

```

26 var errors = false;
27 var name = request.body.name;
28 var slug = request.body.slug;
29
30 if(validation.isNullOrEmpty([name, slug])) {
31   errors = true;
32 }
33
34 if(errors)
35   validation.errorRedirect(response, '/admin/categories', 'categoryCreateerror');
36 else {
37   var c = new model.CategoryModel({
38     name: name,
39     slug: slug,
40     isDefault: false
41   });
42
43   c.save(function(error){
44
45     if(error) {
46       validation.errorRedirect(response, '/admin/categories', 'categoryCreateerror');
47     }
48     else {
49       validation.SuccessRedirect(response, '/admin/categories', 'categoryCreated');
50     }
51   });
52 }
53
54 // Admin - edit Category
55
56 exports.CategoryEdit = function(request, response){
57
58   var id = request.params.id;
59
60   model.CategoryModel.findById( id, id, function(err, doc) {
61

```



Programming Node.js Applications

with Express.js & MongoDB

Ian Chursky

```

62   model.CategoryModel.findByIdAndUpdate(
63     { _id: request.body.id },
64     {
65       name: name,
66       slug: slug
67     },
68     { multi: true },
69     function(error, result){
70       if(error) {
71         validation.errorRedirect(response, '/admin/categories', 'categoryUpdateerror');
72       }
73       else{
74         validation.SuccessRedirect(response, '/admin/categories', 'categoryUpdated');
75       }
76     }
77   );
78
79   var errors = false;
80   var name = request.body.name;
81   var slug = request.body.slug;
82
83   if(validation.isNullOrEmpty([name, slug])) {
84     errors = true;
85   }
86
87   // Admin - edit Category

```

Programming Node.js Applications with Express.js and MongoDB

Ian Chursky

Contents

Preface	1
Who This Book Is For.....	1
Format	1
Code Samples & Repositories	1
Errata.....	2
Feedback	2
Chapter 1: A Node.js Primer.....	3
Node.js Introduction	3
Installing Node.js	4
Obligatory "Hello World" in Node.js.....	4
Installing Modules with npm	5
Chapter 2: Introduction to Express.js.....	7
Templates & Views.....	8
Dynamic Templates & Passing Data to Views	9
Chapter 3: Creating an MVC Express.js Application	12
Application Setup.....	12
Templating	13
Router.....	17
Controllers.....	18
Passing Data to Views	19
Partial Views	20
Summary	21
Chapter 4: Middleware & Configuration	22
Middleware	22
Configuration	27
Summary	28
Chapter 5: Debugging Node.js Applications	30
Installing Node Inspector.....	30
Debugging With Node Inspector	31
Summary	34
Chapter 6: Data Access & Validation With MongoDB	35
Installing Mongoose.....	36
Database Settings in config.js	37

Creating Data Models	38
Accessing Data in Controllers and Views.....	39
Editing and Deleting Data.....	43
Deleting Data	43
Editing Data	45
Testing It All Out	46
Validation of Data.....	47
Summary	50
Chapter 7: Relational Data in MongoDB.....	51
Updating Data Models	51
Adding Object References	54
Editing Items and Handlebars Helpers.....	56
Summary	59
Chapter 8: APIs & AJAX in Express.js Applications.....	60
What is an API?	60
Setup	62
Sending AJAX Requests.....	63
Summary	65
Chapter 9: Sessions & Authentication in Express.js.....	66
Getting Started with Sessions and Authentication.....	66
Authenticating Users Stored in a Database.....	71
Updating Code to Check Credentials on Sign In	76
Installation and Creating a Default User.....	77
Authenticated AJAX Requests	80
Summary	80
Chapter 10: Security	82
Cross-site Request Forgery	82
Anti-CSRF Tokens and AJAX Requests	84
Cross-Site Scripting (XSS)	84
Session Hijacking / Cookie Stealing.....	85
Use HTTPS	85
Summary	85
Chapter 11: Putting it All Together: Creating an Application	87
Setting Up Our Application	87
Middleware	90

Router.js.....	92
Views	93
Static Files.....	94
Running the Application	94
Data Models	95
Building the Admin Panel.....	96
Installation Routes & Creating a Default User	114
Dynamic Page & Post Routes	116
404 & Error Routes.....	117
Summary	118
Conclusion	119
Appendix A: Code Samples & Repos.....	120
Appendix B: MongoDB Shell.....	121

Acknowledgements

Big thanks to all of my family and friends for their continued love and support over the years.

Big thanks to all of my teachers and colleagues who have invested their time, effort (and patience) with me in so many ways. I would not be where I am today without all of your help.

Biggest thanks of all to my wife Lauren and my 2 daughters: Mia and Calista. Love you all so much.

Preface

Moving into the second decade of the 21st century, if you have been immersed in the web development community - and more specifically, the JavaScript community -- then you likely remember hearing something about [Node.js](#) discussed somewhere at some point. Maybe you already know all about it, maybe you know a little, or maybe you're not yet familiar with it. Regardless of where you are coming from, it is my sincere hope that by the time you reach the end of this discussion you will be well immersed within some of the core concepts of building a Node.js application and you will feel confident and ready to take on any future projects that utilize it as a technology.

This journey is all about how to program a Node.js web application using 2 very popular projects that are also open-source like Node.js. One of these is Express.js which is written on top of Node.js and the other is a NoSQL database called MongoDB which exists independently from Node.js but has been noticeably embraced by the Node.js community. These technologies working together in tandem will be the focus of this exploration.

Who This Book Is For

This book is for the developer who maybe has some experience with writing a bit of JavaScript or jQuery code but has not yet ventured into the world of Node.js and server-sided JavaScript. This book will provide you the essential basics of creating an application using Node and in the process you hopefully will be able to get a grasp of the concepts of building within it for other projects that you take on in the future.

This book is also for the intermediate or seasoned JavaScript application developer who has maybe done a bit of Node.js before but has now been given a project that utilizes Express.js with MongoDB and they need some basic introductory tutorials, a quick reference of working code, and a high-level overview of the common components therein. This book will give you the essentials to get up and going (hopefully as fast as possible).

Please note that this book will not cover a basic introduction to HTML and JavaScript. The content in this book assumes that you have had at least some experience working with HTML and JavaScript – though, honestly, you really do not need all that much. If you do not really know HTML and JavaScript, it would be of great benefit for you to take a few online tutorial courses before returning to what is presented here. [W3Schools](#) is a very popular choice, but there are many others as well. All you really have to do these days is go to Google or YouTube and type in "learn X," where X is whatever subject you want to learn.

This book also assumes that you have a basic understanding of how the web operates and that web pages and resources (CSS files, images, cookies, etc.) are sent from server to client over HTTP(S) and the client (i.e. browser) can send data back to the server (e.g. submitting forms) via the same protocol.

It is also assumed that you have at least a very basic understanding of how [AJAX](#) works using [JSON](#) as a data-exchange format.

Format

The content within this volume is an ongoing discussion that somewhat progressive in nature. That is, each section builds upon concepts learned in previous sections so it is probably not recommended that you attempt to read later chapters as stand-alone discussions. However, those who are already familiar with the basics of Node.js and Express.js might find it worthwhile to skip to chapter 3 as the first 2 chapters are basic level introductions to Node.js and Express.js respectively.

Code Samples & Repositories

I believe that it is important to have examples of the code that is presented within these tutorials so I have essentially packaged what we learn in each chapter up. I have also written a fully functioning CMS in the same coding style and covering the same topics of the code that will be discussed. These are available in Appendix A.

Errata

While I and others have done our best to have proofread the content that is to follow, it is likely that there are a fair number of mistakes that still exist within. You can send me any notifications of such errors via e-mail: books@9bitstudios.com. Errors found will be fixed with future updates of the book, which I will try to churn out as fast as possible.

Feedback

Also, feel free to send additional feedback of any kind via email: books@9bitstudios.com. Just make sure to reference in the subject line your e-mail has to do with this book. I will do my best to respond to all those who make the effort to contact me. I am also on the Twitters [@ianchursky](https://twitter.com/ianchursky) and [@9bitStudios](https://twitter.com/9bitStudios) so if you want to leave 140 characters of feedback (or just say hi) you can contact me there as well.

So, without further ado, let's jump right in. We will start with a little history and an introduction to Node.js...

Chapter 1: A Node.js Primer

What is [Node.js](#)? Having been first published by Ryan Dahl in 2009, Node.js -- the open source software that allows the common web language of JavaScript to run outside the browser -- has absolutely grabbed ahold of imaginations in the world of technology in its first half-decade of life. If you follow the trends on [npm](#) in terms of downloads at the time of this edition in 2015, Node.js has only gotten more and more popular and has made some significant strides over a relatively short time span. Some high-profile companies have adopted it for at least some portion of functionality for their web applications and there are a number of others that have come on board (or are coming on board) as well. Outside of the software/application world, Node.js has also been used to do some pretty neat things. People have run it on the popular [Raspberry Pi microcomputer](#) to power home lighting systems, aquariums, and have even used it to [build robots](#).

In what follows, we're going to take a brief look at some of the components of Node.js, what it is, and why people have gotten excited about it.

Node.js Introduction

Before we get into what Node.js is, let's back up a minute and start by talking about web servers. You know about web servers, right? They're computers that have software running on them that are able to do any number of tasks. One of the more important functions of a web server in the context of the World Wide Web is being able to respond to HTTP requests from sent to it from clients -- which most of the time means browsers like Chrome, Firefox, Safari, and Internet Explorer... either desktop or mobile. Web servers are often written in languages like [Perl](#), [C](#), and [C++](#). They have other code like [Python](#) or [PHP](#) that runs on top of them so that when a client connects to a server, all the code runs, works its magic (hopefully in a semi-functional way) and returns a web page of HTML to the browser... or if the endpoint is a web service: an XML or a JSON payload. This is a pretty standard interaction that occurs billions of times each day across the world. If you do anything on the Internet, you participate in this process most often without even thinking about what is going on behind the scenes in the client-server interaction.

So where does Node.js come into the picture? Basically Node.js is an implementation where, instead of responding to HTTP requests, processing them, and returning web pages using the aforementioned languages, you use JavaScript instead! Node.js becomes a web server written in JavaScript. And it doesn't just stop there. You can also run Node.js right in your own desktop environment and use utilities to perform file processes and other tasks just as you could do if you were using an installed application that was written in Java, C++ or something else. So Node.js could perhaps be better described as "JavaScript running outside of the browser." Local applications are definitely a very useful component of Node.js, but we're going to specifically focus on the aspect of using Node.js as a web server. But I'd definitely recommend looking at a number of different utilities available for installation to see if you can use Node.js in your development processes or just as something that makes your life easier in some way.

So, if the real interesting thing about Node.js is that JavaScript runs on the server and returns web pages and you can program your own web server in JavaScript... why would you want to do that, you ask? After all, a lot of these web servers written in these other languages are really well established, have been around for years, and have been optimized for performance and reliability by a community of super-smart programmers who have been hammering away on these projects for what seems like forever. Why do I have to reinvent the wheel and write my own server software where I'll likely make all kinds of mistakes and leave security holes open? These are definitely all valid concerns. The real trade-off is that, depending on the type of application you are developing, in some cases the performance possibilities and simplicity that Node.js brings can offer real benefits that other languages do not (or at least not as easily). And don't fret. Fortunately there are modules out there -- like [Express](#), a web-application framework that runs on top of Node.js -- that are fully functioning web servers that have great communities and lots of testing behind them! You don't even have to write the web server yourself. You can just install it and be on your way!

But again you may still be wondering why people are going to all this trouble. As it turns out there are some aspects of the JavaScript language that have certain advantages that other languages don't. One of the big challenges for websites living on servers is handling many different requests coming into it at once. As traffic increases, performance often suffers resulting in slow loading web pages and a diminished user-experience (UX). You may have experienced this in your development career or just your everyday browsing experience (though high-traffic is not always the sole cause of slow websites). To "scale" with the increase in traffic a website's server has to do

something to handle it. One approach is to make use of [multithreading](#) which can be challenging to develop for. Another approach companies with high-traffic websites use is just to add a bunch more servers (which can also carry an increased cost of installation and maintenance). Node.js, however, because of the way that it is structured can handle many different concurrent connections and requests from clients because of JavaScript's event-driven architecture and the ability to make use of callback functions that execute when a resource is ready. Thus, Node.js can get a request, say "Here's the callback to call once everything ready" and then move on to handling the next request. As is summarized in the introduction to the book *Node.js: Up and Running*:

JavaScript is an event-driven language, and Node uses this to its advantage to produce highly scalable servers. Using an architecture called an event loop, Node makes programming highly scalable servers both easy and safe. There are various strategies that are used to make servers performant. Node has chosen an architecture that performs very well but also reduces the complexity for the application developer. This is an extremely important feature. Programming concurrency is hard and fraught with danger. Node sidesteps this challenge while still offering impressive performance. As always, any approach still has trade-offs...

So Node.js is definitely not a silver-bullet, but it does offer some advantages that other environments may not have. Without getting too bogged down in the technical details, just know that Node.js is just something you can use to run your website like any other piece of software you might choose for various reasons. Hopefully, this will become a bit clearer as we move along.

Installing Node.js

To install Node.js on your machine it might be a bit different depending on what platform you are using. Head on over to the [Node.js website](#) to follow the instructions there for your operating system (Linux, Mac, Windows). One thing I'd definitely recommend: you'd definitely want to have Node.js available to be run from the command line globally so that you can run Node from the command-line anywhere. In Windows, this means adding it to your environment PATH (which is one of the options when using the installer).

After you have node installed, do a quick test to see that everything is working. Create a file called "test.js" and write the following code in the file...

```
console.log("Hello! Node.js is running...");
```

Then open your favorite command shell (I like BASH) in the same directory and type the following...

```
$ node test.js
```

If you see the text "Hello! Node.js is running..." displayed, Node.js is installed and you are good to go!

Obligatory "Hello World" in Node.js

Below is the obligatory "Hello World" example in Node.js. Recall that we said that that the interesting thing about Node.js is that we are running JavaScript on the server. Well, below is all the code that you need to create a functioning web server written in JavaScript using Node.js. Create a file called "app.js" and put the following code within it...

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, { 'Content-Type': 'text/plain' });
  response.end('Hello World\n');
}).listen(1337);

console.log('Server running at http://127.0.0.1:1337/');
```

Then in the same directory open a command line and run...

```
$ node app.js
```

Now if you open a browser and type in <http://localhost:1337/> you should see the text "Hello World" appear. Of course, this is not a very interesting server and it only has one route, but it shows how little code in JavaScript is needed to get a server up and running with Node.js.

Installing Modules with npm

In the previous section we loaded Node.js's HTTP module with the following line of code...

```
var http = require('http');
```

This is the standard way that we load a module within Node.js. You aren't just limited to the modules that ship with Node. You can also use a utility called "[npm](#)" that is part of the Node.js installation (so you already have it installed because you have Node installed) to install other modules directly from your client using the command line. That's another thing that people really like about Node.js: the community. There are many different helpful utilities that 3rd party developers have written for Node.js and posted them on [npm](#) where they can be installed and used freely by everybody. Sometimes people call the npm utility "Node Package Manager" or "Node Packaged Modules" though the npm website FAQ insisted that this is not the case. Whatever you call it, it's a useful tool. For example, to install [Express](#) -- a 3rd party Node.js module that you can use to easily create web applications just as you would using something like [Django](#), [CodeIgniter](#), or [Laravel](#) in Python or PHP -- we just have to type the following into our command line...

```
$ npm install express
```

This will download all of the files that express requires and will place them in a folder called "node_modules" in the location you are running your commands from.

Then to use the express module in our Node.js files we just have to include it just as we did with our HTTP module. For example, the "Hello World" in Node.js example above could be rewritten using Express to produce the same result...

```
var app = require('express');

app.get('/', function (request, response) {
  response.send("Hello World");
});

app.listen(1337);
```

Now if you run

```
$ node app
```

and go to <http://localhost:1337/> you should see the text displayed.

If you need to install more than one module at once you can separate them by spaces...

```
$ npm install express socket.io grunt
```

Another way that you can install modules is to create a [package.json](#) file and put it in your project root. Then add the following...

```
{
  "name": "Node Primer",
  "description": "Node Primer by 9bit Studios...",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.x"
  }
}
```

```
}
}
```

Then all you have to do is type the following...

```
$ npm install
```

Node.js will look for the package.json file in the same directory and pull down all the dependencies you have specified and place them all in the node_modules folder!

The first few properties of the package.json file define some metadata, but the most important component is the dependencies. You can add more and more to this object (comma-separated), specify version numbers and by typing the above command to install everything that you need. For example, to add the module mongoose, you could edit the package.json as follows...

```
{
  "name": "Node Primer",
  "description": "Node Primer by 9bit Studios...",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.x",
    "mongoose": "~3.5.5"
  }
}
```

And then just run npm install again. It's that easy!

As mentioned previously, there are lots and lots and lots of modules available on npm. Feel free to browse around to see if you can find anything that might be useful. Some of the more popular modules that people install include [Grunt](#) which is a JavaScript task-runner, socket.io which helps developers utilize web-sockets in JavaScript, and [Mongoose](#) which is a utility that focuses on easily integrating the NoSQL database [MongoDB](#) into your application. More info on Grunt can be found on the [Grunt website](#). There many other modules available on npm for all sorts of different uses. You'll just need to do a search for whatever is in line with your particular needs.

Obviously, we've just scratched the surface on some of the exciting and interesting aspects of Node.js. We've looked installation, installing modules, and how Node.js it can be used to create web applications on the server. That's just the beginning. In upcoming discussions, we'll look at some of the more advanced features of Node and how you can leverage some of the some of the more popular modules to use Node in whatever capacity you need it.

Chapter 2: Introduction to Express.js

Having become acquainted with Node.js as a platform, we can now look at some of the important applications that exist in and around the platform. [Express](#) is one of the more popular solutions for running a web-server and creating web applications using [Node.js](#). Express is a web-application framework just like [Django](#), [CodeIgniter](#), or [Laravel](#). The only difference is that instead of running on something like Apache or Nginx, it runs on Node.js and instead of being written in PHP or Python, it is written in JavaScript. There are other web application frameworks that run on top of Node.js but Express is one of the more popular ones.

Express, like other Node.js installations, comes in the form of a module. You can use a utility called "npm" to install it using the command line. To install Express we just have to type the following into our Node.js command line...

```
$ npm install express
```

This will download all of the files that Express requires (dependencies) and will place them in a folder called "node_modules".

Now that we have Express installed we can get started. Create a file named "app.js" (if you have not already from previous tutorials) and add the following...

```
var express = require('express');
var app = express();

app.get('/', function (request, response) {
  response.send("<h1>Our Express App</h1>");
});

app.listen(1337);
```

Then open up your favorite terminal (like [BASH](#)) and type the following...

```
$ node app
```

If you open a browser and type in <http://localhost:1337/> or <http://127.0.0.1:1337/> you should see the text appear. The great thing about Express is that right out-of-the-box we can use it to define some routes and build an entire web application. If you have used [Laravel](#) or the [Slim PHP Framework](#) or something similar this might look familiar. We can also define routes for the other HTTP verbs: POST, PUT, and DELETE. We can return HTML or we can use it to create an API that returns XML or JSON. The possibilities for what we can do with it are all there.

Let's try adding another route...

```
var express = require('express');
var app = express();

app.get('/', function (request, response) {
  response.send("<h1>Our Express App</h1>");
});

app.get('/about', function (request, response) {
  response.send("<h1>About</h1>");
});

app.listen(1337);
```

If you open a browser and type in "<http://127.0.0.1:1337/about>" you should see our different route rendering different markup! Pretty neat, eh?

Of course, right now our application is not all that interesting. It's just static markup. More importantly, we're definitely *not* going to want to have HTML in our routes because we are violating the principle of [separation of concern](#) by mixing our view logic with our route/controller logic making a nice big plate of spaghetti. To solve this,

we're going to want to use some kind of templating engine so that we can add our markup to an HTML file but also have the file render data that we pass to it. That's what we'll look at next.

Templates & Views

Using templates to render different views in JavaScript involves passing data to a file and then using special syntax to render that data to a client -- usually a browser Window of some kind. We're going to be using Handlebars as our templating engine when we use Express.

To install Handlebars we're going to want to type the following into our terminal (just as we did before when installing Express)...

```
$ npm install hbs
```

And now we can add it as a module...

```
var express = require('express');
var app = express();
var hbs = require('hbs');

app.get('/', function (request, response) {
  response.send("<h1>Our Express App</h1>");
});

app.get('/about', function (request, response) {
  response.send("<h1>About</h1>");
});

app.listen(1337);
```

More info on this Handlebars module can be found [here](#).

We're going to want to add a couple of additional lines and make some changes to the code that will tell Express to use HTML templates that we'll be creating shortly. We're going to be using HTML files -- indicated by the use of the `app.set('view engine', 'html')` line and notice that in our routes instead of using `response.send()` we're now using `response.render()` to render markup using the template...

```
var express = require('express');
var app = express();
var hbs = require('hbs');

app.set('view engine', 'html');
app.engine('html', hbs.__express);

app.get('/', function (request, response) {
  response.render('index');
});

app.get('/about', function (request, response) {
  response.render('about');
});

app.listen(1337);
```

By default, Express looks for templates in a "views" folder, so let's add a "views" directory and create an `index.html` file in it with the following...

```
<html>
<head>
  <title>Express App</title>
</head>

<body>
```

```

    <h1>Our Express App with Templates</h1>

  </body>
</html>

```

We can also create an about.html file and place it in the same "views" folder.

Now if we restart our server and go to the home route, we can see that Express is using this template to render the HTML to the browser. Pretty slick!

Dynamic Templates & Passing Data to Views

Our application is still just rendering static HTML. Let's change that by figuring out if there is a way that we can pass data to our templates.

Edit the index.html file in our views folder to look like the following...

```

<html>
<head>
  <title>{{title}}</title>
</head>

<body>
  <h1>{{title}}</h1>

</body>
</html>

```

Now let's edit our application...

```

var express = require('express');
var app = express();
var hbs = require('hbs');

app.set('view engine', 'html');
app.engine('html', hbs.__express);

app.get('/', function (request, response) {
  var welcome = 'Our Express App with Templates';
  response.render('index', { title: welcome });
});

app.get('/about', function (request, response) {
  response.render('about');
});

app.listen(1337);

```

As you can see here, we are now passing data into our index.html template. The text in the title property of the object we pass in is rendered in the {{title}} section by Handlebars.

Let's do something a bit more complicated. Edit the server file so that it looks like the following.

```

var express = require('express');
var app = express();
var hbs = require('hbs');

app.set('view engine', 'html');
app.engine('html', hbs.__express);

app.get('/', function (request, response) {
  var welcome = 'Our Express App with Templates';

```

```

var products = [
  { "id": 1, "name": "Apple", "price": 4.99 },
  { "id": 2, "name": "Pear", "price": 3.99 },
  { "id": 3, "name": "Orange", "price": 5.99 }
];

response.render('index', { title: welcome, products: products });
});

app.get('/about', function (request, response) {
  response.render('about');
});

app.listen(1337);

```

Here in our home route we are getting an array of JSON objects that we've defined statically. In a real application you'd connect to some database here or get data from a web service to get the objects that you want to pass to your views.

After this, edit the index.html view to look like the following...

```

<html>
<head>
  <title>{{title}}</title>
</head>

<body>
  <h1>{{title}}</h1>

  {{#each products}}
  <p>
    <a href="/product/{{id}}">{{name}} - {{ price }}</a>
  </p>
  {{/each}}

</body>
</html>

```

Now if we go to our home route, we see more dynamic data rendered as output. Here we are using an `{{#each}}` loop to iterate over the list of products that we passed in.

Let's add a route for that. Create a `/products/:id` get route as follows.

```

var express = require('express');
var app = express();
var hbs = require('hbs');

app.set('view engine', 'html');
app.engine('html', hbs.__express);

app.get('/', function (request, response) {

  var welcome = 'Our Express App with Templates';

  var products = [
    { "id": 1, "name": "Apple", "price": 4.99 },
    { "id": 2, "name": "Pear", "price": 3.99 },
    { "id": 3, "name": "Orange", "price": 5.99 }
  ];

  response.render('index', { title: welcome, products: products });
});

app.get('/about', function (request, response) {
  response.render('about');
});

```



```
app.get('/product/:id', function (request, response) {  
  var id = request.params.id;  
  response.render('product', { title: 'Product #' + id });  
});  
  
app.listen(1337);
```

The :id part at the end indicates that it is a dynamic "id" parameter in the URL query string. As you can see, we can fetch that property out of the request by using request.params.id. Now create a product.html file and place it in our views folder...

```
<html>  
<head>  
  <title>{{title}}</title>  
</head>  
  
<body>  
  <h1>{{title}}</h1>  
  
</body>  
</html>
```

Here we are just rendering product #1, #2, etc. In a real application we'd do a name lookup based on the id to get all sorts of information about a specific product for this page and pass it to the view. But it gives you an idea of how you would go about rendering a dynamic page based on the id parameter.

So that concludes our very brief introduction to Express! We have just scratched the surface and there is a lot more to explore. Next steps would include looking at things like POST, PUT, and DELETE routes, database integration, and partial templates... just to name a few. Be sure to check out Expressjs.com for more information including the API reference and documentation.

Chapter 3: Creating an MVC Express.js Application

[Express.js](#) is probably currently the most popular Node.js framework for building web applications and APIs. It has a good history, a good community around it, lots of modules built for it, and a fairly solid reputation.

In what follows we're going to walk through how to build an Express.js application using the [Model View Controller \(MVC\)](#) design pattern. We'll start out with a simple application structure and then we'll move on by adding more complex concepts like authentication, dynamic routing, and reading from and writing to a database.

At the time of this writing Express is at version 4.X. If you are reading this at a later point, it's possible that the latest version differs quite significantly from this. Be sure to check out the Express.js homepage to find information on how to properly set things up using the most current version of Express. However, it may be the case that the steps followed in this tutorial will not be fully compatible with the most recent version of Express.js

To start we're going to want to create a [package.json](#) file. You can read more about package.json [here](#). package.json will define some metadata type things like the name and version of the application, but most importantly, it will specify what [modules](#) you will need to download and run the application. You can read more about modules in Node.js [here](#) as well as in any good introduction to Node.js. Express is itself a Node.js module and it makes use of other Node.js modules. Some modules like "http" are core Node.js modules (part of the Node.js core) and some like express.js are third-party. Core Node.js modules will always be available to your application because it is running on Node.js

So our sample package.json file will look like the following...

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
  }
}
```

One item of note: the name of any given package cannot have any spaces. This is to ensure that installation of packages is easy and consistent across all environments. If people were ever wanting to install our application as a module on npm (if we ever put our application on npm) by running `$ npm install MVC-Express-Application` it would not work if the package was named MVC Express Application. If we ran `$ npm install MVC Express Application`, the CLI would get confused.

Now if we run the following from the directory where our package.json file resides...

```
$ npm install -g http-server
```

all of the dependencies that we have specified will be downloaded and installed into a "node_modules" directory. The great thing about this is that if we need to update the versions of our modules or add new ones, we can just add or edit this package.json file and run "npm install" again. The npm installer will figure out all of the files needed. We are limited to the modules we are installing right now, but we will be adding a lot more to this later.

Application Setup

Next we'll want to create an "app.js" file. This file is the kickoff/entry point of our application when it starts up. This is akin to the main() function you'd find in C++ or Java.

The first components we're going to want to add are here...

```
var express = require('express');
```

This loads up the express module. We can now refer to it in our code.

A lot of the modules we'll be loading up are part of the Express.js project itself. There are many useful items that our application will be utilizing at various times throughout our application. As is described in the [Node.js documentation on modules](#), Node.js modules can be loaded via references to filenames so long as those filenames are not named the same as reserved core Node.js modules, e.g. the "http" module.

We will need to install the http and path modules (which are part of the Node.js core modules) as they will be important for our application.

```
var express = require('express');
var http = require('http');
var path = require('path');
```

We are going to want to use the Express.js [set function](#) to do certain things like set application constant values. We are going to want to set the port that our application will run through. To do this we can set our port as a constant using the set function. We can also use the "path" module in combination with the Express.js [use function](#) to set the directories from which static assets will be served. These include things such as images, CSS files, and front-end JavaScript files (such as jQuery) or anything else that our client-side views will need. So create a directory called "static" inside our main directory and update our app.js file to look like the following...

```
var express = require('express');
var http = require('http');
var path = require('path');
var app = express();

app.set('port', 1337);
app.use(express.static(path.join(__dirname, 'static')));

http.createServer(app).listen(app.get('port'), function () {
  console.log('Express server listening on port ' + app.get('port'));
});
```

Now if we were to run...

```
$ node app
```

We could see our app running. At this point, our app doesn't really do much other than run the server, but we're on our way.

Templating

To get our app to display things in the browser (the "view" part of our Model-View-Controller application), we are going to want to choose a templating engine to render variables that we process in our Express.js application out to views. There are many different templating engines out there. [Jade](#) is a popular templating choice, but I'm going to use [Express Handlebars](#) which is based off of the [Handlebars.js](#) JavaScript templating library.

Templates are a very important part of modern day web applications written in JavaScript. Templating essentially encompasses the visual UI (user interface) components of these applications by rendering different "views" depending on the current state of the application. A "view" can be thought of as a certain type of layout. For example, when you sign in to an application and go to your account settings, you might be looking at a "profile" view. Or, if you were using an e-commerce application and you were browsing the site's inventory, you might be looking at a "products" view. These views are dynamic in nature because they receive data that gets sent to them and then they render that data out in a manner that is meaningful and useful to the user.

There are many different templating libraries that can be used to render views in a JavaScript web application. No one library is necessarily better than another and it is likely that they all have their various strengths and weaknesses. Handlebars is one of the templating libraries that I like very much for its simplicity and intuitive syntax. The Handlebars.js homepage has some very clear straightforward examples that allow you to jump right in to learning how to use it.

