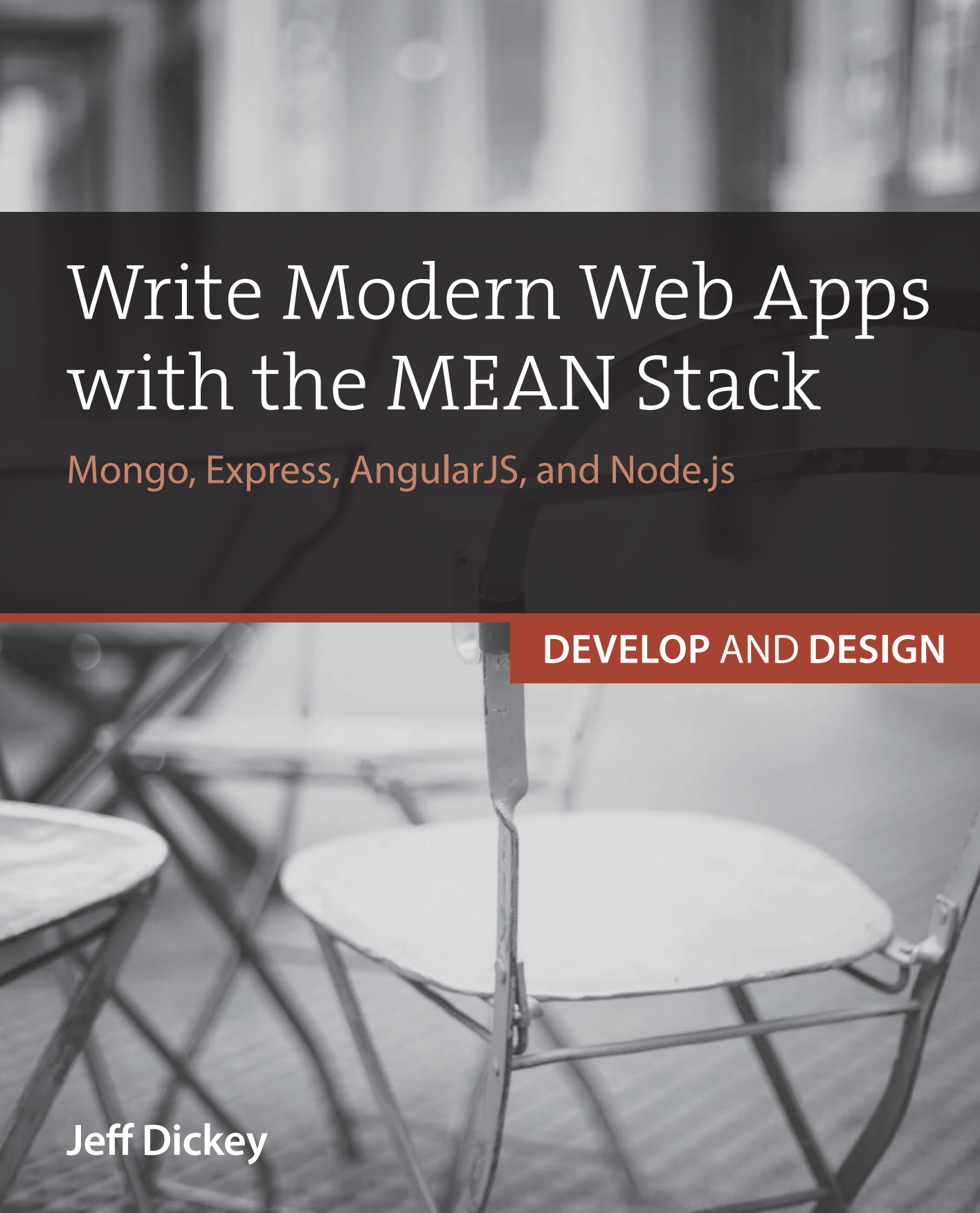


Write Modern Web Apps with the MEAN Stack

Mongo, Express, AngularJS, and Node.js

DEVELOP AND **DESIGN**

Jeff Dickey

Write Modern Web Apps with the MEAN Stack

Mongo, Express, AngularJS, and Node.js

DEVELOP AND DESIGN

Jeff Dickey

PEACHPIT PRESS
WWW.PEACHPIT.COM

Write Modern Web Apps with the MEAN Stack: Mongo, Express, AngularJS, and Node.js
Jeff Dickey

Peachpit Press
www.peachpit.com

To report errors, please send a note to errata@peachpit.com
Peachpit Press is a division of Pearson Education.

Editor: Kim Wimpsett
Production editor: David Van Ness
Proofreader: Liz Welch
Technical editor: Joe Chellman
Compositor: Danielle Foster
Indexer: Danielle Foster
Cover design: Aren Straiger
Interior design: Mimi Heft

ISBN-13: 978-0-13-393015-3
ISBN-10: 0-13-393015-7

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To Mom and Dad,
for sometimes allowing me to sit inside all day on that computer

ABOUT THE AUTHOR

Jeff Dickey is a full-stack web developer with years of startup experience in San Francisco and Los Angeles. Jeff has launched projects, maintained large systems, and led development teams. With more than 10 years of experience on all of the major web platforms, he is continually searching for the latest technology for building applications. Currently Jeff works for Heroku as its first CLI developer. Jeff is also an instructor for General Assembly, teaching a course on back-end web development.

