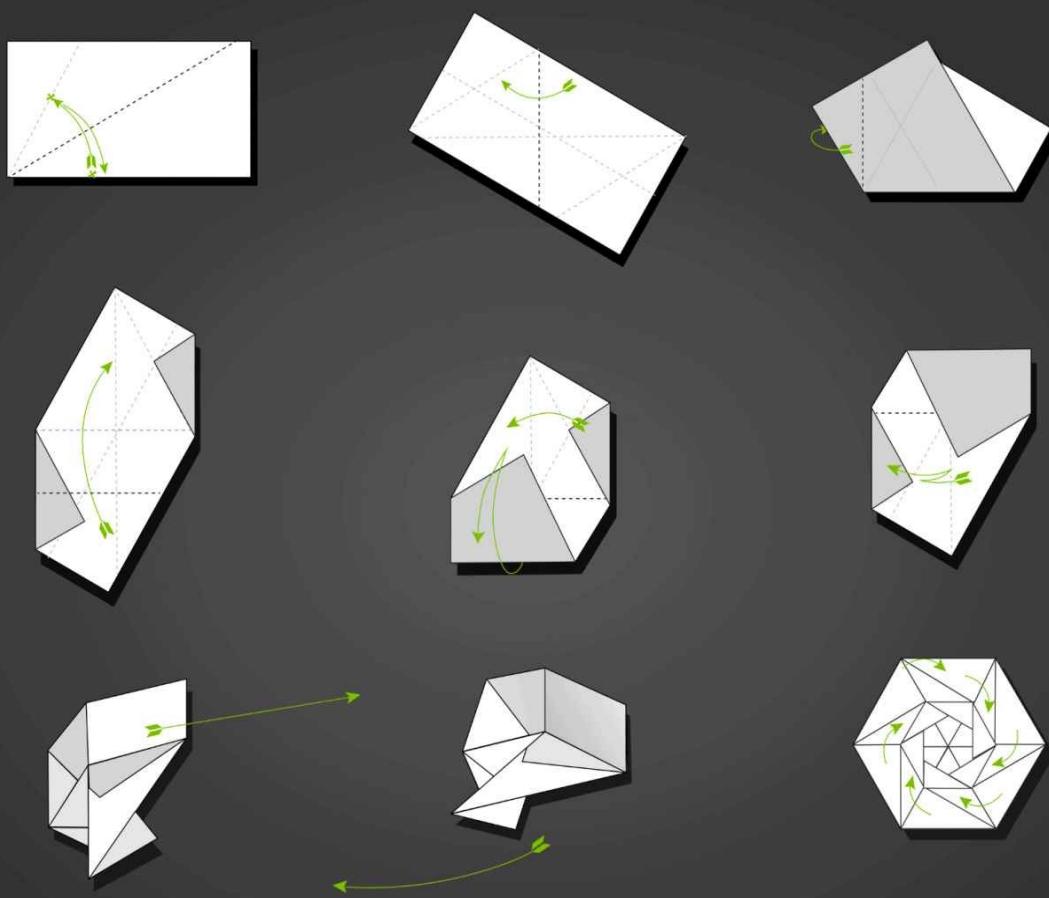


NODEJS

THE COLLECTION



A THREE VOLUME SET

Node.js: The Collection

Copyright © 2018 SitePoint Pty. Ltd.

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher; except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066

Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

While there have been quite a few attempts to get JavaScript working as a server-side language, Node.js (frequently just called Node) has been the first environment that's gained any traction. It's now used by companies such as Netflix, Uber and Paypal to power their web apps. Node allows for blazingly fast performance; thanks to its event loop model, common tasks like network connection and database I/O can be executed very quickly indeed.

From a beginner's point of view, one of Node's obvious advantages is that it uses JavaScript, a ubiquitous language that many developers are comfortable with. If you can write JavaScript for the client-side, writing server-side applications with Node should not be too much of a stretch for you.

This collection contains three books that will help get you up and running with Node. It contains:

- *Your First Week With Node.js*, which will get started using Node, covering all of the basics.
- *9 Practical Node.js Projects*, which offers a selection of hand-on practical projects to develop your skills
- *Node.js: Related Tools & Skills*, which outlines essential tools and skills that all Node developers should know

Who Should Read This Book?

This book is for anyone who wants to start learning server-side development with Node.js. Familiarity with JavaScript is assumed, but we don't assume any previous back-end development experience.

Conventions Used

Code Samples

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

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

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

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

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

```
URL.open("http://www.sitepoint.com/responsive-web-
→design-real-user-testing/?responsive1");
```

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

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

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

Make Sure You Always ...

... pay attention to these important points.

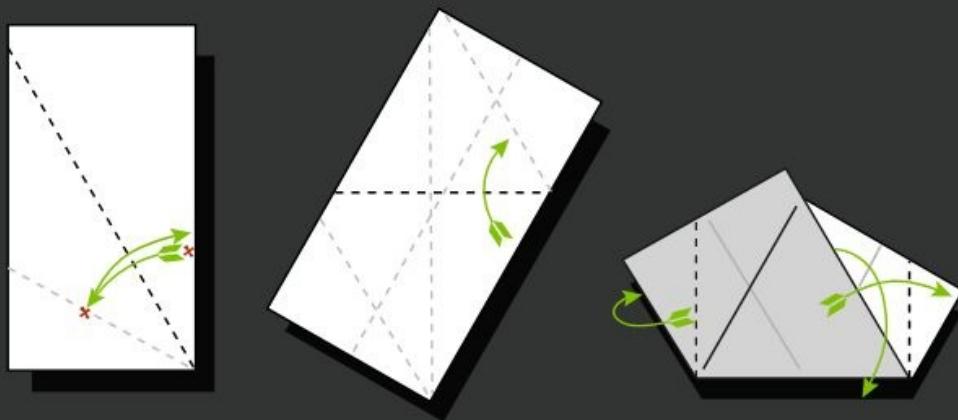
Watch Out!

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

Book 1: Your First Week With Node.js



YOUR FIRST WEEK WITH NODE.JS



BLAZINGLY FAST WEB APPS

Chapter 1: What Is Node and When Should I Use It?

by James Hibbard

So you've heard of Node.js, but aren't quite sure what it is or where it fits into your development workflow. Or maybe you've heard people singing Node's praises and now you're wondering if it's something you need to learn. Perhaps you're familiar with another back-end technology and want to find out what's different about Node.

If that sounds like you, then keep reading. In this article I'll take a beginner-friendly, high-level look at Node.js and its main paradigms. I'll examine Node's main use cases, as well as the current state of the Node landscape, and offer you a wide range of jumping off points (for further reading) along the way.

Node or Node.js?

Please note that, throughout the chapter, I'll use "Node" and "Node.js" interchangeably.

What Is Node.js?

There are plenty of definitions to be found online. Let's take a look at a couple of the more popular ones:

This is [what the project's home page has to say](#):

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

And this is [what StackOverflow has to offer](#):

Node.js is an event-based, non-blocking, asynchronous I/O framework that uses Google's V8 JavaScript engine and libuv library.

Hmmm, “non-blocking I/O”, “event-driven”, “asynchronous” — that's quite a lot to digest in one go. So let's approach this from a different angle and begin by focusing on the other detail that both descriptions mention — the V8 JavaScript engine.

Node Is Built on Google Chrome's V8 JavaScript Engine

[The V8 engine](#) is the open-source JavaScript engine that runs in the Chrome, Opera and Vivaldi browsers. It was designed with performance in mind and is responsible for compiling JavaScript directly to native machine code that your computer can execute.

However, when we say that Node is built on the V8 engine, we don't mean that Node programs are executed in a browser. They aren't. Rather, the creator of Node ([Ryan Dahl](#)) took the V8 engine and enhanced it with various features, such as a file system API, an HTTP library, and a number of operating system-related utility methods.

This means that Node.js is a program we can use to execute JavaScript on our computers. In other words, it's a JavaScript runtime.

How Do I Install Node.js?

In this next section, we'll install Node and write a couple of simple programs. We'll also look at [npm](#), a package manager that comes bundled with Node.

Node Binaries vs Version Manager

Many websites will recommend that you head to [the official Node download page](#) and grab the Node binaries for your system. While that works, I would suggest that you use a version manager instead. This is a program which allows you to install multiple versions of Node and switch between them at will. There are various advantages to using a version manager. For example, it negates potential permission issues which would otherwise see you installing packages with admin permissions.

If you fancy going the version manager route, please consult our quick tip: [Install Multiple Versions of Node.js using nvm](#). Otherwise, grab the correct binaries for your system from the link above and install those.

"Hello, World!" the Node.js Way

You can check that Node is installed on your system by opening a terminal and typing `node -v`. If all has gone well, you should see something like `v8.9.4` displayed. This is the current LTS version at the time of writing.

Next, create a new file `hello.js` and copy in the following code:

```
console.log("Hello, World!");
```

This uses Node's [built-in console module](#) to display a message in a terminal window. To run the example, type the following command:

```
node hello.js
```

If Node.js is configured properly, "Hello, World!" will be displayed.

Node.js Has Excellent ES6 Support

As can be seen on this [compatibility table](#), Node has excellent support for ES6. As you're only targeting one runtime (a specific version of the V8 engine), this means that you can write your JavaScript using the latest and most modern syntax. It also means that you don't generally have to worry about compatibility issues, as you would if you were writing JavaScript that would run in different browsers.

To illustrate the point, here's a second program which makes use of several ES6 features, such as tagged [template literals](#) and [object destructuring](#):

```
function upcase(strings, ...values) {
  return values.map(name => name[0].toUpperCase() + name.slice(1))
    .join(' ') + strings[2];
}

const person = {
  first: 'brendan',
  last: 'eich',
  age: 56,
  position: 'CEO of Brave Software',
};

const { first, last } = person;

console.log(upcase`${first} ${last} is the creator of JavaScript!`);
```

Save this code to a file called `es6.js` and run it from your terminal using the command `node es6.js`. You should see "Brendan Eich is the creator of JavaScript!" output to the terminal.

Introducing npm, the JavaScript Package Manager

As I mentioned earlier, Node comes bundled with a package manager called npm. To check which version you have installed on your system, type `npm -v`.

In addition to being *the* package manager for JavaScript, npm is also the world's largest software registry. There are over 600,000 packages of JavaScript code available to download, with approximately three billion downloads per week. Let's take a quick look at how we would use npm to install a package.

Installing a Package Globally

Open your terminal and type the following:

```
npm install -g jshint
```

This will install the [jshint package](#) globally on your system. We can use it to lint the `es6.js` file from the previous example:

```
jshint es6.js
```

You should now see a number of ES6-related errors. If you want to fix them up, add `/* jshint esversion: 6 */` to the top of the `es6.js` file, re-run the command and linting should pass.

If you'd like a refresher on linting, see: [A Comparison of JavaScript Linting Tools](#).

Installing a Package Locally

We can also install packages locally to a project, as opposed to globally on our system. Create a test folder and open a terminal in that directory. Next type:

```
npm init -y
```

This will create and auto-populate a `package.json` file in the same folder. Next, use npm to install the [lodash package](#) and save it as a project dependency:

```
npm install lodash --save
```

Create a file named `test.js` and add the following:

```
const _ = require('lodash');

const arr = [0, 1, false, 2, '', 3];
console.log(_.compact(arr));
```

And finally, run the script using `node test.js`. You should see `[1, 2, 3]` output to the terminal.

Working With the package.json File

If you look at the contents of the `test` directory, you'll notice a folder entitled `node_modules`. This is where npm has saved `lodash` and any libraries that `lodash` depends on. The `node_modules` folder shouldn't be checked in to version control, and can, in fact, be re-created at any time by running `npm install` from within the project's root.

If you open the `package.json` file, you'll see `lodash` listed under the `dependencies` field. By specifying your project's dependencies in this way, you allow any developer anywhere to clone your project and use npm to install whatever packages it needs to run.

If you'd like to find out more about npm, be sure to read our article [A Beginner's Guide to npm — the Node Package Manager](#).

What Is Node.js Used For?

Now that we know what Node and npm are and how to install them, we can turn our attention to the first of their common uses: they're used to install (npm) and run (Node) various build tools — tools designed to automate the process of developing a modern JavaScript application.

These build tools come in all shapes and sizes, and you won't get far in a modern JavaScript landscape without bumping into them. They can be used for anything from bundling your JavaScript files and dependencies into static assets, to running tests, or automatic code linting and style checking.

We have a wide range of articles covering build tooling on SitePoint. Here's a short selection of my favorites:

- [A Beginner's Guide to Webpack and Module Bundling](#)
- [How to Bundle a Simple Static Site Using Webpack](#)
- [Up and Running with ESLint — the Pluggable JavaScript Linter](#)
- [An Introduction to Gulp.js](#)
- [Unit Test Your JavaScript Using Mocha and Chai](#)

And if you want to start developing apps with any modern JavaScript framework (for example, React or Angular), you'll be expected to have a working knowledge of Node and npm (or maybe [Yarn](#)). This is not because you need a Node backend to run these frameworks. You don't. Rather, it's because these frameworks (and many, many related packages) are all available via npm and rely on Node to create a sensible development environment in which they can run.

If you're interested in finding out what role Node plays in a modern JavaScript app, read [The Anatomy of a Modern JavaScript Application](#).

Node.js Lets Us Run JavaScript on the Server

Next we come to one of the biggest use cases for Node.js — running JavaScript on the server. This is not a new concept, and was first attempted by Netscape way back in 1994. Node.js, however, is the first implementation to gain any real traction, and it provides some unique benefits, compared to traditional languages. Node now plays a critical role in the technology stack of [many high-profile companies](#). Let's have a look at what those benefits are.

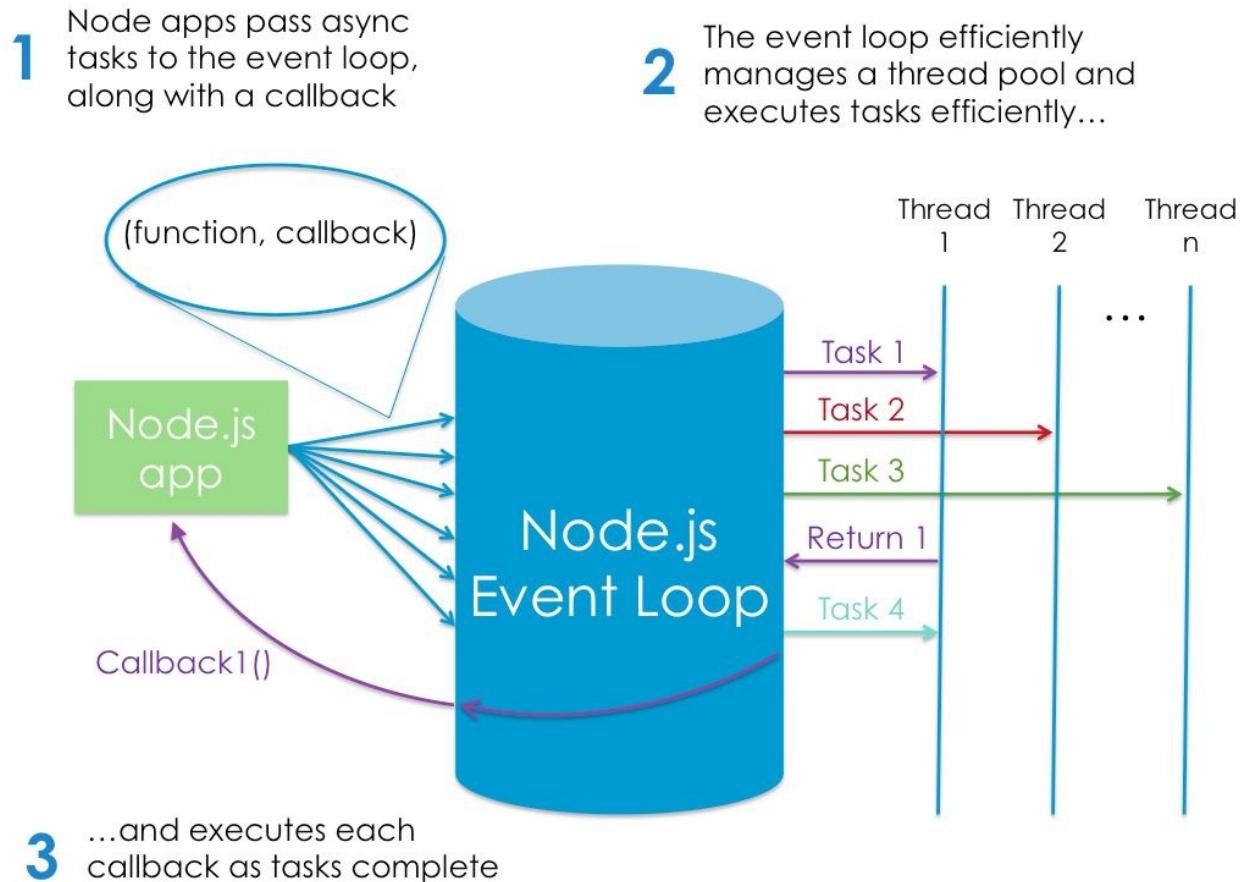
The Node.js Execution model

In very simplistic terms, when you connect to a traditional server, such as Apache, it will spawn a new thread to handle the request. In a language such as PHP or Ruby, any subsequent I/O operations (for example, interacting with a database) block the execution of your code until the operation has completed. That is, the server has to wait for the database lookup to complete before it can move on to processing the result. If new requests come in while this is happening, the server will spawn new threads to deal with them. This is potentially inefficient, as a large number of threads can cause a system to become sluggish — and, in the worse case, for the site to go down. The most common way to support more connections is to add more servers.

Node.js, however, is single-threaded. It is also event-driven, which means that everything that happens in Node is in reaction to an event. For example, when a new request comes in (one kind of event) the server will start processing it. If it then encounters a blocking I/O operation, instead of waiting for this to complete, it will register a callback before continuing to process the next event. When the I/O operation has finished (another kind of event), the server will execute the callback and continue working on the original request. Under the hood, Node uses the [libuv](#) library to implement this asynchronous (i.e. non-blocking) behavior.

Node's execution model causes the server very little overhead, and consequently it's capable of handling a large number of simultaneous connections. The traditional approach to scaling a Node app is to clone it and have the cloned instances share the workload. Node.js even has [a built-in module](#) to help you implement a cloning strategy on a single server.

The following image depicts Node's execution model:



Source: [*Introduction To Node.js by Prof. Christian Maderazo, James Santos*](#)

Are There Any Downsides?

The fact that Node runs in a single thread does impose some limitations. For example, blocking I/O calls should be avoided, and errors should always be handled correctly for fear of crashing the entire process. Some developers also dislike the callback-based style of coding that JavaScript imposes (so much so that there is even [a site dedicated to the horrors of writing asynchronous JavaScript](#)). But with the arrival of [native Promises](#), followed closely by [async await](#) (which is enabled by default as of Node version 7.6), this is rapidly becoming a thing of the past.

"Hello, World!" — Server Version

Let's have a quick look at a "Hello, World!" example HTTP server.

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

http.createServer((request, response) => {
  response.writeHead(200);
  response.end('Hello, World!');
}).listen(3000);

console.log('Server running on http://localhost:3000');
```

To run this, copy the code into a file named `hello-world-server.js` and run it using `node hello-world-server.js`. Open up a browser and navigate to <http://localhost:3000> to see "Hello, World!" displayed in the browser.

Now let's have a look at the code.

We start by requiring Node's native [HTTP module](#). We then use its [createServer](#) method to create a new web server object, to which we pass an anonymous function. This function will be invoked for every new connection that is made to the server.

The anonymous function is called with two arguments (`request` and `response`) which contain the request from the user and the response, which we use to send back a 200 HTTP status code, along with our "Hello World!" message.

Finally, we tell the server to listen for incoming requests on port 3000, and output a message to the terminal to let us know it's running.

Obviously, there's lots more to creating even a simple server in Node (for example, it is important to handle errors correctly), so I'd advise you to [check the documentation](#) if you'd like to find out more.

What Kind of Apps Is Node.js Suited To?

Node is particularly suited to building applications that require some form of real-time interaction or collaboration — for example, chat sites, or apps such as [CodeShare](#), where you can watch a document being edited live by someone else. It's also a good fit for building APIs where you're handling lots of requests that are I/O driven (e.g. which need to perform operations on database), or for sites involving data streaming, as Node makes it possible to process files while they're still being uploaded. If this real-time aspect of Node is something you'd like to look into more, check out our series [Build a Node.js-powered Chatroom Web App](#).

Yet saying this, not everyone is going to be building the next Trello or the next Google Docs and really, there's no reason that you can't use Node to build a simple CRUD app. However, if you follow this route, you'll soon find out that Node is pretty bare-bones and that the way you build and structure the app is left very much up to you. Of course, there are various frameworks you can use to reduce the boilerplate, with [Express](#) having established itself as the primary framework of choice. Yet even a solution such as Express is minimal, meaning that if you want to do anything slightly out of the ordinary, you'll need to pull in additional modules from npm. This is in stark contrast to frameworks such as Rails or Laravel, which come with a lot of functionality out of the box.

If you'd like to look at building a basic, more traditional app, check out our tutorial [How to Build and Structure a Node.js MVC Application](#).

What Are the Advantages of Node.js?

Aside from speed and scalability, an often touted advantage of using JavaScript on a web server — as well as in the browser — is that your brain no longer needs to switch modes. You can do everything in the same language, which, as a developer, makes you more productive (and hopefully, happier). For example, you can share code between the server and the client.

Another of Node's big pluses is that it speaks JSON. JSON is probably the most important data exchange format on the Web, and the lingua franca for interacting with object databases (such as MongoDB). JSON is ideally suited for consumption by a JavaScript program, meaning that when you're working with Node, data can flow neatly between layers without the need for reformatting. You can have one syntax from browser to server to database.

Finally, JavaScript is ubiquitous: most of us are familiar with, or have used JavaScript at some point. This means that transitioning to Node development is potentially easier than to other server-side languages. To quote Craig Buckler in his [Node vs PHP Smackdown](#), JavaScript might remain [the world's most misunderstood language](#) — but, once the concepts click, it makes other languages seem cumbersome.

Other Uses of Node

And it doesn't stop at the server. There are many other exciting and varied uses of Node.js!

For example [it can be used as a scripting language](#) to automate repetitive or error prone tasks on your PC. It can also be used to [write your own command line tool](#), such as this [Yeoman-Style generator](#) to scaffold out new projects.

Node.js can also be used to [build cross-platform desktop apps](#) and even to [create your own robots](#). What's not to love?

Conclusion

JavaScript is everywhere, and Node is a vast and expansive subject. Nonetheless, I hope that in this article I've offered you the beginner-friendly, high-level look at Node.js and its main paradigms that I promised at the beginning. I also hope that when you re-read the definitions we looked at previously, things make a lot more sense.

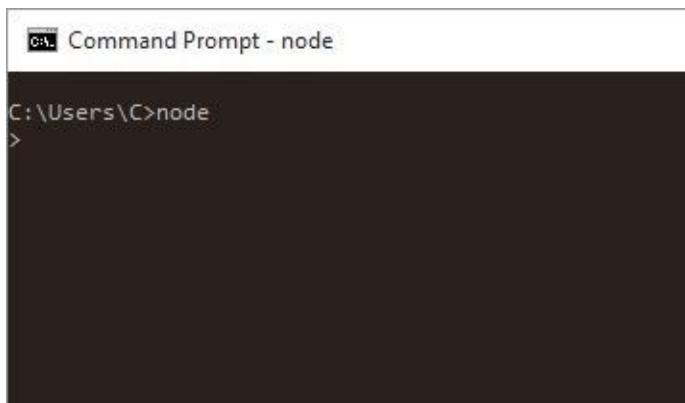
Chapter 2: A Beginner Splurge in Node.js

by Camilo Reyes & Michiel Mulders

It's 3 a.m. You've got your hands over the keyboard, staring at an empty console. The bright prompt over a dark backdrop is ready, yearning to take in commands. Want to hack up Node.js for a little while?

One exciting thing about Node.js is that it runs anywhere. This opens up various possibilities for experimenting with the stack. For any seasoned veteran, this is a fun run of the command line tooling. What's extra special is that we can survey the stack from within the safety net of the command line. And it's cool that we're still talking about JavaScript — so most readers who are familiar with JS shouldn't have any problem understanding how it all works. So, why not fire up node up in the console?

In this chapter, we'll introduce you to Node.js. Our goal is to go over the main highlights while hiking up some pretty high ground. This is an intermediate overview of the stack while keeping it all inside the console. If you want a beginner-friendly guide to Node.js, check out SitePoint's [Build a Simple Back-end Project with Node.js](#) course.



Why Node.js?

Before we begin, let's go over the tidbits that make Node.js stand out from the crowd:

- it's designed for non-blocking I/O
- it's designed for asynchronous operations
- it runs on Chrome's V8 JavaScript engine.

You may have heard these points through many sources, but what does it all mean? You can think of Node.js as the engine that exposes many APIs to the JavaScript language. In traditional computing, where processes are synchronous, the API waits before it runs the next line of code when you perform any I/O operation. An I/O operation is, for example, reading a file or making a network call. Node.js doesn't do that; it's designed from the beginning to have asynchronous operations. In today's computing market, this has a tremendous advantage. Can you think of the last time you bought a new computer because it had a faster single processor? The number of cores and a faster hard drive is more important.

In the remainder of this article, when you see a >, which is a prompt symbol, it means you should hit Enter to type up the next command. Moreover, before running the code in this article, you have to open the CLI and execute the command node. With that said, let's begin our tour!

Callbacks

To start, type up this function:

```
> function add(a, b, callback) { var result = a + b; callback(result)
```

To a newbie, a callback in JavaScript may seem strange. It certainly doesn't look like any classical OOP approach. In JavaScript, functions are objects and objects can take in other objects as parameters. JavaScript doesn't care what an object has, so it follows that a function can take in an object that happens to be yet another function. The **arity**, which is the number of parameters, goes from two in `add()` to a single parameter in the callback. This system of callbacks is powerful, since it enables encapsulation and implementation hiding.

In Node.js, you'll find a lot of APIs that take in a callback as a parameter. One way to think about callbacks is as a delegate. Programming lingo aside, a delegate is a person sent and authorized to represent others. So a callback is like sending someone to run an errand. Given a list of parameters, like a grocery list for example, they can go and do a task on their own.

To play around with `add`:

```
> add(2, 3, function (c) { console.log('2 + 3 = ' + c) });
> add(1, 1, function (c) { console.log('Is 1 + 1 = 3? ' + (c === 3))})
```

There are plenty more creative ways to play around with callbacks. Callbacks are the building blocks for some important APIs in Node.js.

Asynchronous Operations

With callbacks, we're able to start building asynchronous APIs. For example:

```
> function doSomething (asyncCallback) { asyncCallback(); }
> doSomething(function () { console.log('This runs synchronously.'));
```

This particular example has a synchronous execution. But we have everything we need for asynchronicity in JavaScript. The `asyncCallback`, for example, can get delayed in the same thread:

```
> function doSomething (asyncCallback) { setTimeout(asyncCallback, M
➥+ 1000); }
> doSomething(function () { console.log('This runs asynchronously.')
➥console.log('test');
```

We use a `setTimeout` to delay execution in the current thread. Timeouts don't guarantee time of execution. We place a `Math.random()` to make it even more fickle, and call `doSomething()`, followed by a `console.log('test')`, to display delayed execution. You'll experience a short delay between one to two seconds, then see a message pop up on the screen. This illustrates that asynchronous callbacks are unpredictable. Node.js places this callback in a scheduler and continues on its merry way. When the timer fires, Node.js picks up right where execution happens to be and calls the callback. So, you must wrap your mind around petulant callbacks to understand Node.js.

In short, callbacks aren't always what they seem in JavaScript.

Let's go on with something cooler — like a simple DNS lookup in Node.js:

```
> dns.lookup('bing.com', function (err, address, family) { console.l
➥ + address + ', Family: ' + family + ', Err: ' + err);});
```

The callback returns `err`, `address`, and `family` objects. What's important is that return values get passed in as parameters to the callback. So this isn't like your traditional API of `var result = fn('bing.com');`. In Node.js, you must get callbacks and asynchrony to get the big picture. (Check out the [DNS Node.js API](#) for more specifics.) This is what `DNS.lookup` can look like in a console:

```
> dns.lookup('bing.com', function (err, address, family) {
... console.log(' Address: ' + address + ', Family: ' +
..... family + ', Err: ' + err); });
GetAddrInfoReqWrap {
  callback: { [Function: asyncCallback] immediately: true },
  family: 0,
  hostname: '',
  oncomplete: [Function: onlookup],
  domain:
    Domain {
      domain: null,
      _events: { error: [Function] },
      _eventsCount: 1,
      _maxListeners: undefined,
      members: [] } }
> Address: 204.79.197.200, Family: 4, Err: null
```

File I/O

Now let's pick up the pace and do file I/O on Node.js. Imagine this scenario where you open a file, read it and then write content into it. In modern computer architecture, I/O-bound operations lag. CPU registers are fast, the CPU cache is fast, RAM is fast. But you go read and write to disk and it gets slow. So when a synchronous program performs I/O-bound operations, it runs slowly. The better alternative is to do it asynchronously, like so:

```
> var fs = require('fs');
> fs.writeFile('message.txt', 'Hello Node.js', function () { console
→ }); console.log('Writing file...');
```

Because the operation is asynchronous, you'll see "Writing file..." before the file gets saved on disk. The natural use of callback functions fits well in this API. How about reading from this file? Can you guess off the top of your head how to do that in Node.js? We'll give you a hint: the callback takes in `err` and `data`. Give it a try.

Here's the answer:

```
> fs.readFile('message.txt', function(err, data) {
→ console.log(data);});
```

You may also pass in an `encoding` option to get the utf-8 contents of the file:

```
> fs.readFile('message.txt', {encoding: 'utf-8'}, function(err, data
→ );});
```

The use of callback functions with async I/O looks nice in Node.js. The advantage here is that we're leveraging a basic building block in JavaScript. Callbacks get lifted to a new level of pure awesomeness with asynchronous APIs that don't block.

A Web Server

So, how about a web server? Any good exposé of Node.js must run a web server. Imagine an API named `createServer` with a callback that takes in `request` and `response`. You can explore [the HTTP API](#) in the documentation. Can you think of what that looks like? You'll need the `http` module. Go ahead and start typing in the console.

Here's the answer:

```
> var http = require('http');
> var server = http.createServer(function (request, response) { respon
➥ 'Hello Node.js'); });

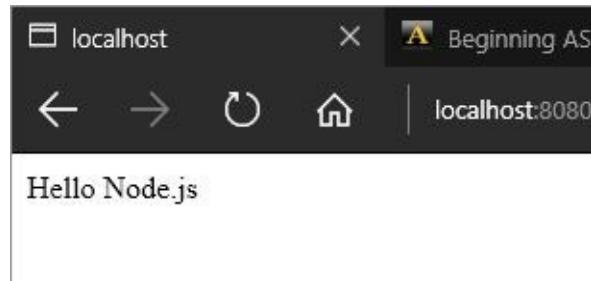
```

The Web is based on a client-server model of requests and responses. Node.js has a `request` object that comes from the client and a `response` object from the server. So the stack embraces the crux of the Web with this simple callback mechanism. And of course, it's asynchronous. What we're doing here is not so different from the file API. We bring in a module, tell it to do something and pass in a callback. The callback works like a delegate that does a specific task given a list of parameters.

Of course, everything is nonsense if we can't see it in a browser. To fix this, type the following in the command line:

```
server.listen(8080);
```

Point your favorite browser to `localhost:8080`, which in my case was Edge.



Imagine the `request` object as having a ton of information available to you. To rewire the server, let's bring it down first:

```
> server.close();
> server = http.createServer(function (request, response) { response
->headers['user-agent']); }); server.listen(8081);
```

Point the browser to `localhost:8081`. The `headers` object gives you user-agent information which comes from the browser. We can also loop through the `headers` object:

```
> server.close();
> server = http.createServer(function (request, response) { Object.k
->headers).forEach(function (key) { response.write(key + ': ' + requ
-> + ' '); }); response.end(); }); server.listen(8082);
```

Point the browser to `localhost:8082` this time. Once you've finished playing around with your server, be sure to bring it down. The command line might start acting funny if you don't:

```
> server.close();
```

So there you have it, creating web servers all through the command line. I hope you've enjoyed this psychedelic trip around node.

Async Await

ES 2017 introduced asynchronous functions. Async functions are essentially a cleaner way to work with asynchronous code in JavaScript. Async/Await was created to simplify the process of working with and writing chained promises. You've probably experienced how unreadable chained code can become.

Creating an `async` function is quite simple. You just need to add the `async` keyword prior to the function:

```
async function sum(a,b) {  
    return a + b;  
}
```

Let's talk about `await`. We can use `await` if we want to force the rest of the code to wait until that Promise resolves and returns a result. `Await` only works with Promises; it doesn't work with callbacks. In addition, `await` can only be used within an `async` function.

Consider the code below, which uses a Promise to return a new value after one second:

```
function tripleAfter1Second(number) {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve(number * 3);  
        }, 1000);  
    });  
}
```

When using `then`, our code would look like this:

```
tripleAfter1Second(10).then((result) => {  
    console.log(result); // 30  
})
```

Next, we want to use `async/await`. We want to force our code to wait for the tripled value before doing any other actions with this result. Without the `await` keyword in the following example, we'd get an error telling us it's not possible to take the modulus of 'undefined' because we don't have our tripled value yet:

```
const finalResult = async function(number) {
  let triple = await tripleAfter1Second(number);
  return triple % 2;
}
```

One last remark on `async/await`: watch out for uncaught errors. When using a `then` chain, we could end it with `catch` to catch any errors occurring during the execution. However, `await` doesn't provide this. To make sure you catch all errors, it's a good practice to surround your `await` statement with a `try ... catch` block:

```
const tripleResult = async function(number) {
  try {
    return await tripleAfter1Second(number);
  } catch (error) {
    console.log("Something wrong: ", error);
  }
}
```

For a more in-depth look at `async/await`, check out [Simplifying Asynchronous Coding with Async Functions](#).

Conclusion

Node.js fits well in modern solutions because it's simple and lightweight. It takes advantage of modern hardware with its non-blocking design. It embraces the client-server model that's intrinsic to the Web. Best of all, it runs JavaScript — which is the language we love.

It's appealing that the crux of the stack is not so new. From its infancy, the Web got built around lightweight, accessible modules. When you have time, make sure to read Tim Berners-Lee's [Design Principles](#). The principle of least power applies to Node.js, given the choice to use JavaScript.

Hopefully you've enjoyed this look at command line tooling. Happy hacking!

Chapter 3: A Beginner's Guide to npm — the Node Package Manager

by Michael Wanyoike & Peter Dierx

To make use of these tools (or packages) in Node.js we need to be able to install and manage them in a useful way. This is where npm, the Node package manager, comes in. It installs the packages you want to use and provides a useful interface to work with them.

In this chapter, I'm going to look at the basics of working with npm. I will show you how to install packages in local and global mode, as well as delete, update and install a certain version of a package. I'll also show you how to work with package.json to manage a project's dependencies. If you're more of a video person, why not sign up for SitePoint Premium and watch our free screencast: [What is npm and How Can I Use It?.](#)

But before we can start using npm, we first have to install Node.js on our system. Let's do that now...

Installing Node.js

Head to the Node.js [download page](#) and grab the version you need. There are Windows and Mac installers available, as well as pre-compiled Linux binaries and source code. For Linux, you can also install Node via the package manager, [as outlined here](#).

For this tutorial we are going to use v6.10.3 Stable. At the time of writing, this is the current [Long Term Support \(LTS\) version of Node](#).

Using a Version manager for Installation

You might also consider [installing Node using a version manager](#). This negates the permissions issue raised in the next section

Let's see where node was installed and check the version.

```
$ which node  
/usr/bin/node  
$ node --version  
v6.10.3
```

To verify that your installation was successful let's give Node's REPL a try.

```
$ node  
> console.log('Node is running');  
Node is running  
> .help  
.break Sometimes you get stuck, this gets you out  
.clear Alias for .break  
.exit Exit the repl  
.help Show repl options  
.load Load JS from a file into the REPL session  
.save Save all evaluated commands in this REPL session to a file  
> .exit
```

The Node.js installation worked, so we can now focus our attention on npm, which was included in the install.

```
$ which npm  
/usr/bin/npm
```

```
$ npm --version  
3.10.10
```

Node Packaged Modules

npm can install packages in local or global mode. In local mode it installs the package in a `node_modules` folder in your parent working directory. This location is owned by the current user. Global packages are installed in `{prefix}/lib/node_modules/` which is owned by root (where `{prefix}` is usually `/usr/` or `/usr/local`). This means you would have to use `sudo` to install packages globally, which could cause permission errors when resolving third-party dependencies, as well as being a security concern. Lets change that:

Changing the Location of Global Packages

Let's see what output `npm config` gives us.

```
$ npm config list
; cli configs
user-agent = "npm/3.10.10 node/v6.10.3 linux x64"

; userconfig /home/sitepoint/.npmrc
prefix = "/home/sitepoint/.node_modules_global"

; node bin location = /usr/bin/nodejs
; cwd = /home/sitepoint
; HOME = /home/sitepoint
; "npm config ls -l" to show all defaults.
```

This gives us information about our install. For now it's important to get the current global location.

```
$ npm config get prefix
/usr
```

This is the prefix we want to change, so as to install global packages in our home directory. To do that create a new directory in your home folder.

```
$ cd ~ && mkdir .node_modules_global
$ npm config set prefix=$HOME/.node_modules_global
```

With this simple configuration change, we have altered the location to which global Node packages are installed. This also creates a `.npmrc` file in our home directory.

```
$ npm config get prefix
/home/sitepoint/.node_modules_global
$ cat .npmrc
prefix=/home/sitepoint/.node_modules_global
```

We still have npm installed in a location owned by root. But because we changed our global package location we can take advantage of that. We need to install npm again, but this time in the new user-owned location. This will also install the latest version of npm.

```
$ npm install npm --global
└─ npm@5.0.2
    ├─ abbrev@1.1.0
    └─ ansi-regex@2.1.1
...
└─ wrappy@1.0.2
└─ write-file-atomic@2.1.0
```

Finally, we need to add `.node_modules_global/bin` to our `$PATH` environment variable, so that we can run global packages from the command line. Do this by appending the following line to your `.profile`, `.bash_profile` or `.bashrc` and restarting your terminal.

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

Now our `.node_modules_global/bin` will be found first and the correct version of `npm` will be used.

```
$ which npm
/home/sitepoint/.node_modules_global/bin/npm
$ npm --version
5.0.2
```

Installing Packages in Global Mode

At the moment we only have one package installed globally — that is the npm package itself. So let's change that and install [UglifyJS](#) (a JavaScript minification tool). We use the `--global` flag, but this can be abbreviated to `-g`.

```
$ npm install uglify-js --global
/home/sitepoint/.node_modules_global/bin/uglifyjs ->
/home/sitepoint/.node_modules_global/lib/node_modules/uglify-js/bin/
+ uglify-js@3.0.15
added 4 packages in 5.836s
```

As you can see from the output, additional packages are installed — these are UglifyJS's dependencies.

Listing Global Packages

We can list the global packages we have installed with the `npm list` command.

```
$ npm list --global
home/sitepoint/.node_modules_global/lib
  └── npm@5.0.2
    ├── abbrev@1.1.0
    ├── ansi-regex@2.1.1
    ├── ansicolors@0.3.2
    └── ansistyles@0.1.3
  .....
  └── uglify-js@3.0.15
    ├── commander@2.9.0
    │   └── graceful-readlink@1.0.1
    └── source-map@0.5.6
```

The output however, is rather verbose. We can change that with the `--depth=0` option.

```
$ npm list -g --depth=0
/home/sitepoint/.node_modules_global/lib
  └── npm@5.0.2
    └── uglify-js@3.0.15
```

That's better — just the packages we have installed along with their version numbers.

Any packages installed globally will become available from the command line. For example, here's how you would use the Uglify package to minify `example.js` into `example.min.js`:

```
$ uglifyjs example.js -o example.min.js
```

Installing Packages in Local Mode

When you install packages locally, you normally do so using a package.json file. Let's go ahead and create one.

```
$ npm init
package name: (project)
version: (1.0.0)
description: Demo of package.json
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

Press Enter to accept the defaults, then type yes to confirm. This will create a package.json file at the root of the project.

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

A Quicker Way

If you want a quicker way to generate a package.json file use `npm init --y`

The fields are hopefully pretty self-explanatory, with the exception of `main` and `scripts`. The `main` field is the primary entry point to your program and the `scripts` field lets you specify script commands that are run at various times in the lifecycle of your package. We can leave these as they are for now, but if you'd like to find out more, see the [package.json documentation on npm](#) and this article on [using npm as a build tool](#).

Now let's try and install [Underscore](#).

```
$ npm install underscore
npm notice created a lockfile as package-lock.json. You should commit
npm WARN project@1.0.0 No description
npm WARN project@1.0.0 No repository field.

+ underscore@1.8.3
added 1 package in 0.344s
```

A Lockfile?

Note that a lockfile is created. We'll be coming back to this later.

Now if we have a look in package.json we will see that a dependencies field has been added:

```
{
  ...
  "dependencies": {
    "underscore": "^1.8.3"
  }
}
```

Managing Dependencies with package.json

As you can see, Underscore v1.8.3 was installed in our project. The caret (^) at the front of the version number indicates that when installing, npm will pull in the highest version of the package it can find where the only the major version has to match (unless a `package-lock.json` file is present). In our case, that would be anything below v2.0.0. This method of versioning dependencies (major.minor.patch) is known as semantic versioning. You can read more about it here: [Semantic Versioning: Why You Should Be Using it.](#)

Also notice that Underscore was saved as a property of the `dependencies` field. This has become the default in the latest version of npm and is used for packages (like Underscore) required for the application to run. It would also be possible to save a package as a `devDependency` by specifying a `--save-dev` flag. `devDependencies` are packages used for development purposes, for example for running tests or transpiling code.

You can also add `private: true` to `package.json` to prevent accidental publication of private repositories as well as suppressing any warnings generated when running `npm install`.

By far and away the biggest reason for using `package.json` to specify a project's dependencies is portability. For example, when you clone someone else's code, all you have to do is run `npm i` in the project root and npm will resolve and fetch all of the necessary packages for you to run the app. We'll look at this in more detail later.

Before finishing this section, let's quickly check Underscore is working. Create a file called `test.js` in the project root and add the following:

```
const _ = require('underscore');
console.log(_.range(5));
```

Run the file using `node test.js` and you should see `[0, 1, 2, 3, 4]` output to the screen.

Uninstalling Local Packages

npm is a package manager so it must be able to remove a package. Let's assume that the current Underscore package is causing us compatibility problems. We can remove the package and install an older version, like so:

```
$ npm uninstall underscore
removed 2 packages in 0.107s
$ npm list
project@1.0.0 /home/sitepoint/project
└── (empty)
```

Installing a Specific Version of a Package

We can now install the Underscore package in the version we want. We do that by using the @ sign to append a version number.

```
$ npm install underscore@1.8.2
+ underscore@1.8.2
added 1 package in 1.574s

$ npm list
project@1.0.0 /home/sitepoint/project
└── underscore@1.8.2
```

Updating a Package

Let's check if there's an update for the Underscore package:

```
$ npm outdated
Package      Current  Wanted  Latest  Location
underscore   1.8.2    1.8.3   1.8.3   project
```

The *Current* column shows us the version that is installed locally. The *Latest* column tells us the latest version of the package. And the *Wanted* column tells us the latest version of the package we can upgrade to without breaking our existing code.

Remember the `package-lock.json` file from earlier? Introduced in npm v5, the purpose of this file is to ensure that the dependencies remain the same on all machines the project is installed on. It is automatically generated for any operations where npm modifies either the `node_modules` folder, or `package.json` file.

You can go ahead and try this out if you like. Delete the `node_modules` folder, then re-run `npm i`. The latest version of npm will install Underscore v1.8.2 (as this is what is specified in the `package-lock.json` file). Earlier versions will pull in v1.8.3 due to the rules of semantic versioning. In the past inconsistent package versions have proven a big headache for developers. This was normally solved by using an `npm-shrinkwrap.json` file which had to be manually created.

Now, let's assume the latest version of Underscore fixed the bug we had earlier and we want to update our package to that version.

```
$ npm update underscore
+ underscore@1.8.3
updated 1 package in 0.236s

$ npm list
project@1.0.0 /home/sitepoint/project
└── underscore@1.8.3
```

Underscore has to be listed as a dependency in `package.json`

For this to work, Underscore has to be listed as a dependency in `package.json`. We can also execute `npm update` if we have many outdated modules we want to update.

Searching for Packages

We've used the `mkdir` command a couple of times in this tutorial. Is there a node package that does the same? Let's use `npm search`.

```
$ npm search mkdir
NAME      | DESCRIPTION          | AUTHOR        | DATE      | VE
mkdir     | Directory crea...    | =joehewitt   | 2012-04-17 | 0.
fs-extra  | fs-extra conta...  | =jprichardson... | 2017-05-04 | 3.
mkdirp    | Recursively mkdir,... | =substack   | 2015-05-14 | 0.
...
```

There is ([mkdirp](#)). Let's install it.

```
$ npm install mkdirp
+ mkdirp@0.5.1
added 2 packages in 3.357s
```

Now create a file `mkdir.js` and copy-paste this code:

```
const mkdirp = require('mkdirp');
mkdirp('foo', function (err) {
  if (err) console.error(err)
  else console.log('Directory created!')
});
```

And run it from the terminal:

```
$ node mkdir.js
Directory created!
```

Re-installing Project Dependencies

Let's first install one more package:

```
$ npm install request
+ request@2.81.0
added 54 packages in 15.92s
```

Check the package.json.

```
"dependencies": {
  "mkdirp": "^0.5.1",
  "request": "^2.81.0",
  "underscore": "^1.8.2"
},
```

Note the dependencies list got updated automatically. In previous versions of npm, you would have had to execute `npm install request --save` to save the dependency in `package.json`. If you wanted to install a package without saving it in `package.json`, just use `--no-save` argument.

Let's assume you have cloned your project source code to another machine and we want to install the dependencies. Let's delete the `node_modules` folder first then execute `npm install`

```
$ rm -R node-modules
$ npm list
project@1.0.0 /home/sitepoint/project
├── UNMET DEPENDENCY mkdirp@^0.5.1
├── UNMET DEPENDENCY request@^2.81.0
└── UNMET DEPENDENCY underscore@^1.8.2

npm ERR! missing: mkdirp@^0.5.1, required by project@1.0.0
npm ERR! missing: request@^2.81.0, required by project@1.0.0
npm ERR! missing: underscore@^1.8.2, required by project@1.0.0

$ npm install
added 57 packages in 1.595s
```

If you look at your `node_modules` folder, you'll see that it gets recreated again. This way, you can easily share your code with others without bloating your project and source repositories with dependencies.

Managing the Cache

When npm installs a package it keeps a copy, so the next time you want to install that package, it doesn't need to hit the network. The copies are cached in the `.npm` directory in your home path.

```
$ ls ~/.npm  
anonymous-cli-metrics.json  _cacache  _locks  npm  registry.npmjs.or
```

This directory will get cluttered with old packages over time, so it's useful to clean it up occasionally.

```
$ npm cache clean
```

You can also purge all `node_module` folders from your workspace if you have multiple node projects on your system you want to clean up.

```
find . -name "node_modules" -type d -exec rm -rf '{}' +
```

Aliases

As you may have noticed, there are multiple ways of running npm commands. Here is a brief list of some of the commonly used npm aliases:

- `npm i <package>` - install local package
- `npm i -g <package>` - install global package
- `npm un <package>` - uninstall local package
- `npm up` - npm update packages
- `npm t` - run tests
- `npm ls` - list installed modules
- `npm ll` or `npm la` - print additional package information while listing modules

You can also install multiple packages at once like this:

```
$ npm i express moment lodash mongoose body-parser webpack
```

If you want to learn all common npm commands, just execute `npm help` for the full list. You can also learn more in our article [10 Tips and Tricks That Will Make You an npm Ninja](#).

Version Managers

There are a couple of tools available that allow you to manage multiple versions of Node.js on the same machine. One such tool is [n](#). Another such tool is [nvm](#) (Node Version Manager). If this is something you're interested in, why not check out our tutorial: [Install Multiple Versions of Node.js using nvm](#).

Conclusion

In this tutorial, I have covered the basics of working with npm. I have demonstrated how to install Node.js from the project's download page, how to alter the location of global packages (so we can avoid using sudo) and how to install packages in local and global mode. I also covered deleting, updating and installing a certain version of a package, as well as managing a project's dependencies. If you would like to learn more about the new features in the latest releases, you can visit the [npm Github releases page](#).

With version 5, npm is making huge strides into the world of front-end development. [According to its COO](#), its user base is changing and most of those using it are not using it to write Node at all. Rather it's becoming a tool that people use to put JavaScript together on the frontend (seriously, you can use it to install just about anything) and one which is becoming an integral part of writing modern JavaScript. Are you using npm in your projects? If not, now might be a good time to start.

Chapter 4: Forms, File Uploads and Security with Node.js and Express

by Mark Brown

If you're building a web application, you're likely to encounter the need to build HTML forms on day one. They're a big part of the web experience, and they can be complicated.

Typically the form handling process involves:

- displaying an empty HTML form in response to an initial GET request
- user submitting the form with data in a POST request
- validation on both the client and the server
- re-displaying the form populated with escaped data and error messages if invalid
- doing *something* with the sanitized data on the server if it's all valid
- redirecting the user or showing a success message after data is processed.

