Much love to Tina, for simply being there.
Contents at a Glance

Introduction 1

I: Learning to Walk 7
  1 Getting Started 9
  2 A Closer Look at JavaScript 23
  3 Asynchronous Programming 49

II: Learning to Run 63
  4 Writing Simple Applications 65
  5 Modules 89
  6 Expanding Your Web Server 115

III: Writing Web Applications 135
  7 Building Web Applications with Express 137
  8 Databases I: NoSQL (MongoDB) 165
  9 Databases II: SQL (MySQL) 193

IV: Getting the Most Out of Node.js 213
  10 Deployment and Development I: Rolling Your Own 215
  11 Deployment and Development II: Heroku and Azure 235
  12 Command-Line Programming 259
  13 Testing 277

Index 287
Contents

Introduction 1

I: Learning to Walk 7

1 Getting Started 9
   Installing Node.js 9
      Installation on Windows 9
      Installation on the Mac 12
      Installation on Linux 14
   Running Node.js and “Hello World!” 15
      The Node Shell 15
      Editing and Running JavaScript Files 16
   Your First Web Server 16
   Debugging Your Node.js Programs 18
   Staying Up-to-Date and Finding Help 21
   Summary 22

2 A Closer Look at JavaScript 23
   Types 23
      Type Basics 24
      Constants 24
      Numbers 25
      Booleans 26
      Strings 27
      Objects 30
      Arrays 32
   Type Comparisons and Conversions 36
   Functions 37
      Basics 37
      Function Scope 40
      Arrow Functions 41
   Language Constructs 41
   Classes, Prototypes, and Inheritance 43
      Prototypes and Inheritance 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching for Modules</td>
<td>93</td>
</tr>
<tr>
<td>Module Caching</td>
<td>94</td>
</tr>
<tr>
<td>Cycles</td>
<td>94</td>
</tr>
<tr>
<td>Writing Modules</td>
<td>95</td>
</tr>
<tr>
<td>Creating Your Module</td>
<td>95</td>
</tr>
<tr>
<td>Developing with Your Module</td>
<td>101</td>
</tr>
<tr>
<td>Publishing Your Modules</td>
<td>102</td>
</tr>
<tr>
<td>Managing Asynchronous Code</td>
<td>103</td>
</tr>
<tr>
<td>The Problem</td>
<td>103</td>
</tr>
<tr>
<td>Our Preferred Solution—<code>async</code></td>
<td>104</td>
</tr>
<tr>
<td>Making Promises and Keeping Them</td>
<td>111</td>
</tr>
<tr>
<td>Summary</td>
<td>113</td>
</tr>
</tbody>
</table>

### 6 Expanding Your Web Server  115

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serving Static Content with Streams</td>
<td>115</td>
</tr>
<tr>
<td>Reading a File</td>
<td>116</td>
</tr>
<tr>
<td>Serving Static Files in a Web Server with Buffers</td>
<td>117</td>
</tr>
<tr>
<td>Serving Up More Than Just HTML</td>
<td>120</td>
</tr>
<tr>
<td>Assembling Content on the Client: Templates</td>
<td>122</td>
</tr>
<tr>
<td>The HTML Skeleton Page</td>
<td>124</td>
</tr>
<tr>
<td>Serving Static Content</td>
<td>125</td>
</tr>
<tr>
<td>Modifying Your URL Scheme</td>
<td>126</td>
</tr>
<tr>
<td>The JavaScript Loader/Bootstrapper</td>
<td>128</td>
</tr>
<tr>
<td>Templating with Mustache</td>
<td>129</td>
</tr>
<tr>
<td>Your Home Page Mustache Template</td>
<td>131</td>
</tr>
<tr>
<td>Putting It All Together</td>
<td>131</td>
</tr>
<tr>
<td>Summary</td>
<td>134</td>
</tr>
</tbody>
</table>

### III: Writing Web Applications  135

#### 7 Building Web Applications with Express  137

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing Express</td>
<td>137</td>
</tr>
<tr>
<td>Hello World in Express</td>
<td>138</td>
</tr>
<tr>
<td>Routing and Layers in Express</td>
<td>139</td>
</tr>
<tr>
<td>Routing Basics</td>
<td>140</td>
</tr>
<tr>
<td>Updating Your Photo Album App for Routing</td>
<td>141</td>
</tr>
</tbody>
</table>
Contents

Updating Your Photo Albums App  175
  Writing the Low-Level Operations  175
  Modifying the API for the JSON Server  181
  Updating Your Handlers  182
  Adding Some New Pages to the Application  188
Recapping the App Structure  192
Summary  192

9 Databases II: SQL (MySQL)  193
  Getting Ready  193
    Installing MySQL  193
    Adding the mysql Module from npm  194
  Creating a Schema for the Database  194
  Basic Database Operations  195
    Connecting  195
    Adding Queries  196
  Updating the Photo Sharing Application  197
  Authenticating via the Database  198
    Updating the API to Support Users  198
    Examining the Core User Data Operations  198
    Updating the Express Application for Authentication  202
    Implementing User Authentication  203
    Creating the User Handler  205
    Hooking up Passport and Routes  207
    Creating the Login and Register Pages  208
Summary  211

IV: Getting the Most Out of Node.js  213

10 Deployment and Development I: Rolling Your Own  215
  Deployment  215
    Level: Basic  216
    Level: Ninja  218
  Multiprocessor Deployment: Using a Proxy  220
    Multiple Servers and Sessions  223
  Virtual Hosting  227
    Express Support  227
Securing Your Projects with HTTPS/SSL 229
  Generating Test Certificates 229
  Built-in Support 230
  Proxy Server Support 231
Multiplatform Development 232
  Locations and Configuration Files 232
  Handling Path Differences 233
Summary 233

11 Deployment and Development II: Heroku and Azure 235
Deploying to Heroku 235
  Before We Begin 236
  Preparing Your Deployment 236
  Create and Deploy the Application on Heroku 238
  We Have a Problem 241
  And That’s It! 244
Deploying to Microsoft Azure 244
  Before We Begin 244
  Preparing Your Deployment 245
  Create and Deploy the Application on Azure 247
  We Have a Problem (and Déjà Vu!) 252
  And That’s It! 256
Summary 257

12 Command-Line Programming 259
Running Command-Line Scripts 259
  UNIX and Mac 259
  Windows 260
  Scripts and Parameters 261
Working with Files Synchronously 262
  Basic File APIs 263
  Files and Stats 264
  Listing Contents of Directories 265
Interacting with the User: stdin/stdout 266
  Basic Buffered Input and Output 266
  Unbuffered Input 267
  The readline Module 268
13 Testing 277
   Choosing a Framework  277
      Installing Nodeunit  278
   Writing Tests  278
      Simple Functional Tests  279
      Groups of Tests  281
      Testing Asynchronous Functionality  282
   API Testing  283
   Summary  286

Index  287
Acknowledgments

I’d like to thank all the Marks at Pearson (it’s a common name, it seems) who have helped me make this book and other projects a reality. The copy editors have been brilliant and helpful.

A huge debt of gratitude is due to Gustavo Moreira for his excellent technical and style reviews.

And finally, much love to Tina, for making it all worthwhile.

About the Author

Marc Wandschneider is a software developer who has spent much time working on scalable web applications and responsive mobile apps. A graduate of McGill University’s School of Computer Science, he spent five years at Microsoft, developing and managing developers on the Visual Basic, Visual J++, and .NET Windows Forms teams. As a Software Developer/Architect at SourceLabs, he built the SWiK open source Wiki platform and then co-founded Adylitica in Beijing. He currently works for Google in London. He authored *PHP and MySQL LiveLessons* and *Core Web Application Development with PHP and MySQL*. 
Introduction

Welcome to Learning Node.js. Node.js is an exciting platform for writing applications of all sorts, ranging from powerful web applications to simple scripts you can run on your local computer. The project has grown from a reasonably small software package managed by one company into a production-quality system governed by a Technical Steering Committee (TSC) and has a sizeable following in the developer community. In this book, I teach you more about it and why it is special, then get you up and writing Node.js programs in short order. You'll soon find that people are rather flexible with the name of Node.js and will refer to it frequently as just Node or even node. I certainly do a lot of that in this book as well.

Why Node.js?

