

M E A N MACHINE

The Beginner's Guide
to the
JavaScript Stack



MEAN Machine

A beginner's practical guide to the JavaScript stack.

Chris Sevilleja and Holly Lloyd

This book is for sale at <http://leanpub.com/mean-machine>

This version was published on 2015-03-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Chris Sevilleja and Holly Lloyd

Tweet This Book!

Please help Chris Sevilleja and Holly Lloyd by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I'm learning about #nodeJS and #angularJS by reading #MEANmachine!
<https://leanpub.com/mean-machine>

The suggested hashtag for this book is [#MEANmachine](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#MEANmachine>

Contents

Preface	1
Conventions Used in This Book	1
Code Samples	2
Get In Contact	2
Introduction	3
Why MEAN?	3
When To Use the MEAN Stack	3
When NOT To Use the MEAN Stack	4
Who's Getting MEAN?	4
Primers	6
MongoDB	6
Node.js	8
ExpressJS	11
AngularJS	12
MEAN Thinking	15
Client-Server Model	15
Book Outline	16
Getting Started and Installation	17
Requirements	17
Tools	17
Installation	17
Starting Node	21
Configuration (package.json)	21
Initialize Node App	22
Creating a Very Simple Node App	23
Starting a Node Application	24
Packages	26
Recap	28
Starting a Node Server	29

CONTENTS

Sample Application	29
Method #1: Pure Node (no Express)	30
Method #2: Using Express	33

Preface

Conventions Used in This Book



Note

This icon signifies a tip, suggestion, or general note.



Warning

This icon indicates a warning or caution.



Tip

This icon indicates a pro tip that will help your development.

Code Samples

This book will mix in concept and code by building applications in each chapter. Each application will be useful in understanding the core concepts and building up to a fully fledged MEAN stack application.

Throughout the chapters, we will work with code samples that build off of each other, leading up to one full application. We'll add links to the sample code so that you can download and follow along if you wish. After seeing real examples and concepts in action, you'll be able to use these concepts to build your very own projects.

Code License

The sample code in this book is released under the [MIT License](http://opensource.org/licenses/MIT)¹. Feel free to use any and all parts of them in your own applications and anything you build or write.

Code Repository

The code for the samples in this book can be found at: <http://github.com/scotch-io/mean-machine-code>².

We'll provide links to the specific folders at the start of every application so stay tuned for those.

Get In Contact

If you have any questions, comments, kind words about the book (we love those), or corrections in the book (we like those), feel free to contact us at chris@scotch.io³ and holly@scotch.io⁴.

Also, take a look at our site ([Scotch.io](http://scotch.io)⁵) for great articles on all sorts of web development topics.

¹<http://opensource.org/licenses/MIT>

²<http://github.com/scotch-io/mean-machine-code>

³<mailto:chris@scotch.io>

⁴<mailto:holly@scotch.io>

⁵<http://scotch.io>

Introduction

Node is an exciting JavaScript language for web development that has been growing in popularity in recent years. It started out for small development projects and has since penetrated the enterprise and can be seen in large companies like Microsoft, eBay, LinkedIn, Yahoo, WalMart, Uber, Oracle, and several more.

Why MEAN?

The MEAN stack uses four pieces of software: MongoDB, ExpressJS, AngularJS, and NodeJS. Using these four tools together lets developers create efficient, well organized, and interactive applications quickly.

Since every component of the stack uses JavaScript, you can glide through your web development code seamlessly. Using all JavaScript lets us do some great things like:

- Use JavaScript on the server-side (Node and Express)
- Use JavaScript on the client-side (Angular)
- Store JSON objects in MongoDB
- Use JSON objects to transfer data easily from database to server to client

A single language across your entire stack increases productivity. Even client side developers that work in Angular can easily understand most of the code on the server side.

Starting with the database, we store information in a JSON like format. We can then write JSON queries on our Node server and send this directly to our front-end using Angular. This is especially useful when you have multiple developers working on a project together. Server-side code becomes more readable to front-end developers and vice versa. This makes everything a little more transparent and has been shown to greatly increase development time. The ease of development will become much more apparent once we start digging into examples and hopefully save you and your team some headaches in the future.

When To Use the MEAN Stack

The MEAN stack benefits greatly from the strengths of Node. Node let's us build real-time open APIs that we can consume and use with our frontend Angular code. Transferring data for applications like chat systems, status updates, or almost any other scenario that requires quick display of real-time data.

- Chat client
- Real-time user updates (like Twitter feed)
- RSS feed
- Online shop
- Polling app

When NOT To Use the MEAN Stack

As with any language or set of languages, there are plenty of scenarios where MEAN wouldn't be the best fit and it's very important to recognize this before diving into coding. A lot of the benefits of the MEAN stack and reasons why you would use it are rooted in its use of Node. We see this same trend again with reasons you may not want to use it.

Node is generally not the best pick for CPU intensive tasks. There have been a few [arguments](#)⁶ for cases where Node actually did well in computationally heavy applications, but for the novice it's best to steer away from Node if you know your application requires a lot of computing (in other words let's not try to calculate the 1000th prime number here).

Who's Getting MEAN?

Many developers have shouted their praise for the MEAN stack. This stack uses JavaScript for every operation, which makes it appealing to developers who want to flow smoothly through a project. Some large companies are already reaping the benefits and have integrated Node into many of their operations.



Walmart

Walmart: Walmart began using Node.js in 2012 to provide mobile users with a modern front end experience. Making use of the JavaScript platform, they were able to quickly and easily integrate their existing APIs with their Node application. They also stated that 53% of their Black Friday online traffic went to their Node servers with zero downtime.

⁶<http://neilk.net/blog/2013/04/30/why-you-should-use-nodejs-for-CPU-bound-tasks/>



Yahoo

Yahoo!: Yahoo started experimenting with Node back in 2010. At first they just used it for small things like file uploads, and now they use Node to handle nearly 2 million requests per minute. They have noted increases in speed and a simpler development process.



Linkedin

LinkedIn LinkedIn began developing the server side of their mobile app entirely with Node. They were previously using Ruby, but since the switch they have seen huge increases in performance, ranging from 2 to 10 times faster.



Paypal

PayPal: PayPal has recently jumped onboard and began migrating some of their Java code to Node. They began experimenting with just their Account Overview page, but once they saw a 35% speed increase and half the amount of time spent on development, they started moving all sites to Node.js.

For a larger and maintained list, visit the [Node Industry](http://nodejs.org/industry/)⁷ page.

⁷<http://nodejs.org/industry/>

Primers

Let's take a quick look at the technologies we'll be using. Remember, this book is meant to teach you how all these pieces work together, so we won't be diving into the most advanced techniques and concepts of each (though we will be going pretty far). We will provide links to more resources to further your knowledge for each topic in each section.

MongoDB

MongoDB, named from the word “humongous”, is an open-source NoSQL database that stores documents in JSON-style format. It is the [leading NoSQL database](http://www.mongodb.com/leading-nosql-database)⁸ based on the number of Google searches, job postings and job site (Indeed.com) trends.

Mongo vs. MySQL

LAMP, which uses MySQL, has been the leading stack for several years now. MySQL is classified as a **relational database**.

Relational database Data is stored in tables that hold not only the data, but also its relationship to other information in the database. —

Document Databases

MongoDB, on the other hand, is classified as a non-relational database, or more specifically a **document-oriented database**. This means that you define your data structure however you want. You get the data-modeling options to match your application and its performance requirements. You can easily take complex objects and insert them into your database using JSON, XML, BSON, or many other similar formats that are better suited to your application. You can even store PDF documents in certain document databases if the use case ever arises.

Document-oriented Database A type of NoSQL database which stores and retrieves data in a semi- structured document (as opposed to tables in relational databases). —

⁸<http://www.mongodb.com/leading-nosql-database>

Data modeling in MongoDB is extremely flexible. One way to store data is by creating separate documents and creating references to connect information. You can see below our Elf Info Document contains basic information which we can reference with the Elf Address Document.

