sitepoint

JUMP START NODE.JS

BY **DON NGUYEN**

Jump Start Node.js

by Don Nguyen

Product Manager: Simon Mackie
Technical Editor: Diana MacDonald
Expert Reviewer: Giovanni Ferron
Indexer: Fred Brown
Editor: Kelly Steele
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9873321-0-3 (print)

ISBN 978-0-9873321-1-0 (ebook)
Printed and bound in the United States of America

About the Author

Don began his programming career with strongly typed, object-oriented languages such as Java and C++. He used his engineering training to build real-time trading systems designed to be scalable and fault tolerant.

While his first introduction to functional programming was somewhat of a shock to the system, the beauty and elegance of weakly typed dynamic languages such as Python and Ruby shone through. Don has programmed in a variety of web environments including ASP, PHP, and Python, but feels that Node.js is foremost at handling the modern demands of the real-time web.

About the Expert Reviewer

Giovanni Ferron is a web developer currently living in Melbourne, Australia. He has worked for major media companies such as MTV and DMG Radio Australia, and co-founded the website Stereomood.com.[1] A couple of years ago, he fell in love with Node.js and has been spending his nights programming in JavaScript ever since.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums.

[1] http://stereomood.com

This book is dedicated to my Mum and Dad.

To my Dad for his endless patience in driving me to rowing, martial arts, and tennis practice, and for his never-ending support.

To my Mum for cooking dinner with one hand and sketching out the basics of object-oriented programming and database normalization with the other.

Table of Contents

Preface ix

Who Should Read This Book x

What's in This Excerpt x

What's in The Rest of The Book x

Where to Find Help xi

The SitePoint Forums xii

The Book's Website xii

The SitePoint Newsletters xii

Your Feedback xiii

Friends of SitePoint xiii

Online Quiz xiii

Acknowledgments xiii

Conventions Used in This Book xiv

Code Samples xiv

Tips, Notes, and Warnings xv

Chapter 1 Coming to a Server Near You 1

Why Node.js? 1

Strengths and Weaknesses 4

In the Beginning 5

Installation 5

Assembling the Pieces 7

A Basic Form 9

The Database 12

Summary 16

Preface

One of the difficulties I had when trying to learn Node.js was how to get started. The references that I found either dealed with quasi-academic topics such as datagrams and event emitters, or else myopically focused on a topic without regard for the big picture. This book takes you through the complete process of building an application in Node.js. It starts with the canonical "Hello World" example, and goes on to build a real-time web application capable of sending trading information to thousands of connected clients.

What make Node.js different? First, it provides a unified language between the back end and front end. This means that all your thinking can be in a single language, with no cognitive overhead when switching from front end to back. Furthermore, it allows for shared code libraries and templates. This opens up a lot of interesting possibilities, the surface of which is just beginning to be properly explored.

Second, it's fast. One of the common complaints of interpreted languages such as PHP, Python, and Ruby is speed. Jason Hoffman, CTO of Joyent, has discussed how Node.js is at the point where its performance can break operating systems. A single core with less than 1GB of RAM is capable of handling 10GB of traffic and one million connected end points. Combining 24 of these into a single machine produces an overall level of throughput that exceeds the capacity of operating systems and TCP/IP stacks. In other words, with a properly designed application it's not Node.js that's the bottleneck—it's your operating system.

Third, its nonblocking architecture is made for the real-time web. JavaScript was chosen as a language because it's based on nonblocking callbacks and has a very small core API. This means it was possible to build the entire Node.js ecosystem around nonblocking packages, of which there are currently in excess of ten thousand. The end result is a platform and ecosystem that architecturally fits perfectly with the modern demands of the real-time web.

I trust by now that you're excited by the possibilities of what Node.js can do for your real-time application. By the end of this book, I'm confident that you'll have the skills to be able to start dissecting and solving all but the most esoteric of problems. There is no greater joy in software than solving a complicated task and thinking at the end of it, "That was all I had to do!" It is one I've experienced many

times working with Node.js, and it's my hope that you enjoy the same satisfaction both throughout the book and in using Node.js to solve your real-world problems.

Who Should Read This Book

This book is aimed at two target audiences. You might be a front-end engineer who is interested in looking at server-side development. An existing knowledge of JavaScript will prove useful, even if you are unfamiliar with certain server-side engineering techniques. Rest assured that by the end of the book, the topic will be covered in sufficient detail for you to know how to apply your front-end skills to back-end problems.