Node.js has arisen for a couple of primary reasons, which I explain next.

The Web

In the past, writing web applications was a pretty standard process. You have one or more servers on your machine that listen on a port (for example, 80 for HTTP), and when a request is received, it forks a new process or a thread to begin processing and responding to the query. This work frequently involves communicating with external services, such as a database, memory cache, external computing server, or even just the file system. When all this work is finally finished, the thread or process is returned to the pool of available servers, and more requests can be handled.

It is a reasonably linear process, easy to understand and straightforward to code. There are, however, a couple of disadvantages that continue to plague the model:

1. Each of these threads or processes carries some overhead with it. On some machines, PHP and Apache can take up as much as 10–15MB per process. Even in environments where a large server runs constantly and forks threads to process the requests, each of these carries some overhead to create a new stack and execution environment, and you frequently run into the limits of the server's available memory.

2. In most common usage scenarios where a web server communicates with a database, caching server, external server, or file system, it spends most of its time sitting around doing nothing and waits for these services to finish and return their responses. While it is sitting there doing nothing, this thread is effectively blocked from doing anything else. The resources it consumes and the process or thread in which it runs are entirely frozen waiting for those responses to come back.
Only after the external component has finally sent back its response will that process or thread be free to finish processing, send a response to the client, and then reset to prepare for another incoming request.

So, although it’s pretty easy to understand and work with, you do have a model that can be quite inefficient if your scripts spend most of their time waiting for database servers to finish running a query—an extremely common scenario for many modern web applications.

Many solutions to this problem have been developed and are in common use. You can buy ever bigger and more powerful web servers with more memory. You can replace more powerful and feature-rich HTTP servers such as Apache with smaller, lightweight ones such as lighttpd or nginx. You can build stripped-down or reduced versions of your favorite web programming language such as PHP or Python (indeed, for a time, Facebook took this one step further and built a system that converts PHP to native C++ code for maximal speed and optimal size). Or you can throw more servers at the problem to increase the number of simultaneous connections you can accommodate.

**New Technologies**

Although the web developers of the world have continued their eternal struggle against server resources and the limits on the number of requests they can process, a few other interesting things have happened in the world.

JavaScript, that old (meaning 1995 or so) language that came to be known mostly (and frequently reviled) for writing client-side scripts in the web browser, has become hugely popular. Modern versions of web browsers are cleaning up their implementations of it and adding new features to make it more powerful and less quirky. With the advent of client libraries for these browsers, such as jQuery, script.aculo.us, or Prototype, programming in JavaScript has become fun and productive. Unwieldy APIs have been cleaned up, and fun, dynamic effects have been added.

At the same time, a new generation of browser competition has erupted, with Google's Chrome, Mozilla's Firefox, Apple's Safari, and Microsoft's Edge all vying for the crown of browser king. As part of this, all these companies are investing heavily in the JavaScript portion of these systems as modern web applications continue to grow ever-more dynamic and script-based. In particular, Google Chrome's V8 JavaScript runtime is particularly fast and also open sourced for use by anybody.

With all these things in place, the opportunity arose for somebody to come along with a new approach to network (web) application development. Thus, the birth of Node.js.

**What Exactly Is Node.js?**

In 2009, a fellow named Ryan Dahl was working for Joyent, a cloud and virtualization services company in California. He was looking to develop push capabilities for web applications, similar to how Gmail does it, and found most of what he looked at not quite appropriate. He eventually settled on JavaScript because it lacked a robust input/output (I/O)
model (meaning he could write his own new one), and had the fast and fully programmable V8 runtime readily available.

Inspired by some similar projects in the Ruby and Python communities, he eventually took the Chrome V8 runtime and an event-processing library called libev and came up with the first versions of a new system called Node.js. The primary methodology or innovation in Node.js is that it is built entirely around an event-driven, nonblocking model of programming. In short, you never (well, rarely) write code that blocks.

If your web application—in order to process a request and generate a response—needs to run a database query, it runs the request and then tells Node.js what to do when the response returns. In the meantime, your code is free to start processing other incoming requests or, indeed, do any other task it might need, such as cleaning up data or running analyses.

Through this simple change in the way the application handles requests and work, you are able to easily write web servers that can handle hundreds, if not thousands, of requests simultaneously on machines without much processing or memory resources. Node runs in a single process, and your code executes largely in a single thread, so the resource requirements are much lower than for many other platforms.

This speed and capacity come with a few caveats, however, and you need to be fully aware of them so you can start working with Node with your eyes wide open.

First and foremost, the new model is different from what you may have seen before and can sometimes be a bit confusing. Until you’ve wrapped your brain fully around some of the core concepts, some code you see written in Node.js can seem a bit strange. Much of this book is devoted to discussing the core patterns many programmers use to manage the challenges of the asynchronous, nonblocking way of programming that Node uses and how to develop your own.

Another limitation with this model of programming is that it really is centered around applications that are doing lots of different things with lots of different processes, servers, or services. Node.js truly shines when your web application is juggling connections to databases, caching servers, file systems, application servers, and more. The flip side of this, however, is that it’s actually not necessarily an optimal environment for writing compute servers that are doing serious, long-running computations. For these, Node’s model of a single thread in a single process can create problems if a given request is taking a ton of time to generate a complicated password digest or processing an image. In situations in which you’re doing more computationally intensive work, you need to be careful how your applications use resources or perhaps even consider farming those tasks out to other platforms and run them as a service for your Node.js programs to call.

Node.js’s path to adulthood has been a somewhat rocky one—the 0.x series of Node.js lingered for quite a while, releasing often but seemingly not making much progress, and some grew impatient with the governance of the project. This caused a schism in late 2014, with a group of people forking the open sourced code and creating io.js, a new version of node with the goals of being more open and transparent and responsive to the developer community. Fortunately, this break did not last long, and within nine months, Joyent agreed to hand over guidance of Node.js to the Technical Steering Committee (TSC) in autumn 2015.
Today, however, the platform is quite stable and predictable, and has adopted semantic versioning, where your version numbers have the format major.minor.patchlevel. In this model you only make breaking API changes with major version number changes, add features in minor version number changes, and can update and fix anything necessary in patch-level changes. Each major version is developed for 18 months and then supported for another 12 months after that, meaning you have 2.5 years of use for each version. After that, you’ll need (and definitely want) to migrate to the latest version to be sure you’re getting the latest features and most secure version of the software).

To help you keep track of and manage all of these updates, the developers have taken to labeling portions of the system with different degrees of stability, ranging from Unstable to Stable to Locked. Changes to Stable or Locked portions of the runtime are rare and involve much community discussion to determine whether the changes will generate too much pain. As you work your way through this book, we point out which areas are less stable than others and suggest ways you can mitigate the dangers of changing APIs. Newer versions of Node.js have introduced the concept of Deprecated APIs. If part of Node.js is becoming too difficult to maintain and is not heavily used, or otherwise doesn’t make sense to continue supporting, it will (again, after much community discussion) be marked as Deprecated and not included in the next major version update. This gives developers plenty of time to move to alternatives (of which there are always going to be dozens).

The good news is that Node.js already has a large and active user community and a bunch of mailing lists, forums, and user groups devoted to promoting the platform and providing help where needed. A simple Internet search will get you answers to 99 percent of your questions in a matter of seconds, so never be afraid to look!

Who Is This Book For?

I wrote this book under the assumption that you are comfortable programming computers and are familiar with the functionality and syntax of at least one major programming language such as Java, C/C++, PHP, or C#. Although you don’t have to be an expert, you’ve probably moved beyond “Learn X in Y days” level tasks.

If you’re like me, you have probably written some HTML/CSS/JavaScript and thus have “worked with” JavaScript, but you might not be intimately familiar with it and have just largely templated heavily off code found on blog posts or mailing lists. Indeed, because of its clunky UI and frustrating browser mismatches, you might even frown slightly at the mere mention of JavaScript. Fear not—by the end of the first section of this book, distasteful memories of the language will be a thing of the past and, I hope, you’ll be happily writing your first Node.js programs with ease and a smile on your face!

I also assume that you have a basic understanding of how web applications work: browsers send HTTP requests to a remote server; the server processes each request and sends a response with a code indicating success or failure, and then optionally some data along with that response (such as the HTML for the page to render or perhaps JavaScript Object Notation, or JSON, containing data for that request). You’ve probably connected to database servers in the past, run queries, and waited for the resulting rows, and so on. When I start to describe
concepts beyond these in the samples and programs, I explain and refresh everybody’s memory on anything new or uncommon.