Handling form data also comes with extra security considerations.

We'll go through all of these and explain how to build them with Node.js and Express — the most popular web framework for Node. First, we'll build a simple contact form where people can send a message and email address securely and then take a look what's involved in processing file uploads.

Oops, please correct the following:

- That email doesn't look right

Send us a message

MESSAGE

EMAIL

Send

MESSAGE

EMAIL

That email doesn't look right

Send

Setup

Make sure you've got a recent version of Node.js installed; `node -v` should return `8.9.0` or higher.

Download the starting code from here with git:

```
git clone https://github.com/sitepoint-editors/node-forms.git
cd node-forms
npm install
npm start
```

There's not *too much* code in there. It's just a bare-bones Express setup with EJS templates and error handlers:

```
// server.js
const path = require('path')
const express = require('express')
const layout = require('express-layout')
const routes = require('./routes')
const app = express()

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

const middleware = [
  layout(),
  express.static(path.join(__dirname, 'public')),
]
app.use(middleware)

app.use('/', routes)

app.use((req, res, next) => {
  res.status(404).send("Sorry can't find that!")
})

app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})

app.listen(3000, () => {
```

```
  console.log(`App running at http://localhost:3000`)
})
```

The root url / simply renders the index.ejs view.

```
// routes.js
const express = require('express')
const router = express.Router()

router.get('/', (req, res) => {
  res.render('index')
})

module.exports = router
```

Displaying the Form

When people make a GET request to /contact, we want to render a new view contact.ejs:

```
// routes.js
router.get('/contact', (req, res) => {
  res.render('contact')
})
```

The contact form will let them send us a message and their email address:

```
<!-- views/contact.ejs -->
<div class="form-header">
  <h2>Send us a message</h2>
</div>
<form method="post" action="/contact" novalidate>
  <div class="form-field">
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" aut
      </textarea>
  </div>
  <div class="form-field">
    <label for="email">Email</label>
    <input class="input" id="email" name="email" type="email" value=
  </div>
  <div class="form-actions">
    <button class="btn" type="submit">Send</button>
  </div>
</form>
```

See what it looks like at <http://localhost:3000/contact>.

Form Submission

To receive POST values in Express, you first need to include the `body-parser` middleware, which exposes submitted form values on `req.body` in your route handlers. Add it to the end of the `middlewares` array:

```
// server.js
const bodyParser = require('body-parser')

const middlewares = [
  // ...
  bodyParser.urlencoded()
]
```

It's a common convention for forms to POST data back to the same URL as was used in the initial GET request. Let's do that here and handle POST /contact to process the user input.

Let's look at the invalid submission first. If invalid, we need to pass back the submitted values to the view so they don't need to re-enter them along with any error messages we want to display:

```
router.get('/contact', (req, res) => {
  res.render('contact', {
    data: {},
    errors: {}
  })
}

router.post('/contact', (req, res) => {
  res.render('contact', {
    data: req.body, // { message, email }
    errors: {
      message: {
        msg: 'A message is required'
      },
      email: {
        msg: 'That email doesn't look right'
      }
    }
  })
})
```

If there are any validation errors, we'll do the following:

- display the errors at the top of the form
- set the input values to what was submitted to the server
- display inline errors below the inputs
- add a `form-field-invalid` class to the fields with errors.

```
<!-- views/contact.ejs -->
<div class="form-header">
  <% if (Object.keys(errors).length === 0) { %>
    <h2>Send us a message</h2>
  <% } else { %>
    <h2 class="errors-heading">Oops, please correct the following:</h2>
    <ul class="errors-list">
      <% Object.values(errors).forEach(error => { %>
        <li><%= error.msg %></li>
      <% }) %>
    </ul>
  <% } %>
</div>

<form method="post" action="/contact" novalidate>
  <div class="form-field <%= errors.message ? 'form-field-invalid' : '' %>">
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" autogrow><%= data.message %></textarea>
    <% if (errors.message) { %>
      <div class="error"><%= errors.message.msg %></div>
    <% } %>
  </div>
  <div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">
    <label for="email">Email</label>
    <input class="input" id="email" name="email" type="email" value="<%= data.email %>" />
    <% if (errors.email) { %>
      <div class="error"><%= errors.email.msg %></div>
    <% } %>
  </div>
  <div class="form-actions">
    <button class="btn" type="submit">Send</button>
  </div>
</form>
```

Submit the form at `http://localhost:3000/contact` to see this in action.
That's everything we need on the view side.

Validation and Sanitization

There is a handy middleware `express-validator` for validating and sanitizing data using the [validator.js](#) library, let's include it in our `middlewares` array:

```
// server.js
const validator = require('express-validator')

const middlewares = [
  // ...
  validator()
]
```

Validation

With the [validators](#) provided we can easily check that a message and a valid email was provided:

```
// routes.js
const { check, validationResult } = require('express-validator/check')

router.post('/contact', [
  check('message')
    .isLength({ min: 1 })
    .withMessage('Message is required'),
  check('email')
    .isEmail()
    .withMessage('That email doesn't look right')
], (req, res) => {
  const errors = validationResult(req)
  res.render('contact', {
    data: req.body,
    errors: errors.mapped()
  })
})
```

Sanitization

With the [sanitizers](#) provided we can trim whitespace from the start and end of the values, and normalize the email into a consistent pattern. This can help remove duplicate contacts being created by slightly different inputs. For example, '

`'Mark@gmail.com'` and `'mark@gmail.com '` would both be sanitized into `'mark@gmail.com'`.

Sanitizers can simply be chained onto the end of the validators:

```
const { matchedData } = require('express-validator/filter')

router.post('/contact', [
  check('message')
    .isLength({ min: 1 })
    .withMessage('Message is required')
    .trim(),
  check('email')
    .isEmail()
    .withMessage('That email doesn't look right')
    .trim()
    .normalizeEmail()
], (req, res) => {
  const errors = validationResult(req)
  res.render('contact', {
    data: req.body,
    errors: errors.mapped()
  })

  const data = matchedData(req)
  console.log('Sanitized:', data)
})
```

The `matchedData` function returns the output of the sanitizers on our input.

The Valid Form

If there are errors we need to re-render the view. If not, we need to do something useful with the data and then show that the submission was successful. Typically, the person is redirected to a success page and shown a message.

HTTP is stateless, so you can't redirect to another page *and* pass messages along without the help of a [session cookie](#) to persist that message between HTTP requests. A “flash message” is the name given to this kind of one-time-only message we want to persist across a redirect and then disappear.

There are three middlewares we need to include to wire this up:

```
const cookieParser = require('cookie-parser')
const session = require('express-session')
const flash = require('express-flash')

const middlewares = [
  // ...
  cookieParser(),
  session({
    secret: 'super-secret-key',
    key: 'super-secret-cookie',
    resave: false,
    saveUninitialized: false,
    cookie: { maxAge: 60000 }
  }),
  flash()
]
```

The `express-flash` middleware adds `req.flash(type, message)` which we can use in our route handlers:

```
// routes
router.post('/contact', [
  // validation ...
], (req, res) => {
  const errors = validationResult(req)
  if (!errors.isEmpty()) {
    return res.render('contact', {
      data: req.body,
      errors: errors.mapped()
    })
  }
})
```

```
        })
    }

const data = matchedData(req)
console.log('Sanitized: ', data)
// Homework: send sanitized data in an email or persist in a db

req.flash('success', 'Thanks for the message! I'll be in touch :)')
res.redirect('/')
})
```

The express-flash middleware adds messages to `req.locals` which all views have access to:

```
<!-- views/index.ejs -->
<% if (messages.success) { %>
  <div class="flash flash-success"><%= messages.success %></div>
<% } %>

<h1>Working With Forms in Node.js</h1>
```

You should now be redirected to index view and see a success message when the form is submitted with valid data. Huzzah! We can now deploy this to production and be sent messages by the prince of Nigeria.

Security considerations

If you're working with forms and sessions on the Internet, you need to be aware of common security holes in web applications. The best security advice I've been given is "Never trust the client!"

TLS over HTTPS

Always use TLS encryption over `https://` when working with forms so that the submitted data is encrypted when it's sent across the Internet. If you send form data over `http://`, it's sent in plain text and can be visible to anyone eavesdropping on those packets as they journey across the Internet.

Wear Your Helmet

There's a neat little middleware called [helmet](#) that adds some security from HTTP headers. It's best to include right at the top of your middlewares and is super easy to include:

```
// server.js
const helmet = require('helmet')

middlewares = [
  helmet()
  // ...
]
```

Cross-site Request Forgery (CSRF)

You can protect yourself against [cross-site request forgery](#) by generating a unique token when the user is presented with a form and then validating that token before the POST data is processed. There's a middleware to help you out here as well:

```
// server.js
const csurf = require('csurf')

middlewares = [
  // ...
]
```

```
    csrf({ cookie: true })
]
```

In the GET request we generate a token:

```
// routes.js
router.get('/contact', (req, res) => {
  res.render('contact', {
    data: {},
    errors: {},
    csrfToken: req.csrfToken()
  })
})
```

And also in the validation errors response:

```
router.post('/contact', [
  // validations ...
], (req, res) => {
  const errors = validationResult(req)
  if (!errors.isEmpty()) {
    return res.render('contact', {
      data: req.body,
      errors: errors.mapped(),
      csrfToken: req.csrfToken()
    })
  }
  // ...
})
```

Then we just need include the token in a hidden input:

```
<!-- view/contact.ejs -->
<form method="post" action="/contact" novalidate>
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  <!-- ... -->
</form>
```

That's all that's required.

We don't need to modify our POST request handler as all POST requests will now require a valid token by the `csurf` middleware. If a valid CSRF token isn't provided, a `ForbiddenError` error will be thrown, which can be handled by the error handler defined at the end of `server.js`.

You can test this out yourself by editing or removing the token from the form with your browser's developer tools and submitting.

Cross-site Scripting (XSS)

You need to take care when displaying user-submitted data in an HTML view as it can open you up to [cross-site scripting\(XSS\)](#). All template languages provide different methods for outputting values. The EJS `<%= value %>` outputs the *HTML escaped* value to protect you from XSS, whereas `<%- value %>` outputs a raw string.

Always use the escaped output `<%= value %>` when dealing with user submitted values. Only use raw outputs when you're sure that's is safe to do so.

File Uploads

Uploading files in HTML forms is a special case that requires an encoding type of "multipart/form-data". See [MDN's guide to sending form data](#) for more detail about what happens with multipart form submissions.

You'll need additional middleware to handle multipart uploads. There's an Express package named `multer` that we'll use here:

```
// routes.js
const multer = require('multer')
const upload = multer({ storage: multer.memoryStorage() })

router.post('/contact', upload.single('photo'), [
  // validation ...
], (req, res) => {
  // error handling ...

  if (req.file) {
    console.log('Uploaded: ', req.file)
    // Homework: Upload file to S3
  }

  req.flash('success', 'Thanks for the message! I\'ll be in touch :)')
  res.redirect('/')
})
```

This code instructs `multer` to upload the file in the “photo” field into memory and exposes the `File` object in `req.file` which we can inspect or process further.

The last thing we need is to add the `enctype` attribute and our file input:

```
<form method="post" action="/contact?_csrf=<%= csrfToken %>">
  →novalidate enctype="multipart/form-data">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  <div class="form-field <%= errors.message ? 'form-field-invalid' : 'form-field-valid'">
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" aut
      →data.message %></textarea>
    <% if (errors.message) { %>
      <div class="error"><%= errors.message.msg %></div>
```

```
<% } %>
</div>
<div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">
  <label for="email">Email</label>
  <input class="input" id="email" name="email" type="email" value="<%- data.email %>" />
  <% if (errors.email) { %>
    <div class="error"><%= errors.email.msg %></div>
  <% } %>
</div>
<div class="form-field">
  <label for="photo">Photo</label>
  <input class="input" id="photo" name="photo" type="file" />
</div>
<div class="form-actions">
  <button class="btn" type="submit">Send</button>
</div>
</form>
```

Unfortunately, we also needed to include `_csrf` as a GET param so that the `csurf` middleware plays ball and doesn't lose track of our token during multipart submissions.

Try uploading a file, you should see the `File` objects logged in the console.

Populating File Inputs

In case of validation errors, we can't re-populate file inputs like we did for the text inputs. A common approach to solving this problem involves these steps:

- uploading the file to a temporary location on the server
- showing a thumbnail and filename of the attached file
- adding JavaScript to the form to allow people to remove the selected file or upload a new one
- moving the file to a permanent location when everything is valid.

Because of the additional complexities of working with multipart and file uploads, they're often kept in separate forms.

Thanks For Reading

I hope you enjoyed learning about HTML forms and how to work with them in Express and Node.js. Here's a quick recap of what we've covered:

- displaying an empty Form in response to a GET request
- processing the submitted POST data
- displaying a list of errors, inline errors and submitted data
- checking submitted data with validators
- cleaning up submitted data with sanitizers
- passing messages across redirects with a flash message
- protecting yourself against attacks like CSRF and XSS
- processing file uploads in multipart form submissions.

Chapter 5: MEAN Stack: Build an App with Angular 2+ and the Angular CLI

by Manjunath M

The MEAN stack comprises advanced technologies used to develop both the server-side and the client-side of a web application in a JavaScript environment. The components of the MEAN stack include the MongoDB database, Express.js (a web framework), Angular (a front-end framework), and the Node.js runtime environment. Taking control of the MEAN stack and familiarizing different JavaScript technologies during the process will help you in becoming a full-stack JavaScript developer.

JavaScript's sphere of influence has dramatically grown over the years and with that growth, there's an ongoing desire to keep up with the latest trends in programming. New technologies have emerged and existing technologies have been rewritten from the ground up (I'm looking at you, Angular).

This tutorial intends to create the MEAN application from scratch and serve as an update to the [original MEAN stack tutorial](#). If you're familiar with MEAN and want to get started with the coding, you can skip to the overview section.

Introduction to the MEAN Stack

Node.js - [Node.js](#) is a server-side runtime environment built on top of Chrome's V8 JavaScript engine. Node.js is based on an event-driven architecture that runs on a single thread and a non-blocking IO. These design choices allow you to build real-time web applications in JavaScript that scale well.

Express.js - [Express](#) is a minimalistic yet robust web application framework for Node.js. Express.js uses middleware functions to handle HTTP requests and then either return a response or pass on the parameters to another middleware. Application-level, router-level, and error-handling middlewares are available in Express.js.

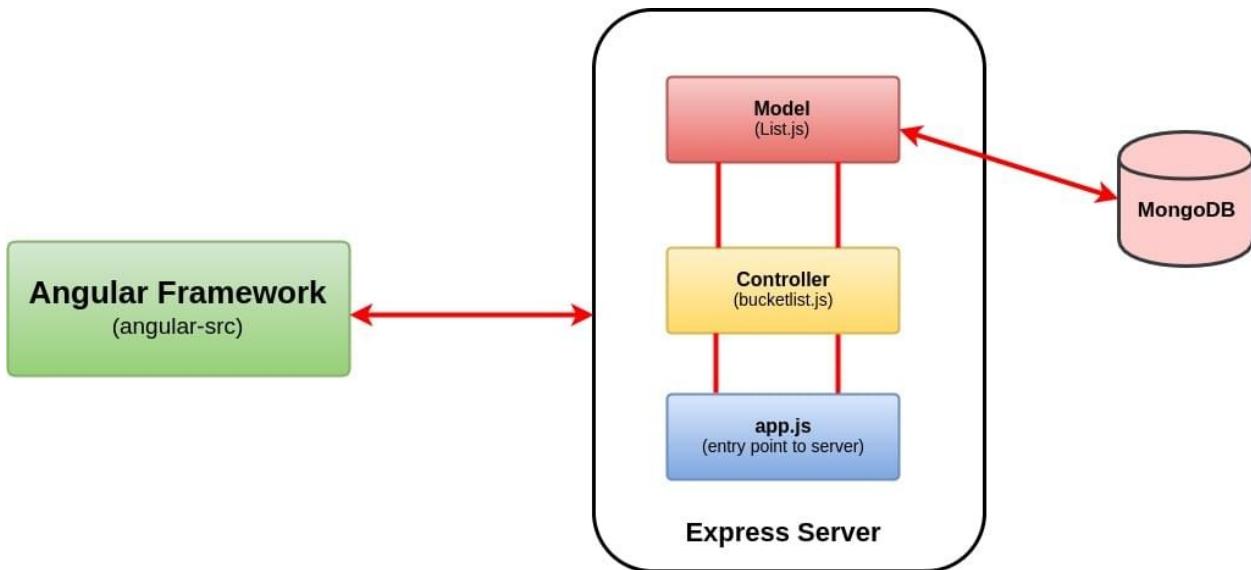
MongoDB - [MongoDB](#) is a document-oriented database program where the documents are stored in a flexible JSON-like format. Being an NoSQL database program, MongoDB relieves you from the tabular jargon of the relational database.

Angular - [Angular](#) is an application framework developed by Google for building interactive Single Page Applications. Angular, originally AngularJS, was rewritten from scratch to shift to a Component-based architecture from the age old MVC framework. Angular recommends the use of TypeScript which, in my opinion, is a good idea because it enhances the development workflow.

Now that we're acquainted with the pieces of the MEAN puzzle, let's see how we can fit them together, shall we?

Overview

Here's a high-level overview of our application.



We'll be building an Awesome Bucket List Application from the ground up without using any boilerplate template. The front end will include a form that accepts your bucket list items and a view that updates and renders the whole bucket list in real time.

Any update to the view will be interpreted as an event and this will initiate an HTTP request. The server will process the request, update/fetch the MongoDB if necessary, and then return a JSON object. The front end will use this to update our view. By the end of this tutorial, you should have a bucket list application that looks like this.

Awesome Bucketlist Application!

Developed by Manjunath

Priority Level	Title	Description	Delete
Medium	Something	I have something to do	<button>DELETE</button>
High	Something else	But I have something else which is more important	<button>DELETE</button>
Low	Irrelevant	I don't care about this one	<button>DELETE</button>

Title	Category	Description	
Eat food	High Priority	Don't forget to eat!	<button>SUBMIT</button>

The entire code for the Bucket List application is available on [GitHub](#).

Prerequisites

First things first, you need to have Node.js and MongoDB installed to get started. If you're entirely new to Node, I would recommend reading the [Beginner's Guide to Node](#) to get things rolling. Likewise, setting up MongoDB is easy and you can check out their [documentation](#) for installation instructions specific to your platform.

```
$ node -v  
# v8.0.0
```

Start the mongo daemon service using the command.

```
sudo service mongod start
```

To install the latest version of Angular, I would recommend using Angular CLI. It offers everything you need to build and deploy your Angular application. If you're not familiar with the Angular CLI yet, make sure you check out [The Ultimate Angular CLI Reference](#).

```
npm install -g @angular/cli
```

Create a new directory for our bucket list project. That's where both the front-end and the back-end code will go.

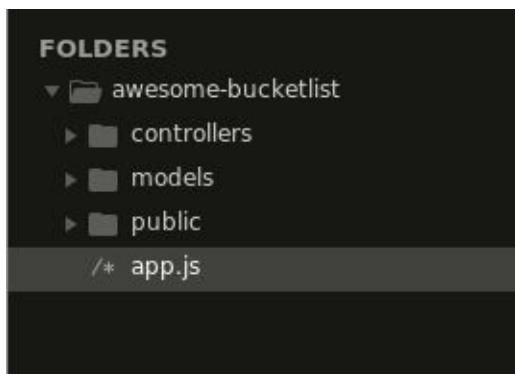
```
mkdir awesome-bucketlist  
cd awesome-bucketlist
```

Creating the Backend Using Express.js and MongoDB

Express doesn't impose any structural constraints on your web application. You can place the entire application code in a single file and get it to work, theoretically. However, your codebase would be a complete mess. Instead, we're going to do this the MVC way (Model, View, and Controller)—minus the view part.

MVC is an architectural pattern that separates your models (the back end) and views (the UI) from the controller (everything in between), hence MVC. Since Angular will take care of the front end for us, we'll have three directories, one for models and another one for controllers, and a public directory where we'll place the compiled angular code.

In addition to this, we'll create an `app.js` file that will serve as the entry point for running the Express server.



Although using a model and controller architecture to build something trivial like our bucket list application might seem essentially unnecessary, this will be helpful in building apps that are easier to maintain and refactor.

Initializing npm

We're missing a package `.json` file for our back end. Type in `npm init` and, after you've answered the questions, you should have a `package.json` made for you.

We'll declare our dependencies inside the package.json file. For this project we'll need the following modules:

- **express**: Express module for the web server
- **mongoose**: A popular library for MongoDB
- **bodyparser**: Parses the body of the incoming requests and makes it available under req.body
- **cors**: CORS middleware enables cross-origin access control to our web server.

I've also added a start script so that we can start our server using `npm start`.

```
{  
  "name": "awesome-bucketlist",  
  "version": "1.0.0",  
  "description": "A simple bucketlist app using MEAN stack",  
  "main": "app.js",  
  "scripts": {  
    "start": "node app"  
  },  
  
  //The ~ is used to match the most recent minor version (without breakage)  
  
  "dependencies": {  
    "express": "~4.15.3",  
    "mongoose": "~4.11.0",  
    "cors": "~2.8.3",  
    "body-parser": "~1.17.2"  
  },  
  
  "author": "",  
  "license": "ISC"  
}
```

Now run `npm install` and that should take care of installing the dependencies.

Filling in app.js

First, we require all of the dependencies that we installed in the previous step.

```
// We'll declare all our dependencies here  
const express = require('express');  
const path = require('path');  
const bodyParser = require('body-parser');
```

```
const cors = require('cors');
const mongoose = require('mongoose');

//Initialize our app variable
const app = express();

//Declaring Port
const port = 3000;
```

As you can see, we've also initialized the `app` variable and declared the port number. The `app` object gets instantiated on the creation of the Express web server. We can now load middleware into our Express server by specifying them with `app.use()`.

```
//Middleware for CORS
app.use(cors());

//Middleware for bodyparsing using both json and urlencoding
app.use(bodyParser.urlencoded({extended:true}));
app.use(bodyParser.json());

/*express.static is a built in middleware function to serve static files
We are telling express public folder is the place to look for the static files*/
app.use(express.static(path.join(__dirname, 'public')));
```

The `app` object can understand routes too.

```
app.get('/', (req,res) => {
  res.send("Invalid page");
})
```

Here, the `get` method invoked on the `app` corresponds to the GET HTTP method. It takes two parameters, the first being the path or route for which the middleware function should be applied.

The second is the actual middleware itself, and it typically takes three arguments: the `req` argument corresponds to the HTTP Request; the `res` argument corresponds to the HTTP Response; and `next` is an optional callback argument that should be invoked if there are other subsequent middlewares that follow this one. We haven't used `next` here since the `res.send()` ends the request-response cycle.

Add this line towards the end to make our app listen to the port that we had declared earlier.

```
//Listen to port 3000
app.listen(port, () => {
  console.log(`Starting the server at port ${port}`);
});
```

npm start should get our basic server up and running.

By default, npm doesn't monitor your files/directories for any changes, and you have to manually restart the server every time you've updated your code. I recommend using nodemon to monitor your files and automatically restart the server when any changes are detected. If you don't explicitly state which script to run, nodemon will run the file associated with the main property in your package.json.

```
npm install -g nodemon
nodemon
```

We're nearly done with our app.js file. What's left to do? We need to

1. connect our server to the database
2. create a controller, which we can then import to our app.js.

Setting up mongoose

Setting up and connecting a database is straightforward with MongoDB. First, create a config directory and a file named database.js to store our configuration data. Export the database URI using module.exports.

```
// 27017 is the default port number.
module.exports = {
  database: 'mongodb://localhost:27017/bucketlist'
}
```

And establish a connection with the database in app.js using mongoose.connect().

```
// Connect mongoose to our database
const config = require('./config/database');
mongoose.connect(config.database);
```

“But what about creating the bucket list database?”, you may ask. The database will be created automatically when you insert a document into a new collection on that database.