The second potential reader is the server-side engineer who uses another language such as PHP, Python, Ruby, Java, or .NET. The main benefit you'll derive is seeing how your existing architectural, design, and pattern knowledge is applied to the world of Node.js. You may have little to no knowledge of JavaScript, but this should pose no big hindrance. By design, it's an easy language to learn, and we will have covered many examples of both its syntax and idiosyncratic features by the end of the book.

What's in This Excerpt

This excerpt comprises one chapter:

Chapter 1: *Coming to a Server Near You*

Node.js is introduced and its features and benefits explained. We then build a simple application to introduce Node.js. The application sends data from a form to MongoDB, a NoSQL database.

What's in The Rest of The Book

The rest of the book comprises the following chapters.

Chapter 2: Let's Get Functional

This chapter introduces programming with Node.js in a functional style. We'll build a stock exchange trading engine that's capable of accepting orders and matching trades.

Chapter 3: Persistence Pays

Here we explore MongoDB. I'll explain the use cases of MongoDB and how it compares to traditional SQL databases. We'll then look at the MongoDB query language and show how it can be integrated with your Node.js projects.

Chapter 4: Beautifying with Bootstrap

Bootstrap is a front-end framework from Twitter that makes it easy to build professional-looking sites. We'll look at some of the most common widgets and use them to build a login screen and stock portfolio tracker.

Chapter 5: The Real-time Web

In this chapter we examine Socket.IO. We'll see how learning one simple API can allow real-time communication across a range of projects without needing to worry about browser versions or communications protocols.

Chapter 6: Backbone

We'll discuss how frameworks are useful in managing client-side JavaScript in this chapter. We'll then show how Backbone.js can be incorporated into your project by updating trades in the browser in real time.

Chapter 7: Production

In the final chapter, we'll look at the main differences between a development and production environment. We'll cover various deployment options before getting our application deployed and running live!

Where to Find Help

Node.js represents a paradigm shift in web development, providing a unifying language all the way from the front end to the back end. It has experienced a boom in popularity on GitHub, so chances are good that by the time you read this, some minor detail or other of the Node.js platform will have changed from what's described in this book. Thankfully, SitePoint has a thriving community of JavaScript developers ready and waiting to help you out if you run into trouble, and we also maintain a list of known errata for this book that you can consult for the latest updates.

The SitePoint Forums

The SitePoint Forums[1] are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions, too. That's how a discussion forum site works—some people ask, some people answer and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of fun and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and just have fun.

The JavaScript & jQuery Forum[2] is probably the best place to head to ask any questions.

The Book's Website

Located at http://www.sitepoint.com/books/nodejs1/, the website that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.[3]

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's website will always have the latest information about known typographical and code errors.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters such as the *SitePoint* newsletter, *JSPro*, *PHPMaster*, *CloudSpring*, *RubySource*, *DesignFestival*, and *BuildMobile*. In them you'll read about the latest news, product releases,

[1] http://www.sitepoint.com/forums/
[2] http://www.sitepoint.com/forums/forumdisplay.php?15-JavaScript-amp-jQuery
[3] http://www.sitepoint.com/books/nodejs1/code.php

trends, tips, and techniques for all aspects of web development. Sign up to one or more of these newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you're unable to find an answer through the forums, or if you wish to contact SitePoint for any other reason, the best place to write is `books@sitepoint.com`. We have a well-staffed email support system set up to track your inquiries, and if our support team members are unable to answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Friends of SitePoint

Thanks for buying this book. We really appreciate your support! We now think of you as a "Friend of SitePoint," and so would like to invite you to our special "Friends of SitePoint" page.[4] Here you can SAVE up to 43% on a range of other super-cool SitePoint products, just by using the password: **`friends`**.

Online Quiz

Once you've mastered Node.js, test yourself with our online quiz. With questions based on the book's content, only true Node.js ninjas can achieve a perfect score. Head on over to http://quizpoint.com/#categories/NODE.JS.

Acknowledgments

I'd like to thank the wonderful team at SitePoint for guiding me through the process of publishing my first book: to Simon Mackie for patiently answering all my questions and keeping everything on track; to Giovanni Ferron for reviewing my code and pointing out bugs; to Diana MacDonald for ensuring clarity in my code; to Kelly Steele for keeping my English stylistically and grammatically correct; and to Alex Walker for the wonderful cover art.