Elf Info json { id: "1234", name: "holly", age: "400", type: "high-elf" }

Elf Address Book json { elven_id: "1234", city: "rivendell", state: "middle-earth" }

Another method is to embed the Elf Address straight into the Elf Info document so that now you only have one document for each Elf, which allows for less write operations to update information about an Elf as well as faster performance (assuming your document doesn't grow too large).

```
1  {
2    id: "1234",
3    name: "holly",
4    age: "400",
5    type: "high-elf",
6    address: {
7      city: "rivendell",
8      state: "middle-earth"
9    }
10 }
```

Once you begin storing information about multiple elves, each of their documents can be classified together as one Elf Info Collection. A collection is just a group of related documents. In this case they will all have the same fields with different values.

CAP Theorem There is a concept known as CAP or Brewer's theorem which states that distributed network systems can only provide two of the three following functionalities at the same time:

- Consistency: All nodes in your application are available to each other and showing the same data at the same time
- Availability: All nodes are available to read and write
- Partition Tolerance: Even if part of the system fails, your application will continue to operate

There has been some confusion about how to interpret this, so Brewer later pointed out that it really comes down to **consistency** vs. **availability**. Just looking at the definitions alone, it becomes apparent that these are mutually exclusive concepts. Let's say you allow all nodes to be **available** for reading and writing at all times. A real world example of where this would be important is in a banking application. For the sake of example, let's pretend overdraft fees and all that fun stuff doesn't exist. The bank needs to know your exact account balance at all times. If you have \$100 in your account and you withdraw \$100, you're of course left with \$0. Because all nodes are available to read and write, there's a chance that another debit could go through in the tiny fraction of time that your account balance still reads \$100. So then once the account balance updates you will actually

be in the negative because the second debit was allowed through even though it should have been denied. This is why **consistency** is more important in the case of this banking application.

The cost of consistency is giving up some availability. As soon as you withdrew money in the previous example, there should have been some locks put into place preventing the account balance to be read as \$100. If your application is unable to read every node at every second, then it may appear down or unavailable to some users. Some applications favor availability and performance over consistency, which is where a lot of document-oriented databases shine.

MongoDB, by default, favors consistency over availability, but still allows you to tweak some settings to give you more support in either direction. Read-write locks are scoped to each database, so each node within a database will always see the most up-to-date data. Because MongoDB supports **sharding**, once your database begins to grow, your data may partition into multiple databases (or shards) across several servers. Each shard will be an independent database, together forming one collection of databases. This allows for faster queries because you only need to access the shard that contains that information rather than the entire database. It can also, however, cause inconsistency from shard to shard for a short period of time after a write. This is called eventual consistency and is a common compromise between consistency and availability.

Main Features

- Agile and Scalable
- [Document-Oriented Storage](#)⁹ - JSON-style documents with dynamic schemas offer simplicity and power.
- [Full Index Support](#)¹⁰ - Index on any attribute, just like you're used to.
- [Replication & High Availability](#)¹¹ - Mirror across LANs and WANs for scale and peace of mind.
- [Auto-Sharding](#)¹² - Scale horizontally without compromising functionality.
- [Querying](#)¹³ - Rich, document-based queries.
- [Fast In-Place Updates](#)¹⁴ - Atomic modifiers for contention-free performance.
- [Map/Reduce](#)¹⁵ - Flexible aggregation and data processing.
- [GridFS](#)¹⁶ - Store files of any size without complicating your stack.

Node.js

Node is built on Google Chrome's V8 JavaScript runtime and sits as the server-side platform in your MEAN application. So what does that mean? In a LAMP stack you have your web server (Apache,

⁹<http://docs.mongodb.org/manual/core/data-modeling/>

¹⁰<http://docs.mongodb.org/manual/indexes/>

¹¹<http://docs.mongodb.org/manual/replication/>

¹²<http://docs.mongodb.org/manual/sharding/>

¹³<http://docs.mongodb.org/manual/applications/read/>

¹⁴<http://docs.mongodb.org/manual/applications/update/>

¹⁵<http://docs.mongodb.org/manual/applications/map-reduce/>

¹⁶<http://docs.mongodb.org/manual/applications/gridfs/>

Nginx, etc.) running with the server-side scripting language (PHP, Perl, Python) to create a dynamic website. The server-side code is used to create the application environment by extracting data from the database (MYSQL) and is then interpreted by the web server to produce the web page.

When a new connection is requested, Apache creates a new thread or process to handle that request, which makes it **multithreaded**. Often you will have a number of idle child processes standing by waiting to be assigned to a new request. If you configure your server to only have 50 idle processes and 100 requests come in, some users may experience a connection timeout until some of those processes are freed up. Of course there are several ways to handle this scalability more efficiently, but in general Apache will use one thread per request, so to support more and more users you will eventually need more and more servers.

Multithreading A programming model where the flow of the program has multiple threads of control. Different parts of the program (threads) will be able to execute simultaneously and independently, rather than waiting for each event to finish. This is nice for the user who now doesn't have to wait for every event to finish before they get to see some action, but new threads require more memory, so this performance comes with a memory trade-off.

—

This is where Node.js shines. Node is an **event driven** language that plays the same role as Apache. It will interpret the client-side code to produce the web page. They are similar in that each new connection fires a new event, but the main distinction comes from the fact that Node is asynchronous and single threaded. Instead of using multiple threads that sit around waiting for a function or event to finish executing, Node uses only one thread to handle all requests. Although this may seem inefficient at first glance, it actually works out well given the **asynchronous** nature of Node.

Event Driven Programming The flow of a program is driven by specific events (mouse clicks, incoming messages, key presses, etc). Most GUIs are event based and this programming technique can be implemented in any language. —

Asynchronous Programming Asynchronous events are executed independently of the main program's "flow". Rather than doing nothing while waiting for an event to occur, the program will pass an event to the event handler queue and continue with the main program flow. Once the event is ready, the program will return to it with a callback, execute the code, and then return to the main flow of the program. Because of this, an asynchronous program will most likely not run in the normal top to bottom order that you see with synchronous code. —

Say, for example, a database query request comes in. Depending on how large the query is, it could take a couple of seconds to return anything. Since there is only one thread, it may seem that nothing else would be able to process while the query is executing. This would of course result in slow load times for your users which could be detrimental to the success of your site. Fortunately Node handles multiple requests much more gracefully than that by using **callbacks**.

Asynchronous Callbacks A function that is passed as an argument to be executed at a later time (when it is ready). Callbacks are used when a function may need more time to execute in order to return the correct return values. —

There are two types of callbacks: synchronous and asynchronous. Synchronous callbacks are considered “blocking” callbacks, meaning that your program will not continue running until the callback function is finished executing. Because I/O operations take a great deal of time to execute, this may make your application appear to be slow or even frozen to users. Node’s use of asynchronous callbacks (also known as non-blocking callbacks) allow your program to continue executing while I/O operations are taking place. Once the operations are complete, they will issue an interrupt/callback to tell your program that its ready to execute. Once the function is complete, your program will return back to what it was doing. Of course having several callbacks throughout your code can get very chaotic very fast, so it’s up to you, the programmer, to make sure that you do it correctly.

NPM and Packages

One of the benefits of Node is its package manager, [npm](https://www.npmjs.org/)¹⁷. Like Ruby has RubyGems and PHP has Composer, Node has npm. npm comes bundled with Node and will let us pull in a number of packages to fulfill our needs.

Packages can extend functionality in Node and this package system is one thing that makes Node so powerful. The ability to have a set of code that you can reuse across all your projects is incredible and makes development that much easier.

Multiple packages can be brought together and intertwined to create a number of complex applications.

Listed below are a few of the many [popular packages](#)¹⁸ that are used in Node:

- [ExpressJS](http://expressjs.com/)¹⁹ is currently the most starred package on [npm’s site](https://www.npmjs.org/browse/star)²⁰ (we’ll use this in the book of course)

¹⁷<https://www.npmjs.org/>

¹⁸<https://www.npmjs.org/browse/star>

¹⁹<http://expressjs.com/>

²⁰<https://www.npmjs.org/browse/star>

- [Mongoose](#)²¹ is the package we will use to interact with MongoDB.
- [GruntJS](#)²² for automating tasks (we'll use this later in the book)
- [PassportJS](#)²³ for authentication with many social services.
- [Socket.io](#)²⁴ for building real time websocket applications (we'll use this later in the book)
- [Elasticsearch](#)²⁵ for providing high scalability search operations.