Working on the controller and the model

Now let's move on to create our bucket list controller. Create a `bucketlist.js` file inside the **controller** directory. We also need to route all the `/bucketlist` requests to our bucketlist controller (in `app.js`).

```
const bucketlist = require('./controllers/bucketlist');

//Routing all HTTP requests to /bucketlist to bucketlist controller
app.use('/bucketlist', bucketlist);
```

Here's the final version of our `app.js` file.

```
// We'll declare all our dependencies here
const express = require('express');
const path = require('path');
const bodyParser = require('body-parser');
const cors = require('cors');
const mongoose = require('mongoose');
const config = require('./config/database');
const bucketlist = require('./controllers/bucketlist');

//Connect mongoose to our database
mongoose.connect(config.database);

//Declaring Port
const port = 3000;

//Initialize our app variable
const app = express();

//Middleware for CORS
app.use(cors());

//Middlewares for bodyparsing using both json and urlencoding
app.use(bodyParser.urlencoded({extended:true}));
app.use(bodyParser.json());
```

```

/*express.static is a built in middleware function to serve static files
We are telling express server public folder is the place to look for static files

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

app.get('/', (req,res) => {
    res.send("Invalid page");
})

//Routing all HTTP requests to /bucketlist to bucketlist controller
app.use('/bucketlist',bucketlist);

//Listen to port 3000
app.listen(port, () => {
    console.log(`Starting the server at port ${port}`);
});

```

As previously highlighted in the overview, our awesome bucket list app will have routes to handle HTTP requests with GET, POST, and DELETE methods. Here's a bare-bones controller with routes defined for the GET, POST, and DELETE methods.

```

//Require the express package and use express.Router()
const express = require('express');
const router = express.Router();

//GET HTTP method to /bucketlist
router.get('/',(req,res) => {
    res.send("GET");
});

//POST HTTP method to /bucketlist

router.post('/', (req,res,next) => {
    res.send("POST");
});

//DELETE HTTP method to /bucketlist.
//Here, we pass in a params which is the object id.
router.delete('/:id', (req,res,next)=> {
    res.send("DELETE");
}

```

```
})

module.exports = router;
```

I'd recommend using [Postman app](#) or something similar to test your server API. Postman has a powerful GUI platform to make your API development faster and easier. Try a GET request on <http://localhost:3000/bucketlist> and see whether you get the intended response.

And as obvious as it seems, our application lacks a model. At the moment, our app doesn't have a mechanism to send data to and retrieve data from our database.

Create a `list.js` model for our application and define the bucket list Schema as follows:

```
//Require mongoose package
const mongoose = require('mongoose');

//Define BucketlistSchema with title, description and category
const BucketlistSchema = mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    description: String,
    category: {
        type: String,
        required: true,
        enum: ['High', 'Medium', 'Low']
    }
});
```

When working with mongoose, you have to first define a Schema. We have defined a `BucketlistSchema` with three different keys (`title`, `category`, and `description`). Each key and its associated `SchemaType` defines a property in our MongoDB document. If you're wondering about the lack of an `id` field, it's because we'll be using the default `_id` that will be created by Mongoose.

Mongoose Assigns `_id` Fields by Default

Mongoose assigns each of your schemas an `_id` field by default if one is not passed into the Schema constructor. The type assigned is an ObjectId to coincide with MongoDB's default behavior.

You can read more about it in the [Mongoose Documentation](#)

However, to use our Schema definition we need to convert our `BucketlistSchema` to a model and export it using `module.exports`. The first argument of `mongoose.model` is the name of the collection that will be used to store the data in MongoDB.

```
const BucketList = module.exports = mongoose.model('BucketList', BucketlistSchema);
```

Apart from the schema, we can also host database queries inside our `BucketList` model and export them as methods.

```
//BucketList.find() returns all the lists
module.exports.getAllLists = (callback) => {
  BucketList.find(callback);
}
```

Here we invoke the `BucketList.find` method which queries the database and returns the `BucketList` collection. Since a callback function is used, the result will be passed over to the callback.

Let's fill in the middleware corresponding to the GET method to see how this fits together.

```
const bucketlist = require('../models/List');

//GET HTTP method to /bucketlist
router.get('/',(req,res) => {
  bucketlist.getAllLists((err, lists)=> {
    if(err) {
      res.json({success:false, message: `Failed to load all lists
        > ${err}`});
    }
    else {
      res.write(JSON.stringify({success: true, lists:lists}),null);
      res.end();
    }
  });
});
```

```
});
```

We've invoked the `getAllLists` method and the callback takes two arguments, error and result.

All callbacks in Mongoose use the pattern: `callback(error, result)`. If an error occurs executing the query, the error parameter will contain an error document, and result will be null. If the query is successful, the error parameter will be null, and the result will be populated with the results of the query.

-- MongoDB Documentation

Similarly, let's add the methods for inserting a new list and deleting an existing list from our model.

```
//newList.save is used to insert the document into MongoDB
module.exports.addList = (newList, callback) => {
    newList.save(callback);
}

//Here we need to pass an id parameter to BucketList.remove
module.exports.deleteListById = (id, callback) => {
    let query = {_id: id};
    BucketList.remove(query, callback);
}
```

We now need to update our controller's middleware for POST and DELETE also.

```
//POST HTTP method to /bucketlist
router.post('/', (req, res, next) => {
    let newList = new bucketlist({
        title: req.body.title,
        description: req.body.description,
        category: req.body.category
    });
    bucketlist.addList(newList, (err, list) => {
        if(err) {
            res.json({success: false, message: `Failed to create a n
                => ${err}`});
        }
    })
});
```

```

        else
            res.json({success:true, message: "Added successfully."})

    });
});

//DELETE HTTP method to /bucketlist. Here, we pass in a param which
router.delete('/:id', (req,res,next)=> {
    //access the parameter which is the id of the item to be deleted
    let id = req.params.id;
    //Call the model method deleteListById
    bucketlist.deleteListById(id,(err,list) => {
        if(err) {
            res.json({success:false, message: `Failed to delete the
                ➔ ${err}`});
        }
        else if(list) {
            res.json({success:true, message: "Deleted successfully"})
        }
        else
            res.json({success:false});
    })
});

```

With this, we have a working server API that lets us create, view, and delete the bucket list. You can confirm that everything is working as intended by using Postman.

The screenshot shows the Postman application interface. At the top, there is a header with 'POST' dropdown, URL 'localhost:3000/bucketlist', 'Params' button, and a large blue 'Send' button. Below the header, tabs include 'Authorization', 'Headers (3)', 'Body' (which is selected), 'Pre-request Script', and 'Tests'. Under 'Body', options for 'form-data', 'x-www-form-urlencoded' (selected), 'raw', and 'binary' are shown. A table lists three key-value pairs: 'title' with value 'First Project', 'description' with value 'Something goes here', and 'category' with value 'High'. A 'New key' row is present with a 'Value' field and a 'Description' field. Below the table, tabs for 'Body', 'Cookies', 'Headers (7)', and 'Tests' are visible, along with a status indicator 'Status: 200 OK'. The bottom section shows a JSON response with four numbered lines:

```
1 {  
2   "success": true,  
3   "message": "Added successfully."  
4 }
```

We'll now move on to the front end of the application using Angular.

Building the Front End Using Angular

Let's generate the front-end Angular application using the Angular CLI tool that we set up earlier. We'll name it `angular-src` and place it under the `awesome-bucketlist` directory.

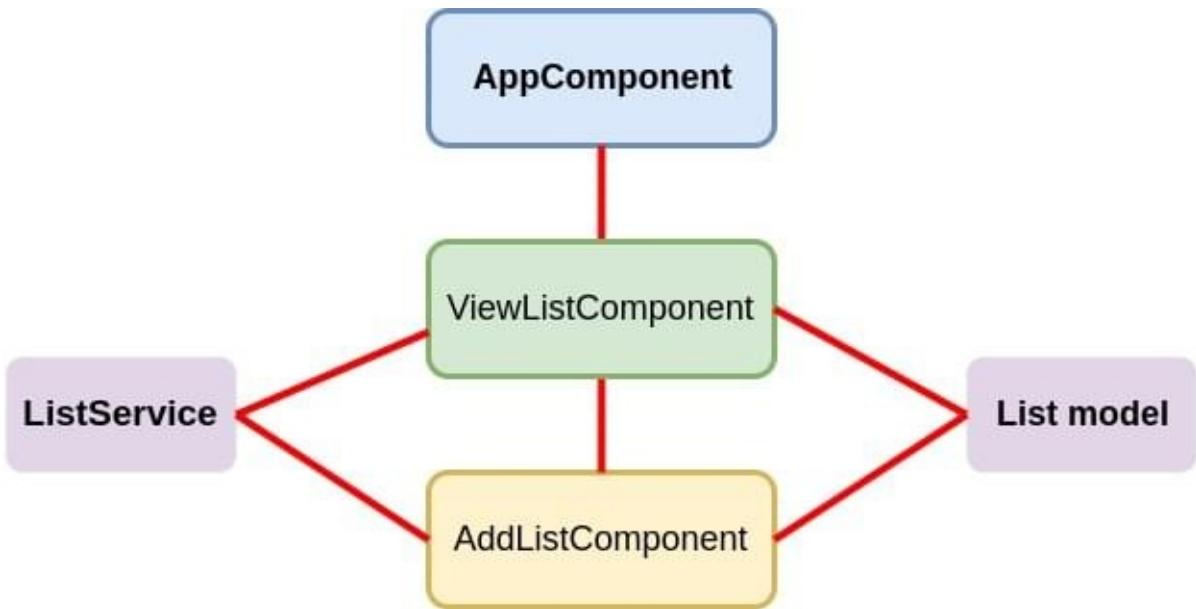
```
ng new angular-src
```

We now have the entire Angular 2 structure inside our `awesome-bucketlist` directory. Head over to the `.angular-cli.json` and change the 'outDir' to `"./public"`.

The next time you run `ng build` — which we'll do towards the end of this tutorial — Angular will compile our entire front end and place it in the `public` directory. This way, you'll have the Express server and the front end running on the same port.

But for the moment, `ng serve` is what we need. You can check out the boilerplate Angular application over at <http://localhost:4200>.

The directory structure of our Angular application looks a bit more complex than our server's directory structure. However, 90% of the time we'll be working inside the `src/app/` directory. This will be our work space, and all of our components, models, and services will be placed inside this directory. Let's have a look at how our front end will be structured by the end of this tutorial.



Creating Components, a Model, and a Service

Let's take a step-by-step approach to coding our Angular application. We need to:

1. create two new components called `ViewListComponent` and `AddListComponent`
2. create a model for our `List`, which can then be imported into our components and services
3. generate a service that can handle all the HTTP requests to the server
4. update the `AppModule` with our components, service, and other modules that may be necessary for this application.

You can generate components using the `ng generate component` command.

```
ng generate component AddList
ng generate component ViewList
```

You should now see two new directories under the `src/app` folder, one each for our newly created components. Next, we need to generate a service for our `List`.

```
ng generate service List
```

I prefer having my services under a new directory(inside `src/app/`).

```
mkdir services
mv list.service.ts services/
```

Since we've changed the location of `list.service.ts`, we need to update it in our `AppModule`. In short, `AppModule` is the place where we'll declare all our components, services, and other modules.

The generate command has already added our components into the `appModule`. Go ahead and import `ListService` and add it to the `providers` array. We also need to import `FormsModule` and `HTTPModule` and declare them as imports. `FormsModule` is needed to create the form for our application and `HTTPModule` for sending HTTP requests to the server.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
```

```

import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

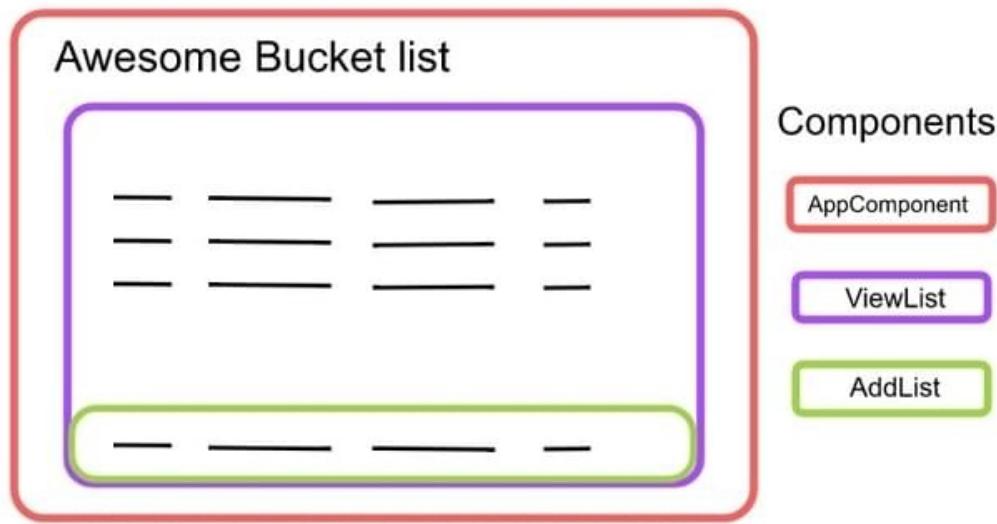
import { AddListComponent } from './add-list/add-list.component';
import { ViewListComponent } from './view-list/view-list.component';
import { ListService } from './services/list.service';
@NgModule({
  declarations: [
    AppComponent,
    AddListComponent,
    ViewListComponent
  ],
  //Modules go here
  imports: [
    BrowserModule,
    HttpModule,
    FormsModule
  ],
  //All the services go here
  providers: [ListService],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Now we are in a position to get started with our components. Components are the building blocks in an Angular 2 application. The `AppComponent` is the default component created by Angular. Each component consists of:

- a TypeScript class that holds the component logic
- an HTML file and a stylesheet that define the component UI
- an `@Component` decorator, which is used to define the metadata of the component.

We'll keep our `AppComponent` untouched for the most part. Instead, we'll use the two newly created components, `AddList` and `ViewList`, to build our logic. We'll nest them inside our `AppComponent` as depicted in the image below.



We now have a hierarchy of components — the `AppComponent` at the top, followed by `ViewListComponent` and then `AddListComponent`.

```
/*app.component.html*/
<!--The whole content below can be removed with the new code.-->
<div style="text-align:center">
  <h1>
    {{title}}!
  </h1>

  <app-view-list> </app-view-list>

</div>

/*view-list.component.html*/
<app-add-list> </app-add-list>
```

Create a file called `List.ts` under the **models** directory. This is where we'll store the model for our List.

```
/* List.ts */
export interface List {
  _id?: string;
  title: string;
```

```
    description: string;
    category: string;

}
```

View-List Component

The `ViewListComponent`' component's logic includes:

1. `lists` property that is an array of `List` type. It maintains a copy of the lists fetched from the server. Using Angular's binding techniques, component properties are accessible inside the template.
2. `loadLists()` loads all the lists from the server. Here, we invoke `this.ListSev.getAllLists()` method and subscribe to it. `getAllLists()` is a service method (we haven't defined it yet) that performs the actual `http.get` request and returns the list; `loadLists()` then loads it into the Component's `list` property.
3. `deleteList(list)` handles the deletion procedure when the user clicks on the *Delete* button. We'll call the List service's `deleteList` method with `id` as the argument. When the server responds that the deletion is successful, we call the `loadLists()` method to update our view.

```
/*view-list.component.ts*/

import { Component, OnInit } from '@angular/core';
import { ListService } from '../services/list.service';
import { List } from '../models/List'

@Component({
  selector: 'app-view-list',
  templateUrl: './view-list.component.html',
  styleUrls: ['./view-list.component.css']
})
export class ViewListComponent implements OnInit {

  //lists property which is an array of List type
  private lists: List[] = [];

  constructor(private listServ: ListService) { }

  ngOnInit() {
    //Load all list on init
    this.listServ.getAllLists().subscribe(lists => this.lists = lists);
  }
}
```

```

        this.loadLists();
    }

public loadLists() {

    //Get all lists from server and update the lists property
    this.listServ.getAllLists().subscribe(
        response => this.lists = response,
    )

    //deleteList. The deleted list is being filtered out using the .fi
    public deleteList(list: List) {
        this.listServ.deleteList(list._id).subscribe(
            response =>      this.lists = this.lists.filter(lists => lists !
        )
    }

}

```

The template (`view-list.component.html`) should have the following code:

```

<h2> Awesome Bucketlist App </h2>

<!-- Table starts here -->
<table id="table">
    <thead>
        <tr>
            <th>Priority Level</th>
            <th>Title</th>
            <th>Description</th>
            <th> Delete </th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let list of lists">
            <td>{{list.category}}</td>
            <td>{{list.title}}</td>
            <td>{{list.description}}</td>
            <td> <button type="button" (click)="deleteList(list); $eve
                =>stopPropagation();">Delete</button></td>
        </tr>
    </tbody>
</table>

<app-add-list> </app-add-list>

```

We've created a table to display our lists. There's a bit of unusual code in there that is not part of standard HTML. Angular has a rich template syntax that adds a bit of zest to your otherwise plain HTML files. The following is part of the Angular template syntax.

- The `*ngFor` directive lets you loop through the `lists` property.
- Here `list` is a template variable whereas `lists` is the component property.
- We have then used Angular's interpolation syntax `{{ }}` to bind the component property with our template.
- The event binding syntax is used to bind the click event to the `deleteList()` method.

We're close to having a working bucket list application. Currently, our `list.service.ts` is blank and we need to fill it in to make our application work. As highlighted earlier, services have methods that communicate with the server.

```
/*list.service.ts*/  
  
import { Injectable } from '@angular/core';  
import { Http, Headers } from '@angular/http';  
import { Observable } from 'rxjs/Observable';  
import { List } from '../models>List'  
  
import 'rxjs/add/operator/map';  
  
@Injectable()  
export class ListService {  
  
    constructor(private http: Http) {}  
  
    private serverApi = 'http://localhost:3000';  
  
    public getAllLists(): Observable<List[]> {  
  
        let URI = `${this.serverApi}/bucketlist/`;  
        return this.http.get(URI)  
            .map(res => res.json())  
            .map(res => <List[]>res.lists);  
    }  
  
    public deleteList(listId: string) {  
        let URI = `${this.serverApi}/bucketlist/${listId}`;  
        let headers = new Headers;  
        headers.append('Content-Type', 'application/json');
```

```

        return this.http.delete(URI, {headers})
          .map(res => res.json());
    }
}

```

The underlying process is fairly simple for both the methods:

1. we build a URL based on our server address
2. we create new headers and append them with { Content-Type: application/json }
3. we perform the actual http.get/http.delete on the URL
4. We transform the response into json format.

If you're not familiar with writing services that communicate with the server, I would recommend reading the tutorial on [Angular and RxJS: Create an API Service to Talk to a REST Backend](#).

Head over to <http://localhost:4200/> to ensure that the app is working. It should have a table that displays all the lists that we have previously created.

Add-List Component

We're missing a feature, though. Our application lacks a mechanism to add/create new lists and automatically update the `ViewListComponent` when the list is created. Let's fill in this void.

The `AddListComponent`'s template is the place where we'll put the code for our HTML form.

```

<div class="container">

  <form (ngSubmit)="onSubmit()">
    <div>
      <label for="title">Title</label>
      <input type="text" [(ngModel)]=" newList.title" name="title"
      </div>

    <div>
      <label for="category">Select Category</label>
      <select [(ngModel)]=" newList.category" name = "category" >

        <option value="High">High Priority</option>

```

```

        <option value="Medium">Medium Priority</option>
        <option value="Low">Low Priority</option>

    </select>
</div>

<div>
    <label for="description">description</label>
    <input type="text" [(ngModel)]="newList.description" name="d
    => required>
</div>

    <button type="submit">Submit</button>

</form>
</div>

```

Inside our template, you can see several instances of `[(ngModel)]` being used. The weird-looking syntax is a directive that implements two-way binding in Angular. Two-way binding is particularly useful when you need to update the component properties from your view and vice versa.

We use an event-binding mechanism (`ngSubmit`) to call the `onSubmit()` method when the user submits the form. The method is defined inside our component.

```

/*add-list.component.ts*/

import { Component, OnInit } from '@angular/core';
import { List } from '../models>List';
import { ListService } from '../services/list.service';

@Component({
  selector: 'app-add-list',
  templateUrl: './add-list.component.html',
  styleUrls: ['./add-list.component.css']
})
export class AddListComponent implements OnInit {
  private newList :List;

  constructor(private listServ: ListService) { }

  ngOnInit() {
    this.newList = {
      title: '',
      category:'',

```

```

        description:'',
        _id:''
    }

}

public onSubmit() {
    this.listServ.addList(this newList).subscribe(
        response=> {
            if(response.success== true)
                //If success, update the view-list component
        },
    );
}

}

```

Inside `onSubmit()`, we call the `listService`'s `addList` method that submits an `http.post` request to the server. Let's update our `list service` to make this happen.

```

/*list.service.ts*/

public addList(list: List) {
    let URI = `${this.serverApi}/bucketlist/`;
    let headers = new Headers();
    let body = JSON.stringify({title: list.title, description:
        list.description, category: list.category});
    console.log(body);
    headers.append('Content-Type', 'application/json');
    return this.http.post(URI, body ,{headers: headers})
        .map(res => res.json());
}

```

If the server returns `{ success: true }`, we need to update our lists and incorporate the new list into our table.

However, the challenge here is that the `lists` property resides inside the `ViewList` component. We need to notify the parent component that the list needs to be updated via an event. We use `EventEmitter` and the `@Output` decorator to make this happen.

First, you need to import `Output` and `EventEmitter` from `@angular/core`.

```
import { Component, OnInit, Output, EventEmitter } from '@angular/core'
```

Next, declare EventEmitter with the @Output decorator.

```
@Output() addList: EventEmitter<List> = new EventEmitter<List>();
```

If the server returns success: true, emit the addList event.

```
public onSubmit() {
    console.log(this.newList.category);
    this.listServ.addList(this.newList).subscribe(
        response=> {
            console.log(response);
            if(response.success== true)
                this.addList.emit(this.newList);
        },
    );
}
```

Update your viewlist.component.html with this code.

```
<app-add-list (addList)='onAddList($event)'> </app-add-list>
```

And finally, add a method named onAddList() that concatenates the newly added list into the lists property.

```
public onAddList(newList) {
    this.lists = this.lists.concat(newList);
}
```

Finishing Touches

I've added some styles from bootswatch.com to make our bucket list app look awesome.

Build your application using:

```
ng build
```

As previously mentioned, the build artifacts will be stored in the public directory. Run `npm start` from the root directory of the MEAN project. You should now have a working MEAN stack application up and running at <http://localhost:3000/>

Wrapping It Up

We've covered a lot of ground in this tutorial, creating a MEAN stack application from scratch. Here's a summary of what we did in this tutorial. We:

- created the back end of the MEAN application using Express and MongoDB
- wrote code for the GET/POST and DELETE routes
- generated a new Angular project using Angular CLI
- designed two new components, `AddList` and `ViewList`
- implemented the application's service that hosts the server communication logic.

Chapter 6: Debugging JavaScript with the Node Debugger

by Camilo Reyes

It's a trap! You've spent a good amount of time making changes, nothing works. Perusing through the code shows no signs of errors. You go over the logic once, twice or thrice, and run it a few times more. Even unit tests can't save you now, they too are failing. This feels like staring at an empty void without knowing what to do. You feel alone, in the dark, and starting to get pretty angry.

A natural response is to throw code quality out and litter everything that gets in the way. This means sprinkling a few print lines here and there and hope something works. This is shooting in pitch black and you know there isn't much hope.

Does this sound all too familiar? If you've ever written more than a few lines of JavaScript, you may have experienced this darkness. There will come a time when a scary program will leave you in an empty void. At some point, it is not smart to face peril alone with primitive tools and techniques. If you are not careful, you'll find yourself wasting hours to identify trivial bugs.

The better approach is to equip yourself with good tooling. A good debugger shortens the feedback loop and makes you more effective. The good news is Node has a very good one out of the box. The Node debugger is versatile and works with any chunk of JavaScript.

Below are strategies that have saved me from wasting valuable time in JavaScript.

The Node CLI Debugger

The [Node debugger command line](#) is a useful tool. If you are ever in a bind and can't access a fancy editor, for any reason, this will help. The tooling uses a [TCP-based protocol](#) to debug with the debugging client. The command line client accesses the process via a port and gives you a debugging session.

You run the tool with `node debug myScript.js`, notice the `debug` flag between the two. Here are a few commands I find you must memorize:

- `sb('myScript.js', 1)` set a breakpoint on first line of your script
- `c` continue the paused process until you hit a breakpoint
- `repl` open the debugger's Read-Eval-Print-Loop (REPL) for evaluation

Don't Mind the Entry Point

When you set the initial breakpoint, one tip is that it's not necessary to set it at the entry point. Say `myScript.js`, for example, requires `myOtherScript.js`. The tool lets you set a breakpoint in `myOtherScript.js` although it is not the entry point.

For example:

```
// myScript.js
var otherScript = require('./myOtherScript');

var aDuck = otherScript();
```

Say that other script does:

```
// myOtherScript.js
module.exports = function myOtherScript() {
  var dabbler = {
    name: 'Dabbler',
    attributes: [
      { inSeaWater: false },
      { canDive: false }
    ]
};
```

```
    return dabbler;
};
```

If `myScript.js` is the entry point, don't worry. You can still set a breakpoint like this, for example, `sb('myOtherScript.js', 10)`. The debugger does not care that the other module is not the entry point. Ignore the warning, if you see one, as long as the breakpoint is set right. The Node debugger may complain that the module hasn't loaded yet.

