web applications that require high-volume, low data-intensive processing of requests or in applications that want to employ push technology using web sockets connections. Today, the vast majority of web applications rely on the stateless request-response paradigm. In this paradigm only the client can initiate communication or data transfer. But with Node.js skills you can quickly build web applications with real-time two-way connections in which both the client and server can initiate communication.

Node.js websites are constructed using the standard open web stack composed of HTML, CSS and JavaScript. It allows for commonly used styling libraries to be added to your chosen collection of npm processing modules. Find out why Node.js is becoming a go-to platform for certain uniquely demanding types of web development.

Who should read this book

This book exists to help current .NET web developers learn the essentials of Node.js web development.

Assumptions

This book expects that you have at least a minimal understanding of .NET development and object-oriented programming concepts. With a heavy focus on web development, this book also assumes that you have a basic understanding of ASP.NET, including the core concepts of web development contained in ASP.NET, such as clients and servers, HTML, CSS, JavaScript, and HTTP post/get. The book also assumes that you have an IDE in which to work, specifically a fairly recent version of Visual Studio.

This book might not be for you if...

This book might not be for you if you have no web programming experience or if your interests within web programming are primarily to desig elegant user interfaces.

Organization of this book

This book is divided into nine chapters, which are designed to walk you through every required aspect of doing node.js development. The first few chapters cover the setup and basics of coding in node.js. The middle section of the book focuses on specific techniques within JavaScript that make life much easier when working with node.js. The last few chapters bring it all together to build a working application from end to end including a few special features, such as token-based authentication.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

- The book includes command line and JavaScript sample code, clearly separated from standard text
- The book includes references to named open-source modules available on the web. The first reference to each is in bold text

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

One of Windows XP with Service Pack 3 (except Starter Edition), Windows Vista with Service Pack 2 (except Starter Edition), Windows 7/8/10, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2.

- Visual Studio 2010 or later, any edition (web developer for Express Edition products).
- SQL Server 2008 Express Edition or higher (2008 or R2 release), with SQL Server Management. Studio 2008 Express or higher (included with Visual Studio, Express Editions require separate download).
- A computer that has a 1.6GHz or faster processor (2GHz recommended).
- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions).
- 3.5GB of available hard disk space.
- 5400 RPM hard disk drive.
- DirectX 9 capable video card running at 1024 x 768 or higher-resolution display.
- DVD-ROM drive (if installing Visual Studio from DVD).
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010 and SQL Server 2008 products.

Downloads: Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text on your way to building a fully functional web application. Fully working examples of the pages used in the application can be found here:

http://aka.ms/node.js/files

Follow the instructions to download the Nodejs_662988_CompanionContent.zip file.

Using the code samples

The folder created by the setup.exe program contains two kinds of files:

■ **JavaScript files** These files contain the Node. js code that runs your application including navigation, page data content, etc.

 EJS files These files are used in place of standard HTML files for rendering pages. Although they do contain all necessary HTML, they also contain special binding syntax that allows the file to interact with its associated node.js JavaScript file.

Acknowledgments

I'd like to thank the following people: Devon Musgrave and Marc Young for helping me polish this project and get it to print, Devon again for special efforts connected to the project, and of course my wife Samantha for her endless support.

Errata, updates,& book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

http://aka.ms/nodejs/errata

If you discover an error that is not already listed, pleasesubmitit to us at the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress

Working with site data in Node.js

Data comes in many shapes and sizes. It can come from within your application as it works, from users, or from outside data stores. As with the rest of Node.js, there is an npm package to deal individually with each of the myriad ways data might be provided to your application.

In general, this data can be broken into three major categories, with each having a couple of primary ways that data would likely come to you:

- Data from URLs
 - Route/Path pattern
 - QueryString
- Data from users
 - Form posts
 - Field input
- Data from external sources
 - Databases (covered in Chapter 6, "Working with external data in Node.js")
 - File systems (covered in Chapter 7, "Working with file data in Node.js")

We'll walk through these and incorporate them into your Node.js application.