Frameworks

There are several Node frameworks in existence. We're using Express in this book, but the concepts taught here will easily transfer over to other popular frameworks.

The other main frameworks of interest are:

- [HapiJS](#)²⁶ - Great framework being used by more and more enterprise companies.
- [KoaJS](#)²⁷ - A fork of Express
- [Restify](#)²⁸ - Borrows from Express syntax to create a framework devoted to building REST APIs
- [Sails](#)²⁹ - Framework built to emulate the MVC model

Express will handle nearly all of the tasks that you need and it is extremely robust, usable, and now has commercial backing as it was recently bought/sponsored by [StrongLoop](#)³⁰.

While those other frameworks are great in their own rights (definitely take a look at them), we will be focusing on Express. It is the **MEAN** stack after all.

ExpressJS

Express is a lightweight platform for building web apps using NodeJS. It helps organize web apps on the server side. The [ExpressJS website](#)³¹ describes Express as “a minimal and flexible node.js web application framework”.

Express hides a lot of the inner workings of Node, which allows you to dive into your application code and get things up and running a lot faster. It's fairly easy to learn and still gives you a bit of flexibility with its structure. There is a reason it is currently the most popular framework for Node. Some of the big names using Express are:

²¹<http://mongoosejs.com/>

²²<http://gruntjs.com/>

²³<http://passportjs.org/>

²⁴<http://socket.io/>

²⁵<http://www.elasticsearch.com/>

²⁶<http://hapijs.com/>

²⁷<http://koa.js.com/>

²⁸<http://mcavage.me/node-restify/>

²⁹<http://sailsjs.org/#/>

³⁰<http://strongloop.com/strongblog/tj-holowaychuk-sponsorship-of-express/>

³¹<http://expressjs.com>

- MySpace
- LinkedIn
- Klout
- Segment.io

For a full list of Express users, visit the [Express list](#)³².

Express comes with several great features that will add ease to your Node development.

- Router
- Handling Requests
- Application Settings
- Middleware

Don't worry if these terms are new to you. As we build our sample applications, we'll dive into each of these components, learn about them, and use them. Onto the last part of the MEAN stack (probably my favorite part of the stack)!

AngularJS

Angular, created by Google, is a JavaScript framework built for fast and dynamic front-end deployment.

Angular allows you to build your normal HTML application and then extend your markup to create dynamic components. If you've ever made a dynamic web page without Angular, you've probably noticed some of the common complications, such as data binding, form validation, DOM event handling, and much more. Angular introduces an all-in-one solution to these problems.

For those of you worried about learning so much at once, you're in luck. The learning curve for Angular is actually quite small, which may explain why its adoption has skyrocketed. The syntax is simple and its main principles like data-binding and dependency injection are easy to grasp with just a few examples, which of course will be covered in this book.

Two of the major features of Angular are data binding and dependency injection. Data binding deals with how we handle data in our applications while dependency injection deals more with how we architect them.

³²<http://expressjs.com/applications.html>

Data Binding

If you come from the land of jQuery, you will be familiar with using your CSS selectors to traverse the DOM. Every time you need to grab a value from an input box, you use `$('#input').val();`. This is great and all, but when you have large applications with multiple input boxes, this becomes a little harder to manage.

When you're pulling and injecting data into different places in your application, there is no longer **one true source of data**. With Angular, you have something similar to the MVC (model-view-controller) model where your data is in one spot. When you need information, you can be sure that you are looking at the correct information. This is because if you change data in your view (HTML files) or in your controller (JavaScript files), **the data changes everywhere**.

Dependency Injection

An Angular application is a collection of several different modules that all come together to build your application. For example, your application may have a model to interact with a specific item in an API, a controller to hand data to our views, or a module to handle routing our Angular application.

Dependency Injection A dependency in your code occurs when one object depends on another. There are different degrees of dependency, but having too much of it can sometimes make it difficult to test your code or even make some processes run longer. Dependency injection is a method by which we can give an object the dependencies that it requires to run. —

Having compartmentalized and modular applications gives us many benefits. Like packages in Node and PHP or gems in Ruby, we can reuse modules across different projects and even pull in modules that other developers have already created.

By injecting these modules into our application, we can also test each module separately. We are able to determine what parts of our application are failing and narrow down the problem to a certain codeset.

Other Main Features

- MVC
- Directives
- Scopes
- Templates
- Testing

Check out this Tuts+ article for a more in depth overview of Angular: [5 Awesome AngularJS Features](http://code.tutsplus.com/tutorials/5-awesome-angularjs-features--net-25651)³³.

³³<http://code.tutsplus.com/tutorials/5-awesome-angularjs-features--net-25651>

MEAN Thinking

When building MEAN stack applications throughout our book, there's a certain way of thinking that we'll want to use. We're talking about the **client-server model**.

We are going to think of our application as two separate parts that handle specific tasks. The providers of a resource or service (servers/backend/Node) will handle the data layer and will provide information to our service requesters (clients/frontend/Angular).

Client-Server Model

When building this book, we will be thinking in the [client-server model]. This is a very important concept while we are building our applications and learning the MEAN stack.

Client-Server Model A network architecture in which one program, the client, requests service from another program, the server. They may reside on the same computer or communicate across a network. —

There are many benefits to thinking of your application as two separate parts. By having our server be its own entity with an API from which we can access all of our data, we provide a way to create a scalable application.

This sort of thinking doesn't just have to be exclusive to the MEAN stack either. There are numerous applications that would benefit from this type of architecture. Having the server-side code be separate lets us create multiple front-end client applications like **websites**, **Android apps**, **iPhone apps**, and **Windows apps** that all connect to the same data.

We can iterate on our server side code and this will not affect our frontend code. We also see **large companies like Facebook, Google, Twitter, and GitHub using this method**. They create the API and their frontend clients (website, mobile applications, and third party applications) integrate with it.

Fun Note: The practice of using your own API to build a frontend client is called **dogfooding**³⁴ (one of my absolute favorite words).

Here are the parts of our application separated as server and client.

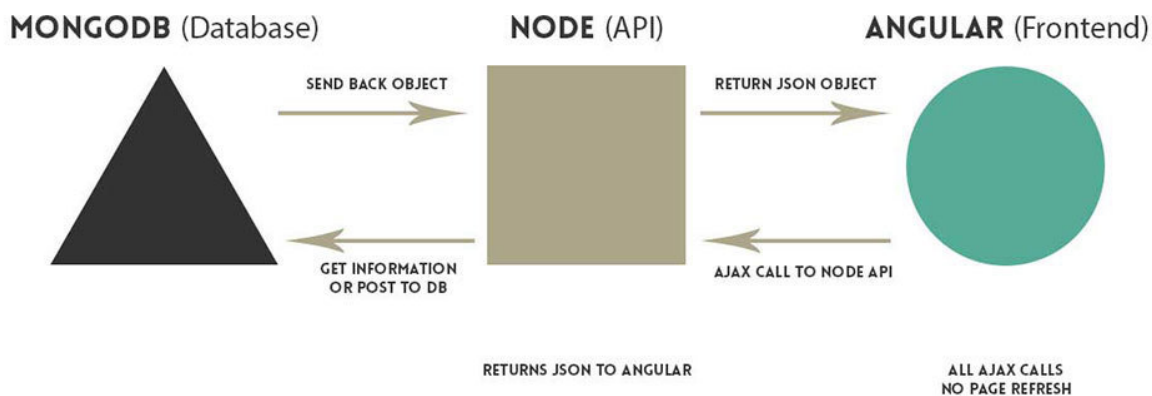
³⁴http://en.wikipedia.org/wiki/Eating_your_own_dog_food

Server Components

- Database (MongoDB)
- Server/API (Node and Express)

Client Components

- Frontend Layer (Angular)



Client to Server Model

Book Outline

When building MEAN applications, we are going to look at the duties of the server (Node) and the client (Angular) separately.