Handlebars template setups will look something like the following...

```
<!DOCTYPE html>
<html>
<head>
  <title>Handlebars</title>
  <script src="js/jquery.js" type="text/javascript"></script>
  <script src="js/handlebars.js" type="text/javascript"></script>
</head>
<body>

  <div class="content"></div>

  <script id="artist-list-template" type="text/x-handlebars-template">

    <h1>{{ title }}</h1>

    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Hometown</th>
          <th>Favorite Color</th>
        </tr>
      </thead>
      <tbody>
        {{#each artists}}
          <tr>
            <td>{{ name }}</td>
            <td>{{ hometown }}</td>
            <td>{{ favoriteColor }}</td>
          </tr>
        {{/each}}
      </tbody>
    </table>
  </script>

  </script>

  <script type="text/javascript">
var data = {
  title: 'Artist Table',
  artists: [
    { id: 1, name: 'Notorious BIG', birthday: 'May 21, 1972', hometown: 'Brooklyn, NY', favoriteColor:
'green' },
    { id: 2, name: 'Mike Jones', birthday: 'January 6, 1981', hometown: 'Houston, TX', favoriteColor:
'blue' },
    { id: 3, name: 'Taylor Swift', birthday: 'December 13, 1989', hometown: 'Reading, PA',
favoriteColor: 'red' }
  ]
}

var source = jQuery('#artist-list-template').html();
var template = Handlebars.compile(source);
var html = template(data);
jQuery('.content').html(html);

  </script>
</body>
</html>
```

What we are doing here is passing data (in this case the JSON object in our data into) our artist list template (the one with id: #artist-list-template). Note that with our array of artists inside the {{ #each artists }} block (where we are looping over an array of JSON objects) the context has changed so we can directly refer to the artist name, hometown, and favoriteColor properties in for each item

In this example, we are using a static data object that we have created inline but in a real application it's likely that you'd be getting that data via AJAX requests to APIs somewhere. But whether you get your data from a server or create it locally, the important part is that data can then be passed into the Handlebars template to be displayed by Handlebars. That is where using a templating engine becomes very efficient and very useful. The view/template does not need to worry about handling anything in the data. All of the code to handle getting data and then searching and sorting through it can be handled elsewhere. All your Handlebars template has to worry about is displaying the "final result" that your code generated elsewhere has produced.

There is also support for doing "multi-level" references of you have a JSON object that is x layers deep, you can refer to the properties inside it by normal dot syntax you might be familiar with.

```
<div class="entry">
  <h1>{{title}}</h1>
  <h2>By {{author.name}}</h2>

  <div class="body">
    {{body}}
  </div>
</div>

<script type="text/javascript">
  var data = {
    title: "A Blog Post!",
    author: {
      id: 42,
      name: "Joe Smith"
    },
    body: "This is the text of the blog post"
  };
</script>
```

And there is even the possibility of doing conditionals via the built-in helper methods. For example, here is an if conditional in a Handlebars template where if a property does not exist or is null in an object that is passed to the view, a different "block" of code will be rendered.

```
<div class="entry">
  {{#if author}}
  <h1>{{firstName}} {{lastName}}</h1>
  {{else}}
  <h1>Unknown Author</h1>
  {{/if}}
</div>
```

And this is just scratching the surface. There are many other features and functionality that Handlebars.js provides. So head on over to the [Handlebars.js](http://handlebarsjs.com/) homepage to start taking a look through the examples there to see what can be accomplished.

Getting back to our application, to install the express-handlebars module we are going to want to update our package.json file by adding a new item to the dependencies section...

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "express-handlebars": "1.1.0"
  }
}
```

And we are going to want to run npm install again...

```
$ npm install
```

which will pull down the express-handlebars module.

We will start out by just adding basic templating, but we will likely go back and modify this later on to be a bit more sophisticated. We'll first want to create a "views" directory for our views. So create a new directory and call it "views" and update our code to look like the following...

```
var express = require('express');
var http = require('http');
var path = require('path');
var handlebars = require('express-handlebars'), hbs;
var app = express();

app.set('port', 1337);
app.set('views', path.join(__dirname, 'views'));

/* express-handlebars - https://github.com/ericf/express-handlebars
A Handlebars view engine for Express. */
hbs = handlebars.create({
  defaultLayout: 'main'
});

app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

app.use(express.static(path.join(__dirname, 'static')));

http.createServer(app).listen(app.get('port'), function () {
  console.log('Express server listening on port ' + app.get('port'));
});
```

Notice how we are using `app.set('views', path.join(__dirname, 'views'))`; to tell our handlebars engine to look for views in this directory.

Following the [documentation](#) of the express-handlebars module, we will want to create a layout. A layout is kind of like a master view or parent view that will house all of our different views that we will load up depending on the route. We need all of the common HTML that is essential to any webpage -- such as the `<!DOCTYPE html>` as well as and the `<head>` and `<body>` elements and anything else we might need. Adding these to each and every view definitely would not be most efficient approach and will lead to a lot of repetitive code. What if we ever have to make any changes to our main HTML structure? We would have to change each and every view file. This might not seem so bad if we only have a few views. However, if the size and complexity of our application were to grow and we had dozens or even hundreds of views, things would get out of hand pretty quickly.

Fortunately, there is a better way. With Express Handlebars we can use a "layout" which will serve as a common container for our views. Some more info on using layouts can be found [here](#). So we need to create a directory called "layouts" inside of our views directory. After we do this, we can put a `Main.handlebars` file in our layouts directory that will serve as our layout...

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>MVC Express Application</title>
</head>
<body>

  {{{body}}}

</body>
</html>
```

We're also going to want to create 2 different views that we'll make use of later. Add a directory called "home" in the views directory and add 2 files...

Index.handlebars

```
<p>Welcome to the homepage</p>
```

Other.handlebars

```
<p>Welcome to other. This is a different route using a different view...</p>
```

Don't worry too much about this syntax now. We'll revisit the practical usage of these later.

Router

Once we put all of the items that we want into our app.js file and have our views set. We are going to want to create a router.js file. A lot of MVC application frameworks such as [Laravel](#) (PHP), [ASP.NET MVC](#) (.NET/C#) or [Django](#) (Python) have a router file, where you can set certain routes and tell the application which code to run depending on the route that the user goes to. Express.js routing is discussed [here](#).

So for example, if in our application we wanted to have a list of all books we might have a route that looks like the following...

```
app.get('/books', function (request, response) {
  console.log('List of all books');
});
```

The callback function that we pass into our router will run whenever a user goes to that particular route (/books). Of course, we are going to be a lot more sophisticated with the functions we have running (we are going to call our controllers from here), but this is just a simple illustration to show how routers work.

Note the request and response arguments we are passing into our function. These two items are very important in Express.js and will contain a lot of useful information about the request sent by the client and the response that we are going to use to send data back to the client from our server.

And if we wanted to set a "details view" for each book depending on the book id our route would look like the following...

```
app.get('/books/:id', function (request, response) {
  console.log('Book details from book id ' + request.params.id);
});
```

Now if we were to go to the route "/books/4" we'd go to this route with the id 4. The id value will be available to us on the request object.

Setting up a bunch of different routes is all well and good for GET requests, but we're going to also want to have a place for the client to send data to us -- either via form submissions or AJAX requests or some other means. Fortunately, we can also easily create a POST route to receive data from the client and process it.

```
app.post('/books/add', function (request, response) {
  console.log('Post route to handle the addition of a new book');
});
```

Express.js also has functions for PUT and DELETE requests.

So now that we have a basic overview of routing, let's create a router.js file and put it in the main directory. In this file, add the following...

```
// Routes
```

```

module.exports = function (app) {

  // Main Routes

  app.get('/', function (request, response) {

  });

  app.get('/other', function (request, response) {

  });

};

```

As we can see here we have created 2 routes. One main route: / and one route: /other.

We also have to update our app.js and send the app object to the router. In app.js pass the app object we have been adding things to the router like so...

```

// send app to router
require('./router')(app);

```

We have seen in the router that we can send in a callback and have it be available to us whenever a user navigates to a particular route. It's fine to do things this way, but as the number of routes we create grows, our router.js file is going to get pretty big. So to add a little bit of organization to our application, it's probably a good idea to separate these callback functions out into separate controllers because we are using the Model View Controller (MVC) design pattern to create our application.

Controllers

Let's call our controller "HomeController". Create a directory called controllers and in that directory create a file called HomeController.js In HomeController, add the following...

```

exports.Index = function (request, response) {
  response.render('home/Index');
};

exports.Other = function (request, response) {
  response.render('home/Other');
};

```

We have added 2 controllers, one for each route in our router file. Notice the response.render function and what we are passing into it. Here we are telling express to use the views in the views/home directory and use the particular view found in there. So for the / route we are using the "Index.handlebars" view and for the /other route we are using the "Other.handlebars" view Both of these will use the Main.handlebars file as a layout, because this is what we specified in our app.js file.

So let's update the router.js file to use our newly created controllers. Note that we need to add a reference to our controller at the beginning of our file.

```

var HomeController = require('./controllers/HomeController');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

};

```


Recall that we had created 2 different views earlier `Index.handlebars` and `Other.handlebars`. Now if we were to run our app with `$ node app` and if we navigate to `localhost:1337` and `localhost:1337/other` we can see the different views being used for our different routes.

Passing Data to Views

Rendering different views in response to different routes is a great start. However, it's kind of bland if we are only rendering static content in each view. Let's figure out how we can pass data to our views. Update the 2 views to the following...

`Index.handlebars`

```
<p>Welcome to the homepage of {{ title }}...</p>
```

`Other.handlebars`

```
<p>Welcome to {{ title }}. This is a different route using a different view...</p>
```

And let's also update our `HomeController.js` file as well and attach the "title" property to the response object and pass that object to our view.

```
exports.Index = function (request, response) {
  response.title = 'Hello World';
  response.render('home/Index', response);
};

exports.Other = function (request, response) {
  response.title = 'Other';
  response.render('home/Other', response);
};
```

Notice how we are passing the dynamic data in "title" to the view by attaching it to the response object and we can see that this data is rendered out to our views using the `{{ title }}` syntax. We can name these properties anything we want (we could have just as easily written `response.cheese`), the names just have to match in our views e.g `{{ cheese }}`.

Now if we were to run our app and navigate to `localhost:1337` and `localhost:1337/other` we can see the different views being used rendering our data

These properties are also available to our layouts files as well. So if we wanted to we could update our `Main.handlebars` file to display the title in the `<title>` tag found in the `<head>` section of our HTML document...

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{title}}</title>
</head>
<body>

  {{{body}}}

</body>
</html>
```

It is worth noting that it is probably not the best idea to add individual properties directly onto the response object (which could get pretty messy and unmaintainable pretty quick) so let's pull all of our data into a single object that we'll attach to the response. We'll call this object "pageInfo." So let's update our controller to look like the following...

```
exports.Index = function (request, response) {
```

```

    response.pageInfo.title = 'Hello World';
    response.render('home/Index', response.pageInfo);
  });

exports.Other = function (request, response) {
  response.pageInfo.title = 'Other';
  response.render('home/Other', response.pageInfo);
};

```

Fortunately, we don't have to do anything as far as updating our views go. This is one of the nice things about separating different areas of functionality out into different places.

Obviously this is just a simple example. In a more complex application it would be here in our controllers where we might use a model object to connect to a database and get the data back from there. But that's another tutorial for another day.

Partial Views

The express-handlebars module also supports "partial views." What is a partial view? A partial view is merely part of a view within another view. That is, it's a subset of HTML markup within a larger container layout. So an example of this might be a menu partial view with a list of links like "Home," "About Us," "Contact," and others of the sort. This menu partial view can be referenced (included) in a larger view containing the <html> and <body> elements along with any other outer container <div> elements.

You may have seen the concept of partial views utilized in various way with other web languages. For example, you might see something like within another PHP file to include a menu. ASP.NET web forms has user control .ascx files and ASP.NET MVC has it's own implementation of partial views as well. They're pretty common

So to do a partial with Handlebars create a folder in the views directory called "partials" and make a file called Menu.handlebars with the following in it...

```

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/about">About</a></li>
  <li><a href="/contact">Contact</a></li>
</ul>

```

And you can include a partial view within your layout by using the {{> Menu}} syntax (with the right angle bracket). So in our Main.handlebars ;layout we can have the following...

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{title}}</title>
</head>
<body>

  {{> Menu}}

  {{{body}}}
</body>
</html>

```

We just need to make sure that we use the correct name of our partial view. They need to match.

Like with layouts, partial views are also able to handle data passed to them from the controller code as well, so we could really put whatever we want in them. Can't complain about that!

Summary

We have covered a lot in this tutorial already so this is probably a good place to stop for now. We have learned how to organize a simple Express.js application into an MVC type architecture, and create a router to handle routes, controllers to process data when the user navigates to a particular route, and views to render the items we want when we pass that data to the view. In the future, we will discuss more complex components of creating an Express.js application including middleware, reading and writing data to and from a database (the model portion of MVC) and abstracting more components of an application into configuration files. Feel free to download the files below and play around with them. Don't forget to run

```
$ npm install
```

to install the modules set in the package.json file.

Chapter 4: Middleware & Configuration

Now that we've covered the basics and we have a good solid foundation for structuring an Express.js application, we can move on to some of the additional important components of building within this framework. We'll pick up where we left off in what follows. The focal point of what we'll explore in this section will center around middleware and configuration.

Middleware

One of the things that we are going to want to implement in our application is middleware. There is a pretty good list of Express.js middleware in the [Express.js project](#). Middleware is an important component of Express.js applications. You could probably think of a middleware function or method as a function that runs on every request/response cycle at whatever point you want to specify. For example, if you wanted to see if the current user who is making the request is authenticated on every single request, you might have this authentication check run in a middleware function before issuing the response. That's just one example, but there are many other pieces of functionality that you can run within middleware functions.

So let's add some of the common Express.js middlewares. One useful middleware utility that we can use is a logging component. This will give us information about all of the requests made as our application runs. Express.js uses something called [Morgan](#) to accomplish this.

To add middleware, we're going to want to add it to our package.json file as a dependency. So let's update our package.json file to look like the following

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "express-handlebars": "1.1.0",
    "morgan": "1.1.1",
  }
}
```

Don't forget that we have to install our new module as well. Fortunately, Node.js and npm make this easy. All we have to do is run `$ npm install` again and all of our packages will be updated...

```
$ npm install
```

Now that our dependency is installed, we need to add a reference to it in our main application file...

```
var logger = require('morgan');
```

Let's add the following lines of code to our main app.js file...

```
/* Morgan - https://github.com/expressjs/morgan
   HTTP request logger middleware for node.js */
app.use(logger({ format: 'dev', immediate: true }));
```

Now if we run our application and look at our CLI window we can see all the requests being output on every request response cycle. And just like that, we have made use of our first middleware.

We will also want a utility to log any errors that occur while we are doing development. So let's add an error handling utility to our package.json file...

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
```

```

    "dependencies": {
      "express": "4.4.4",
      "express-handlebars": "1.1.0",
      "morgan": "1.1.1",
      "errorhandler": "1.1.1"
    }
  }
}

```

After we run `$npm install` again to install the error handling module, we again need to add a reference to it...

```
var errorHandler = require('errorhandler');
```

Now we can also add the following code to our main application file as well...

```

/* errorHandler - https://github.com/expressjs/errorhandler
 Show errors in development. */
app.use(errorHandler({ dumpExceptions: true, showStack: true }));

```

We'd probably want to comment this out if we were ever to move our application into a production environment, but for development this is a very handy thing to have for debugging because it can tell us where, when, and how any errors in our application are occurring.

We are not just limited to using middleware that Express.js provides. We can also create our own as well. Let's add a directory called "utilities" to our project and put a `Middleware.js` file in this folder.

We will want to make sure that we include a reference to this file in our main application file.

```
var Middleware = require('./utilities/Middleware');
```

What can we add to our `Middleware.js` file? Well, whatever we want really. For example, we can add something that will check to see if there is a set value in the query string and tell our application to show a notification depending what is set. Notifications such as "Profile updated successfully" or "There was an error uploading the file" or other relevant messages are a very important part of a good user-experience in any application. So our application is going to need these as well and using a middleware function to handle this aspect of functionality seems like a decent approach.

Recall that previously we had appended a "pageInfo" object to the response pass to our views. We will utilize this same approach here and add a "notifications" property to this object. Because the middleware runs **before** the controllers that handle our routes, we can initialize the pageInfo property here instead...

```

exports.AppendNotifications = function (request, response, next) {

  response.pageInfo = {};
  response.pageInfo.notifications = {};

  if (request.param('success')) {
    response.pageInfo.notifications.success = "Success!"
  }
  else if (request.param('error')) {
    response.pageInfo.notifications.error = "Sorry, an error occurred"
  }

  next();
};

```

Notice the `next()`; function at the end of our custom middleware function. This is a common function utilized in middleware implementations. Basically all it does is say "go to the next middleware function, if any." If there are no more middleware functions to run, execution of the application will just continue on as normal. This way, you can group middleware functions together to run sequentially in the order that you want to specify.

We can also add additional properties on this object e.g. like one to store a message with specific information about the event that occurred. So we'll add a message property to the notifications object...

```
exports.AppendNotifications = function (request, response, next) {

  response.pageInfo = {};
  response.pageInfo.notifications = {};
  response.pageInfo.notifications.message = '';

  if (request.param('message')) {
    response.pageInfo.notifications.message = request.param('message');
  }

  if (request.param('success')) {
    response.pageInfo.notifications.success = "Success!"
  }
  else if (request.param('error')) {
    response.pageInfo.notifications.error = "Error!"
  }

  next();

};
```

Now we can add our middleware to our main application file, just as we did with the Express.js proved middleware...

```
app.use(Middleware.AppendNotifications);
```

Now that we have our middleware in place we'll need to update our views to be able to show the notifications. Fortunately, we can add something to show these notifications in our Main.handlebars layout...

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">
  <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">

  <title>{{ title }}</title>
</head>
<body>

  <div class="notifications">

    {{#if notifications.success}}
    <div class="success">
      {{ notifications.success }} {{ notifications.message }}
    </div>
    {{/if}}

    {{#if notifications.error}}
    <div class="error">
      {{ notifications.error }} {{ notifications.message }}
    </div>
    {{/if}}

  </div>

  {{{body}}}
</body>
```

```
</html>
```

This uses the conditional `{{#if }}` syntax that the Handlebars.js templating engine provides. This is one of the built-in helpers of Handlebars. You can read more about conditionals and other helpers [here](#).

Now if you navigate to one of our routes and add the query string to the end e.g. `http://localhost:1337/?success=true` or `http://localhost:1337/other?error=true` we can see the notification appear. We can also add a message parameter as well e.g. `http://localhost:1337/?success=true&message=User added to database`. In the normal flow of application, we would usually do a redirect with these parameters after performing some action depending on the outcome but we don't really have that functionality in our application yet so to see the notifications we just have to add them end of our URL.

We will use middleware more extensively as the development of our application matures. But for now this illustration shows how middleware can be implemented in useful ways.

Now that we've added a spot for notifications, it might be a good idea to add a little bit of style to those notifications. Recall earlier that in our `app.js` file we had added the following line...

```
app.use(express.static(path.join(__dirname, 'static')));
```

This was, if you recall, a way set a directory to serve up some static files from. In this case, we'd want to serve up a CSS file to add a bit of styling in our app. So in this "static" directory create a "css" folder and create a `style.css` file in the "css" folder. In this file add the following...

```
.notifications .success {
  padding: 5px;
  border: 1px solid #090;
  color: #090;
  background-color: #9fc;
}

.notifications .error {
  padding: 5px;
  border: 1px solid #900;
  color: #900;
  background-color: #fc9;
}
```

This will add some styling to our notifications. Of course we'll also have to update our `Main.handlebars` layout with a reference to this CSS file in the `<head>` section...

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">
  <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">
  <link href="/css/style.css" rel="stylesheet">

  <title>{{ title }}</title>
</head>
<body>

  <div class="notifications">

    {{#if notifications.success}}
    <div class="success">
      {{ notifications.success }} {{ notifications.message }}
    </div>
  </div>
</body>
```

```

    </div>
    {{/if}}

    {{#if notifications.error}}
    <div class="error">
        {{ notifications.error }} {{ notifications.message }}
    </div>
    {{/if}}

</div>

{{{body}}}

</body>
</html>

```

Now if you add the success or error and/or message query string(s) to the end of any of our routes you will see the same behavior as before only this time with some slick styling for our notifications! This static directory can be used to serve up any other file type that we might need (JavaScript files, images, etc). We just need to add the correct path reference in the layouts or our individual views.

This configuration file is actually a great candidate to be a partial view. Partial views were discussed earlier on. The way that we would do this is to create a file called Notifications.handlebars and put it in our "partials" directory in our "views" folder. Notifications.handlebars just needs to contain our notifications <div> block...

```

<div class="notifications">

    {{#if notifications.success}}
    <div class="success">
        {{ notifications.success }} {{ notifications.message }}
    </div>
    {{/if}}

    {{#if notifications.error}}
    <div class="error">
        {{ notifications.error }} {{ notifications.message }}
    </div>
    {{/if}}

</div>

```

And then we can update our Main.handlebars file to include our partial view...

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">
    <link href="/css/style.css" rel="stylesheet">

    <title>{{ title }}</title>

</head>

<body>

    {{> Notifications }}

    {{{body}}}

</body>
</html>

```


Now we'll shift gears and talk about configuration, which is another component that is important to any well-structured application.

Configuration

If you have used any framework in any language — such as [Laravel](#) (PHP) or [Django](#) (Python) — you know that it is often a good idea to have a configuration file where you can store application variables that will remain constant throughout the application. These sorts of settings include things like database names, connection strings, root URLs, ports, or anything else that you want to utilize. It will also give you a place where you can easily make changes to settings when moving from one environment to another, e.g. when moving from your development environment to production.

So we'll want to create a `config.js` file for our application as well. Create a `config.js` file and place it in the main top-level project folder. In `config.js` place the following code...

```
var config = {};

config.development = {
  application: {
    port: 1337
  }
};

config.production = {
  application: {
    port: 80
  }
};

config.environment = 'development';

module.exports = config;
```

And we'll need to make sure that we include a reference to our config file in our main application file.

```
var config = require('./config');
```

So, for example, if we do something like changing the following line in our main application file from...

```
app.set('port', 1337);
```

to

```
app.set('port', config[config.environment].application.port);
```

we are making use of our configuration file because the value of the port we have set will be looked up in the config file. Now if we ever want to push our application from development to staging to production, the only thing we need to edit is the `config.js` file because, we have a way to quickly and easily set our "environment". So if we change that string value from "development" to "production," all of the variables within the object will be switched over across our application. The rest of the application will remain unchanged.

The great thing about this too is that as we add more features and functionality to our application, we can store the needed values for these in our `config.js` file. This way our application will scale well as it grows and progresses on. Other values important in an application such as database connection strings, credentials, SMTP servers, and strings used for creating hashes and tokens can all be stored in a configuration file. At the end of it all, our `config.js` file might end up looking more like the following...

```

var config = {};

config.development = {
  database: {
    name: 'MVCAApplication',
    host: 'localhost',
    port: '27017',
    credentials: '' // username:password@
  },
  smtp: {
    username: "username",
    password: "password",
    host: "smtp.gmail.com",
    port: 587,
    ssl: false
  },
  application: {
    port: 1337,
    cookieKey: '8YQM5GUAtLAT34'
  }
};

config.production = {
  database: {
    name: 'proddb',
    host: 'localhost',
    port: '8080',
    credentials: 'admin:password@' // username:password@
  },
  smtp: {
    username: "username",
    password: "password",
    host: "smtp.yourmailserver.com",
    port: 25,
    ssl: false
  },
  application: {
    port: 80,
    cookieKey: '5SCjWfsTW8ySul'
  }
};

config.environment = 'development';

module.exports = config;

```

This is one way to do configuration. Keep in mind that it's not the only way and there are many other ways you could implement a configuration component in your application. It will probably often come down to a matter of personal preference. The important thing is that you do it in some manner or another to give your application a greater degree of flexibility as you move between environments for development, test, and production. You'll be glad that you did and you'll thank yourself later.

Summary

Overall, in this section we looked at 2 important aspects of creating an Express.js application: middleware and configuration. In upcoming explorations, we'll take a look at some more advanced templating with our views as well as data access and authentication. But for now, it's good to digest this information in small portions. So feel free to download the files below and play around with application configuration and middleware. Remember to run...

```
$ npm install
```

after you download to install dependencies.

Chapter 5: Debugging Node.js Applications

Now that we have gotten a measure of familiarity with creating a Node.js application, it is worth taking a step-back to talk about an essential component to successfully developing within the Node.js ecosystem. I am talking, of course, about debugging. Debugging is a very important part any sort of application development. If you have errors or unexpected results occurring in your code you need to be able to step through your code and find where things are going awry. Thus it is very important that you are able to see what is happening at specific times when your code is executing. There are some IDEs (integrated development environments like Visual Studio or Eclipse) where debugging for the native application language like C# or Java is all built right into the tools and you don't have to do anything to set it up. At the time of this writing in the later part of the year 2015, there are a few IDEs such as WebStorm and Komodo IDE that natively support Node.js debugging. However, it should be noted that Node.js, being a younger and relatively more immature platform sometimes does not have debugging for it natively implemented in a number of IDEs yet. Fortunately, there are often plugins available that will enable Node.js debugging within the IDE. [Node.js tools for Visual Studio 2012 and 2013](#) is one example. [Nodeclipse](#) is a plugin that will enable Node.js debugging capabilities in [Eclipse](#). I would also *highly recommend* [Visual Studio Code](#) which is a fantastic open source editor produced by Microsoft that gives you Node.js debugging right out of the box. It is in the same family of quick, lightweight, and modernized code editors such as GitHub's [Atom](#) and [SublimeText](#).

Because there are so many different development environments and workflows that different developers have different preferences for we won't look at debugging Node.js applications in a specific IDE. But we will look at a specific debugger called [Node Inspector](#). There are other Node.js debuggers out there and if you want to use another debugger that is fine. You would just have to look at the docs for that particular debugger. It should be noted that Node Inspector works in Chrome and Opera only. You have to re-open the inspector page in one of those browsers if another browser is your default web browser (e.g. Safari or Internet Explorer). This is another indicator that shows widespread support for many things in and within Node.js is not entirely quite there just yet. So much of the Node.js module ecosystem is community driven, which has very noticeable pros and cons. The upside to this it is that there are a lot of awesome components of functionality that you can leverage via installation with a simple `$ npm install` of a module. The downside of this environment is that support and/or consistency for bug fixes and releases can vary quite a bit depending on whose module(s) you are using. Just as a personal opinion, I find that on the whole the positives outweigh the negatives when it comes to open source software. I would much rather take this scenario over, say, a behemoth corporation owning and managing all of the releases that might seem more "professional" in its support and maintenance (as opposed to hobby/side-project code). But developing in and for open source applications is definitely far from perfect.

But all that aside, let's get back to fun with Node.js debugging. Node Inspector works almost exactly as the Google Chrome Developer Tools. If you are not entirely familiar with Google Chrome's developer tools read the [DevTools overview](#) to get started. Dev Tools can be used to set breakpoints in your application that halt the execution of the code when a certain statement (or statements) are reached. From there you can examine the state of particular objects to see what values they contain at that point in time. You can then step through your code moving from one statement to the next to see how the values change. If this all seems a little bit confusing at this point, not to worry. We will revisit this a bit later when we actually take on the task of debugging our application. But first we need to install the stuff we need to get debugging up and going.

Installing Node Inspector

To install Node inspector we will use the npm utility to install [Node Inspector from npm](#)

```
$ sudo npm install -g node-inspector
```

Note: Windows 8.1 Users: At the time of this writing in the later part of 2015 for Windows 8.1 I had to omit installing an optional dependency which apparently breaks the install using npm. The way that you do this is by setting the `--no-optional` flag...

```
$ npm install -g node-inspector --no-optional
```

That should get it working for you. To check that it installed correctly you can always run...