Time for a Demo of Ducks

Time for a demo! Say you want to debug the following program:

```
function getAllDucks() {
  var ducks = { types: [
    {
      name: 'Dabbler',
      attributes: [
        { inSeawater: false },
        { canDive: false }
      ]
    },
    {
      name: 'Eider',
      attributes: [
        { inSeawater: true },
        { canDive: true }
      ]
    }
  ] };

  return ducks;
}

getAllDucks();
```

Using the CLI tooling, this is how you'd do a debugging session:

```
> node debug debuggingFun.js
> sb(18)
> c
> repl
```

Using the commands above it is possible to step through this code. Say, for example, you want to inspect the duck list using the REPL. When you put a

breakpoint where it returns the list of ducks, you will notice:

```
> ducks
{ types:
  [ { name: 'Dabbler', attributes: [Object] },
    { name: 'Eider', attributes: [Object] } ] }
```

The list of attributes for each duck is missing. The reason is the REPL only gives you a shallow view when objects are deeply nested. Keep this in mind as you are spelunking through code. Consider avoiding collections that are too deep. Use variable assignments to break those out into reasonable chunks. For example, assign `ducks.types[0]` to a separate variable in the code. You'll thank yourself later when pressed for time.

For example:

```
var dabbler = {
  name: 'Dabbler',
  attributes: [
    { inSeawater: false },
    { canDive: false }
  ]
};

// ...

var ducks = { types: [
  dabbler,
  // ...
] };
```

Client-Side Debugging

That's right, the same Node tool can debug client-side JavaScript. If you conceptualize this, the Node debugger runs on top of the V8 JavaScript engine. When you feed it plain old vanilla JavaScript, the debugger will just work. There is no crazy magic here, only making effective use of the tool. The Node debugger does one job well, so take advantage of this.

Take a look at the following quick demo:

A List of Ducks

See the Pen [A List of Ducks](#).

If you click on a duck, say an Eider, it will fade out. If you click it once more it will reappear. All part of fancy DOM manipulations, yes? How can you debug this code with the same server-side Node tooling?

Take a peek at the module that makes this happen:

```
// duckImageView.js
var DuckImageView = function DuckImageView() {
};

DuckImageView.prototype.onClick = function onClick(e) {
  var target = e.currentTarget;

  target.className = target.className === 'fadeOut' ? '' : 'fadeOut'
};

// The browser will ignore this
if (typeof module === 'object') {
  module.exports = DuckImageView;
}
```

How Is This Debuggable Through Node?

A Node program can use the code above, for example:

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

```
var DuckImageView = require('./duckImageView');

var event = { currentTarget: {} };

var view = new DuckImageView();
view.onClick(event);

var element = event.currentTarget;

assert.equal(element.className, 'fadeOut', 'Add fadeIn class in ele
```

As long as your JavaScript is not tightly coupled to the DOM, you can debug it anywhere. The Node tooling doesn't care that it is client-side JavaScript and allows this. Consider writing your modules in this way so they are debuggable. This opens up radical new ways to get yourself out of the empty void.

If you've ever spent time staring at an empty void, you know how painful it is to reload JavaScript in a browser. The context switch between code changes and browser reloads is brutal. With every reload, there is the opportunity to waste more time with other concerns. For example, a busted database or caching.

A better approach is to write your JavaScript so it gives you a high level of freedom. This way you can squash big nasty bugs with ease and style. The aim is you keep yourself focused on the task at hand, happy and productive. By decoupling software components, you reduce risk. Sound programming is not only about solving problems but avoiding self-inflicted issues too.

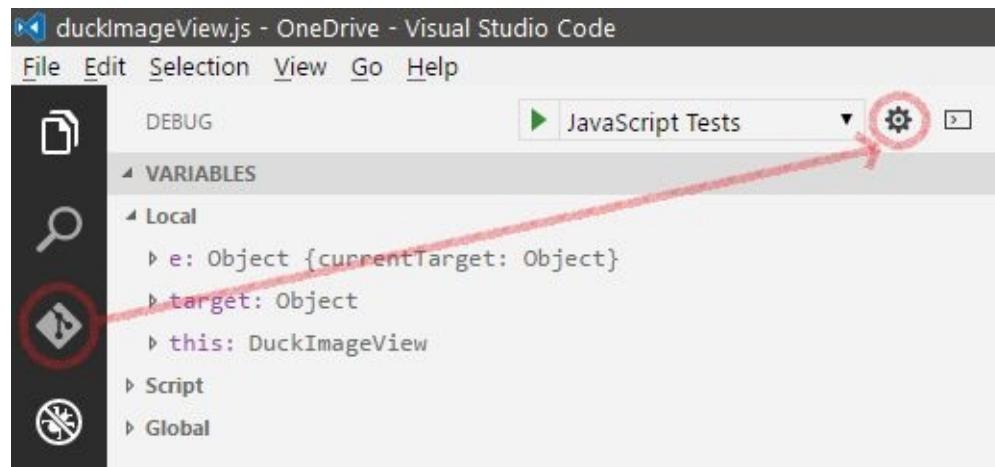
Debugging inside an Editor

Now debugging via the command line is pretty slick, but most developers don't code in it. At least, personally, I'd prefer to spend most of my time inside a code editor. Wouldn't it be nice if the same Node tooling could interact with the editor? The idea is to equip yourself with tooling right where you need it and that means the editor.

There are many editors out there and I can't cover all of them here. The tool that you pick needs to make debugging accessible and easy. If you ever find yourself stuck, it's valuable to be able to hit a shortcut key and invoke a debugger. Knowing how the computer evaluates your code as you write it is important. As for me, there is one editor that stands out as a good tool for debugging JavaScript.

[Visual Studio Code](#) is one tool I recommend for debugging JavaScript. It uses the same debugging protocol the command line tooling uses. It supports a shortcut key (F5 on both Windows and Mac), inspections, everything you expect from a good debugger.

If you already have VS Code installed and haven't played with the debugger, do yourself a favor. Click on the debugging tab on the left and click the gear button:



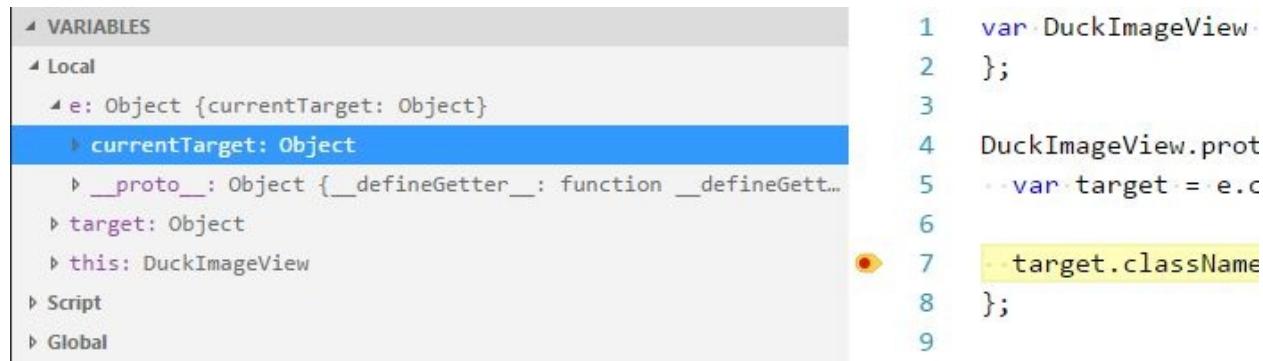
A `launch.json` file will open up. This allows you to configure the debugging entry point, for example:

```
{
```

```
"type": "node",
"request": "launch",
"name": "JavaScript Tests",
"program": "${workspaceRoot}\\entryPoint.js",
// Point this to the same folder as the entry point
"cwd": "${workspaceRoot}"
}
```

At a high level, you tell VS Code what to run and where. The tool supports both npm and Node entry points.

Once it is set up, set a breakpoint, hit a shortcut key and done:



The Node debugger enables you to inspect variables and step through the code. The tooling is happy to reveal what happens to the internals when it evaluates changes. All part of the necessary equipment to destroy nasty bugs.

The same principles apply across the board. Even if VS Code, for example, is not your tool of choice. You tell the tooling what to run and where. You set a breakpoint, hit a shortcut key, and get dropped into a debugging session.

Debugging Transpiled JavaScript

The same Node tooling supports transpiled JavaScript through npm packages. Each language has its own set of tools. One gotcha is each language is very different. TypeScript, for example, has different debugging tools from other transpilers. Node debugging with transpilers boils down to your framework choices and tools.

One idea is to pick the tool that integrates with your workflow. Stay close to where you are making changes and shorten the feedback loop. Give yourself the capability to set a breakpoint and hit it in less than a second.

Editors make use of source maps for debugging, so consider enabling this.

Anything short of a quick feedback loop is only asking for punishment. Your tool choices must not get in the way of sound debugging practices.

Conclusion

I can't discourage enough the use of `console.log()` for debugging. Often I find myself in panic mode when I choose this route. It feels like shooting in the dark.

If the analogy holds true, shots fired at random can ricochet off walls and hurt you or cause friendly fire. Littering the code base with a ton of logging commands can confuse you or the next person to look at the code. I find debugging lines from a previous commit say nothing about nothing and only obfuscate the code. JavaScript can also throw exceptions if the variable does not exist, which adds to the maelstrom.

Developers at times make themselves believe they can run programs with their eyes. If one gets philosophical, the naked human eye can lie and it's not a reliable source of truth. Your feeble sense of sight can make you believe whatever it is you want to believe and leave you in the dark.

A good debugger will give you a peek inside what the computer does with your program. This empowers you to write better software that the computer understands. The Node debugger enables you to verify changes and eliminates wishful thinking. It is a tool every good programmer should master.

Chapter 7: Using MySQL with Node.js and the mysql JavaScript Client

by Jay Raj

NoSQL databases are all the rage these days, and probably the preferred back end for Node.js applications. But you shouldn't architect your next project based on what's hip and trendy. The type of database you use should depend on the project's requirements. If your project involves dynamic table creation, real-time inserts etc. then NoSQL is the way to go. But on the other hand, if your project deals with complex queries and transactions, then an SQL database makes much more sense.

In this tutorial, we'll have a look at getting started with the [mysql module](#) — a Node.js driver for MySQL, written in JavaScript. I'll explain how to use the module to connect to a MySQL database, perform the usual CRUD operations, before examining stored procedures and escaping user input.

Quick Start: How to Use MySQL in Node

Maybe you've arrived here looking for a quick leg up. If you're just after a way to get up and running with MySQL in Node in as little time as possible, we've got you covered!

Here's how to use MySQL in Node in 5 easy steps:

1. Create a new project: `mkdir mysql-test && cd mysql-test`
2. Create a `package.json` file: `npm init -y`
3. Install the `mysql` module: `npm install mysql -save`
4. Create an `app.js` file and copy in the snippet below.
5. Run the file: `node app.js`. Observe a “Connected!” message.

```
//app.js

const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'user',
  password: 'password',
  database: 'database name'
});
connection.connect((err) => {
  if (err) throw err;
  console.log('Connected!');
});
```

Installing the mysql Module

Now let's take a closer look at each of those steps. First of all, we're using the command line to create a new directory and navigate to it. Then we're creating a package.json file using the command `npm init -y`. The `-y` flag means that npm will use only defaults and not prompt you for any options.

This step also assumes that you have Node and npm installed on your system. If this is not the case, then check out this SitePoint article to find out how to do that: [Install Multiple Versions of Node.js using nvm](#).

After that, we're installing the [mysql module](#) from npm and saving it as a project dependency. Project dependencies (as opposed to dev-dependencies) are those packages required for the application to run. You can read [more about the differences between the two here](#).

```
mkdir mysql-test
cd mysql-test
npm install mysql -y
```

If you need further help using npm, then be sure to check out [Chapter 3](#), or ask in [our forums](#).

Getting Started

Before we get on to connecting to a database, it's important that you have MySQL installed and configured on your machine. If this is not the case, please consult the [installation instructions on their home page](#).

The next thing we need to do is to create a database and a database table to work with. You can do this using a graphical interface, such as [phpMyAdmin](#), or using the command line. For this article I'll be using a database called sitepoint and a table called employees. Here's a dump of the database, so that you can get up and running quickly, if you wish to follow along:

```
CREATE TABLE employees (
    id int(11) NOT NULL AUTO_INCREMENT,
    name varchar(50),
    location varchar(50),
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=5 ;

INSERT INTO employees (id, name, location) VALUES
(1, 'Jasmine', 'Australia'),
(2, 'Jay', 'India'),
(3, 'Jim', 'Germany'),
(4, 'Lesley', 'Scotland');
```

localhost / localhost / sitepoint

localhost/phpmyadmin/db_sql.php?db=sitepoint&token=2c...

phpMyAdmin

Server: localhost » Database: sitepoint

Structure SQL Search Query Export More

New

information_schema

mysql

performance_schema

phpmyadmin

sitepoint

sys

Run SQL query/queries on database sitepoint:

```
1 CREATE TABLE employees (
2     id int(11) NOT NULL AUTO_INCREMENT,
3     name varchar(50),
4     location varchar(50),
5     PRIMARY KEY (id)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=5;
7
8 INSERT INTO employees (id, name, location) VALUES
9 (1, 'Jasmine', 'Australia'),
10 (2, 'Jay', 'India'),
11 (3, 'Jim', 'Germany'),
12 (4, 'Lesley', 'Scotland');
```

Clear Format Get auto-saved query

Bind parameters

Bookmark this SQL query: []

[Delimiter :] Show this query here again Retain query box
 Rollback when finished Enable foreign key checks Go

Hide query box

✓ # 4 rows affected.

Console

A screenshot of the phpMyAdmin interface. The left sidebar shows a tree view of databases: information_schema, mysql, performance_schema, phpmyadmin, sitepoint (which is selected), and sys. The main area has a title bar "Run SQL query/queries on database sitepoint:" with a refresh icon. Below it is a code editor containing a SQL script to create a 'employees' table and insert four rows of data. Below the code are buttons for "Clear", "Format", and "Get auto-saved query". There is also a checkbox for "Bind parameters" and a field for bookmarking the query. A section at the bottom allows setting a delimiter and checkboxes for "Show this query here again", "Retain query box", "Rollback when finished", and "Enable foreign key checks", with a "Go" button. A message box at the bottom indicates "# 4 rows affected." with a green checkmark icon. At the very bottom is a "Console" tab.

Connecting to the Database

Now, let's create a file called `app.js` in our `mysql-test` directory and see how to connect to MySQL from Node.js.

```
// app.js
const mysql = require('mysql');

// First you need to create a connection to the db
const con = mysql.createConnection({
  host: 'localhost',
  user: 'user',
  password: 'password',
});

con.connect((err) => {
  if(err){
    console.log('Error connecting to Db');
    return;
  }
  console.log('Connection established');
});

con.end((err) => {
  // The connection is terminated gracefully
  // Ensures all previously enqueued queries are still
  // before sending a COM_QUIT packet to the MySQL server.
});
```

Now open up a terminal and enter `node app.js`. Once the connection is successfully established you should be able to see the 'Connection established' message in the console. If something goes wrong (for example you enter the wrong password), a callback is fired, which is passed an instance of the JavaScript Error object (`err`). Try logging this to the console to see what additional useful information it contains.

Using Grunt to Watch the Files for Changes

Running `node app.js` by hand every time we make a change to our code is going to get a bit tedious, so let's automate that. This part isn't necessary to follow along with the rest of the tutorial, but will certainly save you some

keystrokes.

Let's start off by installing a couple of packages:

```
npm install --save-dev grunt grunt-contrib-watch grunt-execute
```

[Grunt](#) is the well-known JavaScript task runner, [grunt-contrib-watch](#) runs a pre-defined task whenever a watched file changes, and [grunt-execute](#) can be used to run the node app.js command.

Once these are installed, create a file called Gruntfile.js in the project root and add the following code.

```
// Gruntfile.js

module.exports = (grunt) => {
  grunt.initConfig({
    execute: {
      target: {
        src: ['app.js']
      }
    },
    watch: {
      scripts: {
        files: ['app.js'],
        tasks: ['execute'],
      },
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-execute');
};
```

Now run `grunt watch` and make a change to `app.js`. Grunt should detect the change and re-run the `node app.js` command.

Executing Queries

Reading

Now that you know how to establish a connection to MySQL from Node.js, let's see how to execute SQL queries. We'll start by specifying the database name (`sitepoint`) in the `createConnection` command.

```
const con = mysql.createConnection({
  host: 'localhost',
  user: 'user',
  password: 'password',
  database: 'sitepoint'
});
```

Once the connection is established we'll use the `connection` variable to execute a query against the database table `employees`.

```
con.query('SELECT * FROM employees', (err, rows) => {
  if(err) throw err;

  console.log('Data received from Db:\n');
  console.log(rows);
});
```

When you run `app.js` (either using `grunt-watch` or by typing `node app.js` into your terminal), you should be able to see the data returned from database logged to the terminal.

```
[ { id: 1, name: 'Jasmine', location: 'Australia' },
  { id: 2, name: 'Jay', location: 'India' },
  { id: 3, name: 'Jim', location: 'Germany' },
  { id: 4, name: 'Lesley', location: 'Scotland' } ]
```

Data returned from the MySQL database can be parsed by simply looping over the `rows` object.

```
rows.forEach( (row) => {
  console.log(` ${row.name} is in ${row.location}`);
});
```

Creating

You can execute an insert query against a database, like so:

```
const employee = { name: 'Winnie', location: 'Australia' };
con.query('INSERT INTO employees SET ?', employee, (err, res) => {
  if(err) throw err;

  console.log('Last insert ID:', res.insertId);
});
```

Note how we can get the ID of the inserted record using the callback parameter.

Updating

Similarly, when executing an update query, the number of rows affected can be retrieved using `result.affectedRows`:

```
con.query(
  'UPDATE employees SET location = ? WHERE ID = ?',
  ['South Africa', 5],
  (err, result) => {
    if (err) throw err;

    console.log(`Changed ${result.changedRows} row(s)`);
  }
);
```

Destroying

Same thing goes for a delete query:

```
con.query(
  'DELETE FROM employees WHERE id = ?',
  [5],
  (err, result) => {
    if (err) throw err;

    console.log(`Deleted ${result.affectedRows} row(s)`);
  }
);
```

Advanced Use

I'd like to finish off by looking at how the mysql module handles stored procedures and the escaping of user input.

Stored Procedures

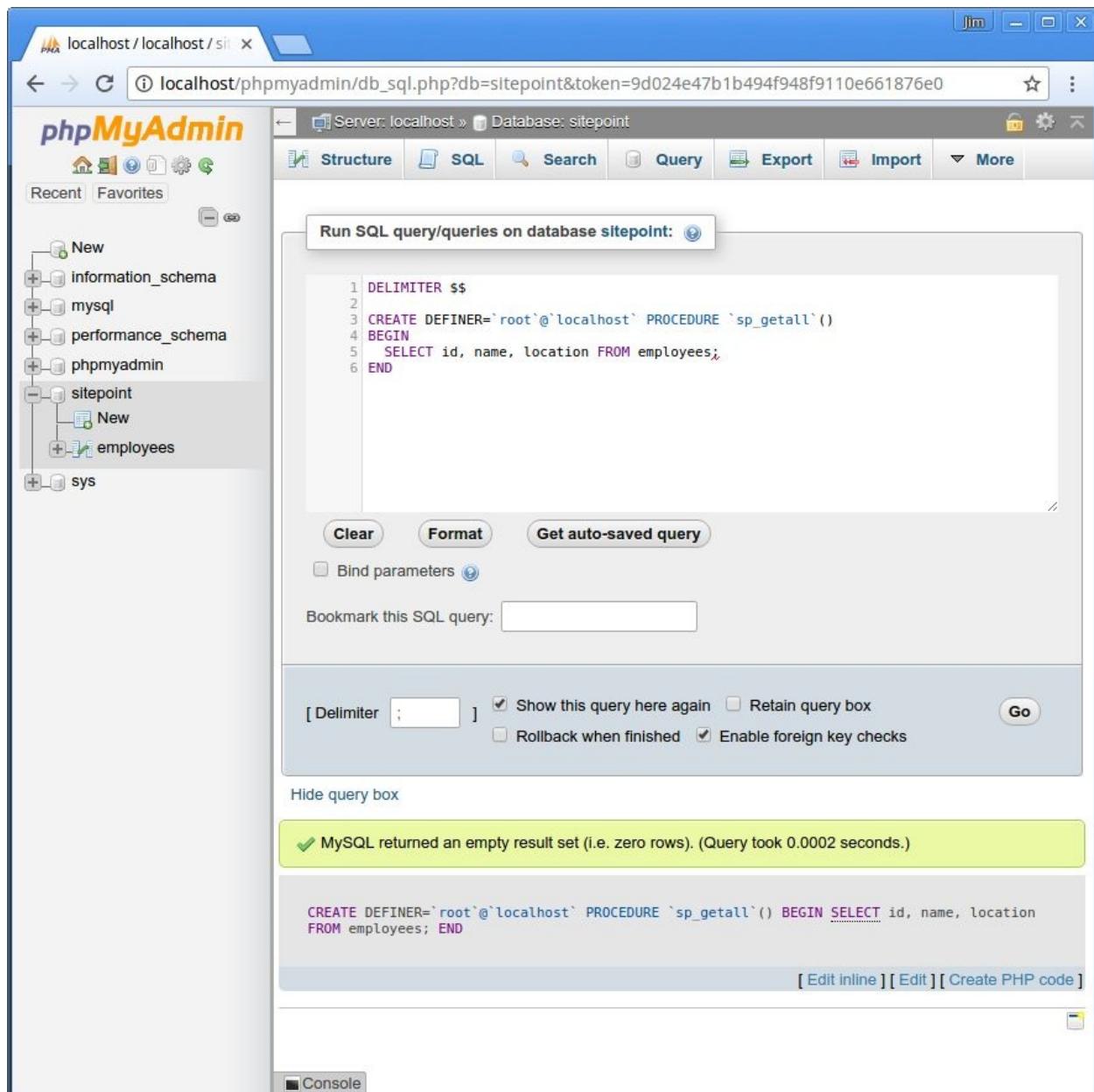
Put simply, a stored procedure is a procedure (written in, for example, SQL) stored in a database which can be called by the database engine and connected programming languages. If you are in need of a refresher, then please check out [this excellent article](#).

Let's create a stored procedure for our sitepoint database which fetches all the employee details. We'll call it `sp_getall`. To do this, you'll need some kind of interface to the database. I'm using [phpMyAdmin](#). Run the following query on the sitepoint database:

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `sp_getall`() 
BEGIN
    SELECT id, name, location FROM employees;
END
```

This will create and store the procedure in the `information_schema` database in the `ROUTINES` table.



Next, establish a connection and use the connection object to call the stored procedure as shown:

```

con.query('CALL sp_getall()', function(err, rows){
  if (err) throw err;

  console.log('Data received from Db:\n');
  console.log(rows);
});

```

Save the changes and run the file. Once executed you should be able to view the

data returned from the database.

```
[ [ { id: 1, name: 'Jasmine', location: 'Australia' },
  { id: 2, name: 'Jay', location: 'India' },
  { id: 3, name: 'Jim', location: 'Germany' },
  { id: 4, name: 'Lesley', location: 'Scotland' } ],
{ fieldCount: 0,
  affectedRows: 0,
  insertId: 0,
  serverStatus: 34,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0 } ]
```

Along with the data, it returns some additional information, such as the affected number of rows, `insertId` etc. You need to iterate over the 0th index of the returned data to get employee details separated from the rest of the information.

```
rows[0].forEach( (row) => {
  console.log(` ${row.name} is in ${row.location}`);
});
```

Now lets consider a stored procedure which requires an input parameter.

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `sp_get_employee_detail`
  in employee_id int
)
BEGIN
  SELECT name, location FROM employees where id = employee_id;
END
```

Now we can pass the input parameter while making a call to the stored procedure:

```
con.query('CALL sp_get_employee_detail(1)', (err, rows) => {
  if(err) throw err;

  console.log('Data received from Db:\n');
  console.log(rows[0]);
});
```

Most of the time when we try to insert a record into the database, we need the

last inserted ID to be returned as an out parameter. Consider the following insert stored procedure with an out parameter:

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `sp_insert_employee`(
    out employee_id int,
    in employee_name varchar(25),
    in employee_location varchar(25)
)
BEGIN
    insert into employees(name, location)
    values(employee_name, employee_location);
    set employee_id = LAST_INSERT_ID();
END
```

To make a procedure call with an out parameter, we first need to enable multiple calls while creating the connection. So, modify the connection by setting the multiple statement execution to true.

```
const con = mysql.createConnection({
  host: 'localhost',
  user: 'user',
  password: 'password',
  database: 'sitepoint',
  multipleStatements: true
});
```

Next when making a call to the procedure, set an out parameter and pass it in.

```
con.query(
  "SET @employee_id = 0; CALL sp_insert_employee(@employee_id, 'Ron'
  → SELECT @employee_id",
  (err, rows) => {
    if (err) throw err;

    console.log('Data received from Db:\n');
    console.log(rows);
  }
);
```

As seen in the above code, we have set an out parameter `@employee_id` and passed it while making a call to the stored procedure. Once the call has been made we need to select the out parameter to access the returned ID.

Run app.js. On successful execution you should be able to see the selected out parameter along with various other information. rows[2] should give you access to the selected out parameter.