To my longtime friends, Jarrod Mirabito and Andrew Prolov: thanks for helping review my initial work long before it was even a book proposal. To my flatmate,

[4] http://sitepoint.com/friends

Angelo Aspris: thanks for patiently accepting "busy writing" as an excuse for long stretches of absenteeism. To Andy Walker: thanks for keeping the flame of entrepreneurship burning brightly. To my lifelong friend Chuong Mai-Viet: thanks for dragging me away from the desk on bright and sunny days to keep my golf handicap down and my Vitamin D intake up.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a ⋮ will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://jspro.com/raw-javascript/how-to-create-custom-even
➥ts-in-javascript/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 1

Coming to a Server Near You

> "You see things and you say, 'Why?' But I dream things that never were, and I say, 'Why not?'"
>
> —George Bernard Shaw

Why Node.js?

If a picture speaks a thousand words, what would it take to speak a thousand pictures? Or for that matter, an infinite number of pictures? My first introduction to Node.js was through WordSquared,[1] seen in Figure 1.1. This is an online, real-time, infinite game of Scrabble built using the same technologies that we'll discuss in this book. As soon as I set eyes on the game, I had to find out more about the technology behind it, and I hope you feel the same.

What's incredible about the game is that it was prototyped in just 48 hours as part of Node.js Knockout.[2] Bryan Cantrill, VP of Engineering at Joyent (which manufactures Node.js) has said that when doing things in Node.js, you sometimes get the feeling of "Is this it? Surely it needs to be more complicated." This is a sentiment

[1] http://wordsquared.com/
[2] http://nodeknockout.com/

I share. Node.js is a joy to work with, and I intend to share that with you through the code we'll write throughout this book.

Figure 1.1. WordSquared: a way to examine Node.js in action

Node.js is a server-side JavaScript platform that consists of a deliberately minimalist core library alongside a rich ecosystem. It runs atop the V8 JavaScript engine, which makes it very fast thanks to the talented engineers at Google. JavaScript is popular on the client side, but it was chosen as the target language primarily for engineering considerations, the details of which will be discussed as the chapter unfolds.

The home page of Node.js describes it thus:

> "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."
>
> —http://nodejs.org/

This may seem cryptic to newcomers, but it succinctly summarizes some of the key strengths of Node.js and is worth exploring in more detail. People are often taken aback when they hear that JavaScript is the targeted programming language. That's because there's a perception in the programming community that JavaScript is not a "real" language such as C, C++, or Java. Yet JavaScript did have its genesis as an interpreted language for the browser; the "Java" part of the name was actually chosen to capitalize upon the perceived popularity of the Java programming language at the time.

Since its humble beginnings, JavaScript has proliferated and is now supported in every major web browser, including those on mobile devices. Not only is it a popular language, but the tools and frameworks currently available for it make it qualify as a powerful engineering tool. JavaScript as a server-side platform supports continuous integration, continuous deployment, connections to relational databases, service-oriented architecture, and just about every other technique available to its more well-established brethren.

In conjunction with Google's V8 JavaScript engine, it is now extremely fast; in fact, it's several times faster than other scripted languages such as Ruby and Python. Against Python3, JavaScript V8 Engine has a median benchmark 13 times as fast with a roughly similar code size.[3] Against Ruby 1.9, the median benchmark is eight times as fast.[4] These are incredible benchmarks for a dynamic language, and are due in no small part to V8 optimizations such as compilation into machine code pre-execution.

On Benchmarking

Benchmarking is an intricate topic,[5] and the numbers above should be taken with the proverbial grain of salt. Whenever any discussion of benchmarking arises, it is generally qualified with "it depends." The main purpose of discussing the benchmark was to dispel any misconception of JavaScript being inherently slow.

The official description talks about the event-driven, non-blocking I/O model. Traditionally, programming is done in a synchronous manner: a line of code is executed, the system waits for the result, the result is processed, and then execution resumes. Sometimes waiting for the result can take a long time; for example, reading from a database or writing to a network.

In languages such as Java and C#, one solution is to spawn a new **thread**. A thread may be thought of as a lightweight process that performs tasks. Threaded programming can be difficult because multiple threads can be trying to access the same resource concurrently. Without dissecting the intricacies of multi-threaded programming, you can imagine it would be disastrous for one thread to be incrementing a counter while another thread is decrementing the counter at the same time.