```
$ node-debug --version
```

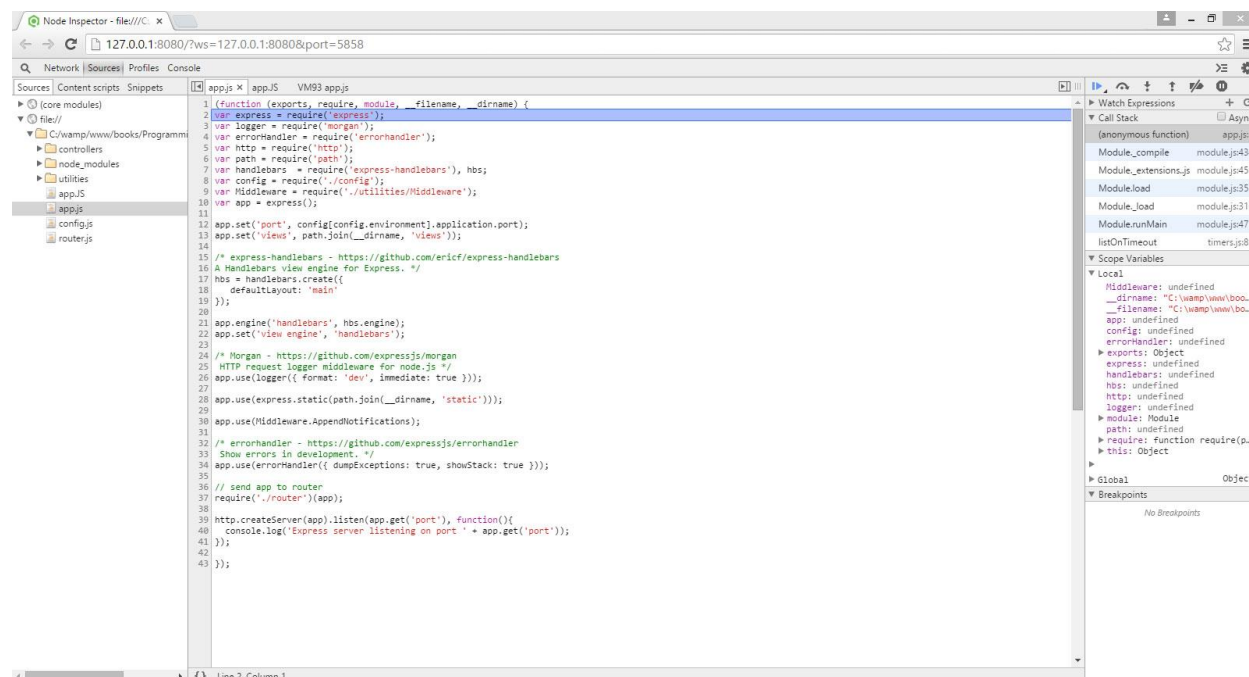
which should output the version number for you without any error messages.

Debugging With Node Inspector

To debug our application with Node Inspector, instead of running our application with the "node" command as we did above we can run it with node-debug instead...

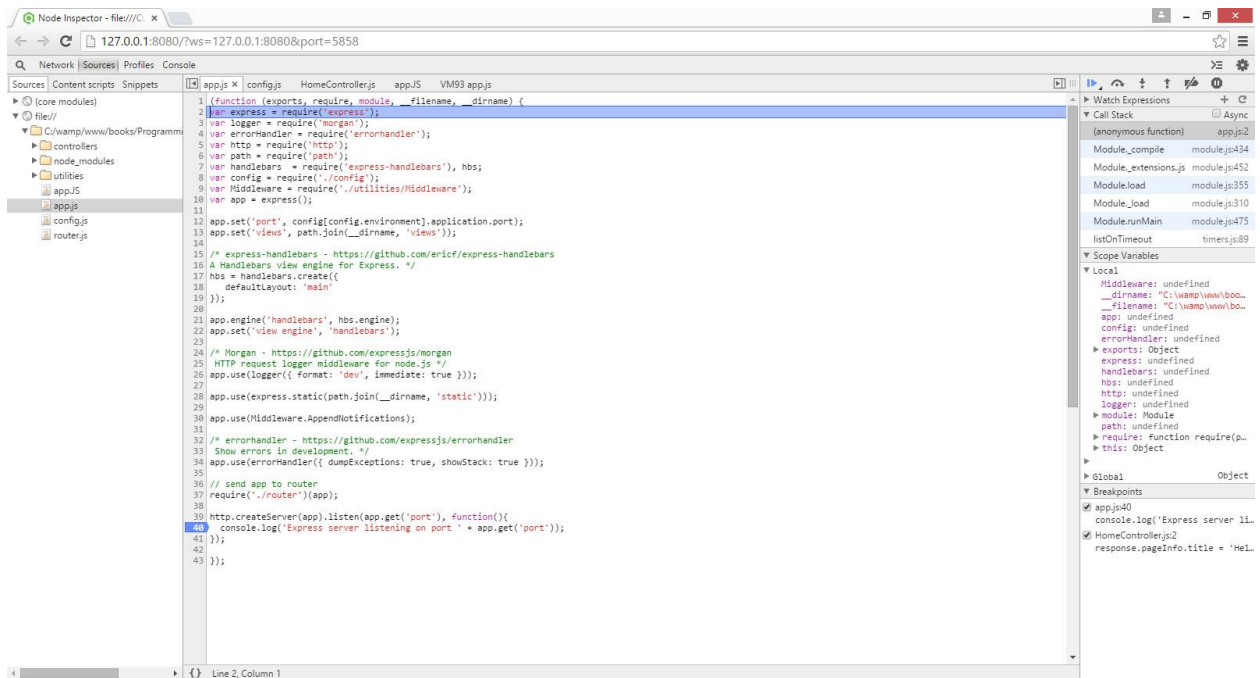
```
$ node-debug app
```

This will open up a tab in your browser. As mentioned previously it will look a lot like Google Chrome's dev tools because it uses a lot of the same structure from dev tools. By default the tools should stop the execution of the code at the first statement of the application as is shown.

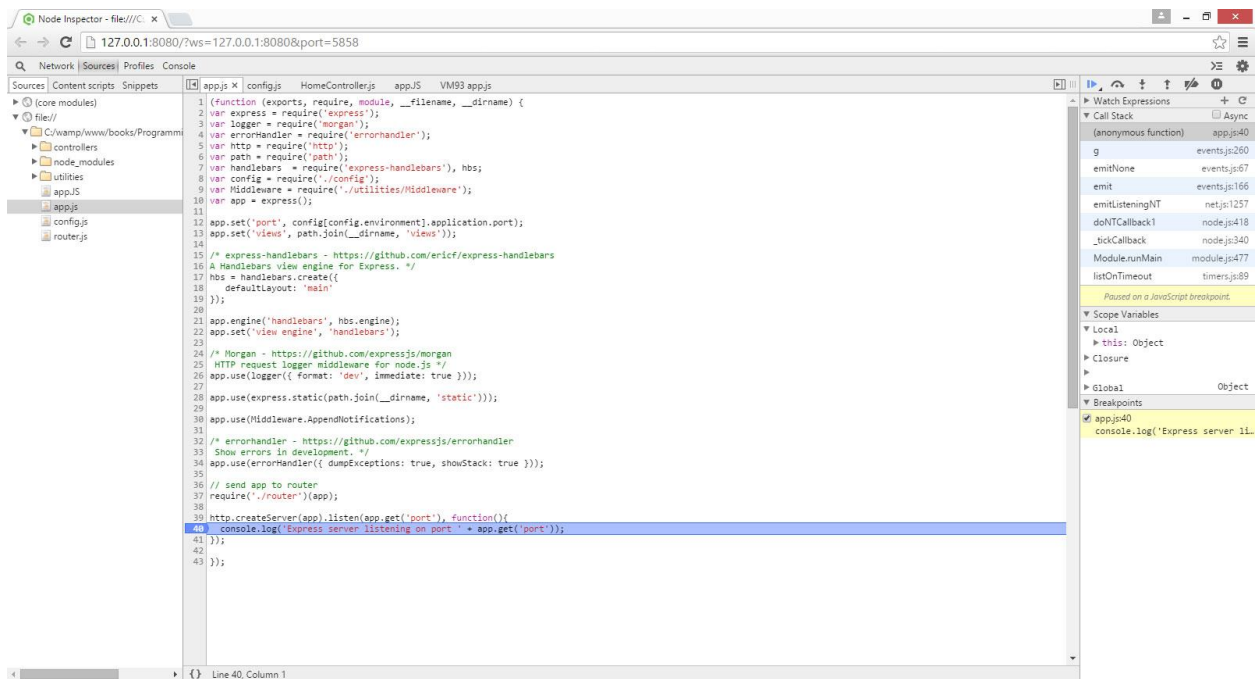


You can see all of the files that are present in our application (as well as the modules we have installed in the "node_modules" folder) on the left.

To set some breakpoints you can click on the line numbers in the code like as is shown in the following screenshot. For example you can click on the line-number on the "console.log('Express server listening on port ' + app.get('port'));" statement. As shown below...



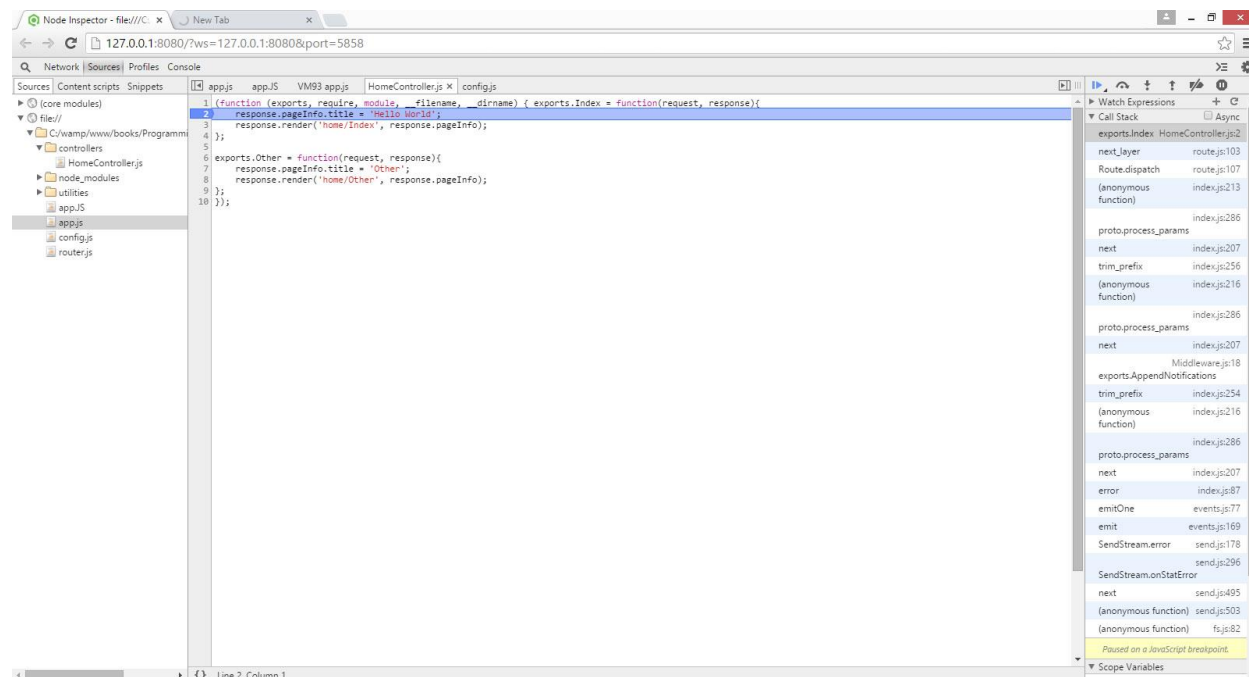
As things exist in their current state script, execution is currently paused. That is, the JavaScript interpreter that steps through code line-by-line is currently halted on the first line of our application. Up in the top-right of this window you can see a number of different buttons. One button, that sort of looks like a "Play" button is the "Resume script execution" button. Clicking on this button will continue running the application and executing all of the statements and functions within it until the next breakpoint is hit. In this case, all the rest of the lines in this initial `app.js` file will execute and we should hit our breakpoint down inside the `http.createServer` method. As is shown below...



So in this case all of the code above this line has run and we are now paused inside of our callback function. So, all of our middleware has run, all of our routes have been created, and our server has been started up.

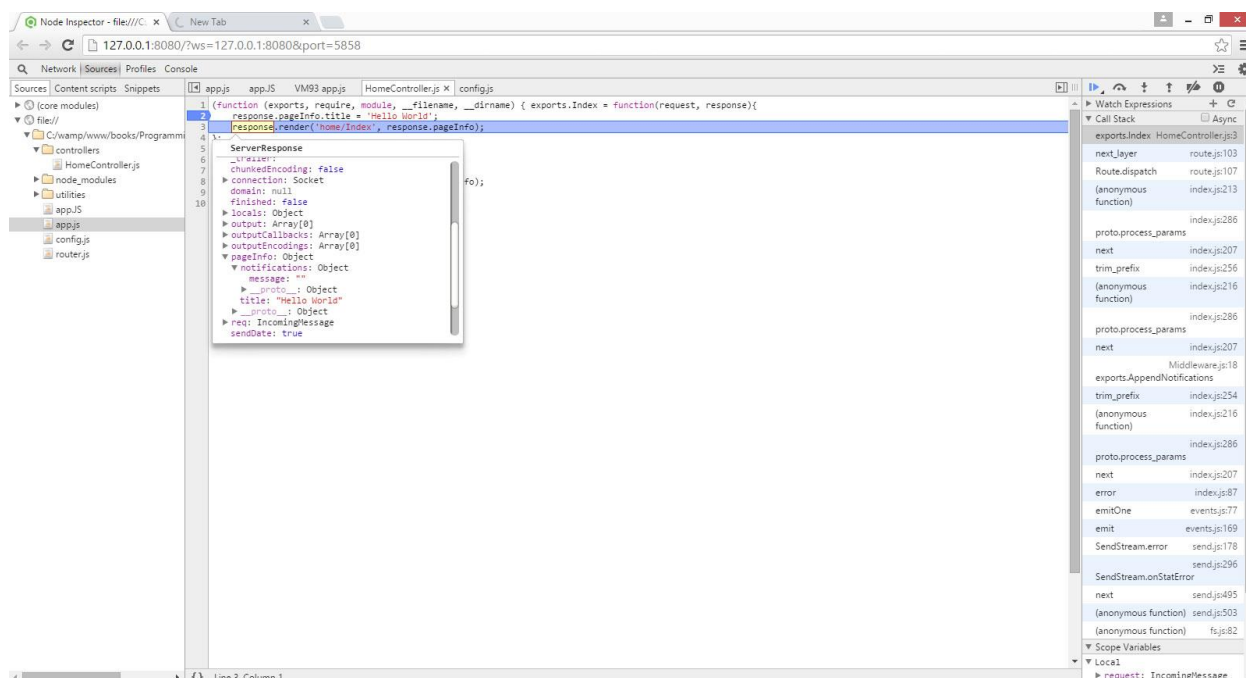
Next, try opening the "controllers" folder and opening up the HomeController.js file placing a breakpoint on the first line of the "Index" callback function where we set the page title.

Now we can open a new tab and go `http://localhost:1337`. If we do this you might notice that the browser sort of hangs and does not do anything. This is because our script execution is paused. So go back inside the Node Inspector screen and click the "Resume script execution" (Play) button. Because we are trying to go into our home route in our browser, you should run into the breakpoint we have set inside of our Index callback function in HomeController.js.



Another button you can see in the top right (next to the "Resume script execution" button) sort of looks like a curved arrow. This is the "Step over next function call" button. What this will do is essentially move to the next statement. From script execution being paused inside of the "Index" method, click on this button. You will see the highlighted line move down to the next line and pause there.

One of the most useful features of debugging when stepping through code like this is the ability to hover over variables and objects when the code is in this paused suspended state. Try hovering your cursor over the response object. As you can see, the properties of that object are displayed in a nice little pop-up window...



We can see our `pageInfo` property with the "Hello World" value set for the "title." We also have our `notifications` property that was added by our middleware, but because we have not set a parameter for this in our query string, the value is currently an empty string.

The most important take-away from all of this is that this is the essence of how you debug applications. You step through code line-by-line and examine certain objects to see how the values on that object change as the code progresses through its execution. If you, say, were seeing a page in one of our views that had some really strange and unexpected values showing up, you would set some breakpoints in the controller that returned that view to see where and when things were going sideways in your code. In doing this, you can narrow down the problematic lines of code in your application and apply a fix so that you get the results that you expect. It is a very sequential, logical, and understandable process to approaching errors that you encounter. As applications grow in size and complexity, having a good handle on how to debug your code becomes all the more essential.

Summary

In the preceding discussion, we have only looked at a few initial aspects of debugging Node.js applications, but in reality there are many more features of debugging that you can make use of within these tools. There is a lot of great information within the developer tools documentation that discuss all of the aspects of debugging that you can leverage in detail. It is definitely an aspect of development in which it is very worthwhile to be expanding your knowledge in on a continual and ongoing basis. This guide has only just scratched the surface. You can set watch expressions and utilize the "step into" and "step out of" function call buttons as well. These allow you to drill down even deeper into when certain functions are being called and when certain statements are executed. These are especially handy when debugging the various middleware functions that run before you even get to the code that handles your routes. You can even step through the third-party modules that you are including within your application with the "require" function. Who knows? You might even find a bug in one of the modules that you can report to the developer/maintainer of that module. If you are especially ambitious, you could actually take on fixing the bug yourself and submitting a pull request to the main repository of that module. This is one of the great benefits of community-driven open-source code. Lots of people testing (and even fixing) code to progressively making it better and more robust as time goes on.

All in all, as your application gets more complex debugging can be a powerful tool to track down errors that occur in Node.js applications. Hopefully, this simple guide has helped to get you on your way. So venture onwards and happy debugging!

Chapter 6: Data Access & Validation With MongoDB

In previous installments we looked at getting started with creating an Express.js MVC application, creating controllers and views. And then we looked at middleware and creating a config file to store application constants.

Now we will look at something that truly makes an Express.js application dynamic (and fun): data-access... which essentially boils down to the use of a database to store, retrieve, and manipulate information.

Like many things in technology, people will always argue about what is the "best" way to do something and what the best technologies to use for doing that something are. And the way in which an application should store data is not going to be excluded from this discussion (read: Internet flame-war). It is not really part of our purposes to argue what is "best" or "better" when it comes to data-access, but it is probably worth pointing out that a lot of the Express.js and Node.js communities seem to have embraced NoSQL databases such as MongoDB, and CouchDB. There is even a full "stack" you will hear mentioned known as the MEAN stack (MongoDB, Express, AngularJS, and Node.js) -- just like you'd hear of a LAMP stack (Linux, Apache, MySQL, and PHP). The inclusion of MongoDB in this shows its prominence in developer preference in Express and Node applications.

So, it is probably worth knowing a bit about how to use a MongoDB database in a Node.js and Express.js application because you are more than likely to come across it at some point in your life as a developer. So, resultantly, we'll utilize MongoDB in our application.

Head on over to [MongoDBs home page](#) and download MongoDB for your OS. You'll probably just want to pick the standard install and won't need any fancy cloud services at this point. Be sure to read through the installation section in the [docs section](#) of the site and do the subsequent tutorials to make sure you can test out that MongoDB was installed correctly. There is a ["Getting Started" section](#) that should walk you through this where you can try out creating, reading, inserting and deleting data a sample MongoDB database.

After you have installed MongoDB create a new folder in the root of your application can call it "db". This is where we are going to store our database and we're going to point MongoDB to put all the files for this database in here.

To start things up, next go to the directory where you installed Mongo and navigate into the bin directory and open a command window. So on Windows, for example, if you installed in the C:\mongodb directory, you would open a command window or shell in C:\mongodb\bin. You could also get there by opening a command window anywhere and typing

```
> cd "C:\mongodb\bin"
```

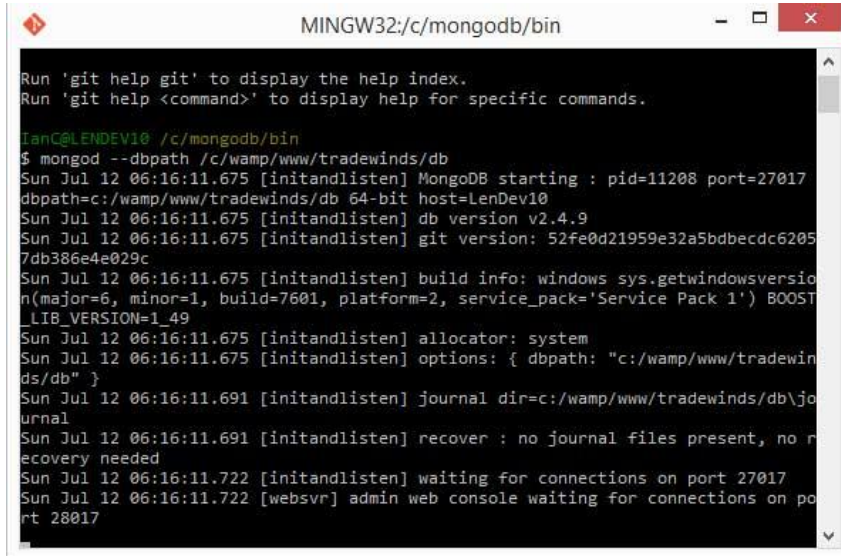
Then you would type the following into the command window where we should specify the path to the "db" folder we just created. So wherever your application lives on your system you'll want to specify that path by running the following command.

```
> mongod --dbpath "C:\path\to\application\db"
```

or if you're using [BASH](#), something like this...

```
$ mongod --dbpath /c/path/to/application/db
```

If all goes well, you should see a message saying that MongoDB is waiting for connections. By default at the time of this writing, the version of MongoDB being used waits for connections on port 28017... so you should see that reflected somewhere in the messaging. Leave this running by leaving this window open (you can minimize if you like).



```

MINGW32:/c/mongodb/bin

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

IlanC@LENDEV10 /c/mongodb/bin
$ mongod --dbpath /c/wamp/www/tradewinds/db
Sun Jul 12 06:16:11.675 [initandlisten] MongoDB starting : pid=11208 port=27017
dbpath=c:/wamp/www/tradewinds/db 64-bit host=LenDev10
Sun Jul 12 06:16:11.675 [initandlisten] db version v2.4.9
Sun Jul 12 06:16:11.675 [initandlisten] git version: 52fe0d21959e32a5bdbecdc6205
7db386e4e029c
Sun Jul 12 06:16:11.675 [initandlisten] build info: windows sys.getwindowsversion
n(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST
LIB_VERSION=1_49
Sun Jul 12 06:16:11.675 [initandlisten] allocator: system
Sun Jul 12 06:16:11.675 [initandlisten] options: { dbpath: "c:/wamp/www/tradewin
ds/db" }
Sun Jul 12 06:16:11.691 [initandlisten] journal dir=c:/wamp/www/tradewinds/db\jo
urnal
Sun Jul 12 06:16:11.691 [initandlisten] recover : no journal files present, no r
ecover needed
Sun Jul 12 06:16:11.722 [initandlisten] waiting for connections on port 27017
Sun Jul 12 06:16:11.722 [websvr] admin web console waiting for connections on po
rt 28017

```

NOTE: When you first run this command, MongoDB will create all the files it needs in the "db" folder we created, so you should see a number of files pop up in there. Some are quite large.

Now that we have MongoDB installed and we can connect to it, let's write some Node.js code to do some data operations! Woo hoo! Like with so many things we have done before, we will utilise a module to act as a wrapper for our data access with MongoDB. Remember, you could always set things up and write your own code in Node.js for accessing data. Using modules is basically just a way to circumvent all of that grunt work and manual labor involved in those processes. If somebody else has already done the heavy lifting in making a module, might as well make use of it.

Installing Mongoose

I am going to use a popular module for accessing MongoDB on npm called [Mongoose.js](#) (which like pretty much all Node.js modules is [available on npm](#)). There are probably some other modules that you could use, we're just going to go with this one to start. So as we have done before when we are adding a new module, we need to update our package.json with a new reference to Mongoose. We will also add another module called [body-parser](#) which is going to be useful for getting data out of forms so we can save it in our database.

```

{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "body-parser": "1.4.3",
    "express-handlebars": "1.1.0",
    "morgan": "1.1.1",
    "errorhandler": "1.1.1",
    "mongoose": "3.8.8"
  }
}

```

Then again, we should open a shell in our project root directory and run

```
$ npm install
```

which will pull down all the Mongoose and body-parser files.

The body-parser module is middleware (which we discussed earlier), so we need to add it to our app.js file. You can add it along with the other middleware, right after the express.static statement...

```
app.use(express.static(path.join(__dirname, 'static')));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Database Settings in config.js

Now, we'll want to add some initial code to connect our running MongoDB instance. Recall earlier that we created a config.js file. This is where having a config.js file is helpful because we are extending the functionality of our application but we have a handy place to store commonly used components of it. We will need to provide Mongoose a number of different pieces of information (name, host, etc), and we can put that into our config.js file for easy access.

Recall that when we created our config.js file we added a database property to it. We won't have to provide any credentials for our MongoDB connection string when we are doing this development locally. However, if we were to move our app to production it's very possible that we'd need to do this (e.g. if the username was "admin" and the password was "password"). So the development section of this config object might look different than the production section. We are only using the development configuration for this tutorial, but we'll also add a database object to the production section as well only to illustrate the possibility of different configurations in different environments.

```
var config = {};

config.development = {

  database: {
    name: 'MVCAApplication',
    host: 'localhost',
    port: '27017',
    credentials: ''
  },
  application: {
    port: 1337
  }
};

config.production = {

  database: {
    name: 'tradewinds',
    host: 'localhost',
    port: '8080',
    credentials: 'admin:password@' // username:password@
  },
  application: {
    port: 80
  }
};

config.environment = 'development';

module.exports = config;
```

So we've got our database name, port, and host set. We can now create some data models to store in this database.

Creating Data Models

Create another folder in the root directory called "models" and in it create a new JavaScript file called "Models.js". In this file we are going to add the following...

```
var config = require('../config');
var mongoose = require('mongoose');
var connectionString = 'mongodb://' + config[config.environment].database.credentials +
config[config.environment].database.host + ':' + config[config.environment].database.port + '/' +
config[config.environment].database.name;
var db = mongoose.connection;
```

Here we have a reference to our config file where we can use the database information we've set in the config file. We'll also pull in the mongoose module. The db variable will be the object that represents our database connection. One thing that will be handy for us is having some sort of messaging to tell us if we have connected successfully to the database or if there was an error. Mongoose provides a way for us to do this through the db object...

```
db.on('error', function () {
  console.log('There was an error connecting to the database');
});

db.once('open', function () {
  console.log('Successfully connected to database');
});
```

This code will log a message to the console whether we are successful in connecting to the database or if we encounter an error.

Lastly, we're going to want to call the code to actually connect to MongoDB through Mongoose.

```
mongoose.connect(connectionString);
```

Reading through the [Mongoose.js docs](#) you can see how to create a data model through the schema interface that mongoose provides. We'll do something simple like create a Book model definition through the mongoose.Schema syntax and then create an actual model based off of this schema using mongoose.model. Lastly, because our Models.js file is itself a module, we're going to want to export an object containing our models for other parts of our application (such as our Controllers) to use.

```
var Book = new mongoose.Schema({
  title: String,
  author: String,
  isPublished: Boolean
});

var BookModel = mongoose.model('Book', Book);

module.exports = {
  BookModel: BookModel
};
```

We can add as many models as we need for whatever data we want to store by repeating this approach. So if we added a "User" model (e.g. to store account information), our file would look like this...

```
var Book = new mongoose.Schema({
  title: String,
  author: String,
  isPublished: Boolean
});

var User = new mongoose.Schema({
  name: String,
  password: String,
  email: String
```

```
});

var BookModel = mongoose.model('Book', Book);
var UserModel = mongoose.model('User', User);

module.exports = {
  BookModel: BookModel,
  UserModel: UserModel
};
```

You can create whatever models you want with whatever properties you want for whatever data you need to store.

Accessing Data in Controllers and Views

So let's now put in some code to read, write, and delete data from our controllers and views. In our "controllers" folder, create a new controller called BookController.js and add the following code...

```
var Model = require('../models/Models');

exports.Index = function (request, response) {
};

exports.BookAdd = function (request, response) {
};

exports.BookCreate = function (request, response) {
};
```

We'll add code to get and insert data into the database using our data models momentarily, but before we do that, don't forget to update our router.js file with a reference to BookController.js and these new routes.

```
var HomeController = require('./controllers/HomeController');
var BookController = require('./controllers/BookController');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

  // Book Routes

  app.get('/books', BookController.Index);
  app.get('/books/add', BookController.BookAdd);
  app.post('/books/add', BookController.BookCreate);

};
```

Note that here we essentially have 2 add routes. Notice though that one is a GET route and one is a POST route. We'll use the GET route to display the form to add a new book, and we'll use the post route to take in the data entered into that form and add it to the database.

Now back in our BookController.js file, for our main controller in our BookController.js file (the one named Index), we'll want to set things up so that when a user goes to the /books route, we will show all the books in the database. So do do this we'll need add some code to fetch the books from our database and then we'll loop over the results in our view. This is how we'll do this in our controller...

```
var Model = require('../models/Models');

exports.Index = function (request, response) {

  Model.BookModel.find(function (error, result) {
```

```

    if (error) {
      console.log('There was an error');
    } else {
      response.pageInfo.title = 'Books';
      response.pageInfo.books = result;
      response.render('books/Index', response.pageInfo);
    }
  });
});

```

Basically what this code says to do is find all books. We'll be doing the same thing we did in earlier exercises by attaching everything to the pageInfo object to send to the view. Note that the result parameter (in which will be stored a set of items that the Mongoose find query finds in our MongoDB database) will be set to a "books" property on the pageInfo object. We'll then send the pageInfo object to the view.

Before we create the view it's worth noting that if we have an error that occurs with a database operation (or any other operation for that matter) it's a good idea to do a redirect with messaging to the user so he/she knows that an error occurred. So we can make use of the notifications middleware that we implemented earlier and redirect with a message in case an error occurs. So it would be good practice change the above code to..

```

var Model = require('../models/Models');

exports.Index = function (request, response) {

  Model.BookModel.find(function (error, result) {
    if (error) {
      console.log('There was an error');
      response.redirect('/?error=true&message=There was an error getting books from the database');
    } else {
      response.pageInfo.title = 'Books';
      response.pageInfo.books = result;
      response.render('books/Index', response.pageInfo);
    }
  });
});

```

Here we are redirecting to the home route "/" with the query string and message to display an error notifications. If we build our application right, we hope that we'll seldom see this error. But errors do happen and it's important that your application handles things gracefully when they do.

But if we don't get an error in our database operation, we'll be able to pass the data we get back to the view. Speaking of the view, let's create that right now. In our views folder create a new folder called "books" and create an Index.handlebars file in it. Add the following code to this file...

```

<ul>
  {{#books}}
    <li>
      <strong>{{title}}</strong> by: {{ author }}
    </li>
  {{ else }}
    <p>There are no books.</p>
  {{/books}}
</ul>

<a href="/books/add">Add Book</a>

```

Because we passed response.pageInfo directly into the view, the "books" property is directly available to be accessed using Handlebars syntax. So the {{#books}} marker basically says loop over the "books" object that was passed to the view. The "books" object was set to the result we got back from our Mongoose query, which contains a set of book models. Thus, inside the books object both the title and author properties are available because those are part of the book model schema.

Of course, we don't have any books in our database right now so the message stating that there are no books will appear in this case. In Handlebars, the content that follows the `{{ else }}` block will render if the `{{ books }}` set is empty.