CONTENTS

Preface ... ix
Introduction ... x

CHAPTER 1 HOW MODERN WEB ARCHITECTURE IS CHANGING ... 2
The Rise of the Static App ... 4
Enter the Thick Client ... 6

CHAPTER 2 WHY JAVASCRIPT IS A GOOD CHOICE FOR MODERN APPS ... 8
What Is Angular.js? ... 10
What Is Node.js? ... 13
What Is Express? ... 20
What Is MongoDB? ... 22

CHAPTER 3 INTRODUCING THE SOCIAL NETWORKING PROJECT ... 28
Creating a Static Mockup of the Recent Posts Page ... 30
Angularizing the Page ... 31
Adding New Posts ... 34
Next Steps ... 38

CHAPTER 4 BUILDING A NODE.JS API ... 40
The Stock Endpoint ... 42
Creating Posts via the API ... 44
MongoDB Models with Mongoose ... 45
Using Mongoose Models with the POST Endpoint ... 46
Next Steps ... 49

CHAPTER 5 INTEGRATING NODE WITH ANGULAR ... 50
$http ... 52
Reading Posts from the API with $http ... 53
Serving posts.html Through Node ... 55
Saving Posts to the API with $http ... 56
Fixing the Post Ordering ... 57
Cleaning Up server.js ... 58
Cleaning Up Angular ... 63
Next Steps ... 67

CHAPTER 6 AUTOMATING YOUR BUILD WITH GULP ... 68
Introducing Grunt and Gulp ... 70
Gulp Hello World ... 71
Building JavaScript with Gulp ... 72
Building CSS with Gulp ... 80
Gulp Dev Task ... 82
Other Gulp Plug-ins ... 84
Next Steps ... 85

CHAPTER 7 BUILDING AUTHENTICATION IN NODE.JS ... 86
Introducing Token Authentication ... 88
JSON Web Token (JWT) ... 89
Using BCrypt ... 94
Authentication with MongoDB ... 97
Next Steps ... 101

CHAPTER 8 ADDING ROUTING AND CLIENT AUTHENTICATION ... 102
Routing ... 104
Creating a Login Form ... 107
Express Authentication ... 110
Angular Events ... 114
Authenticating Social Posts ... 116
HTML5 Pushstate ... 118
Registration ... 119
Logout ... 120
Remember Me ... 121
User Foreign Key ... 122
Next Steps ... 123

CHAPTER 9 PUSHING NOTIFICATIONS WITH WEBSOCKETS ... 124
Introducing WebSockets ... 126
How WebSockets Work ... 127
What Should You Use WebSockets For? ... 128
WebSockets in Your Social App ... 129
WebSockets in Angular.js ... 133
WebSocket Architecture ... 135
Dynamic WebSocket Hostname ... 140
Next Steps ... 141

CHAPTER 10 PERFORMING END-TO-END TESTING ... 142
Setting Up Protractor ... 144
JavaScript Testing Frameworks ... 145
Writing a Basic Protractor Test ... 146
Protractor Expectations ... 156
chai-as-promised ... 159
When to Use End-to-End Tests ... 160
Next Steps ... 161
CHAPTER 11 TESTING THE NODE SERVER ... 162
Not Quite Unit Testing ... 164
Mocha for Node ... 165
Post Controller ... 167
SuperTest ... 168
Base Router ... 169
Using the Base Router with SuperTest ... 170
Models in Controller Tests ... 171
Testing Controllers with Authentication ... 173
Code Coverage ... 175
The npm test Command ... 177
JSHint ... 178
Next Steps ... 179
CHAPTER 12 TESTING ANGULAR.JS ... 180
Using Karma ... 182
Using Bower ... 183
Setting Up Karma ... 185
Basic Karma Service Test ... 187
HTTP Testing with Karma ... 189
Karma Controller Test ... 192
Testing Spies ... 197
Next Steps ... 199

CHAPTER 13 DEPLOYING TO HEROKU 200
Platform-as-a-Service 202
How Heroku Works 203
Twelve-Factor Apps 204
Deploying an Application to Heroku 205
MongoDB on Heroku 207
Redis on Heroku 208
Compiling Assets 210
Node Cluster 212
Next Steps 213

CHAPTER 14 DEPLOYING TO DIGITAL OCEAN 214
What Is Digital Ocean? 216
Single-Server vs. Multiserver Architecture 217
Fedora 20 218
Creating a Server 219
Installing Node 222
Installing MongoDB 223
Installing Redis 225
Running the Social App 227
Running Social App Under systemd 228
Zero-Downtime Deploys 229
Multiserver Migration 234
Next Steps 236
Conclusion 237

Index 238

PREFACE

WHO IS THIS BOOK FOR?

This book is for web developers wanting to learn how building web applications has changed. The book assumes a basic knowledge of JavaScript. Knowledge of Node or Angular is helpful as well but not required.

WHY I WROTE THIS BOOK

I've been a web developer since 2004 and have professionally worked with most of the major web platforms. I love to seek out new technology that helps me write my applications better.

Applications built with an MVC framework such as Angular has been the largest paradigm shift that I've seen in the web community. Frameworks and tools have come and gone, but client-side MVC applications are fundamentally different.

