MEAN 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 2015-07-31

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.

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

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[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.io[3] and holly@scotch.io[4].

Also, take a look at our site (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 lets 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[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 "hu**mongo**us", is an open-source NoSQL database that stores documents in JSON-style format. It is the 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).

```
{
  id: "1234",
  name: "holly",
  age: "400",
  type: "high-elf",
  address: {
    city: "rivendell",
    state: "middle-earth"
  }
}
```

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[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[19] is currently the most starred package on npm's site[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://koajs.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[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.

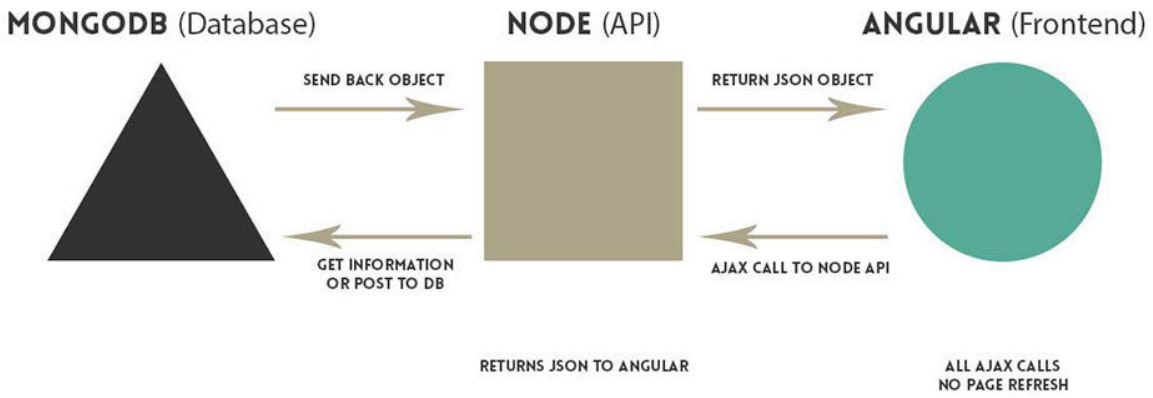
[34] http://en.wikipedia.org/wiki/Eating_your_own_dog_food

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

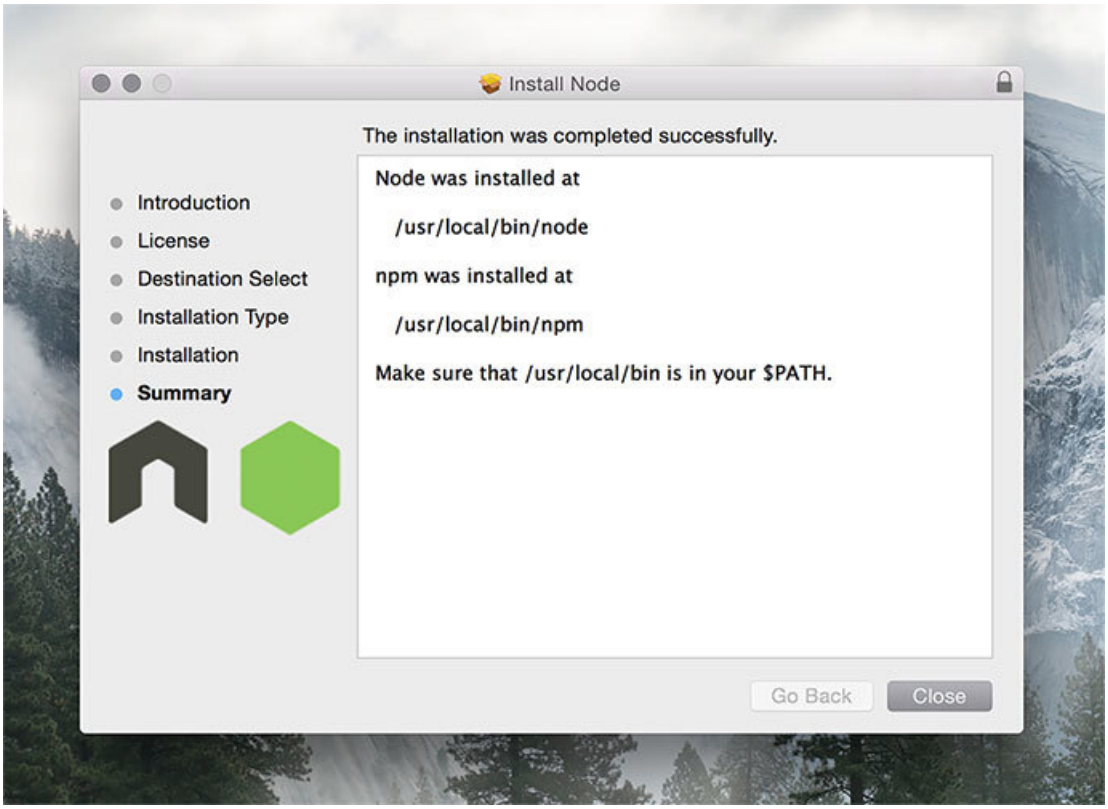
- Sublime Text
- Terminal - We like using iterm2[36] (mac/linux users)
- Git Bash + ConEmu[37] (windows users)
- Postman[38] (Chrome)
- RESTClient[39] (Firefox)

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.

---

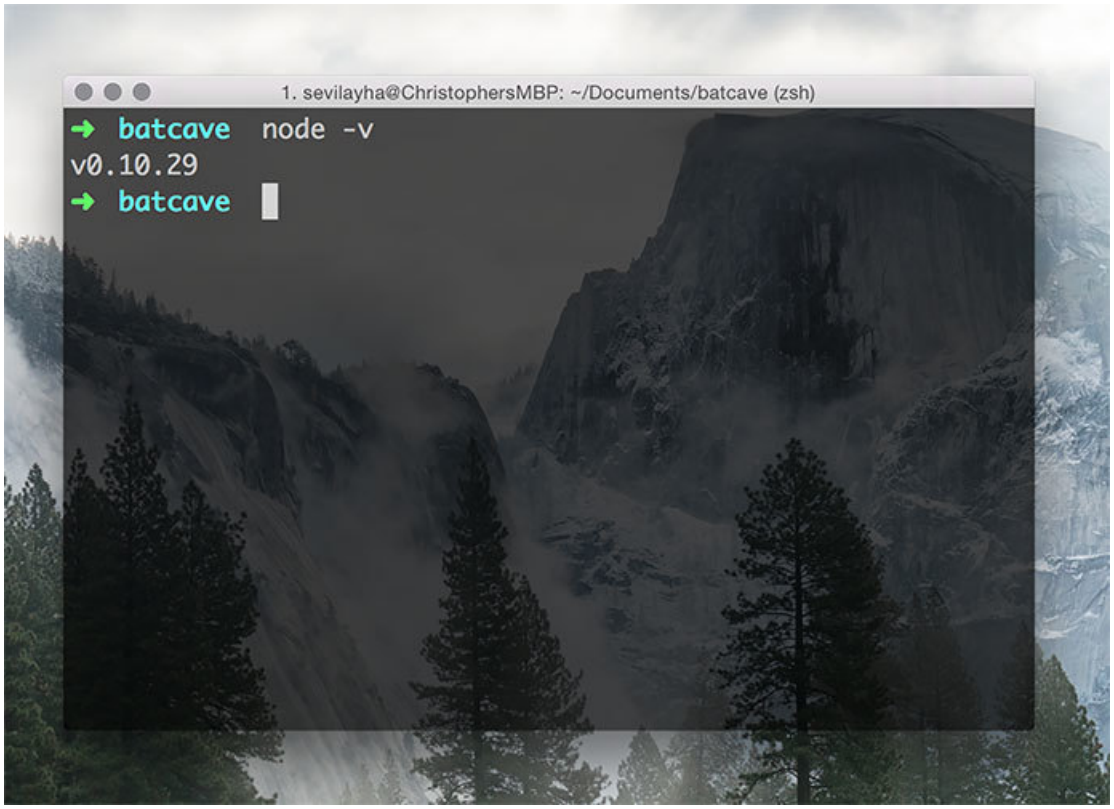
[35] http://nodejs.org
[36] http://iterm2.com/
[37] http://scotch.io/bar-talk/get-a-functional-and-sleek-console-in-windows
[38] https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en
[39] https://addons.mozilla.org/en-US/firefox/addon/restclient/
[40] http://nodejs.org



Node Installation

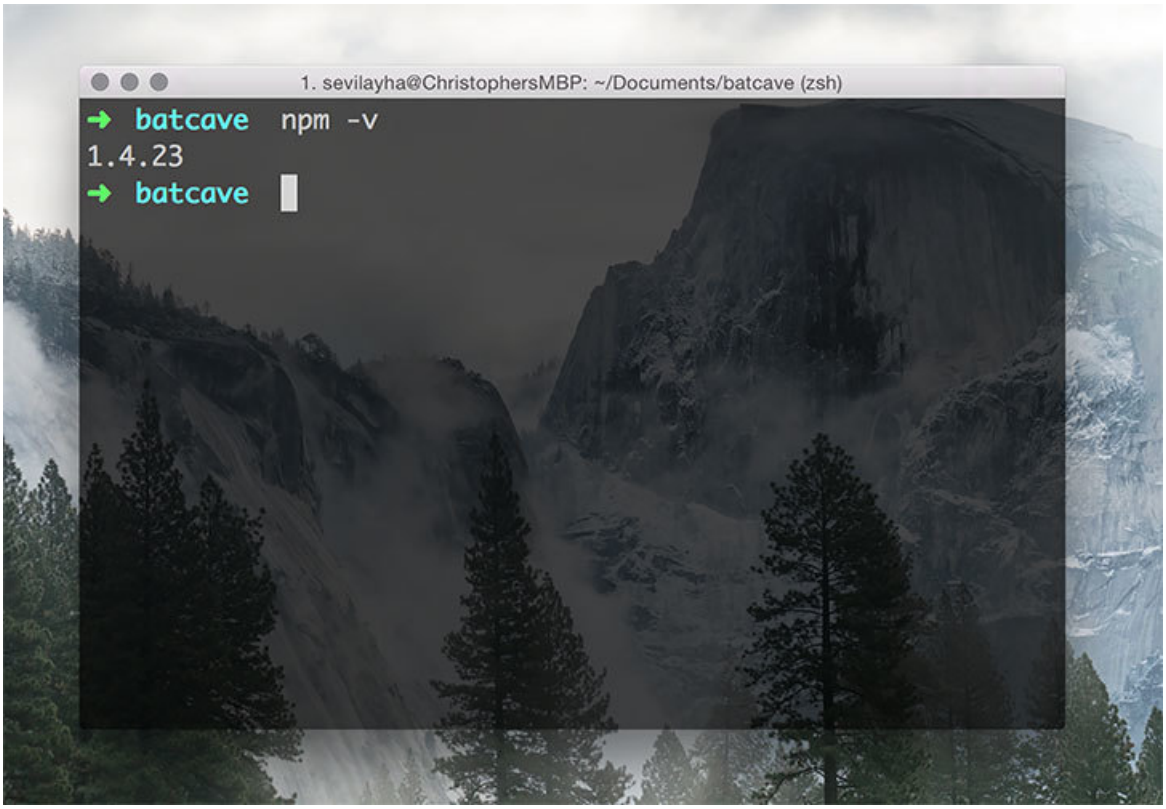
Let's double check that everything is working, go into your console and type the following: `node -v` and `npm -v`.

```
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

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

```
{
  "name": "mean-machine-code",
  "version": "1.0.0",
  "description": "The code repository for the book, MEAN Machine.",
  "main": "server.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/scotch-io/mean-machine-code"
  },
  "dependencies": {
    "express": "latest",
    "mongoose": "latest"
  },
  "author": "Chris Sevilleja & Holly Lloyd",
  "license": "MIT",
  "homepage": "https://github.com/scotch-io/mean-machine-code"
}
```

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.

```
{
  "name": "mean-machine-code",
  "main": "server.js"
}
```

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[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**.

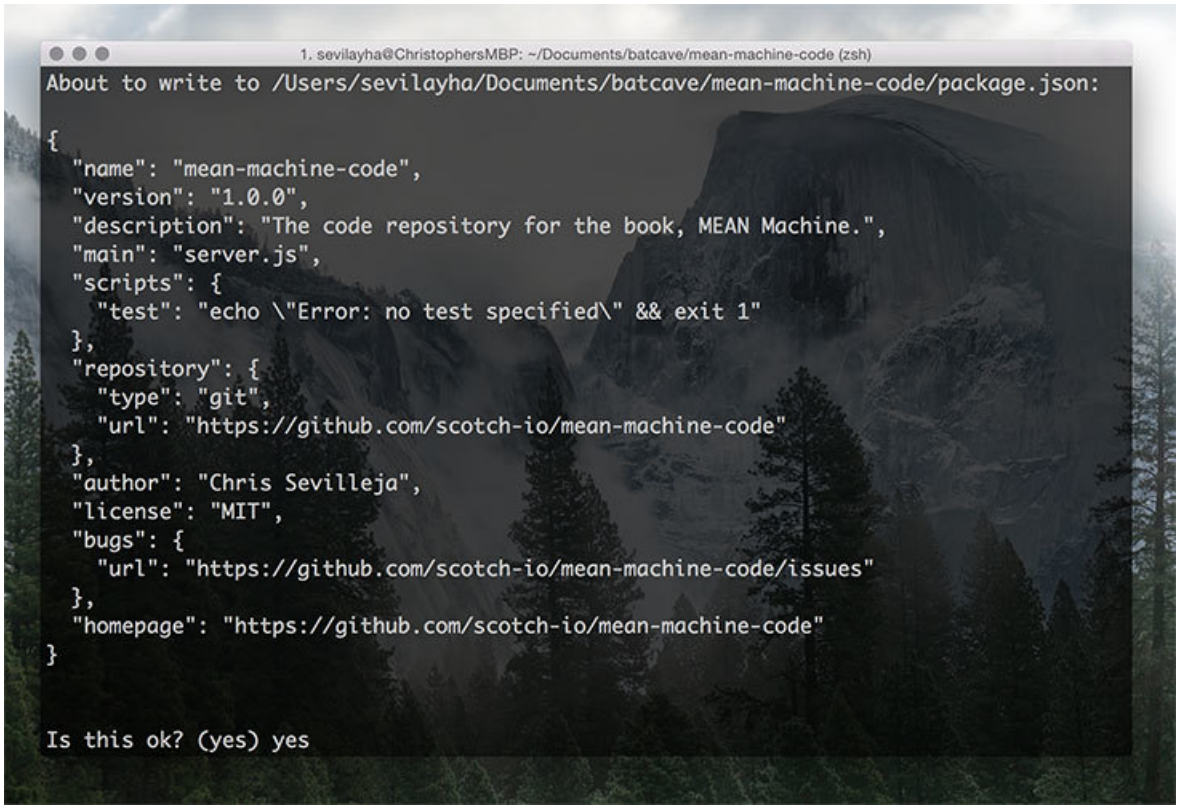
Let's create a sample project and test out the `npm init` command.

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



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:

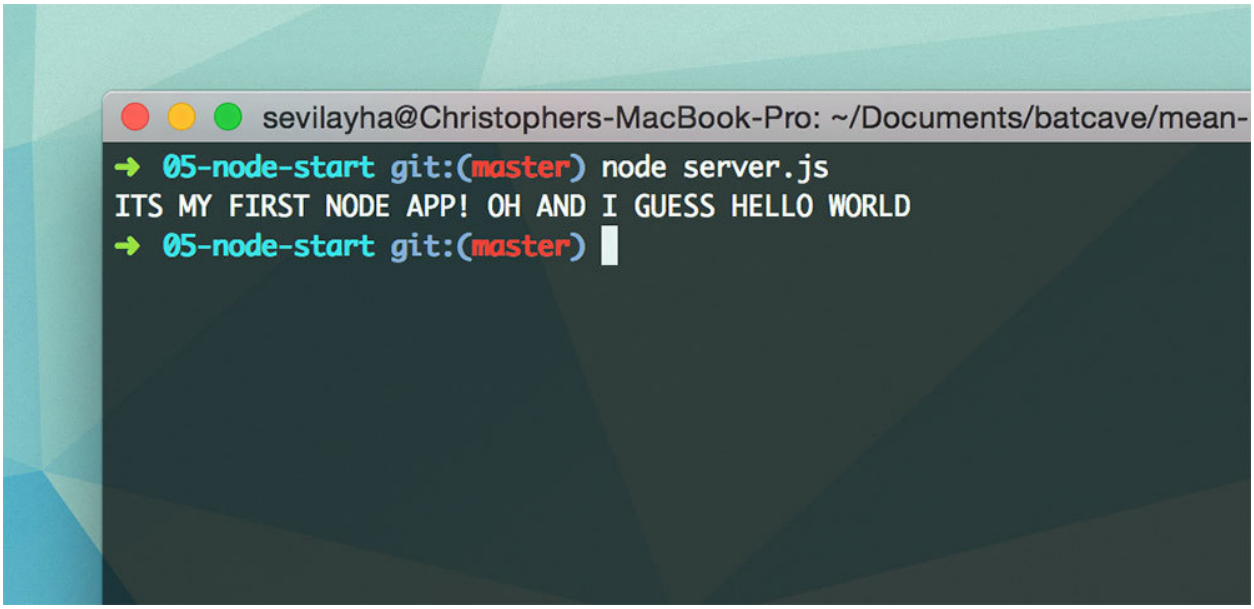
```
{
  "name": "awesome-test",
  "main": "server.js"
}
```

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`



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[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.

```
{
  "name": "packages-install",
  "main": "server.js",
  "dependencies": {
    "express": "~4.8.6"
  }
}
```

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[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:

```
{
  "name": "http-server",
  "main": "server.js"
}
```

index.html

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

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <title>Super Cool Site</title>
        <style>
                body {
                        text-align:center;
                        background:#EFEFEF;
                        padding-top:50px;
                }
        </style>
</head>
<body>

        <h1>Hello Universe!</h1>

</body>
</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[46] is used to start up HTTP servers and respond to HTTP requests from users. The fs module[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:

[46] http://nodejs.org/api/http.html#http_http
[47] http://nodejs.org/api/fs.html#fs_file_system

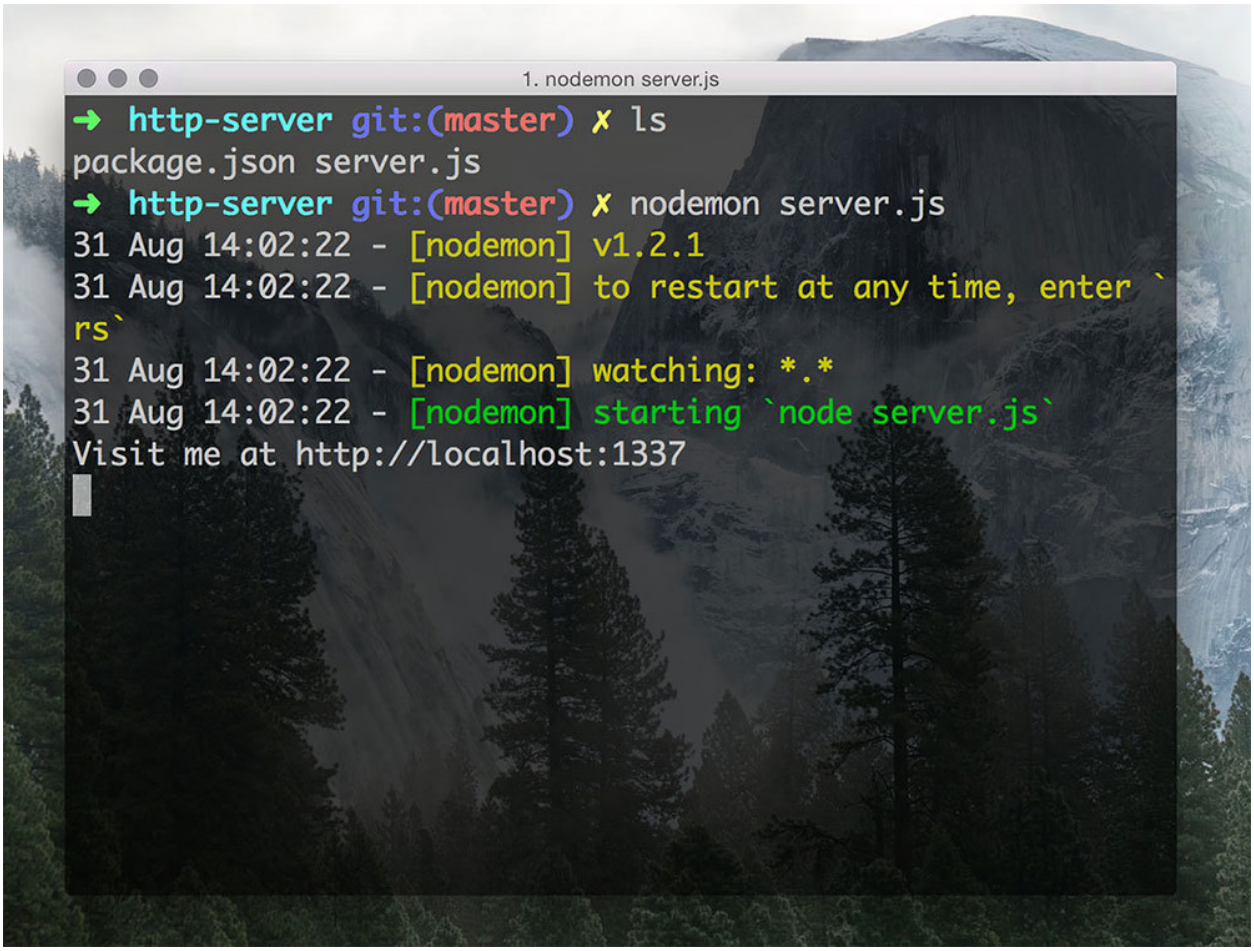
```
// get the http and filesystem modules
var http = require('http'),
  fs = require('fs');

// create our server using the http module
http.createServer(function(req, res) {

  // write to our server. set configuration for the response
  res.writeHead(200, {
    'Content-Type': 'text/html',
    'Access-Control-Allow-Origin' : '*'
  });

  // grab the index.html file using fs
  var readStream = fs.createReadStream(__dirname + '/index.html');

  // send the index.html file to our user
  readStream.pipe(res);

}).listen(1337);

// tell ourselves what's happening
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.

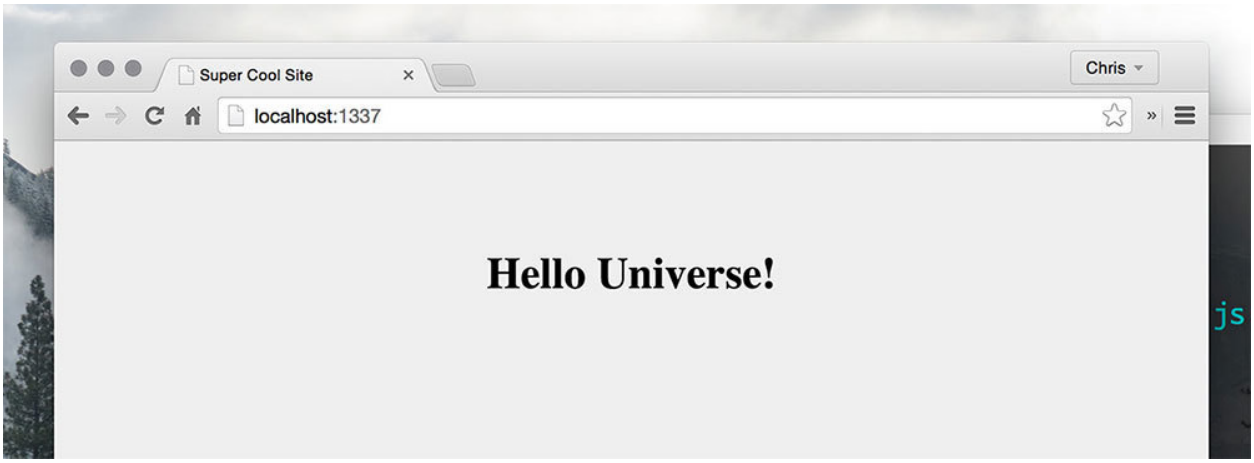
```
nodemon server.js
```

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



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 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`.

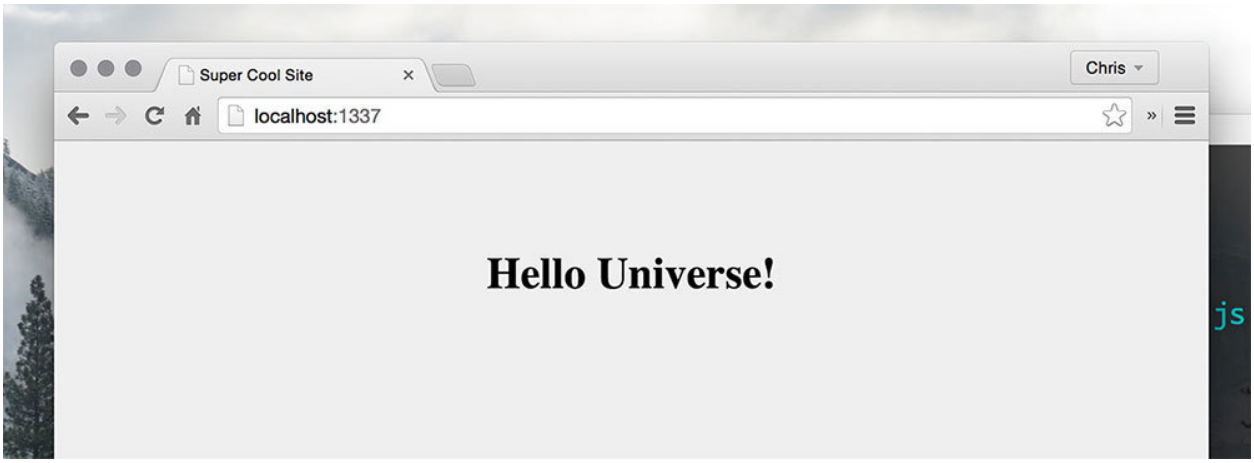
```
npm install express --save
```

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:

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

// the exact same result

var express = require('express'),
    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.

Routing Node Applications

To add more pages to our site, we will need more routes. This can be done using a feature built into Express, the Express Router. This is going to be a big chapter since routing is going to be a giant portion of all our applications moving forward. This is how we route basic to advanced websites, and also how we will ultimately build out our RESTful APIs that an Angular frontend application will consume. That's exciting to look forward that far! One step at a time though. Let's get to the routing.

Express Router

What exactly is the Express Router? You can consider it a mini express application without all the bells and whistles, just the routing stuff. It doesn't bring in views or settings, but provides us with the routing APIs like `.use()`, `.get()`, `.param()`, and `route()`. Let's take a look at exactly what this means.

There are a few different ways to use the router. We've already used one of the methods when we created the home page route in the last chapter by using 'app.get('/', ...). We'll look at the other methods by building out more sections of our site and discuss why and when to use them.

Sample Application Features

These are the main features we will add to our current application:

- Basic Routes (We've already created the homepage)
- Site Section Routes (Admin section with sub routes)
- Route Middleware to log requests to the console
- Route with Parameters (`http://localhost:1337/users/holly`)
- Route Middleware for Parameters to validate specific parameters
- Login routes doing a GET and POST on /login
- Validate a parameter passed to a certain route

What is Route Middleware? Middleware is invoked between a user's request and the final response. We'll go over this concept when we log data to the console on every request. A user will request the page, we will log it to the console (the middleware), and then we'll respond with the page they want. More on middleware soon.

Like we've done so far, we will keep our routes in the `server.js` file. We won't need to make any changes to our `package.json` file since we already have Express installed.

Basic Routes

We've already defined our basic route in the home page. Express let's us define routes right onto our `app` object. We can also handle multiple HTTP actions like `GET`, `POST`, `PUT/PATCH`, AND `DELETE`.

This is the easiest way to define routes, but as our application gets larger, we'll need more organization for our routes. Just imagine an application that has an administration section and a frontend section, each with multiple routes. Express's router helps us to organize these when we define them.

For the following routes, we won't be sending views to the browser, just messages. This will be easier since what we want to focus on is the routing aspects.

express.Router()

The express.Router()[48] acts as a mini application. You can call an instance of it (like we do for Express) and then define routes on that. Let's look at an example so we know exactly what this means. Add this to your 'server.js' file if you'd like to follow along.

Underneath our `app.get()` route inside of `server.js`, add the following. We'll 1. **call an instance of the router** 2. **apply routes to it** 3. and then **add those routes to our main app**

```
1  // create routes for the admin section
2
3  // get an instance of the router
4  var adminRouter = express.Router();
5
6  // admin main page. the dashboard (http://localhost:1337/admin)
7  adminRouter.get('/', function(req, res) {
8          res.send('I am the dashboard!');
9  });
10
11 // users page (http://localhost:1337/admin/users)
12 adminRouter.get('/users', function(req, res) {
13         res.send('I show all the users!');
14 });
15
16 // posts page (http://localhost:1337/admin/posts)
17 adminRouter.get('/posts', function(req, res) {
18         res.send('I show all the posts!');
19 });
```

[48] http://expressjs.com/api#router

```
20
21  // apply the routes to our application
22  app.use('/admin', adminRouter);
```

We will call an instance of the `express.Router()` and assign it to the `adminRouter` variable, apply routes to it, and then tell our application to use those routes.

We can now access the admin panel page at `http://localhost:1337/admin` and the sub-pages at `http://localhost:1337/admin/users` and `http://localhost:1337/admin/posts`.

Notice how we can set a default root for using these routes we just defined. If we had changed line 22 to app.use('/app', router), then our routes would be `http://localhost:1337/app/` and `http://localhost:1337/app/users`.

This is very powerful because we can create multiple express.Router()s and then apply them to our application. We could have a Router for our basic routes, authenticated routes, and even API routes.

Using the Router, we are allowed to make our applications more modular and flexible than ever before by creating multiple instances of the Router and applying them accordingly. Now we'll take a look at how we can use middleware to handle requests.

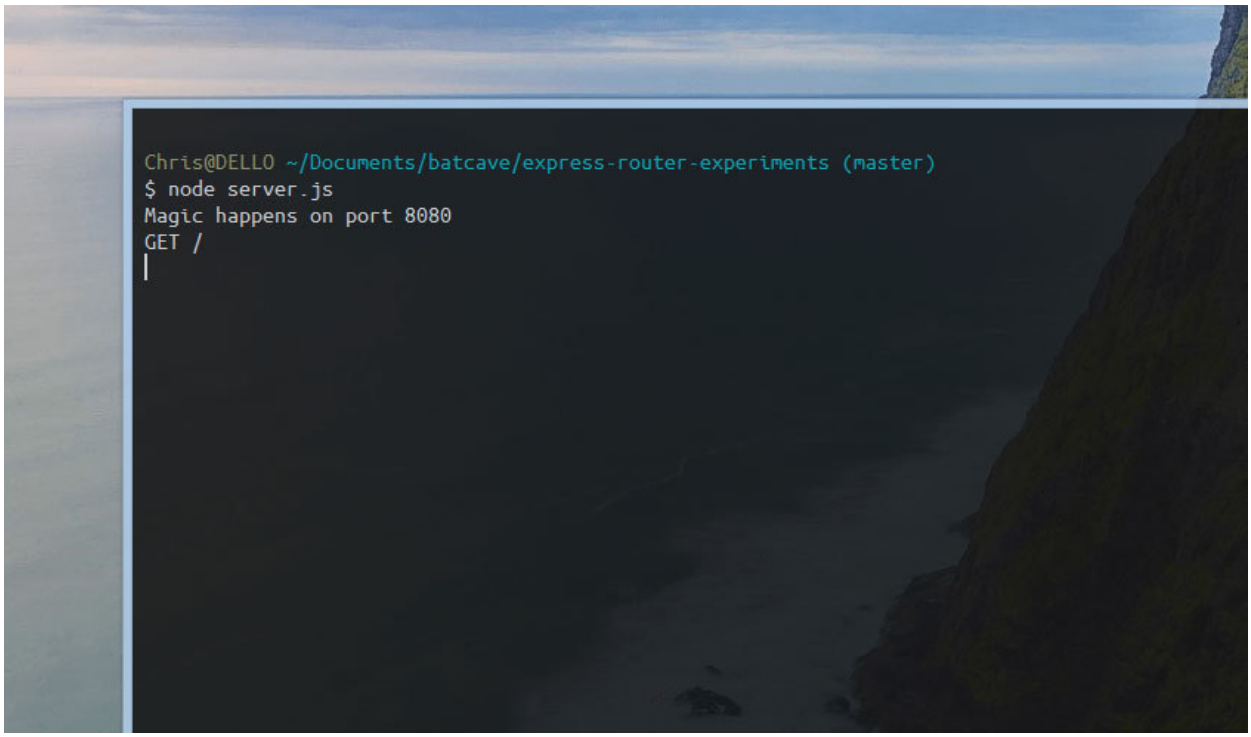
Route Middleware (router.use())

Route middleware in Express is a way to do something before a request is processed. This could be things like checking if a user is authenticated, logging data for analytics, or anything else we'd like to do before we actually spit out information to our user.

Here is some middleware to log a message to our console every time a request is made. This will be a demonstration of how to create middleware using the Express Router. We'll just add the middleware to the adminRouter we created in the last example. Make sure that this is placed **after** your adminRouter declaration and **before** the users and posts routes we defined. You'll also notice the 'next' argument here. This is the only way that Express will know that the function is complete and it can proceed with the next piece of middleware or continue on to the routing.

```
1  // route middleware that will happen on every request
2  adminRouter.use(function(req, res, next) {
3
4          // log each request to the console
5          console.log(req.method, req.url);
6
7          // continue doing what we were doing and go to the route
8          next();
9  });
```

adminRouter.use() is used to define middleware. This will now be applied to all of the requests that come into our application for this instance of Router. Let's go into our browser and go to `http://localhost:1337/admin` and we'll see the request in our console.



Express Router Console Request

The order you place your middleware and routes is very important Everything will happen in the order that they appear. This means that if you place your middleware after a route, then the route will happen before the middleware and the request will end there. Your middleware will not run at that point.

Keep in mind that you can use route middleware for many things. You can use it to check that a user is logged in during the session before letting them continue.

Structuring Routes

By using the Router(), we are able to section off parts of our site. This means you can create a `basicRouter` for routes like the frontend of the site. You could also create an `adminRouter` for administration routes that would be protected by some sort of authentication.

Routing our application this way let's us compartmentalize each piece. This provides us the flexibility that we need for complex applications or APIs. We can also keep our applications clean and organized since we can move each router definition into its own file and then just pull in those files when we call `app.use()` like so:

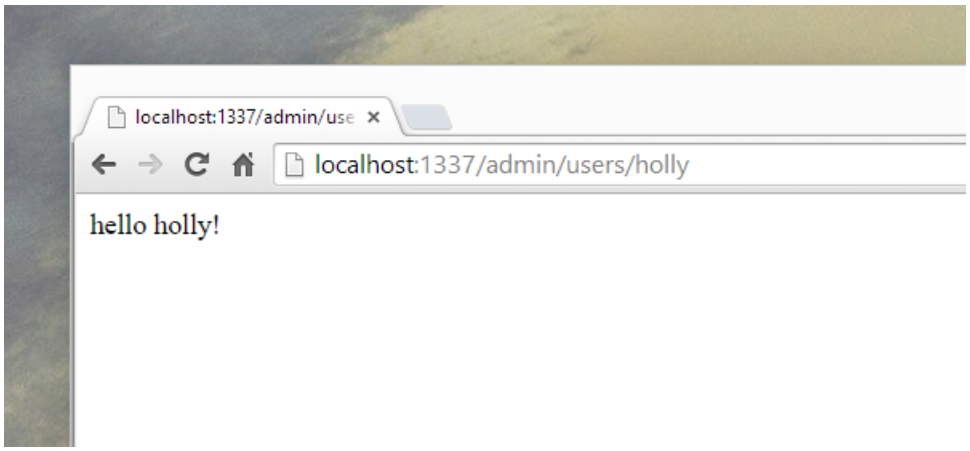
```
app.use('/', basicRoutes);
app.use('/admin', adminRoutes);
app.use('/api', apiRoutes);
```

Routes with Parameters (/hello/:name)

To see how we can add route parameters to our application, let's say we wanted to have a route called `/admin/users/:name` where we could pass in a person's name into the URL and the application would spit out **Hello *name*!** (we could also use this in a real application in order to pull the information for that user. Let's see what that sort of route would look like.

```
// route with parameters (http://localhost:1337/admin/users/:name)
adminRouter.get('/users/:name', function(req, res) {
        res.send('hello ' + req.params.name + '!');
});
```

Now we can visit `http://localhost:1337/admin/users/holly` and see our browser spit out **hello holly!** `req.params` stores all the data that comes from the original user's request. Easy cheesy.



Express Router Parameters

In the future, we could use this to grab all the user data that matches the name `holly`. We could then build an administration panel to manage our users.

Now let's say we wanted to validate this name somehow. Maybe we'd want to make sure it wasn't a naughty word. We would do that validation inside of route middleware. We'll use a special middleware for this.

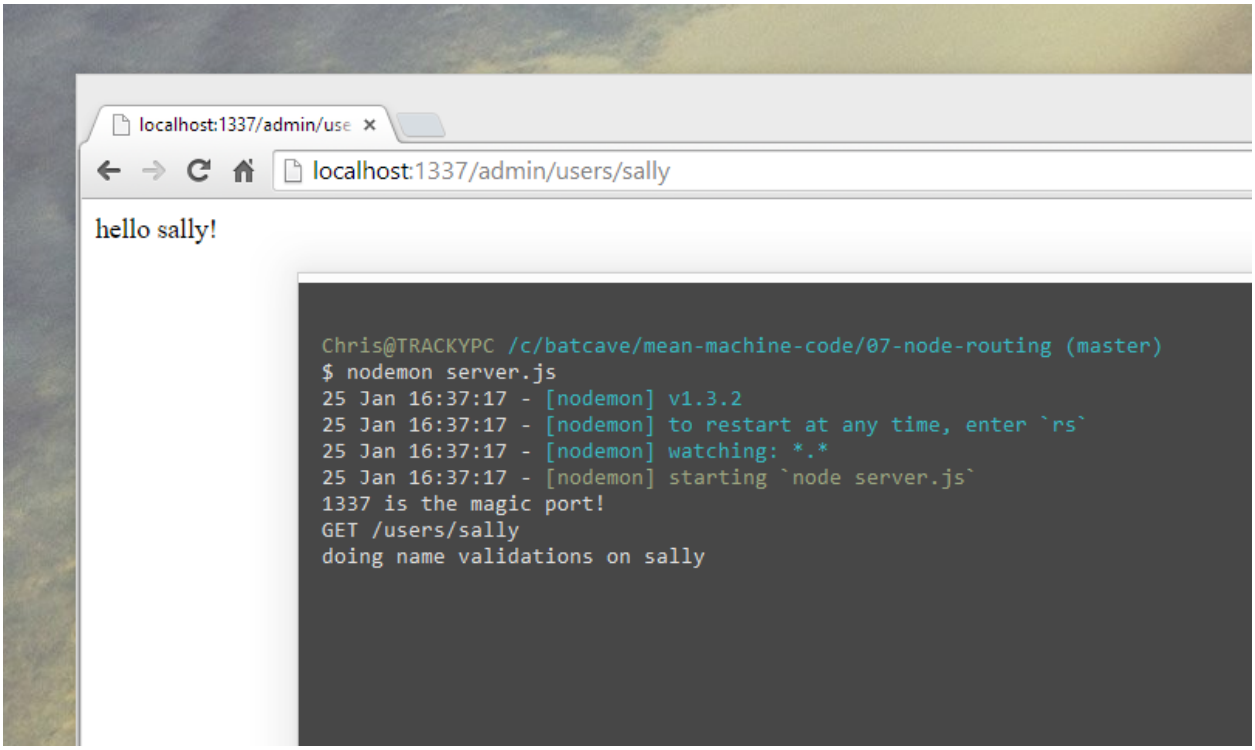
Route Middleware for Parameters (.param())

We will use Express's `.param()` middleware. This creates middleware that will run for a certain route parameter. In our case, we are using `:name` in our hello route. Here's the param middleware. Again, make sure the middleware is placed **before** the request.

```
// route middleware to validate :name
adminRouter.param('name', function(req, res, next, name) {
        // do validation on name here
        // blah blah validation
        // log something so we know its working
        console.log('doing name validations on ' + name);

        // once validation is done save the new item in the req
        req.name = name;
        // go to the next thing
        next();
});

// route with parameters (http://localhost:1337/admin/hello/:name)
adminRouter.get('/hello/:name', function(req, res) {
        res.send('hello ' + req.name + '!');
});
```

Now when we hit the **/hello/:name** route, our route middleware will kick in and be used. We can run validations and then we'll pass the new variable to our `.get` route by storing it in `req` (request). We then access it by changing `req.params.name` to `req.name` since we took `req.params.name` and stored it into `req` in the middleware.

When we visit our browser at `http://localhost:1337/admin/hello/sally` we'll see our request logged to the console.



Express Router Parameter Middleware

Route middleware for parameters can be used to validate data coming to your application. If you have created a RESTful API also, you can validate a token and make sure the user is able to access your information. All of the work we've done with Node so far will lead to building a RESTful API that will become the server-side application that we talked about Chapter 3 when we spoke about the client-server model.

The last Express router feature that we'll look at is how to use `app.route()` to define multiple routes at once.

Login Routes (app.route())

We can define our routes right on our app. This is similar to using `app.get`, but we will use `app.route`. `app.route` is basically a shortcut to call the Express Router. Instead of calling `express.Router()`, we can call `app.route` and start applying our routes there.

Using `app.route()` lets us define multiple actions on a single login route. We'll need a `GET` route to show the login form and a `POST` route to process the login form.

```
app.route('/login')

        // show the form (GET http://localhost:1337/login)
        .get(function(req, res) {
                res.send('this is the login form');
        })

        // process the form (POST http://localhost:1337/login)
        .post(function(req, res) {
                console.log('processing');
                res.send('processing the login form!');
        });
```

Now we have defined our two different actions on our `/login` route. Simple and very clean. These are applied directly to our main `app` object in the 'server.js' file, but we can also define them in the `adminRouter` object we had earlier.

This is a good method for setting up routes since it is clean and makes it easy to see which routes are applied where. We'll be building a RESTful API soon and one of the main things we should do is use different HTTP verbs to signify actions on our application. `GET /login` will provide the login form while `POST /login` will process the login.

Recap

With the Express Router, we are given much flexibility when defining routes. To recap, we can:

- Use express.Router() multiple times to define groups of routes
- Apply the express.Router() to a section of our site using app.use()
- Use route middleware to process requests
- Use route middleware to validate parameters using .param()
- Use app.route() as a shortcut to the Router to define multiple requests on a route

Now that we've got a solid foundation of Node and Express, let's move forward and talk a little more about MongoDB. Then we'll get to the main part of our server-side applications: Building APIs!

Using MongoDB

MongoDB has been touted as an easy to use database that provides high performance and availability. We've already spoken on the virtues of MongoDB in the Primers section of the book (Chapter 3).

In this chapter, we will talk about installing MongoDB and using it locally so that we are able to use it for future applications.

MongoDB

In addition to installing MongoDB locally, we will create a database on a hosted solution like Modulus.io[49] or mongolab[50].

[49] https://modulus.io/
[50] https://mongolab.com/

Mongolab

To reiterate the benefits of MongoDB in a stack like the MEAN stack, the **document-oriented storage** allows us to make development easier across the entire stack. Since we are using JSON style documents in our database, they can be used the same way all the way from the database (MongoDB) to the server (Node.js) to the front-end (AngularJS).

Let's get to the installation and simple usage of MongoDB.

Installing MongoDB Locally

There are a few steps to know when using MongoDB locally. This process is the same for the different OSes, although the small details of each step may be different. The main steps are:

1. Install MongoDB
2. Create a folder (with proper permissions) to store database data
3. Start the MongoDB service
4. Connect or use in an application after the service is started

Let's get to the installation procedures for Mac and Windows (Linux instructions can be found at the MongoDB docs[51]), and then we will move onto some basic commands that can be used within MongoDB.

[51] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

Mac Installation

The default folder when [installing on a Mac] will be **/data/db**. This is where everything will be stored and you must make sure that you have permissions to this folder so that MongoDB can write to it.

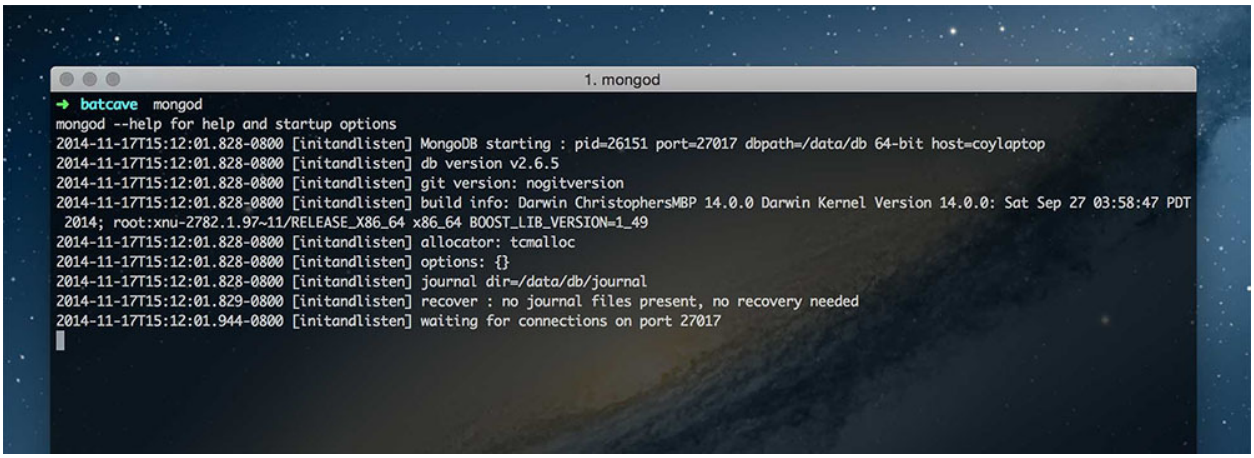
We can install on Macs using either Homebrew or manually[52]. We'll be focusing on Homebrew for this article.

```
// update your packages
$ brew update


// install mongoDB
$ brew install mongodb
```

Make sure that everything works by running:

```
$ mongod
```

This will start the MongoDB service and you should see `waiting for connections on port 27017`. If you see an error about permissions, make sure you change the permissions on your **/data/db** folder so that you have the proper permissions.



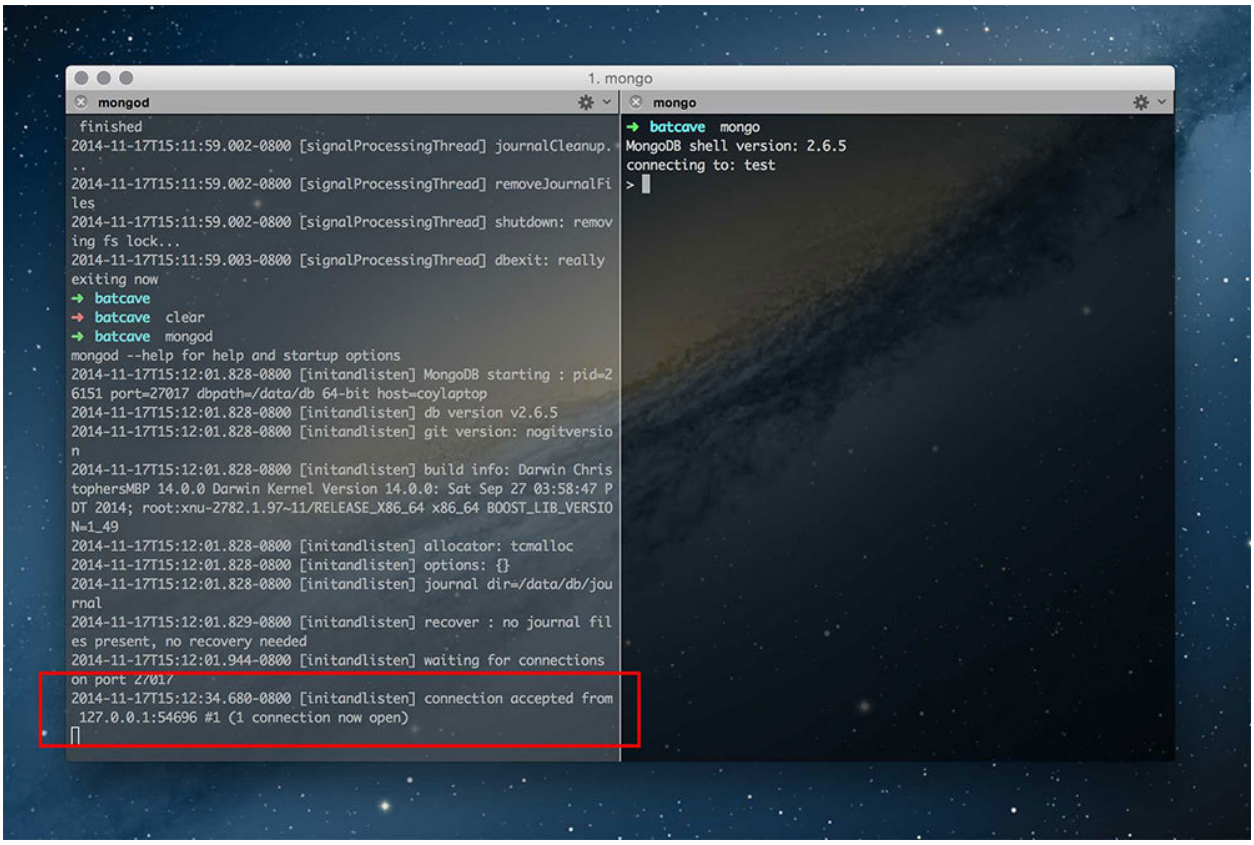
Starting MongoDB Service

Now that the MongoDB service has started, we can connect to it using:

```
mongo
```

[52] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/#install-mongodb-manually

Make sure that you run this command in a separate terminal so that we are sure the service is still running.