1. **Chapter 1-3: Getting Started** Setting up the tools we'll need for the book.
2. **Chapters 4-9: Server/Backend** Concepts, applications, and best practices.
3. **Chapters 10-16: Client/Frontend** Concepts, applications, and best practices.
4. **Chapters 17-19: MEAN Stack Applications** Bringing it all together so that we can build amazing applications.
5. **Chapters 20-Infinity: More Sample Applications!**

Getting Started and Installation

Requirements

- [Node](#)³⁵
- npm - included in Node installation

Tools

- Sublime Text
- Terminal - We like using [iterm2](#)³⁶ (mac/linux users)
- [Git Bash](#) + [ConEmu](#)³⁷ (windows users)
- [Postman](#)³⁸ (Chrome)
- [RESTClient](#)³⁹ (Firefox)

Installation

Go ahead and visit the [node website](#)⁴⁰ and download Node. Run through the installation and you'll have Node and npm installed! That will probably be the easiest part of the book.

³⁵<http://nodejs.org>

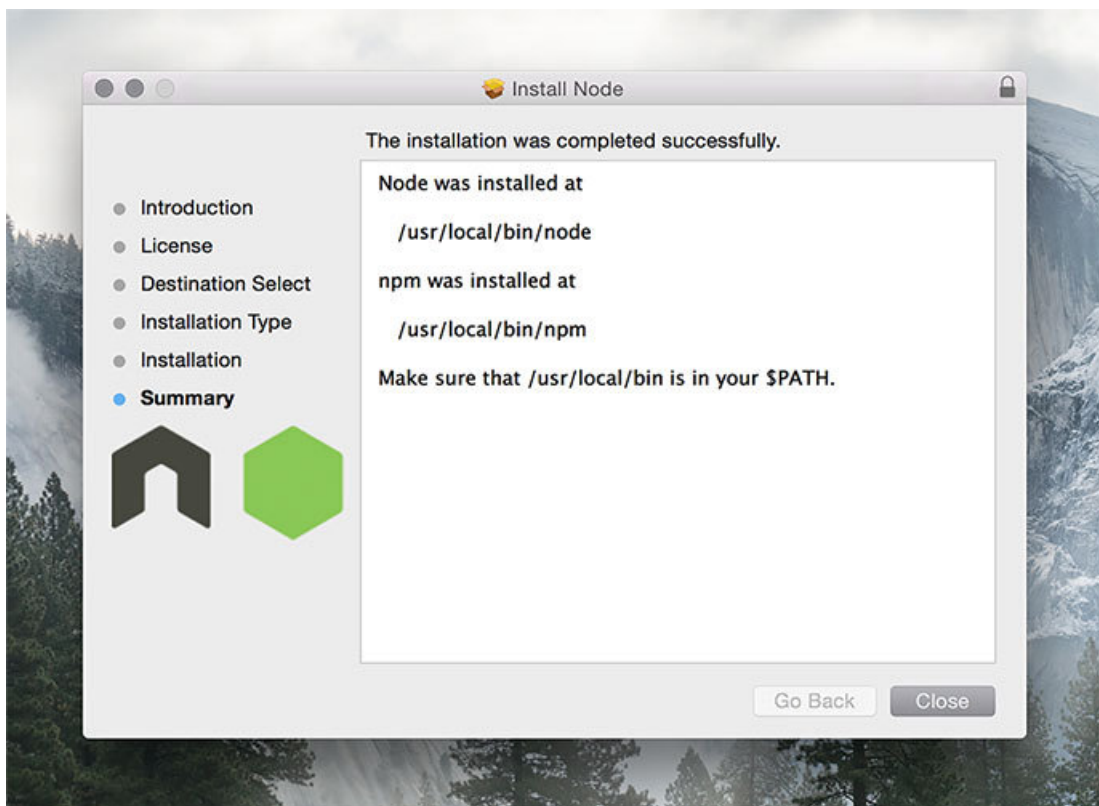
³⁶<http://iterm2.com/>

³⁷<http://scotch.io/bar-talk/get-a-functional-and-sleek-console-in-windows>

³⁸<https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en>

³⁹<https://addons.mozilla.org/en-US/firefox/addon/restclient/>

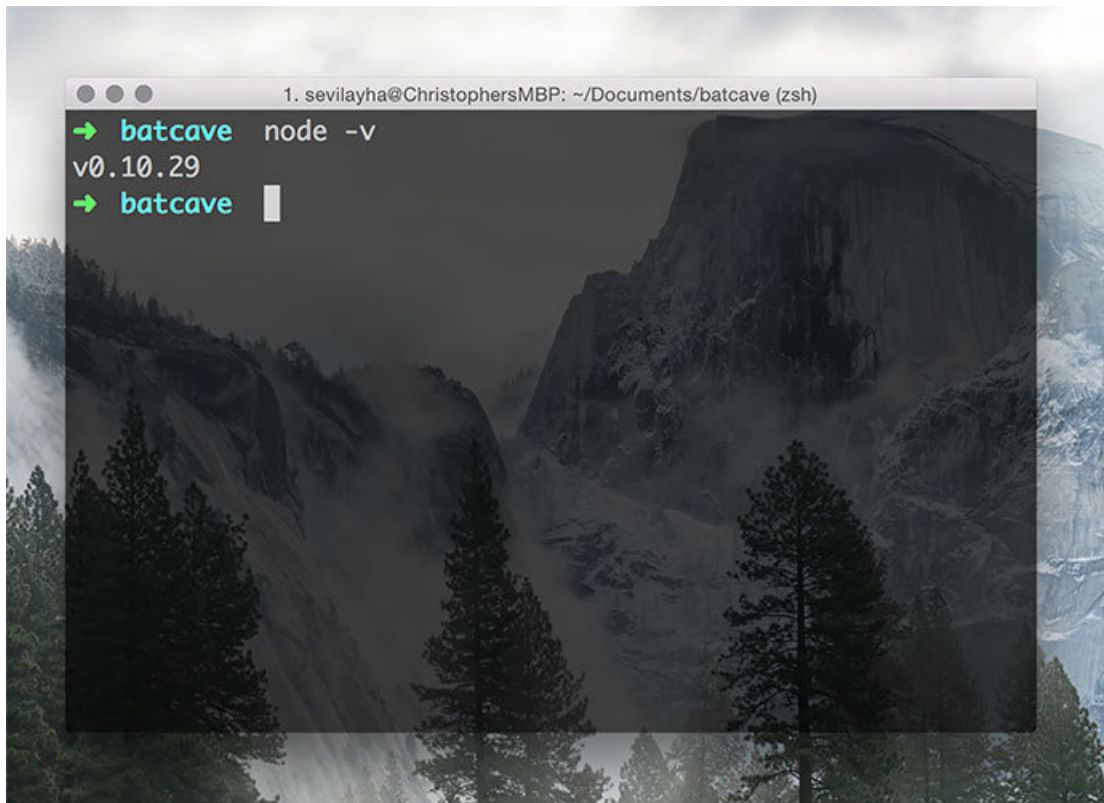
⁴⁰<http://nodejs.org>



Node Installation

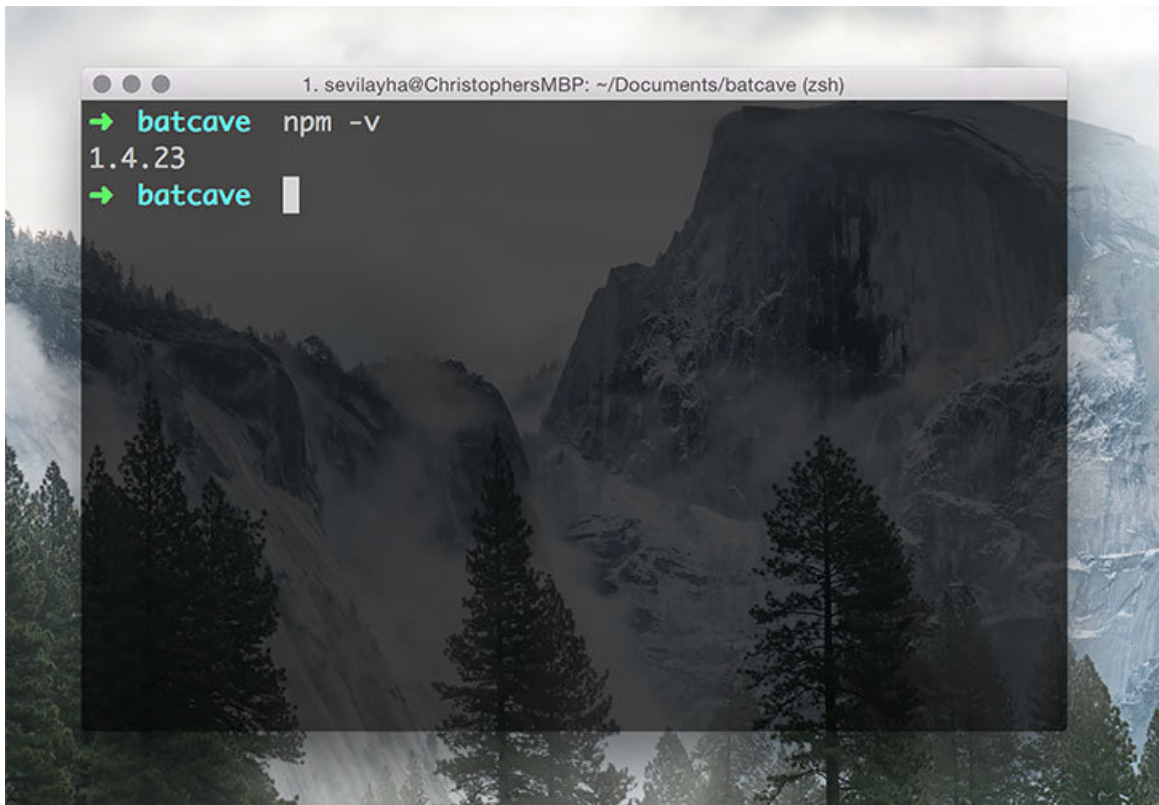
Let's double check that everything is working, go into your console and type the following: `node -v` and `npm -v`.

```
1 node -v
```



Node Version

```
1 npm -v
```

NPM Version

If you're getting the dreaded "command not found" on either of these, make sure that node has been added to your PATH. This provides the node and npm commands to the command line. Simply search for where npm was installed, open up your environment variables window, and add the path to npm. Restart your computer and let the magic begin.

For more detailed troubleshooting instructions, see the following:

Windows: [Add Node to your Windows PATH](#)⁴¹

Mac: [Add Node to your Mac PATH](#)⁴²

And if you're still having trouble, feel free to contact either of us with your questions. Now that your installation is all done, let's move forward and start building things! We'll begin with the foundation of our MEAN stack applications, Node!

⁴¹<http://stackoverflow.com/a/8768567/2976743>

⁴²http://architectryan.com/2012/10/02/add-to-the-path-on-mac-os-x-mountain-lion/#_wCiFNdWqc

Starting Node

Let's look at how we can start to build out our Node applications. We'll go through **basic Node configuration**, installing npm packages, and creating a **simple app**.