So let's add some code to actually add a book to the database. In our `BooksController.js` file, add the code to say we're going to render a form when the user goes to the `"books/add"` route with a GET request (i.e. navigates there in his/her browser).

```
exports.Index = function (request, response) {
  Model.BookModel.find(function (error, result) {
    if (error) {
      console.log('There was an error');
      response.redirect('/?error=true&message=There was an error getting books from the database');
    } else {
      response.pageInfo.title = 'Books';
      response.pageInfo.books = result;
      response.render('books/Index', response.pageInfo);
    }
  });
};

exports.BookAdd = function (request, response) {
  response.pageInfo.title = 'Add a Book';
  response.render('books/BookAdd', response.pageInfo);
};

exports.BookCreate = function (request, response) {
};
```

All we need to do in our `BookAdd` GET route is set the page `"title"` attribute (recall that this is in our `Main.handlebars` layout) and then tell Express.js to render the `"BookAdd"` view — which we will create next. So now create a view called `BookAdd.handlebars` in our `"views/books"` directory. In this file we will add a form.

```
<form method="post" action="/books/add">
  <label>Title</label> <input name="title" type="text" />
  <label>Author</label> <input name="author" type="text" />
  <input type="submit" value="Submit" />
</form>
```

Note that this form, when submitted, will send a POST request to the `BookCreate` method in our controller. Even though the action is set to `"/books/add"`, recall in our `router.js` we added the line:

```
app.post('/books/add', BookController.BookCreate);
```

This means that a POST request sent to the `"/books/add"` endpoint will be handled by the `BookCreate` method in our controller. So even though the route `"/books/add"` is the same as the GET request, it's the type of request that differentiates which controller method will execute.

So in our `BookCreate` controller we'll need to add some code to add a new book model object to our database. So in our `BookCreate` method in `BookController.js` add the following code...

```
exports.BookCreate = function (request, response) {
  var title = request.body.title;
  var author = request.body.author;

  var b = new Model.BookModel({
    title: title,
    author: author,
    isPublished: true
  });
```

```

    });

    b.save(function (error) {
      if (error) {
        console.log('Error');
        response.redirect('/books?error=true&message=There was an error adding the book to the
database');
      } else {
        response.redirect('/books?success=true&message=Book created successfully');
      }
    });
  });
};

```

Here the request object will contain the values submitted with the form on the request.body. The name attributes on each of the text fields get added to the request body. This data is then used to create a new book model object containing these values which can then be saved to the database by calling Mongoose's save method on it. If the save is successful, we will redirect back to the home books route which will now have the book rendering as part of the list. We'll even pass in some parameters which if you recall in an earlier discussion will be acted upon by our notification middleware to give us a "Success" message when we've successfully added a book! If we are for some reason unable to add a book to the database, we will redirect back to the "/books" route with an error message.

Note that we are not really doing any kind of validation of the data submitted to the form. Normally we would want to check for empty fields and/or invalid inputs. For books we can basically allow a user to enter pretty much anything (as long as there is something there) for both title and author. However, if we were, for example, storing a user's email address on a "UserModel," we would want to validate that the e-mail was in valid e-mail format before attempting to add the record to the database. If the e-mail provided by the user was invalid we would want to redirect back to the form with a message informing the user that the e-mail that they had entered was invalid. You would normally do these checks for all data types and fields before you attempted to add anything to the database. This is something that we'll revisit more in-depth later on, but it is something that is important and it's worth mentioning here if only in passing.

Let's update our views and controller to include the "isPublished" field. First the view...

```

<form method="post" action="/books/add">
  <label>Title</label> <input name="title" type="text" />
  <label>Author</label> <input name="author" type="text" />
  <label>Is Published?</label> <input name="published" type="checkbox" value="published" />
  <input type="submit" value="Submit" />
</form>

```

And then the controller...

```

exports.BookCreate = function (request, response) {

  var title = request.body.title;
  var author = request.body.author;
  if (request.body.published === 'published') {
    isPublished = true;
  }

  var b = new Model.BookModel({
    title: title,
    author: author,
    isPublished: true
  });

  b.save(function (error) {
    if (error) {
      console.log('Error');
      response.redirect('/books?error=true&message=There was an error adding the book to the
database');
    } else {

```



```

        response.redirect('/books?success=true&message=Book created successfully');
    }
});
};

```

Here we have a check to see if the "isPublished" field value is coming up in the request body. If it is, it means that the box is checked. Thus we can set this value to true. If the box is not checked we will create our book in the database with the isPublished flag set to false.

Editing and Deleting Data

How would we go about doing editing and deleting items from the database? To do these actions we'll need to have access to some sort of identifier use to have a way to tell which item (in this example which book object) we want to act upon. Fortunately MongoDB has been taking care of this for us behind the scenes. Every document in a MongoDB database has an internal "_id" property that contains a unique value that automatically gets added by MongoDB when a new item is created. More on this is discussed in the [data modeling section on the MongoDB website](#). We could see what this key/value pair looks like if we update our view for a book to look like the following (adding an {{_id}} to the view)...

```

<ul>
  {{#books}}
  <li>
    <strong>{{title}}</strong> by: {{ author }} - {{ _id }}
  </li>

  {{ else }}
  <p>There are no books.</p>

  {{/books}}
</ul>

<a href="/books/add">Add Book</a>

```

Now if you start up your app and go to the "/books" route, you'll see there will now also be a code also displayed next to each item that will look something like this...

```
55a653333cfc15942813fa6e
```

and each one will be different. That collection of random letters and number is the unique _id value for this particular entry in the database. Why is this value important? Well, it's because we're going to make use of this value to edit and delete data. If we know the unique value of the book we want to edit or delete we can act upon it without affecting the other items in the database.

Deleting Data

So let's actually start with deleting data and then we'll move on to editing data. How would we do deleting? Well to start we're going to have create a new controller method to handle delete requests. So let's add the following method to the bottom of our BookController.js file...

```

exports.BookDelete = function (request, response) {
  var id = request.params.id;
  console.log('Delete: ' + id)
}

```

And we're also going to want to specify the route for deleting an item in in our router.js...

```

...
app.get('/books', BookController.Index);
app.get('/books/add', BookController.BookAdd);
app.post('/books/add', BookController.BookCreate);
app.get('/books/delete/:id', BookController.BookDelete);

```

```
};
```

Note that we are adding a parameter for an `:id`. This is because we are going to have to pass an id to the controller as a parameter so that the "BookDelete" controller method will know which item to delete. Recall that we discussed using these named parameters in our introduction to Express.js discussion. Please see that section if you need a quick refresher on what these are and how they work. In essence, our id value is going to be available on the `request.params` object. Because our parameter is named `:id` in the route, that value is made available to us there by Express.js. It is important to note again that there is nothing significant about calling our route `:id`. The name is not really important. If we had named our parameter "tacos" like so...

```
app.get('/books/delete/:tacos', BookController.BookDelete);
```

then we could access the value passed in as the first parameter by using `request.params.tacos`. Of course, it is always good practice to use a descriptive name that is relevant to the code you are writing (so I'd recommend against using "tacos"), but just know that you have the flexibility to name your parameters whatever you like for whatever your purposes are.

Let's update our `Index.handlebars` view for books by adding a link to delete that item. We are going to make use of this unique `_id` key that is present on each item that we have in our database and pass it in as the parameter. So let's update our view to look like the following...

```
<ul>
  {{#books}}
    <li>
      <strong>{{title}}</strong> by: {{ author }} - <a href="/books/delete/{{ _id }}" onclick="return
confirm('Are you sure?')">Delete</a>
    </li>
  {{ else }}
    <p>There are no books.</p>
  {{/books}}
</ul>

<a href="/books/add">Add Book</a>
```

So as can be seen here, we have the path to the delete route where we will be passing in the `_id` key as the parameter which we will use to go find the item we want to delete. We're also adding a bit of confirmation in inline JavaScript to prevent users from accidentally deleting an item. Confirmation of destructive actions is always a good practice when you write software.

Now if you were to run the application, go to the books route and click on one of the Delete buttons next to a database item, you would get the confirmation message and then see the id of the item you would be trying to delete logged to the console. Of course, we have not written the code to actually delete the item from the database yet, so we'll do that next. So, back in our "BookDelete" controller method, let's write the code to actually delete an item. Update the code in the "BookDelete" method to look like the following...

```
exports.BookDelete = function (request, response) {
  var id = request.params.id;
  Model.BookModel.remove({ _id: id }, function (error, result) {
    if (error) {
      console.log('Error');
      response.redirect('/books?error=true&message=There was an error deleting the book');
    } else {
      response.redirect('/books?success=true&message=Book deleted successfully');
    }
  });
};
```

The code will first get the id value from the parameter using `request.params.id` as we've been discussing, and then it will use that variable in Mongoose's `remove` method to look in the database for an item that has its `"_id"` value as that parameter. When it finds it, it will delete the item and then redirect to the the main `"/books"` route. So now if

you run the app again and delete an item, if all goes well you'll see that upon redirect the item that you deleted is no longer there. Success!

Editing Data

Editing data going to be interesting because to create an "edit" view we first have to go and get the item that we want from the database and then populate the form with its values, but we are going to use a lot of the same concepts that we use to delete an item (namely, passing an id parameter to a view).

Hopefully by now the concept of creating routes, controllers, and views is becoming somewhat familiar. So to create editing functionality we should first update our router.js file. As we did with creating an item, we'll need to add 2 routes, one to display the item's data in a form and the other to receive that data when the form is submitted (posted) back to the server ...

```
...
app.get('/books', BookController.Index);
app.get('/books/add', BookController.BookAdd);
app.post('/books/add', BookController.BookCreate);
app.get('/books/edit/:id', BookController.BookEdit);
app.post('/books/edit', BookController.BookUpdate);
app.get('/books/delete/:id', BookController.BookDelete);
};
```

And let's add the controller code for our route to display the form...

```
exports.BookEdit = function (request, response) {
  var id = request.params.id;

  Model.BookModel.findOne({ _id: id }, function (error, result) {
    if (error) {
      console.log('Error');
      response.redirect('/books?error=true&message=There was an error finding a book with this id');
    }
    else {
      response.pageInfo.title = 'Edit Book';
      response.pageInfo.book = result;
      response.render('books/BookEdit', response.pageInfo);
    }
  });
};
```

Here again we are utilizing the `_id` value that gets passed forward into this view as a parameter just as we did with deleting data, only this time we are using that value to go and get (lookup) this item from the database. If we have a successful result of finding the item in the database, we'll then pass this data to the view via the "book" property (along with the other pieces of information as we've been doing)

So in our view because we have appended the book property onto the pageInfo object and set it equal to all of the data in our database, the various properties of that data item will be available to us on the book object...

```
<form method="post" action="/books/edit">
  <input name="id" type="hidden" value="{{ book._id }}" />
  <label>Title</label> <input name="title" type="text" value="{{ book.title }}" />
  <label>Author</label> <input name="author" type="text" value="{{ book.author }}" />
  <label>isPublished?</label> <input name="published" type="checkbox" value="published" {{#if
book.isPublished}} checked{{/if}} />
  <input type="submit" value="Submit" />
</form>
```

Note that we are using the Handlebars "if" statement syntax to check the checkbox if this database record has its "isPublished" flag set to true. Note too we're making use of a hidden input field that stores the `_id`. This will be needed because when we post this form back to the server, the controller that receives the request will need to know which item to update with the new values that we set.

Speaking of which, let's write our controller to receive this POST request...

```
exports.BookUpdate = function (request, response) {

  var title = request.body.title;
  var author = request.body.author;

  Model.BookModel.update(
    { _id: request.body.id },
    {
      title: title,
      author: author
    },
    { multi: true },
    function (error, result) {
      if (error) {
        console.log('Error');
        response.redirect('/books?error=true&message=There was an error updating the book in the
database');
      } else {
        response.redirect('/books?success=true&message=Book updated successfully');
      }
    }
  );
}
```

Here we are using the "body" property that is on the request object to access the form data that contains our updated values. We call Mongoose's update method by passing in updated values. Mongoose will update the database record we tell it to (by giving it the _id value) and will update the properties on that record with what we pass in as the second parameter.

The last thing we should do is put a handy link to edit each item in our Index.handlebars method for books using the _id value just as we did for deleting items...

```
<ul>
  {{#books}}
  <li>
    <strong>{{title}}</strong> by: {{ author }} - <a href="/books/edit/{{ _id }}">Edit</a> <a
href="/books/delete/{{ _id }}" onclick="return confirm('Are you sure?')>Delete</a>
  </li>

  {{ else }}
  <p>There are no books.</p>

  {{/books}}
</ul>

<a href="/books/add">Add Book</a>
```

Testing It All Out

Whew! That is a lot of information to digest all at once but we have written enough code to actually have a functioning system of data access with our MongoDB instance. If we still have our database running and waiting for connections on port 28017 we can open a new shell in the root of our project and run...

```
$ node app
```

You should see a message that the app has started and that we have successfully connected to the database. Now navigate to the "/books" route. Obviously we haven't added any books yet, so there will be a message saying that there are no books. However, we can change that by going to the "/books/add" route and adding a books there. If all goes well we should add the book and then redirect back to the "/books" route with a new shiny book in our list. From there, you can also try editing and deleting data as well.

Validation of Data

In this section we will briefly cover validation of data. We have gone over entering data into the database but we sort of glossed over the validation components i.e. verifying that the data is good before we enter it into the database.

To do validation, we are going to make use of a utilities library that we'll call "Validation" (makes sense). Like with everything else we've done, we'll make it into a module. So in the utilities folder, create a new file called Validation.js. And in the BookController.js file add a reference to this at the top.

```
var Validation = require('../utilities/Validation');
```

This is roughly what our validation file might look like...

```
exports.IsNullOrEmpty = function (check) {
  var errors = false;

  if (Object.prototype.toString.call(check) === '[object Array]') {
    for (var i = 0; i < check.length; i++) {
      if (!check[i]) {
        errors = true;
      }
      if (check[i].trim() === '') {
        errors = true;
      }
    }
  }
  else if (typeof check === 'string') {
    if (!check)
      errors = true;
    if (check.trim() === '')
      errors = true;
  }

  return errors;
};

exports.Equals = function (one, two) {
  if (one === two)
    return true;
  else
    return false;
};

exports.ValidateEmail = function (email) {
  var re = /^((([^\<>()[]\@.\,;:\s@"]+)(\.[^\<>()[]\@.\,;:\s@"]+)*))|(\\".+\\")@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.([0-9]{1,3}|([0-9]{1,3}|([0-9]{1,3}|[0-9]{1,3}))|([a-zA-Z-0-9]+\.)+[a-zA-Z-0-9]{2,}))$)/;
  return re.test(email);
};

exports.ValidateDate = function (dateString) {
  // Check pattern
  if (!/^d{4}\.d{1,2}\.d{1,2}$/.test(dateString))
    return false;

  // Parse the date parts to integers
  var parts = dateString.split("/");
  var year = parseInt(parts[0], 10);
  var month = parseInt(parts[1], 10);
  var day = parseInt(parts[2], 10);
}
```

```

    if (year < 1000 || year > 3000 || month === 0 || month > 12)
        return false;

    var monthLength = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

    // Adjust for leap years
    if (year % 400 === 0 || (year % 100 !== 0 && year % 4 === 0))
        monthLength[1] = 29;

    return day > 0 && day <= monthLength[month - 1];
};

```

Here we have a few different methods that we can pass data to and we'll want to do this before we attempt to do anything with the database. We could definitely add more methods here if we wanted to for any sort of validation that our application needed. These might include things like limiting a max number of characters, or using some regular expressions to make sure that phone numbers, credit cards or whatever else are invalid format. The important thing is that it is all here in one place where we can easily extend upon things.

So for example with our BookCreate view, we can probably just add in some validation to confirm that there are not empty strings entered into any of the fields. To do this we can use our IsNullOrEmpty method. Note that this method gives us the possibility of passing in either a single variable to check or we can pass in an array of variables to check a number of inputs at a single time,

```

exports.BookCreate = function(request, response){

    var title = request.body.title;
    var author = request.body.author;

    if(Validation.IsNullOrEmpty([title, author])) {
        response.redirect('/books/add?error=true&Please fill out all fields');
    }
    ...
}

```

Here we are doing a check if the title field and/or the author fields are empty. If either one of them are we are going to redirect to the books/add route with a message to please fill out all fields.

Speaking of redirects, we have been doing that quite a bit in our application with a lot of the same types of parameters for success and error messages in each place. Rather than write these out every time, we can roll these redirects into methods that will handle this for us. We call these "SuccessRedirect" and "ErrorRedirect" and we can make these part of the Validation module...

```

exports.SuccessRedirect = function (response, route, message) {

    if (typeof message !== 'undefined') {
        response.redirect(route + '?success=true&message=' + message);
    }
    else {
        response.redirect(route + '?success=true');
    }
};

exports.ErrorRedirect = function (response, route, message) {

    if (typeof message !== 'undefined') {
        response.redirect(route + '?error=true&message=' + message);
    }
    else {
        response.redirect(route + '?error=true');
    }
};

```

This just gives us a way to pass in what we need without having to construct the query string every time. So our BookCreate method would become something like...

```
exports.BookCreate = function (request, response) {

  var title = request.body.title;
  var author = request.body.author;

  if (Validation.IsNullOrEmpty([title, author])) {
    Validation.ErrorRedirect(response, '/books', 'Please fill out all fields');
  } else {
    var b = new Model.BookModel({
      title: title,
      author: author,
      isPublished: true
    });
    b.save(function (error) {
      if (error) {
        console.log('Error');
        Validation.ErrorRedirect(response, '/books', 'There was an error adding the book to the
database');
      } else {
        Validation.SuccessRedirect(response, '/books', 'Book created successfully');
      }
    });
  }
};
```

A little bit cleaner and every little bit helps. We'd also want to do the same in our controllers for editing and deleting items as well.

Going even further than this, we could also do validation against the database. For example, when adding a book we could first check to see if a book with the same name already exists in the database and redirect with an error message informing the user that they must enter a unique name -- if we cared about something like that. It's not likely that we'd want to do it with book titles because it is very possible that there could be two books with the same name written by different authors but there definitely are times when you'd want to do this sort of check. For example, with user accounts you'll often want to make sure that all user account names and/or e-mail addresses are unique in your system, so you'd definitely want to check if a user with the same account name/email already existed before attempting it to the database.

But since we only have a "BookModel," let's just hypothetically say for some reason we want all book titles to be unique in our system. This is what implementing a check against this would look like...

```
exports.BookCreate = function (request, response) {

  var title = request.body.title;
  var author = request.body.author;

  if (Validation.IsNullOrEmpty([title, author])) {
    Validation.ErrorRedirect(response, '/books', 'Please fill out all fields');
  } else {
    Model.BookModel.findOne({ title: title }, function (error, result) {

      if (result) {
        Validation.ErrorRedirect(response, '/books', 'There is already a book with this title in
the database');
      } else {
        var b = new Model.BookModel({
          title: title,
          author: author,
          isPublished: true
        });
      }
    });
  }
};
```

```

        b.save(function (error) {
            if (error) {
                console.log('Error');
                Validation.ErrorRedirect(response, '/books', 'There was an error adding the book
to the database');
            } else {
                Validation.SuccessRedirect(response, '/books', 'Book created successfully');
            }
        });
    }

});

}

};

```

As before we first check that all fields are filled out. We then look up if there is already a book with this title in the database. If it finds a result, we redirect with an error message saying, "There is already a book with this title in the database." If not, then it is finally okay to try to add the entry to the database.

Summary

Wow, we have covered a lot a lot on data access so far. In upcoming installments, we will look at some more advanced concepts such as session data (signing in/out) as well as securing our application from common attacks such as [cross-site request forgeries \(CSRFs\)](#).

Download the files below to play around with data access. Be sure that you install MongoDB and point it at the "db" folder in the package. This folder is empty in the download, but MongoDB will automatically add all the database files when you first run the command.

Also be sure to remember to run...

```
$ npm install
```


Chapter 7: Relational Data in MongoDB

In this section we will look at another interesting aspect of creating a web application using Node.js, Express, and MongoDB: referencing other data in the database. MongoDB is part of the [NoSQL](#) class of databases (as opposed to the traditional [relational databases](#) that have been so prevalent for so long). NoSQL databases store records as documents as opposed to relational databases which store records as rows in tables. Documents in NoSQL databases are basically collections of key value pairs sort of like JSON objects. As the MongoDB documentation describes in the [introduction...](#)

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

You can find many articles out on the web discussing the difference between NoSQL databases and relational databases. MongoDB is a solution to a common problem and like with many things it has its advantages and its drawbacks.

But even with their structure, NoSQL database documents still have need to store references to other objects within them (somewhat like foreign keys). We have been creating data models of books in previous examples in this tutorial that have a "title" field, an "author" field, and an "isPublished" field. We have been storing the author as a string value. But really in a real-world application we'd more likely want to store a reference to an author that exists in an authors collection elsewhere. That way we could store additional information about an author or show a collection of books written by the same author. When you start relating and connecting data in this manner your application becomes truly dynamic and useful

Updating Data Models

So let's create an "AuthorModel" and all of the scaffolding to put an "authors" section in the database. Like we've done before, we'll want to follow the same process: define our model schema for our "AuthorModel," add routes, add controllers, and add views. We will move fairly quickly through this. A lot of what we are doing was covered earlier in this tutorial. If there are any parts that seem unclear, please go back and review the content presented earlier to get a refresher on how this application is being structured.

So let's start by defining our author schema in our Models.js file. This one will be a bit simpler than our book model schema. All we need to add here is a name. What will be important later on is the author `_id` value that we can associate with a book as a reference field...

```
var Author = new mongoose.Schema({
  name: String
});

var BookModel = mongoose.model('Book', Book);
var UserModel = mongoose.model('User', User);
var AuthorModel = mongoose.model('Author', Author);

module.exports = {
  BookModel: BookModel,
  AuthorModel: AuthorModel,
  UserModel: UserModel
};
```

And let's continue with adding our author routes to the router.js file making sure to add a reference to our soon-to-be-created AuthorsController...

```
var HomeController = require('./controllers/HomeController');
var BookController = require('./controllers/BookController');
var AuthorController = require('./controllers/AuthorController');
```

```
// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

  // Book Routes

  app.get('/books', BookController.Index);
  app.get('/books/add', BookController.BookAdd);
  app.post('/books/add', BookController.BookCreate);
  app.get('/books/edit/:id', BookController.BookEdit);
  app.post('/books/edit', BookController.BookUpdate);
  app.get('/books/delete/:id', BookController.BookDelete);

  // Author Routes

  app.get('/authors', AuthorController.Index);
  app.get('/authors/add', AuthorController.AuthorAdd);
  app.post('/authors/add', AuthorController.AuthorCreate);
  app.get('/authors/edit/:id', AuthorController.AuthorEdit);
  app.post('/authors/edit', AuthorController.AuthorUpdate);
  app.get('/authors/delete/:id', AuthorController.AuthorDelete);

};
```

Now we can add the respective controllers by creating an AuthorsController. This will look very similar to our BooksController because we are largely needing all the same types of actions (Create, Read, Update, and Delete) as we did with our BookController. The only difference is that in this case we'll be using our author Models instead of our book models...

```
var Model = require('../models/Models');
var Validation = require('../utilities/Validation');

exports.Index = function (request, response) {

  Model.AuthorModel.find(function (error, result) {
    if (error) {
      console.log('Error');
      Validation.ErrorRedirect(response, '/', 'There was an error finding authors in the database');
    } else {
      response.pageInfo.title = 'Authors';
      response.pageInfo.authors = result;
      response.render('authors/Index', response.pageInfo);
    }
  });
};

exports.AuthorAdd = function (request, response) {
  response.pageInfo.title = 'Add an Author';
  response.render('authors/AuthorAdd', response.pageInfo);
};

exports.AuthorCreate = function (request, response) {

  var name = request.body.name;

  if (Validation.IsNullOrEmpty([name])) {
    Validation.ErrorRedirect(response, '/authors', 'Please fill out all fields');
  } else {

    var a = new Model.AuthorModel({
      name: name
    });
  }
};
```

```

    a.save(function (error) {
      if (error) {
        console.log('Error');
        Validation.ErrorRedirect(response, '/authors', 'There was an error adding the author to
the database');
      } else {
        Validation.SuccessRedirect(response, '/authors', 'Author created successfully');
      }
    });
  }
};

exports.AuthorEdit = function (request, response) {
  var id = request.params.id;

  Model.AuthorModel.findOne({ _id: id }, function (error, result) {
    if (error) {
      console.log('Error');
      Validation.ErrorRedirect(response, '/authors', 'There was an error finding an author in the
database with this id');
    }
    else {
      if (result) {
        response.pageInfo.title = 'Edit Author';
        response.pageInfo.author = result;
        response.render('authors/AuthorEdit', response.pageInfo);
      } else {
        Validation.ErrorRedirect(response, '/authors', 'There was an error finding an author in
the database with this id');
      }
    }
  });
}

exports.AuthorUpdate = function (request, response) {

  var name = request.body.name;

  if (Validation.IsNullOrEmpty([name])) {
    Validation.ErrorRedirect(response, '/authors', 'Please fill out all fields');
  } else {

    Model.AuthorModel.update(
      { _id: request.body.id },
      {
        name: name
      },
      { multi: true },
      function (error, result) {
        if (error) {
          console.log('Error')
          Validation.ErrorRedirect(response, '/authors', 'There was an error finding an author
in the database with this id');
        } else {
          Validation.SuccessRedirect(response, '/authors', 'Author updated successfully');
        }
      }
    );
  }
}

exports.AuthorDelete = function (request, response) {
  var id = request.params.id;
  Model.AuthorModel.remove({ _id: id }, function (error, result) {
    if (error) {
      console.log('Error');
      Validation.ErrorRedirect(response, '/authors', 'There was an error deleting the author');
    }
  });
}

```

```

    } else {
      Validation.SuccessRedirect(response, '/authors', 'Author deleted successfully');
    }
  });
}

```

And lastly let's create our views. Create a new folder in the main directory and call it "authors". In this folder we can create the Index.handlebars, AuthorAdd.handlebars and AuthorEdit.handlebars views.

Index.handlebars

```

<ul>
  {{#authors}}
    <li>
      <strong>{{name}}</strong> - <a href="/authors/edit/{{ _id }}">Edit</a> <a href="/authors/delete/{{
_id }}" onclick="return confirm('Are you sure?')">Delete</a>
    </li>

    {{ else }}
    <p>There are no authors.</p>

  {{/authors}}
</ul>

<a href="/authors/add">Add Author</a>

```

AuthorAdd.handlebars

```

<form method="post" action="/authors/add">
  <label>Name</label> <input name="name" type="text" />
  <input type="submit" value="Submit" />
</form>

```

AuthorEdit.handlebars

```

<form method="post" action="/authors/edit">
  <input name="id" type="hidden" value="{{ author._id }}" />
  <label>Name</label> <input name="name" type="text" value="{{ author.name }}" />
</form>

```

This is one of the advantages to building with the model-view-controller design pattern. We are able to quickly add functionality to entirely new data types and all the scaffolding we need to add entries to the database simply by creating a few new files of similar type to others.

Now if we run our application using...

```
$ node app
```

we can go to the "/authors" routes and from there add some new authors to the database. These authors are not associated with any books yet, but we'll change that shortly.

Adding Object References

Now that we have our authors in the db, the next thing we'll want to do is update our book model schema in our Models.js file to reference an author field since a book object will be associated with an author. As described in the Mongoose documentation [here](#), we can create references to other documents in our schemas. So we can do this like so. Before, our "BookModel" looked like this...

```

var Book = new mongoose.Schema({
  title: String,
  author: String,
  isPublished: Boolean

```

```
});

var Author = new mongoose.Schema({
  name: String
});
with our author stored as a string. We can update this to the following...

var Book = new mongoose.Schema({
  title: String,
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'Author' },
  isPublished: Boolean
});

var Author = new mongoose.Schema({
  name: String
});
```

So now we have an author as a reference to an author than will be associated with each book.

Of course, now that we have updated our model, we'll need to update the manner in which we handle things in our controllers because there is a little bit more involved in displaying a book in our views. We are going to have to look up the associated author_id value that exists on the book and pass it into the view along with the other regular book data. It might help to conceptualize things if you delete all of your existing books in the database and start afresh.

As always, we'll start off small and we'll work our way up. We'll start out by working on the controllers and views to add some books with associated authors. Mongoose provides us a method of using promises as a way of retrieving data asynchronously through the exec method (which returns a promise). We can then chain the various function calls together using the then method and they will execute in their proper order. This will become increasingly important as our objects and our associations become more complex. So for our controller to add a book we need to first get all authors, then return the view.

```
exports.BookAdd = function (request, response) {
  Model.AuthorModel.find({}).exec(function (error, result) {

    if (error) {
      Validation.ErrorRedirect(response, '/books', 'There was an error getting the author list');
    } else {
      response.pageInfo.authors = result;
    }
  }).then(function () {

    response.pageInfo.title = 'Add a Book';
    response.render('books/BookAdd', response.pageInfo);

  });
};
```

When we pass in the empty object { } into the exec function Mongoose interprets this to mean "find all" in this case: find all authors.

So we've added the authors object to the pageInfo response. So let's now go and update the view to add a book. We'll use a <select> box to display a dropdown of a list of authors...

```
<form method="post" action="/books/add">
  <label>Title</label> <input name="title" type="text" />
  <label>Author</label>

  <select name="author" class="">
    {{#authors}}
    <option value="{{_id}}">{{name}}</option>
    {{/authors}}
  </select>

  <label>Is Published?</label> <input name="published" type="checkbox" value="published" />
  <input type="submit" value="Submit" />
</form>
```

If you start the app up and go to the "books/add" route, you should see some of your authors populated in the dropdown. So what we send up to the server when we post the form back is no longer an author string, but rather an author "_id" value. Because we updated our BookModel

Of course, once we create a new book with our new schema we see that once we are redirected back to the "/books" route on successful adding of a book we no longer see our author name but rather our internal ugly MongoDB _id value. So let's change our home view to show the author rather than this ugly id.

So like with our controller to add a book, we'll need to get the author of each book so that we can display that author's name. It wouldn't seem very efficient to pull down all the authors and then compare the authors in the list for each book. Fortunately, Mongoose gives us a method that will allow us to populate the books we are getting back from the database with the object reference using the populate method. Thus, we'd want to update our Index controller in our BookController.js file to look like the following...

```
exports.Index = function (request, response) {
  Model.BookModel.find({}).populate('author').exec(function (error, result) {
    if (error) {
      console.log('Error');
      Validation.ErrorRedirect(response, '/', 'There was an error finding books in the database');
    } else {
      response.pageInfo.title = 'Books';
      response.pageInfo.books = result;
      response.render('books/Index', response.pageInfo);
    }
  });
};
```

See how we have used the populate method to populate the author field for each book on our book model with the author object. As a result, in our view we have direct access to this author object.

```
<ul>
  {{#books}}
  <li>
    <strong>{{title}}</strong> by: {{ author.name }} - <a href="/books/edit/{{ _id }}">Edit</a> <a
href="/books/delete/{{ _id }}" onclick="return confirm('Are you sure?')">Delete</a>
  </li>

  {{ else }}
  <p>There are no books.</p>

  {{/books}}
</ul>

<a href="/books/add">Add Book</a>
```

See how we can just access the author name via {{ author.name }} in our Index.handlebars view? Mongoose makes it nice and easy to do this. Nice and easy is always good!