Connecting to MongoDB

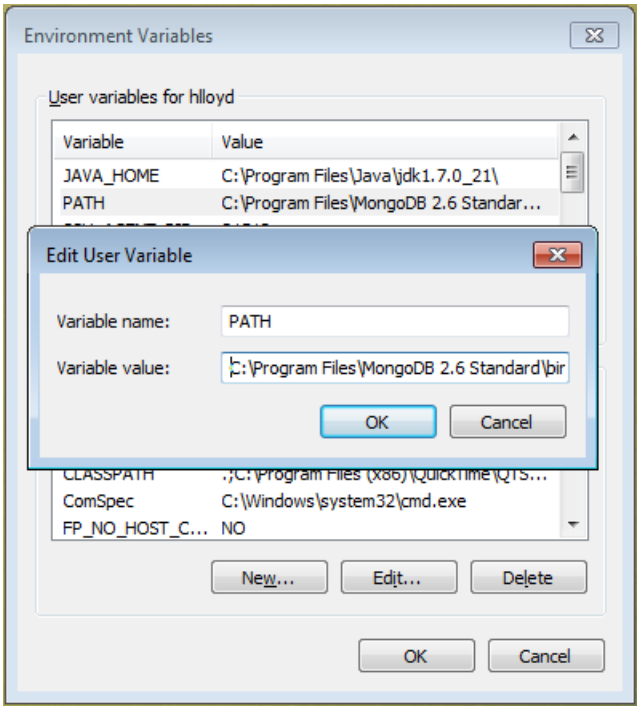
You will see the connection happen on the terminal where the service is started. Go ahead and skip past this next section on Windows installation and we'll get to using MongoDB with simple commands.

Windows Installation

Windows setup is fairly straightforward. Start by downloading MongoDB here[53]. Once that downloads, you're going to want to add the path to the MongoDB executable (usually the Program Files folder) to your environment variables so that you can run Mongo without explicitly typing the full path every single time.

Open up your Environment Variables and enter the path to the bin folder where you installed MongoDB (we are using `C:\Program Files\MongoDB 2.6 Standard\bin`). If you already have something in your PATH variable, you can add a semi-colon and insert the new path right after.

[53] http://www.mongodb.org/downloads



MongoDB Environment Variable

Now you should be able to run:

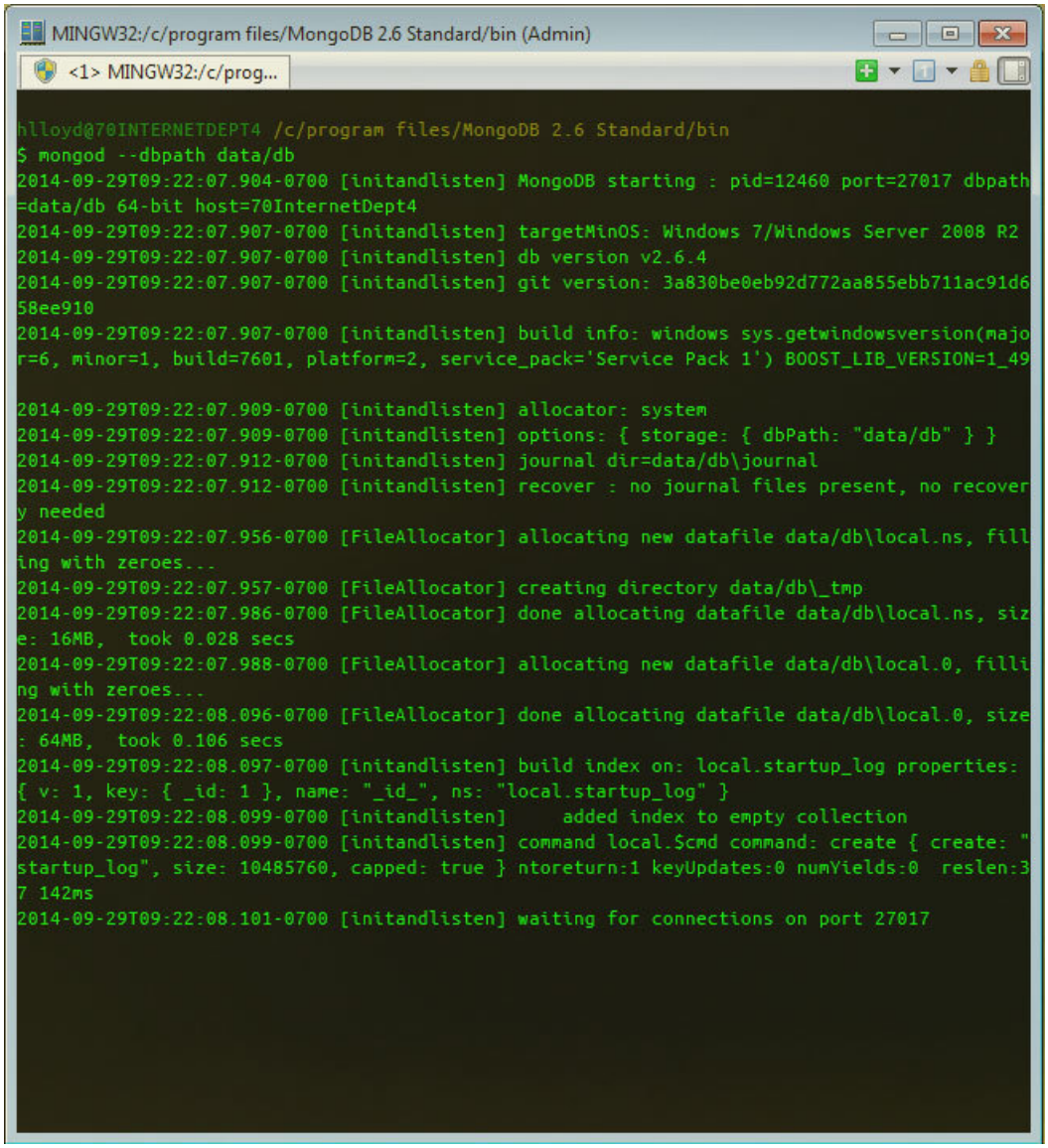
```
mongod
```

in your command line. One common error you may get is `dbpath (\data\db) does not exist`. If you receive this error, go ahead and create this path inside of your C directory 'C:\data\db'. This is where all your data will be stored.

If you would like to specify a path different than the default **data/db**, you can pass in a path like so:

```
mongod --dbpath path/to/folder
```

Once you create the path, start up mongo with 'mongod' and you should see "waiting for connections on port 27017". If you didn't create the PATH just navigate to the where your saved your executable, something like "C:Program FilesMongodb\bin" and startup your mongo connection with 'mongod.exe'.



Starting MongoDB

Now we will be able to connect to our database and run some commands! *Important:* Make sure you leave mongo running in that console and open up a new one to test out these commands.

Common Database Commands

Queries in a document database system may be a little different than what you're used to. Rather than making a query for items in tables, we're going to be querying collections of documents.

Since everything is stored in JSON style documents, our syntax will be similar to how we would query our information in our applications.

Once we have connected to our MongoDB instance using `mongod` and `mongo`, we can start viewing the available databases and using them. Here are the common commands.

List All Databases

```
$ show databases
```

Creating a Database

MongoDB will not create a database unless you insert information into that database.

The trick is you don't have to worry about explicitly creating a database! You can just use a database (even if it's not created), create a collection and document, and everything will automatically be made for you!

We'll look at creating collections and documents in the CRUD Commands section.

Show Current Database

```
$ db
```

Select a Database

```
$ use db_name
```

Now that we have the database basics, let's get to the main CRUD commands.

CRUD Commands

By using the following commands, we can get familiar with MongoDB commands. These commands will be similar to how we will handle CRUD operations in our Node.js applications.

Create

```
// save one user
$ db.users.save({ name: 'Chris' });

// save multiple users
$ db.users.save([{ name: 'Chris'}, { name: 'Holly' }]);
```

By saving a document into the `users` collection of the database you are currently in, you have successfully **created both the database and collection** if they did not already exist.

Read

```
// show all users
$ db.users.find();

// find a specific user
$ db.users.find({ name: 'Holly' });
```

Update

```
db.users.update({ name: 'Holly' }, { name: 'Holly Lloyd' });
```

Delete

```
// remove all
db.users.remove({});

// remove one
db.users.remove({ name: 'Holly' });
```

This is just a quick overview of the types of commands you can run. The MongoDB docs[54] are quite comprehensive and provide a great deal of detail for those that want to dive deeper.

MongoDB also provides a great interactive tutorial[55] for you to walk through the above commands.

GUI Tool: Robomongo

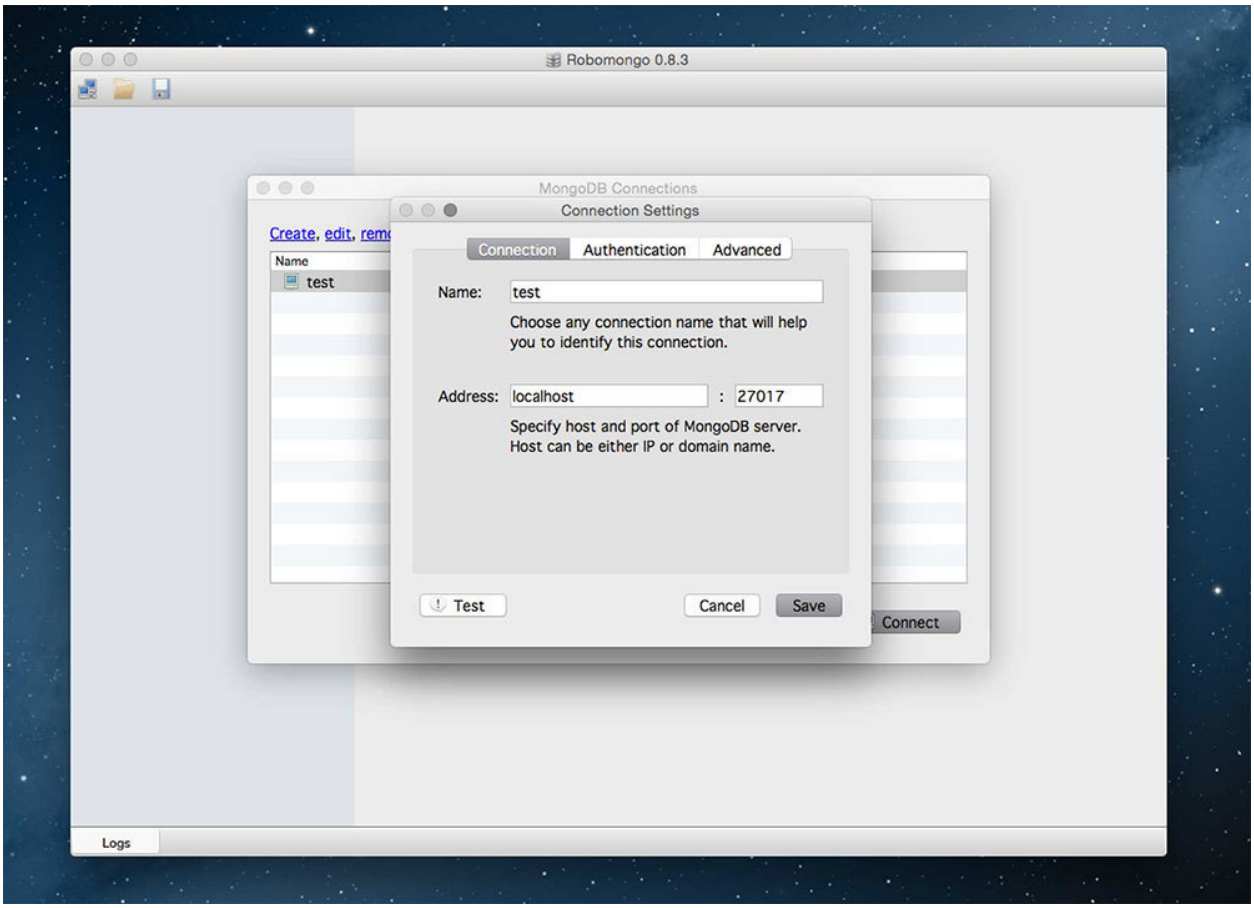
While it's easy enough to use our command line to get into our MongoDB databases, there's also a GUI for those that are inclined.

[54] http://docs.mongodb.org/manual/core/crud-introduction/

[55] http://try.mongodb.org/

Go ahead and download Robomongo[56] and fire it up.

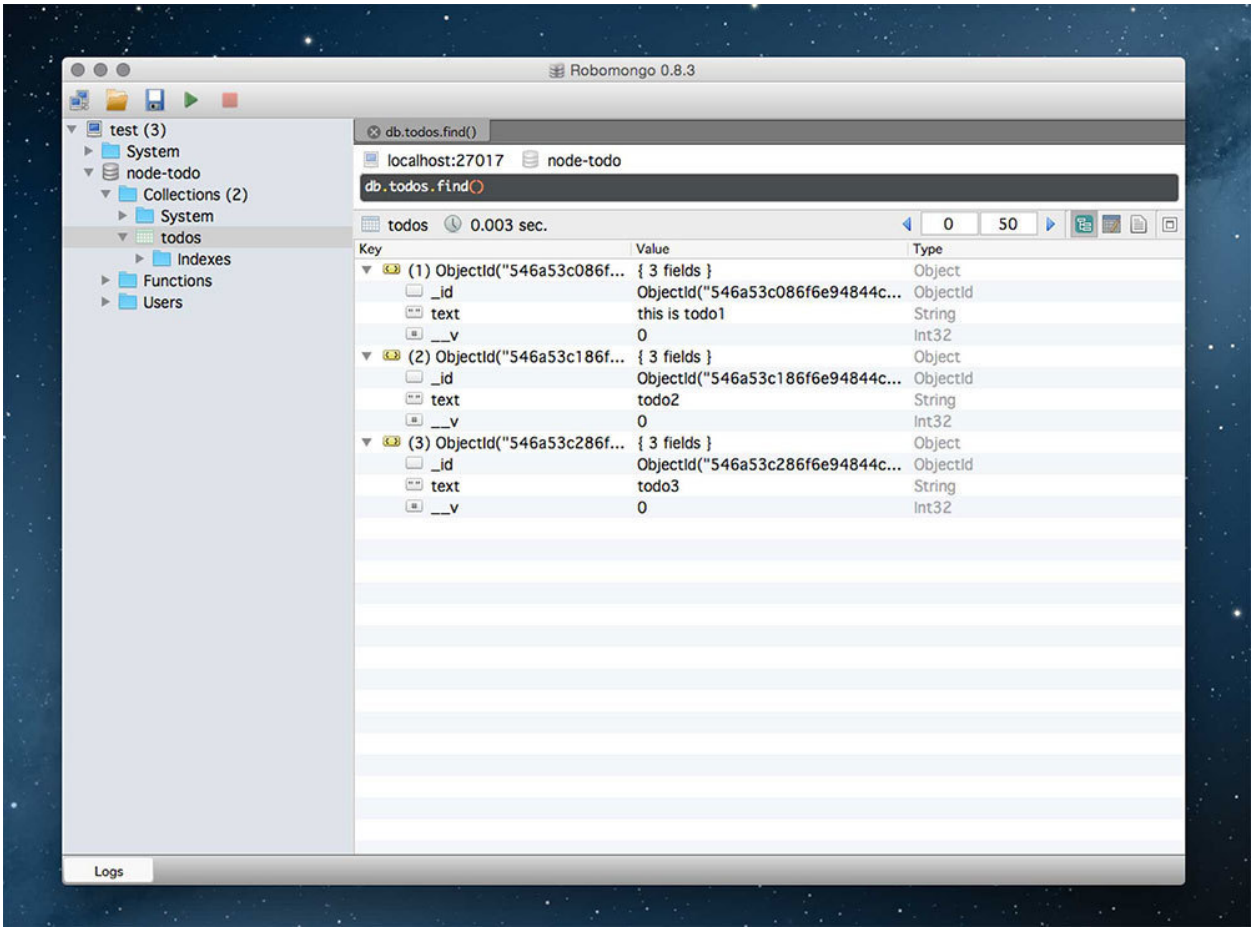
Creating a connection to our database is very easy. Just set the address to `localhost` and the port to `27017`. Name your connection anything you want.



Robomongo Connect

Then once you connect, you have access to all the databases and their respective collections! You now have a GUI to handle database operations and you can even open a shell to get down and dirty with the commands we discussed earlier.

[56] http://robomongo.org/



Robomongo Connect to Database

Using MongoDB in a Node.js Application

As you can see, using a local instance of MongoDB is a pretty straightforward process. Moving forward, feel free to either use a local instance of MongoDB or a hosted one (modulus.io[57] or mongolab.com[58]).

Either way, we will use mongooseJS[59], the Node package for working with MongoDB.

All you have to do is configure mongoose to connect to a local database. This is a simple process since we don't even need to create the database. If we are working with a local MongoDB, we just have to make sure that MongoDB is started up by running the command:

[57] https://modulus.io/
[58] https://mongolab.com/
[59] http://mongoosejs.com/

```
$ mongod
```

Connecting to a MongoDB Database Using Mongoose

Connecting to a database is a very simple process. Here's some sample code to connect to a database in Node.

```
// grab the packages we need
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/db_name');
```

That's it! Once we start saving things into our database, that database named db_name will automatically be created.

Now that we have an understanding of MongoDB and using it, let's implement CRUD and use MongoDB with mongooseJS inside of a Node.js application.

We're going to be building a Node.js API next.

Build a RESTful Node API

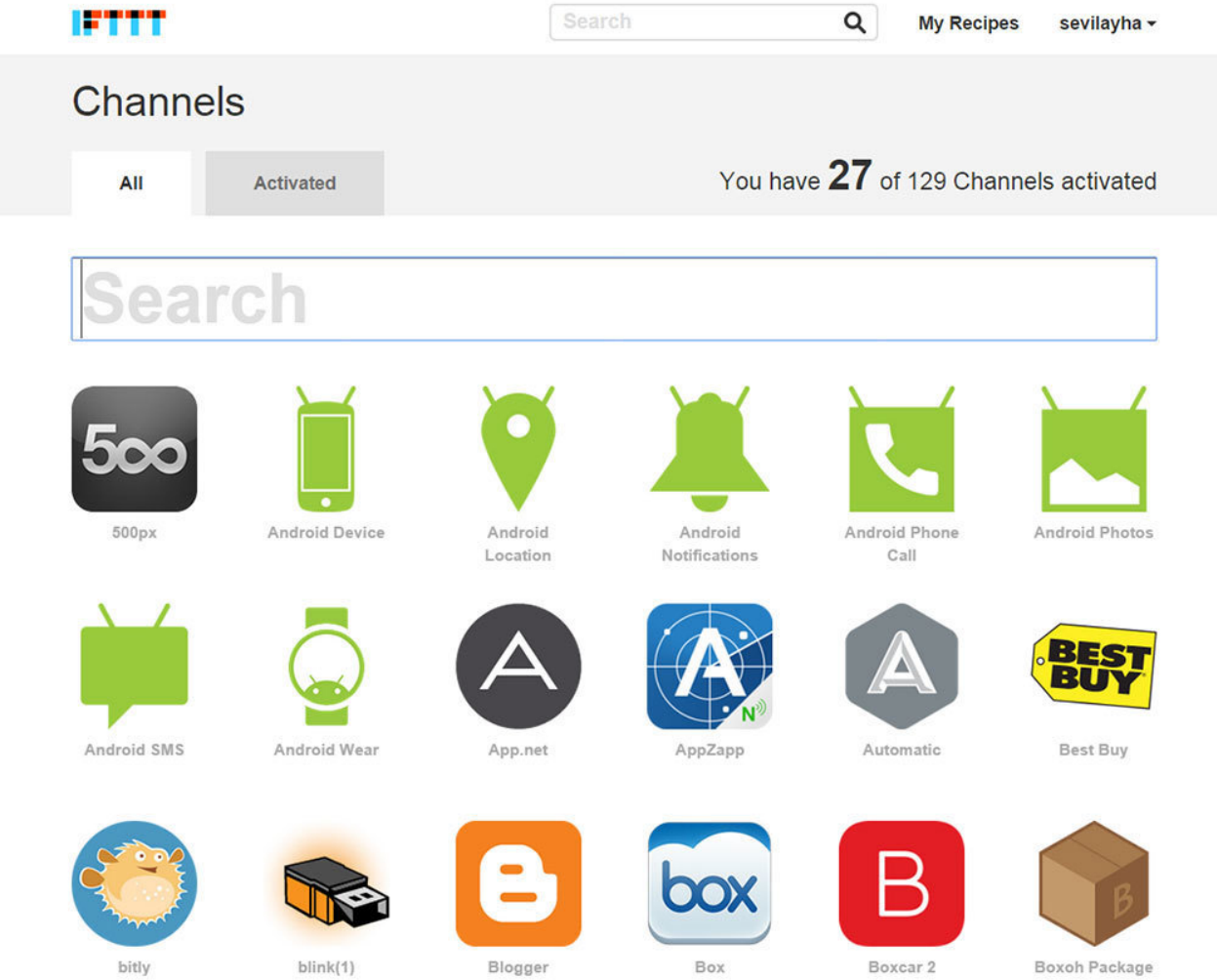
What is REST?

For a little bit on RESTful APIs, take a look at this great presentation: Teach a Dog to REST[60].

RESTful APIs[61] are becoming a standard across services on the web. It's not enough to just build an application now, we're moving towards platforms that can integrate with multiple devices and other websites.

Interconnectivity is quickly becoming the name of the game. With projects like IFTTT[62] and Zapier[63] gaining popularity, users have shown that they like their applications connected. This means that all of their applications have a standard way of "talking" to one another. IFTTT lets you set triggers and responses like "When I write a Tweet, share it as my Facebook status". The APIs of Twitter and Facebook allow us to do this.

[60] http://www.slideshare.net/landlessness/teach-a-dog-to-rest
[61] http://scotch.io/bar-talk/designing-a-restful-web-api
[62] https://ifttt.com/
[63] https://zapier.com/



IFTTT

The big players in the game like Facebook and Google have their own APIs that are used across multiple third party apps. You can even find their API Explorers where you can take their API for a test drive and see what information you can pull from the API. For instance, once you sign in, you can grab the data for all your friends from the API Explorer[64]. Using APIs has quickly become the standard for building applications.

It's not enough to have an application be standalone, you must build a website and multiple mobile apps using the same data. If your application takes off, then you'll want users to be able to build third-party applications on your data. This is where your API comes in. We'll learn how to build one using Node so that multiple applications can use that API and your data.

[64] https://developers.facebook.com/tools/explorer/145634995501895/?method=GET&path=me%3Ffields%3Did%2Cname&version=v2.1

Backend Services for our Angular Frontend

In this chapter, we'll be looking at creating a RESTful API using Node, Express 4 and its Router, and Mongoose to interact with a MongoDB instance. We will also be testing our API using Postman[65] in Chrome or RESTClient[66] in Firefox.

This will be a completely new sample application from our previous chapters and we will be using it in a few chapters when we get to our Angular frontend. Let's take a look at the API we want to build and what it can do.

Sample Application

Let's say we are going to build a CRM (Customer Relations Management) tool. This would mean that we would need to be able to manage and handle CRUD on the users in our database. Users will be the main thing we will focus on when building this API and in a few chapters, Angular will build the frontend views that will access our Users API.

Let's build an API that will:

- Handle CRUD for an item (users)
- Have a standard URL (`http://example.com/api/users` and `http://example.com/api/users/:user_id`)
- Use the proper HTTP verbs to make it RESTful (`GET`, `POST`, `PUT`, and `DELETE`)
- Return JSON data
- Log all requests to the console

Getting Started

Let's look at all the files we will need to create our API. We will need to **define our Node packages**, **start our server using Express**, **define our model**, **declare our routes using Express**, and last but not least, **test our API**.

Here is our file structure. We won't need many files and we'll keep this very simple for demonstration purposes. When moving to a production or larger application, you'll want to separate things out into a better structure (like having your routes in their own file). We'll go over file structure and best practices later in the book.

[65] https://chrome.google.com/webstore/detail/postman-rest-client-packa/fhbjgbiflinjbdggehcddcbncdddomop
[66] https://addons.mozilla.org/en-US/firefox/addon/restclient/

```
- app/
----- models/
---------- user.js         // our user model
- node_modules/     // created by npm. holds our dependencies/packages
- package.json                  // define all our node app and dependencies
- server.js                 // configure our application and create routes
```

Defining our Node Packages (package.json)

As with all of our Node projects, we will define the packages we need in `package.json`. Go ahead and create that file with these packages.

```
{
  "name": "node-api",
  "main": "server.js",
  "dependencies": {
    "morgan": "~1.5.0",
    "express": "~4.10.3",
    "body-parser": "~1.9.3",
    "mongoose": "~3.8.19",
    "bcrypt-nodejs": "0.0.3"
  }
}
```

As a shortcut, you could type the following into your command line and they would add themselves to the `package.json` file as well:

```
npm install express morgan mongoose body-parser bcrypt-nodejs --save
```

What do these packages do?

- `express` is the Node framework.
- `morgan` allows us to log all requests to the console so we can see exactly what is going on.
- `mongoose` is the ODM we will use to communicate with our MongoDB database.
- `body-parser` will let us pull POST content from our HTTP request so that we can do things like create a user.
- `bcrypt-nodejs` will allow us to hash our passwords since it is never safe to store passwords plaintext in our databases

Install the Node Packages

This might be the easiest step. Go into the command line in the root of your application and type:

```
npm install
```

npm will now pull in all the packages defined into a `node_modules` folder in our project.

Node's package manager will bring in all the packages we defined in `package.json`. Simple and easy. Now that we have our packages, let's go ahead and use them when we set up our API.

We'll be looking to our `server.js` file to setup our app since that's the main file we declared in `package.json`.

Setting Up Our Server (server.js)

Node will look here when starting the application so that it will know how we want to configure our application and API.

We will start with the bare essentials necessary to start up our application. We'll keep this code clean and commented well so we understand exactly what's going on every step of the way.

```
// BASE SETUP
// =====================================

// CALL THE PACKAGES --------------------
var express    = require('express'); // call express
var app        = express(); // define our app using express
var bodyParser = require('body-parser'); // get body-parser
var morgan     = require('morgan'); // used to see requests
var mongoose   = require('mongoose'); // for working w/ our database
var port       = process.env.PORT || 8080; // set the port for our app

// APP CONFIGURATION ---------------------
// use body parser so we can grab information from POST requests
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// configure our app to handle CORS requests
app.use(function(req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST');
  res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type, \
Authorization');
  next();
});

// log all requests to the console
```

```
27 app.use(morgan('dev'));
28
29 // ROUTES FOR OUR API
30 // =============================
31
32 // basic route for the home page
33 app.get('/', function(req, res) {
34         res.send('Welcome to the home page!');
35 });
36
37 // get an instance of the express router
38 var apiRouter = express.Router();
39
40 // test route to make sure everything is working
41 // accessed at GET http://localhost:8080/api
42 apiRouter.get('/', function(req, res) {
43         res.json({ message: 'hooray! welcome to our api!' });
44 });
45
46 // more routes for our API will happen here
47
48 // REGISTER OUR ROUTES -------------------------------
49 // all of our routes will be prefixed with /api
50 app.use('/api', apiRouter);
51
52 // START THE SERVER
53 // =============================
54 app.listen(port);
55 console.log('Magic happens on port ' + port);
```

Wow we did a lot there! It's all very simple though so let's walk through it a bit.

Base Setup: In our base setup, we pull in all the packages we pulled in using npm. We'll grab express, mongoose, define our app, get bodyParser and configure our app to use it. We can also set the port for our application and grab the user model that we will define later.

We are setting our configuration to allow requests from other domains to prevent CORS errors. This allows any domain to access our API.

Routes for Our API: This section will hold all of our routes. The structure for using the Express Router let's us pull in an instance of the router. We can then define routes and then apply those routes to a root URL (in this case, API). There is also a home page route to say hello; this is just a basic route to make sure that everything is working.

Start the Server: We'll have our express app listen to the port we defined earlier. Then our application will be live and we can test it!

Starting the Server and Testing

Let's make sure that everything is working up to this point. We will start our Node app and then send a request to the one route we defined to make sure we get a response.

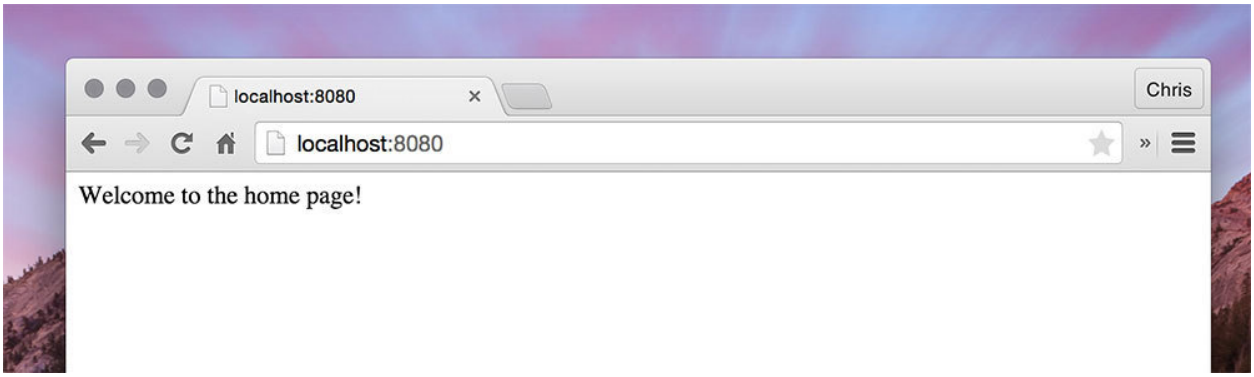
Let's start our server. From the command line, type:

```
1 nodemon server.js
```

You should see your Node app start up and Express will create a server.

Starting Node Server

Now that we know our application is up and running, let's test it. You should be able to see our message returned by visiting the basic route: `http://localhost:8080`. Now let's test the rest of our routes (the API routes).

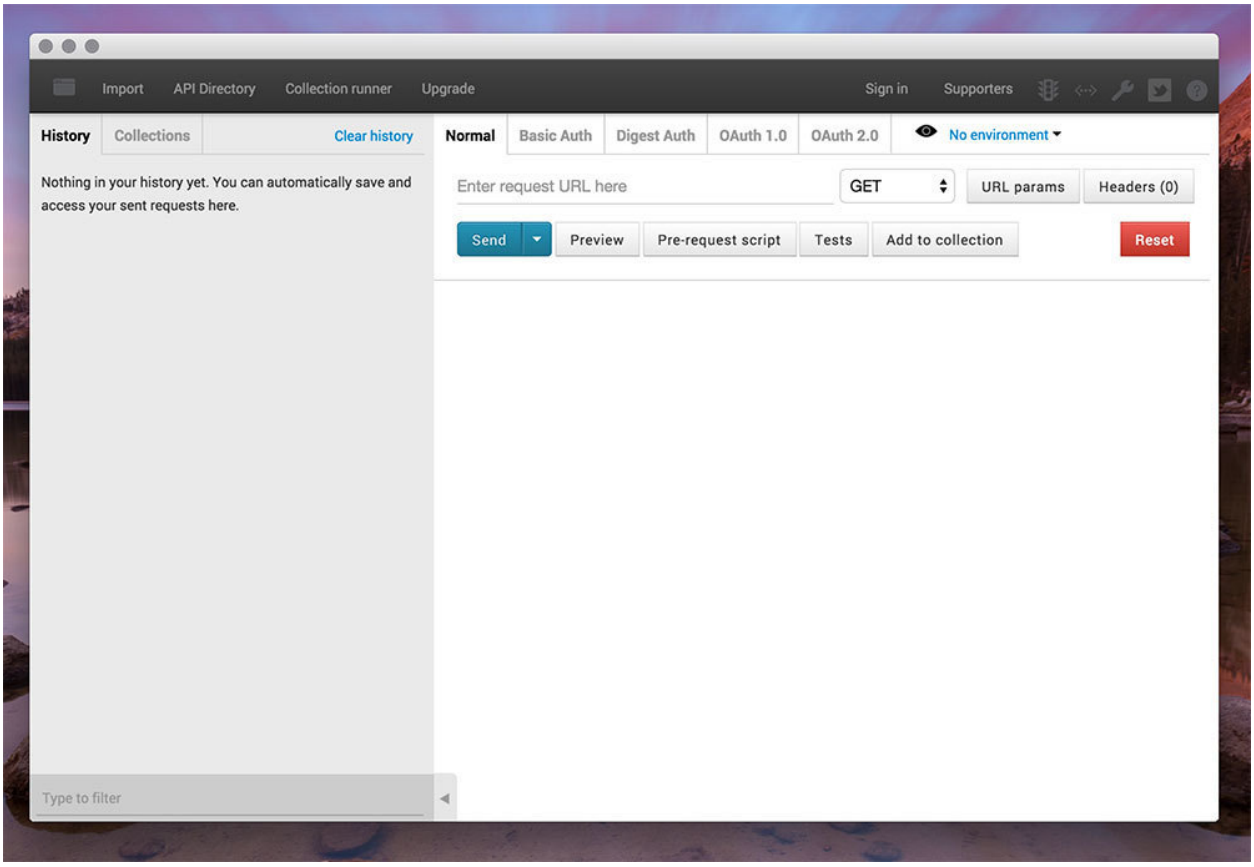


Basic Route

Testing Our API Using Postman

Now navigate to `http://localhost:8080/api` in your browser and you will see our JSON message. Seeing it is great and all, but the best way to test out our API is to use a tool built for this sort of thing. Postman will help us test our API. It will basically send HTTP requests to a URL of our choosing. We can even pass in parameters (which we will soon).

Open up Postman[67] in Chrome and let's walk through how to use it.

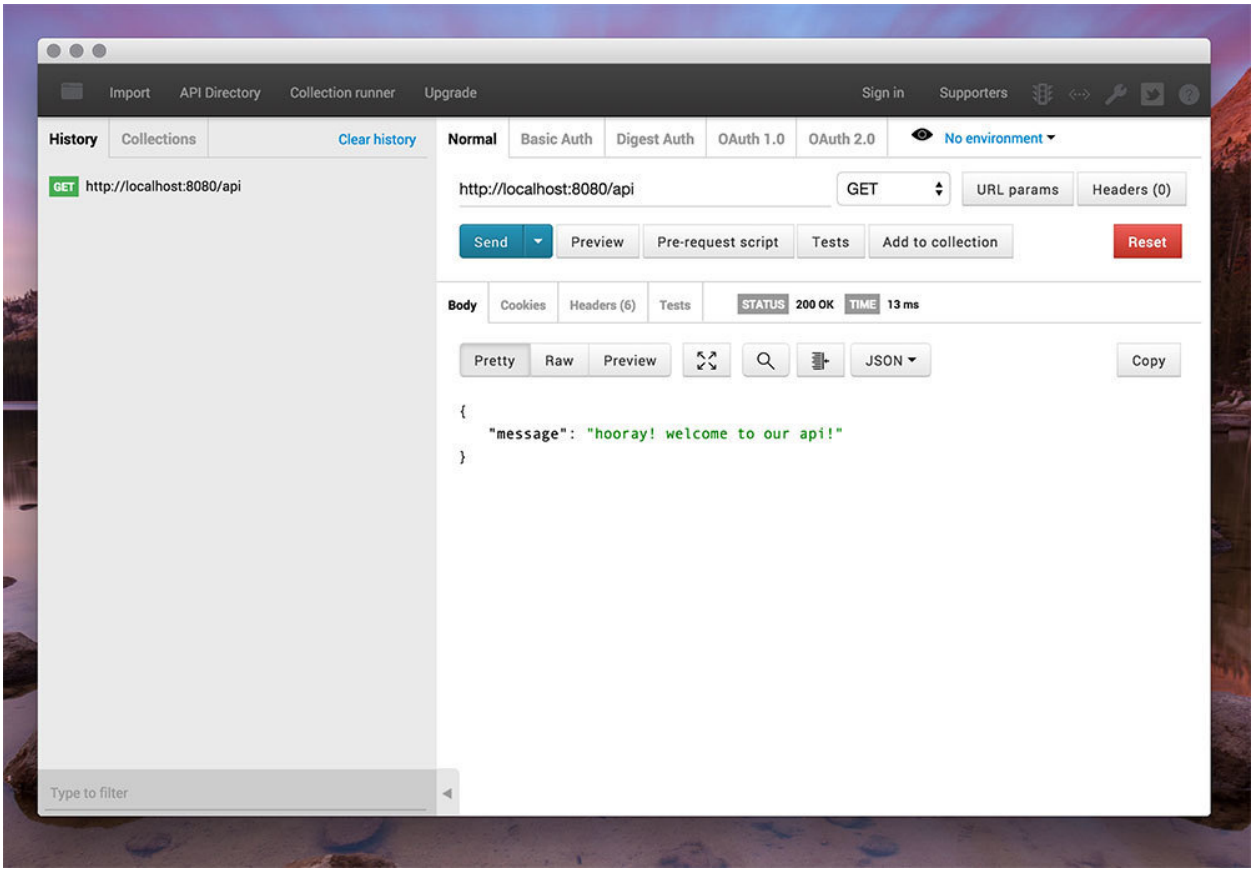


Postman REST Client Node API

All you have to do is **enter your request URL**, **select an HTTP verb**, and click Send. Simple enough right? We can even see a history of the past calls we've made so that we can switch back and forth between calls we need to test. Think creating and deleting records on the fly.

Here's the moment we've been waiting for. Does our application work the way we configured it? Enter `http://localhost:8080/api` into the URL. `GET` is what we want since we just want to get data. Now click **Send**.

[67] https://chrome.google.com/webstore/detail/postman-rest-client-packa/fhbjgbiflinjbdggehcddcbncdddomop



Node API Postman Test

Sweet! We got back exactly what we wanted. Now we know we can serve information to requests. Let's wire up our database so we can start performing CRUD operations on some users.

Database and User Model

We'll keep this short and sweet so that we can get to the fun part of building the API routes. All we need to do is create a MongoDB database and have our application connect to it. We will also need to create a user mongoose model so we can use mongoose to interact with our database.

Creating Our Database and Connecting

We will be using a database provided by Modulus[68]. You can definitely create your own database and use it locally or use online database providers like Modulus[69] or Mongolab[70].

Once you have your database created and have the URI to connect to (`mongodb://node:node@novus.modulusmongo.r`) let's add it to our application. In `server.js` in the Base Setup section, let's add these two lines.

[68] https://modulus.io/
[69] https://modulus.io/
[70] https://mongolab.com/

```
// server.js

// BASE SETUP
// ============================

...

// connect to our database (hosted on modulus.io)
mongoose.connect('mongodb://node:noder@novus.modulusmongo.net:27017/Iganiq8o');

...
```

All you need is a URI like above so that your application can connect. This URI contains the host (`novus.modulusmongo.net`), the port (`27017`), the database (`Iganiq8o`), and the user and password (`node@noder`).

You can also create your MongoDB instance locally like we did in Chapter 8. Your `mongoose.connect()` url would then be: `mongoose.connect(mongodb://localhost:27017/myDatabase`.

Since we already grabbed the mongoose package earlier with 'npm install', we just need to connect to our remote database hosted by Modulus or locally. Now that we are connected to our database, let's create a mongoose model to handle our users.

User Model (app/models/user.js)

Since the model won't be the focus of this tutorial, we'll just create a model and provide our users with a name field. That's it. Let's create that file and define the model.

```
// grab the packages that we need for the user model
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var bcrypt = require('bcrypt-nodejs');

// user schema
var UserSchema   = new Schema({
        name: String,
    username: { type: String, required: true, index: { unique: true }},
    password: { type: String, required: true, select: false }
});

// hash the password before the user is saved
UserSchema.pre('save', function(next) {
        var user = this;
```

```
16
17          // hash the password only if the password has been changed or user is new
18          if (!user.isModified('password')) return next();
19
20          // generate the hash
21          bcrypt.hash(user.password, null, null, function(err, hash) {
22                  if (err) return next(err);
23
24                  // change the password to the hashed version
25                  user.password = hash;
26                  next();
27          });
28  });
29
30  // method to compare a given password with the database hash
31  UserSchema.methods.comparePassword = function(password) {
32          var user = this;
33
34          return bcrypt.compareSync(password, user.password);
35  };
36
37  // return the model
38  module.exports = mongoose.model('User', UserSchema);
```

There is a lot happening here, but it's fairly simple. We must **create our Schema**. We are defining *name*, *username*, and *password* as **String**s. By setting the `index` and `unique` attributes, we are telling Mongoose to create a unique index for this path. This means that a username can not be duplicated. Another mongoose feature that we will implement is `select: false` on the password attribute. When we query the list of users or a single user, there will be no need to provide the password. By setting `select` to `false`, the password will not be returned when listing our users, unless it is explicitly called.

We are also creating a function using `pre` that will ensure that the password is hashed before we save the user.

> It is always important to make sure that we are not saving plaintext passwords to the database. We have to make sure our applications are as secure as can be from the start.

With mongoose, we are also creating a method on our user model to compare the password with a given hash. This is how we will authenticate our users in Chapter 10.

With that file created, let's pull it into our `server.js` so that we can use it within our application. We'll add one more line to that file and use `require` like we have before to pull in features.

```
// server.js

// BASE SETUP
// ==========================

var User     = require('./app/models/user');
```

Now our entire application is ready and wired up so we can start building out our routes. These routes will define our API, which is the main reason this chapter exists. Onward!

Express Router and Routes

We will use an instance of the Express Router to handle all of our API routes. Here is an overview of the routes we will require, what they will do, and the HTTP Verb used to access it.

/api/users	GET	Get all the users
/api/users	POST	Create a user
/api/users/:user_id	GET	Get a single user
/api/users/:user_id	PUT	Update a user with new info
/api/users/:user_id	DELETE	Delete a user

This will cover the basic routes needed for an API. This also keeps to a good format where we have kept the actions we need to execute (GET, POST, PUT, and DELETE) as HTTP verbs.

Route Middleware

We've already defined our first route and seen it in action. The Express Router gives us a great deal of flexibility in defining our routes.

We've talked on route middleware in the Node Routing chapter. For this example we are just going to `console.log()` the request to the console. Let's add that middleware to our 'server.js' file now.

```
// ROUTES FOR OUR API
// =============================
var apiRouter = express.Router();         // get an instance of the express Router

// middleware to use for all requests
apiRouter.use(function(req, res, next) {
        // do logging
        console.log('Somebody just came to our app!');

```

```
10          // we'll add more to the middleware in Chapter 10
11          // this is where we will authenticate users
12
13          next(); // make sure we go to the next routes and don't stop here
14  });
15
16  // test route to make sure everything is working
17  // (accessed at GET http://localhost:8080/api)
18  apiRouter.get('/', function(req, res) {
19          res.json({ message: 'hooray! welcome to our api!' });
20  });
21
22  // more routes for our API will happen here
23
24  // REGISTER OUR ROUTES -------------------------------
25  // all of our routes will be prefixed with /api
26  app.use('/api', apiRouter);
```

All we needed to do to declare that middleware was to use `router.use(function())`. The order of how we define the parts of our router is very important. They will run in the order that they are listed like we talked about earlier.

We are sending back information as **JSON data**. This is standard for an API and the people using your API will be eternally grateful.

We will also add `next()` to indicate to our application that it should continue to the other routes or next middleware. This is important because our application would stop at this middleware without it.

Now when we send a request to our application using Postman, the request will be logged to our Node console (the command line).

Middleware Uses Using middleware like this can be very powerful. We can do validations to make sure that everything coming from a request is safe and sound. We can throw errors here in case something in the request is wrong. We can do some extra logging for analytics or any statistics we'd like to keep. And the big usage is to authenticate the API calls by checking if a user has the correct token, which we will do in the next chapter.

Creating the Basic Routes

We will now create the routes to handle **getting all the users** and **creating a user**. This will both be handled using the `/api/users` route. We'll look at creating a user first so that we have users to work with.

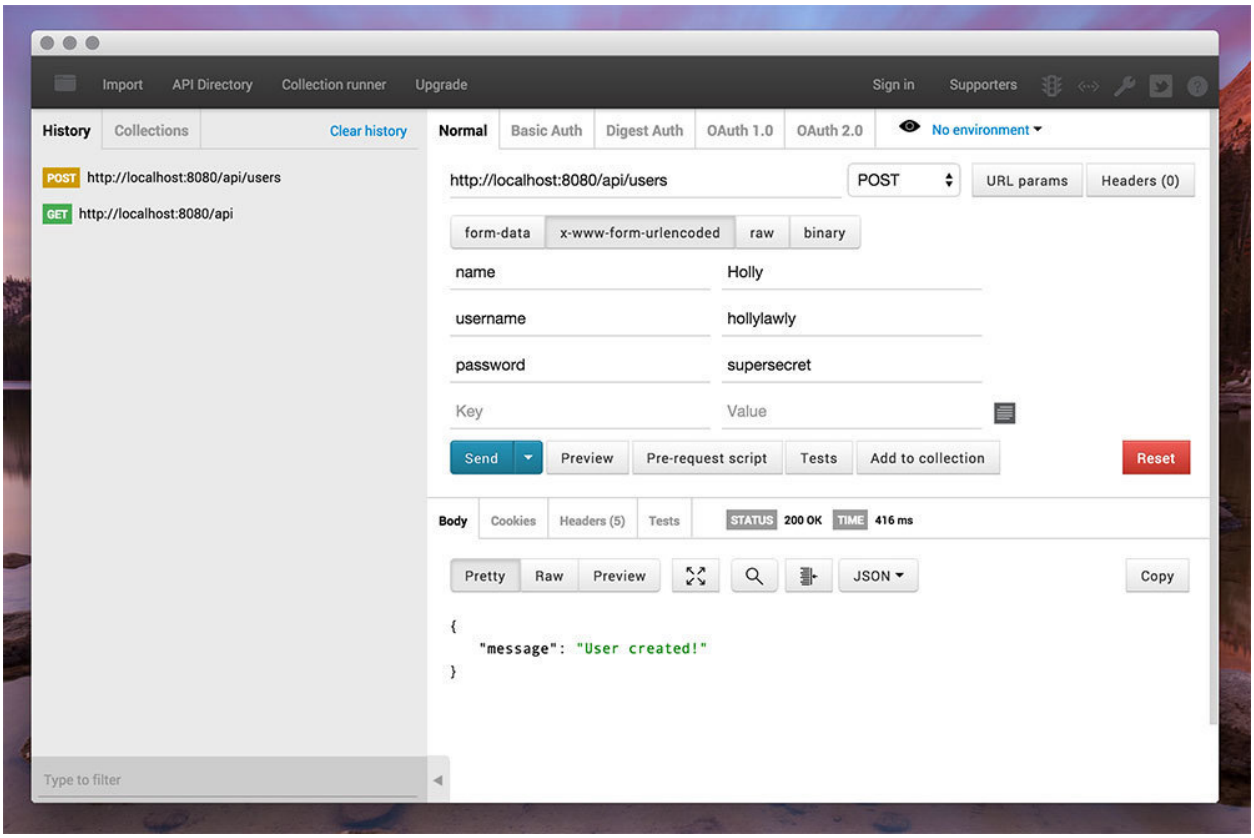
Creating a User POST (/api/users)

We will add the new route to handle POST and then test it using Postman.

```
// ROUTES FOR OUR API
// =============================

// route middleware and first route are here

// on routes that end in /users
// ----------------------------------------------------
apiRouter.route('/users')

        // create a user (accessed at POST http://localhost:8080/api/users)
        .post(function(req, res) {

                // create a new instance of the User model
                var user = new User();

                // set the users information (comes from the request)
                user.name = req.body.name;
                user.username = req.body.username;
                user.password = req.body.password;

                // save the user and check for errors
                user.save(function(err) {
            if (err) {
                // duplicate entry
                if (err.code == 11000)
                    return res.json({ success: false, message: 'A user with that\
 username already exists. '});
                else
                    return res.send(err);
            }

                        res.json({ message: 'User created!' });
                });

        })
```

Now we have created the `POST` route for our application. We will use Express's `apiRouter.route()` to handle multiple routes for the same URI. We are able to handle all the requests that end in `/users`.

Let's look at Postman now to create our user. We are passing the `name` variable. This is how the information would look if it came from a form on our website or application.