How to Use this Book

This book is largely tutorial in nature. I try to balance out explanations with code to demonstrate it as much as possible and avoid long, tedious explanations of everything. For those situations in which I think a better explanation is interesting, I might point you at some resources or other documentation to learn more if you are so inclined (but it is never a necessity).

The book is divided into four major sections:

Part 1. **Learning to Walk**—You start installing and running Node, take another look at the JavaScript language and the extensions used in V8 and Node.js, and then write your first application.

Part 2. **Learning to Run**—You start developing more powerful and interesting application servers in this part of the book, and I start teaching you some of the core concepts and practices used in writing Node.js programs.

Part 3. **Breaking Out the Big Guns**—In this section, you look at some of the powerful tools and modules available to you for writing your web applications, such as help with web servers and communication with database servers.

Part 4. **Getting the Most Out of Node.js**—Finally, I close out the book by looking at a few other advanced topics such as ways in which you can run your applications on production servers, how you can test your code, and how you can use Node.js to write command-line utilities as well!

As you work your way through the book, take the time to fire up your text editor and enter the code, see how it works in your version of Node.js, and otherwise start writing and developing your own code as you go along. You develop your own little photo sharing application as you work through this book, which I hope provides you with some inspiration or ideas for things you can write.

Download the Source Code

Source code for most of the examples and sample projects in this book can be found at [github.com/marcwan/LearningNodeJS](https://github.com/marcwan/LearningNodeJS). You are highly encouraged to download it and play along, but don’t deny yourself the opportunity to type in some of the code as well and try things out.

The GitHub code has some fully functioning samples and has been tested to work on Mac, Linux, and Windows with the latest versions of Node.js. If new updates of Node require updates to the source code, I will put changes and notes there, so please be sure to pull down new versions every few months. Sadly, my code is not perfect, and I always welcome bug reports and pull requests!

If you have any questions or problems with the code in this book, feel free to go to [github.com/marcwan/LearningNodeJS](https://github.com/marcwan/LearningNodeJS) and add an issue; they'll be monitored and answered reasonably quickly.
Part I

Learning to Walk

1 Getting Started  9
2 A Closer Look at JavaScript  23
3 Asynchronous Programming  49
This page intentionally left blank
Asynchronous Programming

Now that you have a refreshed and updated idea of what JavaScript programming is really like, it's time to get into the core concept that makes Node.js what it is: nonblocking IO and asynchronous programming. It carries with it some huge advantages and benefits, which you shall soon see, but it also brings some complications and challenges with it.

The Old Way of Doing Things

In the olden days (2008 or so), when you sat down to write an application and needed to load in a file, you would write something like the following (let's assume you're using something vaguely PHP-ish for the purposes of this example):

```php
$file = fopen('info.txt', 'r');
// wait until file is open

$content = fread($file, 100000);
// wait until contents are read

// do something with those contents
```

If you were to analyze the execution of this script, you would find that it spends a vast majority of its time doing nothing at all. Indeed, most of the clock time taken by this script is spent waiting for the computer's file system to do its job and return the file contents you requested. Let me generalize things a step further and state that for most IO-based applications—those that frequently connect to databases, communicate with external servers, or read and write files—your scripts will spend a majority of their time sitting around waiting (see Figure 3.1).
The way your servers process multiple requests at the same time by running many of these scripts in parallel. Modern computer operating systems are great at multitasking, so you can easily switch out processes that are blocked and let other processes have access to the CPU. Some environments take things a step further and use threads instead of processes.

The problem is that for each of these processes or threads, there is some amount of overhead. For heavier implementations using Apache and PHP, I have seen up to 10–15MB of memory overhead per process—never mind the resources and time consumed by the operating system switching that context in and out constantly. That's not even 100 simultaneously executing servers per gigabyte of RAM! Threaded solutions and those using more lightweight HTTP servers do, of course, have better results, but you still end up in a situation in which the computer spends most of its time waiting around for blocked processes to get their results, and you risk running out of capacity to handle incoming requests.

It would be nice if there were some way to make better use of all the available CPU power and available memory so as not to waste so much. This is where Node.js shines.
To understand how Node.js changes the method demonstrated in the preceding section into a nonblocking, asynchronous model, first look at the `setTimeout` function in JavaScript. This function takes a function to call and a timeout after which it should be called:

```javascript
setTimeout(() => {
    console.log("I've done my work!");
}, 2000);
```

```javascript
console.log("I'm waiting for all my work to finish.");
```

If you run the preceding code, you see the following output:

```
I'm waiting for all my work to finish.
I've done my work!
```

I hope this is not a surprise to you: The program sets the timeout for 2000 ms (2 seconds), giving it the function to call when it fires, and then continues with execution, which prints out the “I’m waiting…” text. Two seconds later, you see the “I’ve done…” message, and the program then exits.

Now, look at a world where any time you call a function that needs to wait for some external resource (database server, network request, or file system read/write operation), it has a similar signature. That is, instead of calling `fopen(path, mode)` and waiting, you would instead call `fopen(path, mode, (file_handle) => { ... })`.

Now rewrite the preceding synchronous script using the new asynchronous functions. You can actually enter and run this program with `node` from the command line. Just make sure you also create a file called `info.txt` that can be read.

```javascript
var fs = require('fs');                           // We'll explain this below
var file;
var buf = new Buffer(100000);
fs.open('info.txt', 'r', (err, handle) => {
    file = handle;
});

// fs.read needs the file handle returned by fs.open. But this is broken.
fs.read(file, buf, 0, 100000, null, (err, length) => {
    console.log(buf.toString());
    fs.close(file, () => { /* don't care */ });
});
```

The first line of this code is something you haven't seen just yet: the `require` function is a way to include additional functionality in your Node.js programs. Node comes with a pretty impressive set of modules, each of which you can include separately as you need functionality.
will work further with modules frequently from now on; you learn about consuming them and writing your own in Chapter 5, “Modules.”

If you run this program as it is, it throws an error and terminates. How come? Because the \texttt{fs.open} function runs \emph{asynchronously}; it returns immediately, before the file has been opened and the callback function invoked. The \texttt{file} variable is not set until the file has been opened and the handle to it has been passed to the callback specified as the third parameter to the \texttt{fs.open} function. Thus, you are trying to access an \texttt{undefined} variable when you try to call the \texttt{fs.read} function with it immediately afterward.

Fixing this program is easy:

\begin{verbatim}
var fs = require('fs');

fs.open('info.txt', 'r', (err, handle) => {
  var buf = new Buffer(100000);
  fs.read(handle, buf, 0, 100000, null, (err, length) => {
    console.log(buf.toString('utf8', 0, length));
    fs.close(handle, () => { /* Don't care */ });
  });
});
\end{verbatim}

The key way to think of how these asynchronous functions work internally in Node is something along the following lines:

- Check and validate parameters.
- Tell the Node.js core to queue the call to the appropriate function for you (in the preceding example, the operating system \texttt{open} or \texttt{read} function) and to notify (call) the provided callback function when there is a result.
- Return to the caller.

You might be asking: if the \texttt{open} function returns right away, why doesn't the node process exit immediately after that function has returned? The answer is that Node operates with an \emph{event queue}; if there are pending events for which you are awaiting a response, it does not exit until your code has finished executing \emph{and} there are no events left on that queue. If you are waiting for a response (either to the \texttt{open} or the \texttt{read} function calls), it waits. See Figure 3.2 for an idea of how this scenario looks conceptually.
Error Handling and Asynchronous Functions

In the preceding chapter, I discussed error handling and events as well as the `try / catch` block in JavaScript. The addition of nonblocking IO and asynchronous
function callbacks in this chapter, however, creates a new problem. Consider the following code:

try {
    setTimeout(() => {
        throw new Error("Uh oh!");
    }, 2000);
} catch (e) {
    console.log("I caught the error: " + e.message);
}

If you run this code, you might very well expect to see the output "I caught the error: Uh oh!". But you do not. You actually see the following:

timers.js:103
    if (!process.listeners('uncaughtException').length) throw e;

    Error: Uh oh, something bad!
    at Object._onTimeout errors_async.js:5:15
    at Timer.list.ontimeout (timers.js:101:19)

What happened? Did I not say that try / catch blocks were supposed to catch errors for you? I did, but asynchronous callbacks throw a new little wrench into this situation.