I've been impressed with the quality of applications that I've shipped with Angular and Node. The tools are simple—sometimes a bit naïve—but this simplicity comes with the fantastic ability to iterate on features and maintain a codebase.

Applications such as those built with the MEAN stack are becoming more popular, but many development teams still feel comfortable with server-generated pages and relational databases.

I've had such good luck with MEAN applications that I want to share my knowledge of how to build them with you.

I hope you'll enjoy exploring this new method of building applications with me. I love discussing these topics, so feel free to reach out to me on Twitter to continue the conversation.

Jeff Dickey
@dickeyxxx
August 2014

INTRODUCTION

The JavaScript community has a strong belief in the power of composability when architecting software. This is in line with the Unix philosophy of simple components that can be used together to quickly build applications.

By no means is this methodology the only one that exists. Ruby on Rails, for example, uses an opinionated framework to make decisions for you about what your application should look like. Opinionated frameworks offer the advantage of being able to quickly learn an application because out of the box it works—you just need to fill in the gaps. Opinionated frameworks are also easier to learn because there is usually a "right" way to do something. The downside is that you're limited to building applications that the framework was made for, and moving outside of the use cases the framework was made for can be difficult.

By contrast, the composition methodology is more flexible. Composing simple pieces together has a clear advantage of allowing you to build anything you want. These frameworks provide you with building blocks, and it's up to you to decide how to put them together. The downside is mostly in the learning phase. It's difficult to know what is a good idea and what is a bad idea without having experience doing both.

For this reason, it's useful to read up on opinionated guides for how to build JavaScript applications. These opinions provide one person's viewpoint on good and bad decisions and give you a road map to what an application should look like.

This book shows you how to build your own MEAN application following my opinions of what a good application should look like. These opinions come from my experience developing applications. While I have a good amount of experience, it's unlikely it will fit perfectly with any other one person. For this reason, I find books such as this are useful to learn not just the "how" of using a tool set but the "why" as well.

In other words, it's useful to know how to use promises in Node but not very useful if you don't understand why they're useful.

The application you will build is called simply Social App (see **Figure I.1**). You can see an example of it running at *https://mean-sample.herokuapp.com* as well as the code at *https://github.com/dickeyxxx/mean-sample*.

The application is similar to Twitter. Users can create an account and add posts. The feature count is not large but does consist of some neat solutions such as WebSockets that immediately display new posts to all users viewing the application. I'll also go over compiling the CSS and JavaScript assets with Gulp, deploying the application to both Heroku and Digital Ocean, building a clean, maintainable API and more.

Having a "newsfeed" that displays live, updating content is a pattern that I see on just about every project I work on. I chose this as an example because it is complicated enough to incorporate many different tools but not so complex that you will become bogged down in the minutiae of this specific application.