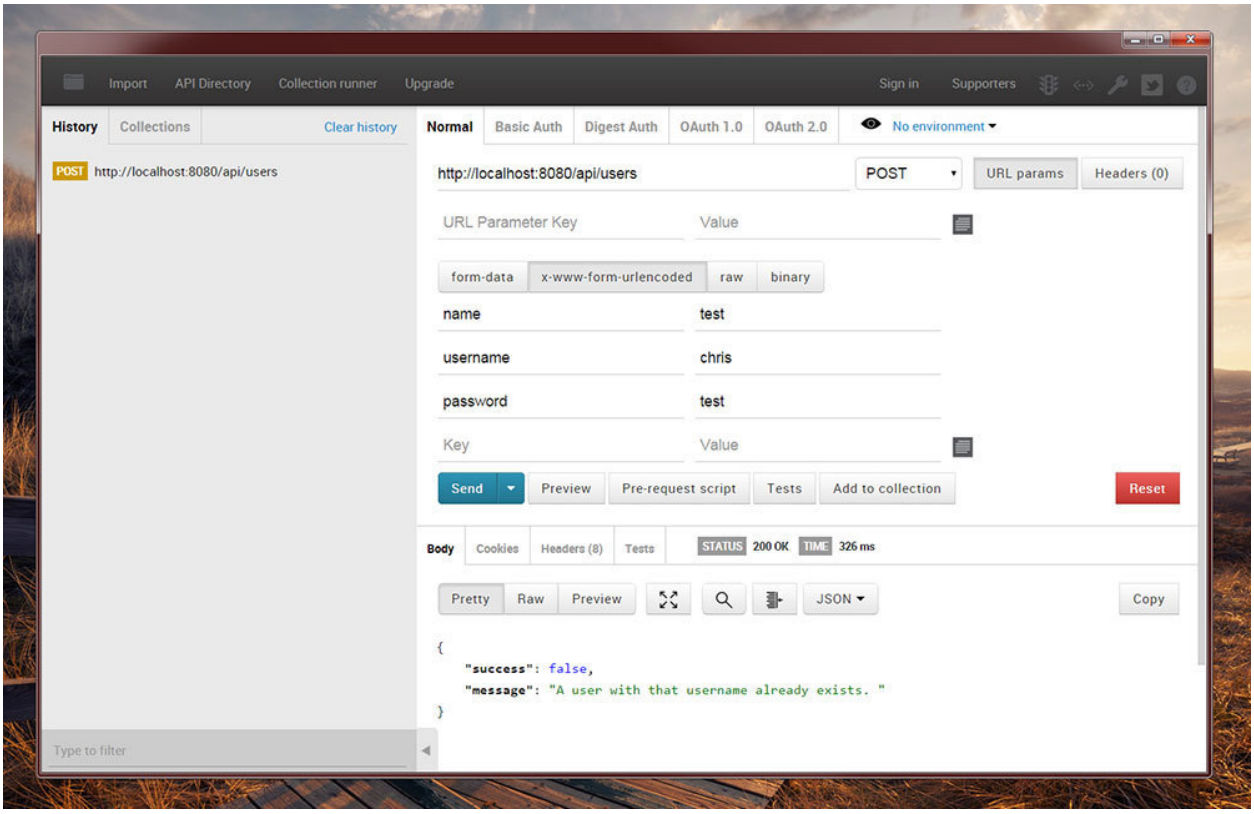


Node API Postman POST Create User

Notice that we are sending the `name` data as `x-www-form-urlencoded`. This will send all of our data to the Node server as query strings.

We get back a successful message that our user has been created. Let's handle the API route to get all the users so that we can see the user that just came into existence.

If the username was already used, mongoose will spit out an error code of `11000`. We are going to check for that error code if there is an error and return the message that `'A user with that username already exists.`.



Duplicate User

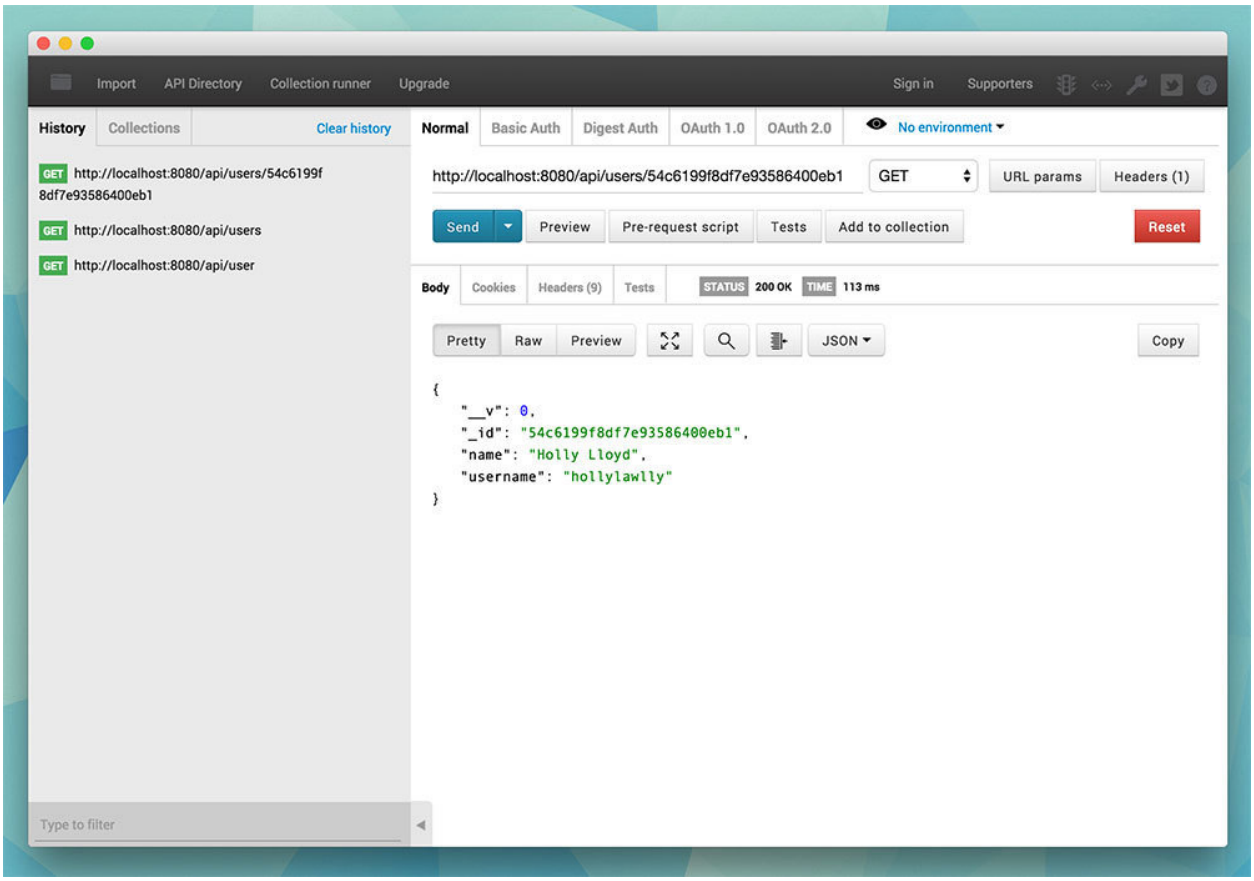
Getting All Users (GET /api/users)

This will be a simple route that we will add to 'server.js' onto the `router.route('/users')` we created for the POST. With `router.route()`, we are able to chain together the different routes using different HTTP actions. This keeps our application clean and organized.

```
1  // <-- route middleware and first route are here
2
3  // on routes that end in /users
4  // ----------------------------------------------------
5  apiRouter.route('/users')
6
7          // create a user (accessed at POST http://localhost:8080/api/users)
8          .post(function(req, res) {
9                  ...
10         })
11
12         // get all the users (accessed at GET http://localhost:8080/api/users)
13         .get(function(req, res) {
```

```
14                  User.find(function(err, users) {
15                          if (err) res.send(err);
16
17                          // return the users
18                          res.json(users);
19                  });
20          });
```

Straightforward route. Just send a `GET` request to `http://localhost:8080/api/users` and we'll get all the users back in JSON format.



Node API Postman GET All

We can see that Holly is in our database and her password has even been hashed thanks to `bcrypt` and that function that we created that works whenever we save a user to the database. Remember that `_id` that was automatically generated for Holly here. We will use it when we create the route to get a single user.

Creating Routes for A Single Item

We've handled the group for routes ending in `/users`. Let's now handle the routes for when we pass in a parameter like a user's id.

The things we'll want to do for this route, which will end in `/users/:user_id` will be:

- Get a single user.
- Update a user's info.
- Delete a user.
- The `:user_id` from the request will be accessed thanks to that `body-parser` package we called earlier.

Getting a Single User (GET /api/users/:user_id)

We'll add another `apiRouter.route()` to handle all requests that have a `:user_id` attached to them. Again, drop it into your 'server.js' file after the 'apiRouter.route('/users')' chunk.

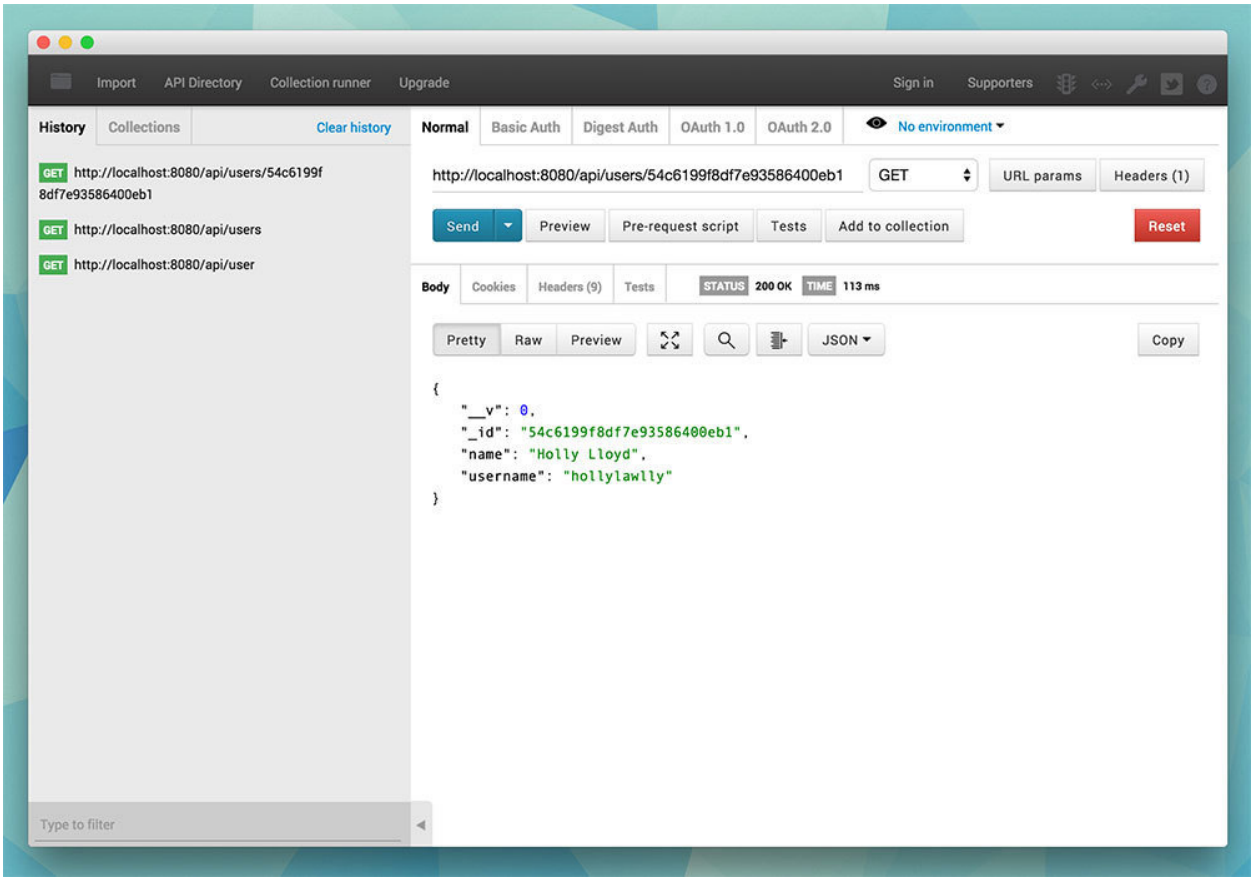
```
1  // on routes that end in /users/:user_id
2  // ----------------------------------------------------
3  apiRouter.route('/users/:user_id')
4
5          // get the user with that id
6          // (accessed at GET http://localhost:8080/api/users/:user_id)
7          .get(function(req, res) {
8                  User.findById(req.params.user_id, function(err, user) {
9                          if (err) res.send(err);
10
11                         // return that user
12                         res.json(user);
13                 });
14         })
```

From our call to get all the users, we can a list of all of our users, each with a unique id. Let's grab one of those ids and test it with Postman.

Enter 'http://localhost:8080/api/users/5481e26e93b9bba7171bfd5e', select GET and hit send.

And this will return our pretty user data. Easy peezy!

```
{
    "password": "$2a$10$rgraupzsDZHbY77CQSInHuZ90RWvq/NWsi0kOueePetqUBvj2.jKa",
    "username": "chris",
    "name": "Chris",
    "_id": "5481e26e93b9bba7171bfd5e",
    "__v": 0
}
```



Node API Postman Get Single User

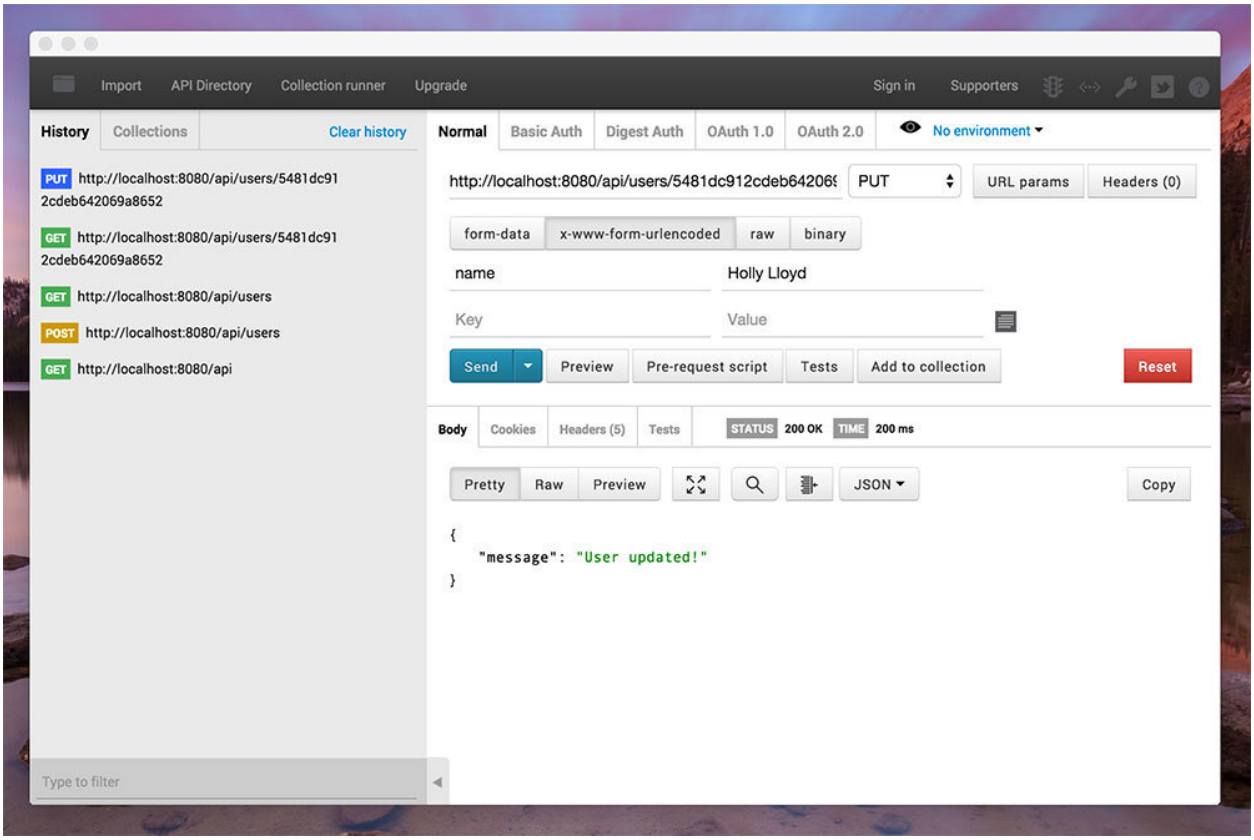
Now that we can grab a user from our API, let's look at updating a user's name. Let's say he wants to be more sophisticated, so we'll rename him from Klaus to **Sir Klaus**.

Updating a User's Info (PUT /api/users/:user_id)

Let's chain a route onto our this `router.route()` and add a `.put()`.

```
// on routes that end in /users/:user_id
// ----------------------------------------
apiRouter.route('/users/:user_id')

        // get the user with that id
        // (accessed at GET http://localhost:8080/api/users/:user_id)
        .get(function(req, res) {
                ...
        })

        // update the user with this id
        // (accessed at PUT http://localhost:8080/api/users/:user_id)
        .put(function(req, res) {

                // use our user model to find the user we want
                User.findById(req.params.user_id, function(err, user) {

                        if (err) res.send(err);

                        // update the users info only if its new
                        if (req.body.name) user.name = req.body.name;
                        if (req.body.username) user.username = req.body.username;
                        if (req.body.password) user.password = req.body.password;

                        // save the user
                        user.save(function(err) {
                                if (err) res.send(err);

                                // return a message
                                res.json({ message: 'User updated!' });
                        });

                });
        })
```

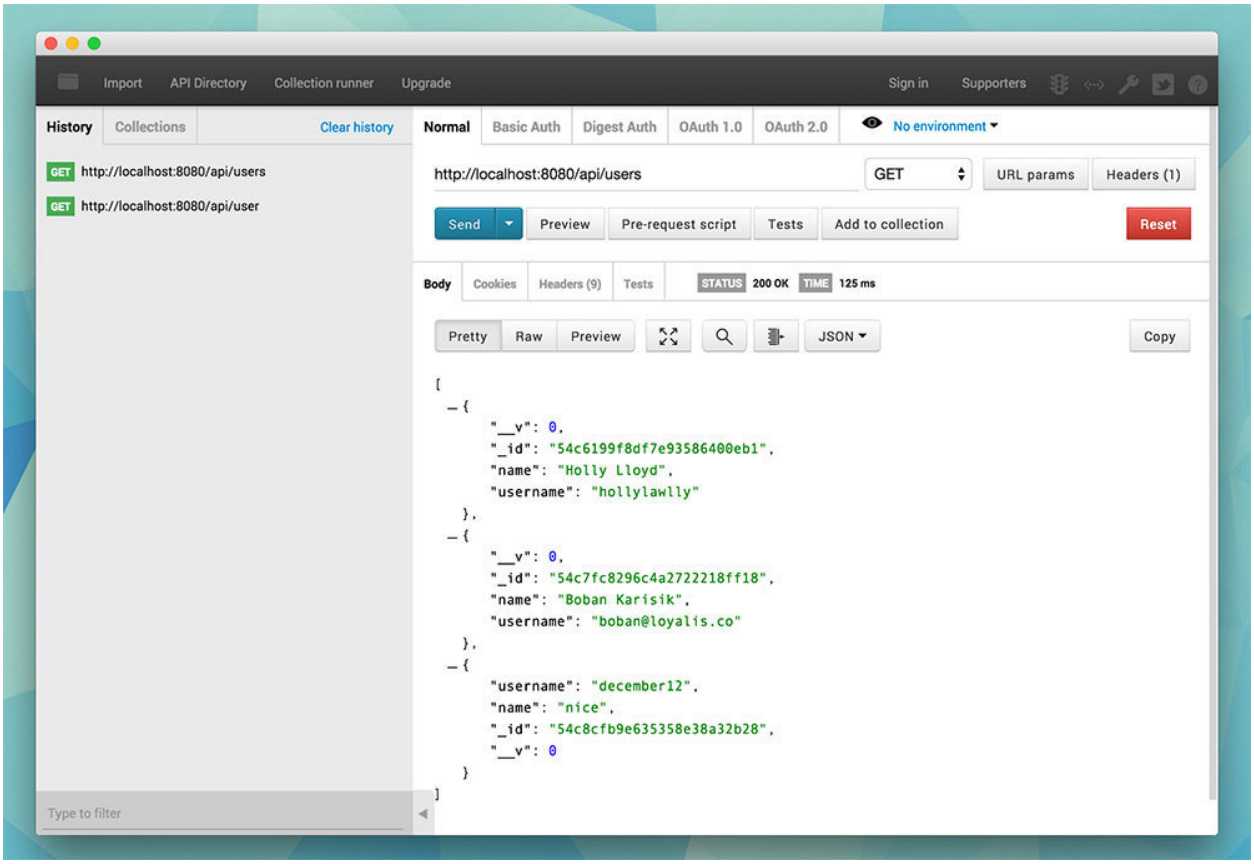
We will use the given `id` from the PUT request, grab that user, make changes, and save him back to the database. It's important to note that we will only change the name, username, or password in the database if it has actually changed.



Node API Postman Update Record

Since we added the conditionals in our PUT route, if the username or password hasn't been passed through, those fields won't be updated. We wouldn't want those updated to be blanks if they weren't passed in.

We can also use the `GET /api/users` call we used earlier to see that her name has changed.



Node API Postman All Updated

Holly's name in the database has now been changed from **Holly** to her full name, **Holly Lloyd**. That was all fun while it lasted. Now let's delete Holly. :(

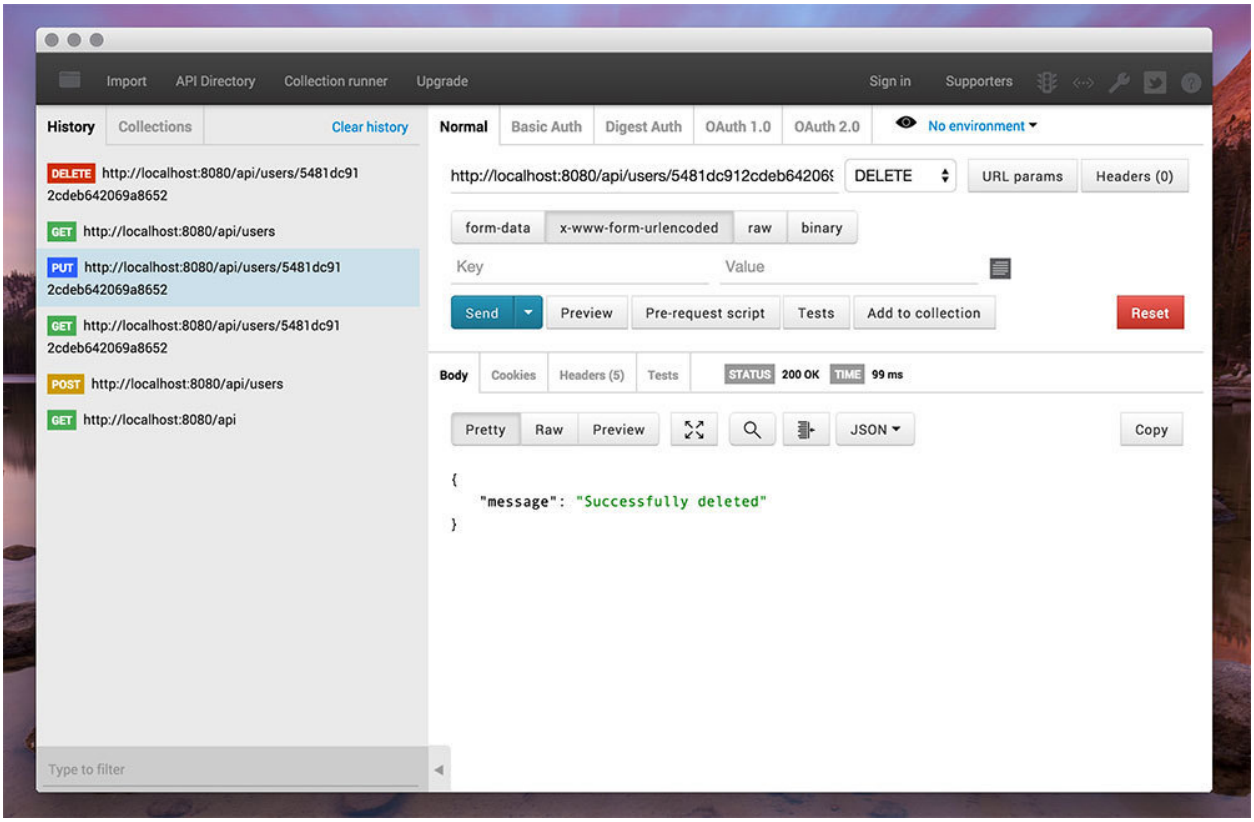
Deleting a User (DELETE /api/users/:user_id)

When someone requests that a user is deleted, all they have to do is send a DELETE to `/api/users/:user_id`

Let's add the code for deleting users.

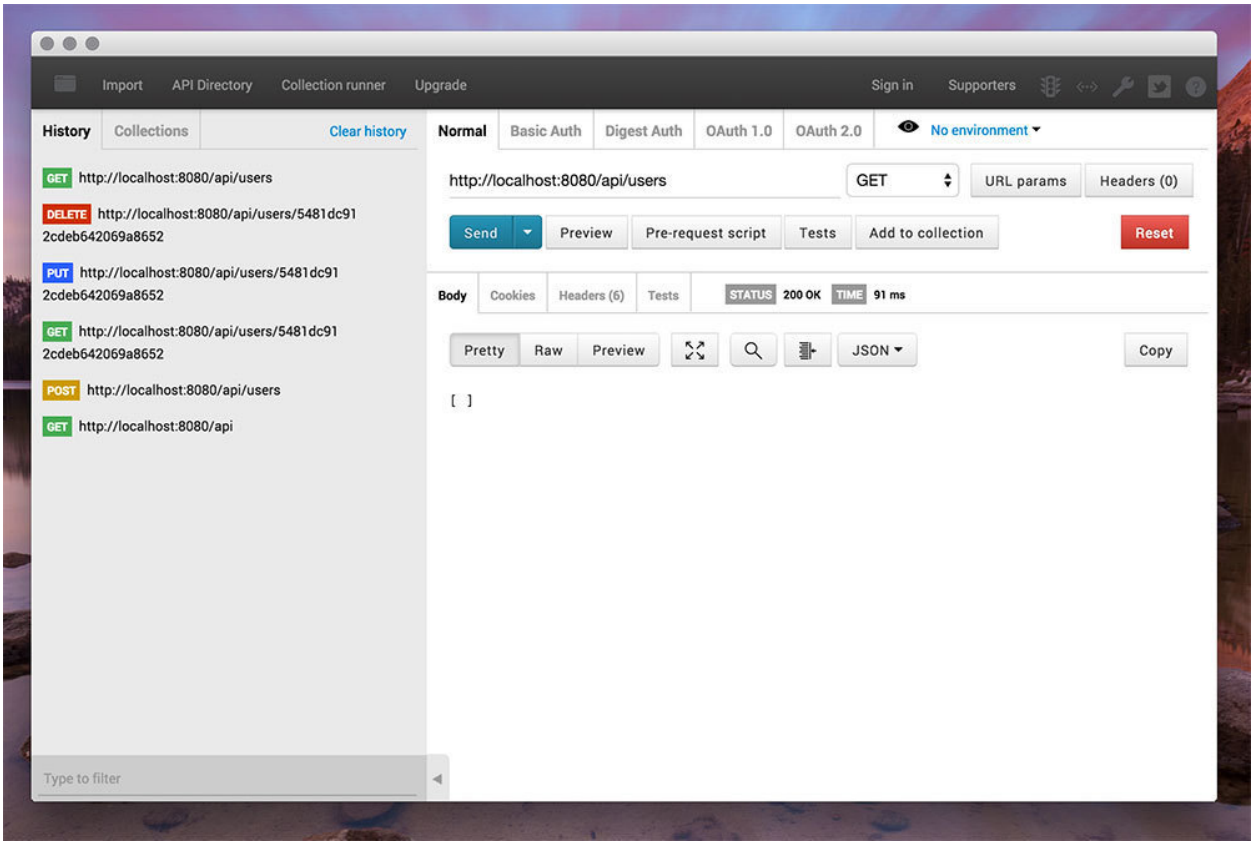
```
// on routes that end in /users/:user_id
// ----------------------------------------------------
apiRouter.route('/users/:user_id')

        // get the user with that id
        // (accessed at GET http://localhost:8080/api/users/:user_id)
        .get(function(req, res) {
                ...
        })

        // update the user with this id
        // (accessed at PUT http://localhost:8080/api/users/:user_id)
        .put(function(req, res) {
                ...
        })

        // delete the user with this id
        // (accessed at DELETE http://localhost:8080/api/users/:user_id)
        .delete(function(req, res) {
                User.remove({
                        _id: req.params.user_id
                }, function(err, user) {
                        if (err) return res.send(err);

                        res.json({ message: 'Successfully deleted' });
                });
        });
```

Now when we send a request to our API using `DELETE` with the proper `user_id`, we'll delete our user from existence.



Node API Postman Delete

When we try to get all the users you should notice that one is now missing.



Node API Get All Nothing

Conclusion

We now have the means to handle CRUD on a specific resource (our beloved users) through our own API. Using the techniques above will be a good foundation to move into building larger and more robust APIs.

This has been a quick look at creating a Node API using Express 4. There are many more things you can do with your own APIs. You can add authentication, create better error messages, and add different sections so you're not just working with users.

We'll be using this API as the data backend of our Angular application that we will start to build. Let's look at authenticating this API with token based authentication next and then we'll be moving onto learning AngularJS.

Node Authentication

In this chapter, we will implement a good basis of authentication for our application. It is important to note that there are multiple ways of implementing authentication and that will all be based on a case-by-case basis of what type of application you are building, who will be accessing your data, and many other factors.

However, even though there can be many ways to implement authentication, there will be a foundation that is widely used and that is **token based authentication**. Using this type of authentication for our application from the start ensures that it will be protected no matter how large it grows.

The main benefits of tokens are:

- Stateless and scalable servers
- Mobile application ready
- Pass authentication to other applications (think OAuth through Facebook third-party applications)
- Extra security as compared to other authentication methods

Let's talk a bit on authentication and why we should use token based authentication.

Why Token Based Authentication Came to Be?

Token based authentication has been used across the web for many services that we use on a daily basis. When a service provides us with an API to access their data, it is usually authenticated using tokens. Take Facebook's API for example. We need to authenticate with Facebook (the first time with username and password) and then we are given a token for future authentication. Every request after that for a given time will only need the token and not our username and password. We'll talk more on the logistics of this, but keep in mind that many other services including GitHub, Google+, and Twitter (among many more) use tokens for authentication.

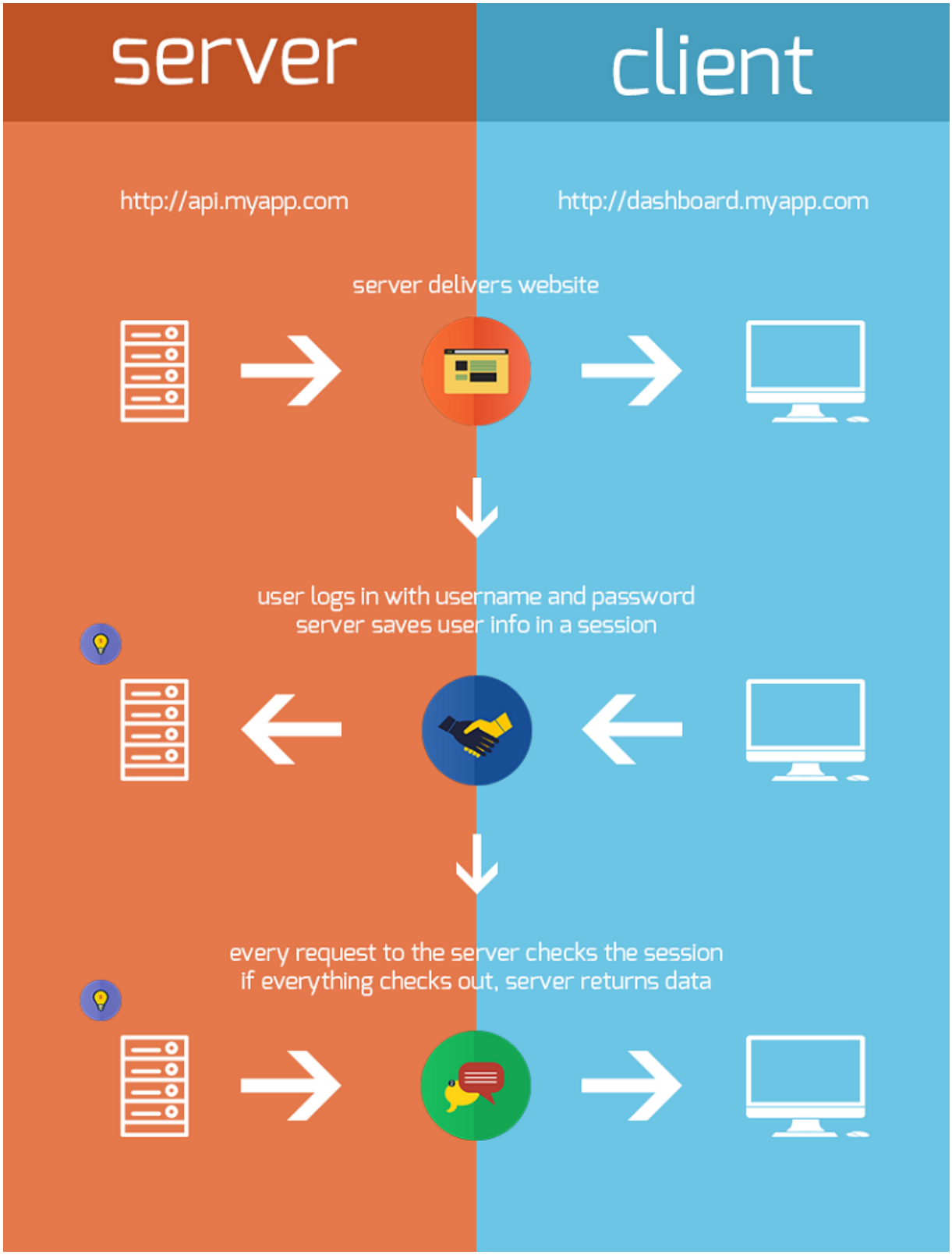
Server Based Authentication (Traditional Method)

Before we can talk about why tokens came about for authentication, let's look at how the norm for authentication used to work.

> Since the HTTP protocol is stateless, if we authenticate a user with a username and password, then on the next request, our application won't know who we are. We would have to authenticate again.

The traditional way of having our applications remember who we are is to store the user logged in information on the server. This can be done in a few different ways on the session, usually in memory or stored on the disk.

Here is a graph of how a server based authentication workflow would look:



Server Based Authentication

As the web, applications, and the rise of the mobile applications have come about, this method of authentication has shown problems, especially in scalability.

The Problems with Server Based Authentication

A few major problems arose with this method of authentication.

Sessions: Every time a user is authenticated, the server will need to create a record somewhere on our server. This is usually done in memory and when there are many users authenticating, the overhead on your server increases.

Scalability: Since sessions are stored in memory, this provides problems with scalability. As our cloud providers start replicating servers to handle application load, having vital information in session memory will limit our ability to scale.

CORS: As we want to expand our application to let our data be used across multiple mobile devices, we have to worry about cross-origin resource sharing (CORS). When using AJAX calls to grab resources from another domain (mobile to our API server), we could run into problems with forbidden requests.

CSRF: We will also have protection against cross-site request forgery (CSRF). Users are susceptible to CSRF attacks since they can already be authenticated with say a banking site and this could be taken advantage of when visiting other sites.

With these problems, scalability being the main one, it made sense to try a different approach.

How Token Based Authentication Works

Token based authentication is stateless. We are not storing any information about our user on the server or in a session.

This concept alone takes care of many of the problems with having to store information on the server.

> No session information means your application can scale and add more machines as necessary without worrying about where a user is logged in.

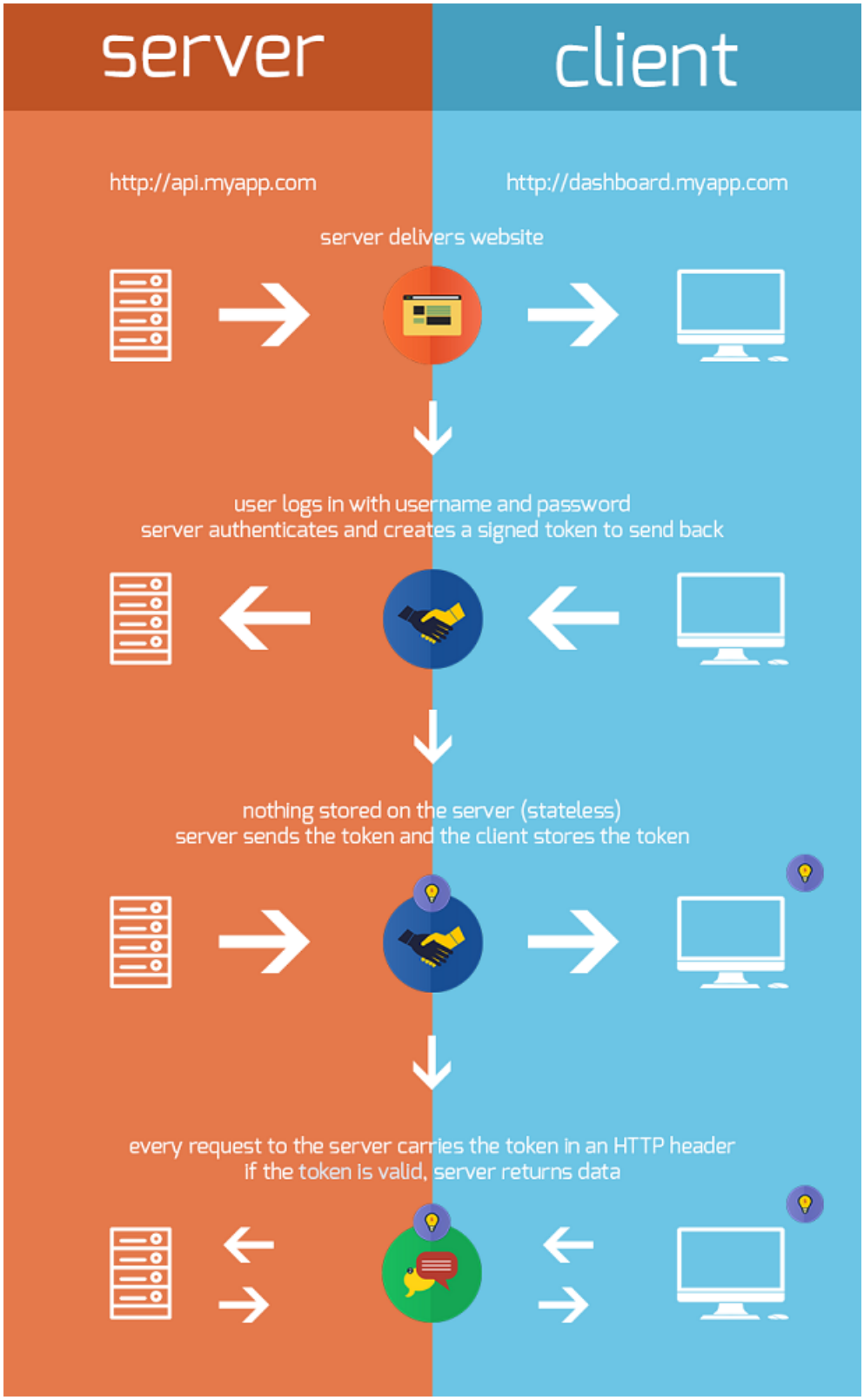
Although this implementation can vary, the gist of it is as follows:

1. User Requests Access with Username / Password
2. Application validates credentials
3. Application provides a signed token to the client
4. Client stores that token and sends it along with every request

5. Server verifies token and responds with data

Every single request after the first will require the token. This token should be sent in the HTTP header so that we keep with the idea of stateless HTTP requests. We will also need to set our server to accept requests from all domains using Access-Control-Allow-Origin: *. What's interesting about designating * in the ACAO header is that it does not allow requests to supply credentials like HTTP authentication, client-side SSL certificates, or cookies.

Here's an infographic to explain the process:



Token Based Authentication

Once we have authenticated with our information and we have our token, we are able to do several things with this token.

We could even create a permission based token and pass this along to a third-party application (say a new mobile app we want to use), and they will be able to have access to our data — **but only the information that we allowed with that specific token**.

The Benefits of Tokens

Stateless and Scalable

Tokens are stored on client side. Completely stateless, and ready to be scaled. Our load balancers are able to pass a user along to any of our servers since there is no state or session information anywhere.

If we were to keep session information for a user that was logged in, this would require us to keep sending that user to the same server that they logged in at (called session affinity).

This brings problems since, some users would be forced to the same server and this could bring about a spot of heavy traffic.

Not to worry though! Those problems are gone with tokens since the token itself holds the data for that user.

Security

The token, not a cookie, is sent on every request and since there is no cookie being sent, this helps to prevent CSRF attacks. Even if your specific implementation stores the token within a cookie on the client side, the cookie is merely a storage mechanism instead of an authentication one. There is no session based information to manipulate since we don't have a session!

The token also expires after a set amount of time, so you will be required to login once again. This helps us stay secure.

Extensibility (Friend of A Friend and Permissions)

Tokens will allow us to build applications that share permissions with another. For example, we have linked random social accounts to our major ones like Facebook or Twitter.

When we login to Twitter through a service (let's say Buffer), we are allowing Buffer to post to our Twitter stream.

By using tokens, this is how we provide selective permissions to third-party applications. We could even build our own API and hand out special permission tokens if our users wanted to give access to their data to another application.

Multiple Platforms and Domains

We talked a bit about CORS earlier. When our application and service expands, we will need to be providing access to all sorts of devices and applications (since our app will most definitely become popular!).

Having our API just serve data, we can also make the design choice to serve assets from a CDN. This eliminates the issues that CORS brings up. Our data and resources are available **as long as a user has a valid token**.

Standards Based

When creating the token, we will be using standards-based JSON Web Tokens[71]. This handy debugger and library chart created by Auth0[72] shows the support for JSON Web Tokens. You can see that it has a great amount of support across a variety of languages. This means you could actually switch out your authentication mechanism if you choose to change backend programming languages in the future!

Now that we've gotten a good foundation of token based authentication, let's get closer to implementing it by examining the method of implementation.

JSON Web Tokens

JSON Web Tokens (JWT), pronounced "jot", are a standard since the information they carry is transmitted via JSON. We can read more about the draft[73] here, but that explanation isn't the most pretty to look at.

JSON Web Tokens work across different programming languages: JWTs work in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can see that these can be used in many different scenarios.

JWTs are self-contained: They will carry all the information necessary within itself. This means that a JWT will be able to transmit basic information about itself, a payload (usually user information), and a signature.

JWTs can be passed around easily: Since JWTs are self-contained, they are perfectly used inside an HTTP header when authenticating an API. You can also pass it through the URL.

What does a JWT look like?

A JWT is easy to identify. It is three strings separated by .

For example:

[71] http://jwt.io/

[72] https://auth0.com/

[73] http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html

```
aaaaaaaaaa.bbbbbbbbbb.cccccccccc
```

Let's break down the 3 parts and see what each contains.

Breaking Down a JSON Web Token

Since there are 3 parts separated by a `.`, each section is created differently. We have the 3 parts which are:

- header
- payload
- signature

JSON Web Token Overview

Header

The header carries 2 parts:

- declaring the type, which is JWT
- the hashing algorithm to use (HMAC SHA256 in this case)

Here's an example:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Now once this is `base64encode`, we have the first part of our JSON web token!

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Payload

The payload will carry the bulk of our JWT, also called the JWT Claims[74]. This is where we will put the information that we want to transmit as well as other information about our token.

There are multiple claims that we can provide. This includes registered claim names, public claim names, and private claim names.

Registered Claims

Claims that are not mandatory whose names are reserved for us. These include:

- `iss`: The issuer of the token
- `sub`: The subject of the token
- `aud`: The audience of the token
- `exp`: This will probably be the registered claim most often used. This will define the expiration in NumericDate value. The expiration MUST be before the current date/time.
- `nbf`: Defines the time before which the JWT MUST NOT be accepted for processing
- `iat`: The time the JWT was issued. Can be used to determine the age of the JWT
- `jti`: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

Public Claims

These are the claims that we create ourselves like user name, information, and other important information.

Private Claims

A producer and consumer may agree to use claim names that are private. These are subject to collision, so use them with caution.

Example Payload

Our example payload has two registered claims (`iss`, and `exp`) and two public claims (`name`, `admin`).

[74] http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html#RegisteredClaimName

```
{
  "iss": "scotch.io",
  "exp": 1300819380,
  "name": "Chris Sevilleja",
  "admin": true
}
```

This will encode to:

```
eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEi\
LCJhZG1pbiI6dHJ1ZX0
```

That will be the second part of our JSON Web Token.

Signature

The third and final part of our JSON Web Token is going to be the signature. This signature is made up of a hash of the following components:

- the header
- the payload
- secret
- The token is sent on every request so there are no CSRF attacks. There is no session based information to manipulate since, well, we don't have a session!

This is how we get the third part of the JWT:

```
  var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);

  HMACSHA256(encodedString, 'secret');
```

The secret is the signature held by the server. This is the way that our server will be able to verify existing tokens and sign new ones. This is the only thing that our server holds in order to verify the user.

This gives us the final part of our JWT.

```
03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773
```

Now we have our full JSON Web Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTk\
zODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0.03f329983b86f7d9a9f5fef\
85305880101d5e302afafa20154d094b229f75773
```

Auth0[75] has created a great site[76] to go through and test out how JWTs are made. You can see as you change the content on the fly, you are able to see the JWT get updated immediately. Auth0 provides great tools and they also maintain the jsonwebtoken[77] Node package to handle creating and verifying JWTs in Node.

With an understanding of both token based authentication and the mechanism to handle authentication, JSON Web Tokens, let's see how we can implement both of these into our Node API that we just built.

Authenticating Our Node.js API

Here is a quick overview of what we will want for our API. Keep in mind that this sort of layout could be used for many types of applications. We can have websites where the unauthenticated routes are the front facing routes of the site and the authenticated are the backend admin sections.

For this next part, we will want:

- A basic route (home page), which will be (unauthenticated)
- Only API routes are authenticated
- Route used to authenticate a user (login)
- Pass in the token to have working auth

Setting Up

We'll be using the application we created in Chapter 9. It has everything we need since it is an API and we want to use tokens to implement authentication into an API.

The main things we will want to do are:

- Set up an authentication route to check a user and make sure they have the correct password
- Give a user a token if they authenticate with the right username and password
- Authenticate the API routes (but not the basic home page route)

Let's get started with the authentication. Get the app you created from Chapter 9. We will need to install the Node jsonwebtoken[78] package so that we will be able to create and verify tokens.

[75] https://auth0.com/
[76] http://jwt.io/
[77] https://github.com/auth0/node-jsonwebtoken
[78] https://github.com/auth0/node-jsonwebtoken

```
npm install jsonwebtoken --save
```

Since we already have a `package.json` file from Chapter 9, this command will keep the packages we already have and add `jsonwebtoken` to the list of dependencies.

Grab the JSON Web Token Package

Let's go ahead and grab the `jsonwebtoken` package in our `server.js` file like so:

```
var jwt = require('jsonwebtoken');
```

Create a Secret to Create Tokens With

As we learned, part of our JWT is made using a secret. We will create a variable so that we can use this string as the secret.