Editing Items and Handlebars Helpers

We will use some of the same Mongoose concepts of getting the author object reference for our edit view for books as well. What is interesting about the edit view is that we also need to find *all* the authors as well so that we can populate the dropdown. We can't just return the current one associated with the book because maybe the user of the site would want to update the author by choosing a different author in the dropdown. So here is what this would look like in our controller...

```
exports.BookEdit = function (request, response) {
  var id = request.params.id;

  Model.BookModel.findOne({ _id: id }).populate('author').exec(function (error, result) {
```

```

    if (error) {
      console.log('Error');
      Validation.ErrorRedirect(response, '/books', 'There was an error finding a book in the
database with this id');
    }
    else {
      if (result) {

        response.pageInfo.book = result;

        Model.AuthorModel.find({}).exec(function (error, result) {

          if (error) {
            Validation.ErrorRedirect(response, '/books', 'There was an error getting the
author list');
          } else {
            response.pageInfo.title = 'Edit Book';
            response.pageInfo.authors = result
            response.render('books/BookEdit', response.pageInfo);
          }
        });
      } else {
        Validation.ErrorRedirect(response, '/books', 'There was an error finding a book in the
database with this id');
      }
    }
  });
}

```

Notice how we first find our book where we look up the record in the database using the id parameter as we have been doing before. But after this we then make a call to get all authors using the `Model.AuthorModel.find({}).exec()` method so that we can populate the dropdown. So in our view we can make some small adjustments to display everything that we need to...

```

<form method="post" action="/books/edit">
  <input name="id" type="hidden" value="{{ book._id }}" />
  <label>Title</label> <input name="title" type="text" value="{{ book.title }}" />

  <select name="author" class="">
    {{#authors}}
    <option value="{{ _id }}">{{name}}</option>
    {{/authors}}
  </select>

  <label>isPublished?</label> <input name="published" type="checkbox" value="published" {{#if
book.isPublished}} checked{{/if}} />
  <input type="submit" value="Submit" />
</form>

```

There is, however a slight problem with our edit view in its current state. You'll notice that if we have multiple authors to choose from in our dropdown we may not have the correct author selected. By default, the first author is always selected in the dropdown (as it was in our create view). But unless the current book we are editing happens to be by the first author in the list, it will not be correct.

How do we solve this? We'll need to have some way of adding a "selected" attribute to the correct `<option>` value by comparing the current book's author `_id` to the `_id` value in the `<select>` dropdown. How to do this? Because our controller does not know anything about the format of the view, as it turns out, we will need to do this comparison in the view itself.

To do this we will need to create a helper method within the handlebars templating engine to select the correct value. You can find more information on how helpers work in Handlebars on the Handlebars.js website. More important for our purposes is info on how to register helpers using the `express-handlebars` module which can be found here on npm. Following the format of registering the helper there, we will go to our `app.js` file and create a Handlebars

helper called "ifCond" You will see in a moment why this is going to be useful. So in our app.js file update the section where we register handlebars as the view template to the following...

```
/* express-handlebars - https://github.com/ericf/express-handlebars
A Handlebars view engine for Express. */
hbs = handlebars.create({
  helpers: {
    ifCond: function (v1, operator, v2, options) {

      v1 = v1.toString();
      v2 = v2.toString();

      switch (operator) {
        case '==':
          return (v1 == v2) ? options.fn(this) : options.inverse(this);
        case '===':
          return (v1 === v2) ? options.fn(this) : options.inverse(this);
        case '<':
          return (v1 < v2) ? options.fn(this) : options.inverse(this);
        case '<=':
          return (v1 <= v2) ? options.fn(this) : options.inverse(this);
        case '>':
          return (v1 > v2) ? options.fn(this) : options.inverse(this);
        case '>=':
          return (v1 >= v2) ? options.fn(this) : options.inverse(this);
        case '&&':
          return (v1 && v2) ? options.fn(this) : options.inverse(this);
        case '||':
          return (v1 || v2) ? options.fn(this) : options.inverse(this);
        default:
          return options.inverse(this);
      }
    },
  },
  defaultLayout: 'main'
});

app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');
```

What this method will do is take in 2 values and compare them as strings within our view engine. So we can now update our BookEdit view to the following...

```
<form method="post" action="/books/edit">
  <input name="id" type="hidden" value="{{ book._id }}" />
  <label>Title</label> <input name="title" type="text" value="{{ book.title }}" />

  <select name="author" class="tw-admin-post-author">
    {{#authors}}
      <option value="{{ _id }}" {{#ifCond _id '==' ../book.author._id}}
selected{{/ifCond}}>{{name}}</option>
    {{/authors}}
  </select>

  <label>isPublished?</label> <input name="published" type="checkbox" value="published" {{#if
book.isPublished}} checked{{/if}} />
  <input type="submit" value="Submit" />
</form>
```

Note the use of the "ifCond" inline in the option values. Because we are doing this in the {{#authors}} block, the context is the list of authors so the _id is the value of each author id within the list. However, we need to compare this to the author _id of the current book we are editing. To get the _id value of the book object we are editing we need to switch the context using the "../" syntax. Handlebars uses this to go "one context up" to the book object where we can get the author _id value of the current book. So what this "ifCond" condition is doing for each <option> is checking to see if the author _id set in value attribute in the current option is equal to the book author

_id of the book we are editing. If it is, we should set the <option> value where this is a match to be selected by adding the "selected" attribute.

Now if we rerun our application and go to edit a book, we should see the correct author selected in the dropdown. Pretty neat!

Summary

We are really coming along in our application development. We have relational data between different data types and we're creating the appropriate views where they are applicable. We could easily build upon this to create all kinds of different relationships and associations between data to make our application truly dynamic. All you really need to do is repeat the processes discussed above for different objects and types.

Chapter 8: APIs & AJAX in Express.js Applications

In building Express.js applications, we have largely been looking at how we get data from controllers and then pass it to views and then in our views we have forms post data from a user back to the server. Doing things via forms is all well and good, but it is not the only way that we can pass data back to our server. We can also make AJAX requests to particular endpoints and receive the data as JSON, parse it, act upon it further, and then return a response of our own.

When the web moved from HTML4 to HTML5, the rise of [AJAX](#) based web applications came along with it. AJAX (which stands for **A**synchronous **J**avaScript and **X**ML) can mean a number of things but in practice it often refers primarily to the client/server interaction where JavaScript can be used to communicate with a server by making calls at various times under various conditions to send and receive data. This can be happening as often as you like in real-time without the user having to navigate to another webpage or refresh their browser (i.e. asynchronously).

What is an example of this? Well, say I wanted to put a stock ticker on my webpage or in my web application. I could use JavaScript to make a call to a service like some stock-market API that will give me quotes on a particular stock, get the data back, and display it in the UI in a visually appealing manner. As these market prices update over time, I can periodically update the quotes that I'm displaying on the page. The format that the server returns data in really depends on the service you are calling. It can often be XML (as the X in AJAX suggests) but another popular format is JSON. I actually like JSON a bit better, myself. The experience of parsing through everything in code just seems a bit cleaner so if there is the option to have the data you are trying to get returned to you in JSON, I'll usually opt for that.

Because JavaScript is such a universally cross-platform language capable of running on so many different devices in different environments, this can be a great way for different applications to communicate with each other. The result, hopefully, is to give your users the best experience possible whether they are on desktop, mobile, tablet, or some other crazy futuristic device.

A lot of Express.js applications heavily utilize AJAX rather than doing the full request/response cycle that comes with rendering views and forms (as we have been doing). Basically, in these sorts of applications the view component will be replaced with a single-page application written in [AngularJS](#), [Backbone.js](#), [Ember.js](#) or some other client-side application framework. The client side will do all of its communication with the server via AJAX requests, sending and receiving JSON and dynamically updating the UI layer accordingly. You may have even heard of people making mention of the MEAN stack. MEAN standing for MongoDB, Express, Angular, and Node. We have been doing MongoDB, Express, and Node.js, we just have not leveraged AngularJS in any fashion yet... but this is where it fits in in this picture.

It is outside the scope of this book to cover a client-side application framework in depth so our AJAXy stuff will be limited to simple usage of [jQuery](#). So, in what follows we will briefly look at how to send an AJAX request from a view back to a controller. We will also look at how we can rig up our controllers to send JSON back to a client. This should give you enough of a starting point that, should you want to invest some time in learning AngularJS (or another framework) you can have a good understanding of how to fit it into your Node.js, Express.js, and MongoDB applications.

What is an API?

If you have done any programming on the web, chances are you have come across someone making mention of [API](#) at some point. What is meant when someone talks about APIs? There are a number of different things that an API can mean in different contexts, but in a strictly functional sense when it comes to communication over the web -- which is what we care about the most -- APIs are merely a way to send and receive data in communication with a server. You send a request to a server and you get some data back usually in the form of a standard data exchange format such as XML or JSON. For example, go to the following API URL below in your browser...

<https://api.spotify.com/v1/artists/1ZwdS5xdxEREPySFridCfh>

You should see a dump of JSON similar to the following...

```

{
  "external_urls": {
    "spotify": "https://open.spotify.com/artist/1ZwdS5xdxEREPySFridCfh"
  },
  "followers": {
    "href": null,
    "total": 1144931
  },
  "genres": [
    "g funk",
    "gangster rap",
    "hip hop",
    "rap"
  ],
  "href": "https://api.spotify.com/v1/artists/1ZwdS5xdxEREPySFridCfh",
  "id": "1ZwdS5xdxEREPySFridCfh",
  "images": [{
    "height": 1000,
    "url": "https://i.scdn.co/image/a3621b6826025390efa30e98c346cd846d8cd31e",
    "width": 1000
  },
  {
    "height": 640,
    "url": "https://i.scdn.co/image/ccf3aea3324e18cc70496fefed0a55a84aa6b22b",
    "width": 640
  },
  {
    "height": 200,
    "url": "https://i.scdn.co/image/e5d7f51fa244d7259de1a5d675359b57d36b14ed",
    "width": 200
  },
  {
    "height": 64,
    "url": "https://i.scdn.co/image/cb3c478244b60448fd73c7f7e1c45e1276399c65",
    "width": 64
  }
],
  "name": "2Pac",
  "popularity": 82,
  "type": "artist",
  "uri": "spotify:artist:1ZwdS5xdxEREPySFridCfh"
}

```

This gives information on the artist 2pac. If I wanted to, I could write my own music app using this API to display information that Spotify makes available within it. There are all different types of APIs for all types of applications and implementations... There are weather APIs, stock ticker APIs, map APIs, news story APIs, flight tracking APIs, sports score APIs, and APIs for just about everything else under the sun. If you can think of something that you want data and information for, there is probably an API for it. Not all of them are always open and publically accessible though and for many APIs you have to register an account with the service that is providing the API data and authenticate your account against the requests you make to get data from the service.

Pretty much all of the big services like Facebook, Twitter, YouTube, Google, etc. all have their own APIs that you can get data from. It's how people are able to display Facebook posts or Twitter tweets right within their own webpages or applications. They use the APIs of those companies to get information from those services to ingrate the data into their own project. Without getting too bogged down in the details APIs are simply a way to pass information around on the web.

There are also a lot of different types of API architectures... different ways of building APIs and implementing different procedures and syntaxes for users to access the data contained within the API. One popular API architecture is known as [REST](#), which stands for "representational state transfer." There are a number of different identifying features to REST API architecture. One important aspect of a REST API is that it is resource based and handles requests depending on the verb of the HTTP request type being sent (GET, POST, PUT, or DELETE). Thus, if your API was handling a collection of products, you could conceivably have 3 different "/products/:id" routes in your API with each handling things different ways depending on whether a GET, POST, PUT, or DELETE request is sent to the "/product/:id" endpoint. Presumably a GET request would get information about an existing product, a PUT request would update the information about the product, and a DELETE request would delete the product. Which product we act upon would depend on the product ID that we pass in.

Note: The :id syntax indicates a variable to store a reference value for the particular product in the system. So the actual HTTP request you would be making would pass in the product id value. This could be an integer like so "/products/4" or a hash string "/products/h5d4f51ij244d381," or anything else. It all depends on what id syntax the system uses. We saw this above with the Spotify API example request to <https://api.spotify.com/v1/artists/1ZwdS5xdxEREPySFridCfh>. Spotify it appears uses a hash string value for its ids. The "1ZwdS5xdxEREPySFridCfh" corresponds to 2Pac. Passing in a different id will get you different information for a different artist. For example, <https://api.spotify.com/v1/artists/06HL4z0CvFAxyc27GXpf02> will get information about Taylor Swift.

We will be implementing a REST API written in Node.js to pass the data in our application (our information on artists) around.

Setup

So let's start out by creating some API endpoints that we can send our requests to. We will send these requests to separate routes from those in our main application. We will just show these here as examples, but we won't really utilize them too heavily in future discussions. But the components are there in case you want to build on them on your own. So let's create a couple test routes in our router.js so that we can send some AJAX requests to them...

```
var APIController = require('./controllers/APIController');

...

// API Routes
app.get('/api', APIController.Index);

app.get('/api/test', APIController.TestGet);
app.post('/api/test', APIController.TestPost);
```

And let's create a new controller called "APIController" in our "controllers" folder with the following...

```
var Model = require('../models/Models');
var Validation = require('../utilities/Validation');

exports.Index = function (request, response) {
  response.pageInfo.title = "API Home"
  response.render('api/Index', response.pageInfo);
};

exports.TestGet = function (request, response) {
};

exports.TestPost = function (request, response) {
};
```

Our "TestGet" and "TestPost" controllers will be ones to consume our AJAX requests. We will write the code to send AJAX data there shortly.

Now we need to make sure that we have jQuery loaded up in our Main.handlebars view. We could definitely do this via the jQuery CDN and pull the file down off the Internet, but recall that we had set things up such that we can serve up a local file from our static directory so that we do not need an internet connect to have our AJAX stuff still work. So create a folder in our "static" directory called "js" and in that folder put the jQuery library in there. At the time of this writing I am using version 2.1.1.

Then in our Main.handlebars view add a reference to jQuery. I have added it at the top, but if you wanted to you could do what is often done for a slightly better page loading experience and add this script reference at the bottom of the layout instead...

```
<!DOCTYPE html>
```

```

<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">
  <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">
  <link href="/css/style.css" rel="stylesheet">
  <script type="text/javascript" src="/js/jquery.js"></script>
  <title>{{ title }}</title>

</head>
...

```

Now that we have jQuery loading in our Main.handlebars layout we can start using it in our any of our views. Create a folder in our views folder called "api" and put an Index.handlebars file in it that contains the following code...

```

<p>Our home API view...</p>

<a href="#" class="ajax-button-get">Send AJAX GET Request</a>

<script type="text/javascript">

  jQuery(document).ready(function () {

    jQuery('.ajax-button-get').on('click', function (e) {

    });

  });

</script>

```

As can be seen, what we are going to be doing here is sending an AJAX request when we click on the link in the view. In the next section we will write the code to do this.

Sending AJAX Requests

So now we are ready to write some code on the client side to try sending some data to our API endpoints. So modify our Index.handlebars view to the following...

```

<p>Our home API view</p>

<a href="#" class="ajax-button-get">Send AJAX GET Request</a>

<script type="text/javascript">

  jQuery(document).ready(function () {

    jQuery('.ajax-button-get').on('click', function (e) {
      jQuery.ajax({
        url: '/api/test',
        dataType: 'json',
        contentType: 'application/json; charset=UTF-8',
        type: 'GET',
        success: function (data, textStatus, jqXHR) {
          alert(JSON.stringify(data));
        },
        error: function () {
          alert('There was an error in the request');
        }
      });
    });

  });

```

```
</script>
```

As you can see, we will send a GET request to the "/api/test" route and we will alert the data that we get back from the route endpoint. But as you recall, we have not yet written any code in the controller to return anything. So let's add in the code to return some JSON to our view. Fortunately, Express.js gives us an easy way to return JSON via the response.json method.

```
exports.TestGet = function (request, response) {

  var obj = {
    "name": "Ian",
    "favoriteColor": "gray"
  };

  response.json(obj)
};
```

As you can see we are creating a generic object and returning upon request. So now if we fire up our application with \$ node app and we go to our "/api" route and click on our link we should see our JSON object returned from the server alerted on the client side. Nice!

What if we wanted to send data *to* the server? We can do this with a few tweaks and couple of extra properties in our AJAX jQuery code. So modify the Index.handlebars view to include some hooks for sending a POST requests...

```
<p>Our home API view</p>
```

```
<a href="#" class="ajax-button-get">Send AJAX GET Request</a>
<a href="#" class="ajax-button-post">Send AJAX POST Request</a>
```

```
<script type="text/javascript">
```

```
  jQuery(document).ready(function () {

    jQuery('.ajax-button-get').on('click', function (e) {
      jQuery.ajax({
        url: '/api/test',
        dataType: 'json',
        contentType: 'application/json; charset=UTF-8',
        type: 'GET',
        success: function (data, textStatus, jqXHR) {
          alert(JSON.stringify(data));
        },
        error: function () {
          alert('There was an error in the request');
        }
      });
    });

    var postData = {
      "book": "War and Peace",
      "author": "Leo Tolstoy"
    };

    jQuery('.ajax-button-post').on('click', function (e) {
      jQuery.ajax({
        url: '/api/test',
        dataType: 'json',
        contentType: 'application/json; charset=UTF-8',
        type: 'POST',
        data: JSON.stringify(postData),
        success: function (data, textStatus, jqXHR) {
          alert(JSON.stringify(data.message));
        },
        error: function () {
```

```

        alert('There was an error in the request');
    }
    });
});
});
</script>

```

Notice that we are going to send the "postData" object up to our "/api/test" route. Just like with other parts of our application, because we are sending a POST request it will be handled by a different controller even though the route is the same.

Now if we modify our controller code to consume the data sent to it...

```

exports.TestPost = function (request, response) {
    console.log(request.body)
    var obj = { "message": "I love " + request.body.author + " My favorite book is " + request.body.book
};
    response.json(obj);
};

```

Now if we run our app and we click on our link to send a POST request you can see in the shell window of your log that the data we are sending is available on the request.body object. Here we are basically just returning a message derived from the data that was sent up to the controller. In a real application we would often want to utilize this data to do things like creating a new model from the data being sent to the controller and saving it to the database.

Summary

We have briefly learned how to implement AJAX functionality into our Express.js applications. We could use this approach in any of our views so long as we set our controllers up to receive and process the data that we send accordingly. This is how client-side applications handle their data exchanges with servers and this can be used to create some very dynamic and interactive applications.

Chapter 9: Sessions & Authentication in Express.js

In this section we will look at [sessions](#) and how to handle them in an Express.js application using MongoDB. The context in which we will be using sessions basically refers to the process of authenticating users (stored in the database) and handling sign in and sign out events.

When you speak of authentication in web applications, there are a number of different implementations you could be talking about. One of the more traditional approaches is the use of sessions. Sessions go back as far as anyone can remember with anything having to do with the web and are distinct from other stateless authentication methods such as OAuth or HMAC. Express.js supports sessions and you can use them just as you would in any other server-side language like Java, C# or PHP.

Getting Started with Sessions and Authentication

Let's modify our package.json file to load up the components that we'll need to handle sessions. We'll need to use the [express-session](#) module as well as the [cookie-parser](#) module to support the handling of sign in and sign out events using sessions.

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "body-parser": "1.4.3",
    "express-session": "1.5.1",
    "cookie-parser": "1.3.1",
    "express-handlebars": "1.1.0",
    "morgan": "1.1.1",
    "errorhandler": "1.1.1",
    "mongoose": "3.8.8"
  }
}
```

And just like always, we'll need to install any new modules that we don't have installed by running

```
$ npm install
```

from the root of our project. We've gotten pretty good at running this command by now.

If we look at the cookie-parser documentation and the express-session documentation we can see that we need to pass in a secret key when we initialize for both of these. These keys are used for creating hashes to authenticate requests to make sure that they are coming from the properly authenticated user and can consist of any string of characters. You could even write the name of your pet dog growing up if you wanted. But since it doesn't matter a random assortment of letters and numbers will work just fine. The session key and the cookie key are great candidates to put in our config.js file. So let's add it there...

```
var config = {};

config.development = {
  database: {
    name: 'MVCApplication',
    host: 'localhost',
    port: '27017',
    credentials: ''
  },
  application: {
    port: 1337,
    sessionKey: 'm2LlqerNK7YmDj',
    cookieKey: 'nP70sqnKa07Rut'
  }
}
```



```

};

config.production = {
  database: {
    name: 'tradewinds',
    host: 'localhost',
    port: '8080',
    credentials: 'admin:password@' // username:password@
  },
  application: {
    port: 80,
    sessionKey: '0Sy6Ln34xyvcpP',
    cookieKey: 'S0QG5MWAtLGG77'
  }
};

config.environment = 'development';

module.exports = config;

```

After this we can now update our app.js file. We first need to add references to our new modules at the top of our app.js file. After this we can put our express-session and cookie-parser initializations further down. We initialize our express-session and cookie-parser modules using the references to the keys in our config file that we just added. Both of these modules run as middleware, so it makes sense to add it somewhere around where we have our other middleware functions defined...

```

var express = require('express');
var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var logger = require('morgan');
var errorHandler = require('errorhandler');
var http = require('http');
var path = require('path');
var handlebars = require('express-handlebars'), hbs;
var config = require('./config');
var Middleware = require('./utilities/Middleware');
var app = express();

app.set('port', config[config.environment].application.port);
app.set('views', path.join(__dirname, 'views'));

/* express-handlebars - https://github.com/ericf/express-handlebars
A Handlebars view engine for Express. */
hbs = handlebars.create({
  helpers: {
    ifCond: function (v1, operator, v2, options) {

      v1 = v1.toString();
      v2 = v2.toString();

      switch (operator) {
        case '==':
          return (v1 == v2) ? options.fn(this) : options.inverse(this);
        case '===':
          return (v1 === v2) ? options.fn(this) : options.inverse(this);
        case '<':
          return (v1 < v2) ? options.fn(this) : options.inverse(this);
        case '<=':
          return (v1 <= v2) ? options.fn(this) : options.inverse(this);

```

```

        case '>':
            return (v1 > v2) ? options.fn(this) : options.inverse(this);
        case '>=':
            return (v1 >= v2) ? options.fn(this) : options.inverse(this);
        case '&&':
            return (v1 && v2) ? options.fn(this) : options.inverse(this);
        case '||':
            return (v1 || v2) ? options.fn(this) : options.inverse(this);
        default:
            return options.inverse(this);
    }
}
},
defaultLayout: 'main'
});

app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

/* Morgan - https://github.com/expressjs/morgan
   HTTP request logger middleware for node.js */
app.use(logger({ format: 'dev', immediate: true }));

app.use(cookieParser(config[config.environment].application.cookieKey));
app.use(session({ secret: config[config.environment].application.sessionKey }));

app.use(express.static(path.join(__dirname, 'static')));

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(Middleware.AppendNotifications);

/* errorHandler - https://github.com/expressjs/errorhandler
   Show errors in development. */
app.use(errorHandler({ dumpExceptions: true, showStack: true }));

// send app to router
require('./router')(app);

http.createServer(app).listen(app.get('port'), function () {
    console.log('Express server listening on port ' + app.get('port'));
});

```

We can do a quick test to see if sessions are working for us. Let's create a couple of new "admin routes" in our router.js file...

```

var HomeController = require('./controllers/HomeController');
var BookController = require('./controllers/BookController');
var AuthorController = require('./controllers/AuthorController');
var AdminController = require('./controllers/AdminController');

...
// Admin Routes
app.get('/admin', AdminController.Index);
app.get('/admin/login', AdminController.Login);
app.post('/admin/login', AdminController.VerifyLogin);
app.get('/admin/logout', AdminController.Logout);

```

And let's create a new file in the controllers folder called AdminController.js and make the controller methods that correspond to our routes. We'll be adding code to these shortly...

```

var Validation = require('../utilities/Validation');

exports.Index = function (request, response) {
};

exports.Login = function (request, response) {
};

exports.VerifyLogin = function (request, response) {
};

exports.Logout = function (request, response) {
};

```

And let's create some sample views as well. Create a new folder in the views directory called "admin" and create the following views within it...

Index.handlebars

```

<p>Welcome to the admin panel {{ userid }}! You should only be seeing this if you are
signed in Click <a href="/admin/logout">here</a> to log out</p>

```

Login.handlebars

```

<form method="post" action="/admin/login">
  <label>Username</label> <input name="username" type="text" />
  <label>Password</label> <input name="password" type="password" />
  <input type="submit" value="Submit" />
</form>

```

Returning to our AdminController.js file let's add the following code to each of our individual methods...

```

var Validation = require('../utilities/Validation');

exports.Index = function (request, response) {
  response.pageInfo.title = 'Administration Panel Home';
  response.pageInfo.userid = request.session.userid;
  response.render('admin/Index', response.pageInfo);
};

exports.Login = function (request, response) {
  response.pageInfo.title = 'Login';
  response.render('admin/Login', response.pageInfo);
};

exports.Logout = function (request, response) {
  request.session.destroy();
  response.redirect('/admin/login');
};

exports.VerifyLogin = function (request, response) {
};

```

This should look fairly familiar as it is much like our approach for creating other components of our application. But what if we wanted to, say, protect the Index.handlebars view from unauthorized access by doing a check to see if a user is logged in (i.e. has an active session) before returning the view to the browser? To do this we can add an "AuthenticateAdmin" function that we can set up to run before we reach certain controller methods. So let's create a new file called Authentication.js and put it in the "utilities" folder of our project. In our file we can add an "AuthenticateAdmin" method.

```
exports.AuthenticateAdmin = function (request, response, next) {
  if (request.session.userid) {
    return next();
  }
  else {
    response.redirect('/admin/login');
  }
};
```

The function will run just like middleware making use of the next() call if there is a session with the key "userid". As we have learned, middleware by default runs on every request and response cycle. But if we set this function to run as middleware for every request, none of our application is going to be accessible because we're never going to have an active session set after login. How would we set things up to run this function before **only** the Index method in our AdminController.js file? As it turns out, Express gives us a nice way of calling individual functions only for specific routes if we place them within the routes themselves. So for us that means modifying the router.js file. As always, we'll want to make sure that we add a reference to this "AuthenticateAdmin" method via the "Authentication" module and then from there we can modify the "/admin" route. This is what it will look like...

```
var HomeController = require('./controllers/HomeController');
var BookController = require('./controllers/BookController');
var AuthorController = require('./controllers/AuthorController');
var AdminController = require('./controllers/AdminController');
var Authentication = require('./utilities/Authentication');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

  // Book Routes

  app.get('/books', BookController.Index);
  app.get('/books/add', BookController.BookAdd);
  app.post('/books/add', BookController.BookCreate);
  app.get('/books/edit/:id', BookController.BookEdit);
  app.post('/books/edit', BookController.BookUpdate);
  app.get('/books/delete/:id', BookController.BookDelete);

  // Author Routes

  app.get('/authors', AuthorController.Index);
  app.get('/authors/add', AuthorController.AuthorAdd);
  app.post('/authors/add', AuthorController.AuthorCreate);
  app.get('/authors/edit/:id', AuthorController.AuthorEdit);
  app.post('/authors/edit', AuthorController.AuthorUpdate);
  app.get('/authors/delete/:id', AuthorController.AuthorDelete);

  //Admin Routes

  app.get('/admin', Authentication.AuthenticateAdmin, AdminController.Index);
  app.get('/admin/login', AdminController.Login);
  app.post('/admin/login', AdminController.VerifyLogin);
  app.get('/admin/logout', AdminController.Logout);

};
```

See how we have added the Authentication.AuthenticateAdmin function as part of the "/admin" route? Express.js is great like this because it allows us to run as many custom functions that we want in any order that we want for any route that we want. This will give us a lot of control over how our application runs in various circumstances.

Now if we run our application and try to go to the `"/admin"` route we will find we are redirected to the `"/admin/login"` route because we do not have a session with the key `"userid"` on the session object. Our `"AuthenticateAdmin"` method has successfully protected the `"Index"` method. The check will run and because there is no session, the user will be redirected to the login route which will return a view that gives him/her a form to sign in.

So how would we go about getting a session with this proper key so that we could access the `"/admin"` route in our dashboard? Well, it would make sense to assign this key to the session object upon successful authentication. So let's add the code to do the authentication piece when the login form is filled out and submitted in our `"VerifyLogin"` method in `AdminController.js`...