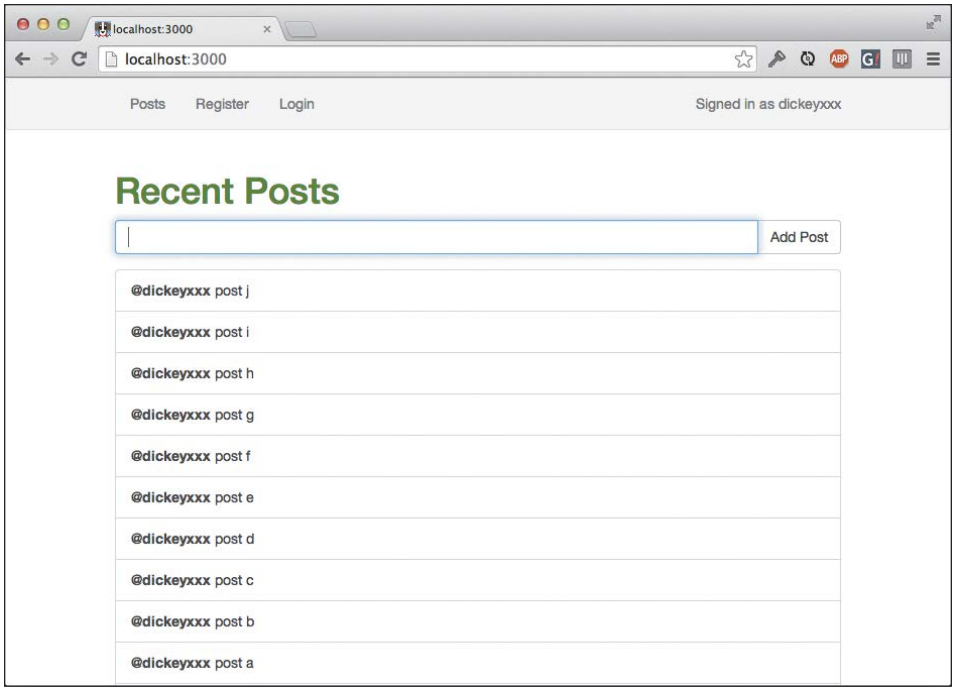


FIGURE I.1 Social app

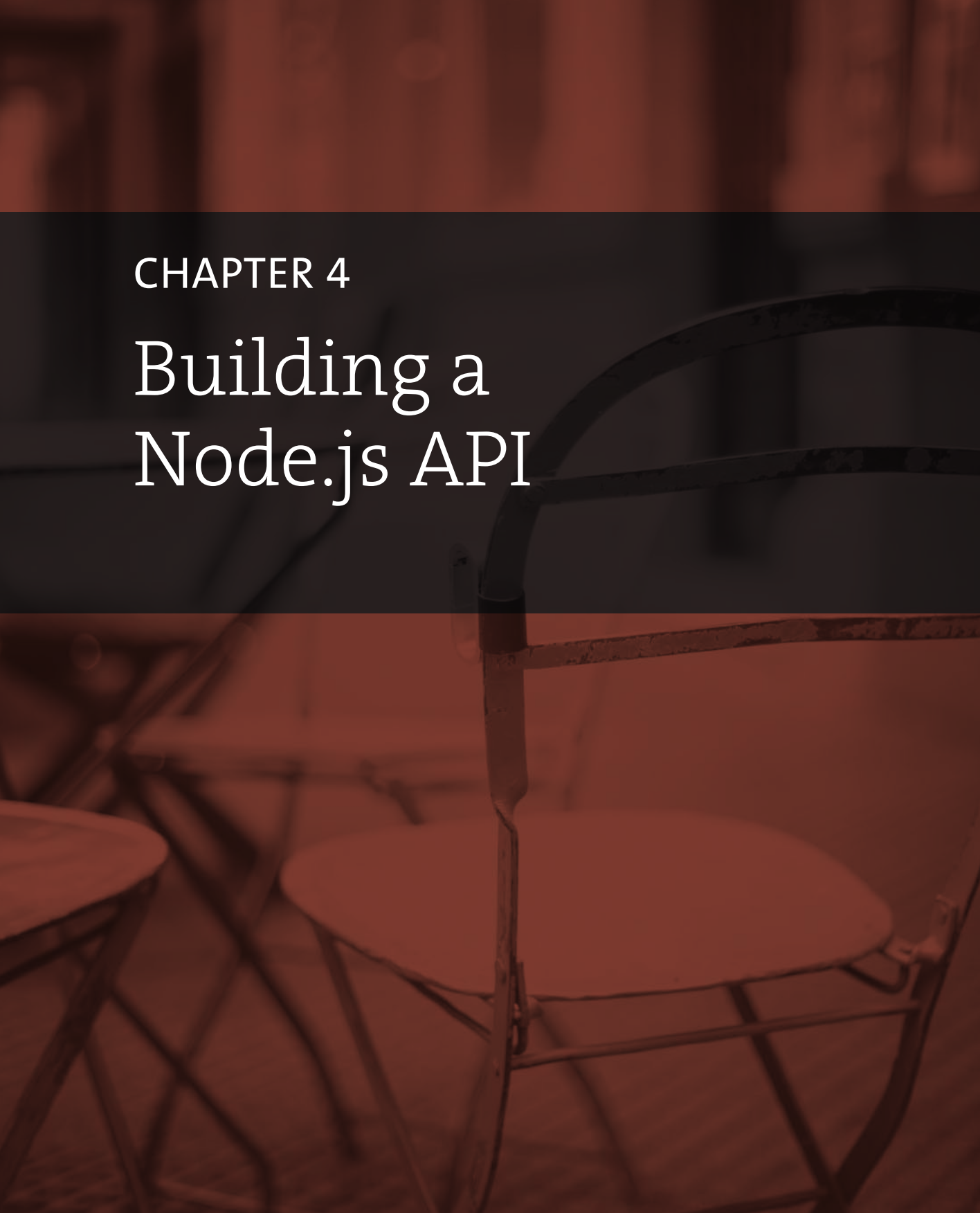
This application is also easily extensible. I encourage you while reading this book to take the time to not only implement the application as I have done but to build features of your own. It's relatively easy to follow along and build the same application, but you know that's not how software is actually written.

Learning a new skill is tough. As a teacher, I've witnessed many people learning something for the first time and been able to witness their progress. One facet of that I've noticed is that learning doesn't feel like anything. You can't tell whether you're learning something when you're learning it—in fact, learning feels a lot more like frustration.

What I've learned is that during this period of frustration is actually when people improve the most, and their improvements are usually obvious to an outsider. If you feel frustrated while trying to understand these new concepts, try to remember that it might not feel like it, but you're probably rapidly expanding your knowledge.

With that, join me in Chapter 1 while you learn a bit about the history of the Web's surprising relationship with JavaScript, how it's changed the way we think of applications, and where the MEAN stack fits in.

This page intentionally left blank



CHAPTER 4

Building a Node.js API

In the previous chapter, you built a fully functioning Angular app for posting status updates. In this chapter, you will build an API for it to get a list of all the posts and to make a new post. The endpoints will be as follows:

- `GET /api/posts` returns a JSON array of all the posts. This is similar to what you had in `$scope.posts`.
- `POST /api/posts` expects a JSON document containing a username and post body. It will save that post in MongoDB.

THE STOCK ENDPOINT

To start, you'll use Node.js and Express to build a stock `/api/posts` endpoint. First, inside a new folder, create a `package.json` file like the following:

```
{
  "name": "socialapp"
}
```

The name can be anything you want, but try to ensure it doesn't conflict with an existing package. The `package.json` file is the only thing you need to make a directory a node project.