```
var superSecret = 'ilovescotchscotchyscotchscotch';
```

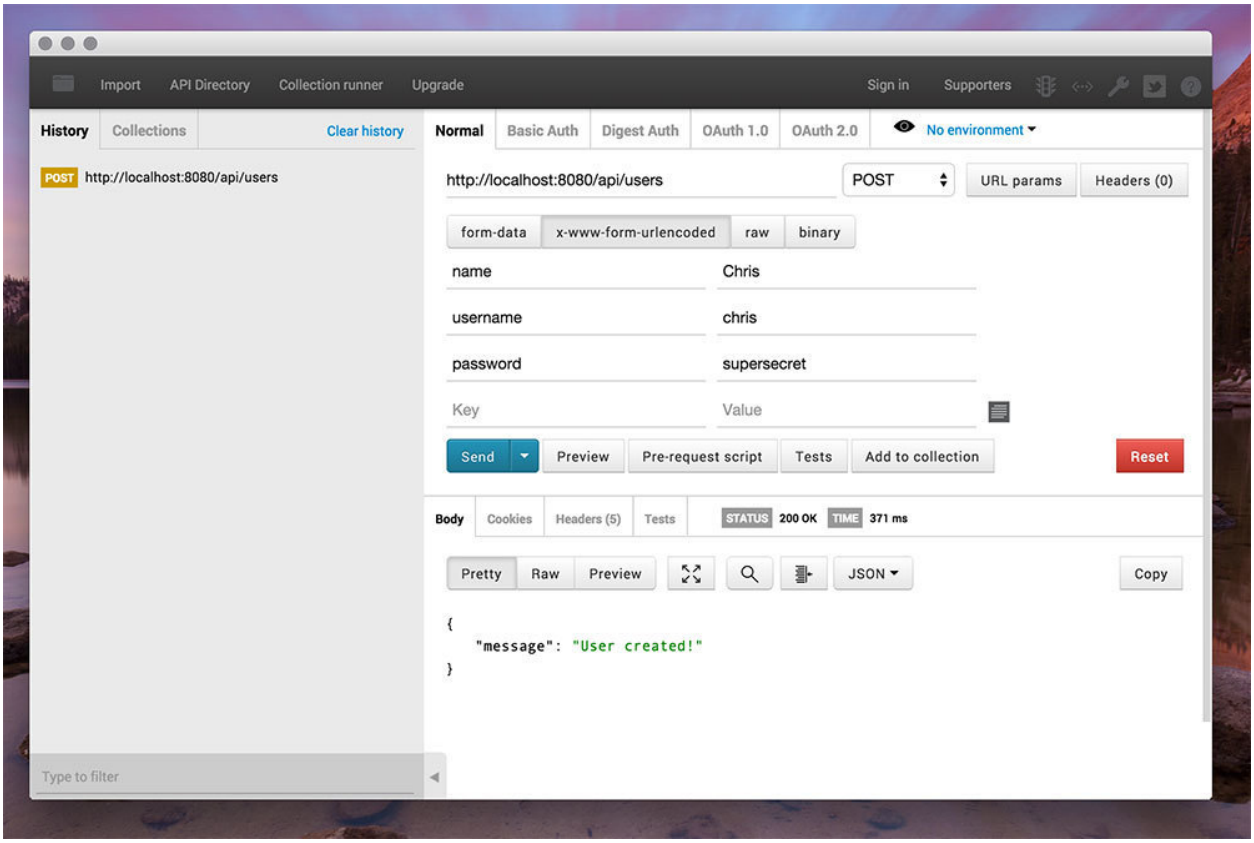
With the package that we need and secret ready to go, let's move onto authenticating a user.

Create a Sample User

First, we need to make sure that we even have a user to authenticate since towards the end of last chapter, we deleted everyone. Let's create the user using the `POST http://localhost:8080/api/users` route we created in our API to add a user to our database.

We will send a POST request with the following information:

Name Chris **Username** chris **Password** supersecret



Create Sample User

Since there is no authentication yet, the user will be created just fine. Now that we have our user, let's create the authentication mechanism for him to sign in and get a token.

Authenticating A User and Giving a Token

We will create a new route inside of our API routes called `POST http://localhost/api/authenticate`. This is where a user will send POST request with username and password. If both of those check out, they will receive a token so that they can have access the information from our API.

The order of how we create our routes is important here. We don't want this `authenticate` route to be protected by our authentication middleware, so it will **be placed before the middleware** that we created in Chapter 9. Here is a look at its placement:

```
// basic route for the home page
app.get('/', ...

// get an instance of the express router
var apiRouter = express.Router();

// route for authenticating users
apiRouter.post('/authenticate', ...

// middleware to use for all requests
apiRouter.use(function(req, res, next) ...

// other api routes. the authenticated routes
apiRouter.get('/', ...
```

Inside of the `apiRouter.post('/authenticate', ...` route is where we will be creating the JSON Web Token and returning it to our user.

Let's create the route to authenticate our user. We will:

- Check to make sure a user with that username exists
- Check to make sure that the user has the correct password (by comparing their password to the hashed one saved in the database)
- Create a token if all is well

```
// route to authenticate a user (POST http://localhost:8080/api/authenticate)
apiRouter.post('/authenticate', function(req, res) {

  // find the user
  // select the name username and password explicitly
  User.findOne({
    username: req.body.username
  }).select('name username password').exec(function(err, user) {

    if (err) throw err;

    // no user with that username was found
    if (!user) {
      res.json({
        success: false,
        message: 'Authentication failed. User not found.'
```

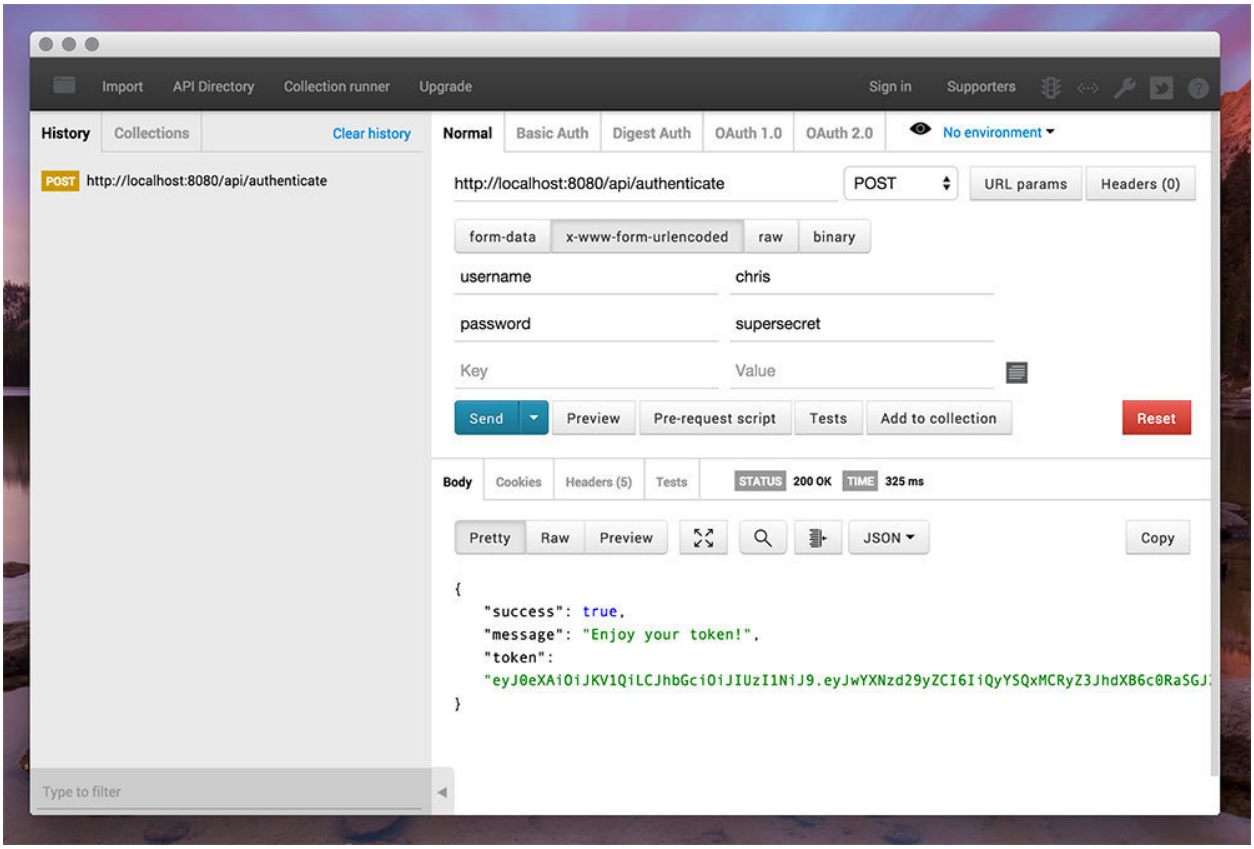
```
17        });
18      } else if (user) {
19
20        // check if password matches
21        var validPassword = user.comparePassword(req.body.password);
22        if (!validPassword) {
23          res.json({
24            success: false,
25            message: 'Authentication failed. Wrong password.'
26          });
27        } else {
28
29          // if user is found and password is right
30          // create a token
31          var token = jwt.sign({
32            name: user.name,
33            username: user.username
34          }, superSecret, {
35            expiresInMinutes: 1440 // expires in 24 hours
36          });
37
38          // return the information including token as JSON
39          res.json({
40            success: true,
41            message: 'Enjoy your token!',
42            token: token
43          });
44        }
45
46      }
47
48    });
49  });
```

This route does the main things that we need it to do: **check if the user with that username exists**, **make sure that the password is correct** using the `comparePassword` method we created in Chapter 9 on our User model, and **create a token**.

We are using the `jwt` (jsonwebtoken) package to sign the token. This package will automatically generate the header and the signature of our JWT when we pass in the payload, which in this case is our user.

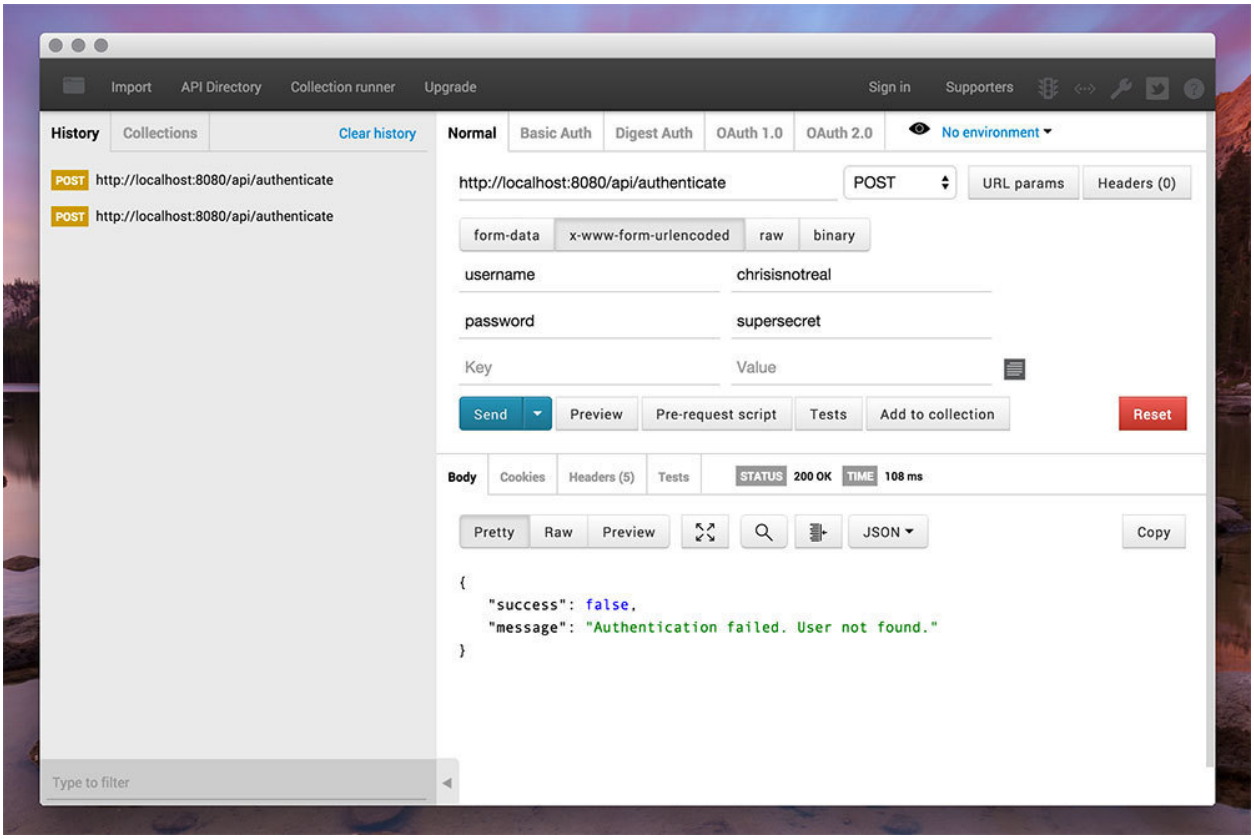
Now, when we try to authenticate a user using the credentials we used earlier with a POST request to `http://localhost:8080/api/authenticate` along with the information we used to create a user

(chris/supersecret), we will be given the token back!



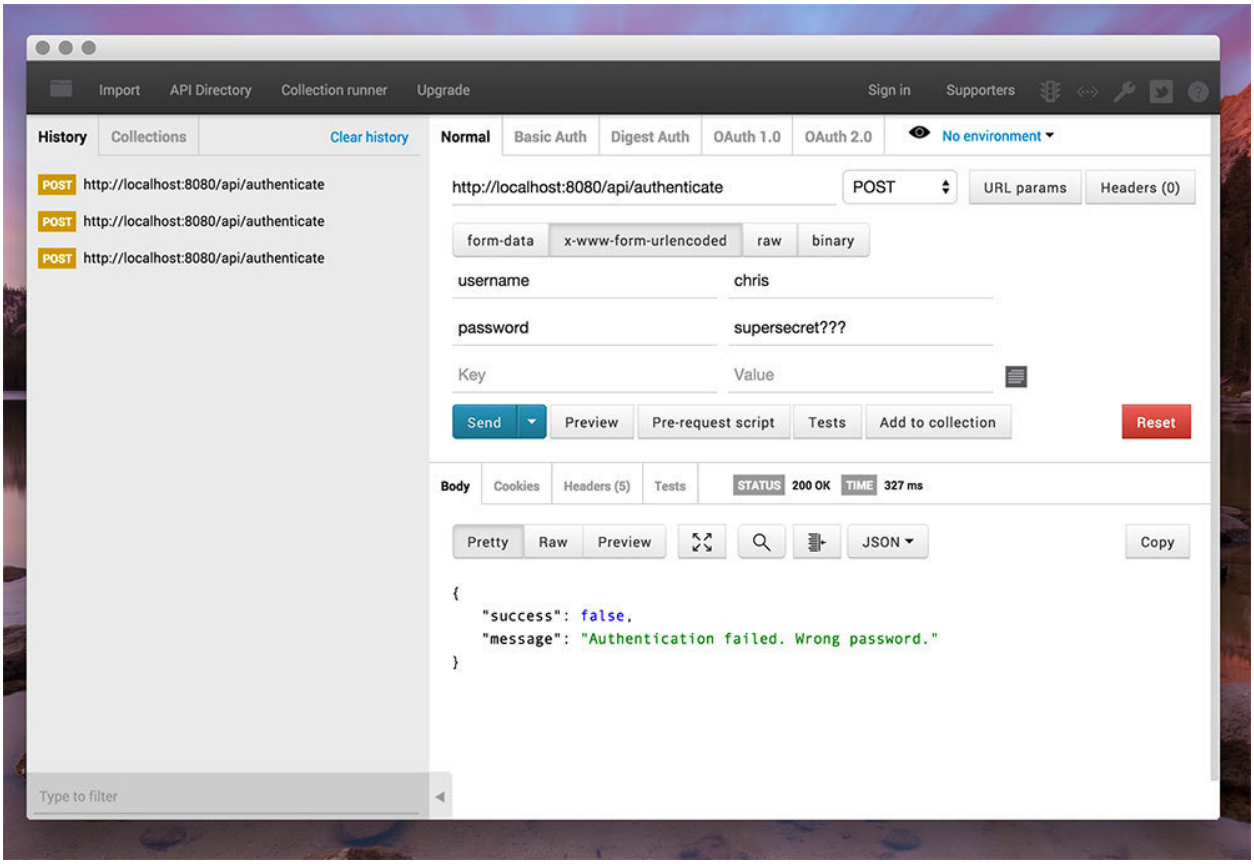
Authenticate User and Get a Token

We can also test our authentication route with the wrong username:



Authenticate Wrong Username

And here it is with the wrong password:



Authenticate Wrong Password

Now that we have given our user the token, it will be up to them to hold the token on the client side (probably in a cookie). They will then send that token to us on every request where they would like to get information. Let's look at how we can check that token on every request to make sure that it is a valid token.

Route Middleware to Protect API Routes

We will use the middleware that we already put in place (`apiRoutes.use(function())`) to check the token on every request for our authenticated routes.

We will create a flexible API by allowing a user to pass the token via POST parameters, the URL parameters, or as an HTTP header. We will verify the token and if the token is good, we will pass the user along to their original destination and give them the information they requested.

Here is that middleware for verifying the token:

```
// route middleware to verify a token
apiRouter.use(function(req, res, next) {

  // check header or url parameters or post parameters for token
  var token = req.body.token || req.query.token || req.headers['x-access-token'];

  // decode token
  if (token) {

    // verifies secret and checks exp
    jwt.verify(token, superSecret, function(err, decoded) {
      if (err) {
        return res.status(403).send({
            success: false,
          message: 'Failed to authenticate token.'
        });
      } else {
        // if everything is good, save to request for use in other routes
        req.decoded = decoded;

        next();

      }
    });

  } else {

    // if there is no token
    // return an HTTP response of 403 (access forbidden) and an error message
    return res.status(403).send({
      success: false,
      message: 'No token provided.'
    });

  }

  // next() used to be here
});
```

We are using the `jsonwebtoken` package again, but this time we are going to verify the token that was passed in. It is important that our secret used here matches the secret that was used to create the token.

If everything looks good and the token was able to be verified, we'll take the information that came out of the token and pass it to the other routes in the `req` object.

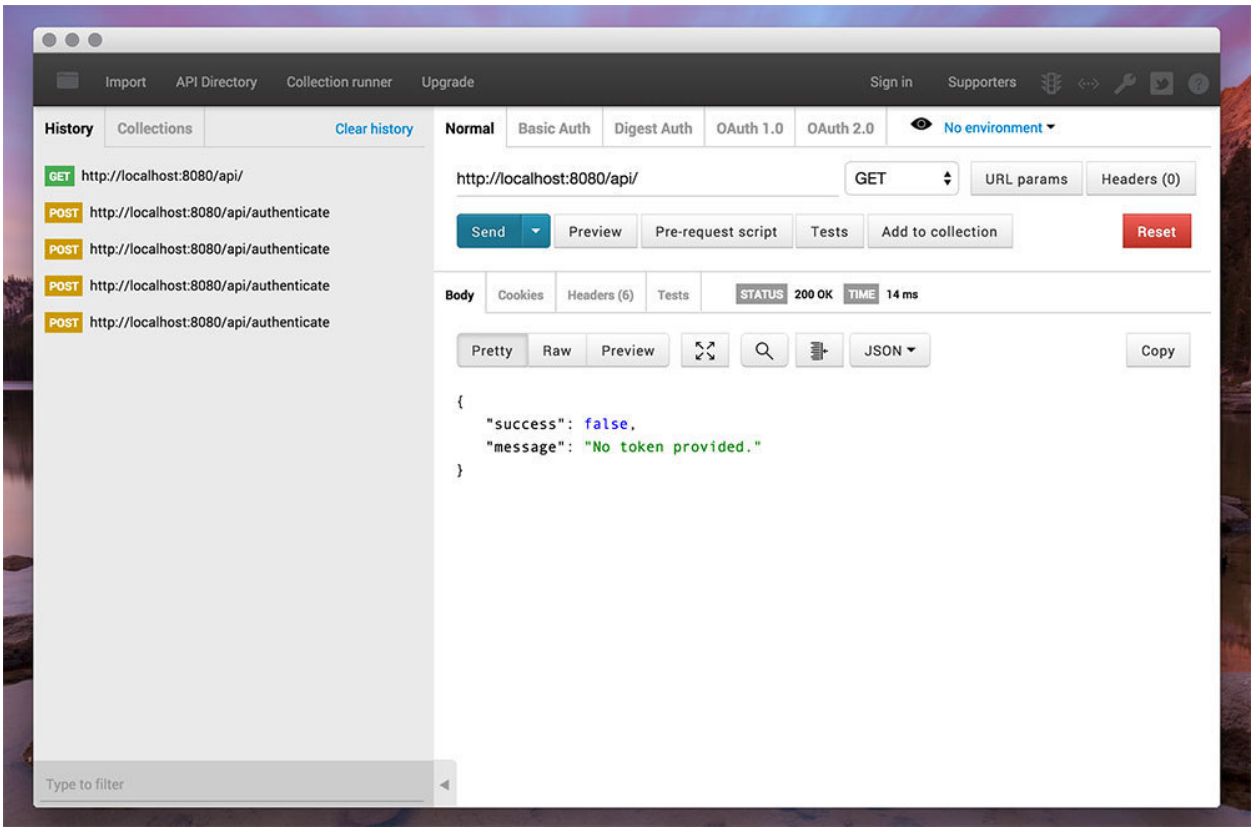
If there is no token, we will want to send an HTTP status of 403 which means access forbidden. We will also send the message that the token was not provided.

We have moved the `next()` to be in the `if/else` statement so that our user will only continue forward if they have a token and it verified correctly.

Testing Our Route Middleware

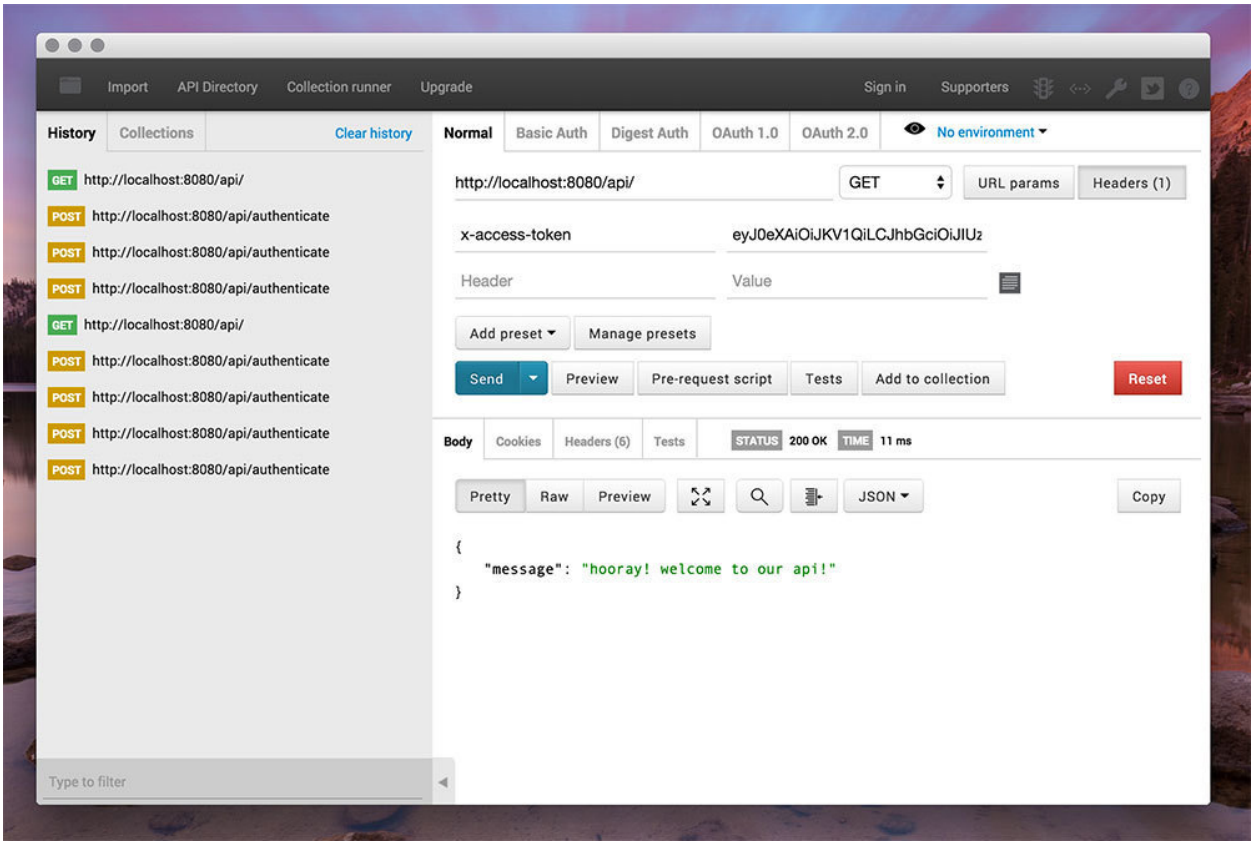
Now that we have built out the middleware that is responsible for protecting all the routes that follow it, let's test what we've just made to make sure that it works properly.

Here is an attempt at trying to reach the main API route without passing in any authentication:



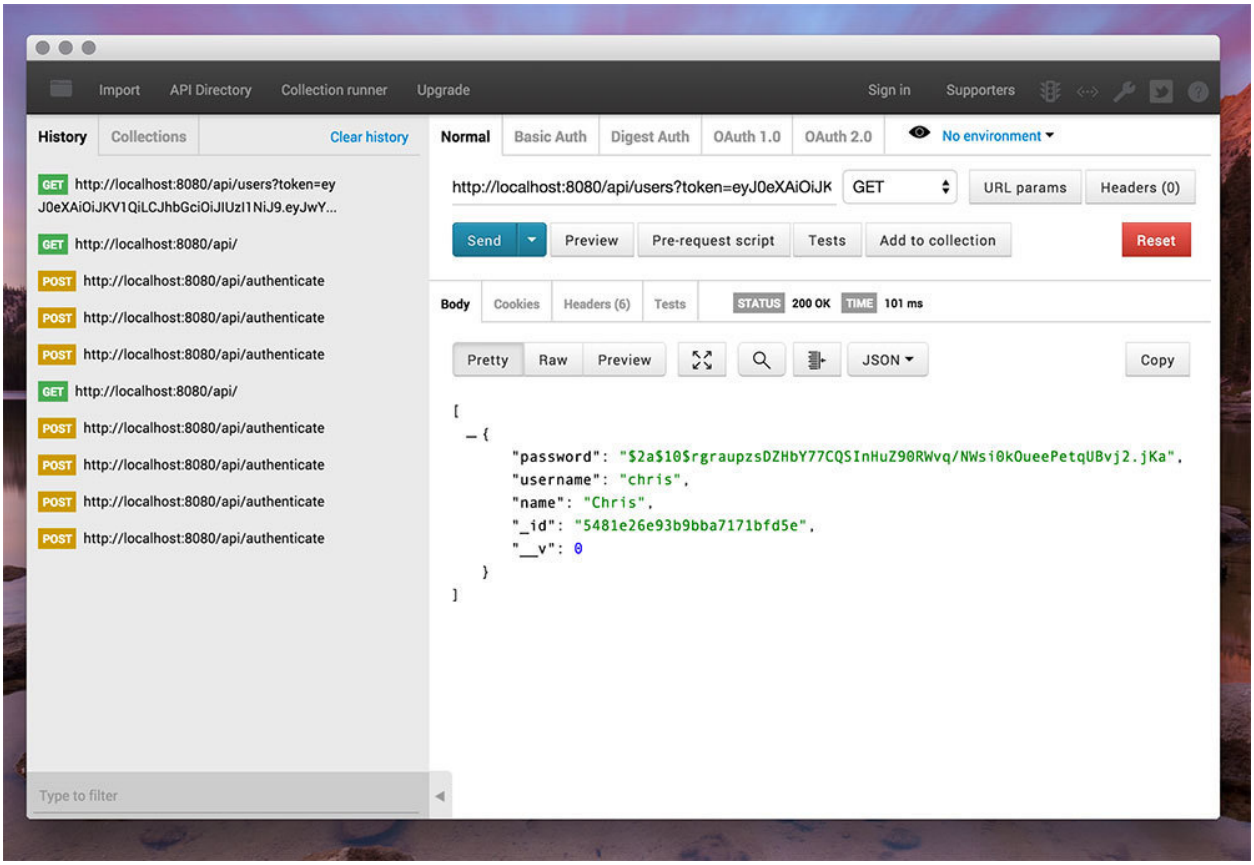
No Token Provided

Now let's pass in a token through the HTTP Headers as `x-access-token`:



Token Passed Through HTTP Header

We can also pass the token in as a URL parameter like so: http://localhost:8080/api/users?token=eyJ0eXAiOiJKV1QiL
b_NS6io8U4srLGuvEAfRhJg



Token Passed Through URL Parameter

Route to Get User Information

Let's quickly add one more route to our API so that we can return a user's information.

We will call this endpoint (`GET http://localhost:8080/api/me`) and this will allow us to get information about the logged in user.

After all our other `apiRouter` routes, let's define the following:

```
1  // api endpoint to get user information
2  apiRouter.get('/me', function(req, res) {
3    res.send(req.decoded);
4  });
```

Remember in our middleware, we stored the logged in user in `req.decoded` for use in the other routes. Now we'll just grab that information and return it. We'll see how this is useful in the future Angular chapters when we want to display a logged in user's name in a message (something like **Hello Chris!**).

Modules to Help with Authentication

Now that we have built all of this from scratch ourselves, it's important to note that there are Node modules out there to help handle authentication for us. It is good to understand the basics by building security ourselves, but it also can help to have modules that are built by teams of people and make sure that security is solid.

For handling authentication in Node, you won't find a better package than PassportJS[79]. This allows us to integrate authentication on our server with session based security, social based authentication, and what we've done in this chapter, JWT security.

There is even a package to handle checking the JWT and protecting routes called express-jwt[80]. This package creates a middleware for us so that we don't have to. It will allow us to set protected and unprotected routes.

Conclusion

Now we have an API with token authentication! Just like the big companies like Facebook, Google, Twitter, and GitHub! Well not exactly like them since they implement more features like user specific permissions, group permissions, and many more advanced features, but this is the foundation of it.

Remember that security is a very important part of our applications and protecting our users' information should be top priority. Keep in mind that this is only just the beginning of security and that security is always evolving to guard from new and old attacks.

This API will be used to hand data to our frontend AngularJS application. By doing so, we are once again adhering to the notion that our backend and frontend are both separate, yet work together to create amazing experiences.

[79] http://passportjs.org/

[80] https://github.com/auth0/express-jwt

Starting Angular

Let's take a step back for a second. So far, we've worked our way through building an awesome backend, server-side service. We've learned how to create a RESTful API, how to use MongoDB to handle CRUD operations, and become fluent in Node.

Now we have to handle the other side of the coin: **the frontend**. We have to become familiar with Angular applications, how they are set up, and how we will access our Node API and data that we've built.

The State of JavaScript Applications

If you've built JavaScript applications before without a framework (let's use jQuery as an example), a lot of times you are grabbing and placing data within your application using `$('#divname').html()` or `$('#divname').append(variable)`. This causes problems since there isn't a true source of data. You're busy grabbing and placing numbers and elements here and there and sometimes you aren't fully sure which is the latest and most valid piece of data.

In MVC applications, the true source of data comes from the model which is facilitated to the view by the controller. This means there is a solid workflow for grabbing data. The view is built to just display data and be as logic free as possible. The model and controller are really where data manipulation need to take place.

Angular helps solve these problems by treating front-end applications more like the back-end applications we're used to. We like having data coming from a consistent place and just being injected into views. This keeps us sane since every part of our application has a true (singular) source of data.

Introduction

Angular calls itself "what HTML would have been, had it been designed for building web-apps". It allows you as the developer to build out an MVC (model-view-controller) architecture for the frontend of your application. Since client-side applications are becoming more than just showing a static HTML page, new tools needed to be built to accommodate the growing JavaScript applications.

Angular allows you to extend HTML so you are building on standards and what you already know, not learning a whole new framework and starting your skills from scratch.

To beginners of Angular, building out their first application can truly **seem like magic**. That's because **it is**... Okay not really, but it is an amazing tool for building incredible applications. Here are two big reasons why it can seem like magic.

Important Angular Concepts

Some awesome concepts about Angular that will carry over to all of the things we build are **data-binding**, **dependency injection**, and **directives**.

Data-Binding

Data-binding[81] allows us to have a centralized source of data. We'll be talking a lot on the **single true source of data**. Forget about injecting data into your views using the jQuery ways of `append`, `val`, `html`, or whatever else jQuery provides. We will use Angular's amazing data-binding to handle all of that for us.

Let's just imagine a scenario where you would create a variable in a controller. Then you would display that variable in your view. If you change the variable in your controller, it would automatically (automagically?) change in your view without you needing to inject it there (Angular does the injection for you by binding the variables in both places).

The Angular docs provide a good explanation of this feature: "The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model."

We'll demo this in the starter Angular application for this chapter.

Dependency Injection

An Angular application is a collection of a lot of different modules like a controller to handle passing data to our views, a service (seen as a model) that interacts with an API, a directive that will help us manipulate our views further, and so much more. When building our applications, we will take the modular approach since that let's us build testable, modular, and scalable applications.

Each module we build will be **injected** into our main application module to create the final product. Our main application will inherit the dependencies of the modules we inject into it and this is one of those times where you can use the saying: "the sum of its parts is greater than the whole".

Directives

Directives[82] are built into Angular and you'll be dealing with them a ton. They are how we extend our HTML templates to tell Angular where and when to apply features to our views. Directives will let Angular know that certain HTML elements should be data-bound to Angular variables or functions we define. They can also help Angular manipulate our DOM.

[81] https://docs.angularjs.org/guide/databinding
[82] https://docs.angularjs.org/guide/directive

> jQuery teaches us to grab and inject into the DOM. Angular teaches us to manipulate data, which in turn, changes the DOM.

Let's get to building out the application and talk more about concepts as we go along.

Setting Up An Angular Application

This demo won't have any dynamic data from our Node API just yet. We'll be hard-coding our data right into our Angular controllers. This will let us get a feel for Angular specific features first.

Here is the file structure for our Angular application. Notice that we won't need any `package.json` or `server.js` file like our Node applications. This is a purely front-end application with just HTML/CSS/JS!

- js/
  - app.js
- index.html

Super simple stuff! 2 files will get us everything we need for our first Angular application.

The things we are looking to accomplish with this application are:

1. setup an Angular frontend application
2. display data (message and a list) from our Angular controller to the view
3. show off data-binding by updating the message using an input box

Creating Your Angular App (app.js)

We will start by creating the main module for our application. Let's go into that `app.js` file and see how Angular modules are made:

```
1  // name our angular app
2  angular.module('firstApp', [])
3
4  .controller('mainController', function() {
5
6      // bind this to vm (view-model)
7      var vm = this;
8
9      // define variables and objects on this
10     // this lets them be available to our views
```

```
11
12      // define a basic variable
13      vm.message = 'Hey there! Come and see how good I look!';
14
15      // define a list of items
16      vm.computers = [
17          { name: 'Macbook Pro', color: 'Silver', nerdness: 7 },
18          { name: 'Yoga 2 Pro', color: 'Gray', nerdness: 6 },
19          { name: 'Chromebook', color: 'Black', nerdness: 5 }
20      ];
21
22  });
```

You can see that the only thing necessary for an Angular module is to declare `angular.module`. We can then add a controller onto this module. When building out these applications, we'll try to keep one feature on each application. This means we won't be mixing controllers and services onto the same module. This goes back to the idea of having a compartmentalized application.

So far we have:

- created our main module called **firstApp**
- created a controller called **mainController**
- information in this controller will be bound to itself using `vm`
- created a variable called **message**
- created a list of items called **computers**

Notice how we defined the variable and our list on the `vm` variable. It is a good practice to bind the parent `this` in the controller to `vm`. This helps when referencing the controller since the word `this` can be used within JavaScript callbacks like `success()`.

Controllers have properties defined on them and this is how we will be able to use them in our views with the **controller as** syntax. We'll soon see exactly how this works.

That's it! You just made your first Angular app! Now it doesn't do much so let's give it a pretty look by making our HTML file and applying this module to it.

Applying the Angular App (index.html)

Let's start our basic HTML site. We'll bring in Bootstrap and Angular via a CDN to keep things simple. Here's the very beginnings of our `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My First Angular Application!</title>

    <!-- CSS -->
    <!-- load bootstrap and our stylesheet -->
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/\
css/bootstrap.min.css">
    <style>
        body { padding-top:50px; }
    </style>

    <!-- JS -->
    <!-- load angular and our custom application -->
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular.mi\
n.js"></script>
    <script src="js/app.js"></script>
</head>
<body class="container">

</body>
</html>
```

This `index.html` file is barebones right now. We have loaded Bootstrap, adding a little CSS for spacing, and loaded Angular itself and our custom `app.js`. Nothing will happen if we view this in our browser yet since we haven't applied our Angular app that we made to this view.

We talked about **Angular directives** earlier and this will be your first use of them. Angular provides directives to apply your application to your HTML. These are called `ng-app` and `ng-controller`. Let's add the following to our `<body>` tag:

```
<!-- declare our angular application and angular controller -->
<body class="container" ng-app="firstApp" ng-controller="mainController as main">
```

We are using the **controller as** syntax and naming this controller `as main`. We can now call variables and functions from our controller by prefixing with the word `main`.

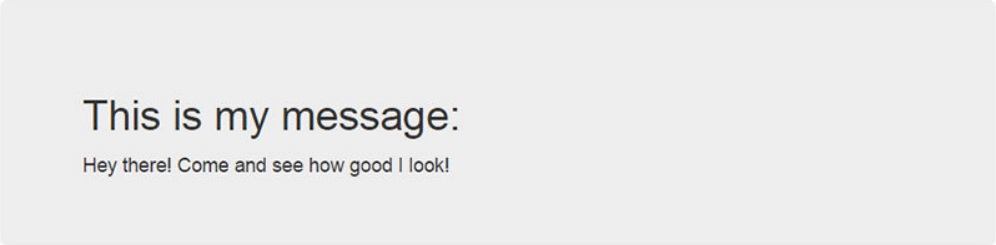
Now we have our application applied to the body of our HTML. You can also place controllers to specific sections of your site so that you have a controller for let's say a sidebar and a controller for the main content. This even further compartmentalizes your site, but for this example, we'll just apply one controller to everything.

Let's do the next easiest step and display the message variable we created earlier.

Add the following inside of your `<body>` tag:

```
<div class="jumbotron">

  <!-- display the message -->
  <h2>This is my message:</h2>
  {{ main.message }}

</div>
```

Now we can see the message in our browser! Notice how we defined this variable on the `main` keyword. Angular uses the curly brackets (`{{ }}`) to display items to the view and will look for variables and functions that were defined in our controller.



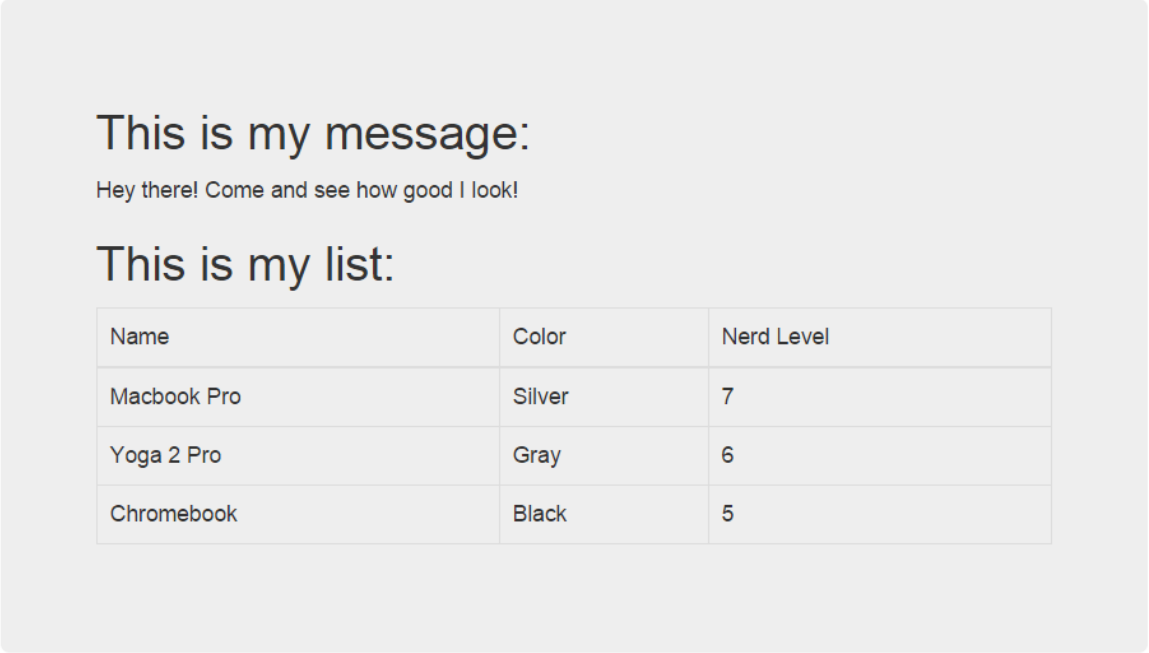
Show Message Variable

You just displayed your first variable to your view! Now let's handle the list of computers. This will use another Angular directive called **ng-repeat**. You'll be using this a lot in your views since it acts as a foreach for showing off data.

Add the following below the message section:

```
<!-- display the list using ng-repeat -->
<h2>This is my list:</h2>
<table class="table table-bordered">
    <thead>
        <tr>
            <td>Name</td>
            <td>Color</td>
            <td>Nerd Level</td>
        </tr>
    </thead>
    <tbody>
        <tr ng-repeat="computer in main.computers">
            <td>{{ computer.name }}</td>
            <td>{{ computer.color }}</td>
            <td>{{ computer.nerdness }}</td>
        </tr>
    </tbody>
</table>
```

Pretty neat right? Angular will know to loop over the computers and display each in its own table row.



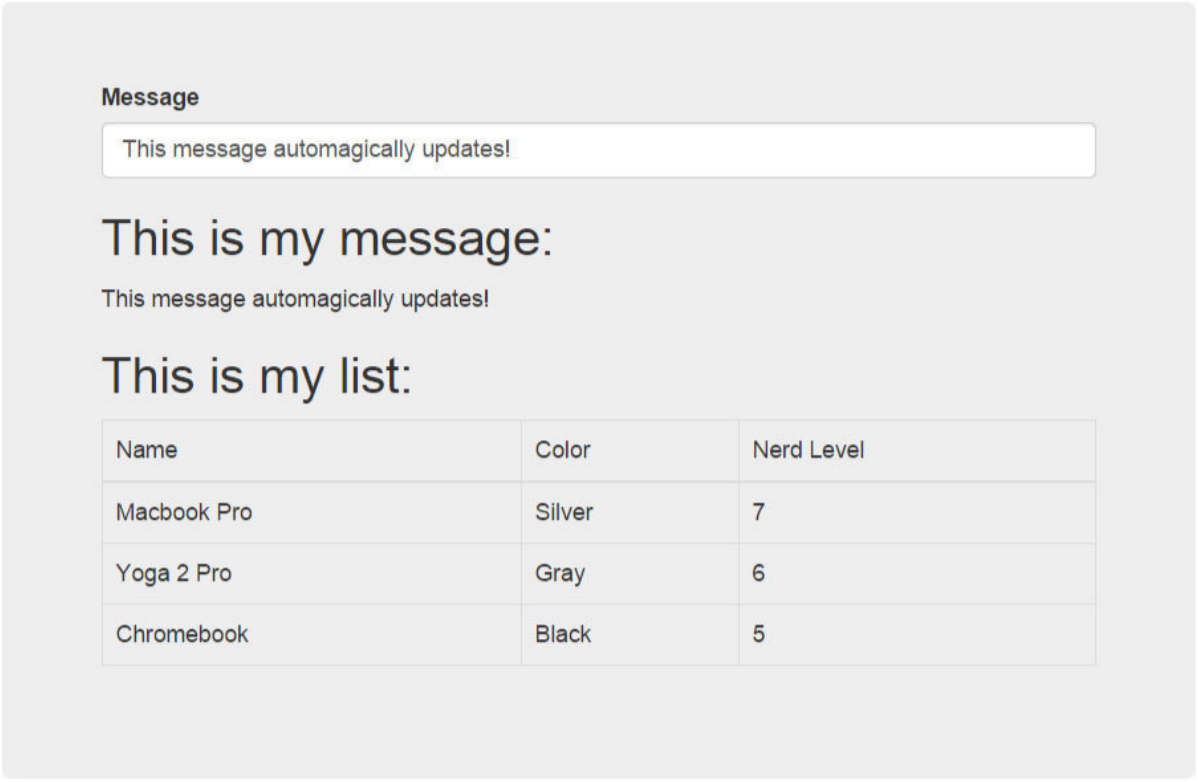
Show List

We'll do one more thing for this demo. This is the magic part where we get to see **data-binding** in action.

We'll place an input field on this site and use yet another Angular directive (see the trend?) called **ng-model** to bind an input to a variable. This will immediately change the message. You'll see the change happen right before your eyes! Let's add the input box above the message section:

```
<!-- form to update the message variable using ng-model -->
<div class="form-group">
    <label>Message</label>
    <input type="text" class="form-control" ng-model="main.message">
</div>
```

Now view the site in your browser and start typing into the input box. The message variable immediately changes thanks to the magic of Angular and its **directives**.



Show Data Binding

Here is a CodePen[83] so that you can see all of this in action quickly. The full code is also available in the code repository for this book.

Tip

$scope vs. this

In other Angular tutorials around the web, you may have seen `$scope` as a way to bind information from the controller to the view. As of Angular 1.3, this method still exists, but it is encouraged to use **controller as** syntax.

[83] http://codepen.io/sevilayha/full/DhLqk/

There are many benefits to this approach since it keeps our code cleaner and more organized. It is even easier to read in our views how our controllers are used.

Here are two quick comparisons of *using $scope* and *not using $scope*.

```
<!-- with $scope -->
<div ng-app="myApp" ng-controller="mainController">
    <p>{{ myVariable }}</p>
</div>

<!-- without $scope - controller as -->
<div ng-app="myApp" ng-controller="mainController as main">
    <p>{{ main.myVariable }}</p>
</div>
```

This has many benefits, especially when we start nesting controllers. It will be easier to see which variables and functions live within each controller.

This also has the benefit that it will force you to prepare for the new Angular 2.0 syntax, which is expected in about a year. The Angular team has already declared that `$scope` will be killed off in favor of a syntax closer to `controller as`.

Don't worry about Angular 2.0 though. If you build your Angular applications now using this syntax, migrating to the new version won't be too painful.

Creating and Processing a Form

Now let's move onto something more advanced. We will often want a form on our site and Angular makes processing forms incredibly easy. Let's create a form that will add to our list of computers.

This will be done in two different parts:

1. Create an Angular function to handle processing the form and adding to our list
2. Add the HTML form and wire it up to work with the Angular function

Angular Function to Process a Form

Let's add this Angular function to our controller.

```
// name our angular app
angular.module('firstApp', [])

.controller('mainController', function() {

    ...

    // information that comes from our form
    vm.computerData = {};

    vm.addComputer = function() {

        // add a computer to the list
        vm.computers.push({
            name: vm.computerData.name,
            color: vm.computerData.color,
            nerdness: vm.computerData.nerdness
        });

        // after our computer has been added, clear the form
        vm.computerData = {};
    };

});
```

We will create an object to hold the data of our form called `vm.computerData`. This step isn't necessary since Angular will automatically create it when it is needed, but we'll define it for clarity.

We are also creating a function that will take the information out of `vm.computerData` and push it into our array of computers. When this is added, it should automatically add the computer to our table without the need to refresh the page (almost like magic).

After the computer has been added to the list, we will clear the form by clearing the object that contains the data. In jQuery applications, clearing a form could be accomplished with:

```
$('input').val('');
```

Not in Angular! We know all about **data-binding** now and since we clear the object on the Angular side of things, which will automatically clear the form on the view side.

With that out of the way, let's create the HTML form and wire it to the function we just created.

The HTML Form

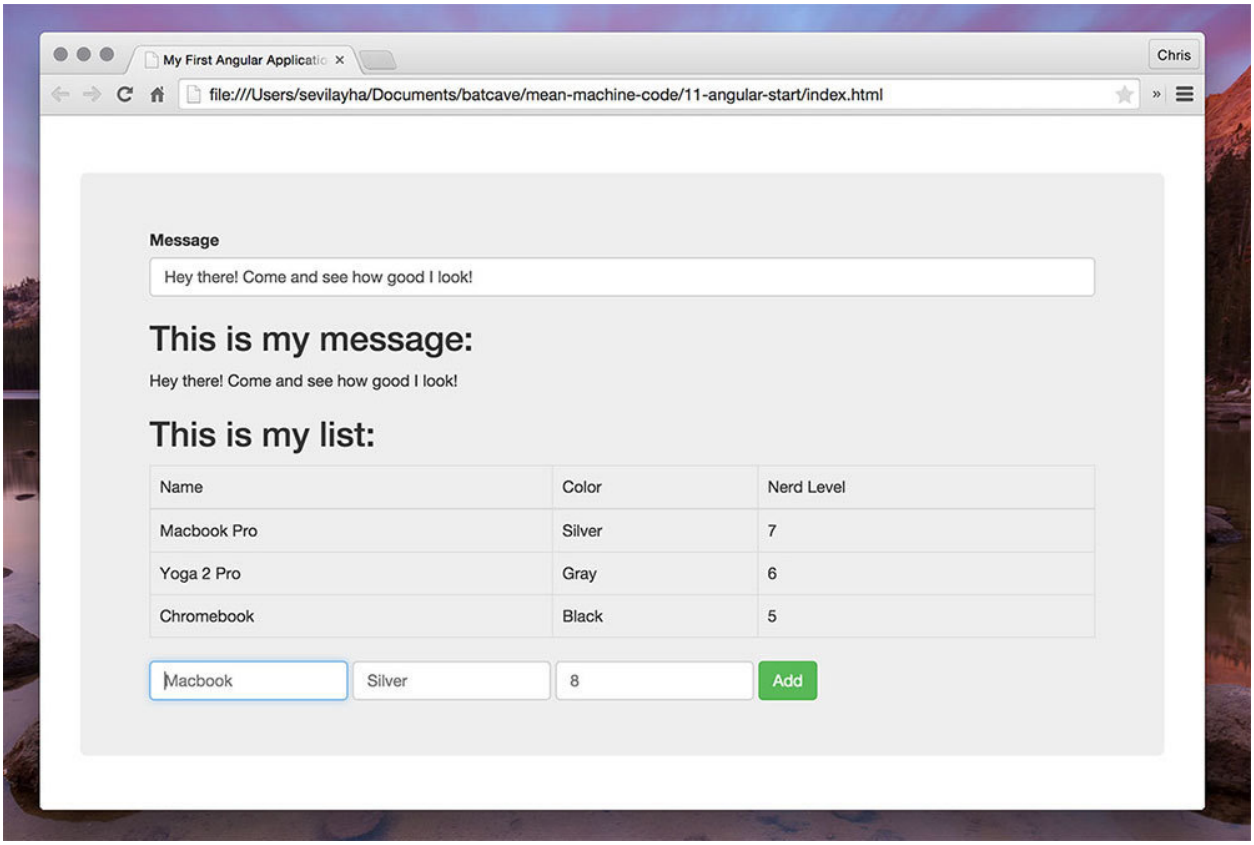
In our `index.html` file, under the HTML table of computers, add the following form:

```
<!-- form to add computer to the list -->
<form class="form-inline" ng-submit="main.addComputer()">

    <input type="text" class="form-control" placeholder="Macbook" ng-model="main\
.computerData.name">
    <input type="text" class="form-control" placeholder="Silver" ng-model="main.\
computerData.color">
    <input type="number" class="form-control" placeholder="8" ng-model="main.com\
puterData.nerdness">

    <button type="submit" class="btn btn-success">Add</button>
</form>
```

We have 3 different inputs here and one submit button. We have used `ng-model` to bind each input to a specific field in the `computerData` object we created in our Angular controller.