```
[ { '@employee_id': 6 } ]
```

Escaping User Input

In order to avoid SQL Injection attacks, you should **always** escape any data from user land before using it inside a SQL query. Let's demonstrate why:

```
const userLandVariable = '4 ';

con.query(
  `SELECT * FROM employees WHERE id = ${userLandVariable}`,
  (err, rows) => {
    if(err) throw err;
    console.log(rows);
  }
);
```

This seems harmless enough and even returns the correct result:

```
{ id: 4, name: 'Lesley', location: 'Scotland' }
```

However, if we change the userLandVariable to this:

```
const userLandVariable = '4 OR 1=1';
```

we suddenly have access to the entire data set. If we then change it to this:

```
const userLandVariable = '4; DROP TABLE employees';
```

then we're in proper trouble!

The good news is that help is at hand. You just have to use the [mysql.escape](#) method:

```
con.query(
  `SELECT * FROM employees WHERE id = ${mysql.escape(userLandVariable)}
  function(err, rows){ ... }
);
```

Or by using a question mark placeholder, as we did in the examples at the beginning of the article:

```
con.query(  
  'SELECT * FROM employees WHERE id = ?',  
  [userLandVariable],  
  (err, rows) => { ... }  
)
```

Why Not Just USE an ORM?

As you may have noticed, a couple of people in the comments are suggesting using an ORM. Before we get into the pros and cons of this approach, let's take a second to look at what ORMs are. The following is taken from [an answer on Stack Overflow](#):

Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the phrase "an ORM".

So basically, this approach means you write your database logic in the domain-specific language of the ORM, as opposed to the vanilla approach we have been taking so far. Here's a contrived example using [Sequelize](#):

```
Employee.findAll().then(employees => {
  console.log(employees);
});
```

Contrasted with:

```
con.query('SELECT * FROM employees', (err, rows) => {
  if(err) throw err;

  console.log('Data received from Db:\n');
  console.log(rows);
});
```

Whether or not using an ORM makes sense for you, will depend very much on what you are working on and with whom. On the one hand, ORMS tend to make developers more productive, in part by abstracting away a large part of the SQL so that not everyone on the team needs to know how to write super efficient database specific queries. It is also easy to move to different database software, because you are developing to an abstraction.

On the other hand however, it is possible to write some really messy and inefficient SQL as a result of not understanding how the ORM does what it does. Performance is also an issue in that it's much easier to optimize queries that don't have to go through the ORM.

Whichever path you take is up to you, but if this is a decision you're in the process of making, check out this Stack Overflow thread: [Why should you use an ORM?](#) as well as this post on SitePoint: [3 JavaScript ORMs You Might Not Know.](#)

Conclusion

In this tutorial, we've only scratched the surface of what the mysql client offers. For more detailed information, I would recommend reading the [official documentation](#). There are other options too, such as [node-mysql2](#) and [node-mysql-libmysqlclient](#).

Chapter 8: How to Use SSL/TLS with Node.js

by **Florian Rappl & Almir Bijedic**

In this article, I'll work through a practical example of how to add a [Let's Encrypt](#)-generated certificate to your Express.js server.

But protecting our sites and apps with HTTPS isn't enough. We should also demand encrypted connections from the servers we're talking to. We'll see that possibilities exist to activate the SSL/TLS layer even if it wouldn't be enabled by default.

Let's start with a short review of HTTPS's current state.

HTTPS Everywhere

The HTTP/2 specification was published as RFC 7540 in May 2015, which means at this point it's a part of the standard. This was a major milestone. Now we can all upgrade our servers to use HTTP/2. One of the most important aspects is the backwards compatibility with HTTP 1.1 and the negotiation mechanism to choose a different protocol. Although the standard doesn't specify mandatory encryption, currently no browser supports HTTP/2 unencrypted. This gives HTTPS another boost. Finally we'll get HTTPS everywhere!

What does our stack actually look like? From the perspective of a website running in the browser (application level) we have roughly the following layers to reach the IP level:

1. Client browser
2. HTTP
3. SSL/TLS
4. TCP
5. IP

HTTPS is nothing more than the HTTP protocol on top of SSL/TLS. Hence all the rules of HTTP still have to apply. What does this additional layer actually give us? There are multiple advantages: we get authentication by having keys and certificates; a certain kind of privacy and confidentiality is guaranteed, as the connection is encrypted in an asymmetric manner; and data integrity is also preserved: that transmitted data can't be changed during transit.

One of the most common myths is that using SSL/TLS requires too many resources and slows down the server. This is certainly not true anymore. We also don't need any specialized hardware with cryptography units. Even for Google, the SSL/TLS layer accounts for less than 1% of the CPU load. Furthermore, the network overhead of HTTPS as compared to HTTP is below 2%. All in all, it wouldn't make sense to forgo HTTPS for having a little bit of overhead.

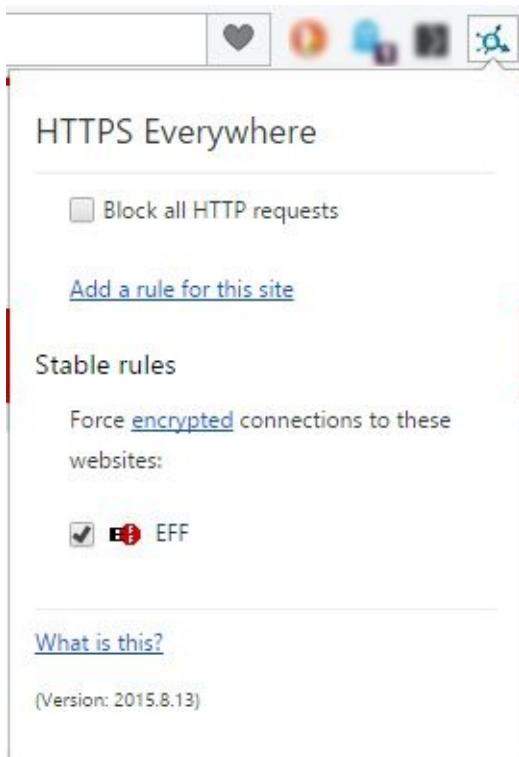
**TLS has exactly one performance problem:
it is not used widely enough.**

Everything else can be optimized.

The most recent version is TLS 1.3. TLS is the successor of SSL, which is available in its latest release SSL 3.0. The changes from SSL to TLS preclude interoperability. The basic procedure is, however, unchanged. We have three different encrypted channels. The first is a public key infrastructure for certificate chains. The second provides public key cryptography for key exchanges. Finally, the third one is symmetric. Here we have cryptography for data transfers.

TLS 1.3 uses hashing for some important operations. Theoretically, it's possible to use any hashing algorithm, but it's highly recommended to use SHA2 or a stronger algorithm. SHA1 has been a standard for a long time but has recently become obsolete.

HTTPS is also gaining more attention for clients. Privacy and security concerns have always been around, but with the growing amount of online accessible data and services, people are getting more and more concerned. A useful browser plugin is "HTTPS Everywhere", which encrypts our communications with most websites.



The creators realized that many websites offer HTTPS only partially. The plugin allows us to rewrite requests for those sites, which offer only partial HTTPS support. Alternatively, we can also block HTTP altogether (see the screenshot above).

Basic Communication

The certificate's validation process involves validating the certificate signature and expiration. We also need to verify that it chains to a trusted root. Finally, we need to check to see if it was revoked. There are dedicated trusted authorities in the world that grant certificates. In case one of these were to become compromised, all other certificates from the said authority would get revoked.

The sequence diagram for a HTTPS handshake looks as follows. We start with the init from the client, which is followed by a message with the certificate and key exchange. After the server sends its completed package, the client can start the key exchange and cipher specification transmission. At this point, the client is finished. Finally the server confirms the cipher specification selection and closes the handshake.



The whole sequence is triggered independently of HTTP. If we decide to use HTTPS, only the socket handling is changed. The client is still issuing HTTP requests, but the socket will perform the previously described handshake and encrypt the content (header and body).

So what do we need to make SSL/TLS work with an Express.js server?

HTTPS

By default, Node.js serves content over HTTP. But there's also an HTTPS module which we have to use in order to communicate over a secure channel with the client. This is a built-in module, and the usage is very similar to how we use the HTTP module:

```
const https = require("https"),
  fs = require("fs");

const options = {
  key: fs.readFileSync("/srv/www/keys/my-site-key.pem"),
  cert: fs.readFileSync("/srv/www/keys/chain.pem")
};

const app = express();

app.use((req, res) => {
  res.writeHead(200);
  res.end("hello world\n");
});

app.listen(8000);

https.createServer(options, app).listen(8080);
```

Ignore the `/srv/www/keys/my-site-key.pem` and `/srv/www/keys/chain.pem` files for now. Those are the SSL certificates we need to generate, which we'll do a bit later. This is the part that changed with Let's Encrypt. Previously, we had to generate a private/public key pair, send it to a trusted authority, pay them and probably wait a bit in order to get an SSL certificate. Nowadays, Let's Encrypt generates and validates your certificates for free and instantly!

Generating Certificates

Certbot

A certificate, which is signed by a trusted certificate authority (CA), is demanded by the TLS specification. The CA ensures that the certificate holder is really who he claims to be. So basically when you see the green lock icon (or any other greenish sign to the left side of the URL in your browser) it means that the server you're communicating with is really who it claims to be. If you're on facebook.com and you see a green lock, it's almost certain you really are communicating with Facebook and no one else can see your communication — or rather, no one else can read it.

It's worth noting that this certificate doesn't necessarily have to be verified by an authority such as Let's Encrypt. There are other paid services as well. You can technically sign it yourself, but then the users visiting your site won't get an approval from the CA when visiting and all modern browsers will show a big warning flag to the user and ask to be redirected "to safety".

In the following example, we'll use the *Certbot*, which is used to generate and manage certificates with Let's Encrypt.

[On the Certbot site](#) you can find instructions on how to install *Certbot* on your OS. Here we'll follow the macOS instructions. In order to install *Certbot*, run

```
brew install certbot
```

Webroot

Webroot is a Certbot plugin that, in addition to the Certbot default functionality which automatically generates your public/private key pair and generates an SSL certificate for those, also copies the certificates to your webroot folder and also verifies your server by placing some verification codes into a hidden temporary directory named `.well-known`. In order to skip doing some of these steps manually, we'll use this plugin. The plugin is installed by default with *Certbot*. In order to generate and verify our certificates, we'll run the following:

```
certbot certonly --webroot -w /var/www/example/ -d www.example.com -
```

You may have to run this command as sudo, as it will try to write to /var/log/letsencrypt.

You'll also be asked for your email address. It's a good idea to put in a real address you use often, as you'll get a notification if your certificate expires is about to expire. The trade off for Let's Encrypt being a free certificate is that it expires every three months. Luckily, renewal is as easy as running one simple command, which we can assign to a cron and then not have to worry about expiration. Additionally, it's a good security practice to renew SSL certificates, as it gives attackers less time to break the encryption. Sometimes developers even set up this cron to run daily, which is completely fine and even recommended.

Keep in mind that you have to run this command on a server to which the domain specified under the -d (for domain) flag resolves — that is, your production server. Even if you have the DNS resolution in your local hosts file, this won't work, as the domain will be verified from outside. So if you're doing this locally, it will most likely not work at all, unless you opened up a port from your local machine to the outside and have it running behind a domain name which resolves to your machine, which is a highly unlikely scenario.

Last but not least, after running this command, the output will contain paths to your private key and certificate files. Copy these values into the previous code snippet, into the cert property for certificate and key property for the key.

```
// ...  
  
const options = {  
  key: fs.readFileSync("/var/www/example/sslcert/privkey.pem"),  
  cert: fs.readFileSync("/var/www/example/sslcert/fullchain.pem") //  
    →might differ for you, make sure to copy from the certbot output  
};  
  
// ...
```

Tightening It Up

HSTS

Have you ever had a website where you switched from HTTP to HTTPS and there were some residual redirects still redirecting to HTTP? HSTS is a web security policy mechanism to mitigate protocol downgrade attacks and cookie hijacking.

HSTS effectively forces the client (browser accessing your server) to direct all traffic through HTTPS - a “secure or not at all” ideology!

Express JS doesn’t allow us to add this header by default, so we’ll use *Helmet*, a node module that allows us to do this. Install *Helmet* by running

```
npm install --save helmet
```

Then we just have to add it as a middleware to our Express server:

```
const https = require("https"),
  fs = require("fs"),
  helmet = require("helmet");

const options = {
  key: fs.readFileSync("/srv/www/keys/my-site-key.pem"),
  cert: fs.readFileSync("/srv/www/keys/chain.pem")
};

const app = express();

app.use(helmet()); // Add Helmet as a middleware

app.use((req, res) => {
  res.writeHead(200);
  res.end("hello world\n");
});

app.listen(8000);

https.createServer(options, app).listen(8080);
```

Diffie–Hellman Strong(er) Parameters

In order to skip some complicated math, let's cut to the chase. In very simple terms, there are two different keys used for encryption, the certificate we get from the certificate authority and one that's generated by the server for key exchange. The default key for key exchange (also called **Diffie–Hellman key exchange**, or DH) uses a “smaller” key than the one for the certificate. In order to remedy this, we'll generate a strong DH key and feed it to our secure server for use.

In order to generate a longer (2048 bit) key, you'll need `openssl`, which you probably have installed by default. In case you're unsure, run `openssl -v`. If the command isn't found, install `openssl` by running `brew install openssl`:

```
openssl dhparam -out /var/www/example/sslcert/dh-strong.pem 2048
```

Then copy the path to the file to our configuration:

```
// ...  
  
const options = {  
  key: fs.readFileSync("/var/www/example/sslcert/privkey.pem"),  
  cert: fs.readFileSync("/var/www/example/sslcert/fullchain.pem"), /  
    →might differ for you, make sure to copy from the certbot output  
  dhparam: fs.readFileSync("/var/www/example/sslcert/dh-strong.pem")  
};  
  
// ...
```

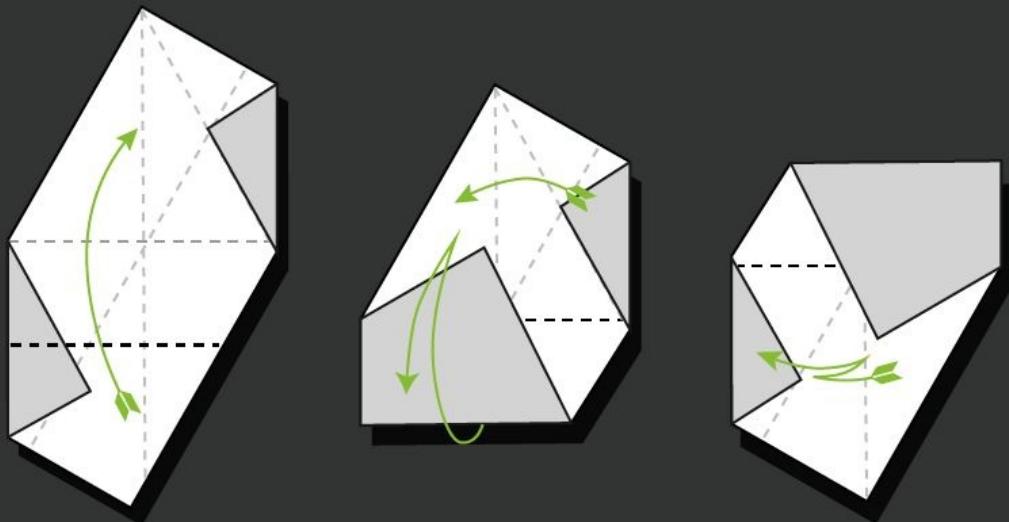
Conclusion

In 2018 and beyond, there's no excuse to dismiss HTTPS. The future direction is clearly visible — HTTPS everywhere! In Node.js, we have a lot of options to utilize SSL/TLS. We can publish our websites in HTTPS, we can create requests to encrypted websites and we can authorize otherwise untrusted certificates.

Book 2: 9 Practical Node.js Projects



9 PRACTICAL NODE.JS PROJECTS



LEVEL UP YOUR NODE KNOWLEDGE

Chapter 1: Build a Simple Beginner App with Node, Bootstrap & MongoDB

by James Hibbard

If you're just getting started with Node.js and want to try your hand at building a web app, things can often get a little overwhelming. Once you get beyond the "Hello, World!" tutorials, much of the material out there has you copy-pasting code, with little or no explanation as to what you're doing or why.

This means that, by the time you've finished, you've built something nice and shiny, but you also have relatively few takeaways that you can apply to your next project.

In this tutorial, I'm going to take a slightly different approach. Starting from the ground up, I'll demonstrate how to build a no-frills web app using Node.js, but instead of focusing on the end result, I'll focus on a range of things you're likely to encounter when building a real-world app. These include routing, templating, dealing with forms, interacting with a database and even basic authentication.

This won't be a JavaScript 101. If that's the kind of thing you're after, [look here](#). It will, however, be suitable for those people who feel reasonably confident with the JavaScript language, and who are looking to take their first steps in Node.js.

What We'll Be Building

We'll be using [Node.js](#) and the [Express framework](#) to build a simple registration form with basic validation, which persists its data to a [MongoDB database](#). We'll add a view to list successful registration, which we'll protect with basic HTTP authentication, and we'll use [Bootstrap](#) to add some styling. The tutorial is structured so that you can follow along step by step. However, if you'd like to jump ahead and see the end result, [the code for this tutorial is also available on GitHub](#).

Basic Setup

Before we can start coding, we'll need to get Node, npm and MongoDB installed on our machines. I won't go into depth on the various installation instructions, but if you have any trouble getting set up, please leave a comment below, or [visit our forums](#) and ask for help there.

Node.js

Many websites will recommend that you head to [the official Node download page](#) and grab the Node binaries for your system. While that works, I would suggest that you use a version manager instead. This is a program which allows you to install multiple versions of Node and switch between them at will. There are various advantages to using a version manager, for example it negates potential permission issues which would otherwise see you installing packages with admin rights.

If you fancy going the version manager route, please consult our quick tip: [Install Multiple Versions of Node.js Using nvm](#). Otherwise, grab the correct binaries for your system from the link above and install those.

npm

npm is a JavaScript package manager which comes bundled with Node, so no extra installation is necessary here. We'll be making quite extensive use of npm throughout this tutorial, so if you're in need of a refresher, please consult: [A Beginner's Guide to npm — the Node Package Manager](#).

MongoDB

MongoDB is a document database which stores data in flexible, JSON-like documents.

The quickest way to get up and running with Mongo is to use a service such as mLab. They have a free sandbox plan which provides a single database with 496 MB of storage running on a shared virtual machine. This is more than adequate for a simple app with a handful of users. If this sounds like the best

option for you, please consult their [quick start guide](#).

You can also install Mongo locally. To do this, please visit the [official download page](#) and download the correct version of the community server for your operating system. There's a link to detailed, OS-specific installation instructions beneath every download link, which you can consult if you run into trouble.

A MongoDB GUI

Although not strictly necessary for following along with this tutorial, you might also like to install [Compass, the official GUI for MongoDB](#). This tool helps you visualize and manipulate your data, allowing you to interact with documents with full CRUD functionality.

At the time of writing, you'll need to fill out your details to download Compass, but you won't need to create an account.

Check that Everything Is Installed Correctly

To check that Node and npm are installed correctly, open your terminal and type:

```
node -v
```

followed by:

```
npm -v
```

This will output the version number of each program (8.9.4 and 5.6.0 respectively at the time of writing).

If you installed Mongo locally, you can check the version number using:

```
mongo --version
```

This should output a bunch of information, including the version number (3.6.2 at the time of writing).

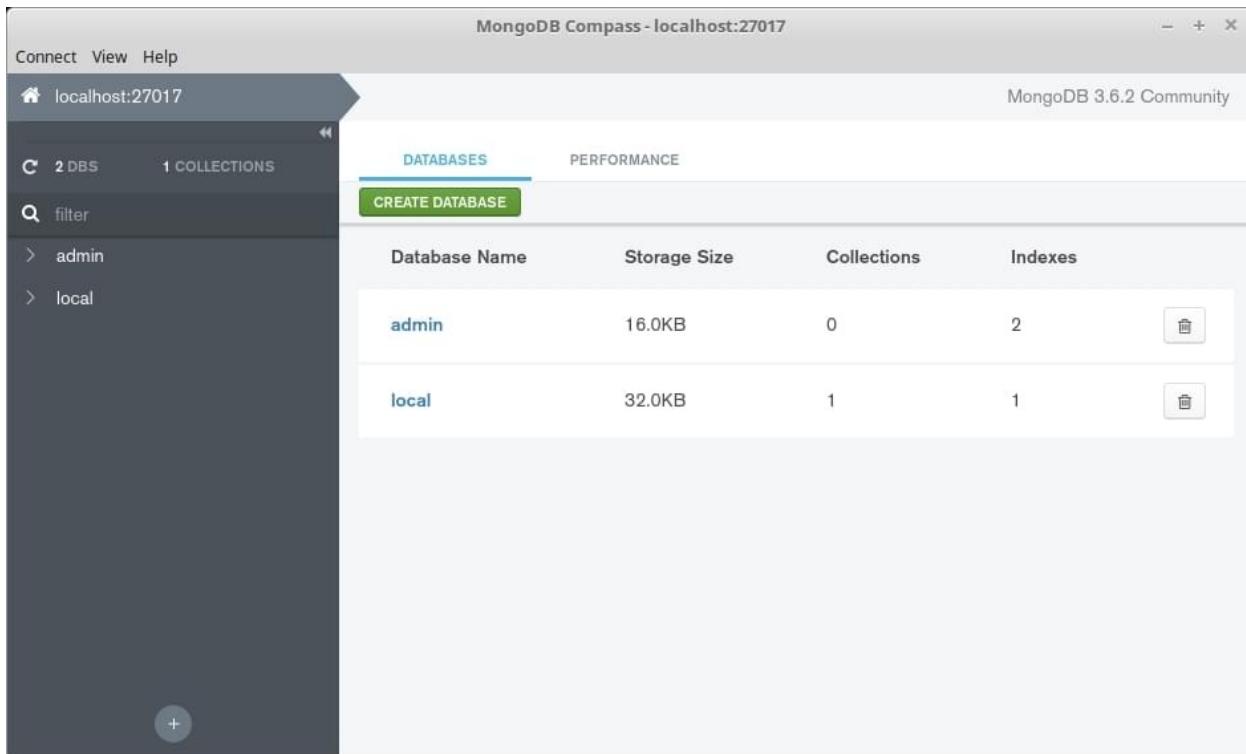
Check the Database Connection Using Compass

If you have installed Mongo locally, you start the server by typing the following

command into a terminal:

```
mongod
```

Next, open Compass. You should be able to accept the defaults (server: localhost, port: 27017), press the *CONNECT* button, and establish a connection to the database server.



The screenshot shows the MongoDB Compass application window titled "MongoDB Compass - localhost:27017". The interface has a top navigation bar with "Connect", "View", and "Help" options. Below the title, it says "localhost:27017". On the left, there's a sidebar with a "Databases" section showing "2 DBS" and "1 COLLECTIONS", and a "filter" search bar. Under "admin", there are two collections: "admin" and "local". The main area is titled "DATABASES" and contains a table with the following data:

Database Name	Storage Size	Collections	Indexes
admin	16.0KB	0	2
local	32.0KB	1	1

Each row in the table has a small trash can icon in the bottom right corner.

MongoDB Compass connected to localhost

Note that the databases `admin` and `local` are created automatically.

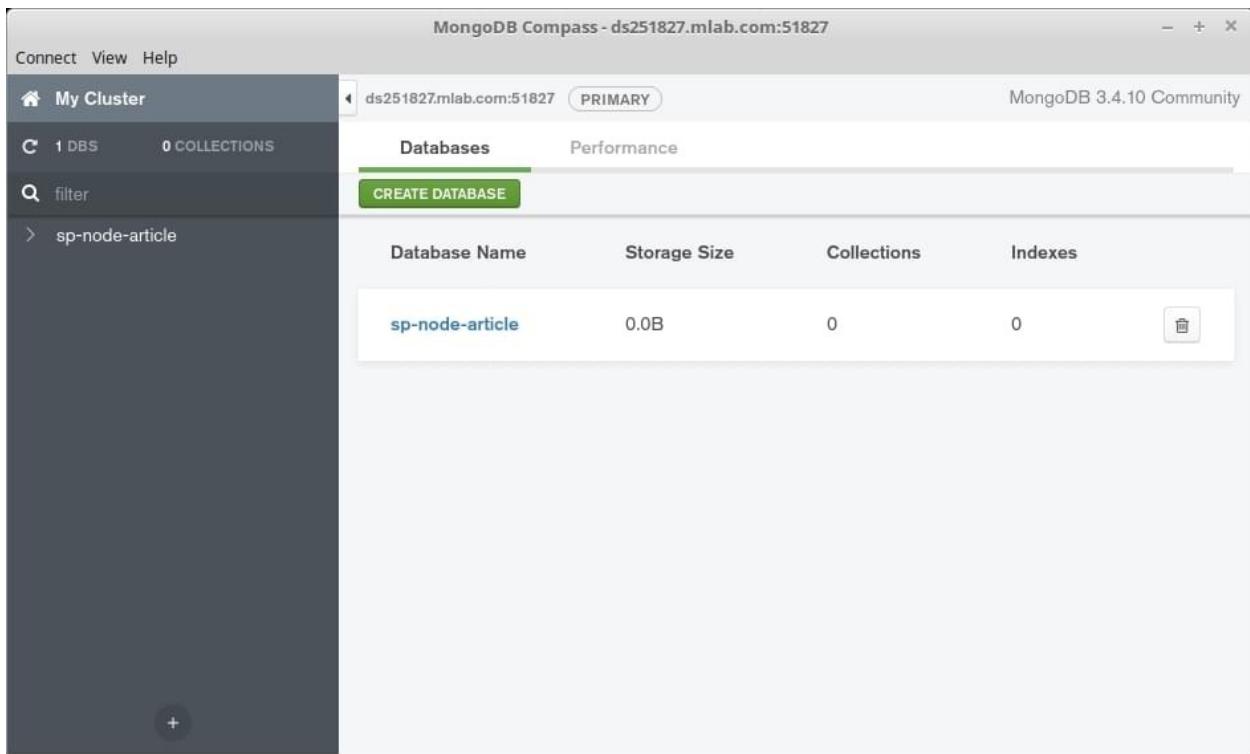
Using a Cloud-hosted Solution

If you're using mLab, create a database subscription (as described in their [quick-start guide](#)), then copy the connection details to the clipboard. This should be in the form:

```
mongodb://<dbuser>:<dbpassword>@ds251827.mlab.com:51827/<dbname>
```

When you open Compass, it will inform you that it has detected a MongoDB connection string and asks if you would like to use it to fill out the form. Click

Yes, noting that you might need to adjust the username and password by hand. After that, click *CONNECT* and you should be off to the races.



The screenshot shows the MongoDB Compass interface connected to a cluster at `ds251827.mlab.com:51827`. The cluster is identified as `PRIMARY`. The interface displays one database, `sp-node-article`, which has 0 collections and 0 indexes. The storage size is listed as 0.0B. A green `CREATE DATABASE` button is visible above the table. The left sidebar shows the cluster name and connection details.

Database Name	Storage Size	Collections	Indexes
<code>sp-node-article</code>	0.0B	0	0

MongoDB Compass connected to mLabs

Note that I called my database `sp-node-article`. You can call yours what you like.

Initialize the Application

With everything set up correctly, the first thing we need to do is initialize our new project. To do this, create a folder named `demo-node-app`, enter that directory and type the following in a terminal:

```
npm init -y
```

This will create and auto-populate a `package.json` file in the project root. We can use this file to specify our dependencies and to create various [npm scripts](#), which will aid our development workflow.

Install Express

Express is a lightweight web application framework for Node.js, which provides us with a robust set of features for writing web apps. These features include such things as route handling, template engine integration and a middleware framework, which allows us to perform additional tasks on request and response objects. There is nothing you can do in Express that you couldn't do in plain Node.js, but using Express means we don't have to re-invent the wheel and reduces boilerplate.

So let's install [Express](#). To do this, run the following in your terminal:

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

By passing the `--save` option to the `npm install` command, Express will be added to the `dependencies` section of the `package.json` file. This signals to anyone else running our code that Express is a package our app needs to function properly.

Install nodemon

[nodemon](#) is a convenience tool. It will watch the files in the directory it was started in, and if it detects any changes, it will automatically restart your Node application (meaning you don't have to). In contrast to Express, nodemon is not something the app requires to function properly (it just aids us with development), so install it using:

```
npm install --save-dev nodemon
```

This will add nodemon to the dev-dependencies section of the package.json file.

Create Some Initial Files

We're almost through with the setup. All we need to do now is create a couple of initial files before kicking off the app.

In the demo-node-app folder create an `app.js` file and a `start.js` file. Also create a `routes` folder, with an `index.js` file inside. After you're done, things should look like this:

```
.
├── app.js
├── node_modules
│   └── ...
└── package.json
└── routes
    └── index.js
└── start.js
```

Now, let's add some code to those files.

In `app.js`:

```
const express = require('express');
const routes = require('./routes/index');

const app = express();
app.use('/', routes);

module.exports = app;
```

Here, we're importing both the `express` module and (the export value of) our `routes` file into the application. The `require` function we're using to do this is a built-in Node function which imports an object from another file or module. If you'd like a refresher on importing and exporting modules, read [Understanding module.exports and exports in Node.js](#).

After that, we're creating a new Express app using the `express` function and

assigning it to an app variable. We then tell the app that, whenever it receives a request from forward slash anything, it should use the routes file.

Finally, we export our app variable so that it can be imported and used in other files.

In start.js:

```
const app = require('./app');

const server = app.listen(3000, () => {
  console.log(`Express is running on port ${server.address().port}`);
});
```

Here we're importing the Express app we created in app.js (note that we can leave the .js off the file name in the require statement). We then tell our app to listen on port 3000 for incoming connections and output a message to the terminal to indicate that the server is running.

And in routes/index.js:

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

const router = express.Router();

router.get('/', (req, res) => {
  res.send('It works!');
});

module.exports = router;
```

Here, we're importing Express into our routes file and then grabbing the router from it. We then use the router to respond to any requests to the root URL (in this case `http://localhost:3000`) with an “It works!” message.

Kick off the App

Finally, let's add an npm script to make nodemon start watching our app. Change the scripts section of the package.json file to look like this:

```
"scripts": {
  "watch": "nodemon ./start.js"
},
```

The `scripts` property of the package `.json` file is extremely useful, as it lets you specify arbitrary scripts to run in different scenarios. This means that you don't have to repeatedly type out long-winded commands with a difficult-to-remember syntax. If you'd like to find out more about what npm scripts can do, read [Give Grunt the Boot! A Guide to Using npm as a Build Tool](#).

Now, type `npm run watch` from the terminal and visit <http://localhost:3000>.

You should see "It works!"

Basic Templating with Pug

Returning an inline response from within the route handler is all well and good, but it's not very extensible, and this is where templating engines come in. As the [Express docs](#) state:

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.

In practice, this means we can define template files and tell our routes to use them instead of writing everything inline. Let's do that now.

Create a folder named `views` and in that folder a file named `form.pug`. Add the following code to this new file:

```
form(action=". " method="POST")
  label(for="name") Name:
    input(
      type="text"
      id="name"
      name="name"
    )

  label(for="email") Email:
    input(
      type="email"
      id="email"
      name="email"
    )

  input(type="submit" value="Submit")
```

As you can deduce from the file ending, we'll be using the [pug templating engine](#) in our app. Pug (formerly known as Jade) comes with its own indentation-sensitive syntax for writing dynamic and reusable HTML. Hopefully the above example is easy to follow, but if you have any difficulties understanding what it does, just wait until we view this in a browser, then inspect the page source to see the markup it produces.

If you'd like a refresher as to what JavaScript templates and/or templating engines are, and when you should use them, read [An Overview of JavaScript Templating Engines](#).

Install Pug and Integrate It into the Express App

Next, we'll need to install pug, saving it as a dependency:

```
npm i --save pug
```

Then configure `app.js` to use Pug as a layout engine and to look for templates inside the `views` folder:

```
const express = require('express');
const path = require('path');
const routes = require('./routes/index');

const app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

app.use('/', routes);

module.exports = app;
```

You'll notice that we're also requiring Node's native [Path module](#), which provides utilities for working with file and directory paths. This module allows us to build the path to our `views` folder using its [join method](#) and `__dirname` (which returns the directory in which the currently executing script resides).

Alter the Route to Use Our Template

Finally, we need to tell our route to use our new template. In `routes/index.js`:

```
router.get('/', (req, res) => {
  res.render('form');
});
```

This uses the [render method](#) on Express's response object to send the rendered view to the client.

So let's see if it worked. As we're using nodemon to watch our app for changes, you should simply be able to refresh your browser and see our brutalist masterpiece.

Define a Layout File for Pug

If you open your browser and inspect the page source, you'll see that Express only sent the HTML for the form: our page is missing a doc-type declaration, as well as a head and body section. Let's fix that by creating a master layout for all our templates to use.

To do this, create a `layout.pug` file in the `views` folder and add the following code:

```
doctype html
html
  head
    title= `${title}`

  body
    h1 My Amazing App

  block content
```

The first thing to notice here is the line starting `title=`. Appending an equals sign to an attribute is one of the methods that Pug uses for interpolation. You can read more about it [here](#). We'll use this to pass the title dynamically to each template.

The second thing to notice is the line that starts with the `block` keyword. In a template, a block is simply a “block” of Pug that a child template may replace. We'll see how to use it shortly, but if you're keen to find out more, read [this page on the Pug website](#).

Use the Layout File from the Child Template

All that remains to do is to inform our `form.pug` template that it should use the layout file. To do this, alter `views/form.pug`, like so:

```
extends layout
```

```
block content
  form(action=". " method="POST")
    label(for="name") Name:
    input(
      type="text"
      id="name"
      name="name"
    )

    label(for="email") Email:
    input(
      type="email"
      id="email"
      name="email"
    )

    input(type="submit" value="Submit")
```

And in `routes/index.js`, we need to pass in an appropriate title for the template to display:

```
router.get('/', (req, res) => {
  res.render('form', { title: 'Registration form' });
});
```

Now if you refresh the page and inspect the source, things should look a lot better.

Dealing with Forms in Express

Currently, if you hit our form's *Submit* button, you'll be redirected to a page with a message: "Cannot POST /". This is because when submitted, our form POSTs its contents back to / and we haven't defined a route to handle that yet.

Let's do that now. Add the following to `routes/index.js`:

```
router.post('/', (req, res) => {
  res.render('form', { title: 'Registration form' });
});
```

This is the same as our GET route, except for the fact that we're using `router.post` to respond to a different HTTP verb.

Now when we submit the form, the error message will be gone and the form should just re-render.

Handle Form Input

The next task is to retrieve whatever data the user has submitted via the form. To do this, we'll need to install a package named [body-parser](#), which will make the form data available on the request body:

```
npm install --save body-parser
```

We'll also need to tell our app to use this package, so add the following to `app.js`:

```
const bodyParser = require('body-parser');
...
app.use(bodyParser.urlencoded({ extended: true }));
app.use('/', routes);
module.exports = app;
```

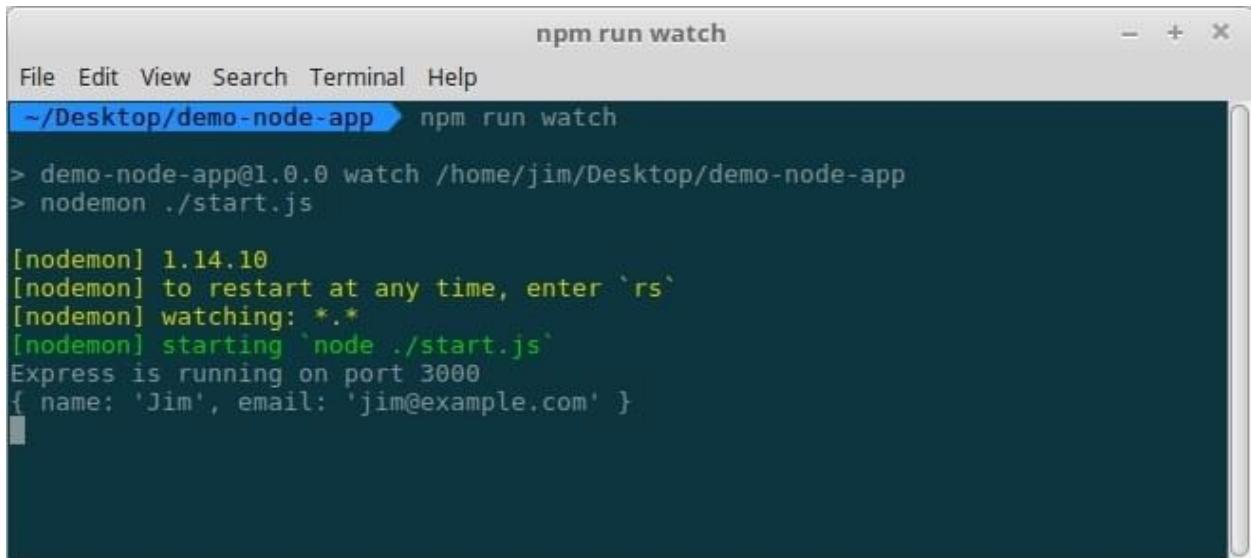
Note that there are various ways to format the data you POST to the server, and using `body-parser`'s [urlencoded](#) method allows us to handle data sent as `application/x-www-form-urlencoded`.

Then we can try logging the submitted data to the terminal. Alter the route handler like so:

```
router.post('/', (req, res) => {
  console.log(req.body);
  res.render('form', { title: 'Registration form' });
});
```

Now when you submit the form, you should see something along the lines of:

```
{name: 'Jim', email: 'jim@example.com'}
```



```
npm run watch
File Edit View Search Terminal Help
~/Desktop/demo-node-app ➜ npm run watch
> demo-node-app@1.0.0 watch /home/jim/Desktop/demo-node-app
> nodemon ./start.js

[nodemon] 1.14.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node ./start.js`
Express is running on port 3000
{ name: 'Jim', email: 'jim@example.com' }
```

Form output logged to terminal

A Note about Request and Response Objects

By now you've hopefully noticed the pattern we're using to handle routes in Express.

```
router.METHOD(route, (req, res) => {
  // callback function
});
```

The callback function is executed whenever somebody visits a URL that matches the route it specifies. The callback receives a `req` and `res` parameter, where `req` is an object full of information that is coming in (such as form data or query parameters) and `res` is an object full of methods for sending data back to the

user. There's also an optional next parameter which is useful if you don't actually want to send any data back, or if you want to pass the request off for something else to handle.

Without getting too deep into the weeds, this is a concept known as middleware (specifically, router-level middleware) which is very important in Express. If you're interested in finding out more about how Express uses middleware, I recommend you read the [Express docs](#).

Validating Form Input

Now let's check that the user has filled out both our fields. We can do this using [express-validator module](#), a middleware that provides a number of useful methods for the sanitization and validation of user input.

You can install it like so:

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

And require the functions we'll need in `routes/index.js`:

```
const { body, validationResult } = require('express-validator/check')
```

We can include it in our route handler like so:

```
router.post('/',  
  [  
    body('name')  
      .isLength({ min: 1 })  
      .withMessage('Please enter a name'),  
    body('email')  
      .isLength({ min: 1 })  
      .withMessage('Please enter an email'),  
  ],  
  (req, res) => {  
    ...  
  }  
);
```

As you can see, we're using the [body method](#) to validate two properties on `req.body` — namely, `name` and `email`. In our case, it's sufficient to just check that these properties exist (i.e. that they have a length greater than one), but

express-validator offers a whole host of other methods that you can read about on the [project's home page](#).

In a second step, we can call the `validationResult` method to see if validation passed or failed. If no errors are present, we can go ahead and render out a “Thanks for registering” message. Otherwise, we’ll need to pass these errors back to our template, so as to inform the user that something’s wrong.

And if validation fails, we’ll also need to pass `req.body` back to the template, so that any valid form inputs aren’t reset:

```
router.post(
  '/',
  [
    ...
  ],
  (req, res) => {
  const errors = validationResult(req);

  if (errors.isEmpty()) {
    res.send('Thank you for your registration!');
  } else {
    res.render('form', {
      title: 'Registration form',
      errors: errors.array(),
      data: req.body,
    });
  }
});
```

Now we have to make a couple of changes to our `form.pug` template. We firstly need to check for an `errors` property, and if it’s present, loop over any errors and display them in a list:

```
extends layout

block content
  if errors
    ul
      for error in errors
        li= error.msg
    ...
```

If the `li=` looks weird, remember that pug does interpolation by following the tag name with an equals sign.

Finally, we need to check if a `data` attribute exists, and if so, use it to set the values of the respective fields. If it doesn't exist, we'll initialize it to an empty object, so that the form will still render correctly when you load it for the first time. We can do this with some JavaScript, denoted in Pug by a minus sign:

```
-data = data || {}
```

We then reference that attribute to set the field's value:

```
input(
  type="text"
  id="name"
  name="name"
  value=data.name
)
```

That gives us the following:

```
extends layout

block content
  -data = data || {}

  if errors
    ul
      for error in errors
        li= error.msg

  form(action"." method="POST")
    label(for="name") Name:
    input(
      type="text"
      id="name"
      name="name"
      value=data.name
    )

    label(for="email") Email:
    input(
      type="email"
      id="email"
      name="email"
```

```
    value=data.email  
)  
  
input(type="submit" value="Submit")
```

Now, when you submit a successful registration, you should see a thank you message, and when you submit the form without filling out both field, the template should be re-rendered with an error message.

Interact with a Database

We now want to hook our form up to our database, so that we can save whatever data the user enters. If you're running Mongo locally, don't forget to start the server with the command `mongod`.

Specify Connection Details

We'll need somewhere to specify our database connection details. For this, we'll use a configuration file (which *should not* be checked into version control) and the [dotenv package](#). Dotenv will load our connection details from the configuration file into Node's [process.env](#).

Install it like so:

```
npm install dotenv --save
```

And require it at the top of `start.js`:

```
require('dotenv').config();
```

Next, create a file named `.env` in the project root (note that starting a filename with a dot may cause it to be hidden on certain operating systems) and enter your Mongo connection details on the first line.

If you're running Mongo locally:

```
DATABASE=mongodb://localhost:27017/node-demo-application
```

If you're using mLab:

```
mongodb://<dbuser>:<dbpassword>@ds251827.mlab.com:51827/<dbname>
```

Note that local installations of MongoDB don't have a default user or password. This is definitely something you'll want to change in production, as it's otherwise a security risk.

Connect to the Database

To establish the connection to the database and to perform operations on it, we'll be using [Mongoose](#). Mongoose is an [ORM](#) for MongoDB, and as you can read on the [project's home page](#):

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

What this means in real terms is that it creates various abstractions over Mongo, which make interacting with our database easier and reduce the amount of boilerplate we have to write. If you'd like to find out more about how Mongo works under the hood, be sure to read our [Introduction to MongoDB](#).

To install Mongoose:

```
npm install --save mongoose
```

Then, require it in start.js:

```
const mongoose = require('mongoose');
```

The connection is made like so:

```
mongoose.connect(process.env.DATABASE, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
mongoose.connection
  .on('connected', () => {
    console.log(`Mongoose connection open on ${process.env.DATABASE}`);
  })
  .on('error', (err) => {
    console.log(`Connection error: ${err.message}`);
  });
});
```

Notice how we use the DATABASE variable we declared in the .env file to specify the database URL. We're also telling Mongo to use ES6 Promises (these are necessary, as database interactions are asynchronous), as its own default promise library is deprecated.

This is what start.js should now look like:

```
require('dotenv').config();
const mongoose = require('mongoose');
```

```

mongoose.connect(process.env.DATABASE, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
mongoose.connection
  .on('connected', () => {
    console.log(`Mongoose connection open on ${process.env.DATABASE}`);
  })
  .on('error', (err) => {
    console.log(`Connection error: ${err.message}`);
  });

const app = require('./app');
const server = app.listen(3000, () => {
  console.log(`Express is running on port ${server.address().port}`);
});

```

When you save the file, nodemon will restart the app and, if all's gone well, you should see something along the lines of:

```
Mongoose connection open on mongodb://localhost:27017/node-demo-app
```

Define a Mongoose Schema

MongoDB can be used as a loose database, meaning it's not necessary to describe what data will look like ahead of time. However, out of the box it runs in strict mode, which means it'll only allow you to save data it knows about beforehand. As we'll be using strict mode, we'll need to define the shape our data using a [schema](#). Schemas allow you to define the fields stored in each document along with their type, validation requirements and default values.

To this end, create a `models` folder in the project root, and within that folder, a new file named `Registration.js`.

Add the following code to `Registration.js`:

```

const mongoose = require('mongoose');

const registrationSchema = new mongoose.Schema({
  name: {
    type: String,
    trim: true,
  },
  email: {
    type: String,
    trim: true,
  }
});

```

```
  },
});

module.exports = mongoose.model('Registration', registrationSchema);
```

Here, we're just defining a type (as we already have validation in place) and are making use of the [trim helper method](#) to remove any superfluous white space from user input. We then [compile a model](#) from the Schema definition, and export it for use elsewhere in our app.

The final piece of boilerplate is to require the model in `start.js`:

```
...

require('./models/Registration');
const app = require('./app');

const server = app.listen(3000, () => {
  console.log(`Express is running on port ${server.address().port}`)
});
```

Save Data to the Database

Now we're ready to save user data to our database. Let's begin by requiring Mongoose and importing our model into our `routes/index.js` file:

```
const express = require('express');
const mongoose = require('mongoose');
const { body, validationResult } = require('express-validator/check')

const router = express.Router();
const Registration = mongoose.model('Registration');
...
```

Now, when the user posts data to the server, if validation passes we can go ahead and create a new `Registration` object and attempt to save it. As the database operation is an asynchronous operation which returns a Promise, we can chain a `.then()` onto the end of it to deal with a successful insert and a `.catch()` to deal with any errors:

```
if (errors.isEmpty()) {
  const registration = new Registration(req.body);
  registration.save()
```

```

        .then(() => { res.send('Thank you for your registration!'); })
        .catch(() => { res.send('Sorry! Something went wrong.'); }));
} else {
    ...
}
...

```

Now, if you enter your details into the registration form, they should be persisted to the database. You can check this using Compass (making sure to hit the refresh button in the top left if it's still running).

The screenshot shows the MongoDB Compass interface connected to a cluster named 'My Cluster' at 'ds251827.mlab.com:51827'. The primary database is selected. The 'sp-node-article' database is expanded, showing the 'registrations' collection. The 'Documents' tab is active, displaying a single document with the following fields:

```

_id: ObjectId("5a5745ca940283029766c897")
name: "Jim"
email: "jim@example.com"
__v: 0

```

The document count is 1, total size is 112B, and average size is 112B. There is one index with a total size of 8.0KB and an average size of 8.0KB.

Using Compass to check that our data was saved to MongoDB

Retrieve Data from the Database

To round the app off, let's create a final route, which lists out all of our registrations. Hopefully you should have a reasonable idea of the process by now.

Add a new route to `routes/index.js`, as follows:

```
router.get('/registrations', (req, res) => {
```

```
res.render('index', { title: 'Listing registrations' });
});
```

This means that we'll also need a corresponding view template (`views/index.pug`):

```
extends layout

block content
  p No registrations yet :(
```

Now when you visit <http://localhost:3000/registrations>, you should see a message telling you that there aren't any registrations.

Let's fix that by retrieving our registrations from the database and passing them to the view. We'll still display the "No registrations yet" message, but only if there really aren't any.

In `routes/index.js`:

```
router.get('/registrations', (req, res) => {
  Registration.find()
    .then((registrations) => {
      res.render('index', { title: 'Listing registrations', registrations })
    })
    .catch(() => { res.send('Sorry! Something went wrong.'); });
});
```

Here, we're using Mongo's [Collection#find method](#), which, if invoked without parameters, will return all of the records in the collection. Because the database lookup is asynchronous, we're waiting for it to complete before rendering the view. If any records were returned, these will be passed to the view template in the `registrations` property. If no records were returned `registrations` will be an empty array.

In `views/index.pug`, we can then check the length of whatever we're handed and either loop over it and output the records to the screen, or display a "No registrations" message:

```
extends layout

block content
```

```
if registrations.length
  table
    tr
      th Name
      th Email
    each registration in registrations
      tr
        td= registration.name
        td= registration.email
  else
    p No registrations yet :(
```

Add HTTP Authentication

The final feature we'll add to our app is [HTTP authentication](#), locking down the list of successful registrations from prying eyes.

To do this, we'll use the [http-auth module](#), which we can install using:

```
npm install --save http-auth
```

Next we need to require it in `routes/index.js`, along with the Path module we met earlier:

```
const path = require('path');
const auth = require('http-auth');
```

Next, let it know where to find the file in which we'll list the users and passwords (in this case `users.htpasswd` in the project root):

```
const basic = auth.basic({
  file: path.join(__dirname, '../users.htpasswd'),
});
```

Create this `users.htpasswd` file next and add a username and password separated by a colon. This can be in plain text, but the `http-auth` module also supports hashed passwords, so you could also run the password through a service such as [Htpasswd Generator](#).

For me, the contents of `users.htpasswd` looks like this:

```
jim:$apr1$FhFmamtz$PgXfrNI95HFCuXIm30Q4V0
```

This translates to user: `jim`, password: `password`.