Now that you have this, you can add `express` and `body-parser` as dependencies:

```
$ cd <path-to-project-folder>
$ npm install --save express
$ npm install --save body-parser
```

`body-parser` is used for `express` to read in JSON on POST requests automatically.

The `--save` flag saves the dependency to the `package.json` file so you know what versions of what packages the app was built with (and therefore depends on). In fact, if you open your `package.json`, you'll see something similar to this:

```
{
  "name": "socialapp",
  "dependencies": {
    "body-parser": "^1.4.3",
    "express": "^4.4.4"
  }
}
```

Now that you've done this, you can `require('express')` to include it in a node script.

Create a `server.js` file with the following contents:

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

var app = express()
app.use(bodyParser.json())

app.get('/api/posts', function (req, res) {
  res.json([
    {
      username: 'dickeyxxx',
      body: 'node rocks!'
    }
  ])
})

app.listen(3000, function () {
  console.log('Server listening on', 3000)
})
```

Try running this file:

```
$ node server.js
```

You can access it in your browser at *http://localhost:3000/api/posts*. You should see that your stubbed JSON comes back.

You now have the basic Node request in place, so you need to add the POST endpoint for adding posts and back it against MongoDB.

CREATING POSTS VIA THE API

Now let's build the POST endpoint for creating posts. Add this endpoint to `server.js`:

```
app.post('/api/posts', function (req, res) {
  console.log('post received!')
  console.log(req.body.username)
  console.log(req.body.body)
  res.send(201)
})
```

This is just a request that checks to see whether you're reading the data properly. The client would receive only the HTTP status code 201 (created). It's good to build a lot of these stubbed-out sorts of logic to check to see whether your plumbing is in order when building MEAN applications. Because you can't test a POST request using the browser, you should check to see whether it is working using `curl`:

```
curl -v -H "Content-Type: application/json" -XPOST --data
→ "{\"username\":\"dickeyxxx\", \"body\":\"node rules!\"}"
→ localhost:3000/api/posts
```

If you are unfamiliar with `curl`, this says "Make a POST request to *localhost:3000/api/posts*. Be verbose." Setting your `Content-Type` header to `json` includes the JSON document as the body.

The `Content-Type` header is necessary to be able to parse this content into the friendly `req.body.username` objects from the JSON.

If the command line isn't your thing, you can do this same thing using the great Postman app for Chrome to test APIs. Regardless of what method you use, it is crucial you test your APIs using stub clients like this rather than building your app in lockstep.

MongoDB MODELS WITH MONGOOSE

To interact with MongoDB, you will be using the Mongoose ODM. It's a light layer on top of the Mongo driver. To add the `npm` package, do this:

```
$ npm install --save mongoose
```

It'll be good to keep this code modularized so your `server.js` file doesn't get huge. Let's add a `db.js` file with some of the base database connection logic:

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/social', function () {
  console.log('mongodb connected')
})
module.exports = mongoose
```

You can get access to this `mongoose` instance by using the `require` function. Now let's create a `mongoose` model to store the posts. Place this code in `models/post.js`:

```
var db = require('../db')
var Post = db.model('Post', {
  username: { type: String, required: true },
  body:     { type: String, required: true },
  date:     { type: Date, required: true, default: Date.now }
})
module.exports = Post
```

Now you have a model you can get with `require`. You can use it to interact with the database.

USING MONGOOSE MODELS WITH THE POST ENDPOINT

Now requiring this module will give you the `Post` model, which you can use inside of your endpoint to create posts.

In server.js, change your `app.post('/api/posts')` endpoint to the following:

```
var Post = require('./models/post')
app.post('/api/posts', function (req, res, next) {
  var post = new Post({
    username: req.body.username,
    body: req.body.body
  })
  post.save(function (err, post) {
    if (err) { return next(err) }
    res.json(201, post)
  })
})
```

First, you require the `Post` model. When a request comes in, you build up a new instance of the `Post` model with `new Post()`. Then, you save that `Post` model and return a JSON representation of the model to the user with status code 201.

While it isn't totally necessary to return the JSON here, I like for my create API actions to do so. The client can sometimes make use of it. It might be able to use the `_id` field or show data that the server generated (such as the date field, for example).

Note the `err` line. In Node, it's common for code to return callbacks like this that start with an error argument, and then the data is returned. It's your responsibility to check whether there is an error message and do something about it. In this case, you call the `next()` callback with an argument, which triggers a 500 in Express. An error in this case would typically mean the database was having issues. Other programming languages use exceptions to handle errors like this, but Node.js made the design decision to go with error objects because of its asynchronous nature. It's simply not possible to bubble up an exception with evented code like Node.js.