Configuration (package.json)

Node applications are configured within a file called `package.json`. You will need a `package.json` file for each project you create.

This file is where you configure the name of your project, versions, repository, author, and the all important dependencies.

Here is a sample `package.json` file:

```
1 {
2   "name": "mean-machine-code",
3   "version": "1.0.0",
4   "description": "The code repository for the book, MEAN Machine.",
5   "main": "server.js",
6   "repository": {
7     "type": "git",
8     "url": "https://github.com/scotch-io/mean-machine-code"
9   },
10  "dependencies": {
11    "express": "latest",
12    "mongoose": "latest"
13  },
14  "author": "Chris Sevilleja & Holly Lloyd",
15  "license": "MIT",
16  "homepage": "https://github.com/scotch-io/mean-machine-code"
17 }
```

That seems overwhelming at first, but if you take it line by line, you can see that a lot of the attributes created here make it easier for other developers to jump into the project. We'll look through all these different parts later in the book, but here's a very simple `package.json` with only the required parts.

```
1 {  
2   "name": "mean-machine-code",  
3   "main": "server.js"  
4 }
```

These are the most basic required attributes.

main tells Node which file to use when we want to start our applications. We'll name that file `server.js` for all of our applications and that will be where we start our applications.

For more of the attributes that can be specified in our `package.json` files, here are the [package.json docs](https://www.npmjs.org/doc/files/package.json.html)⁴³.

Initialize Node App

The `package.json` file is how we will start every application. It can be hard to remember exactly what goes into a `package.json` file, so npm has created an easy to remember command that let's you build out your `package.json` file quickly and easily. That command is **npm init**.

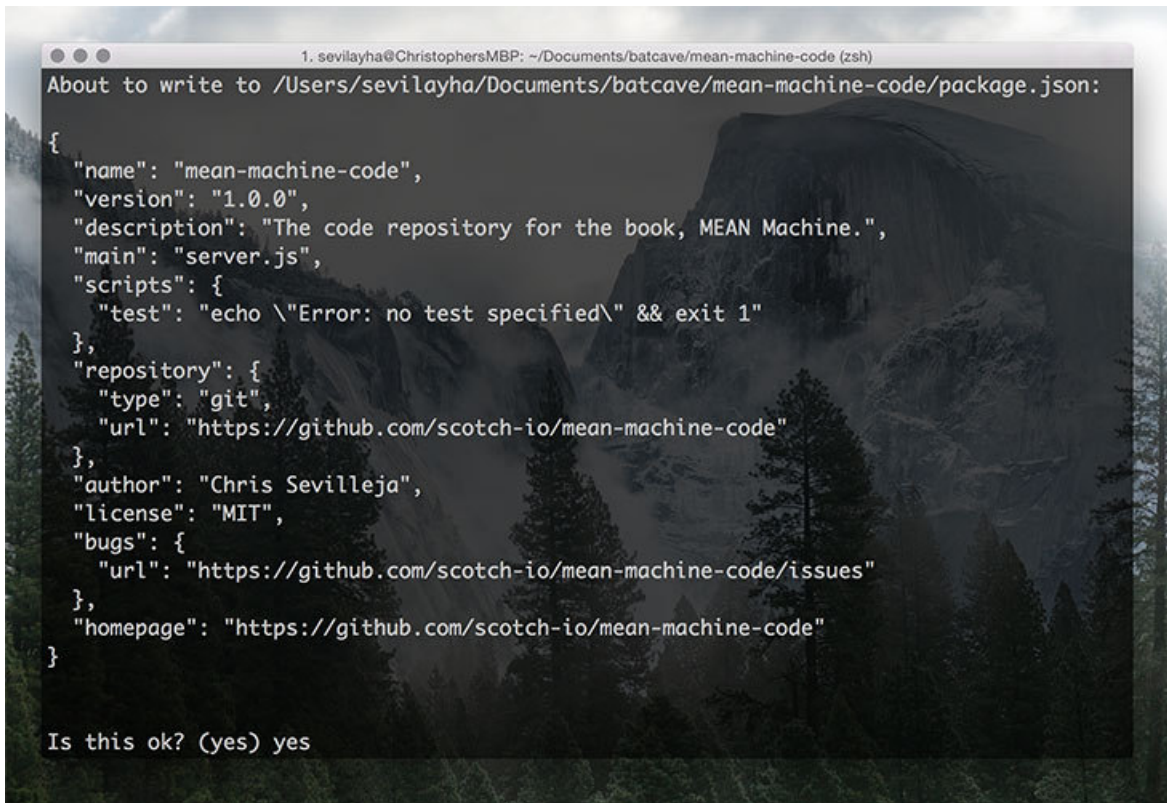
Let's create a sample project and test out the `npm init` command.

1. Create a folder: `mkdir awesome-test`
2. Jump into that folder: `cd awesome-test`
3. Start our Node project: `npm init`

It will give you a few options that you can leave as default or customize as you wish. For now, you can leave everything default except for the main (entry point) file. Ours will be called `server.js`.

You can see that our new `package.json` file is built and we have our first Node project!