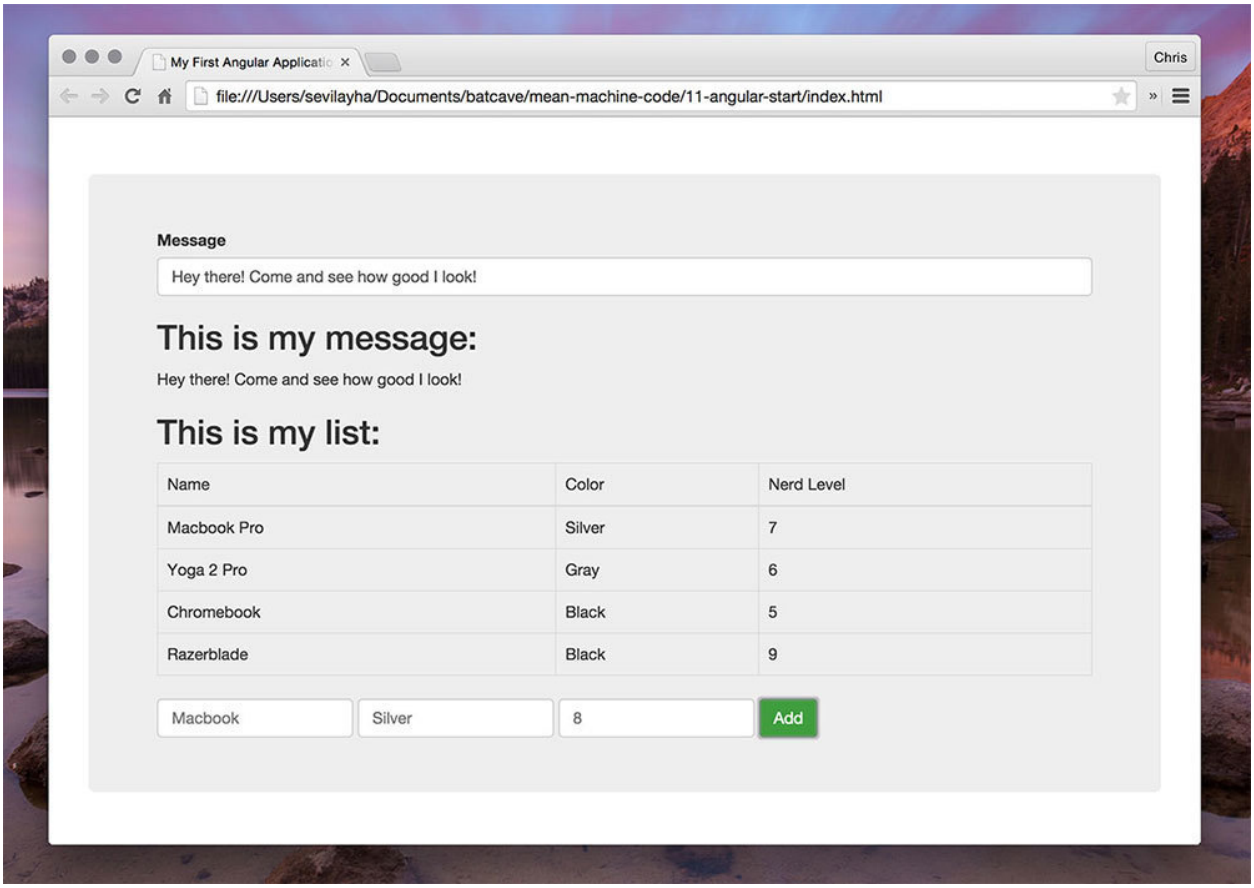
The new Angular directive we see here is `ng-submit`. This is used on forms and tells Angular what to do when the form is submitted. We are telling Angular to submit the form and use the function called `addComputer()` that we created in our controller.

Here is our form:



HTML Form

Go ahead and fill in some data and click **Add**. Watch as your table is automatically updated and the new computer is added immediately without a page refresh!



Computer Created

Conclusion

This chapter introduced you to quite a few Angular concepts all at once. They are all tools to help you build out amazing applications and we'll be expanding on these even further in more advanced implementations. Data-binding, dependency injection, and directives are the foundation of Angular applications and you'll use them all in every app you build.

Next up, we'll explore routing our Angular applications so that we have more fully featured sites/applications with multiple pages.

Routing Angular Applications

We'll demonstrate Angular routing with a brand new Angular app/site. It will be good to create a few apps from scratch so that the process becomes familiar.

Before we start that new application, however, let's talk about the different routing techniques Angular has. There are two main modules for providing routing features: ngRoute[84] and AngularUI

[84] https://docs.angularjs.org/api/ngRoute

Router[85].

ngRoute

ngRoute[86] is the module that is the standard when building out routing in Angular applications. It is supported by Angular and provides routing through a service called `$route` (we'll show off how this works later in this chapter).

This routing module allows us to create **single page applications and websites** that feel like the rich websites and mobile applications that users are growing more and more accustomed to. Think Gmail, Facebook, Twitter, and some of the great mobile newsreaders like Feedly or Flipboard.

Single page apps are becoming increasingly popular. Sites that mimic the single page app behavior are able to provide the feel of a phone/tablet application. Angular helps to create applications like this easily. Let's dive into how we can create a 3 page application using Angular.

Node Server for Our Routing Application

In order for us to use ngRoute, we will want to use a server so that we can mimic how this app will work in browser. We're going to spin up a server because, as you'll see, we can't just open files locally anymore. Our Angular app will need to make requests for view files and we will run into issues without a server.

Let's create that server using Node and Express. This will be a good example because this will be our first glimpse of how we can build our Node and Angular applications together in the same codebase. Remember, we will be sticking to the client-server model, so we will need to maintain a good separation of our backend and frontend code.

Here is the file structure for our application:

```
- public/    // all of our frontend code (HTML/CSS/JS) will go here
----- views/
---------- pages/
----- index.html
----- js/
- package.json
- server.js
```

Those are the files we'll need to start up our server and our frontend Angular application. For now, we will only focus on `package.json` and `server.js` to create a server.

[85] https://github.com/angular-ui/ui-router

[86] https://docs.angularjs.org/api/ngRoute

Starting the Node Project

Like we've done before, `package.json` will house the packages and project information. Go ahead and run `npm init` or just create the file yourself. Here's the barebones `package.json` file.

```
{
  "name": "routing-app",
  "main": "server.js",
  "dependencies": {
    "express": "~4.9.1"
  }
}
```

Once you have that `package.json` file, run: `npm install` to get your dependencies (Express in this case) into your **node_modules** folder. Remember, you can also add dependencies from the command line using: `npm install express --save`.

Setting Up the Express Server

Like we have before, our server will be setup in `server.js`. Let's go ahead and create a very simple server where we send an `index.html` to our users.

```
// get the things we need
var express = require('express');
var app     = express();
var path = require('path');

// set the public folder to serve public assets
app.use(express.static(__dirname + '/public'));

// set up our one route to the index.html file
app.get('*', function(req, res) {
    res.sendFile(path.join(__dirname + '/public/views/index.html'));
});

// start the server on port 8080 (http://localhost:8080)
app.listen(8080);
console.log('Magic happens on port 8080.');
```

Whenever a request comes into our server (we do this by using the * wildcard method), we will send the user the `index.html` file which will have all of our Angular/HTML/CSS code. We are also using

Express to set the directory for static resources using `app.use(express.static())`. This means that whenever our client requests a file like a CSS file, image, or JS file, Node will serve that resource by looking in the public folder. All of our frontend code lives in this **public** folder so this is a good setup for our project.

Create a quick `public/views/index.html` file and just write whatever you want in it. Start your server with:

```
nodemon server.js
```

Now we can visit the app in our browser at `http://localhost:8080`.

This is what's so cool about Node. Even if you aren't using it in your final project, you can use it as a server for any projects you want to build. As we work on our Angular app, we can just work within this URL.

Now that we have the foundation ready to build out our Angular routing application, let's get moving on that.

Sample Application

We're just going to make a simple site with a home, about, and contact page. Angular is built for much more advanced applications than this, but this tutorial will show many of the concepts needed for those larger projects.

Goals

- Single page application
- No page refresh on page change
- Different data on each page

File Structure

These are the files that will be inside of our **public** folder that we set up in the earlier step.

```
public/
- views/
----- pages/        // the pages that will be injected into the main layout
---------- home.html
---------- about.html
---------- contact.html
----- index.html        // main layout
- js/
----- app.js        // stores all our angular code
----- app.routes.js // stores all our angular routes
```

The HTML For Our App

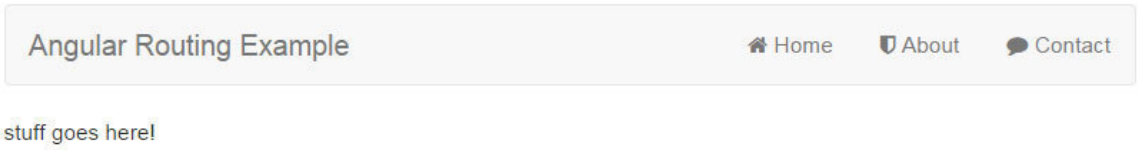
This is the simple part. We're using Bootstrap[87] and Font Awesome[88] to make our styles easier. Open up your `index.html` file and we'll add a simple layout with a navigation bar.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>My Routing App!</title>

    <!-- set the base path for angular routing -->
    <base href="/">

    <!-- CSS -->
    <!-- load bootstrap and fontawesome via CDN -->
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/\
css/bootstrap.min.css">
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/font-awesome/4.0\
.0/css/font-awesome.css">
    <style>
        body { padding-top:50px; }
    </style>

    <!-- JS -->
    <!-- load angular and angular-route via CDN -->
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular.mi\
n.js"></script>
```

[87] http://getbootstrap.com
[88] http://fontawesome.io

```
24      <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-ro\
25  ute.js"></script>
26
27      <!-- load our custom angular app files -->
28      <script src="js/app.js"></script>
29      <script src="js/app.routes.js"></script>
30  </head>
31  <body class="container">
32
33    <!-- HEADER AND NAVBAR -->
34    <header>
35      <nav class="navbar navbar-default">
36        <div class="navbar-header">
37          <a class="navbar-brand" href="/">Angular Routing Example</a>
38        </div>
39
40        <ul class="nav navbar-nav navbar-right">
41          <li><a href="/"><i class="fa fa-home"></i> Home</a></li>
42          <li><a href="/about"><i class="fa fa-shield"></i> About</a></li>
43          <li><a href="/contact"><i class="fa fa-comment"></i> Contact</a></li>
44        </ul>
45      </nav>
46    </header>
47
48    <!-- MAIN CONTENT AND INJECTED VIEWS -->
49    <main>
50
51      stuff goes here!
52
53      <!-- angular templating will go here -->
54      <!-- this is where content will be injected -->
55
56    </main>
57
58  </body>
59  </html>
```

We're loading Angular and Angular Route from the Google CDN. It is important to load up the routing module separately since we will be injecting that into our main Angular app. This is another example of how we can see dependency injection working in Angular applications. We'll also store our application routes inside of their own `app.routes.js` file which we will inject into our main application.



Angular Routing Foundation

Next up, we have to create our Angular application (`angular.module`) and then we will come back to the HTML to apply that app.

Angular Application

Module and Controller

We're going to setup our application. Let's create the angular module and controller. Check out the docs for more information on each. We'll create this in our JavaScript file (`public/js/app.js`):

```
1  angular.module('routerApp', [])
2
3  // create the controllers
4  // this will be the controller for the ENTIRE site
5  .controller('mainController', function() {
6
7    var vm = this;
8
9    // create a bigMessage variable to display in our view
10   vm.bigMessage = 'A smooth sea never made a skilled sailor.';
11 })
12
13 // home page specific controller
14 .controller('homeController', function() {
15
16   var vm = this;
17
18   vm.message = 'This is the home page!';
19 })
```

```
20
21  // about page controller
22  .controller('aboutController', function() {
23
24    var vm = this;
25
26    vm.message = 'Look! I am an about page.';
27  })
28
29  // contact page controller
30  .controller('contactController', function() {
31
32    var vm = this;
33
34    vm.message = 'Contact us! JK. This is just a demo.';
35  });
```

Let's add the module and controller to our HTML so that Angular knows how to bootstrap/initialize our application. To test that everything is working, we will also show the `vm.bigMessage` variable that we created.

We also have a `homeController`, `aboutController` and `contactController` here that we aren't using yet. This is because we will use these for our About and Contact pages when we eventually show them using Angular routing. `mainController` will encompass everything inside of the `<body>` tag while the other controllers are specific to each page.

Applying the Angular Application

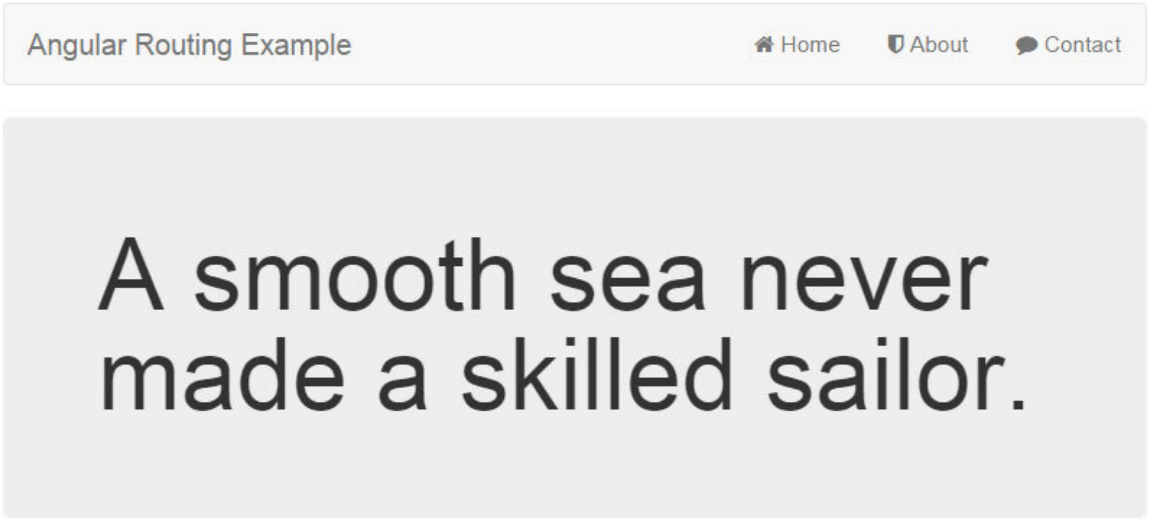
Let's go back into our `index.html` and apply our app and controller using **ng-app** and **ng-controller**. Since we already loaded our `app.js` file in the `<head>` of our document, we'll apply both to the `<body>` tag.

```
1  <body class="container" ng-app="routerApp" ng-controller="mainController as main\
2  ">
```

Notice again that we are using the **controller as** syntax when we write `mainController as main`. We will also show off our bigMessage inside the `<main>` section of our site:

```
<main>
    <div class="jumbotron">
        <h1>{{ main.bigMessage }}</h1>
    </div>

    <!-- angular templating will go here -->
    <!-- this is where content will be injected -->
</main>
```

Now if we display our `index.html` file in our browsers, we should be able to see our message.



Angular Routing Message Display

Inside of our `<main>`, we will now see the message that we created. Since we have our module and controller set up and there are no errors in our JavaScript console, we know that Angular is working properly. Now we will start working on using this layout to show the different pages all without ever refreshing the page.

Injecting Pages into the Main Layout

`ng-view` is an Angular directive that will include the template of the current route (**/home**, **/about**, or **/contact**) in the main layout file.

How does this work? Angular route (ngRoute) will look at the current URL, and then match that URL with a view that we specify. So if we are at the **/about** page, we'll tell ngRoute to grab a specific view file (pages/about.html in this case) and inject it where we put `ng-view`.

We haven't set up those routing rules we need yet, so let's add `ng-view` to our template right now and then move forward to the routing.

Inside of the `<main>` section of our site, delete the message part and add the following:

```
<main>

    <!-- angular templating -->
    <!-- this is where content will be injected -->

    <div ng-view></div>

</main>
```

Configuring Routes

Now this won't do much for our site yet. We have to define the routes and the views that will get used for each. This is how we use Angular's routing capabilities so that our pages don't refresh.

The Angular routing module provides us with the $routeProvider[89] service which is how we will configure the routes. Routes are defined on the `$routeProvider` object using `.when()`.

Let's create our routes now by creating a new file 'public/js/app.routes.js'. We'll also apply each of the controllers we created earlier to each page.

```
// inject ngRoute for all our routing needs
angular.module('routerRoutes', ['ngRoute'])

// configure our routes
.config(function($routeProvider, $locationProvider) {
    $routeProvider

        // route for the home page
        .when('/', {
            templateUrl : 'views/pages/home.html',
            controller  : 'homeController',
            controllerAs: 'home'
        })

        // route for the about page
        .when('/about', {
```

[89] http://docs.angularjs.org/api/ngRoute.\protect\char"0024\relaxrouteProvider

```
17              templateUrl : 'views/pages/about.html',
18              controller  : 'aboutController',
19              controllerAs: 'about'
20          })
21
22          // route for the contact page
23          .when('/contact', {
24              templateUrl : 'views/pages/contact.html',
25              controller  : 'contactController',
26              controllerAs: 'contact'
27          });
28
29      // set our app up to have pretty URLS
30      $locationProvider.html5Mode(true);
31  });
```

Now we have defined our routes with **$routeProvider**. As you can see by the configuration, you can specify the **route**, the **template file** to use, and even a **controller**. This way, each part of our application will use its own view and Angular controller.

We are also allowed to define `controllerAs` here and then each controller will be defined on our site automatically. For example, inside of our `home.html` file, when our router brings in this file, the controller will be defined as `homeController as home`.

Tip: Cleaning Up the Angular URL

By default, AngularJS will route URLs with a hashtag. For example, Angular will have `http://example.com/`, `http://example.com/#/about`, and `http://example.com/#/contact`.

It is very easy to get clean URLs and remove the hashtag from the URL.

There are 2 things that need to be done.

1. Configuring `$locationProvider`
2. Setting `<base>` in the `<head>` of our document for relative links

In Angular, the $location service[90] parses the URL in the address bar and makes changes to your application and vice versa.

We will use the `$locationProvider` service provided by Angular to set the HTML5 Mode of our app to true. This will ensure that our app uses the HTML5 History API (used by all the modern browsers). Older browsers will fall back to the hashtag method of showing URLs.

What is the HTML5 History API? It is a standardized way to manipulate the browser history using a script. This lets Angular change the routing and URLs of our pages without refreshing the page. For more information on this, here is a good HTML5 History API Article[91].

We already set the `<base>` and `$locationProvider.html5Mode(true)`, so we should see clean URLs in this application.

Injecting Routes into Main App

One of the main tenants of Angular development is dependency injection. We'll keep to that by injecting this new routes module we created into our main application. We already loaded the `app.routes.js` file in our `index.html` file so let's go into `app.js` and inject this routes module.

We will add it to the very first line like so:

```
angular.module('routerApp', ['routerRoutes'])
```

Just like that, we now have applied our routes to our application!

Configuring Views

We now have our routes and they are calling the appropriate files and controllers. To finish off this tutorial, we just need to define the pages that will be injected. We will also have them each display a message from its respective controller. This is a straightforward process, just some HTML and displaying the `{{ home.message }}`, `{{ about.message }}`, and `{{ contact.message }}` variables within each view.

`pages/home.html`

[90] http://docs.angularjs.org/guide/dev_guide.services.\protect\char"0024\relaxlocation

[91] http://diveintohtml5.info/history.html

```
<div class="jumbotron text-center">
    <h1>Home Page</h1>

    <p>{{ home.message }}</p>
</div>
```

pages/about.html

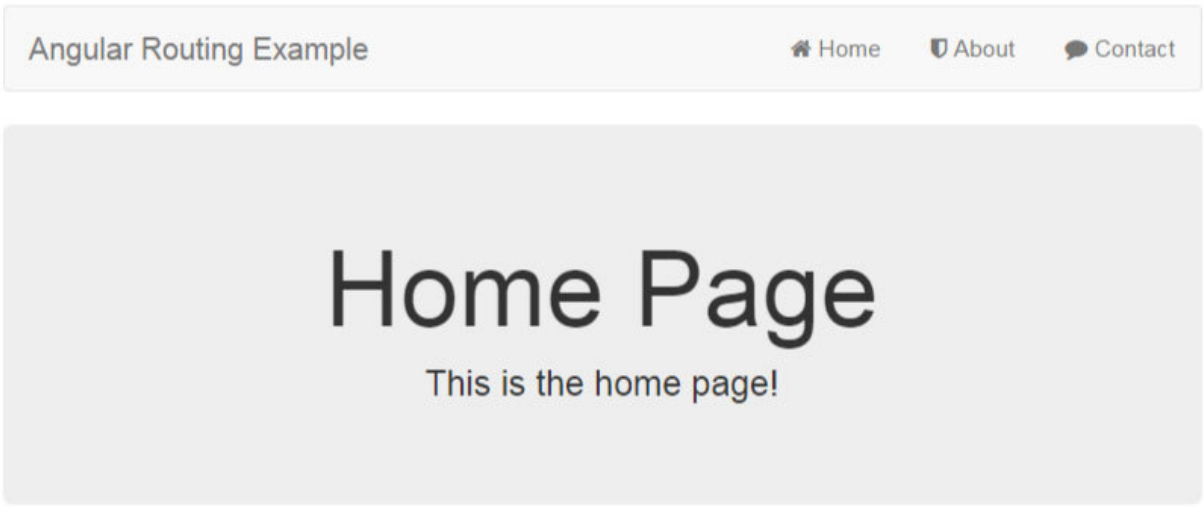
```
<div class="jumbotron text-center">
    <h1>About Page</h1>

    <p>{{ about.message }}</p>
</div>
```

pages/contact.html

```
<div class="jumbotron text-center">
    <h1>Contact Page</h1>

    <p>{{ contact.message }}</p>
</div>
```

We are prefixing each of the variables with the controller name for each HTML file that we defined in our routes using `controllerAs`.

Visit your application and click through the pages. You'll see the data change and your message variable change **all without a page refresh**!



Angular Routing Home Page

Conclusion

There we have it! A single page application with 3 different pages. Each page will also bring in a different Angular controller so that data can be different across our site.

You can see the value of this as it gives our basic websites into more of an integrated application feeling.

As cool as this site is without refreshing the page for routing, we will make it even cooler by animating these pages. You have probably already seen mobile apps that slide in and out of view as you click items. We'll turn what we just built into a clean and impressive animated site (don't worry, it won't be anything too flashy... unless that's what you want).

Note: An Alternative to ngRoute

While ngRoute is the routing tool built to work most closely with Angular, another tool called UI Router was built as another routing framework by the AngularUI team. AngularUI[92] is the companion suite that works hand in hand with Angular applications. They provide many useful tools like UI Router, UI Bootstrap for using Bootstrap JavaScript components within Angular, and a few other awesome tools.

UI Router provides a different approach than ngRoute in that it changes your application views based on **state of the application and not just the route URL**. This means that your application is not tied to the URL path and you can adjust what templates show based on application state (ie if a user is logged in or not).

There are also great benefits by having the ability to **nest states**. ngRoute doesn't provide this functionality and for more advanced UIs, it is helpful to have nested states like having multiple sidebar panels and moving components you would see more in mobile applications.

We will be using ngRoute since it is the standard, but for more reading on UI Router, here's a good starting article: AngularJS Routing Using UI-Router[93]. In the future, the Angular team has stated that the next routing module they build will have features from both ngRoute and UI Router.

---

[92] http://angular-ui.github.io/

[93] http://scotch.io/tutorials/javascript/angular-routing-using-ui-router

Animating Angular Applications

What's the point in creating these great single page applications if they don't act like it? We want our applications to act as similar to native applications as possible so that we can blur the lines between web and native applications and impress our users even further.

AngularJS provides a great way to create single page applications. This allows for our site to feel more and more like a native mobile application since we have a single page app. To make it feel even more native, we can add transitions and animations using ngAnimate module[94].

This module allows us to create beautiful looking applications and we'll be looking at **how to animate ng-view**.

Animating Our Routing Application

We are going to animate the single page application we built in the last chapter. Thanks to our routing, our site will not refresh as people click between pages. Now we will:

- have **ngRoute** handle our routing (done in the last chapter)
- use **ngAnimate** to create page animations
- use **CSS animations** to handle page animations

Before we can get to the animations, let's talk a bit about how the **ngAnimate module works**.

How Does the ngAnimate Module Work?

ngAnimate will add and remove CSS classes to different Angular directives based on if they are entering or leaving the view. This happens automatically without us needing to configure anything.

The classes that are added and removed automatically when we load up our site are `.ng-enter` or `.ng-leave`. These are applied to the div where we apply the `ng-view` directive. We won't see these yet, but if you go into your browser inspector when you are clicking between pages, you'll see the `.ng-enter` and `.ng-leave` classes being applied.

[94] http://docs.angularjs.org/api/ngAnimate

How Animations Are Applied

This module keeps things clean and simple. All it does is add classes. How does it create animations though? It actually doesn't. It gives us the tools to do so and then we are in control of the animations by creating CSS animations. By using CSS animations, we ensure that our application can work in mobile browsers, desktop browsers, and is adheres to the latest standards (CSS3).

We'll be using CSS animations from the Animate.css[95] project since those classes[96] are easily taken into our own projects.

Directives that Use Animation

ngAnimate also applies these classes to more Angular directives automatically. In addition to the `ngView` module, these classes are also applied to ngRepeat, ngInclude, ngIf, ngSwitch, ngShow, ngHide, ngView, and ngClass. This means that you can animate multiple parts of your application. We'll be focusing on **ngView** in this chapter, but the tactics can be easily applied to the other directives.

Animating Our Routing Application

There are three steps to using the ngAnimate module inside of our application from the last chapter:

1. Link to the `angular-animate.js` file
2. Inject the `ngAnimate` module into our main Angular module
3. Add the `animate.css` stylesheet to have prebuilt CSS animations

We'll use a CDN for the `angular-animate.js` and `animate.css` resources that we need. Go back into your **index.html** file and add the two resources to the `<head>` of your document.

[95] http://daneden.github.io/animate.css/

[96] https://github.com/daneden/animate.css/blob/master/animate.css

```
<head>
    <meta charset="utf-8">
    <title>My Routing App!</title>

    <!-- set the base path for angular routing -->
    <base href="/">

    <!-- CSS -->
    <!-- load bootstrap and fontawesome via CDN -->
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/\
css/bootstrap.min.css">
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/font-awesome/4.0\
.0/css/font-awesome.css">
    <link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/animate.c\
ss/3.1.1/animate.min.css">

    <style>
        body { padding-top:50px; }
    </style>

    <!-- SPELLS -->
    <!-- load angular and angular-route via CDN -->
    <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular.min.js"\
></script>
    <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-route.j\
s"></script>
    <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-animate\
.js"></script>

    <script src="js/app.js"></script>
    <script src="js/app.routes.js"></script>
</head>
```

The two main lines that we are looking to add here are for `animate.css` and `angular-animate.js`:

```
<link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/animate.css/3.1.1/\
animate.min.css">
<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-animate.js"\
></script>
```

With the resources we now need, inside of our `app.js` file where we defined our main Angular module (`angular.module('routerApp', ['routerRoutes'])`, we will need to inject the **ngAnimate**

module. Without injecting the module into our application, we won't be able to use it. This is once again the Angular dependency concepts in action.

Injecting the ngAnimate Module

To inject the ngAnimate module, we'll follow the same process that we used to inject our **routerRoutes** module. Just add it when declaring the `angular.module()`.

Here's the module injected in our `app.js` file:

```
1     angular.module('routerApp', ['routerRoutes', 'ngAnimate'])
```

Now when we start up our application and inspect element in our browser, we can see the `ng-enter` and `ng-leave` classes being applied. This won't affect how our application looks or feels... yet. We just need to apply CSS animations now.

CSS Animations and Positioning

In our last chapter, we had all of our custom CSS (just the `1` line for body padding) inline in our `index.html` file. Let's move that into its own file now. Inside of the public folder, create a 'public/css/style.css' file. We will add that to our `index.html` file instead of the inline file:

```
1     <link rel="stylesheet" href="css/style.css">
```

The other change that we will make is to wrap our `<div ng-view></div>` with another div so that we can call it specifically. This will look like so:

```
1     <div id="content-views">
2         <div ng-view></div>
3     </div>
```

Now we just need to go into our newly created stylesheet and add our styles. Don't forget to link your stylesheet '' in your index.html.

```
1     body                          { padding-top:50px; }
2
3     #content-views                { position:relative; }
4
5     .ng-enter, .ng-leave          { width:100%; position:absolute; }
6     .ng-enter                     {
7       -webkit-animation:zoomInDown 1.2s both ease-in;
8       -moz-animation:zoomInDown 1.2s both ease-in;
9         animation:zoomInDown 1.2s both ease-in;
10    }
11    .ng-leave                     {
12      -webkit-animation:zoomOutDown 0.5s both ease-in;
13      -moz-animation:zoomOutDown 0.5s both ease-in;
14        animation:zoomOutDown 0.5s both ease-in;
15    }
```

Vendor Prefixes: Make sure you add the `-moz-animation` and `-webkit-animation` vendor prefixes when pushing this code to production for cross compatibility in browsers.

Go ahead and click through the pages in your application and watch as the pages fly out and fly in! Just like that, we have an animated application! Feel free to try out the rest of the animate.css[97] classes to see the crazy different effects you can create.

Conclusion

This should give you just a sample of how putting together the Angular routing and animating modules together can quickly build complex looking apps. These techniques can be used to build a multitude of applications in all sorts of different styles. So far, on the Angular side of things we have:

- Set up an AngularJS module
- Applied Angular to our HTML views
- Routed our application using ngRoute
- Animated our application using ngAnimate and CSS classes

Having the knowledge of the above will let us build awesome looking applications. We'll explore these concepts further as we build out more sample applications in the coming chapters.

We've focused on the frontend Angular parts so far; next up is how to make Angular applications dynamic. We'll be moving towards integrating the frontend and the backend, but first let's make sure that the foundation of our application is solid so that our applications are scalable and easy to understand.

[97] http://daneden.github.io/animate.css/

MEAN Stack Application Structure

Up to this point, we've been able to place most of our code into a few main files (`server.js` for Node and `app.js` for Angular). As we start moving to larger MEAN stack applications, keeping all of our code bunched into these main files will become cumbersome. It will be hard to maintain our code, find certain parts of our applications, and worst of all, share convoluted code with other developers.

With this in mind, let's take a look at our application structure. Organizing something this simple goes a long way. This will be a short chapter, but a very important one.

Sample Organization

So far, our applications have had this file structure:

```
// node applications
- app/
----- models/

// angular applications
- public/
----- css/
----- js/
---------- controllers/
---------- services/
----- app.js
----- views/
---------- pages/
----- index.html

- package.json
- server.js
```

On the **Node side of things**, we can't keep everything in `server.js`. That file could become a giant spaghetti mix of code when we start putting more routes, more configuration, and more application functionality. That has to be broken down into a more organized structure.

On the **Angular side of things**, this structure could become confusing. Where would we put `lib` files like Bootstrap, or Font Awesome? Would our `js` folder hold all of those things including our Angular application files?

A Better Structure

So what would a better structure look like? Let's combine both the Node and Angular sides since we will be moving towards full stack applications next.

Here is a better organization that lays a good foundation for our applications. This is a **modular** approach that allows us to add and remove components from our applications easily and simply.

```
- app/
----- models/
----- routes/
- public/
----- assets/  // the base css/js/images for our app (not Angular files)
---------- css/      // some custom css
---------- js/       // some custom js (not Angular files)
---------- img/
---------- libs/     // libraries like bootstrap, angular, font-awesome
----- app/      // the Angular part of our application
---------- controllers/
---------- services/
---------- app.js
---------- app.routes.js
---------- views/
---------------- pages/
---------------- index.html
- package.json
- config.js- server.js
```

Let's break down what is happening here. A lot of it is just moving things around into its own file/folder.

App Folder

We have our normal models folder here. What's new is that we have a `routes/` folder now. We can place our basic routes here and call them from our main `server.js` file. Since the routes are the main part of `server.js`, moving those out of the main file will clean it up substantially.

We can even create multiple routes files like `routes/app.js` and `routes/api.js` so that we can differentiate the parts of our application.

config.js

We are just moving configuration variables into this file. This helps since we can set variables like environment, database settings, and other application specific settings.

We can even grab these settings from a dashboard in the future so that we can have a dashboard UI where we can have users log in to set these.

Public Folder

This is where the majority of application movement has happened. Before, we stored everything (Angular files and basic asset files) in the `public/js` folder. This can become confusing as our application grows so now we have it set up so that **assets and application files live in separate directories**.

Anything Angular specific like controllers, services, routes, and views will live in the `app/` folder. Assets like images, custom CSS or JS, and libraries will live in the `assets/` folder.

Here we have also created a `app.routes.js` file. In the future as we create more routes for our application, we may want to move these into its own `routes/` folder like we did for the Node side.

Now that we have seen a better structure, let's take our Chapter 10 application and arrange it so that it adheres to this practice. This will lay the groundwork for moving forward when we create the User CRM.

Organizing Node.js - Backend

Since Chapter 10 was about creating a Node API and authenticating it, we only have Node code to work with here.

Let's see how we can rearrange our Node code (say that 10 times fast… Node code, Node code, Node code). Go ahead and copy your Chapter 10 code and create a new project for it.

Here is an overview of the steps we will be taking:

- move configuration variables into a new `config.js`
- move routes into a new `routes/api.js` file
- create a catchall route to get ready for MEAN app

Configuration Files

We will be moving our configuration variables out of our `server.js`. For our purposes, we don't have many configuration variables, just our `port`, our `database`, and the `secret` that we will be using when configuring JSON Web Tokens.

Create a new file in the root directory called `config.js`.

The following is all we need:

```
module.exports = {
  'port': process.env.PORT || 8080,
  'database': 'mongodb://node:noder@novus.modulusmongo.net:27017/Iganiq8o',
  'secret': 'ilovescotchscotchyscotchscotch'
};
```

Now we can use this file inside of our `server.js` file using `require()`. The following lines will be edited:

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

// connect to our database (hosted on modulus.io)
mongoose.connect(config.database);

// START THE SERVER
// ==================================
app.listen(config.port);
console.log('Magic happens on port ' + config.port);
```

And the following lines have been moved out, so can now be deleted from 'server.js':

```
// super secret for creating tokens
var superSecret = 'ilovescotchscotchyscotchscotch';
```

While this may not seem like it did a lot, the concept will go a long way when we have over 20+ settings for our application.

Note

What is module.exports?

By default, JavaScript doesn't have a way to pass information between different files. `module.exports` is Node's way of fixing this problem.

You can think of `module.exports` as a giant object for our application. When our application starts, this `module.exports` object looks like this:

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

As we start pulling things into our app with `require()`, everything is added to this object.

So now as we add that new `config.js` file with `require('./config.js');`, we have access to those variables since we passed them through `module.exports`.

This is how we will be passing information to and from all of our files and we'll see this in practice again when we create our routes files.

It is also important to note that many tutorials on the web will switch between `module.exports` and `exports`. They mean the same thing and can be used interchangeably.

Routes

Just like our configuration variables, let's move our routes into their own files. Create a folder called `app/routes/` and a file called `app/routes/api.js`.

When creating new files, we will have to `require()` anything that is needed by that file. In our case, we will need `body-parser`, our `User` model, our `config` file for the `secret`, and `jsonwebtoken` since those were used in our routes.

We will also need `app` and `express` but we will be passing those into our routes since we want our application to use the same `app` object we create in `server.js`. Let's take a look at how we'll structure our `routes/api.js` file that holds all of our API routes. Go into your existing 'server.js' file from chapter 10 and grab out all of the route code, starting at 'var apiRouter = express.Router();'. We will be wrapping this in our module.exports so that it can be passed between our javascript files. Also take note of the changes to our variable superSecret and the packages we are including at the top.

```
var User       = require('../models/user');
var jwt        = require('jsonwebtoken');
var config     = require('../../config');

// super secret for creating tokens
var superSecret = config.secret;

module.exports = function(app, express) {

  var apiRouter = express.Router();

  // route to authenticate a user (POST http://localhost:8080/api/authenticate)
```

```
13    apiRouter.post('/authenticate', function(req, res) {
14      console.log(req.body.username);
15
16      // find the user
17      // select the password explicitly since mongoose is not returning it by defa\
18  ult
19      User.findOne({
20        username: req.body.username
21      }).select('password').exec(function(err, user) {
22
23        if (err) throw err;
24
25        // no user with that username was found
26        if (!user) {
27          res.json({
28            success: false,
29            message: 'Authentication failed. User not found.'
30          });
31        } else if (user) {
32
33          // check if password matches
34          var validPassword = user.comparePassword(req.body.password);
35          if (!validPassword) {
36            res.json({
37              success: false,
38              message: 'Authentication failed. Wrong password.'
39            });
40          } else {
41
42            // if user is found and password is right
43            // create a token
44            var token = jwt.sign(user, superSecret, {
45              expiresInMinutes: 1440 // expires in 24 hours
46            });
47
48            // return the information including token as JSON
49            res.json({
50              success: true,
51              message: 'Enjoy your token!',
52              token: token
53            });
54          }
```

```
55
56        }
57
58      });
59    });
60
61    // route middleware to verify a token
62    apiRouter.use(function(req, res, next) {
63        // do logging
64        console.log('Somebody just came to our app!');
65
66      // check header or url parameters or post parameters for token
67      var token = req.body.token || req.query.token || req.headers['x-access-token\
68 '];
69
70      // decode token
71      if (token) {
72
73        // verifies secret and checks exp
74        jwt.verify(token, superSecret, function(err, decoded) {
75          if (err) {
76            return res.json({ success: false, message: 'Failed to authenticate tok\
77 en.' });
78          } else {
79            // if everything is good, save to request for use in other routes
80            req.decoded = decoded;
81
82            next(); // make sure we go to the next routes and don't stop here
83          }
84        });
85
86      } else {
87
88        // if there is no token
89        // return an HTTP response of 403 (access forbidden) and an error message
90        return res.status(403).send({
91          success: false,
92          message: 'No token provided.'
93        });
94
95      }
96
```

```
97    });
98
99    // test route to make sure everything is working
100   // accessed at GET http://localhost:8080/api
101   apiRouter.get('/', function(req, res) {
102       res.json({ message: 'hooray! welcome to our api!' });
103   });
104
105   // on routes that end in /users
106   // ----------------------------------------------------
107   apiRouter.route('/users')
108
109       // create a user (accessed at POST http://localhost:8080/users)
110       .post(function(req, res) {
111
112           var user = new User();      // create a new instance of the User model
113           user.name = req.body.name;  // set the users name (comes from the requ\
114 est)
115           user.username = req.body.username;  // set the users username (comes f\
116 rom the request)
117           user.password = req.body.password;  // set the users password (comes f\
118 rom the request)
119
120           user.save(function(err) {
121               if (err) res.send(err);
122
123               // return a message
124               res.json({ message: 'User created!' });
125           });
126
127       })
128
129       // get all the users (accessed at GET http://localhost:8080/api/users)
130       .get(function(req, res) {
131           User.find(function(err, users) {
132               if (err) res.send(err);
133
134               // return the users
135               res.json(users);
136           });
137       });
138
```

```
139   // on routes that end in /users/:user_id
140   // ----------------------------------------------------
141   apiRouter.route('/users/:user_id')
142
143       // get the user with that id
144       .get(function(req, res) {
145           User.findById(req.params.user_id, function(err, user) {
146               if (err) res.send(err);
147
148               // return that user
149               res.json(user);
150           });
151       })
152
153       // update the user with this id
154       .put(function(req, res) {
155           User.findById(req.params.user_id, function(err, user) {
156
157               if (err) res.send(err);
158
159               // set the new user information if it exists in the request
160               if (req.body.name) user.name = req.body.name;
161               if (req.body.username) user.username = req.body.username;
162               if (req.body.password) user.password = req.body.password;
163
164               // save the user
165               user.save(function(err) {
166                   if (err) res.send(err);
167
168                   // return a message
169                   res.json({ message: 'User updated!' });
170               });
171
172           });
173       })
174
175       // delete the user with this id
176       .delete(function(req, res) {
177           User.remove({
178               _id: req.params.user_id
179           }, function(err, user) {
180               if (err) res.send(err);
```

```
181
182                 res.json({ message: 'Successfully deleted' });
183             });
184         });
185
186     return apiRouter;
187 };
```

If you look carefully, you'll see this doesn't differ too much from how our code looked in `server.js`. The main difference is that we are requiring `config.js` and using `config.secret`.

We are also passing back a function into `module.exports` and requiring that `app` and `express` are passed in. This is so we can use the `express` object to get an instance of `express.Router();`. Currently, we are not using the `app` object, but it note that if you are not using the **express.Router()**, you could directly specify routes on the `app` object as well.

We will then `return apiRouter;` so that we can use it in `server.js`.

Now that this file is created, we just need to call it in `server.js`. This is where we can see just how clean our main file has become and how much easier it is to read than before.

These are the lines to use in `server.js` when calling our newly created routes file:

```
1 // API ROUTES ------------------------
2 var apiRoutes = require('./app/routes/api')(app, express);
3 app.use('/api', apiRoutes);
```

If you didn't do it earlier, go through and remove those routes and calls to packages that we moved into our `api.js` file. And that's it! Now we have moved about 170 lines of code out of our `server.js`!

Catchall Route

The last thing we need to do is create a catchall route to pass users to an Angular application. Currently we have a route that shows our home page in `server.js`. It looks like this:

```
1 // basic route for the home page
2 app.get('/', function(req, res) {
3     res.send('Welcome to the home page!');
4 });
```

For creating MEAN applications, our Node routes will take place here and then any request sent to a route that isn't handled by Node should be taken care of by Angular. This is where a **catchall route** comes in handy.

Any route not handled by Node will be passed to Angular. Creating the route is very easy. We are going to delete the route we created for the home page and create the following:

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

...

var apiRoutes...

// MAIN CATCHALL ROUTE ---------------
// SEND USERS TO FRONTEND ------------
// has to be registered after API ROUTES
app.get('*', function(req, res) {
  res.sendFile(path.join(__dirname + '/public/app/views/index.html'));
});
```

Make sure that `path` is loaded in the packages section since that is required to pass an HTML file. Using the `*` will match all routes. It is important to put this route **after the API routes** since we only want it to catch routes not handled by Node.

If this were placed above the API routes, then our user would always be sent the `index.html` file and never even get to the API routes.

Final Node server.js

Here's the full `server.js` file:

```
// BASE SETUP
// =====================================

// CALL THE PACKAGES --------------------
var express    = require('express');    // call express
var app        = express();         // define our app using express
var bodyParser = require('body-parser');  // get body-parser
var morgan     = require('morgan');     // used to see requests
var mongoose   = require('mongoose');
var config     = require('./config');
var path     = require('path');

// APP CONFIGURATION ==================
// ====================================
// use body parser so we can grab information from POST requests
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

```

```
// configure our app to handle CORS requests
app.use(function(req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST');
  res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type, \
Authorization');
  next();
});

// log all requests to the console
app.use(morgan('dev'));

// connect to our database (hosted on modulus.io)
mongoose.connect(config.database);

// set static files location
// used for requests that our frontend will make
app.use(express.static(__dirname + '/public'));

// ROUTES FOR OUR API =================
// =====================================

// API ROUTES ------------------------
var apiRoutes = require('./app/routes/api')(app, express);
app.use('/api', apiRoutes);

// MAIN CATCHALL ROUTE ---------------
// SEND USERS TO FRONTEND ------------
// has to be registered after API ROUTES
app.get('*', function(req, res) {
  res.sendFile(path.join(__dirname + '/public/app/views/index.html'));
});

// START THE SERVER
// =================================
app.listen(config.port);
console.log('Magic happens on port ' + config.port);
```

Now that we have organized the Node side of things, let's lay the groundwork for our Angular application.

Organizing AngularJS - Frontend

The only file we will create here is the `index.html` file since that is what we are returning with our Node catchall route. We'll just create the folders for the rest of the application so we have the foundation ready to go.

Here's the file structure now:

```
- app/
----- models/
----- routes/
---------- api.js
- public/
----- assests/
---------- css/
--------------- style.css
---------- js/
---------- img/
---------- libs/
----- app/
---------- controllers/
---------- services/
---------- app.routes.js
---------- app.js
---------- views/
--------------- pages/
--------------- index.html
- package.json
- config.js- server.js
```

Go ahead and create the folders and files within the `public/` folder.

Here is a good starting point for our `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My MEAN App</title>
</head>
<body>

HELLO!

</body>
</html>
```

Testing Our Newly Organized App

Let's make sure that everything is working properly.

Run

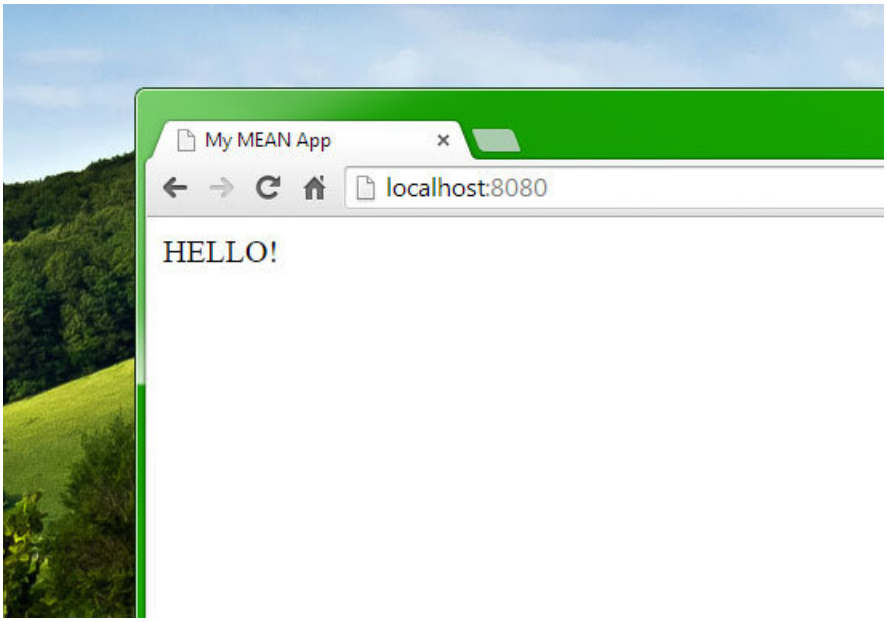
```
npm install
```

if you haven't already to bring in the files that we'll need.

Then go ahead and run:

```
nodemon server.js
```

and visit the application in your browser!



Testing App structure

That was a lot of setting up. This shows you how important something as simple as directory structure is for our applications. This ensures our scalability and our ability to add on new features and components in the future. A solid infrastructure and file directory will also help other developers we work with; they will be able to understand how things are laid out.

Now let's continue our move to a full stack MEAN application by talking about how we can link the frontend and the backend with Angular services. We will also be using services to handle frontend authentication.

After we learn about Angular services and authentication, we'll move on to creating our full stack MEAN app.

Angular Services to Access an API

Separating server-side and client-side applications means that there has to be something that links the two together. When creating Angular applications, **services are the glue between frontend and backend**.

Services are the way we contact an API, get data back, and pass it to our Angular controllers. The controller then passes that information to our views and we have a complete separation of duties like the MVC model[98] states we should.

We will be using Angular services and Angular's built-in $http[99] service to make HTTP requests to our API. You can also use this to make requests to any API.

The beauty of Angular services is that they just make API calls. This means we are able to hook up Angular to any backend API. It doesn't matter if we have a Node API, PHP API, or any other language. Angular can talk to them all as long as we have a backend that allows calls for information and returns valid JSON data.

Types of Angular Services

There are 3 types of Angular services: **service**, **factory**, and **provider**. Each has its own specific use cases.

Service: The simplest type. Services are instantiated with the `new` keyword. You have the ability to add properties to a service and call those properties from within your controllers.