Go ahead and hit this endpoint again with `curl` or Postman. (Make sure you first restart your server. Later you'll see how to automatically restart it with `nodemon`.)

```
$ curl -v -H "Content-Type: application/json" -XPOST --data
→ "{\"username\":\"dickeyxxx\", \"body\":\"node rules!\"}"
→ localhost:3000/api/posts
```

You should see a response like the following (make sure you've started your Mongo server):

```
> POST /api/posts HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:3000
> Accept: */*
> Content-Type: application/json
> Content-Length: 46
>
* upload completely sent off: 46 out of 46 bytes
< HTTP/1.1 201 Created
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 120
< Date: Sun, 22 Jun 2014 00:41:55 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact
{"__v":0,"username":"dickeyxxx","body":"node rules!","_id":
→ "53a62653fa305e5ddb318c1b","date":"2014-06-22T00:41:55.040Z"}
```

Since you see an `_id` field coming back, I'm pretty sure it's working. Just to be sure, though, let's check the database directly with the `mongo` command:

```
$ mongo social
MongoDB shell version: 2.6.1
connecting to social
> db.posts.find()
{ "_id" : ObjectId("53a62653fa305e5ddb318c1b"), "username" : "dickeyxxx",
→ "body" : "node rules!", "date" : ISODate("2014-06-22T00:41:55.040Z"),
→ "__v" : 0 }
```

Looks like it made it into the database!

Now, let's update the GET request to read from the database:

```
app.get('/api/posts', function (req, res, next) {
  Post.find(function(err, posts) {
    if (err) { return next(err) }
    res.json(posts)
  })
})
```

This one is similar to the last one. Call `find` on the `Post` model; then, when the request returns, render out the posts as JSON (so long as no error was returned). Go back to your web browser and reload *http://localhost:3000/api/posts* to see it in action.

You now have a full API you can read and write from in order to support your Angular app.

Here is the final `server.js`:

```
var express = require('express')
var bodyParser = require('body-parser')
var Post = require('./models/post')

var app = express()
app.use(bodyParser.json())

app.get('/api/posts', function (req, res, next) {
  Post.find(function(err, posts) {
    if (err) { return next(err) }
    res.json(posts)
  })
})

app.post('/api/posts', function (req, res, next) {
  var post = new Post({
    username: req.body.username,
    body: req.body.body
  })
  post.save(function (err, post) {
    if (err) { return next(err) }
    res.json(201, post)
  })
})

app.listen(3000, function () {
  console.log('Server listening on', 3000)
})
```

NEXT STEPS

You've now built the full API for you to use with the Angular app. In the next chapter, you'll integrate the API into Angular and serve the Angular app via Node. You'll also take some time to clean up your code a little by breaking it into modules.

INDEX

NUMBER

12-factor apps. *See* Heroku 12-factor apps

A

Ajax-empowered JavaScript, 4
AMQP (RabbitMQ) message broker, 137
Angular applications
  cookie-based authentication, 88
  token-based authentication, 88
Angular code modules, 31
Angular.js. *See also* Node integration with Angular
  benefits, 12
  breaking into services, 64–66
  creating logical sections, 64–66
  events, 114–115
  vs. jQuery, 10–12
  JSON, 12
  overview, 10
  recent posts page, 31–33
  serving static assets, 63–64
  "Unknown provider" error, 75
  use with MEAN stack, 12
  WebSockets in, 133–134
Angular.js testing. *See also* testing frameworks
  Bower, 183–184
  HTTP with Karma, 189–191
  Karma, 182
  Karma controller, 192–196
  Karma service test, 187–188
  setting up Karma, 185–186
  spies, 197–198
application process, traditional, 4
asynchronous code, writing with promises, 52
authenticating social posts, 116–117
authentication in Express, 110–113
authentication in Node.js. *See also* Node.js
  BCrypt, 94–96
  JWT (JSON Web Token), 89–93
  with MongoDB, 97–100
  tokens, 88
automating builds. *See* Gulp

B

back end, 6
base router
  accessing for Node-server testing, 169
  using with SuperTest, 170
BCrypt hashing algorithm, 94–96
`boot.js` script, 229–231
Bootstrap styling, using with login form, 107
Bower, using with Angular, 183–184
BSON data storage, using with MongoDB, 23–24

C

callbacks, 16–17

Chai assertion, performing in Protractor, 157

client authentication. *See* authentication

client-server communication. *See* WebSockets

cloud server. *See* Digital Ocean cloud server

CommonJS spec, 17

"concurrency," handling, 15–16

connecting to `ws` websocket, 129–130

controller tests, models in, 171–173

controllers

- declaring with Angular.js, 31–32
- testing, 192–196

controllers with authentication, testing, 173–174

cookie-based authentication, 88

CSS, building with Gulp, 80–81

D

data, loading with WebSockets, 128

databases. *See* MongoDB document database

Digital Ocean cloud server

- architecture, 217
- centralized databases, 235
- creating account with, 219
- creating droplets, 219–221
- Fedora 20, 218, 220
- installing Redis, 225–226
- load balancer, 234–235
- MongoDB, 223–224
- multiserver migration, 234–235
- overview, 216
- private networking, 235
- running social app, 227
- social app under `systemd`, 228
- SSH key, 219, 221
- zero-downtime deploys, 229–233

documents

- inserting in MongoDB, 26
- querying in MongoDB, 26

droplets, creating on Digital Ocean, 219–221

E

`e2e` directory, using in Protractor test, 146

end-to-end testing

- JavaScript testing frameworks, 145
- Protractor setup, 144
- using, 160

enterprise

- Node.js in, 14
- PayPal, 14
- vs. startup, 13
- Yammer, 14

evented architecture, 15–16

events

- Angular.js, 114–115
- broadcasting in WebSockets, 131

events from clients, publishing, 139

expectations in Protractor, 156–158

Express NPM package

- authentication, 110–113
- databases for Node.js, 20

Express NPM package (*continued*)
installing, 20
JWT with, 90–91
stock endpoint, 42–43
external modules, declaring, 17