⁴³<https://www.npmjs.org/doc/files/package.json.html>

A terminal window with a dark background and a mountain landscape wallpaper. The window title is '1. sevilayha@ChristophersMBP: ~/Documents/batcave/mean-machine-code (zsh)'. The prompt is 'About to write to /Users/sevilayha/Documents/batcave/mean-machine-code/package.json:'. The JSON content is: { "name": "mean-machine-code", "version": "1.0.0", "description": "The code repository for the book, MEAN Machine.", "main": "server.js", "scripts": { "test": "echo \\\"Error: no test specified\\\" && exit 1" }, "repository": { "type": "git", "url": "https://github.com/scotch-io/mean-machine-code" }, "author": "Chris Sevilleja", "license": "MIT", "bugs": { "url": "https://github.com/scotch-io/mean-machine-code/issues" }, "homepage": "https://github.com/scotch-io/mean-machine-code" }. The prompt 'Is this ok? (yes) yes' is at the bottom.

```
1. sevilayha@ChristophersMBP: ~/Documents/batcave/mean-machine-code (zsh)
About to write to /Users/sevilayha/Documents/batcave/mean-machine-code/package.json:
{
  "name": "mean-machine-code",
  "version": "1.0.0",
  "description": "The code repository for the book, MEAN Machine.",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/scotch-io/mean-machine-code"
  },
  "author": "Chris Sevilleja",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/scotch-io/mean-machine-code/issues"
  },
  "homepage": "https://github.com/scotch-io/mean-machine-code"
}

Is this ok? (yes) yes
```

NPM Init

Since we have a `package.json` file now, we can go into our command line and type `node server.js` to start up this Node app! It will just throw an error since we haven't created the `server.js` file that we want to use to begin our Node application. Not very encouraging to see an error on our first time starting a Node server! Let's change that and make an application that does something.

Creating a Very Simple Node App

Open up your `package.json` file and delete everything except those basic requirements:

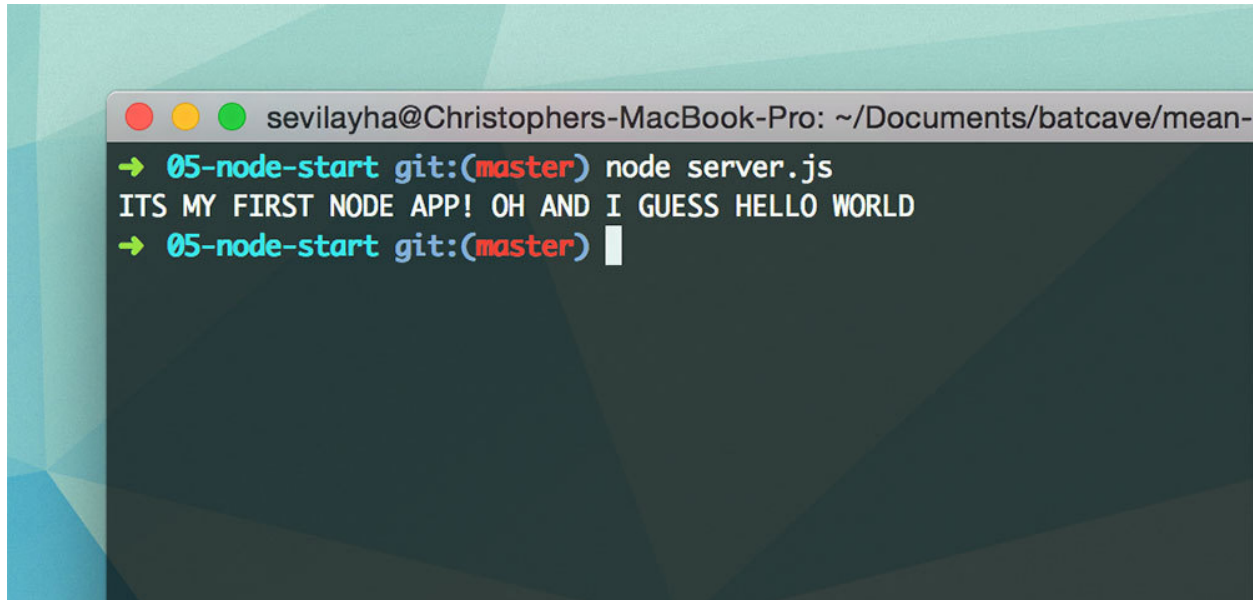
```
1  {
2    "name": "awesome-test",
3    "main": "server.js"
4  }
```

Now we will need to create the `server.js` file. The only thing we will do here is `console.log` out some information. **`console.log()`** is the way we dump information to our console. We're going to use it to send a message when we start up our Node app.

Here is our `server.js` file's contents.

```
1 console.log('ITS MY FIRST NODE APP! OH AND I GUESS HELLO WORLD');
```

Now we can start up our Node application by going into our command line and typing: `node server.js`

A screenshot of a terminal window on a Mac. The title bar shows the user 'sevilayha' and the path '~/Documents/batcave/mean-'. The terminal prompt is '05-node-start git:(master)'. The user has entered 'node server.js' and the output is 'ITS MY FIRST NODE APP! OH AND I GUESS HELLO WORLD'. The prompt is now '05-node-start git:(master)' with a cursor.

`node server.js`

Starting a Node Application

To start a Node application, you just go into the command line and type:

```
node server.js
```

`server.js` is what we defined to be our main file in **package.json** so that's the file we will specify when starting. Our server will now be stopped since all we did was `console.log()`, but in the future, if you would like to stop your Node server, you can just type 'ctrl c'.



Tip

Restarting a Node Application on File Changes

By default, the `node server.js` command will start up our application, but it **won't restart when file changes are made**. This can become tedious when we are developing since we will have to shut down and restart every time we make a change.

Luckily there is an npm package that will watch for file changes and restart our server when changes are detected. This package is called [nodemon](https://github.com/remy/nodemon)⁴⁴ and to install it, just go into your command line and type: `npm install -g nodemon`. The `-g` modifier means that this package will be installed globally for your system. Now, instead of using `node server.js`, we are able to use:

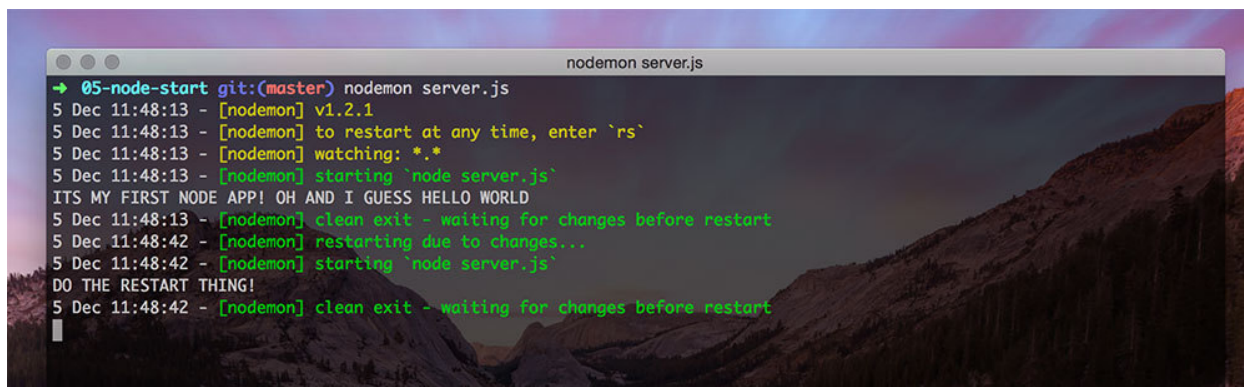
```
nodemon server.js
```

Again, if you're getting the error that nodemon command isn't found you'll have to edit your PATH in environment variables. A quick way to do this is type '`npm config get prefix`'. This will show you the path to npm. To add this to your PATH variable type in

'`set PATH %PATH%;c:\whateverthepathis`' where '`whateverthepathis`' is the result of `npm config get prefix`

You will probably have to restart your computer but after that you're ready to go!

Feel free to go into your `server.js` file and make changes. Then watch the magic happen when application restarts itself!



```
nodemon server.js
```

For the rest of this book, we will reference nodemon when we want to start a server. It's just the easier option when developing.

We've now configured a real simple Node app and started it up from our command line. We're one step closer to building full blown Node applications that are ready to show off to the world.

⁴⁴<https://github.com/remy/nodemon>

Packages

Packages extend the functionality of our application. Like we talked about in our earlier Primers section, even one of the 4 main parts of the MEAN stack, Express, is a Node package.

Let's look at how we can add and install packages. Once we've talked about getting packages into our application, we will move on to using the other components of the MEAN stack.