[3] http://shootout.alioth.debian.org/u32/benchmark.php? test=all&lang=v8&lang2=python3
[4] http://shootout.alioth.debian.org/u32/benchmark.php? test=all&lang=v8&lang2=yarv
[5] http://shootout.alioth.debian.org/dont-jump-to-conclusions.php

JavaScript approaches the problem differently. There is only ever a single thread. When doing slow I/O operations such as reading a database, the program does not wait. Instead, it immediately continues to the next line of code. When the I/O operation returns, it triggers a callback function and the result can be processed. If the mechanics of this seems slightly counterintuitive, rest assured that by the end of the book it will be second nature, because we'll be seeing this pattern over and over again. Node.js offers a simple, fast, event-driven programming model well-suited to the asynchronous nature of modern-day web applications.

Strengths and Weaknesses

Node.js is not a panacea. There is a certain class of problems in which its strengths shine through. Computer programs can be generally classified according to whether they are CPU bound or I/O bound. **CPU bound** problems benefit from an increase in the number of clock cycles available for computation. Prime number calculation is a good example. Node.js, however, is not designed to deal with these CPU bound problems. Even before Node.js formally existed, Ryan Dahl proposed the following:[6]

> "There is a strict requirement that any request calculates for at most, say, 5ms before returning to the event loop. If you break the 5ms limit, then your request handler is never run again (request handlers will be tied to a certain path/method, I think). If your uploaded server-side JavaScript is stupid and is blocking (… it calculates the first 10,000 primes) then it will be killed before completion.
>
> …
>
> Web developers need this sort of environment where it is not possible for them to do stupid things. Ruby, Python, C++, [and] PHP are all terrible languages for web development because they allow too much freedom."
>
> —Ryan Dahl

I/O bound problems are alleviated by increased throughput in I/O such as disk, memory, and network bandwidth, and improved data caching. Many problems are I/O bound, and it's in this domain that Node.js truly shines. Take, for example, the

[6] http://four.livejournal.com/963421.html

C10K problem,[7] which poses the dilemma of how to handle ten thousand or more concurrent connections for a web server. Some technology platforms are ill-equipped for managing this type of capacity and require various patches and workarounds. Node.js excels at this task because it's based on a nonblocking, asynchronous architecture designed for concurrency.

I hope I've whetted your appetite; it is now time to begin.

On Speed

The book's core project involves building a real-time stock market trading engine that will stream live prices into a web browser. The more inquisitive among you may ask, "Shouldn't a stock market trading engine be built using a language such as C for faster-than-light, jaw-dropping speed?" If we were building a stock market engine for actual trading, my answer would be "Yes."

The main goal of this book is to transfer the skill set rather than the actual project into the real world. There is a narrow domain of "hard" real-time applications such as a stock exchange where specialized software and hardware are required because microseconds count. However, there is a much larger number of "soft" real-time applications such as Facebook, Twitter, and eBay where microseconds are of small consequence. This is Node.js's speciality, and you'll understand how to build these types of applications by the end of this book.

In the Beginning

Let's run through installing Node.js, setting up a web framework, building a basic form page, and connecting to a database.

Installation

It is possible to install Node.js from the raw source code, but it's much simpler to install using a package manager.

Go to https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager and follow the instructions for your particular distribution. There are currently instructions for Gentoo, Debian, Ubuntu, openSUSE and SLE (SUSE Linux Enterprises),

[7] http://en.wikipedia.org/wiki/C10k_problem

Fedora, and RHEL/CentOS, as well as several other Linux distributions. Windows and Mac OS X instructions are also available.

After installation, type `node` at the command line to bring up the read-eval-print loop (REPL). You can test your installation by typing:

```
console.log('Hello world')
```

You should receive:

```
Hello world

undefined
```

If you did, congratulations! You have just written your first Node.js program. The console always prints out the return type. This is why `Hello world` is followed by `undefined`. Now that you've written the canonical `Hello world` program, I'm happy to inform you that it will be the first and last boring program you'll see in this book. To prove my point, we'll now jump straight in and build an authentication module to verify usernames and passwords using cloud-based NoSQL technology. (If you're scratching your head and wondering what this means, don't worry—all will be revealed shortly!)