```
exports.VerifyLogin = function (request, response) {
  var username = request.body.username;
  var password = request.body.password;

  if (username == 'demo' && password == 'demo') {
    request.session.userid = username;
    response.redirect('/admin');
  } else {
    Validation.ErrorRedirect('/login', 'Incorrect username or password')
  }
};
```

If you can parse through the code you can probably see that the login form is only going to accept a username of `"demo"` and a password of `"demo"`. If this check passes we will create a session with the username as the session ID (`userid` property on the session object). After this the application will redirect to the `"/admin"` route. The `"Authentication.AuthenticateAdmin"` check will run again on this redirect only this time we `**do**` have a `request.session.userid`, so the application will let us through to the `"/admin"` route.

Now if we run our application again we can see everything in action in working order.

This is obviously a very simple example to illustrate how sessions work. But you'll likely notice that this is going to be completely insufficient for a production level system of any kind because we're going to need a lot more than a mere single user named `"demo"` and a plain-text password of `"demo"`. We're going to want to store these users in a database and we're also going to want to encrypt their passwords for security purposes. We will cover all of these components next.

Authenticating Users Stored in a Database

For the approach that we'll take for handling user authentication against users stored in a database as we have done before, we'll start out by updating our `package.json` file with some new modules so that we can support it. We'll add the [connect-mongo](#) module which we will use for storing sessions in MongoDB and we'll also install the [bcrypt-nodejs](#) module which we will use for password encryption. So add these modules to `package.json`...

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "body-parser": "1.4.3",
    "express-session": "1.5.1",
    "cookie-parser": "1.3.1",
    "express-handlebars": "1.1.0",
    "morgan": "1.1.1",
    "errorhandler": "1.1.1",
    "mongoose": "3.8.8",
    "connect-mongo": "0.4.1",
    "bcrypt-nodejs": "0.0.3"
  }
}
```

And then of course run...

```
$ npm install
```

Now we need to add references to these in our app.js file. The connect-mongo module will be used in conjunction with the session module. So we'll need to add the following to our module definitions in app.js anywhere *after* we have declared our session module because we will pass the session into the connect-mongo module (see the connect-mongo documentation for more info). If we try to add it before we declare our session module variable, it will be undefined.

```
var session = require('express-session');
...
var MongoStore = require('connect-mongo')(session);
...
```

Next we can update our session middleware to use this "MongoStore" module. We initialize it by passing in our database information which we have already handily stored in our config.js file...

```
...
app.use(session({
  secret: config[config.environment].application.sessionKey,
  store: new MongoStore({
    url: 'mongodb://' + config[config.environment].database.host + '/' +
    config[config.environment].database.name
  })
}));
...
```

Next, we'll go back to our Models.js file. We'll need to make sure that we have a "UserModel" where we can store information about a user — name, password, email, or whatever else we need — in our database. We will then use these "UserModels" to distinguish between different user accounts within our system. Let's update our Models.js file in our models folder to include a UserModel...

```
var config = require('../config');
var mongoose = require('mongoose');
var connectionString = 'mongodb://' + config[config.environment].database.credentials +
config[config.environment].database.host + ':' + config[config.environment].database.port + '/' +
config[config.environment].database.name;
var db = mongoose.connection;

db.on('error', function () {
  console.log('There was an error connecting to the database');
});

db.once('open', function () {
  console.log('Successfully connected to database');
});

// Database
mongoose.connect(connectionString);

var Book = new mongoose.Schema({
  title: String,
  author: String,
  isPublished: Boolean
});

var User = new mongoose.Schema({
  name: String,
  password: String,
  email: String,
  isAdmin: Boolean
});
```

```

var BookModel = mongoose.model('Book', Book);
var UserModel = mongoose.model('User', User);

module.exports = {
  BookModel: BookModel,
  UserModel: UserModel
};

```

The information we are storing on our user includes name, password, email, and a boolean flag to determine if this user has access to the admin panel or not.

Next we will follow the same process as we have done before: update routes in the "router.js" file, add a new controller, add the views, and off we go. So let's add the following new routes in router.js. Hopefully by now the routes look familiar. Just as with our example with books and authors, these routes handle the CRUD operations (Create, Read, Update, and Delete) only this time it's for a "UserModel" instead of a "BookModel" or "AuthorModel".

```

var AdminController = require('./controllers/AdminController');
...
// Admin Routes
app.get('/admin', Authentication.Authenticate, AdminController.Index);
app.get('/admin/login', AdminController.Login);
app.post('/admin/login', AdminController.VerifyLogin);
app.get('/admin/logout', AdminController.Logout);
app.get('/admin/users', Authentication.Authenticate, AdminController.UsersViewAll);
app.get('/admin/users/add', Authentication.Authenticate, AdminController.UserAdd);
app.post('/admin/users/add', Authentication.Authenticate, AdminController.UserCreate);
app.get('/admin/users/edit/:id', Authentication.Authenticate, AdminController.UserEdit);
app.post('/admin/users/edit/:id', Authentication.Authenticate, AdminController.UserUpdate);
app.get('/admin/users/delete/:id', Authentication.Authenticate, AdminController.UserDelete);
...

```

And in our AdminController.js file, let's create the methods to handle requests to these new routes. We'll also want to add a reference to our bcrypt-nodejs module that we installed earlier because we will be using it to create a hash...

```

var Model = require('./models/Models');
var Validation = require('./utilities/Validation');
var bcrypt = require('bcrypt-nodejs');
...
// Admin - View All Users

exports.UsersViewAll = function(request, response){
  Model.UserModel.find(function(error, result){
    if (error) {
      Validation.ErrorRedirect(response, '/admin', 'usersNotFound');
    }

    response.pageInfo.title = 'View All Users';
    response.pageInfo.users = result;
    response.render('admin/UsersViewAll', response.pageInfo);
  });
};

// Admin - Add User

exports.UserAdd = function(request, response){
  response.pageInfo.title = 'Add User';
  response.render('admin/UserAdd', response.pageInfo);
};

```

```

// Admin - Create User

exports.UserCreate = function(request, response){

  var errors = false;

  var name = request.body.name;
  var email = request.body.email;
  var password = request.body.password;
  var password2 = request.body.password2;
  var admin = request.body.admin;

  if(Validation.IsNullOrEmpty([name, email, password, password2])) {
    errors = true;
  }
  if(!Validation.Equals(password, password2)) {
    errors = true;
  }
  if(!Validation.ValidateEmail(email)) {
    errors = true;
  }

  if(admin === 'admin')
    isAdmin = true;

  if(errors)
    Validation.ErrorRedirect(response, '/admin/users', 'There was an error adding the user');
  else {

    Model.UserModel.findOne({ email: email }, function(error, result){

      // user email already exists
      if(result){
        Validation.ErrorRedirect(response, '/admin/users', 'This user email already exists in the
database');
      }
      else {

        var salt = bcrypt.genSaltSync(10);
        var passwordHash = bcrypt.hashSync(password, salt);

        var u = new Model.UserModel({
          name: name,
          password: passwordHash,
          email: email,
          isAdmin: isAdmin
        });

        u.save(function(error){

          if(error)
            Validation.ErrorRedirect(response, '/admin/users', 'There was an error adding the
user');
          else
            Validation.SuccessRedirect(response, '/admin/users', 'User added successfully');
        });

      }

    });

  }
};

// Admin - Edit User

exports.UserEdit = function(request, response){

  var id = request.params.id;

```



```

Model.UserModel.findOne({ _id: id }, function(error, result){
    if(error) {
        Validation.ErrorRedirect(response, '/admin/users', 'There was an error finding a user with
this id in the database');
    }
    response.pageInfo.title = 'Edit User';
    response.pageInfo.user = {
        id: result._id,
        name: result.name,
        email: result.email,
        isAdmin: result.isAdmin
    };
    response.render('admin/UserEdit', response.pageInfo);
});

});

// Admin - Update User

exports.UserUpdate = function(request, response){
    var errors = false;
    var isAdmin = false;

    var name = request.body.name;
    var email = request.body.email;
    var admin = request.body.admin;

    if(Validation.IsNullOrEmpty([name, email]))
        errors = true;
    if(!Validation.ValidateEmail(email))
        errors = true;

    if(admin === 'admin')
        isAdmin = true;

    if(errors)
        Validation.ErrorRedirect(response, '/admin/users', 'There was an error updating the user');
    else {
        Model.UserModel.update(
            { _id: request.body.id },
            {
                name: name,
                email: email,
                isAdmin: isAdmin
            },
            { multi: true },
            function(error, result){
                if(error) {
                    Validation.ErrorRedirect(response, '/admin/users', 'There was an error updating the
user');
                }
                else {
                    Validation.SuccessRedirect(response, '/admin/users', 'User updated successfly');
                }
            }
        );
    }
};

// Admin - Delete User

exports.UserDelete = function(request, response){
    Model.UserModel.remove({ _id: request.params.id }, function(error, result) {

```

```

    if (error) {
      Validation.ErrorRedirect(response, '/admin/users', 'There was an error deleting the user');
    }
    else {
      Validation.SuccessRedirect(response, '/admin/users', 'User deleted successfully');
    }
  });
});
};

```

Notice that on user creation or user updating we are first doing a little bit of validation to confirm that all fields are filled out and e-mail is in the correct format of an e-mail address. We're also doing some checking to make sure that passwords match for a little bit of additional confirmation. After that we are making use of our `bcrypt-nodejs` module to encrypt the password.

And of course, we'll need to create the views for these controllers...

UsersViewAll.handlebars

```

<ul>
  {{#users}}
    <li>
      <strong>Name:</strong> {{name}} <strong>Email:</strong> {{email}}
      <a href="/admin/user/edit/{{ _id }}">Edit</a>
      <a href="/admin/users/delete/{{ _id }}" onclick="return confirm('Are you sure?')">Delete</a>
    </li>
  {{/users}}
</ul>

<a href="/admin/users/add">Add User</a>

```

UserAdd.handlebars

```

<form method="post" action="/admin/users/add">
  <label>Name</label> <input name="name" type="text" />
  <label>Password</label> <input name="password" type="password" />
  <label>Confirm Password</label> <input name="password2" type="password" />
  <label>E-Mail</label> <input name="email" type="text" />
  <label>Admin?</label> <input name="admin" type="checkbox" value="admin" />
  <input type="submit" value="Submit" />
</form>

```

UserEdit.handlebars

```

<form method="post" action="/admin/users/edit">
  <label>Name</label> <input name="name" type="text" value="{{ user.name }}" />
  <label>E-Mail</label> <input name="email" type="text" value="{{ user.email }}" />
  <label>Admin?</label> <input name="admin" type="checkbox" value="admin" {{#if user.isAdmin}}
checked{{/if}} />
  <input name="id" type="hidden" value="{{ user.id }}" />
  <input type="submit" value="Submit" />
</form>

```

Now we'll switch gears and work on finishing out the final pieces of putting our system of authentication together.

Updating Code to Check Credentials on Sign In

Many readers will notice that we'll need to update the code in our "VerifyLogin" method to check the credentials entered against our "UserModel" objects that we'll be storing in the database. Update the VerifyLogin" method to the following...

```

exports.VerifyLogin = function (request, response) {

```

```

    Model.UserModel.findOne({ 'name': request.body.username, isAdmin: true }, 'name password', function
(error, user) {
    if (error) {
        Validation.ErrorRedirect(response, '/admin/login', 'There was an error logging in');
    }

    // user found
    if (user) {
        // compare passwords
        if (bcrypt.compareSync(request.body.password, user.password)) {
            request.session.userid = user.name;
            response.redirect('/admin');
        }
        else {
            Validation.ErrorRedirect(response, '/admin/login', 'The username or password were
incorrect');
        }
    }
    else {
        Validation.ErrorRedirect(response, '/admin/login', 'The user is not found in the database');
    }
});
});

```

Here we are doing a check to see if the user exists in the database and has his/her isAdmin flag set to true. We then confirm if the passwords are a match. Because we are storing encrypted passwords in the database, we have to make sure that we run the comparison through our bcrypt-nodejs module. If everything is all good we will set the session as we've done before and we'll redirect to the admin panel home page.

Installation and Creating a Default User

Now that we have updated our method to check credentials many users will notice a slight problem with our implementation. We have, in essence, "coded ourselves into a corner." We have our credentials check updated, but we don't actually have any users stored as "UserModel" objects in the database. So if we try to login the query to find a user will always fail. This is a problem.

So to implement this, we will need to in essence create a controller to handle the initial installation process for our application. This is one of the first thing that you do when you're installing any software that uses logins of any type. If you have ever installed a CMS such as [WordPress](#) you'll likely remember that one of the first things that you do is create a default user. Even your OS (Windows, Linux, and Mac) walks you through this process. It's a very common component of software installs.

So let's update our application to handle installation of our application. In our router.js file at the top let's add a reference to a new controller called "InstallController" (which we will create shortly) and add some new install routes to the file...

```

var AdminController = require('./controllers/AdminController');
var InstallController = require('./controllers/InstallController');

...
// Admin Routes
app.get('/admin', Authentication.Authenticate, AdminController.Index);
app.get('/admin/login', AdminController.Login);
app.post('/admin/login', AdminController.VerifyLogin);
app.get('/admin/logout', AdminController.Logout);
app.get('/admin/users', Authentication.Authenticate, AdminController.UsersViewAll);
app.get('/admin/users/add', Authentication.Authenticate, AdminController.UserAdd);
app.post('/admin/users/add', Authentication.Authenticate, AdminController.UserCreate);
app.get('/admin/users/edit/:id', Authentication.Authenticate, AdminController.UserEdit);
app.post('/admin/users/edit', Authentication.Authenticate, AdminController.UserUpdate);
app.get('/admin/users/delete/:id', Authentication.Authenticate, AdminController.UserDelete);

```

```
// Installation Routes
app.get('/install', InstallController.Index);
app.post('/install', InstallController.Install);
app.get('/install/success', InstallController.InstallSuccess);
...
```

Create a file called "InstallController.js" and place it in the "controllers" directory. In this file add the following code (which are just empty methods) for now. We will also be adding a few references to the utilities that we will be needing when we write the code for this controller...

```
var Validation = require('../utilities/Validation');
var Model = require('../models/Models');
var bcrypt = require('bcrypt-nodejs');

exports.Index = function (request, response) {

};

exports.InstallSuccess = function (request, response) {

};

exports.Install = function (request, response) {

};
```

And then of course we can add views for this controller. Since installation is kind of an admin level type of process we do not need really need to create a separate directory for the "install" views. We actually will only need one view for this controller and can just put it in the "views/admin" folder. Express is flexible enough that you don't necessarily have to have your views in a specific folder that corresponds exactly to the name of your "controllers" folder.

So create a view called "Install.handlebars" and place it in the "views/admin" folder. In this file add the following...

```
<p>Welcome to {{ title }} the installation setup...</p>

{{#if installed}}
<p>You have successfully installed the application.</p>

<p>Click <a href="/admin/login">here</a> to login to the admin panel.</p>
{{else}}
<form method="post" action="/install">
  <label>Default User Name</label>
  <input name="name" type="text" />
  <label>Default User Password</label>
  <input name="password" type="password" />
  <label>Confirm Default User Password</label>
  <input name="password2" type="password" />
  <label>Default User Email</label>
  <input name="email" type="text" />
  <input type="submit" value="Submit" />
</form>

{{/if}}
```

So we will either show one of two things in this view. If we have already installed the application. The message "You have successfully installed the application" will be displayed. If not we show a form where the user can create a default admin login.

So now let's add the code for our controller...

```
var Validation = require('../utilities/Validation');
var Model = require('../models/Models');
```

```

var bcrypt = require('bcrypt-nodejs');

exports.Index = function (request, response) {
  Model.UserModel.findOne({ isAdmin: true }, function (error, result) {
    if (!result || error) {
      response.pageInfo.title = 'Install';
      response.pageInfo.installed = false;
      response.render('admin/Install', response.pageInfo);
    }
    else {
      response.redirect('/install/success');
    }
  });
};

exports.InstallSuccess = function (request, response) {
  response.pageInfo.title = 'Install';
  response.pageInfo.installed = true;
  response.render('admin/Install', response.pageInfo);
};

exports.Install = function (request, response) {
  var errors = false;

  var name = request.body.name;
  var email = request.body.email;
  var password = request.body.password;
  var password2 = request.body.password2;

  if (Validation.IsNullOrEmpty([name, email, password, password2])) {
    errors = true;
  }

  if (!Validation.ValidateEmail(email)) {
    errors = true;
  }

  if (!Validation.Equals(password, password2)) {
    errors = true;
  }

  if (errors) {
    Validation.ErrorRedirect(response, '/install', 'installError');
  }
  else {
    var salt = bcrypt.genSaltSync(10);
    var passwordHash = bcrypt.hashSync(request.body.password, salt);

    var u = new Model.UserModel({
      name: request.body.name,
      password: passwordHash,
      email: request.body.email,
      isAdmin: true,
    });

    u.save(function (error) {
      if (error) {
        Validation.ErrorRedirect(response, '/install', 'There was an error during installation. Please try again');
      }
      else {
        Validation.SuccessRedirect(response, '/install', 'Default user created!');
      }
    });
  }
};

```

```

    });
  }
};

```

In the "Index" method we are going to the database to check to see if there is an admin user installed in the database. Obviously at first there will not be so we will set the "installed" flag to false which will, if you recall the Handlebars condition in our view, display our form.

So the actual installation process takes place in the "Install" method because this is the method that runs when a user sends the form POST after filling out all the fields. There is a bunch of checking that takes place to make sure that the e-mail is valid and that the passwords match (the password needs to be reentered as is the case in a lot of sign-up forms). If all goes well, we will redirect to the "install/success" which will render the same view that displays the form only this time we will set the "installed" flag to true so that the message telling the user that the installation process has been completed will be displayed this time instead of the form. If we wanted to we could have created 2 separate views for these, but the way that we have done it is another way to accomplish it.

So start up your web application and go to the "/install" route. Remember that this route is not protected with our "Authentication.AuthenticateAdmin" method so we can just access it freely in our browser. Fill out the form and if all goes well, you should see a message saying that you have successfully installed the application. Anytime that we try to return to this "/install" route we should see this message (unless of course we delete out default admin user from the database). From here you can go to the "/admin" route (there is a link you can click on successful install) and login with your newly created username and password! Ta da! We can then go to the link to view users and we can see the information for our default user. We can also add some more users here if we want.

Authenticated AJAX Requests

There is one final update we need to make now that we have implemented our authentication. We saw earlier how to implement AJAX in Express.js applications. It is worth noting that the server will not just assume authentication with AJAX even if you have already signed in and have a valid session. If you try to make an AJAX request to one of the routes protected by authentication you will get a 401 Unauthorized error in response. To send AJAX requests to protected routes you need to send an additional object to attach your session to the request. This is done in jQuery via the "xhrFields" property and sending an object with the "withCredentials" property sent to true.

```

jQuery.ajax({
  url: "/admin/protected/url",
  dataType: 'json',
  contentType: 'application/json; charset=UTF-8',
  data: JSON.stringify(postData),
  type: 'POST',
  xhrFields: { withCredentials: true }, // required for authenticating session in AJAX request
  success: function (data, textStatus, jqXHR) {
    alert(data);
  },
  error: function () {
    alert('There was an error with the request');
  }
});

```

Now the server will allow the request go through as authenticated.

Summary

If you have made it this far in this tutorial hopefully you are seeing some great progress in learning how to build an application with Node.js, Express, and MongoDB. We have looked at how to implement authentication (including that for AJAX requests) and we have even created an installation process where we create a default user. I really think that at this point we basically have all the pieces that we need to create any type of application that we want! It's just a matter of extending upon this functionality with more data objects and types integrated together.

Now that we have implemented authentication and we are authenticating against users stored in a database, we should probably take a brief look at security and how we can further lock things down.

Chapter 10: Security

In what follows we will briefly discuss security within our Express.js web applications. Web security is a fairly deep and involved field and many volumes of literature have been written about the concepts surrounding it. So it is probably outside the scope of this book to go into a truly in-depth examination of all the ins and outs of every approach that one can use to protect an application from malicious users. And even with an in-depth knowledge base and even with a detailed understanding of how to leverage modern, sophisticated implementations of web security to protect against ill-intentioned users no application, it seems, is ever truly 100% secure.

Nonetheless, it is still important to have at least a general understanding of some of the more common attacks that hackers will try to use within a web site or web application and to follow some best practices when it comes to standard approaches to minimize the opportunity for such exploits as much as possible. We want to, at the very least, remove the low hanging fruit. We will cover some of those in what follows. For readers interested in a more in-depth understanding of different aspects of web security the [Open Web Application Security Project](#) (OWASP) has a great wiki to peruse through and there are many numbers of books one can pick up on the subject.

Cross-site Request Forgery

The first type of attack we will look at is [Cross-site Request Forgery](#), abbreviated CSRF, sometimes pronounced "sea surf". This type of attack is basically a method where, if I were an attacker, I'd try to get another user who is signed in/authenticated in his or her account within a system to unknowingly do things with that authentication level. So for example, let's say you were the administrator for a content management system (CMS) and you were signed into the admin panel and you received an email with the following content

Hey admin,

I think there is a bug on one of your pages here It doesn't look right.

Thanks!

Now in the e-mail, all you will see is the link and unless you closely examine it you might be apt to just click on it in your eagerness to find out what the bug is. Many readers will notice that the link is not actually a link to a page but rather a request to the system to delete a user with ID 123. If you are signed in to your account the request to this route will be executed because the browser that you are using has a valid authenticated session. Without CSRF protection I, as an attacker, have used your authentication permissions to get you to execute a request that I would not have otherwise been authorized to do. I have executed a cross-site request forgery.

So what is the solution for this? The solution for this is marking sure that every request to our system comes from a form within our system that we know can only be coming from that place within the system. The way that we do this is we put a token -- a randomly generated string of letters and numbers -- within our forms and check for its presence and value on every request. The token value is stored on the server behind the scenes. When the request comes through these values are compared and if they do not match we do not execute anything any further. It's sort of like the application is doing a password check against itself. In this manner, we can know when the request is indeed coming from our system and not elsewhere and can be allowed to proceed.

So for Express.js applications we'll use a module called [csurf](#) that we will use to put an anti-CSRF token in our forms. So when we would add a module to our package.json file

```
{
  "name": "MVC-Express-Application",
  "description": "An Express.js application using the MVC design pattern...",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.4.4",
    "body-parser": "1.4.3",
    "express-session": "1.5.1",
    "cookie-parser": "1.3.1",
    "express-handlebars": "1.1.0",
```



```

    "morgan": "1.1.1",
    "errorhandler": "1.1.1",
    "mongoose": "3.8.8",
    "connect-mongo": "0.4.1",
    "bcrypt-nodejs": "0.0.3",
    "csurf": "1.2.2"
  }
}

```

And, of course, do

```
$ npm install
```

again.

Now we can add a reference to this module in app.js and to use the middleware, it's just as easy as calling it as a function.

```

...
var logger = require('morgan');
var csrf = require('csurf');
var methodOverride = require('method-override');
...
app.use(csrf());
app.use(Middleware.AppendNotifications);
app.use(Middleware.CSRFToken);
...

```

Recall earlier that we created a Middleware.js file in the utilities folder. We can add the following function within it.

```

exports.CSRFToken = function (request, response, next) {
  response.locals.csrfToken = request.csrfToken();
  next();
};

```

This function will append a CSRF token onto the locals property in the response.

Now in all of our views that contain forms, we'll want to do something like the following (using the UserAdd.handlebars view as an example)...

```

<form method="post" action="/admin/users/add">
  <label>Name</label> <input name="name" type="text" />
  <label>Password</label> <input name="password" type="password" />
  <label>Confirm Password</label> <input name="password2" type="password" />
  <label>E-Mail</label> <input name="email" type="text" />
  <label>Admin?</label> <input name="admin" type="checkbox" value="admin" />
  <input name="_csrf" type="hidden" value="{{ csrfToken }}" />
  <input type="submit" value="Submit" />
</form>

```

If you examine the source of the form via developer tools or "View Source," you should see something like the following...

```

<form method="post" action="/admin/users/add">
  <label>Name</label> <input name="name" type="text">
  <label>Password</label> <input name="password" type="password">
  <label>Confirm Password</label> <input name="password2" type="password">
  <label>E-Mail</label> <input name="email" type="text">
  <label>Admin?</label> <input name="admin" type="checkbox" value="admin">
  <input name="_csrf" type="hidden" value="bVOPfeJu-FLNXMgYG1Gwry3vLyV2B8ggzMyU">
  <input type="submit" value="Submit">
</form>

```

Because we have passed the token value to our form, the server is now expecting it. Now if you try to submit a form without this input tag within it, the request will not go through. We have successfully protected against CSRF attacks.

Anti-CSRF Tokens and AJAX Requests

We saw earlier that in order to send AJAX requests to a route that required authentication and a valid session we had to send up some additional information with our request. We accomplished this in jQuery by sending an object consisting of a "withCredentials" property set to true and sending it in the "xhrFields" property of a jQuery AJAX request so that our request and session would be recognized by the server.

With CSRF protection, the story is the same. Unless we send the token along with our request, it will be rejected by our anti-CSRF protection middleware. So how do we do this? Turns out that the csrf module is looking for a header named "x-csrf-token" with the CSRF token value. So, if we are using jQuery we can set this header using the "beforeSend" property. One way or another, we'll have to pass the token value to our client side JavaScript...

```
jQuery.ajax({
  url: "/admin/protected/url",
  beforeSend: function (request) {
    request.setRequestHeader("x-csrf-token", {{ csrfToken }}); // required for CSRF
  },
  dataType: 'json',
  contentType: 'application/json; charset=UTF-8',
  data: JSON.stringify(postData),
  type: 'POST',
  xhrFields: {withCredentials: true}, // required for authenticating session in AJAX request
  success: function(data, textStatus, jqXHR){
    alert(data);
  },
  error: function(){
    alert('There was an error with the request');
  }
});
```

Now the CSRF token will be recognized and the request will be allowed through by the middleware...

Cross-Site Scripting (XSS)

[Cross-Site Scripting](#) (or XSS) is the process by which a malicious user injects scripts within the information that he/she is sending to the server. There are a lot of different flavors with varying levels of sophistication but the main idea is that I try to get script to execute within the web application. So if I were using a comment board and I entered the following into the comment text area...

```
<script>alert('Go to XXX SPAMMY INSECURE SITE XXX')</script>
```

If this comment gets posted to the page as it is, every user is going to see the alert because the script will just execute as JavaScript within the page. What if instead of alert, I use window.location to redirect anyone who visits the page to my own URL? From there, I might be able to further exploit hapless users with additional various types of attacks. Left unchecked, XSS attacks can cause a lot of damage and compromise both your users and your application.

The key to protecting against XSS attacks is sanitization, sanitization, sanitization. We covered checking user input in an earlier discussion of validation, but this takes things a bit further. Anytime you are going to take in user input from anywhere, it's important to run it through a function that will escape potentially malicious characters. It's part of that old adage of web security. Never trust your users EVER!

There are lots of modules that will do this for you [express-sanitized](#) being one of them. These modules will either escape unsafe blocks of text or characters or remove them altogether. So if we were to escape the malicious comment example text above, it would show up in the page as...

```
&lt;script&gt;alert('Go to XXX SPAMMY INSECURE SITE XXX')&lt;/script&gt;
```

When you sanitize a string the script tag characters will get encoded into HTML special chars `<` and `>`. As a result, this text block will not be interpreted by the browser as script and executed but will rather be rendered as mere plain text.

Session Hijacking / Cookie Stealing

An XSS attack can be used in conjunction with cookie stealing to gain access to cookies of different users of the system. Because cookies by default can be accessed via JavaScript an injected XSS script might look for cookie values of any users who come to the page and then send that information via AJAX back to some location where an attacker might be able to use that information to gain access to your system under a different identity or perform other exploits utilizing the cookies they have taken.

If you use any cookies at all to store session information or do any other form of authentication or maintaining state via cookies it's important that you set your cookies to be HTTP only (i.e. not accessible to JavaScript). The express-session module should set this flag automatically for you, but if you were using another module or were setting your own cookies to keep track of state in some manner or another e.g. a remember me cookie, here is how you would make sure your cookie

```
response.cookie('sessionId', '123', { httpOnly: true });
```

This will make the cookie inaccessible to JavaScript...

Use HTTPS

This last one does not really have as much to do with code as it does with how you set up the server that you will be running your Node.js application on. If you plan on having your application be public and visible to any number of users out in the Internet, it's always going to be a good idea to encrypt all of your web application's traffic via [TLS / SSL](#), sent over HTTPS. Getting a certificate just gives you a lot of protection up front against a number of different vulnerabilities and potential exploits that naturally exist from having an unsecure connection. One of these vulnerabilities is known as a [Man in the Middle](#) (MitM) attack. Basically, if you are on a network on an unsecure connection with a sever, anyone else on that same network can see the messages you are sending back and forth between a server and your browser because everything is sent in plain text over the wire. This includes logins, passwords, cookies and anything else. So do not log into any accounts that you care about at the local coffee shop on their WiFi if that website does not use HTTPS.

However, if you use HTTPS, all traffic sent back and forth between a client and a server is encrypted. A "man in the middle" might be able to see it but they will not be able to decipher it if proper encryption algorithms are used. All sent messages will be encoded and decoded on the server and/or the client. All the MitM will see is a big hash string of letters and numbers.

If you are using SSL you will want to make sure that your cookie are HTTPS only cookies. The way that you set this in your application is

```
response.cookie('sessionId', '123', { secure: true });
```

But remember, only set this value if you are using HTTPS.

All in all, purchasing a certificate is well worth it if your application is public and there is potential liability in you storing user information on your server. There are many different resources out there that will let you know how you go about obtaining one.

Summary

Web security is a battle in which you must stay ever-vigilant in your quest to thwart, or at the very least slow down the ill-intentions of users who are looking to exploit your web application. In short, the battle will never be over but

if you take as many steps as you possibly can to deter malicious attackers it can go a long ways. Hopefully this discussion has helped on some level to move you in that direction.

Chapter 11: Putting it All Together: Creating an Application

I would have to say that we have come a long ways since we began this journey and if you have made it this far hopefully you have learned one or two things that you've found useful in your aspirations to learn Node.js, Express.js, and MongoDB. In this chapter we will put it all together and create a content management system (CMS). Our application is probably not going to have as many amazing features as something like a [WordPress](#) or a [Drupal](#), but it will be functional and you if you wanted to you could take it and expand upon it or use it as a starting point for other applications. The final code for this application is available in a GitHub repository [here](#). We are going to name our application "Trade Winds CMS" because, well, you have to name your application something memorable. When I first started creating this application I was in Hawaii. Having spent quite a bit of time in Hawaii (my wife is from there), I like the trade winds. They're very relaxing. Hopefully it evokes images of a relaxing developer and end-user experience. When you write your own application, you can name it what you want.

All of that aside, this application that we will write in what follows will incorporate a lot of the same aspects of Node.js, Express.js, and MongoDB that were covered in depth in earlier discussions. Thus, this chapter will move at a pace that is a bit quicker than earlier tutorials. But we will be writing the application using the same approaches and in the same style that we have been doing all along in preceding sections. So we won't go too in-depth in our discussions of the various bits of code that we write, but hopefully most of it will look somewhat to what we have seen before. If there is anything that you come across that seems unclear or unfamiliar, there should be some more detailed explanations that can be found in earlier corresponding paragraphs.