In reality, the call to setTimeout does execute within the try / catch block. If that function were to throw an error, the catch block would catch it, and you would see the message that you had hoped to see. However, the setTimeout function just adds an event to the Node event queue (instructing it to call the provided function after the specified time interval—2000 ms in this example) and then returns. The provided callback function actually operates within its own entirely new context and scope!

As a result, when you call asynchronous functions for nonblocking IO, very few of them throw errors, but instead use a separate way of telling you that something has gone wrong.

In Node, you use a number of core patterns to help you standardize how you write code and avoid errors. These patterns are not enforced syntactically by the language or runtime, but you will see them used frequently and should absolutely use them yourself.

The callback Function and Error Handling

One of the first patterns you will see is the format of the callback function you pass to most asynchronous functions. It always has at least one parameter, the success or failure status of the last operation, and very commonly a second parameter with some sort of additional results or information from the last operation (such as a file handle, database connection, rows from a query, and so on); some callbacks are given even more than two:

do_something(param1, param2, ..., paramN, function (err, results) { ... });
The *err* parameter is either

- null, indicating the operation was a success, and (if there should be one) there will be a result.
- An instance of the Error object class. You will occasionally notice some inconsistency here, with some people always adding a code field to the Error object and then using the message field to hold a description of what happened, whereas others have chosen other patterns. For all the code you write in this book, you will follow the pattern of always including a code field and using the message field to provide as much information as you can. For all the modules you write, you will use a string value for the code because strings tend to be a bit easier to read. Some libraries provide extra data in the Error object with additional information, but at least the two members should always be there.

This standard prototype methodology enables you to always write predictable code when you are working with nonblocking functions. Throughout this book, I demonstrate two common coding styles for handling errors in callbacks. Here’s the first:

```javascript
fs.open('info.txt', 'r', (err, handle) => {
  if (err) {
    console.log("ERROR: " + err.code + " (" + err.message + ")");
    return;
  }
  // success!! continue working here
});
```

In this style, you check for errors and return if you see one; otherwise, you continue to process the result. And now here’s the other way:

```javascript
fs.open('info.txt', 'r', (err, handle) => {
  if (err) {
    console.log("ERROR: " + err.code + " (" + err.message + ")");
  } else {
    // success! continue working here
  }
});
```

In this method, you use an if ... then ... else statement to handle the error.

The difference between these two may seem like splitting hairs, but the former method is a little more prone to bugs and errors for those cases when you forget to use the return statement inside the if statement, whereas the latter results in code that indents itself much more quickly and you end up with lines of code that are quite long and less readable. We’ll look at a solution to this second problem in the section titled “Managing Asynchronous Code” in Chapter 5.
A fully updated version of the file loading code with error handling is shown in Listing 3.1.

**Listing 3.1  File Loading with Full Error Handling**

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

fs.open('info.txt', 'r', (err, handle) => {
  if (err) {
    console.log("ERROR: " + err.code + " (" + err.message + ")");
    return;
  }
  var buf = new Buffer(100000);
  fs.read(handle, buf, 0, 100000, null, (err, length) => {
    if (err) {
      console.log("ERROR: " + err.code + " (" + err.message + ")");
      return;
    }
    console.log(buf.toString('utf8', 0, length));
    fs.close(handle, () => { /* don't care */ });
  });
});
```

**Who Am I? Maintaining a Sense of Identity**

Now you’re ready to write a little class to help you with some common file operations:

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

function FileObject () {
  this.filename = ''; 

  this.file_exists = function (callback) {
    console.log("About to open: " + this.filename);
    fs.open(this.filename, 'r', function (err, handle) {
      if (err) {
        console.log("Can't open: " + this.filename);
        callback(err);
        return;
      }
      fs.close(handle, function () {
        /* don't care */
      });
    });
  };
}
```
You have currently added one property, `filename`, and a single method, `file_exists`. This method does the following:

- It tries to open the file specified in the `filename` property read-only.
- If the file doesn’t exist, it prints a message and calls the callback function with the error info.
- If the file does exist, it calls the callback function indicating success.

Now, run this class with the following code:

```javascript
var fo = new FileObject();
fo.filename = "file_that_does_not_exist";

fo.file_exists((err, results) => {
  if (err) {
    console.log("Error opening file: " + JSON.stringify(err));
    return;
  }
  console.log("file exists!!!");
});
```

You might expect the following output:

About to open: file_that_does_not_exist
Can't open: file_that_does_not_exist

But, in fact, you see this:

About to open: file_that_does_not_exist
Can't open: undefined

What happened? Most of the time, when you have a function nested within another, it inherits the scope of its parent/host function and should have access to all the same variables. So why does the nested callback function not get the correct value for the `filename` property?

The problem lies with the `this` keyword and asynchronous callback functions. Don’t forget that when you call a function like `fs.open`, it initializes itself, calls the underlying operating system function (in this case to open a file), and places the provided callback function on the event queue. Execution immediately returns to the `FileObject#file_exists` function, and then you exit. When the `fs.open` function completes its work and Node runs the callback, you no longer have the context of the `FileObject` class any more, and the callback function is given a new `this` pointer representing some other execution context!

The bad news is that you have, indeed, lost your `this` pointer referring to the `FileObject` class. The good news is that the callback function for `fs.open` does still have its function scope. A common solution to this problem is to “save” the disappearing `this` pointer in a variable called `self` or `me` or something similar. Now rewrite the `file_exists` function to take advantage of this:

```javascript
this.file_exists = function (callback) {
  var self = this;
```
console.log("About to open: " + self.filename);
fs.open(this.filename, 'r', function (err, handle) {
  if (err) {
    console.log("Can't open: " + self.filename);
    callback(err);
    return;
  }

  fs.close(handle, function () { });
  callback(null, true);
});

Because local function scope is preserved via closures, the new `self` variable is maintained for you even when your callback is executed asynchronously later by Node.js. You will make extensive use of this in all your applications. Some people like to use `me` instead of `self` because it is shorter; others still use completely different words. Pick whatever kind you like and stick with it for consistency.

The above scenario is another reason to use `arrow functions`, introduced in the previous chapter. Arrow functions capture the `this` value of the enclosing scope, so your code actually works as expected! Thus, as long as you are using `=>`, you can continue to use the `this` keyword, as follows:

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

function FileObject () {
  this.filename = '';

  // Always use "function" for member fns, not =>, see below for why
  this.file_exists = function (callback) {
    console.log("About to open: " + this.filename);
    fs.open(this.filename, 'r', (err, handle) => {
      if (err) {
        console.log("Can't open: " + this.filename);
        callback(err);
        return;
      }

      fs.close(handle, () => { });
      callback(null, true);
    });
  }
}
```

One other thing to note is that we do not use arrow functions for declaring member functions on objects or prototypes. This is because in those cases, we actually do want the `this` variable to update with the context of the currently executing object. Thus, you’ll see us using `=>` only when we’re using anonymous functions in other contexts.
The key takeaway for this section should be: If you’re using an anonymous function that’s not a class or prototype method, you should stop and think before using this. There’s a good chance it won’t work the way you want. Use arrow functions as much as possible.

**Being Polite—Learning to Give Up Control**

Node runs in a single thread with a single event loop that makes calls to external functions and services. It places callback functions on the event queue to wait for the responses and otherwise tries to execute code as quickly as possible. So what happens if you have a function that tries to compute the intersection between two arrays:

```javascript
function compute_intersection(arr1, arr2, callback) {
  var results = [];
  for (var i = 0; i < arr1.length; i++) {
    for (var j = 0; j < arr2.length; j++) {
      if (arr2[j] == arr1[i]) {
        results[results.length] = arr2[j];
        break;
      }
    }
  }
  callback(null, results);  // no error, pass in results!
}
```

For arrays of a few thousand elements, this function starts to consume significant amounts of time to do its work, on the order of a second or more. In a single-threaded model, where Node.js can do only one thing at a time, this amount of time can be a problem. Similar functions that compute hashes, digests, or otherwise perform expensive operations are going to cause your applications to temporarily “freeze” while they do their work. What can you do?

In the introduction to this book, I mentioned that there are certain things for which Node.js is not particularly well suited, and one of them is definitely acting as a compute server. Node is far better suited to more common network application tasks, such as those with heavy amounts of IO and requests to other services. If you want to write a server that does a lot of expensive computations and calculations, you might want to consider moving these operations to other services that your Node applications can then call remotely.

I am not saying, however, that you should completely shy away from computationally intensive tasks. If you’re doing these only some of the time, you can still include them in Node.js and take advantage of a method on the `process` global object called `nextTick`. This method basically says “Give up control of execution, and then when you have a free moment, call the provided function.” It tends to be significantly faster than just using the `setTimeout` function.

Listing 3.2 contains an updated version of the `compute_intersection` function that yields every once in a while to let Node process other tasks.
Listing 3.2  Using Process#nextTick to be Polite

```javascript
function compute_intersection(arr1, arr2, callback) {
  // let's break up the bigger of the two arrays
  var bigger = arr1.length > arr2.length ? arr1 : arr2;
  var smaller = bigger == arr1 ? arr2 : arr1;
  var biglen = bigger.length;
  var smlen = smaller.length;

  var sidx = 0;           // starting index of any chunk
  var size = 10;          // chunk size, can adjust!
  var results = [];       // intermediate results

  // for each chunk of "size" elements in bigger, search through smaller
  function sub_compute_intersection() {
    for (var i = sidx; i < (sidx + size) && i < biglen; i++) {
      for (var j = 0; j < smlen; j++) {
        if (bigger[i] == smaller[j]) {
          results.push(smaller[j]);
          break;
        }
      }
    }
    if (i >= biglen) {
      callback(null, results);   // no error, send back results
    } else {
      sidx += size;
      process.nextTick(sub_compute_intersection);
    }
  }

  sub_compute_intersection();
}
```

In this new version of the function, you basically divide the bigger of the input arrays into chunks of 10 (you can choose whatever number you want), compute the intersection of that many items, and then call `process#nextTick` to allow other events or requests a chance to do their work. Only when there are no events in front of you any longer, will you continue to do the work. Don’t forget that passing the callback function `sub_compute_intersection` to `process#nextTick` ensures that the current scope is preserved as a closure, so you can store the intermediate results in the variables in `compute_intersection`.

Listing 3.3 shows the code you use to test this new `compute_intersection` function.
Listing 3.3  Testing the compute_intersection Function

```javascript
var a1 = [ 3476, 2457, 7547, 34523, 3, 6, 7,2, 77, 8, 2345,
    7623457, 2347, 23572457, 237457, 234869, 237,
    24572457524 ];
var a2 = [ 3476, 75347547, 2457634563, 56763472, 34574, 2347,
    7, 34652364 , 13461346, 572346, 23723457234, 237,
    234, 24352345, 537, 2345235, 2345675, 34534,
    7582768, 284835, 8553577, 2577257,545634, 457247247,
    2345 ];

compute_intersection(a1, a2, function (err, results) {
  if (err) {
    console.log(err);
  } else {
    console.log(results);
  }
});
```

Although this has made things a bit more complicated than the original version of the function to compute the intersections, the new version plays much better in the single-threaded world of Node event processing and callbacks, and you can use `process.nextTick` in any situation in which you are worried that a complex or slow computation is necessary.

**Synchronous Function Calls**

Now that I have spent nearly an entire chapter telling you how Node.js is very much asynchronous and about all the tricks and traps of programming nonblocking IO, I must mention that Node actually does have synchronous versions of some key APIs, most notably file APIs. You use them for writing command-line tools in Chapter 12, “Command-Line Programming.”

To demonstrate briefly here, you can rewrite the first script of this chapter as follows:

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

var handle = fs.openSync('info.txt', 'r');
var buf = new Buffer(100000);
var read = fs.readSync(handle, buf, 0, 10000, null);
console.log(buf.toString('utf8', 0, read));
fs.closeSync(handle);
```

As you work your way through this book, I hope you are able to see quite quickly that Node.js isn’t just for network or web applications. You can use it for everything from command-line utilities to prototyping to server management and more!
Summary

Switching from a model of programming where you execute a sequence of synchronous or blocking IO function calls and wait for each of them to complete before moving on to the next call, to a model where you do everything asynchronously and wait for Node to tell you when a given task is done requires a bit of mental gymnastics and experimentation. But I am convinced that when you get the hang of this, you'll never be able imagine going back to the other way of writing your web apps.

Next, you write your first simple JSON application server.
Symbols

__proto__ property, 44–45
+ (plus sign) operator, 28

A

accessing parameters, 262
account signup
    Azure, 244
    Heroku, 236
adding
    pages to applications, 188–191
    photos to albums, 180–181, 186–188, 190–191
albums
    creating, 177–179, 183, 188–190
    finding, 179, 184
    listing, 179, 184–185
    photos
        adding, 180–181, 186–188, 190–191
        finding, 180, 185–186
anonymous functions, 39–40
    writing with arrow functions, 41
API design, 144–145
modifying
    for authentication, 198
    for database usage, 181
testing, 283–286
applications. See web applications
arguments, 37
array literal syntax, 32
arrays, 32–35
functions, 34–35
arrow functions, 41
this keyword, 58–59
assert function, 47
assert module, 280
async module, 104–110
asynchronous functions
changing synchronous programming
error handling and, 53–54
managing
with async module, 104–110
problems with, 103–104
with promises pattern, 111–113
this keyword, 56–59
asynchronous looping, 69–71, 110
asynchronous tests, 282
authentication, 157–160
creating login forms, 160–161
flash messages, 162–163
logging in, 161
with MySQL, 198
creating login and registration
pages, 208–211
creating user handlers, 205–207
creating users, 199–201
fetching users, 201–202
implementing, 203–204
modifying API, 198
routing in, 207–208
updating express application,
202–203
restricting page access, 162
auto function, 108–110
autogenerated _id fields, searching for, 174

Azure, 244
account signup, 244
applications
cloud storage in, 252–256
creating, 247–248
deploying, 248
configuring ClearDB add-on, 248–252
downloading CLI tools, 244–245
logging in, 245–246
preparing for deployment, 245–247

B
bcrypt module, 200
BDD (behavior-driven development), 277
bitwise operators, 41
blocking IO, 49–50
changing to nonblocking IO, 51–53
bluebird module, 111–113
booleans, 26
bootstrapper (JavaScript), 128–129
for login and registration pages,
208–211
buffered I/O, 266–267
buffers
serving static content, 117–120
strings versus, 117

caching modules, 94
callback functions
error handling and, 54–56
loops and, 70–71
this keyword, 56–59
Chai, 283
child_process module
exec function, 273–274
spawn function, 274–276
choosing testing frameworks, 277–278

classes, 43
  constructors, 91
  creating event classes, 121–122
  prototypes and inheritance, 43–45

ClearDB add-on, configuring
  in Azure applications, 248–252
  in Heroku applications, 239–241

client-side templates, 122–124
  adding pages, 188–191
  with Mustache, 129–131
  in sample application, 131–134

cloning objects, 179

cloud storage
  in Azure applications, 252–256
  in Heroku applications, 241–244

Cloudinary
  for Azure applications, 252–256
  for Heroku applications, 241–244

collections
  creating, 169–170
  deleting documents, 172
  inserting documents, 170–171
  querying, 172–175
  updating documents, 171–172

command-line scripts. See scripts

comparisons of types, 36–37

compressing output, 156–157

compute servers, 59

configuration files
  creating, 175–176
  in multiplatform development, 232–233

configurations, middleware, 149

configuring ClearDB add-on
  in Azure applications, 248–252
  in Heroku applications, 239–241

connecting
  to memcached, 226–227
  to MongoDB databases, 169
    in sample application, 169
  to MySQL databases, 195–196, 198–199

connection pooling, 196

console object, 47

custom.error function, 266

console.log function, 266

constants, 24

constructors, 91

control, yielding, 59–61

conversions of types, 36–37

cookies with express middleware, 153–155

cURL, 17
  sending POST data, 82–83

cwd function, 47

cycles in modules, 94–95

data structure
  in MongoDB, 167
    in sample application, 192

data types in MongoDB, 168

Databases

MongoDB
  connecting to, 169, 176–177
  creating collections, 169–170
  creating databases, 169, 176–177
  data structure, 167
  data types, 168
  deleting documents from collections, 172
  inserting documents into collections, 170–171
  installing, 165–166
  low-level operations in sample
databases