Data from URLs

The first way you'll have to deal with data in the application relates to data that comes in the URL. Even our simple example code for rendering an array that isn't connected to anything contains within it the idea that we will drill down into some detail about some item on the list. And that item is specified within the URL.

In this case, as shown, we opted to build a path that looks like this:

```
href= 'details/<%= players[i].id%>'>
```

This path yields the following result or similar as the URL path:

'details/5'

You could keep going and add as many path arguments as you like after the original. Each argument is given a name when you declare the path, like this:

```
router.get('/details/:id')
```

Or by adding to it, like this:

```
router.get('/details/:id/:name')
```

Then, in code, these arguments are accessed through the param collection provided by Express:

req.param("id")

Your *get* will look like this:

```
router.get('/details/:id', function (req, res) {
    res.render('details', {
        playerId: req.param("id")
    });
});
```

When you create your player and then set up the *details.js/.ejs* view, you'll have a *playerId* argument to get your hands on and display right away to make sure you have the correct data.

Always remember, as I mentioned earlier, that you must order your code to go from most specific in the path to least specific because the engine will render the first matching pattern that it finds. The warning here is this URL coming in as a web request:

```
http://127.0.0.1:1234/details/1/Payton
```

This will match the following route:

router.get('/details')

And it will match this route as well:

```
router.get('/details/:id')
```

And this route:

```
router.get('/details/:id/:name')
```

The second and third routes will never be reached if these are in the wrong order in your code file. To be properly done, these route entries need to be exactly reversed from what is shown here.

Pulling arguments from structured route paths is one way to pass and pull data from the URL. As an alternative, you could have placed the data into a *QueryString*. In the real world, this is equally as likely to be the pattern you choose to follow for building URLs internally to pass data.

You use the *QueryString* collection to access the URL's name/value pairs either by index or by name. In general, it will be just as easy to code a solution that parses its data as with the *param* collection. That being said, it is your choice, and each option works at least equally well.

To implement access to the *QueryString*, simply reference its collection instead of the *param* collection:

req.query.ID

In this case, as you see, you can actually use dot notation to access the individually named members of the collection you specified. The processing engine recognizes the question mark (?) as the beginning of the collection and the ampersand (&) as the argument separator. Thus, the route itself is still the same as the base route and the previously mentioned issues with route order in the file are not relevant.

So to process this URL:

http://127.0.0.1:1234/details?ID=1&Name=Payton

the route get function to render this data in all cases would simply be

```
router.get('/details', function (req, res) {
    res.render('details', {
    playerId: req.query.ID,
    name: req.query.Name
    });
});
```

As you can see, you simply take apart the arguments by name one at a time to get to the values contained in them. Passing an entire object this way would be done by manually taking apart the object properties to provide the necessary arguments in your assembled link to the details page:

href= 'details?ID=<%= players[i].id%>&Name=<%= players[i].lastName%>'>

Continue on like this in as much depth as required. In real-world practice, this approach is rarely needed for sending information to your own .ejs files. This is because, as you have seen, you can pass entire objects or even collections of objects in this way:

```
res.render('survey', {
    players: arrPlayers
    });
```

For connecting to external resources and assembling a *QueryString* or a route, or for taking in connections to your resources from others and thus parsing an inbound *QueryString* or route, working with data directly inside the URL is often your only option for moving that data from place to place.