So what is **cloud-based NoSQL** technology, and why should you care? There's been a lot of hype around cloud-based technology. For our purposes, cloud technology is useful because it allows our applications to scale: additional virtual servers can be brought online with just a few mouse clicks.

The NoSQL movement is relatively recent and, as such, it is difficult to arrive at a comprehensive definition of what the technology encompasses. Broadly, it can be thought of as a family of database technologies built for handling large masses of unstructured and semi-structured data. Companies such as Google, Amazon, and Facebook make extensive use of NoSQL technology to house the vast quantities of data generated by their users.

For this book, we will be using MongoDB as our NoSQL database for reasons that will be fleshed out in more detail in a later chapter. MongoDB is a mature and scalable document-oriented database that has been deployed successfully in enter-

prise environments such as foursquare and craigslist.[8] A **document-oriented database** is a database where the traditional database row is replaced with a less structured document, such as XML or JSON. MongoDB allows ad hoc queries and so retains much of the flexibility that SQL offers. I've chosen MongoLab[9] as a cloud provider for our stock monitoring application because it has a generous free hosting plan in place.

Assembling the Pieces

The first step is to go to MongoLab, seen in Figure 1.2, and sign up an account. Then click on **Create New** next to Databases. Leave the cloud provider as Amazon EC2 and select the free pricing plan. Choose a database name, a username, and password.

Now we'll set up the web framework. Node.js provides a built-in, bare-bones HTTP server. Built on top of this is Connect, a middleware framework that provides support for cookies, sessions, logging, and compression, to name a few.[10] On top of Connect is Express, which has support for routing, templates (using the Jade templating engine), and view rendering.[11] Throughout this book we'll predominantly use Express.

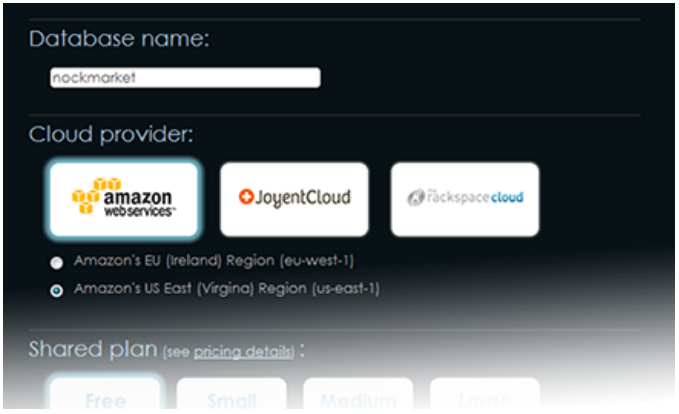

Figure 1.2. Signing up to MongoLab

Express can be installed with the following command:

[8] http://api.mongodb.org/wiki/current/Production%20Deployments.html
[9] http://mongolab.com
[10] http://www.senchalabs.org/connect/
[11] http://expressjs.com/guide.html

```
sudo npm install -g express@2.5.8
```

We use the `-g` switch to indicate that the package should be installed globally and `@2.5.8` to indicate the version we'll use in this book.

Global versus Local

On the official Node.js blog, sparing use of global installs is recommended.[12] The guideline is to use a global install if the package needs to be accessed on the command line, and a local install if it needs to be accessed in your program. In some cases—such as Express where both are required—it's fine to do both.

An application with basic default options can be created using this command:

```
express authentication
```

You should see output similar to the following:

```
create : authentication
create : authentication/package.json

⋮

dont forget to install dependencies:
$ cd authentication && npm install
```

To explain the last line (`$ cd authentication && npm install`), `npm install` will install packages according to the dependencies specified in **package.json**. This is a plain-text file that specifies package dependencies in JavaScript Object Notation. For this project, modify **package.json** to contain the following text:

chapter01/authentication/package.json *(excerpt)*

```
{
    "name": "authentication"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
```

[12] http://blog.nodejs.org/2011/03/23/npm-1-0-global-vs-local-installation/

```
      "express": "2.5.8"
    , "jade": "0.26.1"
    , "mongoose": "2.6.5"
  }
}
```

Avoiding Dependency Hell

Instead of a particular version number, it's possible to specify "*", which will retrieve the latest version from the repository. While it may be tempting to always work with the latest version, just because your application is compatible with one version, it in no way guarantees that it will still be compatible with the next. I recommend always stating the version number at the very least. Even this is no guarantee of correctness. If you use a package that employs the "*" syntax, you will run into a similar problem. For a more robust solution, I recommend you look into Shrinkwrap.[13]

These dependencies can then be installed by changing to the **authentication** directory (`cd authentication`) and typing `npm install`. Then type `node app`, and after navigating to http://localhost:3000 in your browser, you should see the message, "Welcome to Express." Easy, wasn't it?

