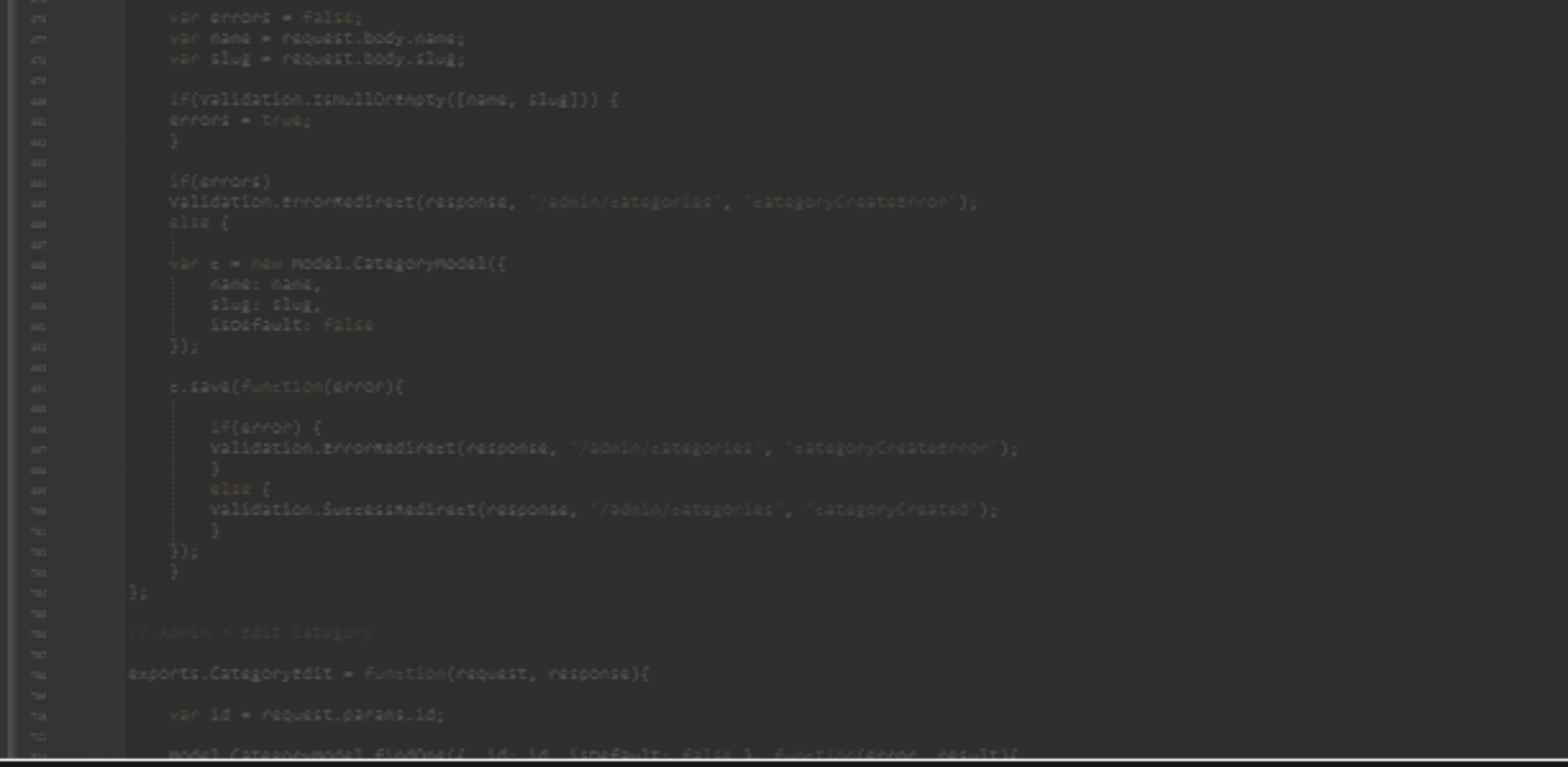
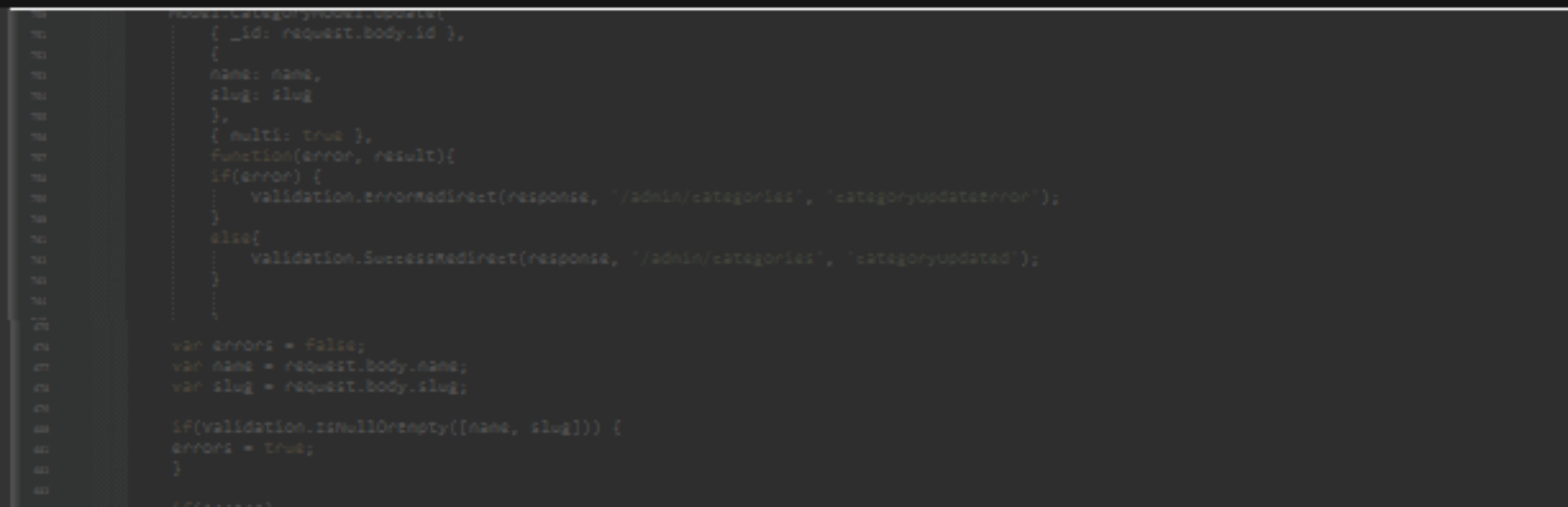


Programming Node.js Applications

with Express.js & MongoDB

Ian Chursky



Programming Node.js Applications with Express.js and MongoDB

Ian Chursky

Contents

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 and @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 Rasberry 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 it's 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 couple 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 pakcage.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 or 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 handing 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 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 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 looks 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.