Data from users

Another technique that should be familiar to you if your background is classic ASP or MVC is *form posting*. With this approach, it is assumed that there is a screen into which a user is entering one or more fields and that the entire collection of those values needs to be quickly and easily transported to a URL. To implement form posting, instead of using a *get* function inside of your .js file, you need to use a *post*:

```
router.post('/survey', function (req, res) { });
```

As you can see, *post* is almost identical in signature to *get*. Before we get more deeply into it, just touching on the subject of *post* as an alternative to *get* leads us back into taking a brief look at the four basic actions universally available over an HTTP web connection:

- Get A typical web request
- Post Usually used for sending a collection of data to be processed
- Put Usually used for updating a single record
- Delete As you might have expected, usually used for deleting a single record

A file that contains routing information will typically contain at least one of each of the four methods just defined, like this:

```
router.get('/survey', function (req, res) { });
router.post('/survey', function (req, res) { });
router.put('/survey', function (req, res) { });
router.delete('/survey', function (req, res) { });
```

As you have seen, often more than one *get* function exists within this collection. Within the *post* will be the code to get the name/value pairs from the inbound form collection. Once again, the key to this is having the proper npm package installed. In this case, it is *body-parser*. So, in your app.js file, make sure you have this line:

```
var bodyParser = require('body-parser');
```

You need the variable because you also need to place this line in the same file after the *bodyParser* declaration just shown to properly format the inbound input:

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

With those pieces in place, from inside your *post* function, you'll be able to access the body property of an inbound request by doing this to get your hands on the value you seek:

```
var sInput = req.body.txtInbound;
```

In this case, you are looking for a control called txtInbound.

To see this is action, you need to add a few things to your HTML/EJS file to activate a form post. Let's start with a button and a textbox. Just to do this demonstration, go ahead and drop a couple of input controls in a separate row below the list control you have in the page:

```
. . .
<
form action="/survey" method="post">
<input id="txtInbound" name="txtInbound" type="text" />
<input type="submit" />
</form>
```

Notice how you have wrapped your input controls in a form and then specified two important attributes—*method='post'* and *action='/survey'*—to tell the form how you want it to behave. When you submit it, you want the form to post its information to the path indicated in the action—in this case, to your survey page.

With all of this wiring in place, let's turn our attention back to the actual *post* function inside your survey.js file to have it respond to your successful form post. Again, we'll do something more useful with this later. Just to see it work, let's have it write any input value to the console:

```
router.post('/survey', function (req, res) {
  var sInput = req.body.txtInbound;
  console.log(sInput);
  res.send('posted');
});
```

At this point, you can pop the page open in a browser. You should see your button control and your text box. Enter any value, and click Submit. You should see the value you typed appear in the console window. Don't forget to respond with something; otherwise, your application will stop responding even if the code works as expected.

Typically, this is where you have your CRUD (Create, Update, Delete) interactions with an external source such as a database. When you return to this code in the next chapter, you'll be taking that value and inserting it into a Microsoft SQL Server database.

Now you've seen the Node.js versions of standard get and post operations.

However, aside from these basic, good old-fashioned web techniques for moving bits of data from here to there, inside your Node.js application you do have other options. One of the best of these is the cache.

To use a cache in Node.js, all you need is the proper npm package. You already installed memory-cache when you set up our application so now you just have to do the usual to enable its use:

```
var cache = require('memory-cache');
```

This component works just as you would hope that it would, similar to the .NET cache but without some of the features. To put a value into the cache, you simply do this:

```
cache.put('players', arrPlayers);
```

And to retrieve that value, this is all it takes:

```
cache.get('players');
```

This caching component also has an expiration argument, expressed in milliseconds:

```
cache.put('players', arrPlayers, 5000);
```

After the time elapses, in this case five seconds, the item will be removed from the cache if it is not renewed.

One of the most important things to notice here is that my example for storing something in the cache uses not just an object but an array of objects. The cache will hold anything you can concoct in JavaScript, and this idea opens the door to the true power of Object-Oriented JavaScript (OOJS) and sophisticated state management that's required in commercial applications. It's a pattern used in .NET to take advantage of the full power of the programming model.

The Node.js cache does for JavaScript objects what the .NET cache does for business-layer objects—it makes them universally accessible. There are two kinds of objects typically stored in the cache: lookup objects and user objects.