F

Fedora 20, 218, 220
foreign keys for users, 122
function calls, testing for, 197–198

G

Git, integrating for Heroku, 205
Grunt and Gulp, 70
Gulp
`angular.module()` method, 72
building CSS, 80–81
building JavaScript, 72–79
getters and setters, 72
and Grunt, 70
"Hello World," 71
rebuilding upon file changes, 77
source maps, 78–79
Uglifier, 74–77
gulp-autoprefixer plug-in, 84
`gulp-concat` plugin, installing, 72
gulp-imagemin plug-in, 84
gulp-jshint plug-in, 84
gulp-livereload plug-in, 84
`gulp.-nodemon`, Uglifier minification tool, 82–83
gulp-rev plug-in, 84
gulp-rimraf plug-in, 84
`gulp.watch`, 77

H

hashing algorithms, using with passwords, 94
"Hello World"
Gulp, 71
Node.js, 19
Heroku 12-factor apps
admin processes, 204
backing services, 204
build, release, run, 204
codebase, 204
concurrency, 204
config, 204
dependencies, 204
dev/prod parity, 204
disposability, 204
logs, 204
port binding, 204
processes, 204
Heroku PaaS
compiling assets, 210–211
deploying applications to, 205–206
error page, 206
function of, 203
hosting UNIX processes, 203
integrating Git for, 205
MongoDB on, 207
Node Cluster, 212
overview, 202
Redis on, 208–209

$http
reading posts from API with, 53–54
saving posts to API, 56
HTML5 pushstate, enabling, 118, 121
HTTP calls, performing in Angular, 52
HTTP module, using with Node.js, 19
HTTP proxy, node-http-proxy, 234–235
HTTP testing with Karma, 189–191

I

installing
Express NPM package, 20
gulp-concat plugin, 72
MongoDB document database, 25
MongoDB for Digital Ocean, 223–224
Node.js, 18
Redis for Digital Ocean, 225
Uglifier minification tool, 72
I/O, handling by JavaScript, 16

J

Jasmine testing framework, 145
JavaScript
Ajax-empowered, 4
design of, 16
maintenance, 5
sharing code on Web, 17
testing frameworks, 145
jQuery vs. Angular.js, 10–12
JSHint, using with Node, 178
JSON
recent posts page, 32
use with Angular. js, 12
JWT (JSON Web Token)
creating tokens, 89
with Express, 90–91
password validation, 91–93
server.js, 90

K

Karma controller test, 192–196
Karma test runner
services, 187–188
setting up, 185–186
using with Angular, 182

L

languages, rise and fall, 14
libraries, handling concurrency, 16
logged-in user, showing, 114–115
login form
Bootstrap styling, 107
creating, 107–109
in Protractor, 153–154
updating nav bar, 114
updating URL, 108
.login() function, calling, 108
logout, 120

M

message brokers
AMQP (RabbitMQ), 137
Redis's pubsub, 137–138
ZeroMQ, 137
messages, passing as strings, 127
minification tool, Uglifier, 74–77

mobile APIs, 5
Mocha testing framework
default reporter, 165
described, 145
for Node, 165–166
Nyan Cat reporter, 166
using with Protractor, 147–148
modularity, 6
modules
declaring controller, 31
and NPM, 17–18
mongod daemon, starting, 25
MongoDB document database
authentication, 97–100
auto-sharding, 24
BSON data storage, 23–24
Collection in hierarchy, 25
collections, 22
Database in hierarchy, 25
Document in hierarchy, 25
document-oriented, 22–23
documents, 22
ensureIndex command, 24
on Heroku, 207
hierarchy of data, 25
horizontal scaling, 24
inserting documents, 26
installing, 25
installing for Digital Ocean, 223–224
Mongoose ODM, 45
overview, 22
playground database, 25
querying, 23
querying documents, 26
schemaless, 24
tables, 22
test database, 25
user and roles, 22
using, 25
Mongoose ODM with Post endpoint, 46–48

N

namespacing routers, 61
nav bar, updating in login form, 114
ng-annotate tool, using with Uglifier, 76
ng-route, 104–106
ng/websockets.js file, creating, 133
Node Cluster, using with Heroku, 212
Node integration with Angular. *See also* Angular.js
$http, 52
addPost() method, 56
cleaning up server.js, 58–62
post ordering, 57
promises, 52
reading posts with $http, 53–54
saving posts with $http, 56
serving posts.html, 55
Node stack, writing tests for, 164
node-http-proxy, 234–235
Node.js. *See also* authentication in Node.js
booting inside Protractor, 149–151
databases for Express, 20
in enterprise, 14
“Hello World” server, 19

HTTP module, 19
installing, 18
modules and NPM, 17–18
ORMs (object related mappers), 20
overview, 13
performance, 15
startup vs. enterprise, 13
Walmart, 15
web server, 19
Node.js API
creating posts, 44
Mongoose ODM, 45
Mongoose with `Post` endpoint, 46–48
`server.js`, 48
stock endpoint, 42–43
Node-server testing. *See also* tests
base router, 169
code coverage, 175–176
controllers with authentication, 173–174
GET action, 167
JSHint, 178
Mocha, 165–166
models in controller tests, 171–173
`npm` package, 175
`npm` test command, 177
`POST /api/posts endpoint`, 173
post controller, 167
SuperTest, 168
NPM and modules, 17–18
NPM package, Express, 20–21
`npm` package, using with Node, 175
`npm` test command, 177