A Basic Form

The next step is to create a basic form to post data to the server. With a web server such as Apache, you'd normally place a file into a directory to be served to the user. Node.js is deliberately minimalist, with very little that is automatically done. If you wish to read a file from the disk, you'll need to explicitly program it. Kill your app and, at the top of **app.js**, add the `fs` dependency to allow reading from the file system. Then add the **/form** route:

chapter01/authentication/app.js *(excerpt)*

```
var express = require('express')
  , routes = require('./routes')
  , fs = require('fs');
⋮
// Routes
```

[13] http://blog.nodejs.org/2012/02/27/managing-node-js-dependencies-with-shrinkwrap/

```
app.get('/', routes.index);

app.get('/form', function(req, res) { (1)
  fs.readFile('./form.html', function(error, content) { (2)
    if (error) {
      res.writeHead(500);
      res.end();
    }
    else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(content, 'utf-8');
    }
  });
});
```

The above listing has several important features.

(1) This line handles all the routing to **/form**. Whenever anybody makes a request to http://localhost:3000/form, Express captures the request and it is handled by the code within.

(2) This line attempts to read **form.html** from the file system and serve the HTML file. If there's an error, a 500 response will be sent back to the browser.

Callback Functions

This is the first time we've broached callback functions, which might be tricky for those starting from other languages. As an example, type the following into a file named **cb.js**:

```
setTimeout(function() {console.log('first or second');}
  , 500);
console.log('we will see');
```

Run `node cb` and note the text order. The first line sets a timeout for a function to be executed in 500ms. The program immediately resumes execution, which is why `we will see` is printed before `first or second`.

In Node.js, everything is asynchronous by design, so only your own code will block a process. The first line in this code sample could be any I/O operation including reading and writing from databases and networks. In every instance, nothing I/O-related will block and the rest of your program will immediately ex-

ecute. This paradigm can be confusing at first, but you'll see it's quite an elegant solution once you start to gain familiarity with it.

We can create a very simple **form.html** as follows:

chapter01/authentication/form.html

```
<form action="/signup" method="post">
  <div>
    <label>Username:</label>
    <input type="text" name="username"/><br/>
  </div>
  <div>
    <label>Password:</label>
    <input type="password" name="password"/>
  </div>
  <div><input type="submit" value="Sign Up"/></div>
</form>
```

When you visit http://localhost:3000/form, you should see your form rendered in the page as in Figure 1.3.

Figure 1.3. Form screen

Not quite what you imagined Web 2.0 to look like? Never mind, we'll cover styling your page in a subsequent chapter. There's enough to absorb at the moment without concerning yourself with how to make your elements shine on the page.

When a user clicks on the **Sign Up** button, the form data will be posted to **/signup**. We'll now set up a handler in Express to process the form data. Copy the following code into **app.js** where you left off:

chapter01/authentication/app.js *(excerpt)*

```
app.post('/signup', function(req, res) {
  var username = req.body.username;
  var password = req.body.password;
  User.addUser(username, password, function(err, user) {
```

```
    if (err) throw err;
    res.redirect('/form');
  });
});
```

The first line of this code listing is similar to what we've seen before except that `app.get` is replaced with `app.post`. The next two lines extract the username and password from the request object. The next line `User.addUser(username, password, function(err, user) {` looks a little mysterious. Where does `User` come from? It comes from the users module, which we'll create shortly. Node.js comes with an excellent module system that allows the programmer to encapsulate functions. Separation of concerns is a core programming principle for managing complexity, and Node.js makes it easy to write modules that perform specific, well-defined tasks. After adding the user, we redirect back to the form page.