A *lookup object* is typically a collection of values that rarely, if ever, changes—something like US state names or US president names. A list of states stored in the database has to be fetched only one time, and then it can be held in memory after that to allow for quicker access from the application. This works quickly and easily because there is no concern that data will get out of sync—in other words, there are no worries that new data will be entered into the database and the cached version of the data will be out of date. With data that never changes, such as US states, that problem is not a problem. A routine that re-created that data once every four or eight years is also not a big issue.

Of course, this design also works for lookup data that changes more regularly. You simply have to account for those changes in memory as well as in the database—for example, by updating the collection in memory on the same button click that allows for a database update. This is one way to greatly improve the performance of your application.

In general, interaction with the database should be avoided except in cases where it simply can't be, such as for CRUD operations. Most other functions that people typically perform in the database, such as filtering and joining, can be done much more quickly by using server memory.

Picture a software application as the city of San Francisco, and imagine the only four bridges over the bay represent the database. No matter how many creative ways you navigate to one of those bridges, you'll be slammed with all the rest of the city traffic doing the same. Those are the only routes. So everyone has to use them. If you keep your database interactive operations to the bare minimum required, traffic will flow better all over your "city." That's the whole idea behind using as much cached data as you possibly can.

A user object holds all the information specific to a site user. That can be data from the system itself, such as profile information containing permissions, and it can be state data that is tracking current user behavior. Both kinds of information will typically be required all throughout the application, and the cache is the perfect tool to use for allowing it.

Only one difference is required in the way you manage the cache. For a lookup object, this will work:

```
cache.put('leagues', arrLeagues);
```

However, for a user-specific object, you need an identifier that ties that specific object to that and only that specific user. The standard technique for doing so is to create a *globally unique identifier* (GUID) that you associate with that user, most often on login. Then you simply prepend the GUID to your cache entry like this:

```
cache.put(GUID + 'User', myUserObj);
```

You should have that GUID for that user included in the QueryString on every request, like this:

http://127.0.0.1:1234/details?GUID=1QW3456tr4RTYUXXYiujjii45UY89898TRFReded

That way, you can then pull it out to get access to your user object in the cache, like this:

```
var sGUID = req.query.GUID;
var myObj = cache.get(sGUID + 'User');
```

You have a rock-solid, state management strategy in place that works for every page of your application, with code consistency, in exactly the same way.

As I mentioned, this caching technique is the only truly viable solution for all web scenarios, even in the world of .NET. If you don't believe it, try to pass a *Session* variable across a protocol change that is, take one of your *Session* values and pass it from HTTP to HTTPS. Good luck! There's no way that coding technique will ever work. Sessions do not cross protocol boundaries. You can, and want to, create *Session* equivalents using the login GUIDs, caching, and OOP, but that's not nearly the same thing as using the *Session* object. You can even take the idea one step further for web-farm scenarios by serializing the data in your objects to external data stores. Serialization turns the state of an object into text. So you serialize to store the data and deserialize to retrieve it. When a request comes in, you check the cache in that specific server for the GUID-related user object. If it isn't there, you pull the user state from the external store according to the GUID in the *QueryString* and reassemble it into objects right there. And then you are back to normal code operations. One technique, all scenarios, infinitely scalable.

Now you use a tech interview question to separate the wheat from the chaff—no .NET developer worth his salt will ever go near *Session*. Like form posting, it's technology from the 1990s, and .NET gave you much better ways to do the same things starting in this century, and it still does. By being well-versed in those best practices, you're fully prepared to implement the same architecture in Node.js.

At this point, you're effectively moving your data from page to page. Next let's connect to some external data and see what you can do with that.