application, 175–181
Node.js usage, 166–167
querying collections, 172–175
updating documents in collections, 171–172
MySQL
authentication, 198–211
configuring ClearDB add-on, 239–241, 248–252
connecting to, 195–196, 198–199
creating schema, 194–195
installing, 193–194
Node.js usage, 194
querying, 196–197
documentation for modules, 96
downloading
Azure CLI tools, 244–245
Heroku CLI tools, 236
source code, 5
URL contents on Windows, 17
dynos, 235
deleting documents from collections, 172
deploying web applications, 215
to Azure, 244–256
basic deployment, 216–217
to Heroku, 235–244
on Linux/Mac, 218–219
multiprocessor deployment, 220–223
problems with basic deployment, 218
sessions on multiple servers, 223–227
virtual hosting, 227–229
on Windows, 219–220
development, multiplatform, 232
locations and configuration files, 232–233
paths, 233
directories
creating, 264–265
listing contents, 265
delimiting numbers in collections
deleting, 172
inserting, 170–171
updating, 171–172
downloading
Azure CLI tools, 244–245
Heroku CLI tools, 236
source code, 5
URL contents on Windows, 17
dynos, 235
deleting documents from collections
deleting, 172
downloading
Azure CLI tools, 244–245
Heroku CLI tools, 236
source code, 5
URL contents on Windows, 17
dynos, 235
deploying web applications
to Azure, 244–256
basic deployment, 216–217
to Heroku, 235–244
on Linux/Mac, 218–219
multiprocessor deployment, 220–223
problems with basic deployment, 218
sessions on multiple servers, 223–227
virtual hosting, 227–229
on Windows, 219–220
development, multiplatform, 232
locations and configuration files, 232–233
paths, 233
directories
creating, 264–265
listing contents, 265
documentation for modules, 96
functions

ordering, 150–151
POST data, cookies, sessions, 153–155
PUT and DELETE support, 156
static file handling, 151–152
usage, 148–149
routing in, 140–141
for authentication, 207–208
updating sample application, 141–144
updating for authentication, 202–203
virtual hosting support, 227–229
web servers, creating, 138–139

F

factory functions, 91
fetching users, 201–202
file operations in scripts, 263–264
files
configuration files, creating, 175–176
reading with streams, 116
uploading, 154–155
finding
albums, 179, 184
photos, 180, 185–186
flash messages, 162–163
forEach function, 35, 110
forEachSeries function, 110
for.in loops, 42
forms, receiving POST data, 86–87
for.of loops, 42
frameworks for testing
installing Mocha, 283
installing nodeunit, 278
selecting, 277–278
fs.open function, 52
fs.readFile function, 128
fs.stat function, 69–71
functional tests, 279–281
functions
array functions, 34–35
arrow functions, 41
this keyword, 58–59
asynchronous functions
changing synchronous programming to, 51–53
error handling and, 53–54
managing with async module, 104–110
managing with promises pattern, 111–113
problems managing, 103–104
this keyword, 56–59
callback functions
error handling and, 54–56
this keyword, 56–59
classes, 43
prototypes and inheritance, 43–45
explained, 37–40
factory functions, 91
I/O functions
buffered I/O, 266–267
readline module, 268–273
stdin, stdout, stderr, 266
unbuffered input, 267–268
scope, 40
string functions, 28–29
synchronous functions, 61, 262
creating directories, 264–265
file operations, 263–264
listing directory contents, 265
yielding control, 59–61
<table>
<thead>
<tr>
<th>GET params, 79–82</th>
</tr>
</thead>
<tbody>
<tr>
<td>global object, 46</td>
</tr>
<tr>
<td>global variables</td>
</tr>
<tr>
<td>console, 47</td>
</tr>
<tr>
<td>global, 46</td>
</tr>
<tr>
<td>process, 47</td>
</tr>
<tr>
<td>groups of tests, 281–282</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>handlers</th>
</tr>
</thead>
<tbody>
<tr>
<td>updating in sample application, 182–188</td>
</tr>
<tr>
<td>user handlers, creating, 205–207</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>help resources, 21–22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heroku, 235–236</td>
</tr>
<tr>
<td>account signup, 236</td>
</tr>
<tr>
<td>applications</td>
</tr>
<tr>
<td>cloud storage in, 241–244</td>
</tr>
<tr>
<td>creating, 238–239</td>
</tr>
<tr>
<td>testing, 239</td>
</tr>
<tr>
<td>configuring ClearDB add-on, 239–241</td>
</tr>
<tr>
<td>downloading CLI tools, 236</td>
</tr>
<tr>
<td>logging in, 236</td>
</tr>
<tr>
<td>preparing for deployment, 236–238</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>History of Node.js, 2–4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML skeleton pages, 124–125</td>
</tr>
<tr>
<td>http module, 17</td>
</tr>
<tr>
<td>HTTP POST data. See POST data</td>
</tr>
<tr>
<td>HTTP response codes, 79</td>
</tr>
<tr>
<td>HTTPS, 229</td>
</tr>
<tr>
<td>built-in support for, 230–231</td>
</tr>
<tr>
<td>generating test certificates, 229–230</td>
</tr>
<tr>
<td>proxy server support, 231–232</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I/O functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffered I/O, 266–267</td>
</tr>
<tr>
<td>readline module, 268–273</td>
</tr>
<tr>
<td>stdin, stdout, stderr, 266</td>
</tr>
<tr>
<td>unbuffered input, 267–268</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isArray function, 33</th>
</tr>
</thead>
<tbody>
<tr>
<td>isFinite function, 26</td>
</tr>
<tr>
<td>isNaN function, 26</td>
</tr>
<tr>
<td>iterable objects, 42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>including modules, 93</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexOf function, 28</td>
</tr>
<tr>
<td>--Infinity value, 25–26</td>
</tr>
<tr>
<td>Infinity value, 25–26</td>
</tr>
<tr>
<td>inheritance, 43–45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input. See also I/O functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>readline module, 268–273</td>
</tr>
<tr>
<td>unbuffered, 267–268</td>
</tr>
</tbody>
</table>

| inserting documents into collections, 170–171 |

<table>
<thead>
<tr>
<th>installing</th>
</tr>
</thead>
<tbody>
<tr>
<td>express, 137–138</td>
</tr>
<tr>
<td>memcached</td>
</tr>
<tr>
<td>on Linux/Mac, 225–226</td>
</tr>
<tr>
<td>on Windows, 225</td>
</tr>
<tr>
<td>Mocha, 283</td>
</tr>
<tr>
<td>modules via NPM, 92–93</td>
</tr>
<tr>
<td>MongoDB, 165–166</td>
</tr>
<tr>
<td>MySQL, 193–194</td>
</tr>
<tr>
<td>Node.js</td>
</tr>
<tr>
<td>on Linux, 14–15</td>
</tr>
<tr>
<td>on Mac, 12–14</td>
</tr>
<tr>
<td>on Windows, 9–12</td>
</tr>
<tr>
<td>nodeunit, 278</td>
</tr>
</tbody>
</table>

| instanceof, 45 |

<table>
<thead>
<tr>
<th>history of Node.js, 2–4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML skeleton pages, 124–125</td>
</tr>
<tr>
<td>http module, 17</td>
</tr>
<tr>
<td>HTTP POST data. See POST data</td>
</tr>
<tr>
<td>HTTP response codes, 79</td>
</tr>
<tr>
<td>HTTPS, 229</td>
</tr>
<tr>
<td>built-in support for, 230–231</td>
</tr>
<tr>
<td>generating test certificates, 229–230</td>
</tr>
<tr>
<td>proxy server support, 231–232</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>including modules, 93</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexOf function, 28</td>
</tr>
<tr>
<td>--Infinity value, 25–26</td>
</tr>
<tr>
<td>Infinity value, 25–26</td>
</tr>
<tr>
<td>inheritance, 43–45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input. See also I/O functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>readline module, 268–273</td>
</tr>
<tr>
<td>unbuffered, 267–268</td>
</tr>
</tbody>
</table>