Factory: The most popular type. In a Factory, you create an object, add properties to that object, and then return it. Your properties and functions will be available in your controllers.

Provider: Providers are the most complex of the services. They are the only service that can be passed into the `config()` function to declare application-wide settings.

For our purposes, we will be using **factories**. They provide a good middle ground of functionality between services and providers.

The $http Module

The $http module[100] gives us a way to communicate with remote HTTP servers. If you are familiar with jQuery API calls, then you will see that the syntax is very similar.

Here's an example call to get all users from our Node API we created earlier:

[98] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[99] https://docs.angularjs.org/api/ng/service/\protect\char"0024\relaxhttp

[100] https://docs.angularjs.org/api/ng/service/\protect\char"0024\relaxhttp

```
angular.module('myApp', [])

// inject $http into our controller
.controller('mainController', function($http) {

  var vm = this;

  // make an API call
  $http.get('/api/users')
    .then(function(data) {

      // bind the users we receive to vm.users
      vm.users = data.users;

    });

});
```

The $http module can be injected into an Angular module, whether it be a controller or service. In the above example, it was injected into a controller. However, we do not want to call the $http module directly in a controller. That logic should be moved into a service so that we have a clear separation of duties. Services get the data while the controller receives it and sends to our views.

Let's look at how we can use the $http module in a factory.

A Sample Angular Service

We're going to build out a sample factory and see how we can use it inside of an Angular controller.

Angular Factory

Here's a quick file called `userService.js` that will access an API that delivers user data.

```
angular.module('stuffService', [])

.factory('Stuff', function($http) {

  // create the object
  var myFactory = {};

  // a function to get all the stuff
  myFactory.all = function() {
    return $http.get('/api/stuff');
  };

  return myFactory;

});
```

We have just created our first Angular service! The syntax is fairly straightforward. We have a **factory** called `Stuff` and a function called `all`. This function will create an HTTP GET call using the `$http` module and return a promise object. We can then act on the promise object by accessing `success()`, `error()`, or `then()`. For our purposes, `success()` will do just fine.

Tip

Promises

Promises are a hard concept to grasp if you are new to JavaScript development. Promises help us deal with the asynchronous nature of JavaScript by acting as sort of a placeholder. We make a request for some kind of data and the promise will wait for the response so the rest of our program can continue with whatever else its doing. Once our promise receives the result, it will notify us and we can carry out our next action based on that result whether it be a success or error.

Here are some good resources for understanding promises:

- promisejs.org[101] - A good vanilla JS explanation

[101] https://www.promisejs.org/

- Promises in AngularJS, Explained as a Cartoon[102]

We will be using promises to handle grabbing data from our API. This ensures that we get the data back as we want it and it is not lost. The `$http` module will return a promise object that we can use.

Now that we have created our `stuffService` Angular module and `Stuff` factory, let's look at how we can use this factory within an Angular controller.

Using a Service in a Controller

```
// inject the stuff service into our main Angular module
angular.module('myApp', ['stuffService'])

// create a controller and inject the Stuff factory
.controller('userController', function(Stuff) {

  var vm = this;

  // get all the stuff
  Stuff.all()

    // promise object
    .success(function(data) {

      // bind the data to a controller variable
      // this comes from the stuffService
      vm.stuff = data;
    });

});
```

Just like that, we have created an Angular service, injected it into a controller, and grabbed all our user data. It is important to understand the separation of concerns here.

> The service is responsible for grabbing data from an external resource (our API) while the controller is responsible for facilitating that data to our views.

We have a separation of concerns between our controllers and services. Services get the data and controllers facilitate that data to the view.

[102] http://andyshora.com/promises-angularjs-explained-as-cartoon.html

User Service

Let's create a service that we will use in our final application. We'll call this our `userService`. The benefit of creating a standalone service like this is that it will be reusable across other projects.

Just like we created API endpoints on the backend, we will use the `$http` module to create functions in our service to go and grab from each of the endpoints. We will need to handle the following:

- get a single user
- get a list of all users
- create a user
- update a user
- delete a user

All of these tasks together combine to make your normal CRUD operations. For reference, let's see how we can match up the frontend needs with the backend API. Our matching table will also show the HTTP verb that needs to be used since we want to stick to the REST pattern we created.

Task	Node API	Angular Service Function
single user	GET /api/users/:user_id	get(id)
list users	GET /api/users	all()
create user	POST /api/users	create(userData)
update a user	PUT /api/users/:user_id	update(id, userData)
delete user	DELETE /api/users/:user_id	delete(id)

Our frontend functions can be named anything. If you wanted to, you could even get these to match the backend functions. For example, instead of `create(userData)`, you could use `postUser(userData)` to keep with the HTTP verb trend.

Remember that when calling these functions within an Angular controller, you will have to prefix the function name with the factory name, so it's nice to keep them simply named. For example, to get all of the users, we will call `User.all()` and to get a single user we will call `User.get(id)`. Having a clean set of function names makes development easier across an entire team.

Whatever you choose, the most important thing is that there is a set standard on the backend and frontend and that all developers involved with the project know the exact naming schemes across the entire stack.

Let's create an Angular module for our **userService**. We will define our Angular module and create a factory called `User` to go along with it.

```
angular.module('userService', [])

.factory('User', function($http) {

  // create a new object
  var userFactory = {};

  // get a single user
  userFactory.get = function(id) {
    return $http.get('/api/users/' + id);
  };

  // get all users
  userFactory.all = function() {
    return $http.get('/api/users/');
  };

  // create a user
  userFactory.create = function(userData) {
    return $http.post('/api/users/', userData);
  };

  // update a user
  userFactory.update = function(id, userData) {
    return $http.put('/api/users/' + id, userData);
  };

  // delete a user
  userFactory.delete = function(id) {
    return $http.delete('/api/users/' + id);
  };

  // return our entire userFactory object
  return userFactory;

});
```

Notice how we are using the `$http` module to create requests to our various API endpoints. We have `$http.get()`, `$http.post()`, `$http.put()`, and `$http.delete()` all accounted for here.

There is nothing too fancy happening here. Our service will return the data from our calls to the API. With our user service done, we'll be able to integrate this into our full MEAN stack application in a couple chapters.

This drop in functionality is what is so great about Angular. We are creating a set of modules and then injecting them into one another. We will inject this into our User CRM application that we create in Chapter 17.

Note: If your API is hosted on a separate server, then you will need to prefix all these `/api/` URLs with your server URL like so: `$http.get('http://example.com/api/users, ...)`.

Next up, we'll create another service that we will use to handle authentication. Services can be used for more than just grabbing data. They can act as data objects and handle all the functions and properties necessary for a certain operation, in this case, authentication.

Angular Authentication

We now have the foundation for our MEAN stack application. We have talked extensively about the separation between backend server-side and frontend client-side code.

Up to this point, we have created a Node API and used JSON web tokens to authenticate it. This has all been server-side code and now we will integrate it with the frontend using the information we just learned about

The goal here is to start writing an Angular application and not have to edit the backend. Treating your applications like this can simulate similar situations where the backend API is built and maintained by someone other than yourself (like Facebook, Twitter, or GitHub APIs). We only have access to the frontend code.

The first step when building an application that will talk to a backend API is handling authentication. After all, we won't be able to grab any data from the API unless we are authenticated.

Hooking Into Our Node API

When we created our Node API with JSON Web Token authentication in chapters 9 and 10, we developed an API endpoint (`POST http://localhost:8080/api/authenticate`) where we could send a username and password to receive a JWT.

Since our backend already has the tools to authenticate and provide a token, all we need to do is wire up the frontend Angular application to hit that endpoint, store the JWT client-side, and then we will be able to access all of the authenticated routes in our API.

Authentication Service

We'll be building two different services, one for authentication and one for connecting to our API and grabbing user data. As a general rule, if you are grabbing or sending data from/to an outside source, a service will likely be your tool of choice.

To authenticate our users, we will create an `authService`. This will have 3 main functions:

- main auth functions (login, logout, get current user, check if logged in)
- token auth functions (get the token, save the token)
- auth interceptor (attach the token to HTTP requests, redirect if not logged in)

Each of the three factories has a very specific purpose which is why we can't just create a single one.

The main auth functions are the ones that will be exposed to our application and usable within controllers.

The auth interceptor will be responsible for attaching the token to all HTTP requests. Remember in token based authentication, the token is required with all authenticated requests; this interceptor is how we will achieve the attachment. This interceptor will also be responsible for redirecting a user to the login page if they are not authenticated.

The token auth functions will be more of private factory for use within the other two auth factories. It will just have to save a token or get a token out of local storage.

Here is a quick overview of how our file will look:

```
angular.module('authService', [])

// ===================================================
// auth factory to login and get information
// inject $http for communicating with the API
// inject $q to return promise objects
// inject AuthToken to manage tokens
// ===================================================
.factory('Auth', function($http, $q, AuthToken) {

  // create auth factory object
  var authFactory = {};

  // handle login

  // handle logout

  // check if a user is logged in

  // get the user info

  // return auth factory object
  return authFactory;

})

// ===================================================
// factory for handling tokens
// inject $window to store token client-side
```

```
// ================================================
.factory('AuthToken', function($window) {

  var authTokenFactory = {};

  // get the token

  // set the token or clear the token

  return authTokenFactory;

})

// ================================================
// application configuration to integrate token into requests
// ================================================
.factory('AuthInterceptor', function($q, AuthToken) {

  var interceptorFactory = {};

  // attach the token to every request

  // redirect if a token doesn't authenticate

  return interceptorFactory;

});
```

These are the three factories we will be creating all lumped into one Angular module called `authService`. For each, we are creating a new object, attaching functions to the object, and returning the object.

Notice that we are able to inject a factory into another factory. We are using the `AuthToken` factory in the other two factories. We are also injecting some Angular modules into our factories like `$http`, `$q`, and `$location`.

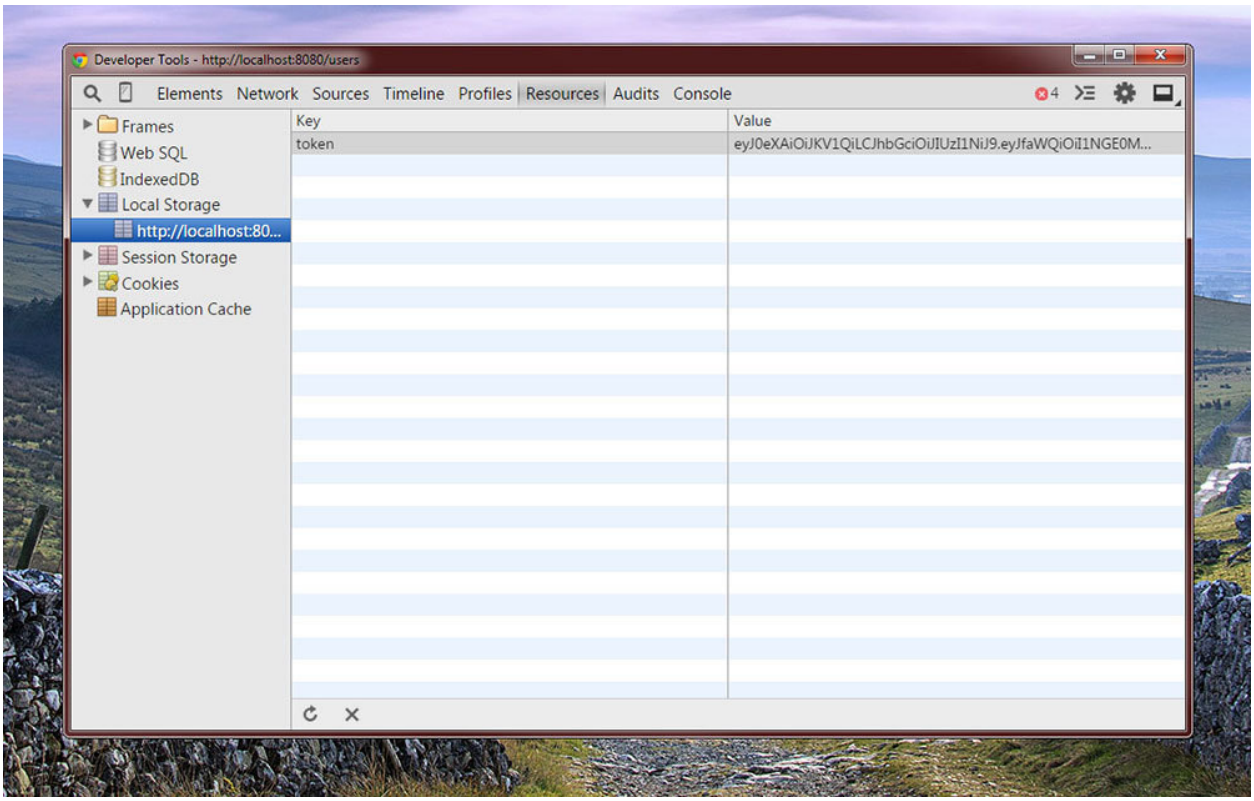
$location will be the module that we use to redirect users. This is how we redirect while still using the Angular routing mechanisms. This means that our user will be redirected without a page refresh.

$q is the module used to return promises. It will allow us to run functions asynchronously and return their values when they are done processing. In the previous chapter, we used `success()`, `error()`, and `then()` to act on a promise; we will be able to use those functions when returning a `$q`.

Let's go through each of these factories one by one so that we can see how our entire `authService` will work.

Auth Token Factory

We'll start with the `AuthTokenFactory` since that will be used by the other two factories. Essentially what we are doing is setting or getting data from our browser's local storage[103]. To check on what is in your local storage, just go into the Chrome Dev tools, click Resources, and check the Local Storage tab.



Local Storage

Here is the code for the `AuthTokenFactory`:

```
1  // ================================================
2  // factory for handling tokens
3  // inject $window to store token client-side
4  // ================================================
5  .factory('AuthToken', function($window) {
6
7    var authTokenFactory = {};
8
9    // get the token out of local storage
10   authTokenFactory.getToken = function() {
```

[103] http://diveintohtml5.info/storage.html

```
    return $window.localStorage.getItem('token');
  };

  // function to set token or clear token
  // if a token is passed, set the token
  // if there is no token, clear it from local storage
  authTokenFactory.setToken = function(token) {
    if (token)
      $window.localStorage.setItem('token', token);
    else
      $window.localStorage.removeItem('token');
  };

  return authTokenFactory;

})
```

We have just two functions here. `getToken()` and `setToken()`. `$window.localStorage` is the way that we can add, set or remove items from local storage.

Next up, let's create the main Auth factory that we will be using within our application's controllers.

Auth Factory

The main Auth factory will contain functions needed to log a user in, log a user out, check if a user is logged in, and get the user information.

Let's see how that factory will look:

```
// ================================================
// auth factory to login and get information
// inject $http for communicating with the API
// inject $q to return promise objects
// inject AuthToken to manage tokens
// ================================================
.factory('Auth', function($http, $q, AuthToken) {

  // create auth factory object
  var authFactory = {};

  // log a user in
  authFactory.login = function(username, password) {

```

```
15      // return the promise object and its data
16      return $http.post('/api/authenticate', {
17        username: username,
18        password: password
19      })
20        .success(function(data) {
21          AuthToken.setToken(data.token);
22          return data;
23        });
24    };
25
26    // log a user out by clearing the token
27    authFactory.logout = function() {
28      // clear the token
29      AuthToken.setToken();
30    };
31
32    // check if a user is logged in
33    // checks if there is a local token
34    authFactory.isLoggedIn = function() {
35      if (AuthToken.getToken())
36        return true;
37      else
38        return false;
39    };
40
41    // get the logged in user
42    authFactory.getUser = function() {
43      if (AuthToken.getToken())
44        return $http.get('/api/me');
45      else
46        return $q.reject({ message: 'User has no token.' });
47    };
48
49    // return auth factory object
50    return authFactory;
51
52  })
```

We have 4 functions that will either return a promise object (through `$http` or `$q`) or a simple `true/false`.

login will create an `$http.post()` request to the **/api/authenticate** endpoint on our Node API.

logout will simply use the `AuthToken` factory to clear the token.

isLoggedIn will return true or false depending on if the token exists in local storage.

getUser will create an `$http.get()` request to the **/me** API endpoint to get the logged in user's information.

Next, we'll create the interceptor factory.

AuthInterceptor Factory

The `AuthInterceptor` factory will be responsible for attaching the token to all HTTP requests coming from our frontend application.

```
// ================================================
// application configuration to integrate token into requests
// ================================================
.factory('AuthInterceptor', function($q, $location, AuthToken) {

  var interceptorFactory = {};

  // this will happen on all HTTP requests
  interceptorFactory.request = function(config) {

    // grab the token
    var token = AuthToken.getToken();

    // if the token exists, add it to the header as x-access-token
    if (token)
      config.headers['x-access-token'] = token;

    return config;
  };

  // happens on response errors
  interceptorFactory.responseError = function(response) {

    // if our server returns a 403 forbidden response
    if (response.status == 403) {
      AuthToken.setToken();
      $location.path('/login');
    }

    // return the errors from the server as a promise
```

```
    return $q.reject(response);
  };

  return interceptorFactory;

});
```

An interceptor in Angular will allow us to react to different HTTP request scenarios. When creating an interceptor, we have a few options available to us.

- **request** lets us intercept requests before they are sent
- **response** lets us change the response that we get back from a request
- **requestError** captures requests that have been cancelled
- **responseError** catches backend calls that fail. In this case, we will use it to catch 403 Forbidden errors if the token does not validate or does not exist.

If we receive a 403 error from our backend, we will be forced to redirect the user to the login page using `$location.path('/login')`. We will also clear any tokens in storage since those weren't good enough to validate our user.

Those are the three factories necessary for creating authentication in an Angular application. We are able to communicate with the backend API, get and set tokens client-side, and attach the token to all our requests.

The Entire Auth Service File (authService.js)

```
angular.module('authService', [])

// ===================================================
// auth factory to login and get information
// inject $http for communicating with the API
// inject $q to return promise objects
// inject AuthToken to manage tokens
// ===================================================
.factory('Auth', function($http, $q, AuthToken) {

  // create auth factory object
  var authFactory = {};

  // log a user in
  authFactory.login = function(username, password) {
```

```
16
17      // return the promise object and its data
18      return $http.post('/api/authenticate', {
19        username: username,
20        password: password
21      })
22        .success(function(data) {
23          AuthToken.setToken(data.token);
24          return data;
25        });
26    };
27
28    // log a user out by clearing the token
29    authFactory.logout = function() {
30      // clear the token
31      AuthToken.setToken();
32    };
33
34    // check if a user is logged in
35    // checks if there is a local token
36    authFactory.isLoggedIn = function() {
37      if (AuthToken.getToken())
38        return true;
39      else
40        return false;
41    };
42
43    // get the logged in user
44    authFactory.getUser = function() {
45      if (AuthToken.getToken())
46        return $http.get('/api/me');
47      else
48        return $q.reject({ message: 'User has no token.' });
49    };
50
51    // return auth factory object
52    return authFactory;
53
54  })
55
56  // ===================================================
57  // factory for handling tokens
```

```
58  // inject $window to store token client-side
59  // =================================================
60  .factory('AuthToken', function($window) {
61
62    var authTokenFactory = {};
63
64    // get the token out of local storage
65    authTokenFactory.getToken = function() {
66      return $window.localStorage.getItem('token');
67    };
68
69    // function to set token or clear token
70    // if a token is passed, set the token
71    // if there is no token, clear it from local storage
72    authTokenFactory.setToken = function(token) {
73      if (token)
74        $window.localStorage.setItem('token', token);
75      else
76        $window.localStorage.removeItem('token');
77    };
78
79    return authTokenFactory;
80
81  })
82
83  // =================================================
84  // application configuration to integrate token into requests
85  // =================================================
86  .factory('AuthInterceptor', function($q, $location, AuthToken) {
87
88    var interceptorFactory = {};
89
90    // this will happen on all HTTP requests
91    interceptorFactory.request = function(config) {
92
93      // grab the token
94      var token = AuthToken.getToken();
95
96      // if the token exists, add it to the header as x-access-token
97      if (token)
98        config.headers['x-access-token'] = token;
99
```

```
100     return config;
101   };
102
103   // happens on response errors
104   interceptorFactory.responseError = function(response) {
105
106     // if our server returns a 403 forbidden response
107     if (response.status == 403)
108       $location.path('/login');
109
110     // return the errors from the server as a promise
111     return $q.reject(response);
112   };
113
114   return interceptorFactory;
115
116 });
```

Conclusion

We now have our two great Angular services that we can use in our application. These two services will be the glue between the frontend Angular application and the backend Node application.

In addition to what we have just created from scratch, there are pre-built solutions on the market as well. A fully-featured Angular module would be the ng-token-auth[104] module. It provides the same options as above but adds some neat features like auth events.

Let's move forward and use these new files in our User CRM application. This is what we've been waiting for - A full MEAN stack application! All of our work in previous chapters has been building up to this point. Roll up your sleeves; we've got a lot of work to do in the next chapter.

[104] https://github.com/lynndylanhurley/ng-token-auth

MEAN App: Users Management CRM

Up to this point, we have created a Node API and used JSON web tokens to authenticate it. This has all been server-side code and now we will integrate it with the frontend using the information we just learned about Angular services.

The goal here is to start writing an Angular application and not have to edit the backend. Treating your applications like this can simulate similar situations where the backend API is built and maintained by someone other than yourself (like Facebook, Twitter, or GitHub APIs). We only have access to the frontend code.

Setting Up the Application

We will be using the foundation and application structure we created in Chapter 14. Since that was our Node API (w/ authentication), this will act as the perfect setup for building a full MEAN stack application.

Folder Structure

We will only be working with the frontend and Angular parts of the MEAN stack, so we will only need to work within the **public/** folder. That separation of frontend and backend allows us to narrow our focus and keep our development clean because we already know exactly where our code needs to go.

Here is the application structure.

```
- app/
----- models/
---------- user.js
----- routes/
---------- api.js
- public/
----- assets/
---------- css/
--------------- style.css
----- app/
---------- controllers/
--------------- mainCtrl.js
```

```
13 --------------- userCtrl.js
14 ---------- services/
15 --------------- authService.js
16 --------------- userService.js
17 ---------- app.routes.js
18 ---------- app.js
19 ---------- views/
20 --------------- pages/
21 -------------------- users/
22 ------------------------- all.html
23 ------------------------- single.html
24 --------------- login.html
25 --------------- home.html
26 ---------- index.html
27 - package.json
28 - config.js- server.js
```

Most of these files were already created in Chapter 14.

The `authService.js` will be the same file we created in the last chapter (Chapter 16) while `userService.js` will be the file created in Chapter 15.

The rest of the work will be focused on writing **controllers**, **routes**, and **views**.

Bringing In the Services

We'll be adding the services that we created last chapter. Go ahead and take the files you created and add them to:

- **public/app/services/authService.js**
- **public/app/services/userService.js**

Now that our services are part of our application, let's get the Angular groundwork ready and then we'll start using them to link the frontend to the backend.

Main Application

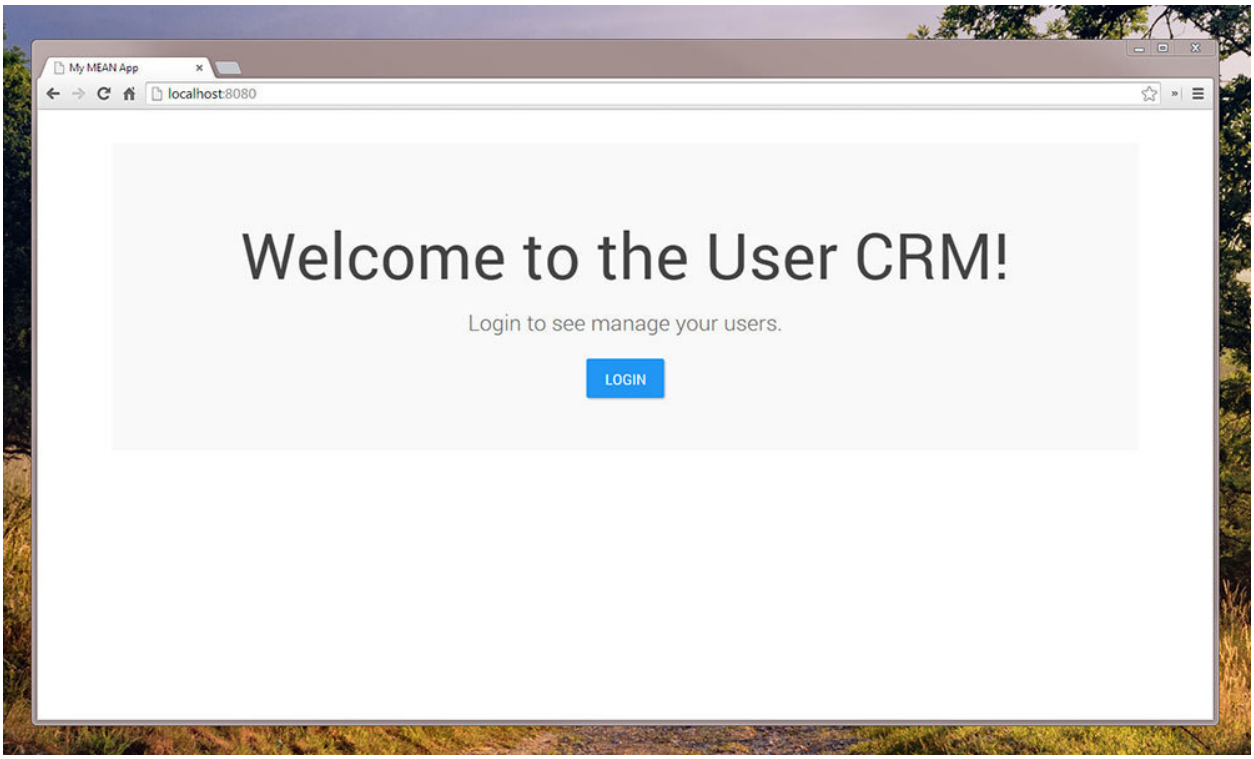
The foundation of our Angular application will require 5 files:

- public/app/controllers/mainCtrl.js
- public/app/app.js

- public/app/app.routes.js
- public/app/views/index.html
- public/app/views/pages/home.html

All of these files will work together to produce the home page of our application. Since the home page doesn't require authentication, we won't need to use the `authService.js` just yet.

This is what our home page will look like when our application is all done:



Home Page

Let's get started creating our files.

app/app.js

```
angular.module('userApp', [
  'ngAnimate',
  'app.routes',
  'authService',
  'mainCtrl',
  'userCtrl',
  'userService'
]);
```

This is a high level overview of our entire application. We are bringing in:

- **ngAnimate** to add animations to all of our Angular directives (specifically ngShow/ngHide)
- **app.routes** will be the routing for our application
- **authService** is the service file created in chapter 16
- **mainCtrl** will be a brand new controller we create that will encompass our main view
- **userCtrl** will have the controllers for all our user management pages
- **userService** is the service file created in chapter 15

Routes

We will start with our route file. For now, we are only adding the first route for our home page.

public/app.routes.js

```
angular.module('app.routes', ['ngRoute'])

.config(function($routeProvider, $locationProvider) {

  $routeProvider

    // home page route
    .when('/', {
      templateUrl : 'app/views/pages/home.html'
    });

  // get rid of the hash in the URL
  $locationProvider.html5Mode(true);

});
```

Let's quickly add the `home.html` file:

```
<div class="jumbotron text-center">
  <h1>Welcome to the User CRM!</h1>
</div>
```

This is enough to produce our homepage. Now let's move on to the main controller.

Controller

public/app/controllers/mainCtrl.js

```
angular.module('mainCtrl', [])

.controller('mainController', function($rootScope, $location, Auth) {

  var vm = this;

  // get info if a person is logged in
  vm.loggedIn = Auth.isLoggedIn();

  // check to see if a user is logged in on every request
  $rootScope.$on('$routeChangeStart', function() {
    vm.loggedIn = Auth.isLoggedIn();

    // get user information on route change
    Auth.getUser()
      .success(function(data) {
        vm.user = data;
      });
  });

  // function to handle login form
  vm.doLogin = function() {

    // call the Auth.login() function
    Auth.login(vm.loginData.username, vm.loginData.password)
      .success(function(data) {

        // if a user successfully logs in, redirect to users page
        $location.path('/users');
      });
  };

  // function to handle logging out
  vm.doLogout = function() {
    Auth.logout();
    // reset all user info
    vm.user = {};
    $location.path('/login');
  };

});
```

The `mainController` will contain some major functions for our application. Since this controller is applied to the overall layout of our application, it will be responsible for holding the logged in and logged out user information.

There are four main tasks that the `mainController` has that will be accomplished by accessing the `Auth` service:

Checking if a user is logged in We will check if a user is logged in using the `Auth.isLoggedIn()` function. This will check to see if there is a token in **localStorage**. We are also using a module we haven't used before called `$rootScope` to detect a route change and check if our user is still logged in. This means that every time we visit a different page, we will check our user's login state.

Getting user data Whenever the page route is changed, we will go and grab information for the current user. This way, we can display a message like `Hello Holly!`. This call uses the `Auth.getUser()` function, which hits the API endpoint `http://localhost:8080/api/me`.

Tip

Caching Service Calls

On every route change, we are going to grab the user data. This is to ensure that our user information is fresh, especially right after login. We won't want to make a call to the API however on every call, so there is a way to cache that information. This can be used for any `$http` calls and can be very useful in ensuring speed and efficiency in our application.

To cache the `getUser()` call, we will need to open up our `AuthService.js` file.

All we have to do here is add to the `$http.get()` call like so:

```
  // get the logged in user
  authFactory.getUser = function() {
    if (AuthToken.getToken())
      return $http.get('/api/me', { cache: true });
    else
      return $q.reject({ message: 'User has no token.' });
  };
```

The important part here is `{ cache: true }`. Now whenever the `Auth.getUser()` call is made, this will check if that information has already been cached. If it is in the cache, then it will return the cached information. If not, then it will make the API call.

Log a user in We will have a function to log a user in. This will authenticate the user with `username` and `password`. `vm.loginData.username` is bound to an input that we will create in our view.

Log a user out We will call `Auth.logout()`, which will delete the user's token in **localStorage** and any information stored in the mainController `vm.user` object. Then we will redirect the user to the home page since they will be unauthenticated and therefore won't have access to any other pages.

Home Page View

Just like in the Angular routing chapter, we will need a main file to be the overall layout for the site. Since Node.js is returning this `index.html` file to our users, this will be our layout file.

public/app/views/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>User CRM</title>

  <!-- FOR ANGULAR ROUTING -->
  <base href="/">

  <!-- CSS  -->
  <!-- load bootstrap from CDN and custom CSS -->
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootswatch/3.3.1/paper/\
bootstrap.min.css">
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/animate.css/3.1.\
```

```
1/animate.min.css">
  <link rel="stylesheet" href="assets/css/style.css">

  <!-- JS -->
  <!-- load angular and angular-route via CDN -->
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular.min.js"><\
/script>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-route.js"\
></script>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-animate.j\
s"></script>

  <!-- controllers -->
  <script src="app/controllers/mainCtrl.js"></script>
  <script src="app/controllers/userCtrl.js"></script>

  <!-- services -->
  <script src="app/services/authService.js"></script>
  <script src="app/services/userService.js"></script>

  <!-- main Angular app files -->
  <script src="app/app.routes.js"></script>
  <script src="app/app.js"></script>
</head>
<body ng-app="userApp" ng-controller="mainController as main">

<!-- NAVBAR -->
<header>

  <div class="navbar navbar-inverse" ng-if="main.loggedIn">
  <div class="container">
    <div class="navbar-header">
      <a href="/" class="navbar-brand"><span class="glyphicon glyphicon-fire tex\
t-danger"></span> User CRM</a>
    </div>
    <ul class="nav navbar-nav">
      <li><a href="/users"><span class="glyphicon glyphicon-user"></span> Users<\
/a></li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li ng-if="!main.loggedIn"><a href="/login">Login</a></li>
      <li ng-if="main.loggedIn" class="navbar-text">Hello {{ main.user.name }}!<\
```

```
/li>
      <li ng-if="main.loggedIn"><a href="#" ng-click="main.doLogout()">Logout</a\
></li>
    </ul>
  </div>
  </div>

</header>

<main class="container">
  <!-- ANGULAR VIEWS -->
  <div ng-view></div>
</main>

</body>
</html>
```

We are applying our Angular application to the `<body>` tag and using the controller as syntax (`mainController as main`). We're also loading the `userCtrl.js` that we'll build soon. Our two services we created in the last two chapters (`authService` and `userService`) are pulled in here as well.

All of our files are loaded for our CSS and our Angular application.

We are making use of the variable `vm.loggedIn` to show if a user is logged in or not. The `ngIf` Angular directives will help us show or hide the navbar, since a user that isn't logged in should not be able to see it.

We have also created a logout link that will use the `doLogout()` function we created in our `mainController`. Right next to the logout button, we have also added a message to say hello to our logged in user. That data came from our `mainController` when we called `Auth.getUser()` and bound that information to `vm.user`.

User Controller

We've added the `userCtrl.js` here in our HTML file, but we haven't created it yet. Let's create the file here with some minimal data. We'll fill this out more soon when we get to our user pages.

```
angular.module('userCtrl', ['userService'])

  .controller('userController', function(User) {

    var vm = this;

    // more stuff to come soon

  });
```

We are injecting the `userService` here because we need the `User` factory we created earlier. We'll deal more with the `userController` soon when we need to start grabbing our users and displaying them.

Tip

ngIf vs. ngShow/ngHide

The biggest difference in using ngIf over ngShow or ngHide is that ngIf **will not** show that HTML element in view source or inspect element.

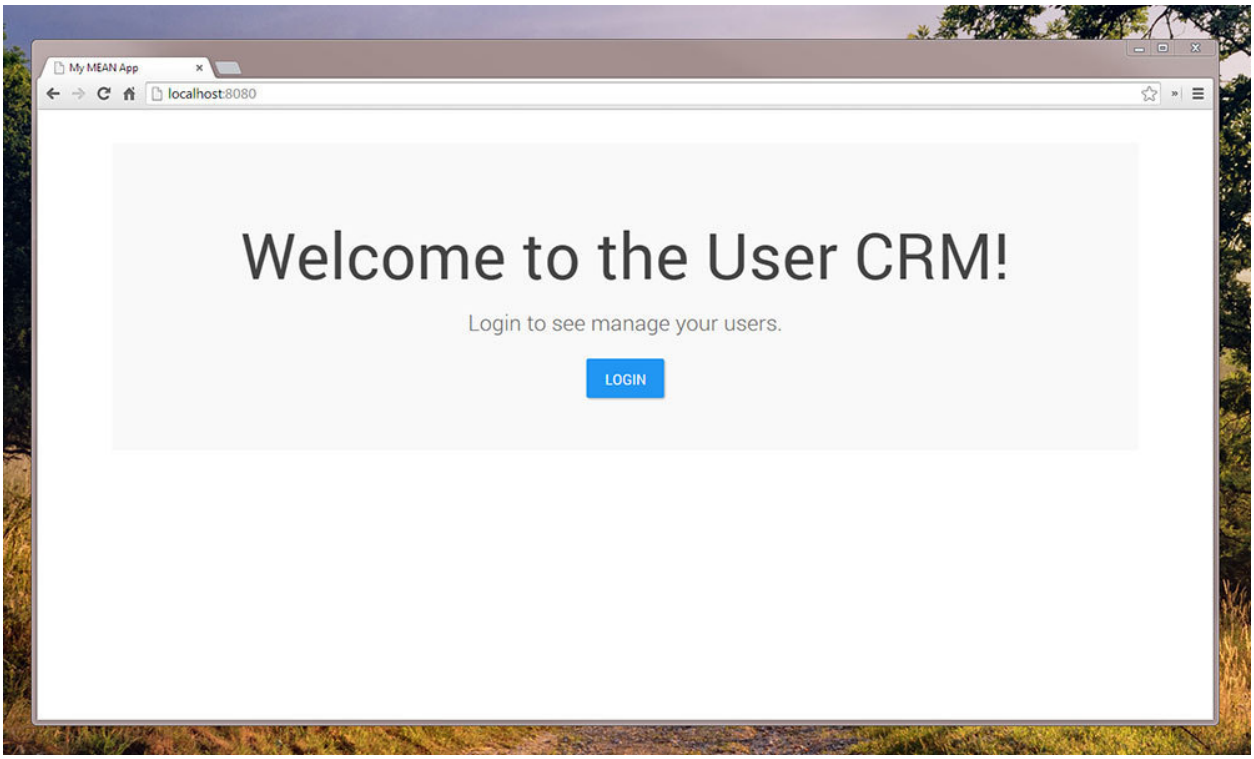
This is helpful if you want to hide information that only authenticated users should see.

To finish off the home page, let's add some custom CSS stylings to our CSS file.

public/assets/css/style.css

```
main          {
  padding-top:30px;
}

/* HEADER
==================== */
header .navbar    {
  border-radius:0;
}
```

Since we aren't logged in, we won't see the navbar and our users will just see the welcome text and the link to the home page.



Home Page

Let's move onto creating the login page now so that our users will actually be able to login.

Login Page

We will need to do two things to create our login page:

- Create the login route

- Create the login view

We'll add the login route under the home page route to our `app.routes.js` file:

```
  // route for the home page
  .when('/', {
    templateUrl : 'app/views/pages/home.html'
  })

  // login page
  .when('/login', {
    templateUrl : 'app/views/pages/login.html',
      controller  : 'mainController',
      controllerAs: 'login'
  });
```

We are applying the `mainController` to this route in addition to the overall site except we are naming the `controllerAs` attribute `login`. This allows us to access the `mainControllers` functions using `login.doLogin()`.

Let's create the login view now:

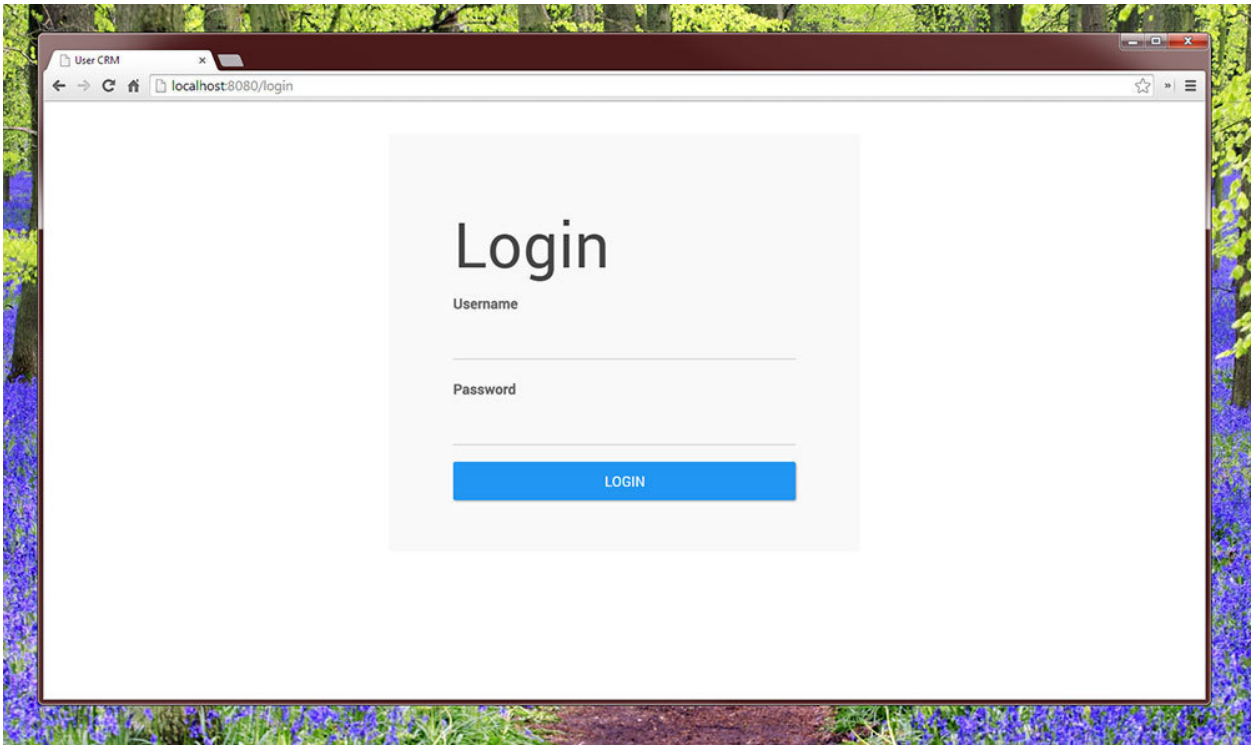
app/views/pages/login.html

```
<div class="row col-sm-6 col-sm-offset-3">

<div class="jumbotron">
  <h1>Login</h1>

  <form ng-submit="login.doLogin()">

    <!-- USERNAME INPUT -->
    <div class="form-group">
      <label>Username</label>
      <input type="text" class="form-control"
        ng-model="login.loginData.username">
    </div>

    <!-- PASSWORD INPUT -->
    <div class="form-group">
      <label>Password</label>
      <input type="password" class="form-control"
        ng-model="login.loginData.password">
```

```
20      </div>
21
22      <!-- LOGIN BUTTON -->
23      <button type="submit" class="btn btn-block btn-primary">
24        <span>Login</span>
25      </button>
26
27    </form>
28  </div>
29
30  </div>
```

We have added line breaks in the `<input>` tags to make them easier to read.

We have created a normal login form and bound our Angular functions and variables using `ng-submit` and `ng-model`. Our user form will look clean and simple:



Login Page

If we go ahead and enter our credentials (`chris`, `supersecret`), then we will be redirected to the `/users` page. That behavior was set in the `doLogin()` function we created in `mainController`. The users page doesn't exist yet since we haven't created the route or views.

Let's add a few things to our login page like a processing icon and handling error messages.

Adding a Processing Icon

It makes sense to show a processing indicator when our form is processing. We can do this with two steps:

1. Creating a variable called `processing`
2. Adding spinning icons in our views

Whenever a user clicks **Login**, we want the view to show a processing icon. When the form is done processing, we want the processing button to disappear.

We will show and hide the processing icon using `ngIf` and bind it to a variable called `processing`. Let's update the `doLogin()` function in our `mainController` file to add this variable.