Index

? (question mark), 41 & (ampersand), 13, 41

Α

ampersand (&), 13, 41 app.js files, 23 body-parser, attaching, 42 cache references, 54 class functions, 54 class references, 73 enabling Embedded JavaScript package, 31 file locations, 25 loading objects into cache memory, 72-73 login page, referencing, 64 Passport package, referencing, 64 socket servers, enabling, 77 static references to, 52 upload routes, 58-59 Application object, 29 applications code layers, 22 components of, 22 login, 63–69 arrays creating, 35 declaring, 36-37 displaying, 73–74 filtering, 74–75 manipulating, 74-75 arr.pop function, 74 arr.shift function, 74 arr.unshift function, 74

ASP.NET, 32–33 asynchronous model of code, 15, 73 authentication, 63 configuring, 64–65 Facebook authentication, 67–68 Google authentication, 66–67 OAuth 2.0, 68–69

В

back-end databases, 47 body-parser package, 6, 42 Bootstrap (and Bower) package, 7–9 linking to pages, 37 location, 37 table class reference, 38 bower_components folder, 21 *busboy* package, 58–59 buttons, creating, 58

С

cache, 43–44 application-wide data, 48 expiration argument, 44 GUIDs, 45 loading objects into, 72–73 lookup objects, 44 management, 45 referencing, 54 universal access to objects, 44 user objects, 44–45 values, inputting and retrieving, 44

callback functions

callback functions, 14-16 Cannot Get errors, 52 cascading style sheets (CSS), 22 chat.htm pages, 77 child classes, adding functions and properties, 19 child/prototype setup, 18-19 class functions, 54 class references, 73 code asynchronous model, 15, 73 documentation, 16 layers, 22 MVVM pattern, 21–22 order of, 40 using a template, 33-34 code encapsulation, 27 code structure, 22 collections looping through, 49 populating for display, 54 receiving, 49, 51 updating, 44 concatenation, 12–13 core errors, 63 createReadStream method, 60 createServer function, 16 createWriteStream method, 59–60 credentials, checking, 65. See also login CRUD (Create, Update, Delete) interactions, 43, 45

D

data from external sources, 39. See also databases fetching, 49 file data, 57–61 serializing, 46 site data, 39–46 from URLs, 39–41 from users, 39, 42–46 databases, 39, 47–55 back-end, 47 row-by-row access, 55 tables, creating, 49 updating, 44–45 data transfers, pausing, monitoring, resuming, 59 data version, 60 declarations, 13 *delete* function, 42 __*dirname* value, 25 displaying file data, 58 *done* function, 65

E

EJS engine, associating files with, 31 .ejs files, 31 connecting to .js files, 35 form posting, activating, 43 input fields, adding, 51 and .js files of same name, 31-32 rendering list data, 35 EJS package, 7 Embedded JavaScript package, 31 encapsulation, 27 end function, 16, 28, 60 error handling, 48 errors Cannot Get errors, 52 core errors, 63 name mismatches, 32 server malfunctions, 32 exports property, 27 Express Node.js projects, 3 Express package, 6 as backbone, 16 fs object, 57, 59-61 external authentication, 63-69 external data, 47-55 external data stores. See also databases getting data from, 39 serializing data to, 46

F

Facebook authentication, 67–68 Facebook Developers, 67 fetching data, 49, 73 rows, 53 field input, 39 file copy operations, 60 file data, 57-61 displaying, 58 metadata, 60-61 reading, 58 strings, cleaning, 61 total size, 60-61 transferring, 58 writing to file, 57 file systems, 39 file transfers, managing, 59-60 file uploads, 57 filtering arrays, 74-75 findByUsername function, 65 forEach function, 75 form posting, 39, 76 activating, 43 implementing, 42 fs object createReadStream method, 60 createWriteStream method, 59 readFile method, 57 statSync function, 61 writeFile method, 57 function declarations, 13 functions, 14, 17 adding, 19 callbacks, 14-16 overriding, 18 Functions, 17

G

/g argument, 61 get function, 26, 42 creating, 52 fetching data, 49 with path arguments, 40–41 rendering code, 35 render method, 32 GIT, 9 globally unique identifiers (GUIDs), 45 Google authentication, 66–67 Passport package, 66