| inserting documents into collections, 170–171 |

<table>
<thead>
<tr>
<th>installing</th>
</tr>
</thead>
<tbody>
<tr>
<td>express, 137–138</td>
</tr>
<tr>
<td>memcached</td>
</tr>
<tr>
<td>on Linux/Mac, 225–226</td>
</tr>
<tr>
<td>on Windows, 225</td>
</tr>
<tr>
<td>Mocha, 283</td>
</tr>
<tr>
<td>modules via NPM, 92–93</td>
</tr>
<tr>
<td>MongoDB, 165–166</td>
</tr>
<tr>
<td>MySQL, 193–194</td>
</tr>
<tr>
<td>Node.js</td>
</tr>
<tr>
<td>on Linux, 14–15</td>
</tr>
<tr>
<td>on Mac, 12–14</td>
</tr>
<tr>
<td>on Windows, 9–12</td>
</tr>
<tr>
<td>nodeunit, 278</td>
</tr>
</tbody>
</table>

| instanceof, 45 |

<table>
<thead>
<tr>
<th>history of Node.js, 2–4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML skeleton pages, 124–125</td>
</tr>
<tr>
<td>http module, 17</td>
</tr>
<tr>
<td>HTTP POST data. See POST data</td>
</tr>
<tr>
<td>HTTP response codes, 79</td>
</tr>
<tr>
<td>HTTPS, 229</td>
</tr>
<tr>
<td>built-in support for, 230–231</td>
</tr>
<tr>
<td>generating test certificates, 229–230</td>
</tr>
<tr>
<td>proxy server support, 231–232</td>
</tr>
</tbody>
</table>
JavaScript, 2. See also scripts
  bootstrapper, 128–129
    for login and registration pages, 208–211
errors and exceptions, 45–46
functions
  arrow functions, 41
classes, 43
explained, 37–40
prototypes and inheritance, 43–45
scope, 40
global variables
  console, 47
global, 46
  process, 47
MongoDB data structure, 167
operators and constructs, 41–42
running Node.js, 16
types
  arrays, 32–35
booleans, 26
  comparisons and conversions, 36–37
constants, 24
explained, 23–24
numbers, 25–26
objects, 30–32
strings, 27–30
join function, 35
JSON (JavaScript Object Notation), 30–31
JSON servers. See also web applications;
web servers
  API design, 144–145
    modifying, 181, 198
creating, 65–66
layers in express, 139–140
length property, 27
line-by-line prompting, 269–271
Linux
  configuration files, 232–233
deploying on, 218–219
  installing memcached, 225–226
  installing Mocha, 283
  installing Node.js, 14–15
  passing parameters, 262
  running scripts, 259–260
listeners, 115
listing
  albums, 179, 184–185
directory contents, 265
listings
  adding photos using API, 187–188
admin_add_album.html, 189–190
admin_add_album.js, 188–189
admin_add_photo.js, 190–191
album-listing server (load_albums.js), 68
all-node node runner (node_runner.js), 275
another Mustache template page
  (album.html), 132–133
building pages (pages.js), 147
db.js, 198–199
express/https module SSL support
  (https_express_server.js), 230
file loading with full error handling, 56
getting all photos in album, 185–186
handling multiple request types, 73–76
helper functions (helpers.js), 146–147
home page template file (home.html), 131
http-proxy SSL support
(https_proxy_server.js), 231
JavaScript page loader (home.js), 128
login page Mustache template (login.html), 211
raw mode on stdin (raw_mode.js), 267–268
registration page Mustache template
(register.html), 209–210
round-robin proxy load balancer
(roundrobin.js), 223
rpn.js file, 279–280
simple app page bootstrapper
(basic.html), 125
simple postfix calculator using readline
(readline.js), 269–271
static middleware usage (server.js), 152
survey program (questions.js), 272
testing the compute_intersection
function, 61
trivial HTTP server, 222
using process#nextTick to be polite, 60
virtual hosts in express
(vhost_server.js), 228
load balancers, 221
loading modules, cycles in, 94–95
log files, writing to, 216–217
logging in, 161
to Azure, 245–246
to Heroku, 236
login forms, creating, 160–161, 208–211
loops, 42
asynchronous looping, 69–71, 110
low-level operations, writing in sample
application, 175–181

M
Macintosh
configuration files, 232–233
deploying on, 218–219
installing memcached, 225–226
installing Mocha, 283
installing Node.js, 12–14
passing parameters, 262
running scripts, 259–260
managing asynchronous functions
with async module, 104–110
problems with, 103–104
with promises pattern, 111–113
memcached, 224–225
connecting to, 226–227
installing
on Linux/Mac, 225–226
on Windows, 225
messages, flash, 162–163
Microsoft Azure. See Azure
middleware, 139, 148
compressing output, 156–157
configurations, 149
ordering, 150–151
POST data, cookies, sessions,
153–155
PUT and DELETE support, 156
static file handling, 151–152
usage, 148–149
mkdir function, 264
mkdirSync function, 264–265
Mocha, 277
installing, 283
testing API design, 283–286
modifying API design. See also updating
for authentication, 198
for database usage, 181
modules, 51, 89
assert, 280
async, 104–110
bcrypt, 200
bluebird, 111–113
caching, 94
child_process
ext function, 273–274
spawn function, 274–276
documentation, 96
including, 93
installing via NPM, 92–93
loading, cycles in, 94–95
node-uuid, 200
passport, 157–160
authentication with MySQL, 202–203
creating login forms, 160–161
flash messages, 162–163
implementing authentication, 203–204
logging in, 161
restricting page access, 162
private package management, 101–102
publishing, 102
readline, 268–273
returning objects from
constructor model, 91
factory model, 91
in sample application, 146–148
searching for, 93
versioning, 94, 97
writing, 89–90, 95–100
MongoDB
collections
creating, 169–170
deleting documents, 172
inserting documents, 170–171
querying, 172–175
updating documents, 171–172
connecting to, 169
in sample application, 176–177
data structure, 167
data types, 168
databases, creating, 169, 176–177
installing, 165–166
Node.js usage, 166–167
writing low-level operations in sample application, 175–181
multiplatform development, 232
locations and configuration files, 232–233
paths, 233
multiple file types, serving in streams, 120–121
multiple request types in web applications, 71–79
multiprocessor deployment, 220–223
sessions and, 223–227
Mustache, 124, 129–131
MySQL
authentication, 198
creating login and registration pages, 208–211
creating user handlers, 205–207
creating users, 199–201
fetching users, 201–202
implementing, 203–204
modifying API, 198
routing in, 207–208
updating express application, 202–203
configuring ClearDB add-on
in Azure applications, 248–252
in Heroku applications, 239–241
connecting to, 195–196, 198–199
creating schema, 194–195
installing, 193–194
Node.js usage, 194
querying, 196–197
N
Nan value, 26
nextTick function, 59–61
Node shell, 15–16
Node.js
debugging, 18–21
history of, 2–4
HTTPS/SSL support, 230–231
installing
  on Linux, 14–15
  on Mac, 12–14
  on Windows, 9–12
limitations of, 3
resources for information, 21–22
running
  from JavaScript files, 16
  with Node shell, 15–16
updating, 21–22
nodeunit, 277
  installing, 278
  writing tests, 278
    asynchronous tests, 282
    functional tests, 279–281
    groups of tests, 281–282
node-uuid module, 200
nonblocking IO, changing blocking IO to,
  51–53. See also asynchronous functions
NoSQL. See MongoDB
NPM (Node Package Manager)
  installing modules, 92–93
    private package management, 101–102
npm help command, 92
npm install command, 92
npm link command, 101
npm ls command, 93
npm publish command, 102
npm search command, 92
npm unpublish command, 102
npm update command, 93
null, 24
numbers, 25–26
O
object literal syntax, 30
objects, 30–32
  cloning, 179
    returning from modules
      constructor model, 91
      factory model, 91
openSync function, 263
operators, 41–42
ordering middleware, 150–151
output, compressing, 156–157. See also
I/O functions
P
packages
  NPM. See NPM (Node Package Manager)
    updating, 93
page access, restricting, 162
pages
  adding to applications, 188–191
    creating login and registration pages,
      208–211
paging functionality, 79–82
parallel code execution, 107–108
parallel function, 107–108
parameters, scripts and, 261–262
parseFloat function, 26
parseInt function, 26
passing parameters, 261–262
passport module, 157–160
authentication with MySQL, 202–203
creating login forms, 160–161
flash messages, 162–163
implementing authentication, 203–204
logging in, 161
restricting page access, 162
paths in multiplatform development, 233
patterns, 54
asynchronous looping, 69–71
callback functions, error handling and, 54–56
promises, 111–113
photos
adding to albums, 180–181, 186–188, 190–191
finding, 180, 185–186
PKG installer, installing Node.js on Mac, 12–14
placeholders, 141, 197
pop function, 34
port numbers, 216
POST data
converting to PUT and DELETE, 156
with express middleware, 153–155
receiving
via forms, 86–87
via streams, 83–86
sending, 82–83
postfix calculations, 269
precise equality operator, 36
private package management, 101–102
process object, 47
processes
blocking IO and, 49–50
creating
with exec function, 273–274
with spawn function, 274–276
process.exit function, 263–264
process.nextTick function, 59–61
promises pattern, 111–113
prompting line-by-line, 269–271
prototypes, 43–45, 98
proxies
HTTPS/SSL support, 231–232
for multiprocessor deployment, 220–223
publishing modules, 102
push function, 34
PUT method, 156
query strings, 79–82
querying. See also searching
collections, 172–175
MySQL databases, 196–197
questions/answers with readline module, 271–273
quotation marks in strings, 27
raw mode, 267
readable event, 115
readdirSync function, 265
reading files with streams, 116
readline module, 268–273
Readme.md, 96
readSync function, 263
receiving POST data
via forms, 86–87
via streams, 83–86
registration pages, creating, 208–211
regular expressions, 29–30
REPL (Read-Eval-Print-Loop), 15–16
replace function, 29