```
// function to handle login form
vm.doLogin = function() {
  vm.processing = true;

  Auth.login(vm.loginData.username, vm.loginData.password)
    .success(function(data) {
      vm.processing = false;

      // if a user successfully logs in, redirect to users page
      $location.path('/users');
    });
};
```

When a user clicks **Login**, we have set the `processing` variable to **true**. When the function is done processing, we will set the variable to **false**.

Now all well have left to do is create the processing icon and show it.

In our `login.html` file, let's add the icon to the Login button.

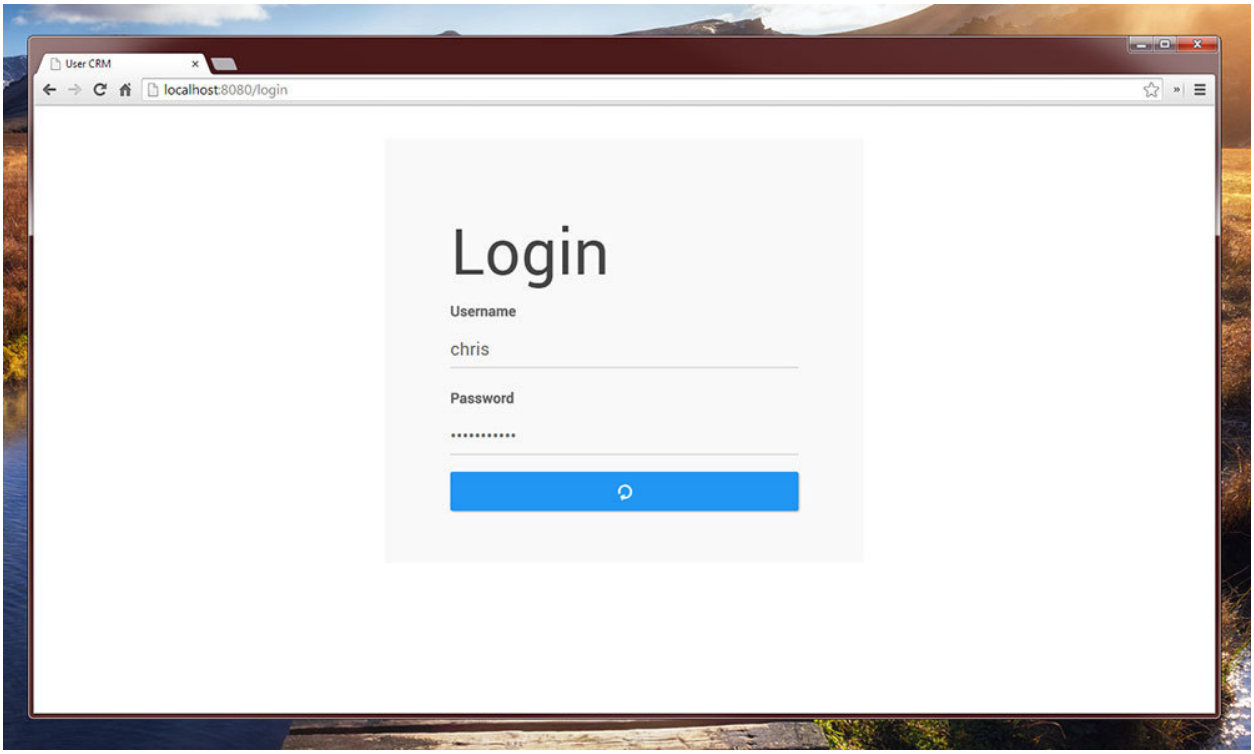
```
<button type="submit" class="btn btn-block btn-primary">

  <span ng-if="!login.processing">Login</span>

  <span ng-if="login.processing" class="spinner"><span class="glyphicon glyphico\
n-repeat"></span></span>

</button>
```

If the variable is true, we will show the spinner. Otherwise, our button will always say **Login**.

Just a little CSS to handle the spinning animation and we should be good. Add the following to `style.css`:

```
/* ANIMATIONS
====================== */
.spinner {
  animation:spin 1s infinite;
  -webkit-animation:spin 1s infinite;
  -moz-animation:spin 1s infinite;
}

@keyframes spin {
  from  { transform:rotate(0deg); }
  to    { transform:rotate(360deg); }
}

@-webkit-keyframes spin {
  from  { -webkit-transform:rotate(0deg); }
  to    { -webkit-transform:rotate(360deg); }
}

@-moz-keyframes spin {
  from  { -moz-transform:rotate(0deg); }
  to    { -moz-transform:rotate(360deg); }
}
```

Whatever element we add this `spinner` class to will start spinning. If we click the Login button now, we will see the spinner.



Login Spinner

This technique can be used throughout our entire application.

Adding a Login Error Message

If login is not successful, we will want to let our users know. Since our API already gives back error messages, all we have to do is display them.

Inside of our login function, we will check for the `success` variable that was returned to us by the API.

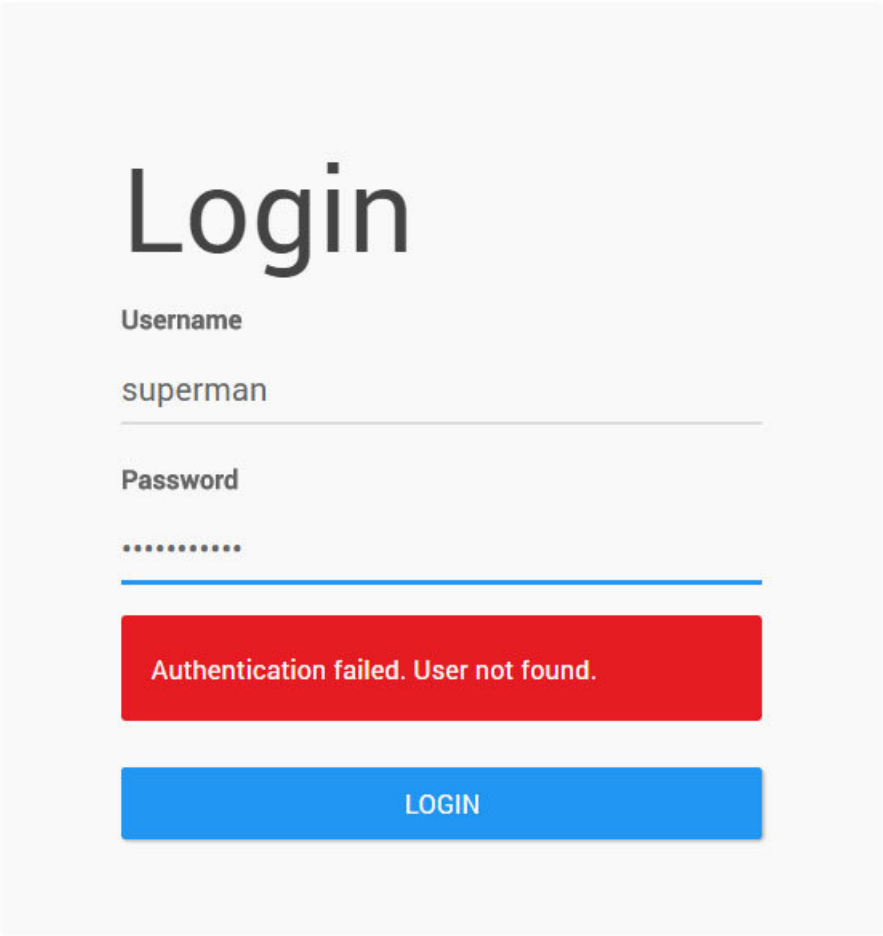
```
1   vm.doLogin = function() {
2     vm.processing = true;
3
4     // clear the error
5     vm.error = '';
6
7     Auth.login(vm.loginData.username, vm.loginData.password)
8       .success(function(data) {
9         ...
10
11        // if a user successfully logs in, redirect to users page
```

```
12          if (data.success)
13            $location.path('/users');
14          else
15            vm.error = data.message;
16        });
```

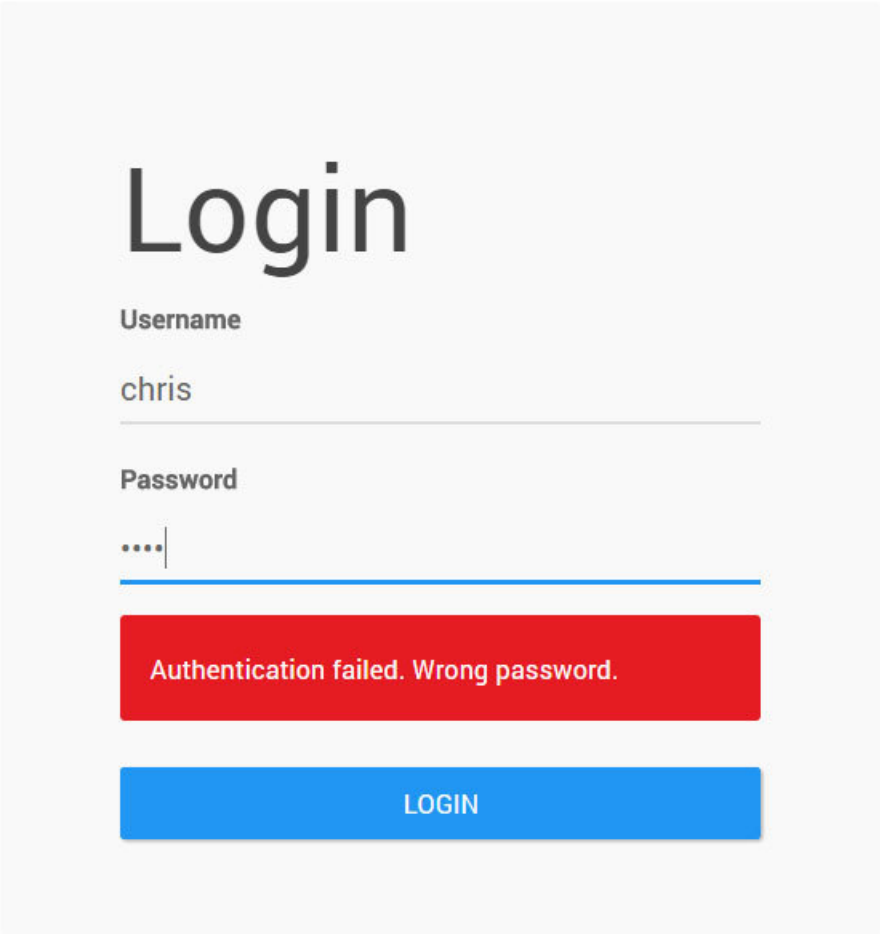
Every time we click login, the error message will be cleared so we don't see an out-of-date error message. In our view, we just have to check for that error. Above the login button, add the following:

```
<div class="alert alert-danger" ng-if="login.error">
  {{ login.error }}
</div>
```

If the error variable exists, then we will display this div.



User Doesn't Exist



Wrong Password

With our foundation of the site and login ready to go, we will now want to handle a little more authentication and then create the pages that will show our users.

Authentication

We have already used the `Auth` factory in our `mainController`. We've called all the functions including `isLoggedIn`, `login`, `logout`, and `getUser`.

Our user can login and their token is stored in localStorage. If you check your browser's local storage, you will be able to see the token. Before we can get our list of users for the users pages, we must attach this token to every request. This is where the `AuthInterceptor` we created in our `authService` comes in handy.

We will use this in the main `app.js` file. Let's look at how it is applied:

```
angular.module('userApp', [
  'ngAnimate',
  'app.routes',
  'authService',
  'mainCtrl',
  'userCtrl',
  'userService'
])

// application configuration to integrate token into requests
.config(function($httpProvider) {

  // attach our auth interceptor to the http requests
  $httpProvider.interceptors.push('AuthInterceptor');

});
```

Just like we create `.controllers` and `.factorys` on our angular.modules, we can use `.config` to add extra configurations to our application. In this case, we are adding the `AuthInterceptor` to the $httpProvider.

The $httpProvider will attach the token to each request. We'll see exactly where this happens when we request user information in the next section.

Now that we are authenticated, we will be redirected to a page that lists all users in the database.

User Pages

To create each user page, we will need a new **controller**, **route**, and **view**. Let's start by showing all the users on a single page.

All Users

We will need to go into our `userCtrl.js` file and add a controller for this page.

All Users Controller

```
// start our angular module and inject userService
angular.module('userCtrl', ['userService'])

// user controller for the main page
// inject the User factory
.controller('userController', function(User) {

  var vm = this;

  // set a processing variable to show loading things
  vm.processing = true;

  // grab all the users at page load
  User.all()
    .success(function(data) {

      // when all the users come back, remove the processing variable
      vm.processing = false;

      // bind the users that come back to vm.users
      vm.users = data;
    });

})
```

When the user page loads, we will use the User factory to go and grab all of our users. Those users will be bound to the `vm.users` variable so that we can use them in our view. Like the login page, we are using the `processing` variable to show loading icons.

All Users View (pages/users/all.html)

Our main view will contain a few different components.

- A header with a button to create a new user
- A message that says "Loading Users..."
- A table of our users
- Button to edit a user

```
<div class="page-header">
  <h1>
    Users
    <a href="/users/create" class="btn btn-default">
      <span class="glyphicon glyphicon-plus"></span>
      New User
    </a>
  </h1>

</div>

<!-- LOADING MESSAGE -->
<div class="jumbotron text-center" ng-show="user.processing">
  <span class="glyphicon glyphicon-repeat spinner"></span>
  <p>Loading Users...</p>
</div>

<table class="table table-bordered table-striped" ng-show="user.users">
  <thead>
    <tr>
      <th>_id</th>
      <th>Name</th>
      <th>Username</th>
      <th class="col-sm-2"></th>
    </tr>
  </thead>
  <tbody>

    <!-- LOOP OVER THE USERS -->
    <tr ng-repeat="person in user.users">
      <td>{{ person._id }}</td>
      <td>{{ person.name }}</td>
      <td>{{ person.username }}</td>
      <td class="col-sm-2">
        <a ng-href="/users/{{ person._id }}" class="btn btn-danger">Edit</a>
      </td>
    </tr>

  </tbody>
</table>
```

This will show all of our users in a table. We will only show the loading message if the `processing` variable is true. We will also only show the user table if the `users` object has users.

We will use the `ng-repeat` directive as a table row to loop over all our users. There is also an edit button that will link to the edit user page and pass in the user's id into the URL.

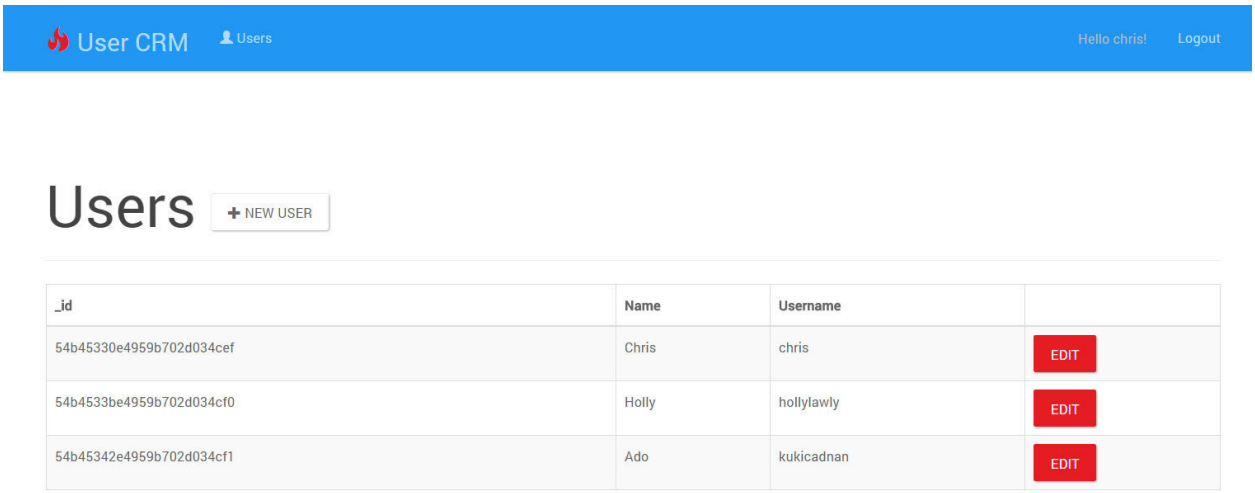
The last part needed to see this users page is to create the route.

All Users Route

In our `app.routes.js` file, add the following below the `login` route:

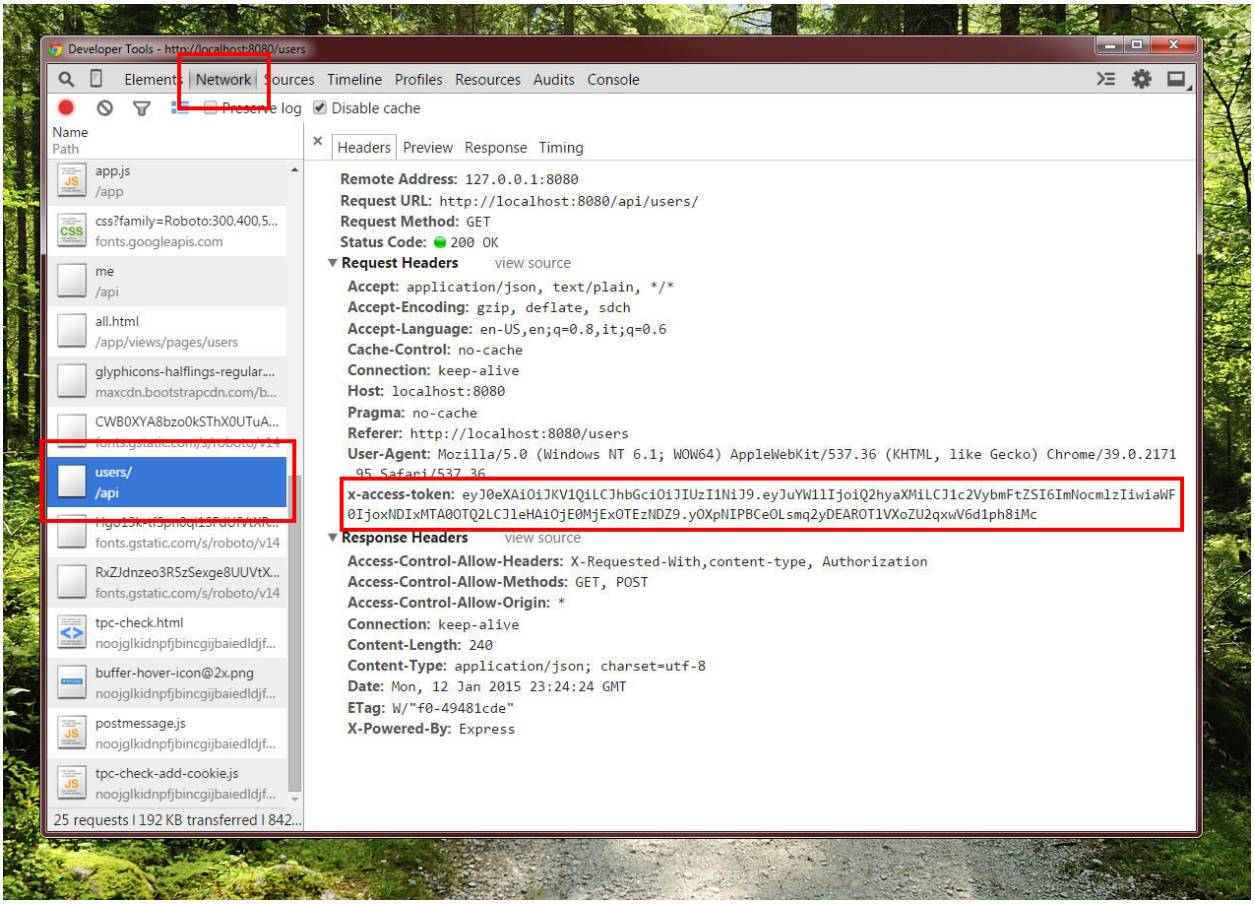
```
// show all users
.when('/users', {
  templateUrl: 'app/views/pages/users/all.html',
  controller: 'userController',
  controllerAs: 'user'
});
```

That will use the controller and view we just created. Visit `http://localhost:8080/users` in your browser and you will see all the work we've done come together.



All Users

Another thing to note is that if we go into our Chrome network tools and look at the request to get all the users, we will see that Angular did in fact attach our token.



Access Token

There is one more thing left to do on this page and that is to provide the functionality to delete a user.

Delete a User

We will need two things to add delete functionality.

1. A function in our controller
2. A button in our view

Let's create the function first. In our `userCtrl.js`, add the function:

```
// function to delete a user
vm.deleteUser = function(id) {
  vm.processing = true;

  // accepts the user id as a parameter
  User.delete(id)
    .success(function(data) {

      // get all users to update the table
      // you can also set up your api
      // to return the list of users with the delete call
      User.all()
        .success(function(data) {
          vm.processing = false;
          vm.users = data;
        });

    });
};
```

This `deleteUser` function will call the delete function in our `User` factory. When that call is successful, we will make a call to grab all our users and then update the `users` object, which in turn will update our table.

All that's needed now is a delete button in the view. In your user table in `views/pages/users/all.html`, add the following right next to the edit button:

```
<a href="#"
  ng-click="user.deleteUser(person._id)"
  class="btn btn-primary">Delete</a>
```

We are passing in the `person._id` and we'll call the `deleteUser` function when this button is clicked thanks to the `ngClick` Angular directive.

Now when we click delete, the user will be deleted and our table will be updated. Next up, let's make the **create a user** components.

Create a User

Just like the list all users page, we will need a **controller**, **route**, and **view** for the create user page.

Create User Controller

We will add a controller to the `userCtrl.js` file beneath the first controller we created. We'll call this one `userCreateController`.

```
// controller applied to user creation page
.controller('userCreateController', function(User) {

  var vm = this;

  // variable to hide/show elements of the view
  // differentiates between create or edit pages
  vm.type = 'create';

  // function to create a user
  vm.saveUser = function() {
    vm.processing = true;

    // clear the message
    vm.message = '';

    // use the create function in the userService
    User.create(vm.userData)
      .success(function(data) {
        vm.processing = false;

        // clear the form
        vm.userData = {};
        vm.message = data.message;
      });

  };

})
```

This is a pretty standard process to us by now. We have a function called `saveUser` that will be used in a form in our view. We are also calling the `User.create()` function in our `userService`. After a user is created, we will show a message and clear the form so that we will be able to enter a new user.

We are passing in the entire `userData` object into the create user function which includes:

```
// all info is bound to our form using ng-model
{
  name: "Holly",
  username: "hollylawly",
  password: "supersecret"
}
```

The other addition here is the `type` variable. This will be used in our view. Since we are using the same file (`views/pages/users/single.html`) for both the creation and the editing pages, we will need a way to differentiate between the two. Sure you could create two separate view files for these pages, but there will only really be 2 minor differences in the view. We are eliminating some repeated code by doing it this way.

Create User Route

```
// form to create a new user
// same view as edit page
.when('/users/create', {
  templateUrl: 'app/views/pages/users/single.html',
  controller: 'userCreateController',
  controllerAs: 'user'
})
```

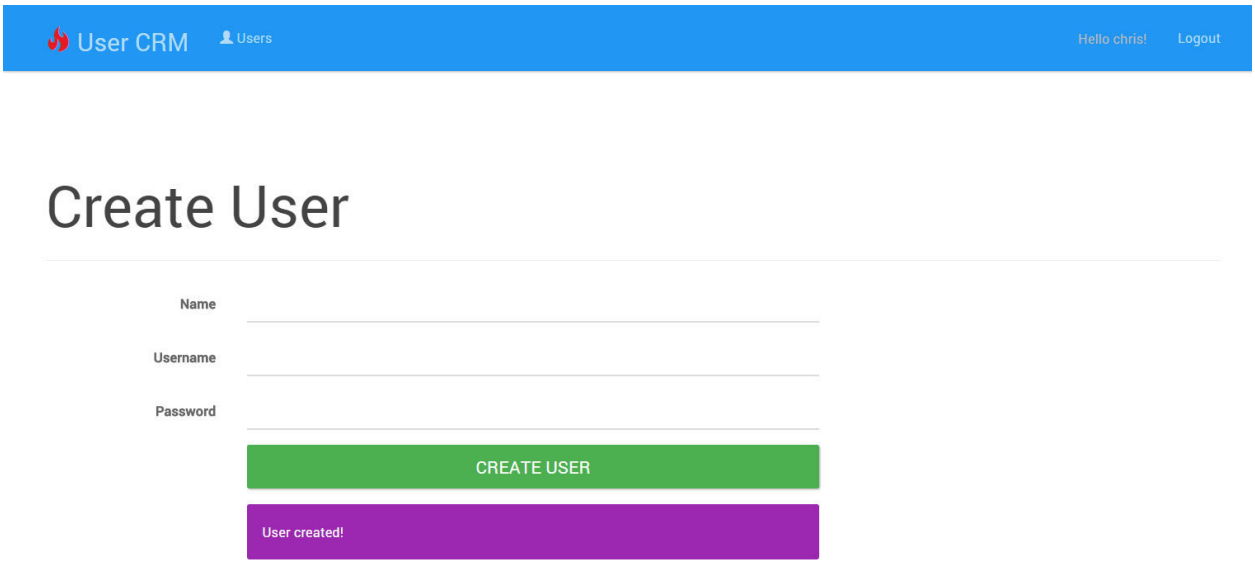
Create User View (pages/users/single.html)

```
<div class="page-header">
  <h1 ng-if="user.type == 'create'">Create User</h1>
  <h1 ng-if="user.type == 'edit'">Edit User</h1>
</div>

<form class="form-horizontal" ng-submit="user.saveUser()">

  <div class="form-group">
    <label class="col-sm-2 control-label">Name</label>
    <div class="col-sm-6">
      <input type="text"
        class="form-control"
        ng-model="user.userData.name">
    </div>
  </div>

  <div class="form-group">
```

```
18      <label class="col-sm-2 control-label">Username</label>
19      <div class="col-sm-6">
20        <input type="text"
21          class="form-control"
22          ng-model="user.userData.username">
23      </div>
24    </div>
25
26    <div class="form-group">
27      <label class="col-sm-2 control-label">Password</label>
28      <div class="col-sm-6">
29        <input type="password"
30          class="form-control"
31          ng-model="user.userData.password">
32      </div>
33    </div>
34
35    <div class="form-group">
36    <div class="col-sm-offset-2 col-sm-6">
37      <button type="submit"
38        class="btn btn-success btn-lg btn-block"
39        ng-if="user.type == 'create'">Create User</button>
40      <button type="submit"
41        class="btn btn-success btn-lg btn-block"
42        ng-if="user.type == 'edit'">Update User</button>
43    </div>
44    </div>
45
46  </form>
47
48  <div class="row show-hide-message" ng-show="user.message">
49    <div class="col-sm-6 col-sm-offset-2">
50
51      <div class="alert alert-info">
52        {{ user.message }}
53      </div>
54
55    </div>
56  </div>
```

We have our basic form here with inputs bound using `ngModel`, the form being submitted using `ngSubmit`, and our error message being shown using `ngShow`.

The thing to notice here is that we are checking for that `type` variable and showing **Create User** vs. **Edit User** and the button of the form will show **Create User** or **Update User**.



Create User

When we add information into our form and click **Create User**, the user will be created and the form will be cleared. Our message will also show up.

Edit a User

Let's run through the edit user page. We will use the same view as the create user page. The only differences are that:

- we have to pass a parameter into the URL (/users/user_id)
- we have to get the users information on page load
- we have to bind that information to our form

Edit User Controller

Since we are being passed the user ID through the URL, we will need to inject Angular's `$routeParams` module to get URL parameters.

We will know what the specific parameter is called based on what we name it in our routes file. In this case, we are grabbing `$routeParams.user_id`.

```
// controller applied to user edit page
.controller('userEditController', function($routeParams, User) {

  var vm = this;

  // variable to hide/show elements of the view
  // differentiates between create or edit pages
  vm.type = 'edit';

  // get the user data for the user you want to edit
  // $routeParams is the way we grab data from the URL
  User.get($routeParams.user_id)
    .success(function(data) {
      vm.userData = data;
    });

  // function to save the user
  vm.saveUser = function() {
    vm.processing = true;
    vm.message = '';

    // call the userService function to update
    User.update($routeParams.user_id, vm.userData)
      .success(function(data) {
        vm.processing = false;

        // clear the form
        vm.userData = {};

        // bind the message from our API to vm.message
        vm.message = data.message;
      });
  };

});
```

When the edit user page loads, we will go and grab that user's data by using the `User.get()` function. This will hit our API and grab the user info. We will then bind that object to `vm.userData`.

Since `userData` is the object that we used in our **views/pages/users/single.html**, our form will automatically populate with this data!

Edit User Route

This is where we define the parameter name. We will pass in the `:user_id` here and that lets `$routeParams` know that `$routeParams.user_id` exists in the controller we just made.

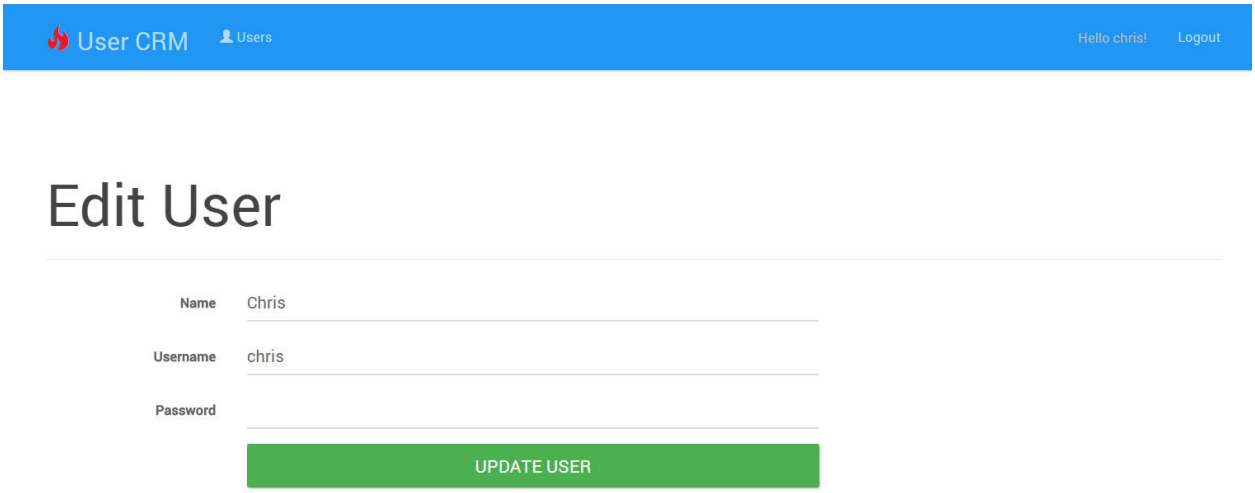
```
// page to edit a user
.when('/users/:user_id', {
  templateUrl: 'app/views/pages/users/single.html',
  controller: 'userEditController',
  controllerAs: 'user'
});
```

Edit User View (pages/users/single.html)

```
<div class="page-header">
  <h1 ng-if="user.type == 'create'">Create User</h1>
  <h1 ng-if="user.type == 'edit'">Edit User</h1>
</div>

<form class="form-horizontal" ng-submit="user.saveUser()">

  <div class="form-group">
    <label class="col-sm-2 control-label">Name</label>
    <div class="col-sm-6">
      <input type="text"
        class="form-control"
        ng-model="user.userData.name">
    </div>
  </div>

  <div class="form-group">
    <label class="col-sm-2 control-label">Username</label>
    <div class="col-sm-6">
      <input type="text"
        class="form-control"
        ng-model="user.userData.username">
    </div>
  </div>

  <div class="form-group">
    <label class="col-sm-2 control-label">Password</label>
    <div class="col-sm-6">
      <input type="password"
```

```
30          class="form-control"
31          ng-model="user.userData.password">
32      </div>
33    </div>
34
35    <div class="form-group">
36    <div class="col-sm-offset-2 col-sm-6">
37      <button type="submit"
38        class="btn btn-success btn-lg btn-block"
39        ng-if="user.type == 'create'">Create User</button>
40      <button type="submit"
41        class="btn btn-success btn-lg btn-block"
42        ng-if="user.type == 'edit'">Update User</button>
43    </div>
44    </div>
45
46  </form>
47
48  <div class="row show-hide-message" ng-show="user.message">
49    <div class="col-sm-6 col-sm-offset-2">
50
51      <div class="alert alert-info">
52        {{ user.message }}
53      </div>
54
55    </div>
56  </div>
```

This is the exact same view as the create users page. What is neat here is that since our inputs are data-bound to the `userData` object and we made the call to grab data in our controller, the user's information will automatically show in our form.



Edit User

Our password of course won't be shown for security purposes since our API does not return that data.

Animating the Message

Since we already pulled in animate.css[105] via CDN in our `index.html` file, we can use its classes to animate our Angular directives.

We just have to add some custom CSS. We'll use the animate.css class to have our message zoom in.

```
/* NGANIMATE
====================== */

/* show and hide */
.show-hide-message.ng-hide-remove {
  -webkit-animation:zoomIn 0.3s both ease;
  -moz-animation:zoomIn 0.3s both ease;
  animation:zoomIn 0.3s both ease;
}
```

To add animations to the `ngShow` and `ngHide` directives, the classes are `.ng-hide-remove` and `.ng-hide-add`.

Our message will now fly in using this CSS animation.

[105] http://daneden.github.io/animate.css/

Conclusion

We now have all the pages done and can handle CRUD on our users. By applying the Node API through the use of Angular services, we are able to create an entire frontend application!

These concepts can be applied to more than just users and you can duplicate these across many types of resources to create a larger application. Hopefully seeing how an entire MEAN application comes together from the backend to the frontend will help understand many things including:

- the separation of backend and frontend
- how using an API can help speed up workflow
- the mechanics of using an API
- creating an Angular application that uses services
- handling frontend Angular authentication
- and much more…

Recap of the Process

Although we have only dealt with users here, this is the foundation for how you can add components to your site. Let's say you wanted to add a resource to your site like **Articles**. You will want to perform the same CRUD options on this resource.

Here are the main steps for creating this new component:

Node.js Side

1. Create the Mongoose Model
2. Create the API endpoints as routes

AngularJS Side

1. Create the service to communicate with the API
2. Create controllers for all the different pages (all, create, edit)
3. Set up your views and routes and assign controllers
4. Polish your application with loading icons, animations, and more

The steps above recap the steps we have taken throughout this book. It doesn't seem like much when broken down into 6 bullet points, but there is a lot of knowledge necessary to execute all those points.

Tip

A More Advanced Approach (Components)

As you build more Angular applications, you will see a pattern arise; your site will start to be separated by sections. For instance, we have a user section and could move forward to create an article section, music, and whatever other resource.

Lumping all those controllers into the **controllers/** folder will become tedious since changing something for our users will require us to go into `controllers/userCtrl.js`, `services/userService.js`, `app.routes.js`, and any `views/pages/users/` views that correspond to that.

We can separate our site out even further so that all the user parts are encompassed into a **user component**. We can then inject the controllers, services, and routes into one main user Angular module and inject that into our entire applications parent module.

To better understand this concept, here's how the directory structure would look.

Components Public Folder Structure

```
- public/
----- assets/
---------- css/
--------------- style.css
----- app/
---------- shared/ // reusable components
--------------- sidebar/
------------------- sidebarView.html
------------------- sidebarController.html
------------------- sidebarService.html
---------- components/
--------------- users/
------------------- userCtrl.js
------------------- userService.js
------------------- userView.html
--------------- articles/
------------------- articleCtrl.js
------------------- articleService.js
------------------- articleView.html
```

```
---------- app.js
---------- app.routes.js
---------- index.html
```

Again, this is a more advanced approach for larger applications. For the purposes of our demos, the structure we've been using works. Just keep it in mind because as our applications grow, it might be worth looking into this structure.

Practice makes perfect and the more MEAN applications that you create the more these concepts will solidify in your mind.

Next Up

We have our MEAN stack application done, but there is still so much more to learn. The next thing we need to do is deploy our site so that other people are able to view it online. After all, what good is all this work if we can't show it off? Let's look at how we can deploy our awesome new site to the web.

Deploying MEAN Applications

Up to this point, we have been working locally on our computers when building all these applications. We have been creating our files, starting our server courtesy of Node and Express, and testing on `http://localhost:8080`.

The great things we create won't be visible to the rest of the world! We'll need to deploy our applications to some hosting service so that we can access it from a website.

There are many different ways to deploy sites to an online server. For our purposes, it is best to look for an online host that specializes in hosting Node applications.

Currently, the most popular server configuration is the LAMP stack (Linux, Apache, MySQL, and PHP). Since we are using Node instead of Apache, MongoDB instead of MySQL, and Node instead of PHP, it makes sense to look for a host that specializes in this sort of thing.

Great Node Hosts

Each of these hosts provides great features and are good choices for deploying your applications.

Modulus[106] - We've been using Modulus to host MongoDB databases in some of our chapters. They also provide great Node hosting and feature really good support.

Digital Ocean[107] - Digital Ocean is a VPS (virtual private server) provider so you are tasked with configuring and setting up your own server. Digital Ocean offers SSH access and plans that start at $5/mo. They also provide server images to install a Node environment with just a click of a button.

Heroku[108] - Our personal favorite for hosting Node applications. Deploying Heroku applications is extremely easy and the command line tools they provide are very simple to get started with. They also let you host smaller applications for free, though there are limits to the amount of traffic they will support.

Honorable Mention

These hosts have great reputations, but we haven't used them personally:

- Joyent[109] - The creator's of Node offer great cloud computing options with scalable pricing.
- Nodejitsu[110] - An easy to use hosting solution created by some of the earliest contributors to Node.

[106] https://modulus.io/
[107] https://www.digitalocean.com/
[108] https://www.heroku.com/
[109] https://www.joyent.com/
[110] https://www.nodejitsu.com/

Heroku provides a great balance between price, features, support, uptime, and tools available. We'll be looking at how to deploy a MEAN stack application to Heroku. This process is similar to other hosts and these techniques can be used across most of them.

Deploying to Heroku

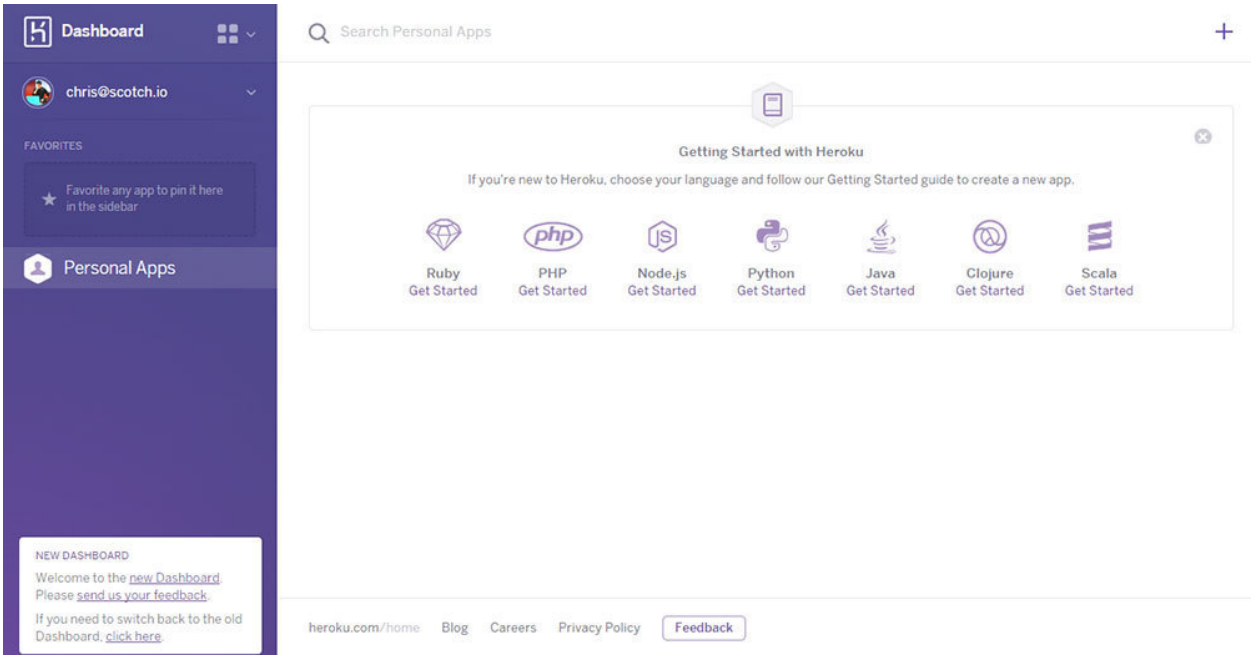
Heroku is known for making server configurations easy and painless. It allows us to build faster and focus on the details of our application rather than trying to configure our own servers.

We'll be deploying the full MEAN stack application that we have been working on throughout the book - our User CRM. The deployment is a fairly pain-free process and we'll have our full application on the web in mere minutes.

Let's get started.

Create a Heroku Account

Go ahead and go to Heroku.com[111] and create your free account. As you can see, the dashboard is incredibly simple and user friendly. It gives us this great **Getting Started with Heroku** dialog where we can find the instructions for each type of app you can deploy.



Heroku Dashboard

[111] https://heroku.com

We'll be walking through Heroku's Node deployment instructions[112]. You can follow along with this book and reference that link in the future since they provide a large amount of scenarios for deployment.

Tools Needed

You'll need a few things to start your deployment:

- Node and npm[113] (pretty sure you already have this one installed)
- Heroku Toolbelt[114]
- Git[115]

Git Repository

First, we will need to turn our application into a git repository. First we are going to create a local git repository and then have a git remote pointed at Heroku. We will then have the ability to push straight to Heroku.

If you already have been using a git repository for your projects and have been working on GitHub or BitBucket, awesome! You already have a git repository and can skip this step.

To create your local git repository, make sure you have git[116] installed and have access to the `git` commands in your console.

`cd` into your project and type the following to create a git repository:

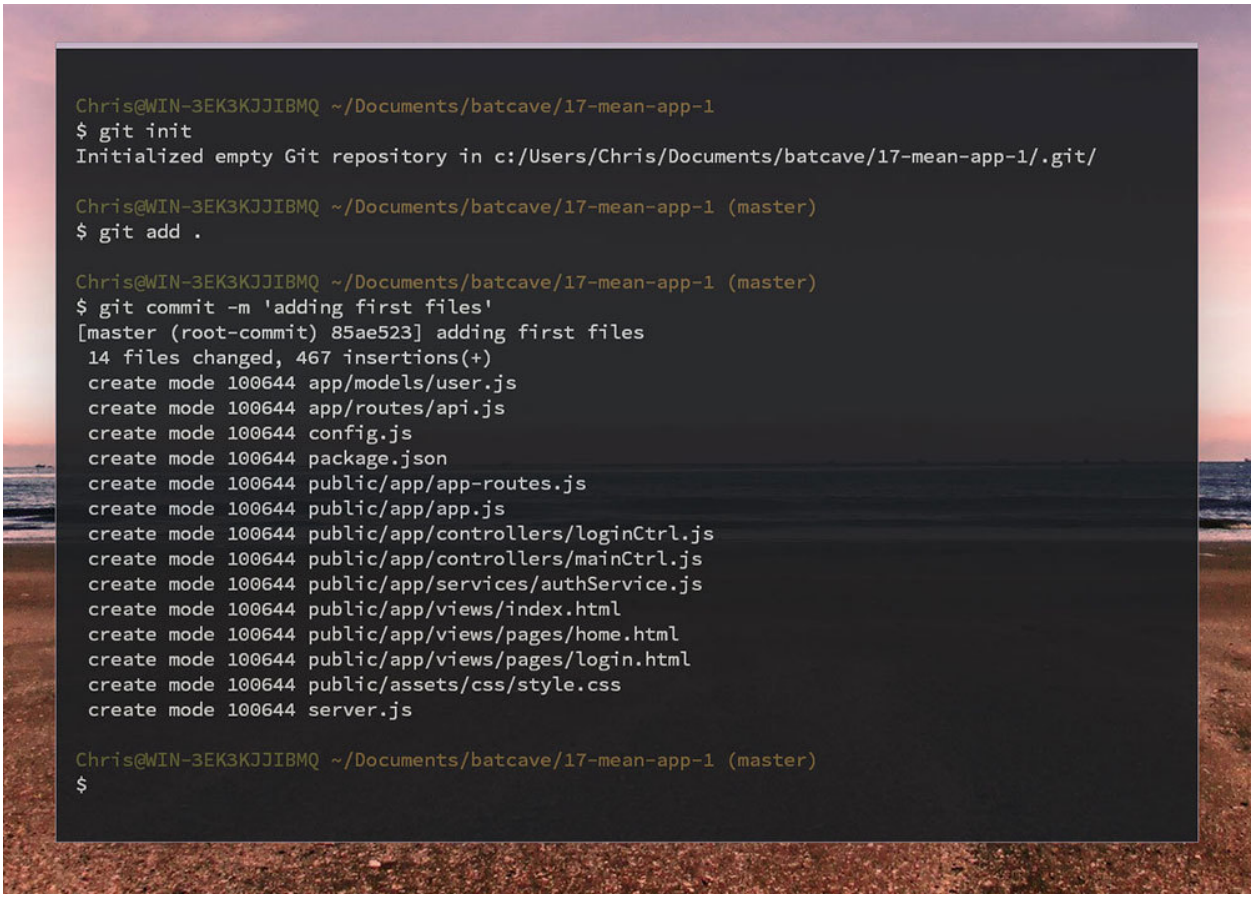
```
git init
```

That will create your git repository. Now we will need to add your files to the repository and commit them.

```
git add .
```

```
git commit -m 'adding first files'
```

[112] https://devcenter.heroku.com/articles/getting-started-with-nodejs
[113] http://nodejs.org/
[114] https://toolbelt.heroku.com/
[115] http://git-scm.com/
[116] http://git-scm.com/

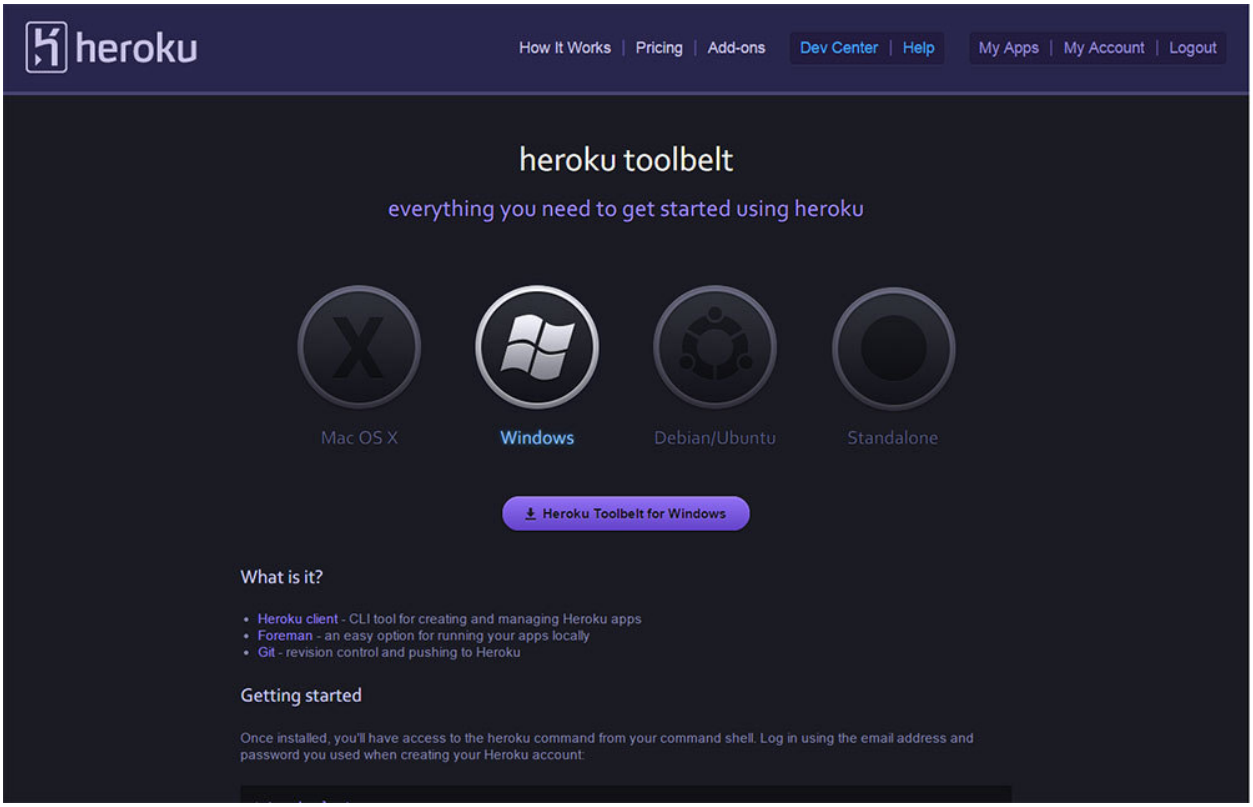


Git Repo Creation

Now that we have our git repository ready to go, we can move onto the Heroku side of things.

The Heroku Toolbelt

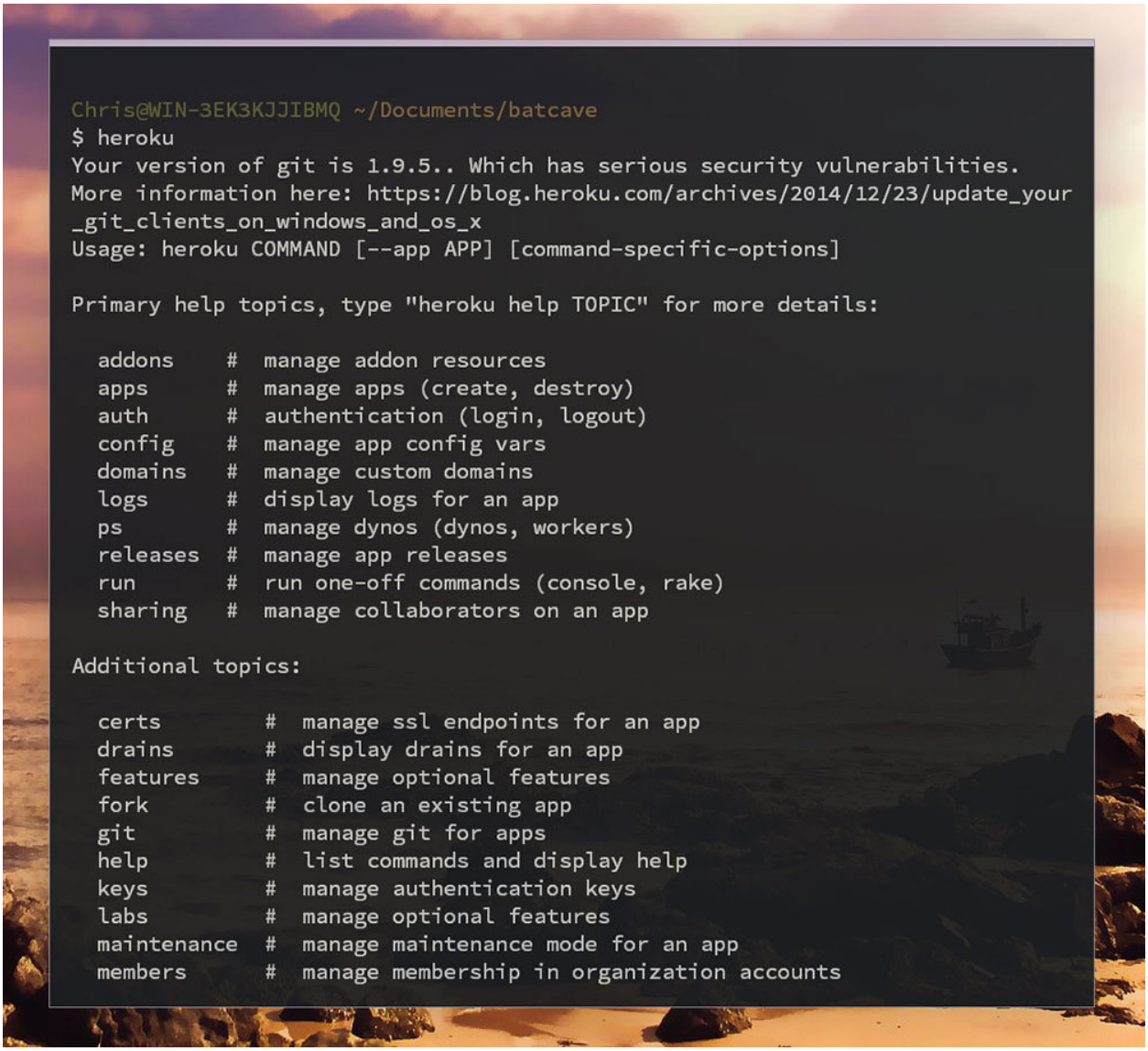
The toolbelt will give us access to the Heroku Command Line Utility which we will need for deploying to Heroku.



Heroku Toolbelt

After we install the Toolbelt, we'll have access to the `heroku` command. Go into your command line and type:

```
1 $ heroku
```



Heroku Command

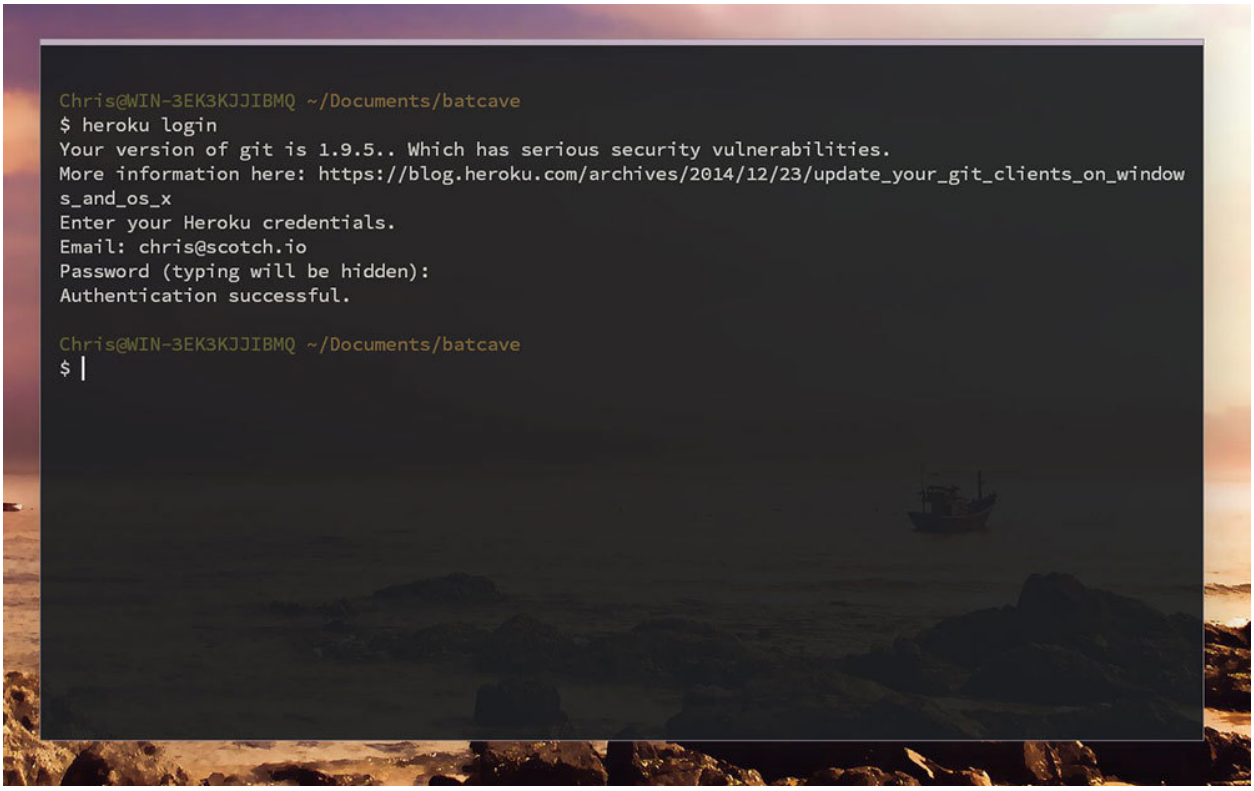
Now you can see all the commands available to us. We'll only be using a few of them when deploying to Heroku in this chapter. The first of these will be logging in.

Logging Into Heroku

This is the way we can link our local desktop to Heroku. We will authenticate from the command line so that Heroku will know that we are authorized to send applications for deployment. This is a fairly easy process.

Just type:

```
$ heroku login
```



Heroku Login

We'll authenticate and Heroku will also ask you to upload a public SSH key. Since I already have my SSH key on Heroku, it did not prompt me for that. Now let's move on and start building a very simple application so that we can deploy it to Heroku.

Deploying Our User CRM App

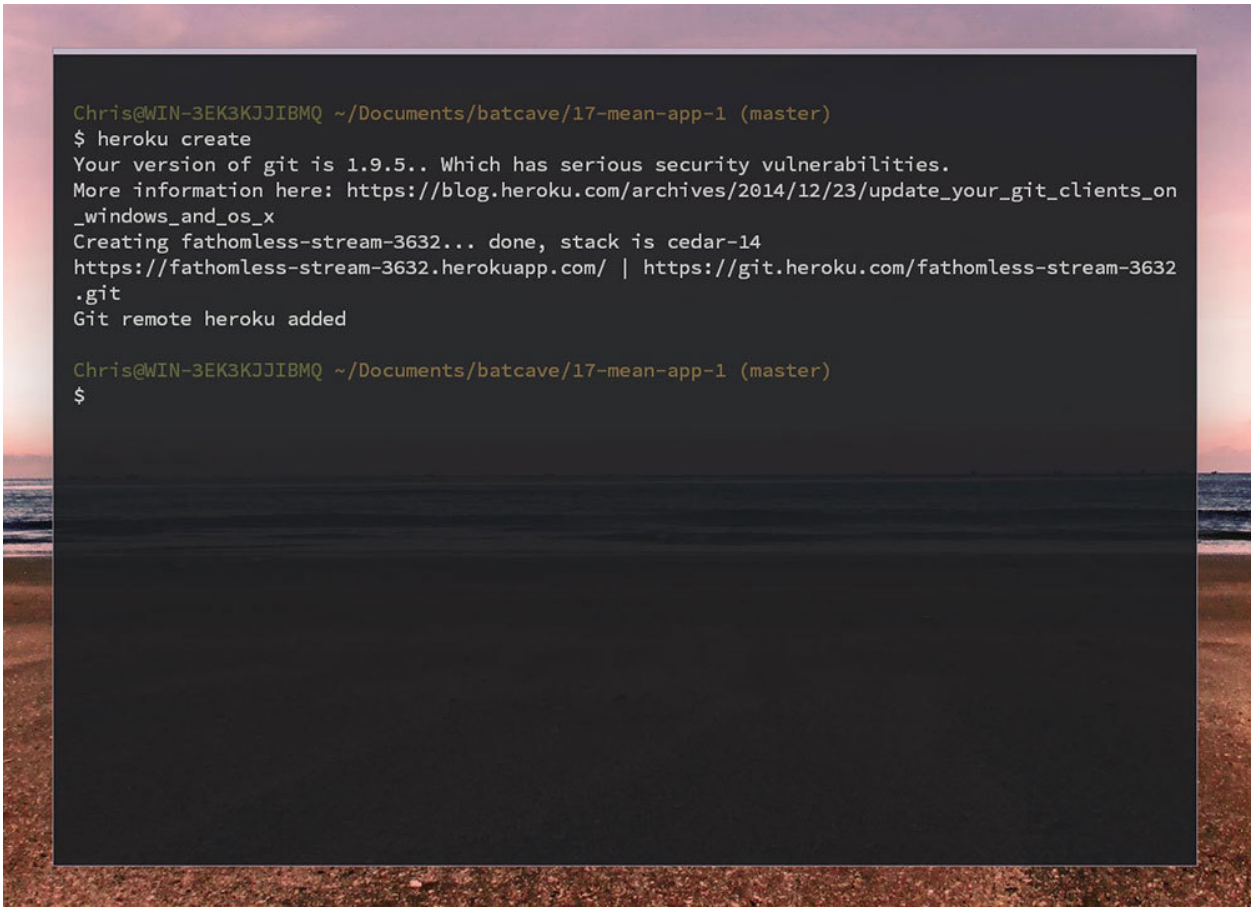
We will be taking the User CRM application that we created and deploy that to Heroku so that we can see exactly how to make our MEAN application live.

This is a very simple process. From within the folder of your git repo, we take the following steps:

1. Create a remote repository (called `heroku` as opposed to our main `origin` remote repository) so our application knows where to push our deployable code
2. Push the repository!

Let's create the remote repository:

```
1 $ heroku create
```



Heroku Create

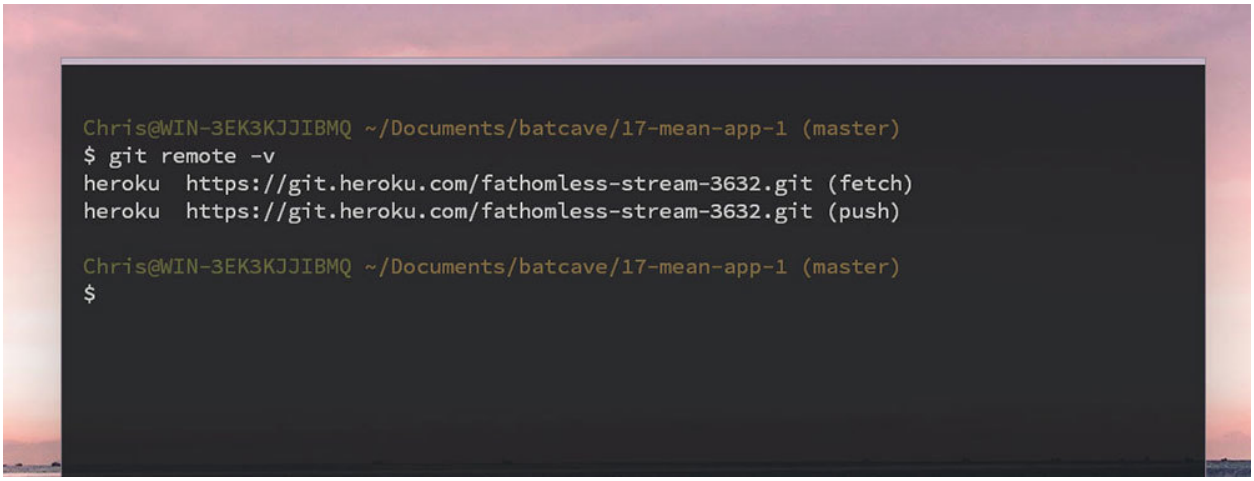
Heroku Warning

Heroku's warning here about updating Git is misplaced. Their blog says to update to version 1.9.5, which is what we are using here. We've reached out to see what can be done about this and will update this chapter with information on that.

Just keep moving forward since this warning does not affect our ability to use git or the Heroku Toolbelt.

We can now see that our remote repository has been created by typing the git command:

```
$ git remote -v
```



Git Remotes

The `heroku create` command will create a random name for our application, in this case, **fathomless-stream-3632**. This means that our application will soon be reachable at `http://fathomless-stream-3632.herokuapp.com`.

That isn't really ideal so let's rename that to **user-crm**. We can see the apps we have created with the command `heroku apps`. Then we'll go ahead and rename it.