Η

HTML, creating buttons, 58 HTML files form posting, activating, 43 Views, 22 HTML tags in templates, 34 *http.createServer();* function call, 14 HTTP web connection actions, 42 Hungarian notation, 13

I

images adding to page, 37 path, setting, 25–26 styling, 38 *indexOf* function, 74 inheritance, 19 input controls, wrapping in forms, 43 insert operations, 49–51

J

JavaScript, 12–17. See also Object-Oriented JavaScript arrays, manipulating, 74–75 bitwise operators, 13 classes (objects), 17 concatenation, 12–13 declarations, 13 function declarations, 13 functions, 14 Functions and functions, 17 module references, 14

JavaScript

JavaScript (continued) non-strong-typing, 13-14 object-oriented, 17-20 object referencing, 17 optimized, 17 rendering, 34. See also rendering require function call, 14 this keyword, 18 types, 13 .is files code running from, 58 connecting to .ejs files, 35 and .ejs files of same name, 31-32 passport-local, referencing, 64 post function, 43 routes and function objects in, 27 JSON code, converting to OOJS format, 36-37

L

lastIndexOf function, 74 *length* method, 60 *listen* function, 16 local file system, mapping, 25 *Local* login strategy, 64–65 login credentials checking, 65. See also authentication done function, 65 Local strategy, 64–65 security, 63 view and view model, 63–64 lookup objects, 44

Μ

match function, 61
memory-cache package, 6, 29
enabling, 44
Microsoft SQL Server, 47. See also SQL Server
middleware, 63
Models, 21–22
Model-View-Controller (MVC) pattern, 21–22

Model-View-ViewModel (MVVM) pattern, 21–22 module objects, 26 MySQL, 47 connection configuration, 49 data inserts and results, 55 installing, 48 MySQL package, 47–48

Ν

name mismatches, 32 name/value pairs accessing through QueryString collection, 41 Application object, 29 custom, 29 .NET built-in libraries, 5 node.exe. 23 Node.js coding, 21-29 command prompt, 3 configuration, 25 connecting to Visual Studio 2010, 3-5 core, 16 downloading, 1 file I/O, 25 installing, 2 MVVM pattern, 22 setup, 1–9 NodeJS.org download page, 1 home page, 11 node_modules folder, 21 non-strong-typing, 13-14 npm command, 7–8 npm package-management application, 3 npm packages authentication, 6 basic/core, 6 database access, 6 download location, 5, 7 installing, 7-8 IO/file upload, 6

rendering, 6 statement/management, 6 support packages, 5

0

OAuth, 63 OAuth 2.0, 68-69 Object-Oriented JavaScript (OOJS), 17-20. See also JavaScript best practices, 20 converting JSON code to, 36-37 Node.js cache, 44 SQL functions, connecting, 53 objects, 71 function declarations, 14, 17 outside data for, 19 properties, 13 properties, setting to forEach loop values, 54 OpenID, 63, 66 optimized JavaScript, 17 overrides, 18

Ρ

pages arguments, supplying, 34 data, supplying, 34 images, adding, 37 routing to, 26 separating functionality, 59 styling, 37–38 param collection, 40 passport.authenticate function, 65, 67 passport-http-bearer package, 68 passport-local package, 64

Passport package, 7, 63 Google flavor, 66 req.authInfo property, 69 strategies, 63, 65 path arguments, 39–40 path-recognition tree, 27

paths, responses for, 28 performance asynchronous processing and, 15 optimized JavaScript and, 17 pipe function, 59 postbacks, 33 post function, 42-43 body properties, accessing, 42 connection configuration, 48 hard-coded credentials, 48 of logins, 64 receiving, 51 post handlers, 51 projects Express versions, 3 npm package references, adding, 23 server, building, 23 setup, 31 use strict declaration, 23 properties adding, 19 declaring, 18 as methods, 18 of objects, 13 property/variable values, 34 protocol boundaries, crossing, 45 prototype base class, 18-19 child/prototype setup, 18-19 push function, adding to arrays, 74 put function, 42