request types, handling multiple, 71–79

require function, 51, 93

response codes (HTTP), 79

REST (Representational State Transfer), API design, 144–145

restricting page access, 162

returning
  data in web applications, 67–69
  objects from modules
    constructor model, 91
    factory model, 91

reverse Polish notation, 269

routing
  for authentication, 207–208
  in express, 140–141
    updating sample application, 141–144

running
  Node.js
    from JavaScript files, 16
    with Node shell, 15–16
  scripts
    on Linux/Mac, 259–260
    on Windows, 260–261

S

schemas, creating, 194–195

scope of functions, 40
  this keyword, 56–59

screen utility, 217

scripts
  I/O functions
    buffered I/O, 266–267
    readline module, 268–273
    stdin, stdout, stderr, 266
    unbuffered input, 267–268

  parameters and, 261–262

  processes
    creating with exec function, 273–274
    creating with spawn function, 274–276

  running
    on Linux/Mac, 259–260
    on Windows, 260–261

  synchronous functions, 262
    creating directories, 264–265
    file operations, 263–264
    listing directory contents, 265

search function, 30

searching. See also querying
  for autogenerated _id fields, 174
  for modules, 93

securing web applications, 229
  built-in support for HTTPS/SSL, 230–231
  generating test certificates, 229–230
  proxy server support for HTTPS/SSL, 231–232

selecting testing frameworks, 277–278

semantic versioning, 4

sending POST data, 82–83

serial code execution, 104–107

series function, 106–107

ServerRequest object, 78

servers
  JSON servers
    API design, 144–145
    creating, 65–66
    modifying API, 181, 198

  web application deployment. See web applications, deploying

web servers
  creating, 16–18
  creating in express, 138–139
HTML skeleton pages, 124–125
JavaScript bootstrapper, 128–129
modifying URL scheme, 126–128
sessions
with express middleware, 153–155
on multiple servers, 223–227
setRawMode function, 267
setTimeout function, 51
shebang, 260
shell. See Node shell; REPL (Read-Eval-Print-Loop)
shift function, 34
sort function, 35
source code, downloading, 5
spawn function, 274–276
splice function, 28, 34
split function, 29
SQL. See MySQL
SSL, 229
built-in support for, 230–231
generating test certificates, 229–230
proxy server support, 231–232
stability levels, 4
static content
serving with express, 151–152
serving with streams, 115
buffers, 117–120
modifying URL scheme, 126–128
multiple file types, 120–121
reading files, 116
in sample application, 125–126
stderr, 266. See also error handling
stdin, 266. See also input
stdout, 266
streams
moving data between, 121
receiving POST data, 83–86
serving static content, 115
with buffers, 117–120
modifying URL scheme, 126–128
multiple file types, 120–121
reading files, 116
in sample application, 125–126
strings, 27–30
buffers versus, 117
functions, 28–29
regular expressions, 29–30
substr function, 28
synchronous functions, 61, 262
creating directories, 264–265
file operations, 263–264
listing directory contents, 265
synchronous programming, changing to
asynchronous programming, 51–53
T
TDD (test-driven development), 277
tee utility, 216–217
templates, client-side, 122–124
adding pages, 188–191
with Mustache, 129–131
in sample application, 131–134
ternary operator, 41
test certificates, generating, 229–230
test-driven development (TDD), 277
testing
API design, 283–286
frameworks
installing Mocha, 283
installing nodeunit, 278
selecting, 277–278
Heroku applications, 239
virtual hosts, 228–229
writing tests, 278
asynchronous tests, 282
functional tests, 279–281
groups of tests, 281–282
this keyword, 56–59
threads, blocking IO and, 49–50
time function, 47
trim function, 29
try/catch blocks, 46
asynchronous functions and, 53–54
typeof, 24, 32
types
arrays, 32–35
booleans, 26
comparisons and conversions, 36–37
constants, 24
explained, 23–24
numbers, 25–26
objects, 30–32
strings, 27–30

unbuffered input, 267–268
undefined, 24, 33
UNIX. See Linux; Macintosh
unlinkSync function, 264
unshift function, 34
updating. See also modifying API design
documents in collections, 171–172
express for authentication, 202–203
handlers in sample application, 182–188
Node.js, 21–22
packages, 93
uploading files, 154–155
URL contents
downloading on Windows, 17
modifying URL scheme, 126–128
url.parse function, 80
user handlers, creating, 205–207
users
authenticating, 206–207
creating, 199–201, 205–206
fetching, 201–202

variable scope. See scope of functions
verifying Windows installation of Node.js, 10–12
versioning of modules, 94, 97
virtual hosting, 227–229

warn function, 47
waterfall function, 104–106
web applications
asynchronous looping, 69–71
authentication, 157–160
creating login forms, 160–161
flash messages, 162–163
logging in, 161
restricting page access, 162
client-side templates, 122–124
adding pages, 188–191
with Mustache, 129–131
in sample application, 131–134
data structure in sample application, 192
deploying, 215
to Azure, 244–256
basic deployment, 216–217
to Heroku, 235–244
on Linux/Mac, 218–219
multiprocessor deployment, 220–223
problems with basic deployment, 218
sessions on multiple servers, 223–227
virtual hosting, 227–229
on Windows, 219–220
error handling, 163–164
error handling in, 66
limitations of, 1–2
multiplatform development, 232
locations and configuration files, 232–233
paths, 233
multiple request types, 71–79
paging functionality, 79–82
POST data
receiving via forms, 86–87
receiving via streams, 83–86
sending, 82–83
returning data, 67–69
securing with HTTPS/SSL, 229
built-in support for, 230–231
generating test certificates, 229–230
proxy server support, 231–232
updating routing, 141–144
web servers. See also JSON servers
creating, 16–18
in express, 138–139
HTML skeleton pages, 124–125
JavaScript bootstrapper, 128–129
modifying URL scheme, 126–128
static content. See static content
wget, 17
Windows
configuration files, 232–233
deploying on, 219–220
downloading URL contents, 17
installing memcached, 225
installing Mocha, 283
installing Node.js, 9–12
passing parameters, 262
running scripts, 260–261
writeSync function, 263
writing
to log files, 216–217
low-level operations in sample application, 175–181
modules, 89–90, 95–100
tests, 278
asynchronous tests, 282
functional tests, 279–281
groups of tests, 281–282
writing to log files, 216–217
low-level operations in sample application, 175–181
modules, 89–90, 95–100
tests, 278
asynchronous tests, 282
functional tests, 279–281
groups of tests, 281–282