Real-world Development

In the real world, Express provides a lot of useful abstractions. This means that in future it won't be necessary to read files from the disk in the manner we are now. Note also that, by default, Node.js does not support "hot swapping" of code, meaning that changes are only reflected when you restart Node.js. This can become tedious after a while, so I suggest at some point you install node-supervisor to automatically restart the system upon changes to the file.[14]

The Database

In order to create a users module, we'll need to write a database module. Place the following code into a file called **db.js** in a directory called **lib**:

chapter01/authentication/lib/db.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

module.exports.mongoose = mongoose;
module.exports.Schema = Schema;

// Connect to cloud database
```

[14] https://github.com/isaacs/node-supervisor

```
var username = "user"
var password = "password";
var address = ' @dbh42.mongolab.com:27427/nockmarket';
connect();

// Connect to mongo
function connect() {
  var url = 'mongodb://' + username + ':' + password + address;
  mongoose.connect(url);
}
function disconnect() {mongoose.disconnect()}
```

It is surprisingly simple to connect to a cloud-based NoSQL provider. You will need to replace `username`, `password`, and `address` with your own personal details. For example, `var username = "user"` might become `var username = "bob927"`. MongoLab provides a user tab for adding new users, seen in Figure 1.4.

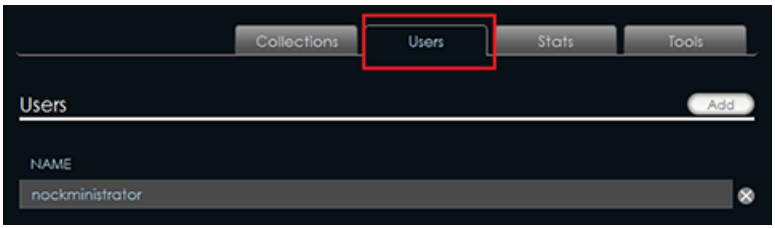

Figure 1.4. The **Users** tab in MongoLab

Figure 1.5 shows where you can obtain the database address.

Figure 1.5. Obtaining the address in MongoLab

We will use the Mongoose package, which allows a connection from Node.js to MongoDB. `module.exports` is the Node.js syntax for exposing functions and variables to other modules.

We will expose the Mongoose driver as well as the Mongoose `Schema` object. A **schema** is simply a way to define the structure of a collection. In this case, our schema consists of a `username` string and a `password` string. Other than that, we define functions to connect and disconnect from the database.

The next step is to define the user module. Place this code in a file called **User.js** in a directory called **models**:

chapter01/authentication/models/User.js *(excerpt)*

```
var db = require('../lib/db');

var UserSchema = new db.Schema({
    username : {type: String, unique: true}
  , password : String
})

var MyUser = db.mongoose.model('User', UserSchema);

// Exports
module.exports.addUser = addUser;

// Add user to database
function addUser(username, password, callback) {
  var instance = new MyUser();
  instance.username = username;
  instance.password = password;
  instance.save(function (err) {
    if (err) {
      callback(err);
    }
    else {
      callback(null, instance);
    }
  });
}
```

We begin by importing our previously defined database module and defining a simple schema. We can then instantiate a new user, set the username and password, and save to the database using the `save` function. If there's an error, we send it to the callback function. Otherwise, we send back the actual user object.

Password Security

In this example, we store the raw password in the database, which presents a security risk. Normally, the password would be encrypted. There are other secure techniques based on salting the encrypted password to make it more resistant to dictionary attack.

We are now ready to integrate our user module with Express. This involves adding one additional line of import code to our **app.js** file:

chapter01/authentication/app.js *(excerpt)*

```
var express = require('express')
  , routes = require('./routes')
  , fs = require('fs')
  , User = require('./models/User.js');
```

Now if you restart your app and submit your form, you should see the data appear in MongoLab, as in Figure 1.6.

Figure 1.6. New user data in MongoLab

Note that the collection "users" (which can loosely be thought of as a table) was created automatically. That's all there is to building your very first cloud-enabled NoSQL solution. If your next project begins to take on the popularity of Facebook or Twitter, your infrastructure will be ready to scale!

Directory Structures

It might seem a little convoluted to be storing a single file in a separate directory, but as your code base grows larger, it is necessary to group files together in a logical manner to enhance understandability and maintainability. By convention, I put database models into a **models** directory, and business logic and supporting functions into a **lib** directory.

Summary

In this chapter, we've covered the following:

- the philosophy behind Node.js and what makes it different from other programming environments
- installing Node.js
- executing programs from the console using the read-eval-print loop
- building a basic Express application
- basic package management
- modularizing code and exporting functions
- submitting data to a form
- storing data in MongoDB using the cloud platform from MongoLab