So without further ado, let's jump into our application setup.

Setting Up Our Application

To start out, create a new directory called "tradewinds." This will serve as the root folder for our project. In this directory, create a package.json file and add the following for our dependencies and metadata...

```
{
  "name": "Tradewinds",
  "description": "A CMS Concept",
  "version": "0.0.1",
  "repository": {
    "type": "git",
    "url": "git://github.com/9bitStudios/tradewinds.git"
  },
  "dependencies": {
    "express": "4.4.4",
    "express-session": "1.5.1",
    "cookie-parser": "1.3.1",
    "body-parser": "1.4.3",
    "multer": "0.1.0",
    "morgan": "1.1.1",
    "csurf": "1.2.2",
    "method-override": "2.0.2",
    "errorhandler": "1.1.1",
    "express-handlebars": "2.0.1",
    "mongoose": "4.0.7",
    "connect-mongo": "0.4.1",
    "bcrypt-nodejs": "0.0.3",
    "emailjs": "0.3.8"
  }
}
```

And of course let's open a command window and run our npm command to install the dependencies defined in our package.json file...

```
$ npm install
```

This should create our "node_modules" folder for us with all of the various modules that we will use in our application.

Next, let's create some additional directories inside of our "tradewinds" folder that will serve to house the various components of our application...

- controllers
- db
- models
- static
- utilities
- views

Will be adding a number of different files to these directories in upcoming sections as we build out our application.

Next, create a file called tradewinds.js at the top level (in our "tradewinds" folder) and add the following code to it...

```
var express = require('express');
var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var multer = require('multer');
var logger = require('morgan');
var csrf = require('csurf');
var methodOverride = require('method-override');
var errorHandler = require('errorhandler');
var MongoStore = require('connect-mongo')(session);
var http = require('http');
var path = require('path');
var handlebars = require('express-handlebars'), hbs;
var config = require('./config');
var Middleware = require('./utilities/Middleware');
var app = express();

app.set('port', config[config.environment].application.port);
app.set('views', path.join(__dirname, 'views'));

/* express3-handlebars - https://github.com/ericf/express-handlebars
A Handlebars view engine for Express. */
hbs = handlebars.create({
  helpers: {
    ifCond: function (v1, operator, v2, options) {

      v1 = v1.toString();
      v2 = v2.toString();

      switch (operator) {
        case '==':
          return (v1 == v2) ? options.fn(this) : options.inverse(this);
        case '===':
          return (v1 === v2) ? options.fn(this) : options.inverse(this);
        case '<':
          return (v1 < v2) ? options.fn(this) : options.inverse(this);
        case '<=':
          return (v1 <= v2) ? options.fn(this) : options.inverse(this);
        case '>':
          return (v1 > v2) ? options.fn(this) : options.inverse(this);
        case '>=':
          return (v1 >= v2) ? options.fn(this) : options.inverse(this);
        case '&&':
          return (v1 && v2) ? options.fn(this) : options.inverse(this);
        case '||':
          return (v1 || v2) ? options.fn(this) : options.inverse(this);
        default:
          return options.inverse(this);
      }
    }
  }
});
```

```

    },
    defaultLayout: 'main'
  });

app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

/* Morgan - https://github.com/expressjs/morgan
   HTTP request logger middleware for node.js */
app.use(logger({ format: 'dev', immediate: true }));

/* cookie-parser - https://github.com/expressjs/cookie-parser
   Parse Cookie header and populate req.cookies with an object keyed by the cookie names. */
app.use(cookieParser(config[config.environment].application.cookieKey));

/* express-session - https://github.com/expressjs/session
   Simple session middleware for Express */
app.use(session({
  secret: config[config.environment].application.sessionKey,
  store: new MongoStore({
    url: 'mongodb://' + config[config.environment].database.host + '/' +
    config[config.environment].database.name
  })
}));

/* multer - https://github.com/expressjs/multer
   Multer is a node.js middleware for handling multipart/form-data. It is written on top of busboy for
   maximum efficiency. */
app.use(multer({ dest: './uploads/' }));

/* body-parser - https://github.com/expressjs/body-parser
   Node.js body parsing middleware. */
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

/* method-override - https://github.com/expressjs/method-override
   Lets you use HTTP verbs such as PUT or DELETE in places where the client doesn't support it. */
app.use(methodOverride());

/* csurf - https://github.com/expressjs/csurf
   Node.js CSRF protection middleware. Requires either a session middleware or cookie-parser to be
   initialized first. */
app.use(csurf());

app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'static')));

app.use(Middleware.CSRFTOKEN);
app.use(Middleware.AppendPageInfo);
app.use(Middleware.AppendNotifications);

/* errorHandler - https://github.com/expressjs/errorhandler
   Show errors in development. */
app.use(errorHandler({ dumpExceptions: true, showStack: true }));

// send app to router
require('./router')(app);

http.createServer(app).listen(app.get('port'), function () {
  console.log('Express server listening on port ' + app.get('port'));
});

```

This file will serve as our application kickoff that we will run first to run our application. We will be using a lot of different middleware, some of it coming from our 3rd party modules, and some of it we will write ourselves. The 3rd party stuff is included in the file above because we have already installed all of the dependencies that we need via npm install from our package.json file. We will add our own middleware on top of this in a "Middleware.js" file that we will put into our "utilities" folder.

Note too from the code above that we will also need a "router.js" and a "config.js file." Create these 2 files at the same level as the "tradewinds.js" file that we created earlier (in our root directory). Our configuration file, config.js, will look much like the ones we used in earlier discussions. Just like in our examples there, we will add both a "development" and "production" configuration to the file even though we will mostly be using the development configuration...

```
var config = {};

config.development = {
  database: {
    name: 'tradewinds',
    host: 'localhost',
    port: '27017',
    credentials: '' // username:password@
  },
  smtp: {
    username: "username",
    password: "password",
    host: "smtp.gmail.com",
    port: 587,
    ssl: false
  },
  application: {
    port: 1337,
    sessionKey: 'Sh9rXDrGh9nABj',
    cookieKey: '8YQM5GUAtLAT34'
  }
};

config.production = {
  database: {
    name: 'tradewinds',
    host: 'localhost',
    port: '8080',
    credentials: 'admin:password@' // username:password@
  },
  smtp: {
    username: "username",
    password: "password",
    host: "smtp.yourmailserver.com",
    port: 25,
    ssl: false
  },
  application: {
    port: 80,
    sessionKey: 'tce6K52sCZOLSV',
    cookieKey: '5SCjWfsTW8ySu1'
  }
};

config.environment = 'development';
module.exports = config;
```

We can leave our router.js file empty for now. We will return to it when we are ready to add our routes later on.

Middleware

Next, in our "utilities" folder create 2 files: one called "Middleware.js" and another called "Notifications.js."

In the "Middleware.js" file place the following code...


```

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

exports.AppendPageInfo = function (request, response, next) {
  response.pageInfo = {
    title: '',
    menus: {},
    userInfo: {
      name: null
    },
    notifications: {
      success: null,
      error: null
    }
  };
  next();
};

exports.AppendNotifications = function (request, response, next) {
  if (request.param('success')) {
    response.pageInfo.notifications.success = Notifications.GetNotification('success',
request.param('message'));
  }
  else if (request.param('error')) {
    response.pageInfo.notifications.error = Notifications.GetNotification('error',
request.param('message'));
  }

  next();
};

exports.CSRFToken = function (request, response, next) {
  response.locals.csrfToken = request.csrfToken();
  next();
};

```

These utility functions will allow us to "attach" a number of additional properties to the response object when we are routing the requests that come into our Node.js server. Note above that we will make use of the parameters to attach notifications to our "pageInfo" object that we will use to store all of our custom data for our views. The message that we send to the view will depend upon the property in the parameter that we will use to look up the notification message that we will want to display. This will come from a function called "GetNotification" in our "Notifications.js" file. So in our "Notifications.js" file let's add this...

```

exports.GetNotification = function (type, id) {

  var success = {
    'userAdded': 'User added',
    'userUpdated': 'User updated',
    'userDeleted': 'User deleted',
    'postCreated': 'Post created',
    'postUpdated': 'Post updated',
    'postDeleted': 'Post deleted',
    'categoryCreated': 'Category created',
    'categoryUpdated': 'Category updated',
    'categoryDeleted': 'Category deleted',
    'profileUpdated': 'Profile updated',
    'installSuccess': 'Installation Success'
  };

  var error = {
    'loginFailed': 'Username or password are incorrect',
    'usersNotFound': 'Users were not found',
    'userNotFound': 'This user ID was not found',

```

```

    'userAddError': 'There was an error adding the user',
    'userExists': 'Username or email already exists. Please choose a unique username and email',
    'userUpdateError': 'There was an error updating the user',
    'userDeleteError': 'There was an error deleting the user',
    'postsNotFound': 'Posts were not found',
    'postCreateError': 'There was an error creating the post',
    'postUpdateError': 'There was an error updating the post',
    'postDeleteError': 'There was an error deleting the post',
    'categoryNotFound': 'Category was not found',
    'categoriesNotFound': 'Categories were not found',
    'categoryCreateError': 'There was an error creating the category',
    'categoryUpdateError': 'There was an error updating the category',
    'categoryDeleteError': 'There was an error deleting the category',
    'installError': 'There was an error during installation'
  };

  if (type === 'success') {
    return success[id];
  }
  else {
    return error[id];
  }
};

```

So as we can see from this, we have a number of different messages that we can return depending on the "key" passed in as the id parameter. This key will be used to look up the corresponding key in the object literal and we will return the message string value that is associated with that key. The reason that we have done things this way in having all of our messaging in this one location is that it makes it easy to scale and add new messages when we need them as our application grows and changes. It also gives a central source to build upon if we ever wanted to implement something like localization (supporting different languages in our application). We would only need to work inside of this module and we wouldn't have to worry about having our messaging scattered across our application. Obviously, there are other ways to incorporate this as a feature and there are different approaches you can take to solving this. I have chosen to take this approach because it sets the application up anticipating future changes and extensions. It is a staple of good application design... building things in a way that they can more easily adapt to both scale and change.

Router.js

In our "router.js" file, we will now add some routes for our application. Let's start by just adding a basic "home" route. To handle the requests to this home route, we will run code in a "HomeController.js" file. So create a "HomeController.js" file and place it in the "controllers" directory. Add the following code to "router.js" file...

```

var HomeController = require('./controllers/HomeController');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

};

```

As we can see we have 2 routes here. One "/" route for the homepage of our application and another route called "/other" that is more-or-less just used for testing purposes.

Now in the "HomeController.js" file add the following code...

```

exports.Index = function (request, response) {
  response.pageInfo.title = 'Hello World';
  response.render('home/Index', response.pageInfo);
};

```

```

};

exports.Other = function (request, response) {
  response.pageInfo.title = 'Other';
  response.render('home/Other', response.pageInfo);
};

```

As we can see we are using the pageInfo object that gets attached by our middleware utility function. We will use this to display certain elements of our application in our views. In this example so far it is just static text because we are starting out slow, but we will get into passing some more complex data objects to our views when we build out more of this application later on.

Views

Lastly of course we will need to create our views. The first thing that we will want to do is create our "layouts" and "partials" directories in our "views" folder. If you recall, these will serve as the "container" or "parent" view(s) that will be used by the individual views called in our controllers. So create a "layouts" folder in the "views" directory and create a "Main.handlebars" file with the following in it...

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">
  <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">
  <link href="/css/style.css" rel="stylesheet">
  <script src="/js/jquery.2.1.1.js" type="text/javascript"></script>
  <script src="/js/jquery-ui.1.10.4.js" type="text/javascript"></script>

  <title>{{ title }}</title>

</head>
<body>

  <div> Title</div>

  <div> Notifications</div>

  <div>{{body}}</div>

</body>
</html>

```

As we can see, we need the partial views for our page title and our notifications. So in our "partials" folder create a view called "Title.handlebars" and "Notifications.handlebars". In the "Title.handlebars" folder add the following code...

```

<h1>{{ title }}</h1>

```

And in the "Notifications.handlebars" file add the following code...

```

<div class="notifications">

  <div class="success">
    {{ notifications.success }}
  </div>
  </div>

  <div class="error">
    {{ notifications.error }}
  </div>

</div>

```

```

    <div class="error">
      {{ notifications.error }}
    </div>
  {{/if}}
</div>

<script type="text/javascript">

  (function ($) {

    $(document).ready(function () {
      setTimeout(function () {
        $('.notifications').fadeOut();
      }, 3000);
    });

  })(jQuery);

</script>

```

As we can see we have the auto-fadeout feature in our jQuery code.

In the "views" folder create a directory called "home" and in it add 2 files: "Index.handlebars" and "Other.handlebars." Both of these will correspond to the controller functions that we have for these "home" routes.

Index.handlebars

```
<p>Welcome to the website...</p>
```

Other.handlebars

```
<p>Welcome to {{ title }} except this is a different route using a different view...</p>
```

Static Files

We saw above that we are including jQuery and a CSS file in our layout view. These files will need to be included in the project. So in our "static" folder we will put jQuery, jQuery UI, and a CSS file that are available for use across our application. There will also be some other JavaScript files utilized in the admin panel that we will build later on. So we will include those as well. Remember that the demo code for this discussion can be found in Appendix A.

Running the Application

We actually now have all of the basic components that we need to run our application. Even though we are not using it yet, we do still have to start up our database because we have configured our application to expect this in our "tradewinds.js" file. So to start things up, go to the directory where you installed Mongo and navigate into the bin directory and open a command window. So on Windows, for example, if you installed in the C:\mongodb directory, you would open a command window or shell in C:\mongodb\bin. You could also get there by opening a command window anywhere and typing

```
cd "C:\mongodb\bin"
```

Then you would type the following into the command window where we should specify the path to the "db" folder we created earlier. So wherever your application lives on your system you'll want to specify that path by running the following command.

```
mongod --dbpath "C:\path\to\application\db"
```

or if you're using [BASH](#), something like this...

```
$ mongod --dbpath /c/path/to/application/db
```

If all goes well, you should see a message saying that MongoDB is waiting for connections. By default at the time of this writing, the version of MongoDB being used waits for connections on port 28017... so you should see that reflected somewhere in the messaging. Leave this running by leaving this window open (you can minimize if you like).

Now we can give our application a test by opening another command window in the root directory of our project where our "tradewinds.js" file resides and running...

```
$ node tradewinds
```

You should see a message saying that the application is running. Open a browser and go to <http://localhost:1337/> and then <http://localhost:1337/other>. You should see the 2 different routes being served up with different content for each. Now that we know that everything is working, we can build out the more complex components of our application.

Data Models

Now that we know that our application "works" at a basic level we can create the data structures that we need to model our application. So create a file called "Models.js" in our models folder and add the following code to it...

```
var config = require('../config');
var mongoose = require('mongoose');
var connectionString = 'mongodb://' + config[config.environment].database.credentials +
config[config.environment].database.host + ':' + config[config.environment].database.port + '/' +
config[config.environment].database.name;
var db = mongoose.connection;

db.on('error', function () {
  console.log('There was an error connecting to the database');
});

db.once('open', function () {
  console.log('Successfully connected to database');
});

// Database
mongoose.connect(connectionString);

//Schemas
var User = new mongoose.Schema({
  name: String,
  password: String,
  email: String,
  avatar: String,
  isAdmin: Boolean,
  isDefault: Boolean
});

var Post = new mongoose.Schema({
  title: String,
  date: Date,
  slug: String,
  category: { type: mongoose.Schema.Types.ObjectId, ref: 'Category' },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  path: String,
  content: String,
  updated: Date
});

var Category = new mongoose.Schema({
  name: String,
  slug: String,
  isDefault: Boolean
});
```

```
//Models
var UserModel = mongoose.model('User', User);
var PostModel = mongoose.model('Post', Post);
var CategoryModel = mongoose.model('Category', Category);

module.exports = {
  UserModel: UserModel,
  PostModel: PostModel,
  CategoryModel: CategoryModel
};
```

That will be all we need for our application data.

Building the Admin Panel

Now that we have things working and our data models structured, we can build our admin panel. To start we will want to update our "router.js" file with all of our admin routes. So let's update our router.js file to include these admin routes...

```
var HomeController = require('./controllers/HomeController');
var AdminController = require('./controllers/AdminController');
var Authentication = require('./utilities/Authentication');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

  // Admin Routes

  app.get('/admin', Authentication.AuthenticateAdmin, AdminController.Index);
  app.get('/admin/login', AdminController.Login);
  app.post('/admin/login', AdminController.VerifyLogin);
  app.get('/admin/logout', AdminController.Logout);
  app.get('/admin/users', Authentication.AuthenticateAdmin, AdminController.UsersViewAll);
  app.get('/admin/user/add', Authentication.AuthenticateAdmin, AdminController.UserAdd);
  app.post('/admin/user/add', Authentication.AuthenticateAdmin, AdminController.UserCreate);
  app.get('/admin/user/edit/:id', Authentication.AuthenticateAdmin, AdminController.UserEdit);
  app.post('/admin/user/edit', Authentication.AuthenticateAdmin, AdminController.UserUpdate);
  app.get('/admin/user/delete/:id', Authentication.AuthenticateAdmin, AdminController.UserDelete);

  app.get('/admin/posts', Authentication.AuthenticateAdmin, AdminController.PostsViewAll);
  app.get('/admin/post/add', Authentication.AuthenticateAdmin, AdminController.PostAdd);
  app.post('/admin/post/add', Authentication.AuthenticateAdmin, AdminController.PostCreate);
  app.get('/admin/post/edit/:id', Authentication.AuthenticateAdmin, AdminController.PostEdit);
  app.post('/admin/post/edit', Authentication.AuthenticateAdmin, AdminController.PostUpdate);
  app.get('/admin/post/delete/:id', Authentication.AuthenticateAdmin, AdminController.PostDelete);

  app.get('/admin/categories', Authentication.AuthenticateAdmin, AdminController.CategoriesViewAll);
  app.get('/admin/category/add', Authentication.AuthenticateAdmin, AdminController.CategoryAdd);
  app.post('/admin/category/add', Authentication.AuthenticateAdmin, AdminController.CategoryCreate);
  app.get('/admin/category/edit/:id', Authentication.AuthenticateAdmin, AdminController.CategoryEdit);
  app.post('/admin/category/edit', Authentication.AuthenticateAdmin, AdminController.CategoryUpdate);
  app.get('/admin/category/delete/:id', Authentication.AuthenticateAdmin,
AdminController.CategoryDelete);

};
```

So as we can see we have a number of common elements for components used within a CMS (user management, blog posts/pages, and categories for those posts).

We are using an authentication module to validate that the user who is logging in can be authenticated against a user stored in the database. So we will need to add an "Authentication.js" file in our "utilities" directory. So create this "Authentication.js file" and put the following code in it...

```
exports.AuthenticateAdmin = function (request, response, next) {
  if (request.session.userid && request.session.username && request.session.admin) {
    return next();
  }
  else {
    response.redirect('/admin/login');
  }
};
```

Pretty simple. We will set a session when a user logs in successfully to the admin panel. All the authentication module will do will check to see that a session with that user name and id exists.

We will also need to add a "Validation.js" file to our "utilities" folder. This file will be used for checking that data is of the correct valid type before we add the data to the database. So create the "Validation.js" file in this folder and add the following code...

```
exports.SuccessRedirect = function (response, route, message) {
  if (typeof message !== 'undefined')
    response.redirect(route + '?success=true&message=' + message);
  else
    response.redirect(route + '?success=true');
};

exports.ErrorRedirect = function (response, route, message) {
  if (typeof message !== 'undefined')
    response.redirect(route + '?error=true&message=' + message);
  else
    response.redirect(route + '?error=true');
};

exports.IsNullOrEmpty = function (check) {
  var errors = false;

  if (Object.prototype.toString.call(check) === '[object Array]') {
    for (var i = 0; i < check.length; i++) {
      if (!check[i]) {
        errors = true;
      }
      if (check[i].trim() === '') {
        errors = true;
      }
    }
  }
  else if (typeof check === 'string') {
    if (!check)
      errors = true;
    if (check.trim() === '')
      errors = true;
  }

  return errors;
};

exports.Equals = function (one, two) {
  if (one === two)
```

The various methods in this file will take some input as function parameters and check that the input passes the test(s) outlined within the message, returning true or false depending on whether the tests pass or not. We will use this pretty extensively in our admin panel controller code because we will need to make sure that data integrity is consistent within our CMS for things to work properly.

```
exports.RandomString = function (strLength) {
    if (typeof strLength === 'undefined')
        strLength = 8;

    var text = "";
    var possible = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    for (var i = 0; i < strLength; i++)
        text += possible.charAt(Math.floor(Math.random() * possible.length));

    return text;
};

exports.GetFileNameWithoutExtension = function (filename) {
    return filename.substring(0, filename.lastIndexOf("."));
};

exports.GetFileExtension = function (filename) {
    var i = filename.lastIndexOf('.');
    return filename.substring(i + 1, filename.length);
};
```



```

    return (i < 0) ? '' : filename.substr(i);
};

exports.GetFormattedDate = function (date) {

    var month = date.getMonth() + 1; // getMonth() returns 0 -11
    if (month <= 9)
        month = '0' + month;

    var day = date.getDate();
    if (day <= 9)
        day = '0' + day;

    return date.getFullYear() + '/' + month + '/' + day;

};

```

This is kind of like a "miscellaneous" bucket of methods that do a variety of tasks that we will need for our CMS application. The great thing about this is that you also now have a spot to add a number of different functions that you need at some point down the road.

So now that we have this set, we can create our "AdminController.js" file. Place this file in our "controllers" directory and add the following code...

```

var Helpers = require('../utilities/Helpers');
var Authentication = require('../utilities/Authentication');
var Validation = require('../utilities/Validation');
var Model = require('../models/Models');
var bcrypt = require('bcrypt-nodejs');
var defaultLayout = 'admin';

// Admin - Home

exports.Index = function (request, response) {

    response.pageInfo.title = 'Administration Panel Home';
    response.pageInfo.layout = defaultLayout;
    response.pageInfo.userInfo.name = request.session.username;
    response.render('admin/Index', response.pageInfo);

};

exports.Login = function (request, response) {

    if (request.session.user && request.session.admin)
        response.redirect('/admin');

    response.pageInfo.title = 'Login';
    response.render('admin/Login', response.pageInfo);

};

exports.Logout = function (request, response) {
    request.session.destroy();
    response.redirect('/admin/login');
};

exports.VerifyLogin = function (request, response) {

    Model.UserModel.findOne({ 'name': request.body.username, isAdmin: true }, 'name password', function
(error, user) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/login', 'loginFailed');
        }

        // user found
        if (user) {
            // compare passwords

```

```

        if (bcrypt.compareSync(request.body.password, user.password)) {
            request.session.userid = user._id;
            request.session.username = user.name;
            request.session.admin = 1;
            response.redirect('/admin');
        }
        else {
            Validation.ErrorRedirect(response, '/admin/login', 'loginFailed');
        }
    }
    else {
        Validation.ErrorRedirect(response, '/admin/login', 'loginFailed');
    }
}

});

};

// Admin - View All Users

exports.UsersViewAll = function (request, response) {

    Model.UserModel.find(function (error, result) {

        if (error) {
            Validation.ErrorRedirect(response, '/admin', 'usersNotFound');
        }

        response.pageInfo.title = 'View All Users';
        response.pageInfo.layout = defaultLayout;
        response.pageInfo.userInfo.name = request.session.username;
        response.pageInfo.users = result;
        response.render('admin/UsersViewAll', response.pageInfo);

    });

};

// Admin - Add User

exports.UserAdd = function (request, response) {

    response.pageInfo.title = 'Add User';
    response.pageInfo.layout = defaultLayout;
    response.pageInfo.userInfo.name = request.session.username;
    response.render('admin/UserAdd', response.pageInfo);

};

// Admin - Create User

exports.UserCreate = function (request, response) {

    var errors = false;

    var name = request.body.name;
    var email = request.body.email;
    var password = request.body.password;
    var password2 = request.body.password2;

    if (Validation.IsNullOrEmpty([name, email, password, password2])) {
        errors = true;
    }
    if (!Validation.Equals(password, password2)) {
        errors = true;
    }
    if (!Validation.ValidateEmail(email)) {
        errors = true;
    }
}

```

```

if (errors)
  Validation.ErrorRedirect(response, '/admin/users', 'userAddError');
else {
  Model.UserModel.findOne({ email: email }, function (error, result) {
    // user email already exists
    if (result) {
      Validation.ErrorRedirect(response, '/admin/users', 'userExists');
    }
    else {
      var salt = bcrypt.genSaltSync(10);
      var passwordHash = bcrypt.hashSync(password, salt);

      var u = new Model.UserModel({
        name: name,
        password: passwordHash,
        email: email,
        avatar: 'placeholder.png',
        isAdmin: false,
        isDefault: false
      });

      u.save(function (error) {
        if (error)
          Validation.ErrorRedirect(response, '/admin/users', 'userAddError');
        else
          Validation.SuccessRedirect(response, '/admin/users', 'userAdded');
      });
    }
  });
}
};

// Admin - Edit User

exports.UserEdit = function (request, response) {
  var id = request.params.id;

  Model.UserModel.findOne({ _id: id }, function (error, result) {
    if (error) {
      Validation.ErrorRedirect(response, '/admin/users', 'userNotFound');
    }
    response.pageInfo.title = 'Edit User';
    response.pageInfo.layout = defaultLayout;
    response.pageInfo.userInfo.name = request.session.username;
    response.pageInfo.user = {
      id: result._id,
      name: result.name,
      email: result.email,
      avatar: result.avatar,
      isAdmin: result.isAdmin,
      isDefault: result.isDefault
    };
    response.render('admin/UserEdit', response.pageInfo);
  });
};

// Admin - Update User

```

```

exports.UserUpdate = function (request, response) {

  var errors = false;
  var isAdmin = false;

  var name = request.body.name;
  var email = request.body.email;
  var avatar = request.body.avatar;
  var admin = request.body.admin;

  if (Validation.IsNullOrEmpty([name, email, avatar]))
    errors = true;
  if (!Validation.ValidateEmail(email))
    errors = true;

  if (admin === 'admin')
    isAdmin = true;

  if (errors)
    Validation.ErrorRedirect(response, '/admin/users', 'userUpdateError');
  else {

    Model.UserModel.update(
      { _id: request.body.id },
      {
        name: name,
        email: email,
        avatar: avatar,
        isAdmin: isAdmin
      },
      { multi: true },
      function (error, result) {
        if (error) {
          Validation.ErrorRedirect(response, '/admin/users', 'userUpdateError');
        }
        else {
          Validation.SuccessRedirect(response, '/admin/users', 'userUpdated');
        }
      }
    );
  }
};

// Admin - Delete User

exports.UserDelete = function (request, response) {

  Model.UserModel.remove({ _id: request.params.id, isDefault: false }, function (error, result) {
    if (error) {
      Validation.ErrorRedirect(response, '/admin/users', 'userDeleteError');
    }
    else {
      Validation.SuccessRedirect(response, '/admin/users', 'userDeleted');
    }
  });
};

// Admin - View All Posts

exports.PostsViewAll = function (request, response) {

  Model.PostModel.find(function (error, result) {

    if (error) {
      Validation.ErrorRedirect(response, '/admin', 'postsNotFound');
    }
  });
};

```

```

        response.pageInfo.title = 'Posts';
        response.pageInfo.layout = defaultLayout;
        response.pageInfo.userInfo.name = request.session.username;
        response.pageInfo.posts = result;
        response.render('admin/PostsViewAll', response.pageInfo);
    });
};

// Admin - Add Post
exports.PostAdd = function (request, response) {
    Model.CategoryModel.find({}).exec(function (error, result) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/posts', 'categoriesNotFound');
        }
        response.pageInfo.categories = result;

    }).then(Model.UserModel.find({}).exec(function (error, result) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/posts', 'usersNotFound');
        }
        authorResultSet = result;

        response.pageInfo.title = 'Add New Post';
        response.pageInfo.layout = defaultLayout;
        response.pageInfo.authors = result;
        response.pageInfo.userInfo.name = request.session.username;
        response.render('admin/PostAdd', response.pageInfo);
    }));
};

// Admin - Create Post
exports.PostCreate = function (request, response) {
    var errors = false;
    var title = request.body.title;
    var date = request.body.date;
    var slug = request.body.slug;
    var category = request.body.category;
    var author = request.body.author;
    var content = request.body.content;
    var postTime;
    var path;

    if (Validation.IsNullOrEmpty([title, date, slug, content])) {
        errors = true;
    }

    if (!Validation.ValidateDate(date)) {
        errors = true;
    }

    if (errors)
        Validation.ErrorRedirect(response, '/admin/posts', 'postCreateError');
    else {
        Model.CategoryModel.findOne({ _id: category }).exec(function (error, result) {
            if (error) {
                Validation.ErrorRedirect(response, '/admin/posts', 'categoriesNotFound');
            }

```

```

    }
    else {
      if (Validation.IsNullOrEmpty(result.slug)) {
        path = slug;
      }
      else {
        path = result.slug + '/' + slug;
      }
    }
  }
}

}).then(Model.UserModel.findOne({ _id: author })).exec(function (error, result) {

  if (error) {
    Validation.ErrorRedirect(response, '/admin/posts', 'userNotFound');
  } else {

    postTime = new Date(date);

    var p = new Model.PostModel({
      title: title,
      date: postTime,
      slug: slug,
      category: category,
      author: author,
      path: path,
      content: content,
      updated: Date.now()
    });

    p.save(function (error) {

      if (error) {
        Validation.ErrorRedirect(response, '/admin/posts', 'postCreateError');
      }
      else {
        Validation.SuccessRedirect(response, '/admin/posts', 'postCreated');
      }
    });

  }
}));
}
});

// Admin - Edit Post

exports.PostEdit = function (request, response) {

  var id = request.params.id;