Installing Packages

The `package.json` file is where we have defined our application name and the main file to start our application. This is also where we will define the packages that are needed.

There are two ways we can add packages to our project: **writing them directly into `package.json`** and **installing via the command line**. Both of these ways will add the packages we need to the `dependencies` section of `package.json`.

Method 1. Writing Packages Directly into `Package.json`

Here is a `package.json` file that we have added the Express dependency to.

```
1 {  
2   "name": "packages-install",  
3   "main": "server.js",  
4   "dependencies": {  
5     "express": "~4.8.6"  
6   }  
7 }
```

Just like that, we now have Express set as a package for our application and have two parts of the MEAN stack! (_E_N)



Note: npm Version Numbers

You may be wondering what that tilde (~) is doing next to the version number of Express. npm uses [semantic versioning](https://www.npmjs.org/doc/misc/semver.html)⁴⁵ when declaring package versions. This means that the tilde will pull in the version that is **reasonably close** to the one you specified. In this example, only versions of Express that are greater than 4.8.6 and less than 4.9 will be installed.

The three numbers each stand for a different portion of that version. For example, in express 4.8.6 the 4 represents a major version, 8 represents minor version, and 6 represents a patch. Usually bug fixes will be categorized as a patch and shouldn't break anything. A minor version update will add new features, but still not break your previous code. And a major update might break existing code, which might make you want to pull your hair out as some of you may already know.

Using this type of versioning is good practice because we ensure that only the version we specify will be pulled into our project. We will be able to grab bug fixes up to the 4.9 version, but not any major changes that would come with the 5.0 version.

If we revisit this application in the future and use `npm install`, we know exactly what version of Express we are using and that we won't break our app by bringing in any newer versions since we made this project.

Method 2. Adding Packages from the Command Line

The second way to add packages to our application is to use the npm shortcuts from the command line. This is often times the easier and better route since npm can automatically save the package to your `package.json` file. And here's the cool part: **it will add the right version number!** If you write packages into `package.json` you'll have to dig around online to find the right version number. This method makes life much easier.

Here is the command to install express and the `--save` modifier will add it to `package.json`.

```
npm install express --save
```

You'll notice that the above command grabs the express package and installs it into a new folder called **node_modules**. This is where packages live inside Node projects. This command installs only the packages that we call specifically (express in this case).

⁴⁵<https://www.npmjs.org/doc/misc/semver.html>

Installing All Packages

Method 2 will install packages for us. Method 1 will add the packages to your `package.json`, but it won't install them just yet. To install all the packages inside the `dependencies` attribute of `package.json` to the `node_modules` folder, just type:

```
npm install
```

That will look at the dependencies we need and pull them into our application in the `node_modules`.

Installing Multiple Packages

npm also comes with a handy way to install multiple packages. Just type in all the packages you want into one `npm install` command and they will be brought into the project.

```
npm install express mongoose passport --save
```

This is a simple and easy way to bring in the packages needed.

Recap

Now when we want to start up a new Node project, we just need to run two commands in the folder we want to create the project:

1. `npm init`
2. Fill out the fields necessary when prompted to create your `package.json` file.
3. `npm install express --save`

Just like that we have everything we need to set up our application!

Let's move forward and dig into using Express and Node to set up the foundation of our MEAN stack apps.

Starting a Node Server

We're going to move forward and look at the ways to set up a node HTTP server so that we can send HTML files and more to our users. In the previous chapter, we only logged something to the console. In this chapter, we will take what we learned a step further so that we can serve a website to our users. We'll be another step closer to fully-fledged web applications.

Method #1: Pure Node (no Express) This is a simple way to create our server and we'll just do it this way so we know how it is done with barebones Node. After this, we will be using the Express method (method #2) exclusively.

Method #2: Using Express Since Express is one of the main four parts of the MEAN stack, we want to use this method. And once you learn this method you'll probably never want to look back anyway.

Sample Application

For this chapter's example, we will send an HTML file to our browser using Method #1 and Method #2. To get started, we need a brand new folder with the following files:

- `package.json`
- `server.js`
- `index.html`

Like we learned before, `package.json` will hold the configuration/packages for our project while `server.js` will have our application setup and configuration. Our `index.html` file will be a basic HTML file.

`package.json` and `index.html` will be the same for both methods. The only file that will change between the two methods is the `server.js` file because that's where we will start up our Node server.

package.json

Here is our simple `package.json` file:

```
1 {  
2   "name": "http-server",  
3   "main": "server.js"  
4 }
```

index.html

Let's fill out a simple `index.html` file also:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3 <head>  
4   <meta charset="UTF-8">  
5   <title>Super Cool Site</title>  
6   <style>  
7     body {  
8       text-align:center;  
9       background:#EFEFEF;  
10      padding-top:50px;  
11    }  
12  </style>  
13 </head>  
14 <body>  
15  
16   <h1>Hello Universe!</h1>  
17  
18 </body>  
19 </html>
```

Method #1: Pure Node (no Express)

When using Method #1, we will pull in two modules that are built into Node itself. The [HTTP module](http://nodejs.org/api/http.html#http_http_module)⁴⁶ is used to start up HTTP servers and respond to HTTP requests from users. The [fs module](http://nodejs.org/api/fs.html#fs_file_system)⁴⁷ is used to read the file system. We will need to read our `index.html` from the file system and then pass it to our user using an HTTP server.

The `server.js` for Method #1 will have the following code:

⁴⁶http://nodejs.org/api/http.html#http_http_module

⁴⁷http://nodejs.org/api/fs.html#fs_file_system

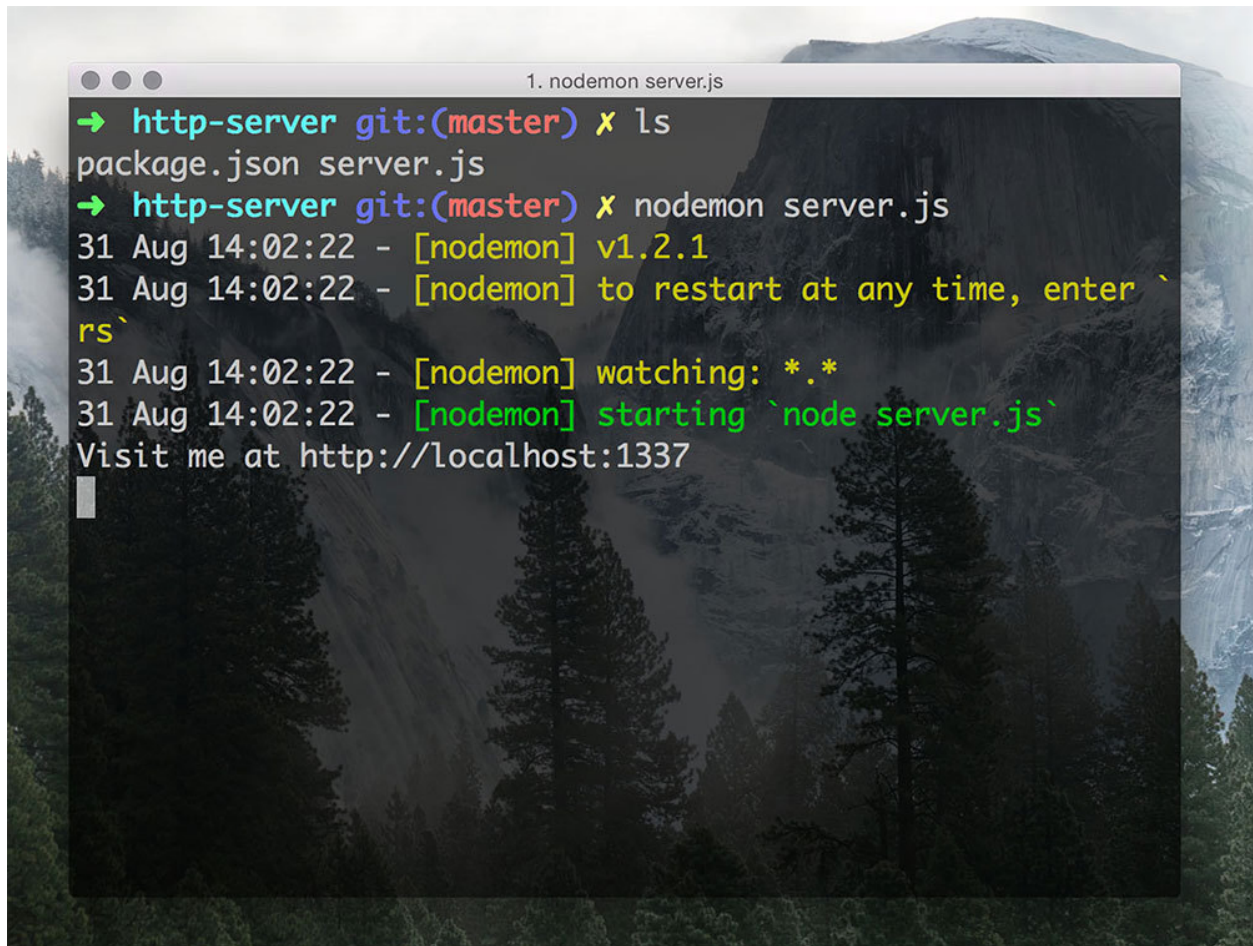
```
1  // get the http and filesystem modules
2  var http = require('http'),
3      fs = require('fs');
4
5  // create our server using the http module
6  http.createServer(function(req, res) {
7
8      // write to our server. set configuration for the response
9      res.writeHead(200, {
10         'Content-Type': 'text/html',
11         'Access-Control-Allow-Origin' : '*'
12     });
13
14     // grab the index.html file using fs
15     var readStream = fs.createReadStream(__dirname + '/index.html');
16
17     // send the index.html file to our user
18     readStream.pipe(res);
19
20 }).listen(1337);
21
22 // tell ourselves what's happening
23 console.log('Visit me at http://localhost:1337');
```

We are using the http module to create a server and the fs module to grab an index file and send it in our response to the user.

With our server.js file defined, let's go into our command line and start up our Node http server.