```
$ heroku apps:rename user-crm --app fathomless-stream-3632
```

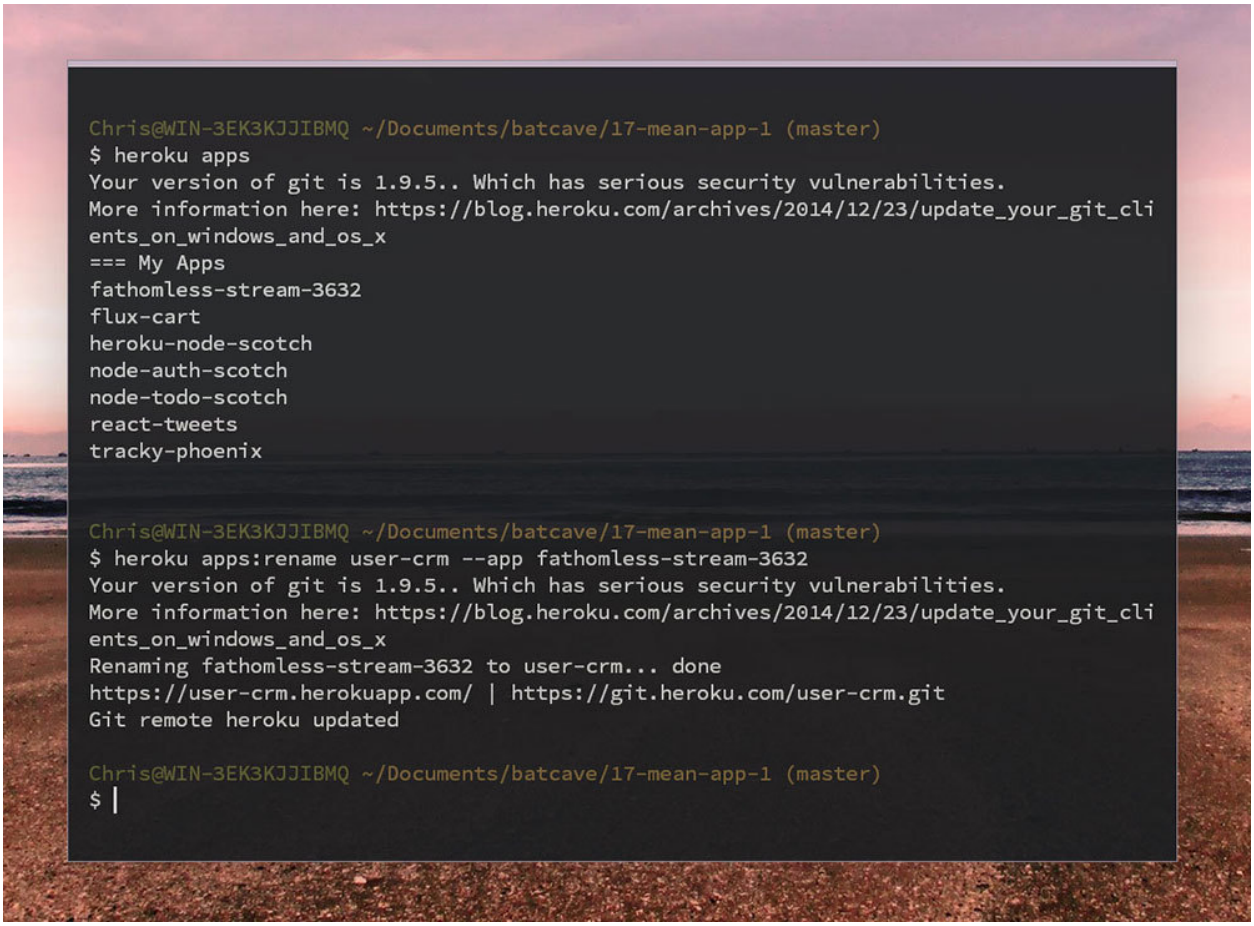
If the name of the application we want is taken (`user-crm` is taken since we used it for this book), then we will have to change the name or add some numbers to the end of it. Let's add `123` to the end of that name.

```
$ heroku apps:rename user-crm-123 --app fathomless-stream-3632
```

You can also bypass all this renaming business by naming your application from the start:

```
$ heroku create user-crm
```

Eventually when you want your application to go live, you probably won't want people to visit it at `http://xxxx.herokuapp.com`. Heroku provides an easy way to point any of the domains you may own to this application so don't worry. We'll revisit how to point a domain to our Heroku app later in this chapter.



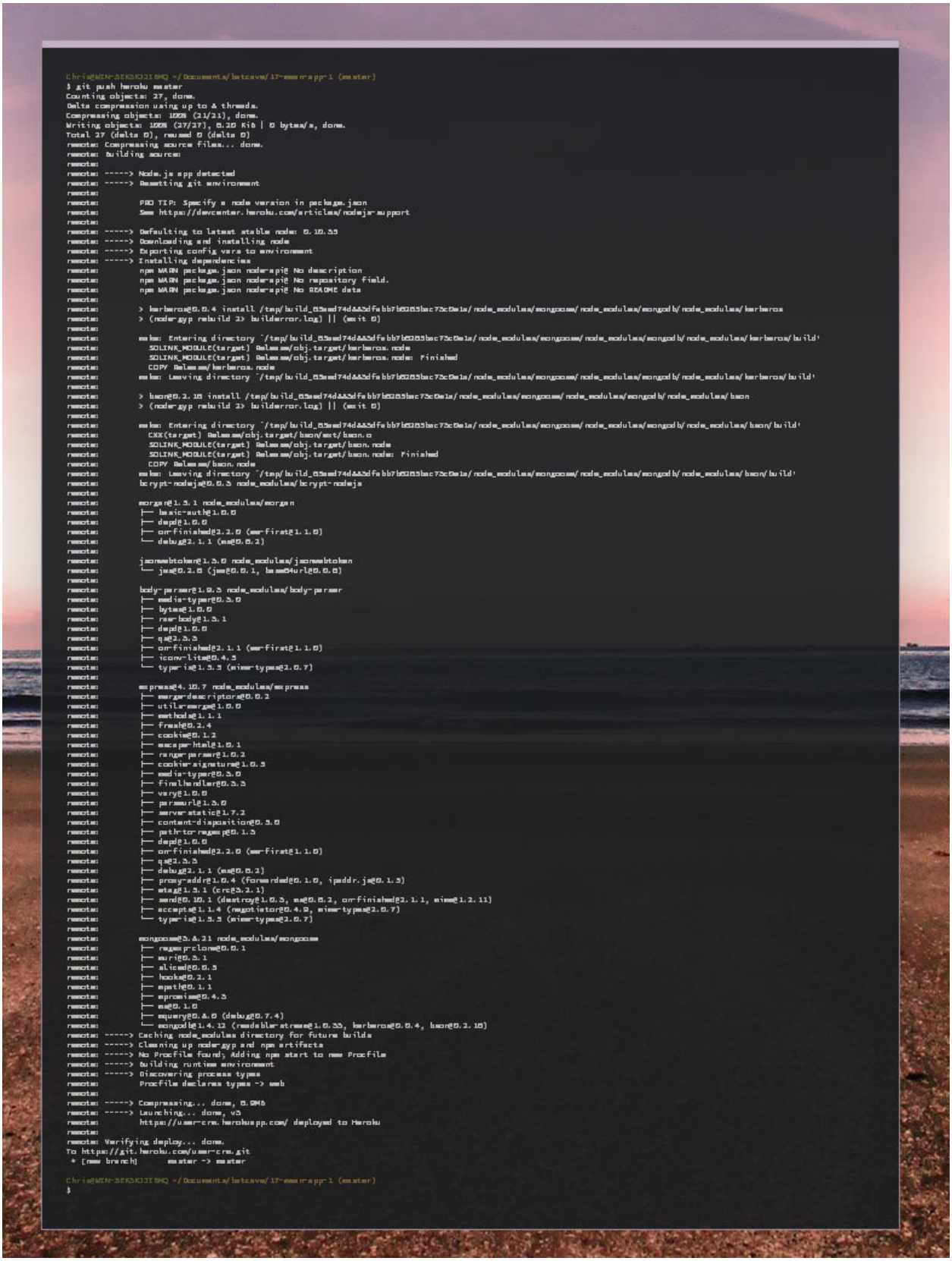
Image

Deploying Code

Now that we have everything in order, we're going to go ahead and get our site up to Heroku! This takes one simple git command. We are essentially saying, push all of our local git repository code to the remote repository hosted by Heroku.

```
1  $ git push heroku master
```

This informs git that we would like to push to the newly create `heroku` repository and the `master` branch (which is the default git branch that was automatically created for us).



Heroku Deploy

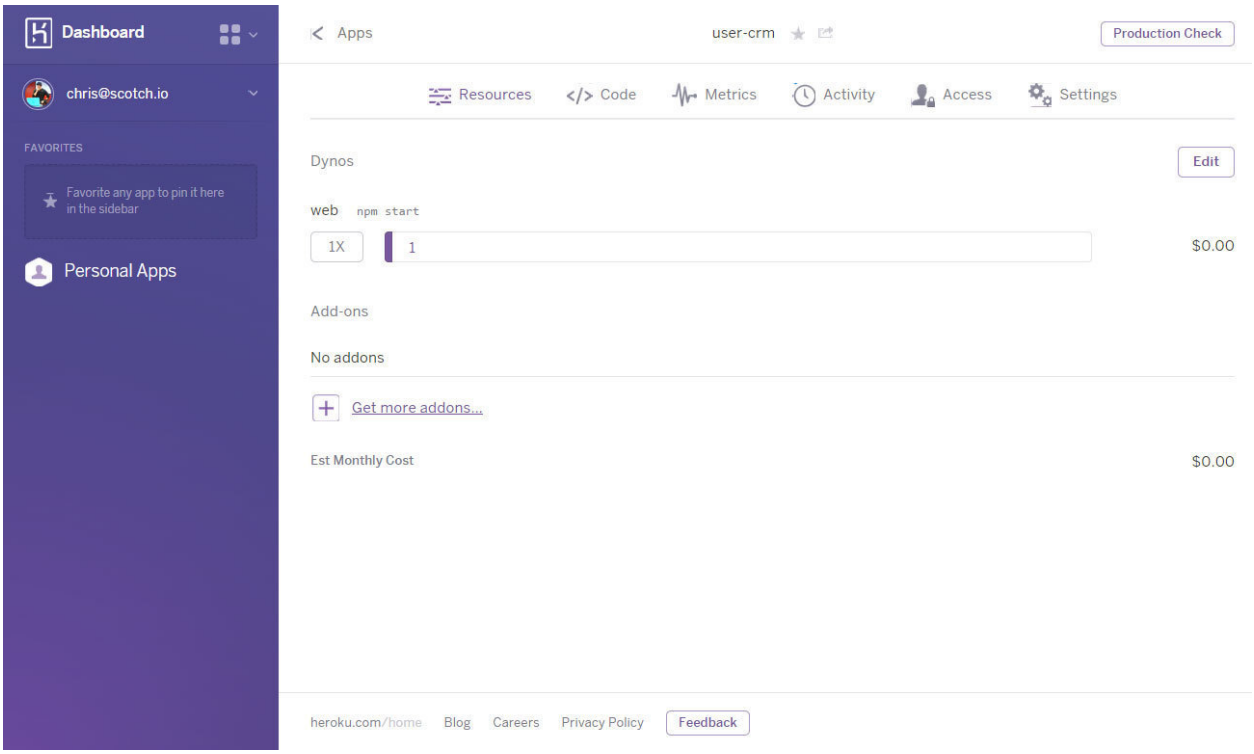
We can see as Heroku goes through a great many things when deploying our application. It sees that our we have a Node based application. It will then go through adding the dependencies by reading our `package.json` file. It will also start our Node server by running the file declared as `main` in the `package.json` file. That file will be `server.js` in our case.

Ensure One Instance Is Running

We want to be sure that our app is running and that Heroku has started a server for us (called dynos). Type the following to get confirmation:

```
1 $ heroku ps:scale web=1
```

If we go into our dashboard, we are able to see under the **Dynos** section, that we have one dyno and it is the free tier. You can quickly scale your applications up or down by editing how many dynos are used.



Heroku Dynos

Our application is now ready for viewing!

View Our Application in Browser

We're finally at the part that we've been waiting for since the beginning of the book! Seeing our full stack MEAN application in the browser!

If you remember the crazy random name that Heroku generated for you, or renamed the app yourself, go ahead and visit that in browser:

https://user-crm.herokuapp.com/[117]

Heroku also provides a shortcut to open your application in the browser using the command line:

```
$ heroku open
```

It's magic! We have our site in the browser just like we wanted.

Defining a Specific Run Command

Sometimes we may have a specific command that we want to run when Heroku gets its hands on our application. This could be using a task runner to start the server like grunt or gulp (we'll get into gulp in the next chapter). Heroku will look at the `main` attribute in our `package.json` file and will also look under `scripts` if we have defined that attribute.

There is another way to tell Heroku exactly what command we would like to see run. You can do this by defining a new file in the root of our project called a `Procfile`.

Just create a `Procfile` in the root of your project and define the start command like so:

```
web: node server.js
```

This tells Heroku that when deploying to the web, this command should be used to start our application. Now when we `git push heroku master`, Heroku will run that specific command and our site will be live.

Using a Current Heroku App

If you switch to a different computer and want to access the Heroku app that you have already been using, there are a few steps that have to be taken. These steps assume you are on a new computer and have not yet installed Heroku.

- Download the Heroku Toolbelt[118]
- Login: `heroku login`
- Add your public key: `heroku keys:add`
- Pull down your current application `heroku git:clone -a app-name`
- Make your improvements

[117] https://user-crm.herokuapp.com/
[118] https://toolbelt.heroku.com/

- Git add and commit your changes
- Push back to heroku: `git push heroku master`

With these simple steps, you can jump onto any computer and immediately grab your current projects.

Using Your Own Domain

Heroku provides very simple steps for adding a domain to an app. For this example, we will assume you have a domain named `www.supercool.com`.

To add that domain to your application, you will need to do two things:

1. Tell Heroku which domains are matched to this application
2. Configure your domain's DNS to point to Heroku

The first part of this is very easy. We will tell Heroku what domain we will need using the following command:

```
1 $ heroku domains:add www.supercool.com
```

Once that step is done, you will need to make sure that you have a CNAME record is generated wherever you have your domain registered. CNAME creation varies across domain registrars, but should look something like the following:

Record	Name	Target
CNAME	www	user-crm.herokuapp.com

This may take some time to propagate before you will be able to visit `http://www.supercool.com` in your browser. This usually depends on the registrar but it usually averages anywhere between 10 minutes to 3 hours.

If you would also like to add the root domain (`supercool.com`), then you will have to repeat the steps above.

As long as you follow the two steps, you will be able to see your site at your domain in no time!

For more information, visit the Heroku Domain Docs[119]. They give more detailed information about more types of domains (wildcards, sub-domains) if you are interested in those. They also provide more instructions for CNAME configuration across a few other registrars.

[119] https://devcenter.heroku.com/articles/custom-domains

Conclusion

We have finally built a full MEAN stack application and have it online. Many of the techniques learned so far in this book can be applied to different applications and use cases.

No matter what you are trying to build, Heroku is a fast and easy way to deploy your apps/websites.

Moving forward, now that we've built an app and deployed it, let's look at different techniques to make our development process easier. We'll be looking at how to use Gulp, a task-runner, and Bower, a front-end resource manager. These two tools will be able to speed up our workflow immensely.

MEAN Development Workflow Tools

Sample MEAN App

We'll need to create a sample MEAN application to see exactly how we can use Bower and Gulp to help our workflow.

Here is the directory structure. Go ahead and create these files and folders.

```
- public/
----- app/
---------- controllers/
--------------- mainCtrl.js
---------- views/
--------------- pages/
-------------------- home.html
--------------- index.html
---------- app.js
---------- app.routes.js
----- assets/
---------- css/
--------------- style.less
- server.js
```

This will be a MEAN project so go ahead and run:

```
$ npm init
```

to create a `package.json` file. All the defaults will be fine when creating this file.

We will now need Express as a dependency so run:

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

Now we have Express. Let's go into `server.js` and start our server to serve up our `index.html` file.

```
// get our packages
var express = require('express');
var app     = express();
var path    = require('path');
var port    = process.env.PORT || 8080;

// configure public assets folder
app.use(express.static(__dirname + '/public'));

// route to send index.html
app.get('/', function(req, res) {
    res.sendFile(path.join(__dirname + '/public/app/views/index.html'));
});

// start the server
app.listen(port);
console.log('Magic happens on http://localhost:' + port);
```

This is a very simple file, and all we need to do is grab our dependencies and serve up our `index.html` file before we start the server.

Let's quickly wire up the rest of our pages:

public/app/controllers/mainCtrl.js

```
angular.module('mainCtrl', [])

.controller('mainController', function() {

  var vm = this;

  vm.message = 'this is my message!';

});
```

public/app/views/pages/home.html

```
Home Page!
```

public/app/views/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Workflow!</title>

  <!-- For Angular Routing -->
  <base href='/'>

  <!-- CSS -->
  <!-- load bootstrap here -->
  <!-- we'll load the css file in the gulp section -->

  <!-- JS / LIBS -->
  <!-- load angular and angular-route here -->

  <!-- APP -->
  <script src="app/controllers/mainCtrl.js"></script>
  <script src="app/app.routes.js"></script>
  <script src="app/app.js"></script>
</head>
<body class="container" ng-app="myApp" ng-controller="mainController as main">

  <div class="jumbotron text-center">
    <h1>{{ main.message }}</h1>
  </div>

  <div ng-view></div>

</body>
</html>
```

public/app/app.js

```
angular.module('myApp', ['app.routes', 'mainCtrl']);
```

public/app/app.routes.js

```
angular.module('app.routes', ['ngRoute'])

.config(function($routeProvider, $locationProvider) {

  $routeProvider

    .when('/', {
      templateUrl : 'app/views/pages/home.html',
      controller  : 'mainController',
      controllerAs: 'main'
    });

  $locationProvider.html5Mode(true);
});
```

public/assets/css/style.less

```
/* VARIABLES
====================== */
@blue: #A6D0C7;
@purple: #993399;
@red: #cc3333;

/* MAIN
====================== */
body  {
  background:@blue;
  color:@purple;
  border-top:20px solid @red;
  padding-top:50px;
}
```

Now we can start our server using:

```
$ nodemon server.js
```

This app won't do much since we haven't grabbed Bootstrap, Angular, or Angular Route. We'll be using a tool called Bower to pull in those resources now.

Bower

Bower[120] is a package manager specifically used for frontend resources. You can use bower to pull in any CSS/JS libraries like Bootstrap, Angular, jQuery, Animate.css, Moment, and so many more.

Bower works very similar to npm. It is just a package manager after all. Just like npm uses a `package.json` file to read all of the packages that it needs to go and grab, bower uses a `bower.json` file.

Installing Bower

We will first need to install bower to use it. Luckily it is an npm package so we can install it by typing the following command:

```
1 $ npm install -g bower
```

We are installing bower globally with `-g` so that we have access to it anywhere on our system.

Let's create a `bower.json` file by using the init command bower provides. We'll just stick to all the defaults. Run the following:

```
1 $ bower init
```

Now that we have bower and our `bower.json` file, we will be able to search for and install packages.

Installing a Package

```
1 $ bower install <package_name> --save
```

Just like npm, if we add the `--save` modifier, this package will be saved to our `bower.json` file.

Let's install Bootstrap and see how that works.

```
1 $ bower install bootstrap
```

We can now see that a new folder was created called **bower_components**. By default, bower will place packages here.

Other installation methods: You can also install a package based on its GitHub URL. Just type `bower install <github-url>`.

Searching for Packages

There are two ways to search for a package. With bower installed, you are able to search from the command line.

[120] http://bower.io/

```
$ bower search <package_name>
```

This can be tedious and doesn't really offer the best interface especially when there are a lot of results (try running `bower search angular`) to see a giant list.

The easier method of searching for packages is through the Bower website. They offer their list of resources available for searching right from their site.

Search Packages on Bower's Site[121]

Specifying A Directory

By default Bower will place all resources in the root directory in a folder called **bower_components**. This isn't the most ideal place to put our files since we have already decided that all files that are associated with the frontend of our applications will be placed in the **public/** folder.

Let's change the default bower folder to **public/assets/libs**. This can be done by creating a new file in the root of our project. This file will be called `.bowerrc` and is a very simple file to create.

Here's our `.bowerrc` file to move the `bower_components` folder.

```
{
  "directory": "public/assets/libs"
}
```

Now our files will be placed in the folder we just specified and we are able to keep our root directly cleaner.

Using a Package

Let's install all the package that we'll need for one of our usual MEAN stack applications. Run the following command:

```
$ bower install bootstrap angular angular-route angular-animate --save
```

Once we have the package in our project, we just need to link to the right files. By clicking through your new **public/assets/libs** folder, you will be able to determine exactly which file you want.

[121] http://bower.io/search

```
<head>
  <meta charset="UTF-8">
  <title>Workflow!</title>

  <!-- CSS -->
  <link rel="stylesheet" href="assets/libs/bootstrap/dist/css/bootstrap.min.css">

  <!-- JS / LIBS -->
  <script src="assets/libs/angular/angular.min.js"></script>
  <script src="assets/libs/angular-route/angular-route.min.js"></script>
  <script src="assets/libs/angular-animate/angular-animate.min.js"></script>

  <!-- APP -->
  <script src="app/controllers/mainCtrl.js"></script>
  <script src="app/app.routes.js"></script>
  <script src="app/app.js"></script>
</head>
```

Now we can see our application in our browser after we start our server using:

```
$ nodemon server.js
```

You can start to see how your workflow becomes much faster. You'll just need to run two commands (`bower init` and `bower install <package_name> --save`) and then you have all the assets you need.

This allows for a much cleaner process than going through and finding the assets online, downloading, linking, and all that mess. Also, like npm, having your dependencies defined in one file lets other developers know exactly what is needed for the current project.

Gulp

You may have heard of the task runner Grunt. Gulp[122] is the newer kid on the block, but it improves on Grunt in a few ways, the most important being a much simpler syntax for configuration.

So what exactly is a task runner? A task runner like Gulp is able to help automate any tasks you may have in your development process. This could include things like:

- linting files (checking them for errors)
- minifying files

[122] https://github.com/gulpjs/gulp/blob/master/docs/getting-started.md

- process LESS or SCSS
- concatenating multiple files into one
- Gulp can even start our nodemon server for us
- so much more…

Installing Gulp

Just like Bower and Nodemon, we are going to install gulp globally so that we have access to it across our projects.

```
1 $ npm install -g gulp
```

We will also want to install gulp in our specific project's **devDependencies**.

```
1 $ npm install gulp --save-dev
```

Great! Now we have Gulp ready to go and we can start to use it. First, let's compile our LESS file we created earlier into a CSS file so our browser will be able to use it.

Compiling LESS

Each task we will want to do is an npm package that extends Gulp. For example, since we want to compile LESS files, we will need to also install gulp-less[123]. Let's do that now

```
1 $ npm install gulp-less --save-dev
```

Now that we have that plugin, let's go ahead and use it. Like Bower and npm, we will need a configuration file in the root of our document. This file will tell Gulp exactly what to do when we start it up.

First we will need to create a `gulpfile.js` in the root of our document.

Inside of our `gulpfile.js` let's start using LESS:

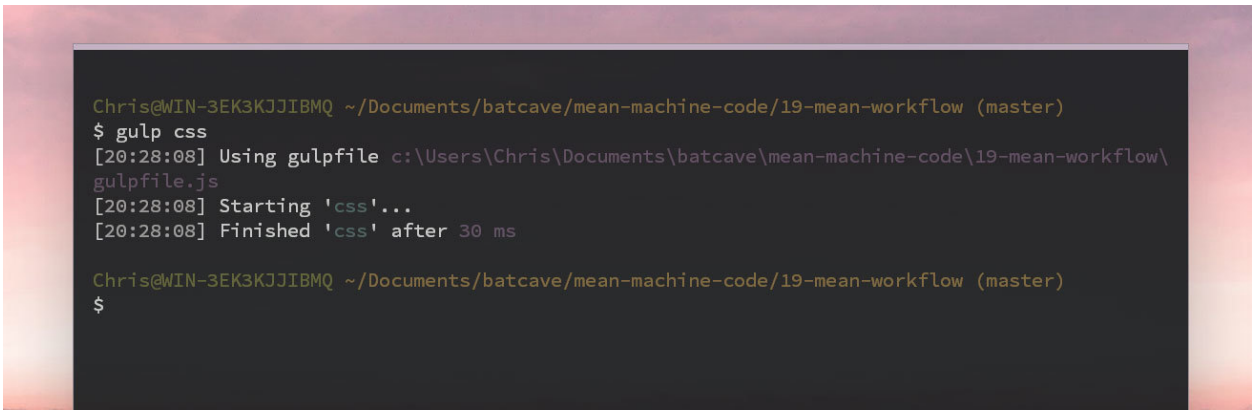
[123] https://github.com/plus3network/gulp-less

```
// load the plugins
var gulp = require('gulp');
var less = require('gulp-less');

// define a task called css
gulp.task('css', function() {

  // grab the less file, process the LESS, save to style.css
  return gulp.src('public/assets/css/style.less')
    .pipe(less())
    .pipe(gulp.dest('public/assets/css'));

});
```

Congratulations, you've just made your first Gulp task! Now all we have to do is go back into our command line and type:

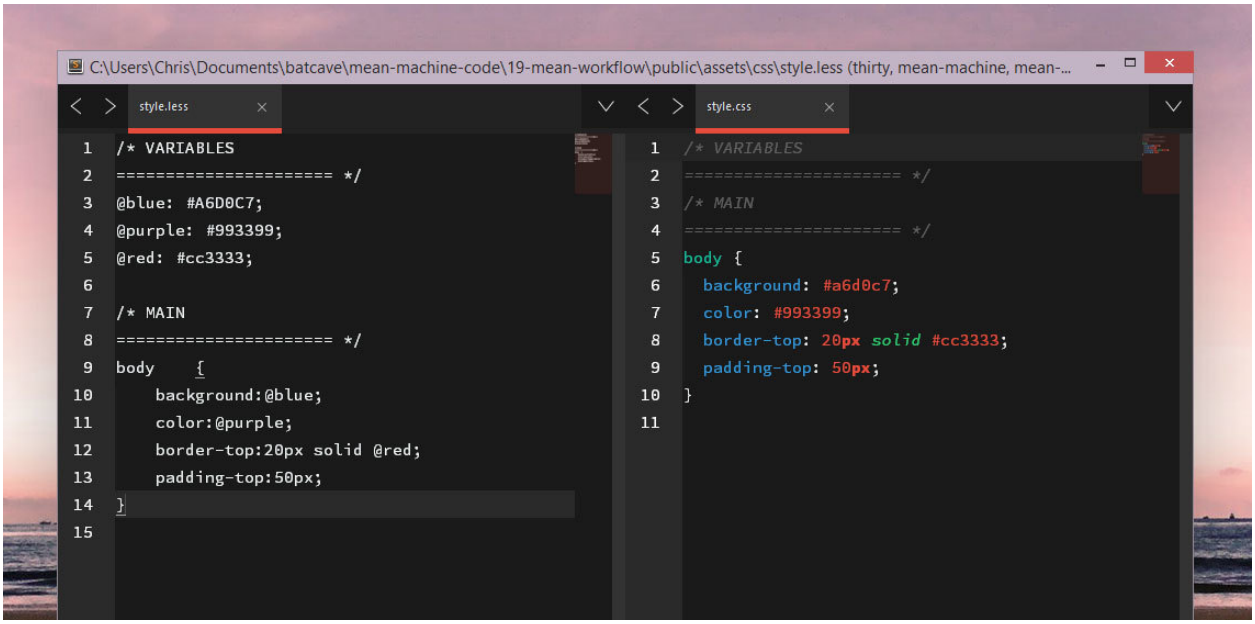
```
$ gulp css
```

That tells Gulp to run that specific task and then we can see that our new `style.css` file will be generated in the `gulp.dest()` folder that we specified.



LESS Compiled

You can see that the LESS is compiled to CSS just as we would expect.



LESS vs. CSS

Minifying CSS

The great thing about Gulp is that we are able to pipe a file (or multiple files) through more than one package in one task. Let's see how we can add CSS minifying to this so that we have a `style.min.css` to use in production.

First install the gulp-minify-css[124] package as well as a package called gulp-rename[125] so that we can rename our file to `style.min.css`:

```
$ npm install gulp-minify-css gulp-rename --save-dev
```

Now we can add it to our `gulpfile.js` and use it in the `css` task.

[124] https://www.npmjs.com/package/gulp-minify-css
[125] https://www.npmjs.com/package/gulp-rename

```
// load the plugins
var gulp      = require('gulp');
var less      = require('gulp-less');
var minifyCSS = require('gulp-minify-css');
var rename    = require('gulp-rename');

// define a task called css
gulp.task('css', function() {

  // grab the less file, process the LESS, save to style.css
  return gulp.src('public/assets/css/style.less')
    .pipe(less())
    .pipe(minifyCSS())
    .pipe(rename({ suffix: '.min' }))
    .pipe(gulp.dest('public/assets/css'));

});
```

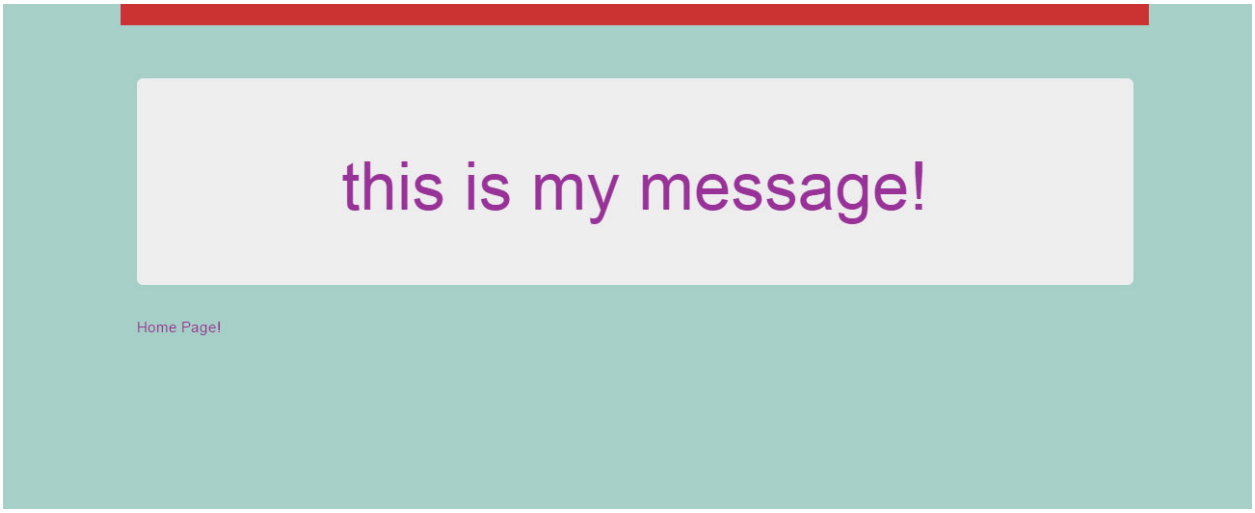
Now when we run

```
$ gulp css
```

We can see a new `style.min.css` file created and it is compiled LESS and minified! We can now go into our `index.html` file and add our new CSS file:

```
<link rel="stylesheet" href="assets/css/style.min.css">
```

All those colors. Isn't it pretty?



Site with LESS

Using a Gulp Package

The process for using a Gulp package is similar to what we just did with LESS.

1. Install the package and `--save-dev`
2. Load the plugin in your `gulpfile.js`
3. Configure a task and use the plugin!

Gulp is easier to configure than Grunt, but the concept is the same: install a package, load it, and configure. Let's move onto other important tasks that deal with JS.

Linting JS

Let's make sure that our JS files have no errors. This includes our Node and Angular files. We'll be using a JSHint plugin called gulp-jshint[126] for this task.

Let's install this new package:

```
$ npm install gulp-jshint --save-dev
```

Now let's bring it into our `gulpfile.js` and create a brand new task called `js`.

```
...
var jshint = require('gulp-jshint');

// css task goes here

// task for linting js files
gulp.task('js', function() {

  return gulp.src(['server.js', 'public/app/*.js', 'public/app/**/*.js'])
    .pipe(jshint())
    .pipe(jshint.reporter('default'));

});
```

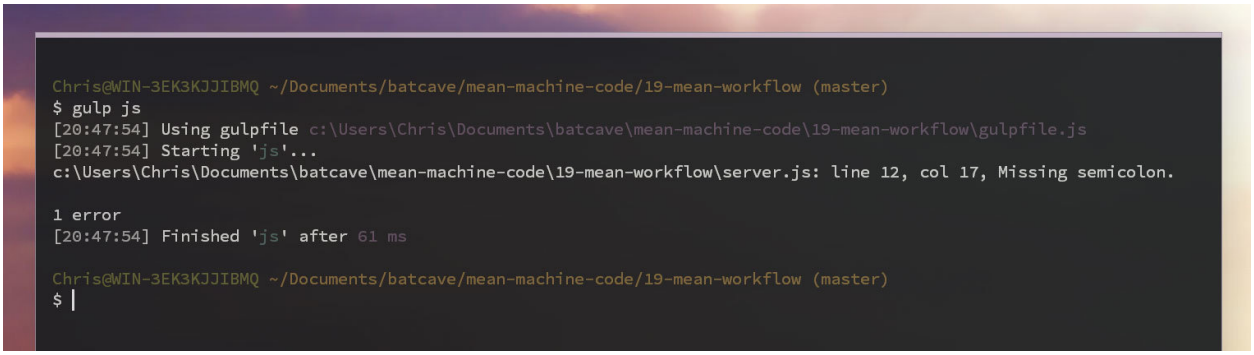
The cool thing we are doing here is using multiple files (passed in as an array) to `gulp.src()`. We are also using the * wildcard to match any files in the **public/app/** folder and any files in subfolders of that folder.

Go into `server.js` and delete a semicolon so that there is something for our jshint to find. Now type:

[126] https://www.npmjs.com/package/gulp-jshint

```
$ gulp js
```

And we will see the error!



JSHint

Minifying, and Concatenating JS

Currently, the `<head>` of our applications have looked like this:

```
<head>
  <meta charset="UTF-8">
  <title>Workflow!</title>

  <!-- For Angular Routing -->
  <base href='/'>

  <!-- CSS -->
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/boo\
tstrap.min.css">
  <link rel="stylesheet" href="assets/css/style.css">

  <!-- JS / LIBS -->
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular.min.js"><\
/script>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.8/angular-route.js"\
></script>

  <!-- APP -->
  <script src="app/controllers/mainCtrl.js"></script>
  <script src="app/app.routes.js"></script>
  <script src="app/app.js"></script>
</head>
```

As our applications start to grow, then we'll have many more requests which will bog down the performance of our application, especially with the custom Angular components we will have like controllers and services.

We will be using two packages to minify and concatenate (bundle together) our JS files so that we will only need to load a single JS file. Our `index.html` file will look very clean with only one JS file to load!

Let's install the two packages, gulp-uglify[127] (for minifying) and gulp-concat[128].

```
$ npm install gulp-uglify gulp-concat --save-dev
```

Let's add them to our `gulpfile.js` and create a new task just for our frontend JS resources since we don't want our backend Node files loaded on the frontend.

```
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');

// task to lint, minify, and concat frontend files
gulp.task('scripts', function() {
  return gulp.src(['public/app/*.js', 'public/app/**/*.js'])
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('public/dist'));
});
```

This new `scripts` task will take all of our frontend Angular files, and bundle them all together in a file called **public/dist/all.js**.

Sure enough, that file shows all of our Angular files (controller, route, and main app file).

[127] https://www.npmjs.com/package/gulp-uglify
[128] https://github.com/wearefractal/gulp-concat

All JS

We can do the same for all of our Angular library files like `angular` and `angular-route` as well. Just add them to the `gulp.src()` array.

Minifying Angular

There is a problem however when we minify Angular files. Angular files have to be declared a certain way or the minifying process will break them.

The Way to Declare Angular Modules for Minification

So far in this book, we have gone with the easier way of defining modules for simplicity. This looks like the following:

```
angular.module('myApp', ['ngRoute'])

.config(function($routeProvider, $locationProvider) {
  // stuff here
})

.controller('mainController', function($http) {
  // stuff here
});
```

The way to declare Angular modules for minification is the following:

```
angular.module('myApp', ['ngRoute'])

.config([
  '$routeProvider',
  '$locationProvider',
  function($routeProvider, $locationProvider) {
    // stuff here
  }
])

.controller('mainController', ['$routeProvider', function($http) {
  // stuff here
}]);
```

Feel free to use the above code and declare modules this way from now on, even though that syntax is annoying to write. There is however another way to minify your Angular files that involves Gulp and another Gulp package.

Using Gulp to Prepare Minifying Angular

Gulp has a package for this specific purpose called gulp-ng-annotate[129].

Let's go ahead and install the package:

```
$ npm install gulp-ng-annotate --save-dev
```

Now we can use the package in our `gulpfile.js` and create a new task:

```
var ngAnnotate = require('gulp-ng-annotate');

// task to lint, minify, and concat frontend angular files
gulp.task('angular', function() {
  return gulp.src(['public/app/*.js', 'public/app/**/*.js'])
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(ngAnnotate())
    .pipe(concat('app.js'))
    .pipe(uglify())
    .pipe(gulp.dest('public/dist'));
});
```

Now can run:

[129] https://www.npmjs.com/package/gulp-ng-annotate

```
$ gulp angular
```

And we will see our new `app.js` created with the right versions of our Angular files.

ng-annotate

Watching for Changes

Next up, we will automate Gulp so that we don't have to go into the command line every time we make a file change and type `gulp <task_name>`.

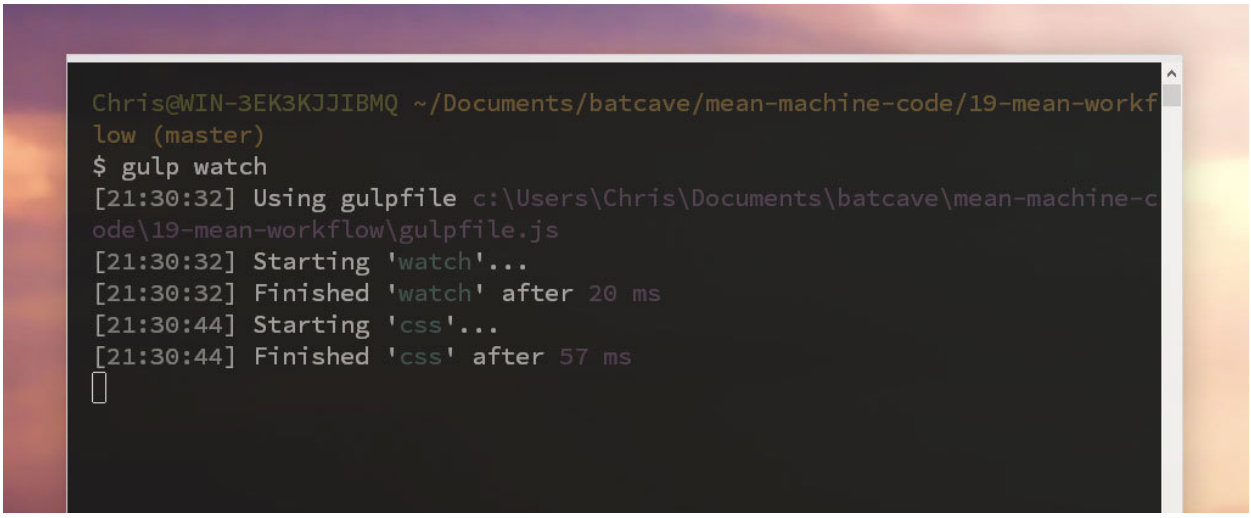
The task to watch files is built into Gulp so there's no need to install a package here. It is important to note however that there is a package out there that is used for more complex setups where there are many files to watch; that package is called gulp-watch[130].

Let's go into our `gulpfile.js` and set it to watch specific files. We will also tell Gulp what tasks to run when a file change has been detected.

```
gulp.task('watch', function() {
  // watch the less file and run the css task
  gulp.watch('public/assets/css/style.less', ['css']);

  // watch js files and run lint and run js and angular tasks
  gulp.watch(['server.js', 'public/app/*.js', 'public/app/**/*.js'], ['js', 'ang\
ular']);
});
```

We have defined a CSS file to watch, and the task to run (which we created earlier). Now when we update our `style.less` file, we can see Gulp go ahead and update the new `style.min.css`.

[130] https://www.npmjs.com/package/gulp-watch



Gulp Watch

Super fast!

Starting a Node Server

We can also use Gulp to start our server. It will just use nodemon but, it's kind of cool to just start up our entire application by typing one command: `gulp`.

The package needed here is gulp-nodemon[131]. Let's install:

```
$ npm install gulp-nodemon --save-dev
```

To configure this package is a little different then we're used to. We will have to define a few things like starting file (`server.js`), types of files to watch (`js less html`), and the tasks to run.

```
var nodemon = require('gulp-nodemon');

// the nodemon task
gulp.task('nodemon', function() {
  nodemon({
    script: 'server.js',
    ext: 'js less html'
  })
    .on('start', ['watch'])
    .on('change', ['watch'])
    .on('restart', function() {
```

[131] https://www.npmjs.com/package/gulp-nodemon

```
12        console.log('Restarted!');
13      });
14  });
15
16  // defining the main gulp task
17  gulp.task('default', ['nodemon']);
```

Notice how we defined the files to start with and the extensions to watch for. We also defined the tasks (`watch` in this case) to work on server start and change.

We are also defining a task called `default` which is the task that Gulp automatically looks for at first.

Now we can run our task with:

```
1  $ gulp
```

We can see our server start and Gulp watching for any files. When a file is changed, Gulp will run the right tasks and then restart the server!

Gulp Nodemon

All of our great development tools are now bundled into this one `gulpfile.js` and we are more efficient developers! There are many more great Gulp plugins to look through, so have fun experimenting with other plugins like imagemin[132] and clean[133].

[132] https://www.npmjs.com/package/gulp-imagemin
[133] https://www.npmjs.com/package/gulp-clean