  Model.PostModel.findOne({ _id: id }).populate('author').populate('category').exec(function (error,
result) {

    if (error) {
      Validation.ErrorRedirect(response, '/admin/posts', 'postNotFound');
    }
    else {

      var postID = result._id;
      var postTitle = result.title;
      var formattedDate = Helpers.GetFormattedDate(result.date);
      var postSlug = result.slug;
      var postCategory = result.category;
      var postAuthor = result.author;
      var postContent = result.content;

      Model.CategoryModel.find({}).exec(function (error, result) {

        if (error) {

```

```

        Validation.ErrorRedirect(response, '/admin/posts', 'categoriesNotFound');
    }

    else {
        response.pageInfo.categories = result;
    }

}).then(Model.UserModel.find({}).exec(function (error, result) {

    if (error) {
        Validation.ErrorRedirect(response, '/admin/posts', 'usersNotFound');
    }
    else {
        response.pageInfo.title = 'Edit Post: ' + postTitle;
        response.pageInfo.layout = defaultLayout;
        response.pageInfo.userInfo.name = request.session.username;
        response.pageInfo.authors = result;
        response.pageInfo.post = {
            id: postID,
            title: postTitle,
            date: formattedDate,
            category: postCategory,
            author: postAuthor,
            slug: postSlug,
            content: postContent
        };
        response.render('admin/PostEdit', response.pageInfo);
    }

}));

});
});
};

```

// Admin - Update Post

```

exports.PostUpdate = function (request, response) {

    var errors = false;

    var title = request.body.title;
    var date = request.body.date;
    var slug = request.body.slug;
    var category = request.body.category;
    var author = request.body.author;
    var content = request.body.content;
    var postTime;
    var path;

    if (Validation.IsNullOrEmpty([title, date, slug, content])) {
        errors = true;
    }

    if (!Validation.ValidateDate(date)) {
        errors = true;
    }

    if (errors)
        Validation.ErrorRedirect(response, '/admin/posts', 'postUpdateError');
    else {

        Model.CategoryModel.findOne({ _id: category }).exec(function (error, result) {

            if (error) {
                Validation.ErrorRedirect(response, '/admin/posts', 'categoriesNotFound');
            }
            else {

```

```

        if (Validation.IsNullOrEmpty(result.slug)) {
            path = slug;
        }
        else {
            path = result.slug + '/' + slug;
        }
    }
}).then(Model.UserModel.findOne({ _id: author })).exec(function (error, result) {

    postTime = new Date(date);

    Model.PostModel.update(
        { _id: request.body.id },
        {
            title: title,
            date: date,
            slug: slug,
            category: category,
            author: author,
            path: path,
            content: content,
            updated: Date.now()
        },
        { multi: true },
        function (error, result) {
            if (error) {
                Validation.ErrorRedirect(response, '/admin/posts', 'postUpdateError');
            }
            else {
                Validation.SuccessRedirect(response, '/admin/posts', 'postUpdated');
            }
        }
    );
});
});
};

// Admin - Delete Post
exports.PostDelete = function (request, response) {

    Model.PostModel.remove({ _id: request.params.id }, function (error, result) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/posts', 'postDeleteError');
        }
        else {
            Validation.SuccessRedirect(response, '/admin/posts', 'postDeleted');
        }
    });
});

// Admin - View All Categories
exports.CategoriesViewAll = function (request, response) {

    Model.CategoryModel.find(function (error, result) {

        if (error) {
            Validation.ErrorRedirect(response, '/admin', 'categoriesNotFound');
        }

        response.pageInfo.title = 'Categories';
        response.pageInfo.layout = defaultLayout;
        response.pageInfo.userInfo.name = request.session.username;
        response.pageInfo.categories = result;
        response.render('admin/CategoriesViewAll', response.pageInfo);
    });
};

```



```

    });
};

// Admin - Add Category
exports.CategoryAdd = function (request, response) {
    response.pageInfo.title = 'Add New Category';
    response.pageInfo.layout = defaultLayout;
    response.pageInfo.userInfo.name = request.session.username;
    response.render('admin/CategoryAdd', response.pageInfo);
};

// Admin - Create Category
exports.CategoryCreate = function (request, response) {
    var errors = false;
    var name = request.body.name;
    var slug = request.body.slug;

    if (Validation.IsNullOrEmpty([name, slug])) {
        errors = true;
    }

    if (errors)
        Validation.ErrorRedirect(response, '/admin/categories', 'categoryCreateError');
    else {
        var c = new Model.CategoryModel({
            name: name,
            slug: slug,
            isDefault: false
        });

        c.save(function (error) {
            if (error) {
                Validation.ErrorRedirect(response, '/admin/categories', 'categoryCreateError');
            }
            else {
                Validation.SuccessRedirect(response, '/admin/categories', 'categoryCreated');
            }
        });
    }
};

// Admin - Edit Category
exports.CategoryEdit = function (request, response) {
    var id = request.params.id;

    Model.CategoryModel.findOne({ _id: id, isDefault: false }, function (error, result) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/categories', 'categoryNotFound');
        }
        else {
            response.pageInfo.title = 'Edit Category';
            response.pageInfo.layout = defaultLayout;
            response.pageInfo.userInfo.name = request.session.username;
            response.pageInfo.category = {
                id: result._id,
                name: result.name,
            };
        }
    });
};

```

```

        slug: result.slug
    };
    response.render('admin/CategoryEdit', response.pageInfo);
}

});

};

// Admin - Update Category

exports.CategoryUpdate = function (request, response) {

    var errors = false;

    var name = request.body.name;
    var slug = request.body.slug;

    if (Validation.IsNullOrEmpty([name, slug])) {
        errors = true;
    }

    if (errors)
        Validation.ErrorRedirect(response, '/admin/categories', 'categoryUpdateError');
    else {

        Model.CategoryModel.update(
            { _id: request.body.id },
            {
                name: name,
                slug: slug
            },
            { multi: true },
            function (error, result) {
                if (error) {
                    Validation.ErrorRedirect(response, '/admin/categories', 'categoryUpdateError');
                }
                else {
                    Validation.SuccessRedirect(response, '/admin/categories', 'categoryUpdated');
                }
            }
        );
    }
};

// Admin - Delete Category

exports.CategoryDelete = function (request, response) {

    Model.CategoryModel.remove({ _id: request.params.id, isDefault: false }, function (error, result) {
        if (error) {
            Validation.ErrorRedirect(response, '/admin/categories', 'categoryDeleteError');
        }
        else {
            Validation.SuccessRedirect(response, '/admin/categories', 'categoryDeleted');
        }
    });
};

```

There is a lot of code/functionality in the methods above. We will not go too in depth for each of these because what they are and what they do are somewhat implied by how they are named and how they are structured has been covered in earlier discussions. Hopefully you are able to use the debugging techniques you used earlier to step through the code and figure out what is happening as the code executes line by line.

Lastly of course we need to create all of the views to pass the data generated in our above methods in our admin controller. One thing that we will need to do is create a new "admin" layout. We will do this because there will be

some additional scripts that we will be including in our admin view. For example, for our page editor we will be using the [TinyMCE](#) editor to edit our post/page content. We will also be including an "admin.js" file for some additional functionality. This admin layout will also give us the ability to include different CSS for our admin layout. This will give users an easy way to know whether they are on the public facing site or if they are in the admin panel.

So in our "layouts" folder in our "views" directory create an "Admin.handlebars" layout file and add the following code...

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="">
  <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">

  <title>{{ title }}</title>

  <!-- Custom styles for this template -->
  <link href="/css/style.css" rel="stylesheet">
  <script src="/js/jquery.2.1.1.js" type="text/javascript"></script>
  <script src="/js/jquery-ui.1.10.4.js" type="text/javascript"></script>
  <script src="/js/tinymce/tinymce.min.js"></script>
  <script src="/js/admin.js" type="text/javascript"></script>
</head>

<body class="tw-admin">

  <{{> admin/UserInfo}}

  <{{> Title}}

  <{{> Notifications}}

  <{{{body}}}>

</body>
</html>
```

The great thing about doing things this way is that we will also be able to add custom parts specific to our admin views. For example, this "user info" section will display the currently logged in user's name and a link to sign out. You may not necessarily want this for your regular, non-admin views.

```
<div class="user">

  <{{#if userInfo.name}}>
  Welcome {{ userInfo.name }} - <a href="/admin">Admin Home</a> <a href="/admin/logout">Logout</a>
  <{{/if}}>

</div>
```

So create a folder called "admin" in our "views" directory. And in this folder let's add the following views with the following code...

CategoriesViewAll.handlebars

```
<ul>
  <{{#categories}}>

  <{{#unless isDefault}}>
  <li>
```

```

        <strong>{{name}}</strong> - {{ slug }}
        <a href="/admin/category/edit/{{ _id }}">Edit</a>
        <a href="/admin/category/delete/{{ _id }}" onclick="return confirm('Are you sure?')">Delete</a>
    </li>
    {{/unless}}
    {{/categories}}
</ul>

<a href="/admin/category/add" class="tw-button-medium tw-button-black-flat">Add Category</a>

```

CategoryAdd.handlebars

```

<form method="post" action="/admin/category/add">
    <label>Name</label> <input name="name" type="text" />
    <label>Slug</label> <input name="slug" type="text" />

    <input name="_csrf" type="hidden" value="{{ csrftoken }}" />
    <input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

```

CategoryEdit.handlebars

```

<form method="post" action="/admin/category/edit">
    <label>Name</label> <input name="name" type="text" value="{{ category.name }}" />
    <label>Slug</label> <input name="slug" type="text" value="{{ category.slug }}" />
    <input name="id" type="hidden" value="{{ category.id }}" />
    <input name="_csrf" type="hidden" value="{{ csrftoken }}" />
    <input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

```

Index.handlebars

```

<p>Welcome to the admin panel...</p>

<ul>
    <li><a href="/admin/users">View Users</a></li>
    <li><a href="/admin/posts">View Posts</a></li>
    <li><a href="/admin/categories">View Categories</a></li>
</ul>

```

Login.handlebars

```

<form method="post" action="/admin/login">
    <label>Username</label> <input name="username" type="text" />
    <label>Password</label> <input name="password" type="password" />
    <input name="_csrf" type="hidden" value="{{ csrftoken }}" />
    <input type="submit" value="Submit" />
</form>

```

PostAdd.handlebars

```

<form method="post" action="/admin/post/add">
    <label>Title</label> <input name="title" type="text" class="tw-admin-post-title" />
    <label>Date</label> <input name="date" type="text" class="tw-admin-post-datepicker" />
    <label>Slug</label> <input name="slug" type="text" class="tw-admin-post-slug" />
    <a href="#" class="tw-admin-convert-slug">Convert Title to Slug</a>

    <label>Category</label>

    <select name="category" class="tw-admin-post-category">
        {{#categories}}
        <option value="{{_id}}" data-slug="{{ slug }}">{{name}}</option>
        {{/categories}}
    </select>

```

```

<label>Author</label>

<select name="author" class="tw-admin-post-author">
  {{#authors}}
    <option value="{{_id}}">{{name}}</option>
  {{/authors}}
</select>

<label>Path</label> <span class="tw-admin-post-path"></span>

<div class="clearout"></div>

<label>Content</label>

<div class="clear"></div>

<textarea name="content" class="tw-admin-editor"></textarea>
<input name="_csrf" type="hidden" value="{{ csrfToken }}" />
<input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

<script type="text/javascript">
  tinymce.init({
    selector: "textarea.tw-admin-editor",
    plugins: [
      "advlist autolink link image lists charmap print preview hr anchor pagebreak spellchecker",
      "searchreplace wordcount visualblocks visualchars code fullscreen insertdatetime media
nonbreaking",
      "save table contextmenu directionality emoticons template paste textcolor"
    ],
  });

  (function ($) {

    $(document).ready(function () {
      $(".tw-admin-post-datepicker").datepicker({ dateFormat: "yy/mm/dd" });

      $(".tw-admin-convert-slug").click(function (e) {
        e.preventDefault();
        $(".tw-admin-post-slug").val(Tradewinds.Admin.ConvertToSlug($(".tw-admin-post-
title").val()));
        Tradewinds.Admin.UpdatePath();
      });

      Tradewinds.Admin.UpdatePath();
      $(".tw-admin-post-category, .tw-admin-post-slug").change(function () {
        Tradewinds.Admin.UpdatePath();
      });

    });

  })(jQuery);
</script>

```

PostEdit.handlebars

```

<form method="post" action="/admin/post/edit">
  <label>Title</label> <input name="title" type="text" value="{{ post.title }}" class="tw-admin-post-
title" />
  <label>Date</label> <input name="date" type="text" value="{{ post.date }}" class="tw-admin-post-
datepicker" />
  <label>Slug</label> <input name="slug" type="text" value="{{ post.slug }}" class="tw-admin-post-slug"
/>
  <a href="#" class="tw-admin-convert-slug">Convert Title to Slug</a>

  <label>Category</label>

```

```

<select name="category" class="tw-admin-post-category">
  {{#categories}}
    <option value="{{_id}}" data-slug="{{ slug }}" {{#ifCond _id '==' ../post.category._id}}
selected{{/ifCond}}>{{name}}</option>
  {{/categories}}
</select>

<label>Author</label>

<select name="author" class="tw-admin-post-author">
  {{#authors}}
    <option value="{{_id}}" {{#ifCond _id '==' ../post.author._id}}
selected{{/ifCond}}>{{name}}</option>
  {{/authors}}
</select>

<label>Path</label> <span class="tw-admin-post-path"></span>

<div class="clearout"></div>

<label>Content</label>

<div class="clear"></div>

<textarea name="content" class="tw-admin-editor">{{ post.content }}</textarea>

<input name="id" type="hidden" value="{{ post.id }}" />
<input name="_csrf" type="hidden" value="{{ csrfToken }}" />
<input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

<script type="text/javascript">
  tinymce.init({
    selector: "textarea.tw-admin-editor",
    plugins: [
      "advlist autolink link image lists charmap print preview hr anchor pagebreak spellchecker",
      "searchreplace wordcount visualblocks visualchars code fullscreen insertdatetime media
nonbreaking",
      "save table contextmenu directionality emoticons template paste textcolor"
    ],
  });

  (function ($) {

    $(document).ready(function () {
      $(".tw-admin-post-datepicker").datepicker({ dateFormat: "yy/mm/dd" });

      $(' .tw-admin-convert-slug').click(function (e) {
        e.preventDefault();
        var text = Tradewinds.Admin.ConvertToSlug($(' .tw-admin-post-title').val());
        $(' .tw-admin-post-slug').val(text);
        Tradewinds.Admin.UpdatePath();
      });

      Tradewinds.Admin.UpdatePath();

      $(' .tw-admin-post-category, .tw-admin-post-slug').change(function () {
        Tradewinds.Admin.UpdatePath();
      });

    });

  })(jQuery);
</script>

```

PostsViewAll.handlebars

```

<ul>
  {{#posts}}
  <li>
    <strong>{{title}}</strong>
    <a href="{{ path }}" target="_blank">View Post</a>
    <a href="/admin/post/edit/{{ _id }}">Edit</a>
    <a href="/admin/post/delete/{{ _id }}" onclick="return confirm('Are you sure?')">Delete</a>
  </li>
  {{/posts}}
</ul>

<a href="/admin/post/add" class="tw-button-medium tw-button-black-flat">Add Post</a>

```

UserAdd.handlebars

```

<form method="post" action="/admin/user/add">
  <label>Name</label> <input name="name" type="text" />
  <label>Password</label> <input name="password" type="password" />
  <label>Confirm Password</label> <input name="password2" type="password" />
  <label>E-Mail</label> <input name="email" type="text" />
  <input name="_csrf" type="hidden" value="{{ csrfToken }}" />
  <input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

```

UserEdit.handlebars

```

<form method="post" action="/admin/user/edit">
  <label>Name</label> <input name="name" type="text" value="{{ user.name }}" />
  <label>E-Mail</label> <input name="email" type="text" value="{{ user.email }}" />
  <label>Avatar</label> <input name="avatar" type="text" value="{{ user.avatar }}" />
  {{#unless user.isDefault}}
  <label>Admin?</label> <input name="admin" type="checkbox" value="admin" {{#if user.isAdmin}}
checked{{/if}} />
  {{/unless}}
  <input name="id" type="hidden" value="{{ user.id }}" />
  <input name="_csrf" type="hidden" value="{{ csrfToken }}" />
  <input type="submit" value="Submit" class="tw-button-medium tw-button-black-flat" />
</form>

```

UsersViewAll.handlebars

```

<p>All of the users can be viewed below...</p>

<ul>
  {{#users}}
  <li>
    <strong>Name:</strong> {{name}} <strong>Email:</strong> {{email}}

    <a href="/admin/user/edit/{{ _id }}">Edit</a>

    {{#unless isDefault}}
    <a href="/admin/user/delete/{{ _id }}" onclick="return confirm('Are you sure?')">Delete</a>
    {{/unless}}
  </li>
  {{/users}}
</ul>

<a href="/admin/user/add" class="tw-button-medium tw-button-black-flat">Add User</a>

```

Whew! There is a lot of code there. Hopefully most of it makes sense just by the naming of the various methods. We will not go through each method in detail, but it basically follows the same patterns we have been using for all of our application creation. There is not really anything that is too much different.

Installation Routes & Creating a Default User

So now that we have our admin panel all set, we still have the hurdle of needing to create a "default user" so we can have at least one user that can actually sign in to our application. The best time to do this is when we give the user a way to install our application. This is one of the first thing that you do when you're installing any software that uses logins of any type. If you have ever installed a CMS such as [WordPress](#) you'll likely remember that one of the first things that you do is create a default user. Even your OS (Windows, Linux, and Mac) walks you through this process. It's a very common component of software installs.

So let's update our "router.js" file by adding the following installation routes...

```
// Installation Routes

app.get('/install', InstallController.Index);
app.post('/install', InstallController.Install);
app.get('/install/success', InstallController.InstallSuccess);
```

And of course as we have been doing let's create an "InstallController.js" file in our "controllers" directory with the following code in it...

```
var Validation = require('../utilities/Validation');
var Model = require('../models/Models');
var bcrypt = require('bcrypt-nodejs');

exports.Index = function (request, response) {
  Model.UserModel.findOne({ isDefault: true }, function (error, result) {
    if (!result || error) {
      response.pageInfo.title = 'Install';
      response.pageInfo.installed = false;
      response.render('admin/Install', response.pageInfo);
    }
    else {
      response.redirect('/install/success');
    }
  });
};

exports.InstallSuccess = function (request, response) {
  response.pageInfo.title = 'Install';
  response.pageInfo.installed = true;
  response.render('admin/Install', response.pageInfo);
};

exports.Install = function (request, response) {
  var errors = false;

  var name = request.body.name;
  var email = request.body.email;
  var password = request.body.password;
  var password2 = request.body.password2;

  if (Validation.IsNullOrEmpty([name, email, password, password2])) {
    errors = true;
  }

  if (!Validation.ValidateEmail(email)) {
    errors = true;
  }

  if (!Validation.Equals(password, password2)) {
    errors = true;
  }
}
```



```

    }

    if (errors) {
        Validation.ErrorRedirect(response, '/install', 'installError');
    }
    else {
        var salt = bcrypt.genSaltSync(10);
        var passwordHash = bcrypt.hashSync(request.body.password, salt);

        var u = new Model.UserModel({
            name: request.body.name,
            password: passwordHash,
            email: request.body.email,
            avatar: 'placeholder.png',
            isAdmin: true,
            isDefault: true
        });

        var c = new Model.CategoryModel({
            name: "--",
            slug: "",
            isDefault: true
        });

        u.save(function (error) {
            if (error) {
                Validation.ErrorRedirect(response, '/install', 'installError');
            }
            else {
                c.save(function (error) {
                    if (error) {
                        Validation.ErrorRedirect(response, '/install', 'installError');
                    }

                    Validation.SuccessRedirect(response, '/install', 'installSuccess');
                });
            }
        });
    }
};

```

As we can see above we are using a view in the "admin" directory of the "views" folder. Since there is really only one "install" view, we will just use the admin panel folder. It makes sense because there is a little bit of cross-over between admin functionality and installation.

So create an "Install.handlebars" file in the "views/admin" directory and add the following code to it...

```

<p>Welcome to {{ title }} the installation setup...</p>

{{#if installed}}
<p>You have installed Tradewinds.</p>

<p>Click <a href="/admin/login">here</a> to login to the admin panel.</p>
{{else}}
<form method="post" action="/install">
    <label>Default User Name</label>
    <input name="name" type="text" />
    <label>Default User Password</label>
    <input name="password" type="password" />
    <label>Confirm Default User Password</label>
    <input name="password2" type="password" />
    <label>Default User Email</label>
    <input name="email" type="text" />
    <input name="_csrf" type="hidden" value="{{ csrfToken }}" />

```

```

    <input type="submit" value="Submit" />
  </form>

  {{/if}}

```

Dynamic Page & Post Routes

For our application pages and blog posts are essentially the same "post" data type. They can be categorized under different categories but ultimately they are all of the same type.

In "router.js" file update the following code to add a dynamic post controller using a "*" wildcard after all the other routes such that the "router.js" file looks like the following...

```

var HomeController = require('./controllers/HomeController');
var AdminController = require('./controllers/AdminController');
var PostController = require('./controllers/PostController');
var InstallController = require('./controllers/InstallController');
var Authentication = require('./utilities/Authentication');

// Routes
module.exports = function (app) {

  // Main Routes

  app.get('/', HomeController.Index);
  app.get('/other', HomeController.Other);

  // Admin Routes

  app.get('/admin', Authentication.AuthenticateAdmin, AdminController.Index);
  app.get('/admin/login', AdminController.Login);
  app.post('/admin/login', AdminController.VerifyLogin);
  app.get('/admin/logout', AdminController.Logout);
  app.get('/admin/users', Authentication.AuthenticateAdmin, AdminController.UsersViewAll);
  app.get('/admin/user/add', Authentication.AuthenticateAdmin, AdminController.UserAdd);
  app.post('/admin/user/add', Authentication.AuthenticateAdmin, AdminController.UserCreate);
  app.get('/admin/user/edit/:id', Authentication.AuthenticateAdmin, AdminController.UserEdit);
  app.post('/admin/user/edit', Authentication.AuthenticateAdmin, AdminController.UserUpdate);
  app.get('/admin/user/delete/:id', Authentication.AuthenticateAdmin, AdminController.UserDelete);

  app.get('/admin/posts', Authentication.AuthenticateAdmin, AdminController.PostsViewAll);
  app.get('/admin/post/add', Authentication.AuthenticateAdmin, AdminController.PostAdd);
  app.post('/admin/post/add', Authentication.AuthenticateAdmin, AdminController.PostCreate);
  app.get('/admin/post/edit/:id', Authentication.AuthenticateAdmin, AdminController.PostEdit);
  app.post('/admin/post/edit', Authentication.AuthenticateAdmin, AdminController.PostUpdate);
  app.get('/admin/post/delete/:id', Authentication.AuthenticateAdmin, AdminController.PostDelete);

  app.get('/admin/categories', Authentication.AuthenticateAdmin, AdminController.CategoriesViewAll);
  app.get('/admin/category/add', Authentication.AuthenticateAdmin, AdminController.CategoryAdd);
  app.post('/admin/category/add', Authentication.AuthenticateAdmin, AdminController.CategoryCreate);
  app.get('/admin/category/edit/:id', Authentication.AuthenticateAdmin, AdminController.CategoryEdit);
  app.post('/admin/category/edit', Authentication.AuthenticateAdmin, AdminController.CategoryUpdate);
  app.get('/admin/category/delete/:id', Authentication.AuthenticateAdmin,
AdminController.CategoryDelete);

  // Installation Routes

  app.get('/install', InstallController.Index);
  app.post('/install', InstallController.Install);
  app.get('/install/success', InstallController.InstallSuccess);

  app.get('/*', PostController.Process);
};

```

The "*" is a wildcard basically what this means is that if the requested route is anything other than our "reserved routes" -- the home, admin, or installation routes that precede it -- we should run this other route in our yet to be created "PostController.js" file. This route in essence becomes a route for "anything other" than our predefined routes. Thus if we have a requested route of something like "/books/a-page-about-a-book" we can run the code in this controller to dynamically look for this combination in the URL in our database and return the resulting markup accordingly. Let's take a look and see how this looks in practice.

So let's create our "PostController.js" file and put it in the "controllers" directory. In it put the following code...

```
var Helpers = require('../utilities/Helpers');
var Validation = require('../utilities/Validation');
var Model = require('../models/Models');
var bcrypt = require('bcrypt-nodejs');

// Posts - Process

exports.Process = function (request, response, next) {

  var path = request.url.substring(1);

  Model.PostModel.findOne({ path: path }).exec(function (error, result) {

    if (error) {
      next();
    }
    else {
      if (result) {
        response.pageInfo.title = result.title;
        response.pageInfo.content = result.content;
        response.render('home/Post', response.pageInfo);
      }
      else {
        next();
      }
    }
  });
};
```

So as we can see we look up the "path" in the database (one of the properties on our post data model). The path is a combination of both the category and the slug stored on the post. If we find an entry with the correct URL path in the database we will get the data for this post and return it to the view. If we do not find the right path in the requested URL we will call the next() middleware function which will cause the executing code to move to the next line down in our "router.js" file which will be an "error handler" in our code. We will add this in the next section.

We still have to create our view for this post controller route. We can put this in the "home" views directory. Create a view called "Post.handlebars" and put it in the "home" folder in the "views" directory. If you wanted to create a separate "post" directory for this view in the views folder you could. You would have to change the "home/post" path in the code above to "post/index" or "post/post" or whatever you wanted to call it.

Wherever you decide to put it add the following to this file...

```
<p>{{{ content }}}</p>
```

Seems like a pretty basic view, right? Basically all that this post controller handler method does is it gets the content that is entered into the TinyMCE text editor and displays it as it is in the page. So what ends up being displayed might be a bit more interesting than just unformatted text. We will delegate that responsibility to the rich text editor.

404 & Error Routes

Lastly of course we will need to have a way to handle errors and/or have a 404 view if a client request coming into our server does not match any of the predefined routes in our "router.js" file or does not match any of the paths

found on our post data types stored in the database. So we need to add some middleware to handle this. We could create a new file for this, but it is also possible to just do this inline in our "router.js" file. So add the following after all the other routes in our "router.js" file...

```
// Errors
// 404

app.use(function (request, response) {
  response.status(404);
  response.render('errors/404', {
    title: '404',
    message: 'The requested resource was not found...',
  });
});
```

Now as we can see from this, we have to create a view for this route. So to do this create a folder in the views directory called "errors" and create a "404.handlebars file" and put the following code in it...

```
<p>{{ message }}</p>
```

Simple enough, right? If we wanted to add more handlers for different types of we could. We would just need to add additional middleware handlers that would return a different HTTP status code (e.g. 401, 403, 500, etc.) and expand upon the approach above.

Summary

In the preceding paragraphs we did a very quick walkthrough on how to create a functioning content management system (CMS) application. It may not have all the features of a commercial or production application, but it is a start and it could be expanded upon and enhanced to make something that is a bit more sophisticated. But as we have seen, the approaches we have taken towards creating a Node.js application with Express.js and MongoDB can be used to create any type of application that we like... big or small, simple or complex. Once you kind of learn how to make your way around in using these technologies, the sky is the limit and the possibilities are endless when it comes to what you can create!

Conclusion

Alas, like all good things our journey must come to the end. BUT, this is merely in reference to the end of this volume. If you have stuck through everything this far, hopefully you have learned one or two things along the way. Like with many things in technology, something can take minutes to learn, but years or decades to master. But, as Confucius supposedly said, "The journey of 1,000 miles begins with a single step" and it is my sincere hope that this discussion and tutorial of Node.js, Express.js, and MongoDB allowed you to take a half-decent first step.

I think that the only place to go from here is to go build an application on your own. That is how you learn and that's how you develop your skills in solving different types of problems.

We created our Trade Winds CMS available [here on GitHub](#). Feel free to clone the repo, modify and utilize it however you like. Also see Appendix A for links to code samples for each of the previously discussed chapters.

With that, I think that is just about it. Here is wishing you all the best and future success on your future travels and endeavors into this exciting world of new technologies. Go and build great things!

Appendix A: Code Samples & Repos

A link to code samples for each chapter can be found [here](#).

Our content management system application "Trade Winds" which is written in the same style as the tutorials in this book is available [here on GitHub](#)...

Appendix B: MongoDB Shell

This appendix covers database administration with MongoDB through the shell (command line). See this documentation [here](#) on the MongoDB website for information on using the MongoDB shell. There is nothing about the shell that is particularly special, it's just a way for us to administer the database via the command line console instead of a web interface. We can run queries, insert records, and make any number of other changes. So as we did before in working with MongoDB we need to start the database up if it is not already. Navigate to the directory where you installed MongoDB and go into the "bin" directory. (e.g. C:\mongodb\bin) and open a shell and run "mongod" and point it at the database we created earlier...

```
mongod --dbpath "C:\path\to\application\db"
```

or if you're using BASH, something like this...

```
$ mongod --dbpath /c/path/to/application/db
```

Now instead of opening up a browser we're going to open a new command window or shell in the MongoDB "bin" directory (same directory that we just used to run mongod) and type the following

```
mongo
```

This should bring up the prompt and say that you are connecting to a test database. MongoDB does this by default. The first thing we need to do after this is switch to our MVCApplication database by typing the following (case-sensitive)...

```
> use MVCApplication
```

This will switch to our MVCApplication database. We can type anything after the "use" keyword and the context database will automatically be switched (even if this database does not exist yet). After this type the following...

```
> db.Users.insert({ name:"temp", password: "temp", email: "temp@temp.com", isAdmin: false })
```

What we are doing here with this command is creating the "users" collection -- because it does not exist yet -- and inserting a user with name "temp" and password "temp" into the collection. You can confirm that this item was inserted into the database by typing...

```
> db.Users.find()
```

And you should see the record displayed. You could add as many other users as you want. Note: you can also run this with other data objects as well. For example, you could run a db.Books.find() to view the books that you inserted earlier if you added any books from earlier on in this tutorial. The data for each insert does not even need to be consistent or match a schema. This flexibility is one of the things that people really like about MongoDB.