```
nodemon server.js
```

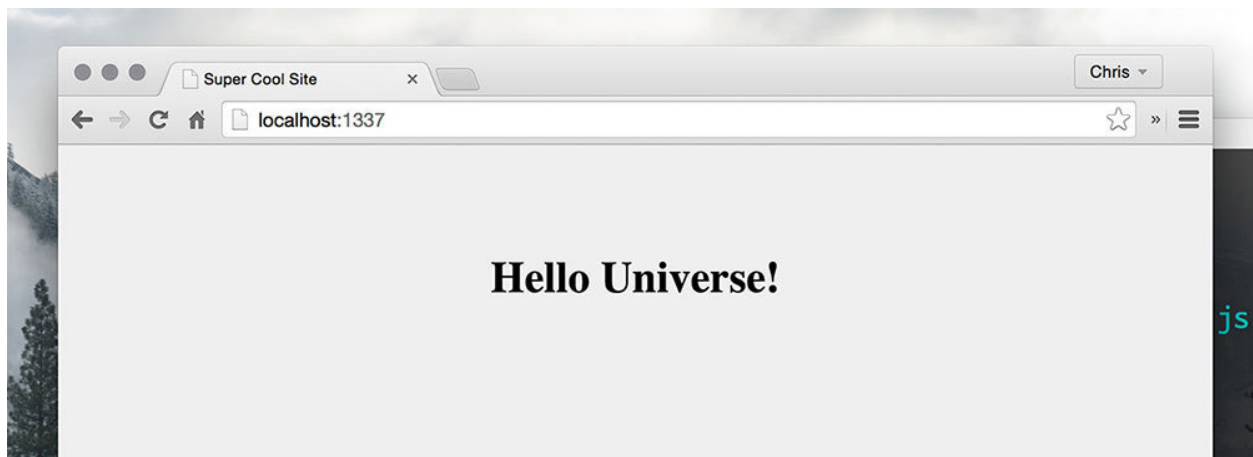
You should see your server start up and a message logged to the console.

A terminal window titled "1. nodemon server.js" is shown against a dark, forested background. The terminal displays the following commands and output:

```
→ http-server git:(master) ✕ ls
package.json server.js
→ http-server git:(master) ✕ nodemon server.js
31 Aug 14:02:22 - [nodemon] v1.2.1
31 Aug 14:02:22 - [nodemon] to restart at any time, enter `rs`
31 Aug 14:02:22 - [nodemon] watching: *.*
31 Aug 14:02:22 - [nodemon] starting `node server.js`
Visit me at http://localhost:1337
```

Node HTTP Server Console

Now we can see our site in browser at <http://localhost:1337>.



Hello Universe Browser

We've finally sent an HTML file to our users! You may be thinking that setting up that http server

took a lot of syntax that you might not be able to remember. Don't worry though, the Express way is much cleaner.

Method #2: Using Express

Now that we have started up a server using the HTTP module, let's look at how we can do the same with Express. You'll find that it's much easier to manage the code.

We will first need to add Express to this project. Let's use the command line to install it and save it to our `package.json`.

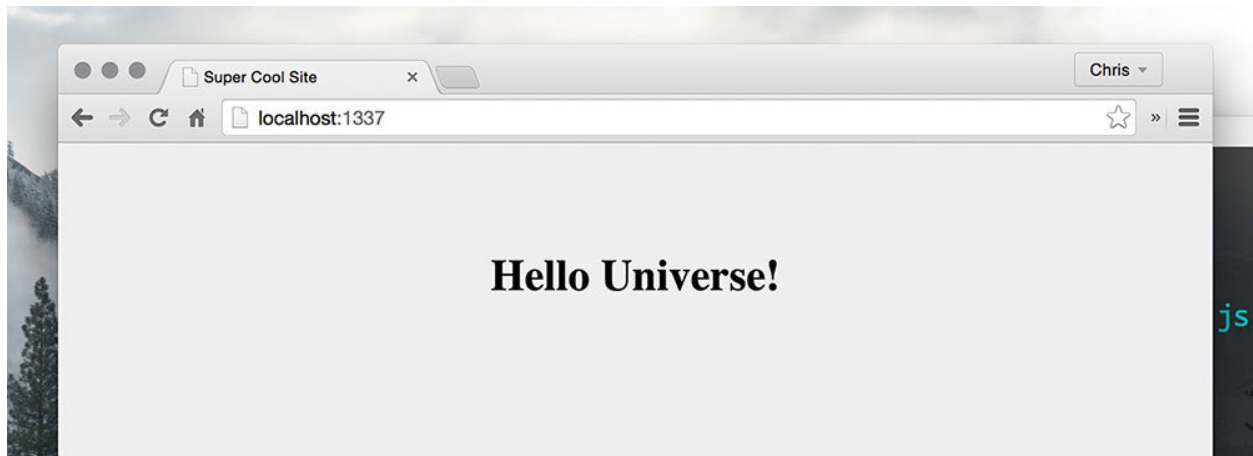
```
npm install express --save
```

Now let's change our `server.js` file to accommodate Express. We'll start by calling Express. Using that instance of Express, we will define a route and send the `index.html` file. Then we'll "listen" on a certain port for a request from the browser.

```
1 // load the express package and create our app
2 var express = require('express');
3 var app     = express();
4 var path    = require('path');
5
6 // send our index.html file to the user for the home page
7 app.get('/', function(req, res) {
8     res.sendFile(path.join(__dirname + '/index.html'));
9 });
10
11 // start the server
12 app.listen(1337);
13 console.log('1337 is the magic port!');
```

Now we have **grabbed Express, set a port, and started our server**. Express has made the job of starting a server much easier and the syntax is cleaner. Starting up this server using `nodemon server.js` will spit out our HTML file to users at `http://localhost:1337` again.

We'll see the same result as before, but now the code is a little easier to follow.



Hello Universe Browser



Tip

Defining Packages Shortcut

Just a quick tip when defining packages, instead of typing out `var` every time you define a package, you can string the packages together using a `" , "`.

Here's an example:

```
1 var express = require('express');
2 var app = express();
3
4 // the exact same result
5
6 var express = require('express'),
7     app = express();
```

This just tidies up your code a bit and lets you skip a few extra keystrokes.

We can now create a Node server and send HTML files to our users. We've defined a simple get route so far. Let's take the next step and add more routes and pages to our site.