O

open source environment, 13
ORMs (object related mappers), 26

P

PaaS (platform-as a service), Heroku, 202
package manager, NPM, 17
pages. *See* recent posts page
password validation in JWT, 91–93
passwords, using hashing algorithms, 94
PayPal in enterprise, 14
performance
as benefit of Web architecture, 6
Node.js, 15
`POST /api/posts endpoint`, testing, 173
`Post` endpoint, using Mongoose with, 46–48
post notifications, publishing, 130–132
post ordering, fixing, 57
posts
adding to recent posts page, 34–37
making with Protractor, 154–155
Node.js API, 44
`posts.html`, serving through Node, 55
progressive enhancement, 4–5
promises
clarifying with `chai-as-promised`, 159
using with Node.js and Angular, 52
prototyping, 6
Protractor. *See also* testing frameworks
booting Node inside, 149–151
Chai assertion, 157
configuring, 147–149

Protractor (*continued*)
DOM element for locator, 151–152
editing `server.js`, 150–151
`enableTimeouts` setting, 147
expectations, 156–158
installing Mocha, 147–148
locators, 151–152
login form, 153–154
login nav link, 152–154
making posts, 154–155
`onPrepare` function, 149–150
running, 147–149
setting up, 144
wiping database after running, 155
Protractor test
`e2e` directory, 146
.spec.js suffix, 147
user login and posting, 146
publishing events from clients, 139
pubsub message broker, 137–138
pushstate, enabling, 118, 121

Q

QUnit testing framework, 145

R

recent posts page
`$scope`, 32
adding posts, 34–37
declaring controller, 31
including Angular, 31–33
JSON representation, 32
`<script>` tag, 31–32
static markup, 26
Redis
on Heroku, 208–209
installing for Digital Ocean, 225–226
pubsub message broker, 137–138
`systemd` commands, 226
registration, 119
remembering users, 121
routers, namespacing, 61
routing, 104–106

S

Sass CSS preprocessor, 80
`<script>` tag, using with recent posts page, 31–32
security in WebSocket, 136
`server.js`
breaking out `/api/posts`, 59–61
breaking out `sendfile` endpoint, 62
cleaning up, 58–62
JWT (JSON Web Token), 90
namespacing routers, 61
social API, WebSockets in, 129–132
social app
running, 232
running on Digital Ocean, 227
running under `systemd`, 228
social networking project. *See* recent posts page
social posts, authenticating, 116–117
.spec.js suffix, using in Protractor test, 147

SSH key, using with Digital Ocean, 221
startup
  vs. enterprise, 13
  Node.js in, 14
static apps
  Ajax-empowered JavaScript, 4
  mobile APIs, 5
  progressive enhancement, 4–5
static assets, serving in Angular, 63–64
static mockup, building, 26
strings, passing messages as, 127
Stylus CSS preprocessor, 80
SuperTest
  done callback, 168
  using base router with, 170
  using with Node, 168

T

TCP layer, 127
templates/posts.html file, 105
test code, placement in directory, 146
testing controllers with authentication, 173–174
testing frameworks. *See also* Angular.js testing; Protractor
  Jasmine, 145
  Mocha, 145
  QUnit, 145
tests, writing for Node stack, 164. *See also* Node-server testing
thick clients, 6
token authentication
  JWT (JSON Web Token), 89
  types, 88

U

Uglifier minification tool
  defining dependencies, 76
  installing, 72
  ng-annotate tool, 76
  rebuilding /assets/app.js, 77
  "Unknown provider" error, 75
unit testing, 164
UNIX processes, hosting, 203
"Unknown provider" error, seeing in Angular, 75
URL, updating, 108
.use method, passing namespaces into, 61
users, foreign keys, 122
users, remembering, 121

W

Walmart, use of Node.js, 15
web architecture
  back end, 6
  benefits, 6
  flow, 6
  modularity, 6
  performance, 6
  prototyping, 6
  standardized tools, 6
web pages. *See* recent posts page
web server, creating with Node.js, 19

websites
Digital Ocean private networking, 235
"Introduction to NPM," 18
JSHint documentation, 178
Node.js, 14
WebSocket architecture
message broker, 137
multiprocess/multiserver design, 137–138
publishing events from clients, 139
reconnection, 135
security, 136
WebSockets
in Angular.js, 133–134
broadcasting events, 131
client-server communication, 128
dynamic hostname, 140
loading data, 128
`ng/websockets.js` file, 133
overview, 127
publishing post notifications, 130–132
in social API, 129–132
trouble with, 128
uses, 128
`ws` websocket, connecting to, 129–130
`wscat` command, 130

Y

Yammer, 14

Z

zero-downtime deploys
`.disconnect()` method, 229
`.fork()` method, 229
`boot.js` script, 229–231
Digital Ocean, 229–233
integration into `systemd`, 233
ZeroMQ message broker, 137