Q

QueryString collection, 39 arguments from, 40–41 GUIDs in, 45 question mark (?), 41

R

readFile method, 57 reading data, 58–60 *redirect* method, 28

redirects

redirects, 28, 55 Regular Expression pattern, 61 relational database systems, 47 relative web file paths, 26 rendering, 31-38 arguments, 34 data, 34 with Object-Oriented JavaScript, 36-37 pages, 75-76 titles, 34 views, 26-27, 64 render method, 27-28 req.authInfo property, 69 req objects, 26 *Request* function redirects, 55 rendering results in, 51 request objects, 16 require function call, 14 require references, 26 res objects, 26 response objects, 16, 27-28 resultsets, separating into columns, 54 route entries order, 40 Route/Path pattern, 39 Router method, 26 routing, 16 action methods available, 42 globalizing router and routes, 52 to login, 65 post handler, 51 references, adding, 51 return objects, 28 upload routes, 58-59 rows, selecting and iterating, 53

S

searching with *match* function, 61 sending data, 59 *send* method, 27–28 serialization and deserialization, 46 server.get method, 29 server.listen method, 23, 25, 29 server malfunctions, 32 server memory, updating data in, 44-45 servers, building (chained and unchained), 23 - 24server.set method, 25, 29 server.use method, 26, 29 single sign-on, 63, 66-67 site data, 39-46 site login, 63. See also login site navigation, 25-27 site users, 45 Socket.io npm package, 76 socket.on function, 77 sockets, 76-77 split function, 61 SQL Express, 47 SQL Server, 47 interacting with, 48 parameters, 49 rows, selecting and iterating, 53 Start menu, Node.js command prompt, 3 state management, 45 statSync function, 61 stored procedures, 47 strategies, 63 Facebook authentication, 67–68 Local Authentication, 64–65 OAuth 2.0, 69 validation functions, 65 streaming files, 59-60 stream.pipe function, 59 strings, cleaning, 61 structured route paths, arguments from, 40-41 styling images, 38 pages, 37 substring function, 61 synchronizing data transfers, 59

Т

tables, creating, 49 tedious package, 7 enabling, 47–48 referencing, 73 requiring, 50 templates, 33–34 for rendering list data, 35 syntax, 34 *this* keyword, 18 *toLowerCase* function, 61 *toUpperCase* function, 61 *toUpperCase* function, 61, 75 transferring file data, 58 *trim* function, 61 types, 13

U

upload buttons, creating, 58 upload routes, 58–59 URLs, data from, 39–41 user data, 39, 42–46 user input, recording, 76 user interface, adding fields, 51 user objects, 44–45 user profiles, 66 *use strict* statement, 13, 23 using a template, 33–34

V

var declaration, 12 "View_Model" file, 31 Views, 21-22 path, setting, 25-26 rendering, 26-27, 64 require reference, 26 Visual Studio CSS folder, 22 Express version plugin, 3 Images folder, 22 .js files, creating, 25 project setup, 21 View, creating, 25 View folder, 22 Visual Studio 2010, 3-5 Visual Studio 2012, 3

W

weak typing, 19 web applications working directory, 5 Web Forms, 32–33 web requests, asynchronous processing, 15 *writeFile* method, 57 *writeHead* function, 16 writing to files, 57

About the author

David Gaynes has nearly 20 years of experience as a developer, architect, and consultant working across the full software development life cycle. He has spent more than a dozen years working with Microsoft and .NET technologies for clients of all sizes. His clients have ranged from the Chicago Public Schools and the Children's Hospital in Seattle to diverse organizations in healthcare, insurance, finance, and gambling.