Finally, add it to the route you wish to protect and you're good to go:

```
router.get('/registrations', auth.connect(basic), (req, res) => {
  ...
});
```

Serve Static Assets in Express

Let's give the app some polish and add some styling using [Twitter Bootstrap](#). We can serve static files such as images, JavaScript files and CSS files in Express using the built-in [express.static middleware function](#).

Setting it up is easy. Just add the following line to `app.js`:

```
app.use(express.static('public'));
```

Now we can load files that are in the `public` directory.

Style the App with Bootstrap

Create a `public` directory in the project root, and in the `public` directory create a `css` directory. Download [the minified version of Bootstrap v4](#) into this directory, ensuring it's named `bootstrap.min.css`.

Next, we'll need to add some markup to our pug templates.

In `layout.pug`:

```
doctype html
html
  head
    title= `{$title}`
    link(rel='stylesheet', href='/css/bootstrap.min.css')
    link(rel='stylesheet', href='/css/styles.css')

  body
    div.container.listing-reg
      h1 My Amazing App

    block content
```

Here, we're including two files from our previously created `css` folder and adding a wrapper `div`.

In `form.pug` we add some class names to the error messages and the form elements:

```

extends layout

block content
  -data = data || {}

  if errors
    ul.my-errors
      for error in errors
        li= error.msg

  form(action=".." method="POST" class="form-registration")
    label(for="name") Name:
    input(
      type="text"
      id="name"
      name="name"
      class="form-control"
      value=data.name
    )

    label(for="email") Email:
    input(
      type="email"
      id="email"
      name="email"
      class="form-control"
      value=data.email
    )

    input(
      type="submit"
      value="Submit"
      class="btn btn-lg btn-primary btn-block"
    )

```

And in `index.pug`, more of the same:

```

extends layout

block content

  if registrations.length
    table.listing-table.table-dark.table-striped
      tr
        th Name
        th Email
      each registration in registrations

```

```

        tr
            td= registration.name
            td= registration.email
    else
        p No registrations yet :(

```

Finally, create a file called styles.css in the css folder and add the following:

```

body {
    padding: 40px 10px;
    background-color: #eee;
}
.listing-reg h1 {
    text-align: center;
    margin: 0 0 2rem;
}

/* css for registration form and errors*/
.form-registration {
    max-width: 330px;
    padding: 15px;
    margin: 0 auto;
}
.form-registration {
    display: flex;
    flex-wrap: wrap;
}
.form-registration input {
    width: 100%;
    margin: 0px 0 10px;
}
.form-registration .btn {
    flex: 1 0 100%;
}
.my-errors {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    color: #333;
    font-size: 1.2rem;
    display: table;
}
.my-errors li {
    margin: 0 0 1rem;
}
.my-errors li:before {
    content: "! Error : ";

```

```
    color: #f00;
    font-weight: bold;
}

/* Styles for listing table */
.listing-table {
    width: 100%;
}
.listing-table th,
.listing-table td {
    padding: 10px;
    border-bottom: 1px solid #666;
}
.listing-table th {
    background: #000;
    color: #fff;
}
.listing-table td:first-child,
.listing-table th:first-child {
    border-right: 1px solid #666;
}
```

Now when you refresh the page, you should see all of the Bootstrap glory!

Conclusion

I hope you've enjoyed this tutorial. While we didn't build the next Facebook, I hope that I was nonetheless able to help you get your feet wet in the world of Node-based web apps and offer you some solid takeaways for your next project in the process.

Of course it's hard to cover everything in one tutorial and there are lots of ways you could elaborate on what we've built here. For example, you could check out [our article on deploying Node apps](#) and try your hand at launching it to [Heroku](#) or [now](#). Alternatively, you might augment the CRUD functionality with the ability to delete registrations, or even write a couple of tests to test the app's functionality.

Chapter 2: How to Build a File Upload Form with Express and Dropzone.js

by Lukas White

Let's face it, nobody likes forms. Developers don't like building them, designers don't particularly enjoy styling them and users certainly don't like filling them in.

Of all the components that can make up a form, the file control could just be the most frustrating of the lot. A real pain to style, chunky and awkward to use and uploading a file will slow down the submission process of any form.

That's why a plugin to enhance them is always worth a look, and [DropzoneJS](#) is just one such option. It will make your file upload controls look better, make them more user-friendly and by using AJAX to upload the file in the background, at the very least make the process *seem* quicker. It also makes it easier to validate files before they even reach your server, providing near-instantaneous feedback to the user.

We're going to take a look at DropzoneJS in some detail; show how to implement it and look at some of the ways in which it can be tweaked and customized. We'll also implement a simple server-side upload mechanism using Node.js.

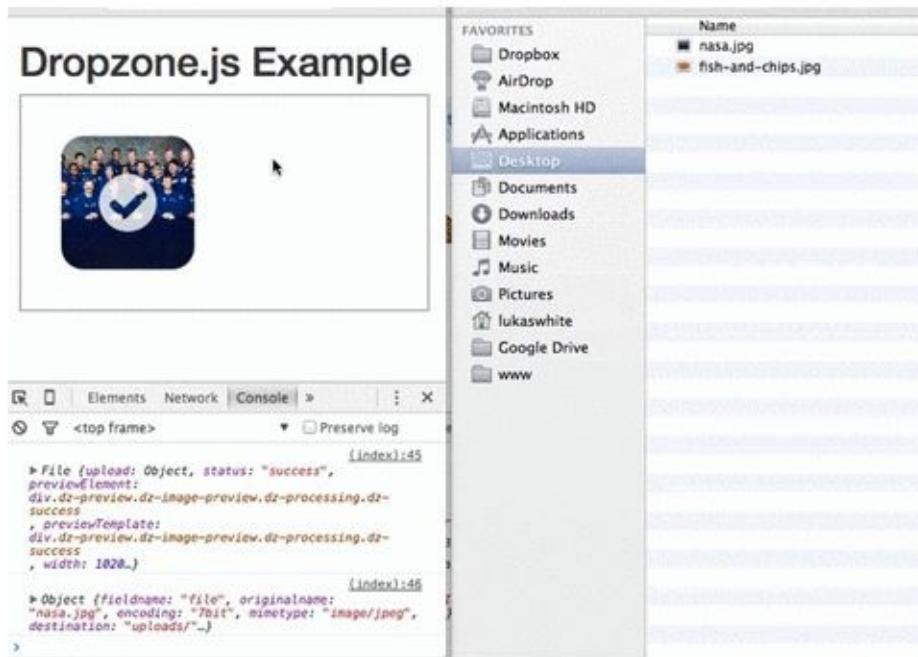
As ever, you can find the code for this tutorial on our [GitHub repository](#).

Introducing DropzoneJS

As the name implies, DropzoneJS allows users to upload files using drag n' drop. Whilst the usability benefits [could justifiably be debated](#), it's an increasingly common approach and one which is in tune with the way a lot of people work with files on their desktop. It's also [pretty well supported](#) across major browsers.

DropzoneJS isn't simply a drag n'drop based widget, however; clicking the widget launches the more conventional file chooser dialog approach.

Here's a screenshot of the widget in action:



Alternatively, take a look at [this, most minimal of examples](#).

You can use DropzoneJS for any type of file, though the nice little thumbnail effect makes it ideally suited to uploading images in particular.

Features

To summarize some of the plugin's features and characteristics:

- Can be used **with or without jQuery**
- Drag and drop support
- Generates thumbnail images
- Supports multiple uploads, optionally in parallel
- Includes a progress bar
- Fully themeable
- Extensible file validation support
- Available as an AMD module or RequireJS module
- It comes in at around 33Kb when minified

Browser Support

Taken from the official documentation, browser support is as follows:

- Chrome 7+
- Firefox 4+
- IE 10+
- Opera 12+ (Version 12 for MacOS is disabled because their API is buggy)
- Safari 6+

There are a couple of ways to handle fallbacks for when the plugin isn't fully supported, which we'll look at later.

Installation

The simplest way to install DropzoneJS is via Bower:

```
bower install dropzone
```

Alternatively you can grab it [from Github](#), or simply download the standalone [JavaScript file](#) — though bear in mind you'll also need [the basic styles](#), also available in the Github repo.

There are also third-party packages providing support for [ReactJS](#) and implementing the widget as an [Angular directive](#).

First Steps

If you've used the Bower or download method, make sure you include both the main JS file and the styles (or include them into your application's stylesheet), e.g:

```
<link rel="stylesheet" href="/path/to/dropzone.css">
<script type="text/javascript" src="/path/to/dropzone.js"></script>
```

Pre-minified versions are also supplied in the package.

If You're Using RequireJS

If you're using RequireJS, use `dropzone-amd-module.js` instead.

Basic Usage

The simplest way to implement the plugin is to attach it to a form, although you can use any HTML such as a `div` tag. Using a form, however, means less options to set — most notably the URL, which is the most important configuration property.

You can initialize it simply by adding the `dropzone` class, for example:

```
<form id="upload-widget" method="post" action="/upload" class="dropz</form>
```

Technically that's all you need to do, though in most cases you'll want to set some additional options. The format for that is as follows:

```
Dropzone.options.WIDGET_ID = {  
    //  
};
```

To derive the widget ID for setting the options, take the ID you defined in your HTML and camel-case it. For example, `upload-widget` becomes `uploadWidget`:

```
Dropzone.options.uploadWidget = {  
    //  
};
```

You can also create an instance programmatically:

```
var uploader = new Dropzone('#upload-widget', options);
```

Next up, we'll look at some of the available configuration options.

Basic Configuration Options

The `url` option defines the target for the upload form, and is the only required parameter. That said, if you're attaching it to a form element then it'll simply use the form's `action` attribute, in which case you don't even need to specify that.

The `method` option sets the HTTP method and again, it will take this from the form element if you use that approach, or else it'll simply default to `POST`, which should suit most scenarios.

The `paramName` option is used to set the name of the parameter for the uploaded file; were you using a file upload form element, it would match the `name` attribute. If you don't include it then it defaults to `file`.

`maxFiles` sets the maximum number of files a user can upload, if it's not set to null.

By default the widget will show a file dialog when it's clicked, though you can use the `clicked` parameter to disable this by setting it to `false`, or alternatively you can provide an HTML element or CSS selector to customize the clickable element.

Those are the basic options, but let's now look at some of the more advanced options.

Enforcing Maximum File Size

The `maxFilesize` property determines the maximum file size in megabytes. This defaults to a size of 1000 bytes, but using the `filesizeBase` property, you could set it to another value — for example, 1024 bytes. You may need to tweak this to ensure that your client and server code calculate any limits in precisely the same way.

Restricting to Certain File Types

The `acceptFiles` parameter can be used to restrict the type of file you want to accept. This should be in the form of a comma-separated list of MIME-types, although you can also use wildcards.

For example, to only accept images:

```
acceptedFiles: 'image/*',
```

Modifying the Size of the Thumbnail

By default, the thumbnail is generated at 120px by 120px; i.e., it's square. There are a couple of ways you can modify this behavior.

The first is to use the `thumbnailWidth` and/or the `thumbnailHeight` configuration options.

If you set both `thumbnailWidth` and `thumbnailHeight` to `null`, the thumbnail won't be resized at all.

If you want to completely customize the thumbnail generation behavior, you can even override the `resize` function.

One important point about modifying the size of the thumbnail; the `dz-image` class provided by the package sets the thumbnail size in the CSS, so you'll need to modify that accordingly as well.

Additional File Checks

The `accept` option allows you to provide additional checks to determine whether a file is valid, before it gets uploaded. You shouldn't use this to check the number of files (`maxFiles`), file type (`acceptedFiles`), or file size (`maxFilesize`), but you can write custom code to perform other sorts of validation.

You'd use the `accept` option like this:

```
accept: function(file, done) {
  if ( !someCheck() ) {
    return done('This is invalid!');
  }
  return done();
}
```

As you can see it's asynchronous; call `done()` with no arguments and validation passes, or provide an error message and the file will be rejected, displaying the

message alongside the file as a popover.

We'll look at a more complex, real-world example later in the article, when we'll look at how to enforce minimum or maximum image sizes.

Sending Additional Headers

Often you'll need to attach additional headers to the uploader's HTTP request.

As an example, one approach to CSRF (Cross Site Request Forgery) protection is to output a token in the view, then have your POST/PUT/DELETE endpoints check the request headers for a valid token. Suppose you outputted your token like this:

```
<meta name="csrf-token" content="CL2tR2J4UHZXcR9BjRtSYOKzSmL8U1zTc7T
```

Then, you could add this to the configuration:

```
headers: {  
  'x-csrf-token': document.querySelectorAll('meta[name=csrf-token]')  
},
```

Alternatively, here's the same example but using jQuery:

```
headers: {  
  'x-csrf-token': $('meta[name="csrf-token"]').attr('content')  
},
```

Your server should then verify the `x-csrf-token` header, perhaps using some middleware.

Handling Fallbacks

The simplest way to implement a fallback is to insert a `<div>` into your form containing input controls, setting the class name on the element to `fallback`. For example:

```
<form id="upload-widget" method="post" action="/upload" class="dropzone">
  <div class="fallback">
    <input name="file" type="file" />
  </div>
</form>
```

Alternatively, you can provide a function to be executed when the browser doesn't support the plugin using the `fallback` configuration parameter.

You can force the widget to use the fallback behavior by setting `forceFallback` to `true`, which might help during development.

Handling Errors

You can customize the way the widget handles errors by providing a custom function using the `error` configuration parameter. The first argument is the file, the error message the second and if the error occurred server-side, the third parameter will be an instance of `XMLHttpRequest`.

As always, client-side validation is only half the battle. You must also perform validation on the server. When we implement a simple server-side component later we'll look at the expected format of the error response, which when properly configured will be displayed in the same way as client-side errors (illustrated below).



Overriding Messages and Translation

There are a number of additional configuration properties which set the various messages displayed by the widget. You can use these to customize the displayed text, or to translate them into another language.

Most notably, `dictDefaultMessage` is used to set the text which appears in the middle of the dropzone, prior to someone selecting a file to upload.

You'll find a complete list of the configurable string values - all of which begin with `dict` - [in the documentation](#).

Events

There are a number of events you can listen to in order to customize or enhance the plugin.

There are two ways to listen to an event. The first is to create a listener within an initialization function:

```
Dropzone.options.uploadWidget = {
  init: function() {
    this.on('success', function( file, resp ){
      ...
    });
  },
  ...
};
```

Or the alternative approach, which is useful if you decide to create the Dropzone instance programmaticaly:

```
var uploader = new Dropzone('#upload-widget');
uploader.on('success', function( file, resp ){
  ...
});
```

Perhaps the most notable is the `success` event, which is fired when a file has been successfully uploaded. The `success` callback takes two arguments; the first, a file object and the second an instance of XMLHttpRequest.

Other useful events include `addedfile` and `removedfile` for when a file has been added or removed from the upload list, `thumbnail` which fires once the thumbnail has been generated and `uploadprogress` which you might use to implement your own progress meter.

There are also a bunch of events which take an event object as a parameter and which you could use to customize the behavior of the widget itself - `drop`, `dragstart`, `dragend`, `dragenter`, `dragover` and `dragleave`.

You'll find a complete list of events in the relevant section [of the documentation](#).

A More Complex Validation Example: Image Dimensions

Earlier we looked at the asynchronous `accept()` option, which you can use to run checks (validation) on files before they get uploaded.

A common requirement when you're uploading images is to enforce minimum or maximum image dimensions. We can do this with DropzoneJS, although it's slightly more complex.

Although the `accept` callback receives a file object, in order to check the image dimensions we need to wait until the thumbnail has been generated, at which point the dimensions will have been set on the file object. To do so, we need to listen to the `thumbnail` event.

Here's the code; in this example we're checking that the image is at least 640 x 480px before we upload it:

```
init: function() {
  this.on('thumbnail', function(file) {
    if ( file.width < 1024 || file.height < 768 ) {
      file.rejectDimensions();
    }
    else {
      file.acceptDimensions();
    }
  });
},
accept: function(file, done) {
  file.acceptDimensions = done;
  file.rejectDimensions = function() {
    done('The image must be at least 1024 by 768 pixels in size');
  };
},
```

A Complete Example

Having gone through the options, events and some slightly more advanced validation, let's look at a complete and relatively comprehensive example. Obviously we're not taking advantage of every available configuration option, since there are so many — making it incredibly flexible.

Here's the HTML for the form:

```
<form id="upload-widget" method="post" action="/upload" class="dropzone">
  <div class="fallback">
    <input name="file" type="file" />
  </div>
</form>
```

If you're implementing CSRF protection, you may want to add something like this to your layouts:

```
<head>
  <!-- -->
  <meta name="csrf-token" content="XYZ123">
</head>
```

Now the JavaScript - notice we're not using jQuery!

```
Dropzone.options.uploadWidget = {
  paramName: 'file',
  maxFilesize: 2, // MB
  maxFiles: 1,
  dictDefaultMessage: 'Drag an image here to upload, or click to select',
  headers: {
    'x-csrf-token': document.querySelectorAll('meta[name=csrf-token]')[0],
  },
  acceptedFiles: 'image/*',
  init: function() {
    this.on('success', function( file, resp ){
      console.log( file );
      console.log( resp );
    });
    this.on('thumbnail', function(file) {
      if ( file.width < 640 || file.height < 480 ) {
        file.rejectDimensions();
      }
    });
  }
};
```

```
        else {
            file.acceptDimensions();
        }
    });
},
accept: function(file, done) {
    file.acceptDimensions = done;
    file.rejectDimensions = function() {
        done('The image must be at least 640 x 480px')
    };
}
};
```

A reminder that you'll find the code for this example on our [GitHub repository](#).

Hopefully, this is enough to get you started for most scenarios; check out the [full documentation](#) if you need anything more complex.

Theming

There are a number of ways to customize the look and feel of the widget, and indeed it's possible to completely transform the way it looks.

For an example of just how customizable the appearance is, here is [a demo](#) of the widget tweaked to look and feel exactly like the [jQuery File Upload](#) widget using Bootstrap.

Obviously the simplest way to change the widget's appearance is to use CSS. The widget has a class of `dropzone` and its component elements have classes prefixed with `dz-`; for example `dz-clickable` for the clickable area inside the dropzone, `dz-message` for the caption, `dz-preview / dz-image-preview` for wrapping the previews of each of the uploaded files, and so on. Take a look at the `dropzone.css` file for reference.

You may also wish to apply styles to the hover state; that is, when the user hovers a file over the dropzone before releasing their mouse button to initiate the upload. You can do this by styling the `dz-drag-hover` class, which gets added automatically by the plugin.

Beyond CSS tweaks, you can also customize the HTML which makes up the previews by setting the `previewTemplate` configuration property. Here's what the default preview template looks like:

```
<div class="dz-preview dz-file-preview">
  <div class="dz-image">
    <img data-dz-thumbnail />
  </div>
  <div class="dz-details">
    <div class="dz-size">
      <span data-dz-size></span>
    </div>
    <div class="dz-filename">
      <span data-dz-name></span>
    </div>
  </div>
  <div class="dz-progress">
    <span class="dz-upload" data-dz-uploadprogress></span>
  </div>
  <div class="dz-error-message">
```

```
<span data-dz-errormessage></span>
</div>
<div class="dz-success-mark">
  <svg>REMOVED FOR BREVITY</svg>
</div>
<div class="dz-error-mark">
  <svg>REMOVED FOR BREVITY</svg>
</div>
</div>
```

As you can see, you get complete control over how files are rendered once they've been queued for upload, as well as success and failure states.

That concludes the section on using the DropzoneJS plugin. To round up, let's look at how to get it working with server-side code.

A Simple Server-Side Upload Handler with Node.js and Express

Naturally you can use any server-side technology for handling uploaded files. In order to demonstrate how to integrate your server with the plugin, we'll build a very simple example using Node.js and Express.

To handle the uploaded file itself we'll use [multer](#), a package which provides some Express middleware that makes it really easy. In fact, this easy:

```
var upload = multer( { dest: 'uploads/' } );

app.post( '/upload', upload.single( 'file' ), function( req, res, ne
  // Metadata about the uploaded file can now be found in req.file
} );
```

Before we continue the implementation, the most obvious question to ask when dealing with a plugin like DropzoneJS which makes requests for you behind the scenes is: “what sort of responses does it expect?”

Handling Upload Success

If the upload process is successful, the only requirement as far as your server-side code is concerned, is to return a 2xx response code. The content and format of your response is entirely up to you, and will probably depend on how you're using it; for example you might return a JSON object which contains a path to the uploaded file, or the path to an automatically generated thumbnail. For the purposes of this example we'll simply return the contents of the file object - i.e. a bunch of metadata - provided by Multer:

```
return res.status( 200 ).send( req.file );
```

The response will look something like this:

```
{fieldname: 'file',
originalname: 'myfile.jpg',
encoding: '7bit',
mimetype: 'image/jpeg',
destination: 'uploads/',
filename: 'fbcc2ddbb0dd11858427d7f0bb2273f5',
path: 'uploads/fbcc2ddbb0dd11858427d7f0bb2273f5',
size: 15458 }
```

Handling Upload Errors

If your response is in JSON format - that is to say, your response type is set to `application/json` - then DropzoneJS default error plugin expects the response to look like this:

```
{  
  error: 'The error message'  
}
```

If you aren't using JSON, it'll simply use the response body, for example:

```
return res.status( 422 ).send( 'The error message' );
```

Let's demonstrate this by performing a couple of validation checks on the uploaded file. We'll simply duplicate the checks we performed on the client — remember, client-side validation is never sufficient on its own.

To verify that the file is an image, we'll simply check that the Mime-type starts with `image/`. ES6's `String.prototype.startsWith()` is ideal for this, but let's install a [polyfill for it](#):

```
npm install string.prototype.startsWith --save
```

Here's how we might run that check and, if it fails, return the error in the format which Dropzone's default error handler expects:

```
if ( !req.file.mimetype.startsWith( 'image/' ) ) {  
  return res.status( 422 ).json( {  
    error : 'The uploaded file must be an image'  
  } );  
}
```

Status Codes

I'm using HTTP Status Code 422, Unprocessable Entity here for validation failure, but 400 Bad Request is just as valid; indeed anything outside of the 2xx range will cause the plugin to report the error.

Let's also check that the image is of a certain size; the [image-size](#) package makes

it really straightforward to get the dimensions of an image. You can use it asynchronously or synchronously; we'll use the latter to keep things simple:

```
var dimensions = sizeOf( req.file.path );

if ( ( dimensions.width < 640 ) || ( dimensions.height < 480 ) ) {
  return res.status( 422 ).json( {
    error : 'The image must be at least 640 x 480px'
  } );
}
```

Let's put all of it together in a complete (mini) application:

```
var express  =  require( 'express' );
var multer   =  require( 'multer' );
var upload   =  multer( { dest: 'uploads/' } );
var sizeOf   =  require( 'image-size' );
var exphbs   =  require( 'express-handlebars' );
require( 'string.prototype.startswith' );

var app = express();

app.use( express.static( __dirname + '/bower_components' ) );

app.engine( '.hbs', exphbs( { extname: '.hbs' } ) );
app.set('view engine', '.hbs');

app.get( '/', function( req, res, next ){
  return res.render( 'index' );
});

app.post( '/upload', upload.single( 'file' ), function( req, res, ne

  if ( !req.file.mimetype.startsWith( 'image/' ) ) {
    return res.status( 422 ).json( {
      error : 'The uploaded file must be an image'
    } );
  }

  var dimensions = sizeOf( req.file.path );

  if ( ( dimensions.width < 640 ) || ( dimensions.height < 480 ) ) {
    return res.status( 422 ).json( {
      error : 'The image must be at least 640 x 480px'
    } );
}
```

```
}

return res.status( 200 ).send( req.file );
});

app.listen( 8080, function() {
  console.log( 'Express server listening on port 8080' );
});
```

CSRF Protection

For brevity, this server-side code doesn't implement CSRF protection; you might want to look at a [package like CSURF](#) for that.

You'll find this code, along with the supporting assets such as the view, in the [accompanying repository](#).

Summary

DropzoneJS is a slick, powerful and highly customizable JavaScript plugin for super-charging your file upload controls and performing AJAX uploads. In this article we've taken a look at a number of the available options, at events and how to go about customizing the plugin. There's a lot more to it than can reasonably be covered in one article, so check out the [official website](#) if you'd like to know more — but hopefully this is enough to get you started.

We've also built a really simple server-side component to handle file uploads, demonstrating how to get the two working in tandem.

Chapter 3: How to Build and Structure a Node.js MVC Application

by James Kolce

In a non-trivial application, the architecture is as important as the quality of the code itself. We can have well-written pieces of code, but if we don't have a good organization, we'll have a hard time as the complexity increases. There's no need to wait until the project is half-way done to start thinking about the architecture. The best time is before starting, using our goals as beacons for our choices.

Node.js doesn't have a de facto framework with strong opinions on architecture and code organization in the same way that Ruby has the Rails framework, for example. As such, it can be difficult to get started with building full web applications with Node.

In this chapter, we're going to build the basic functionality of a note-taking app using the MVC architecture. To accomplish this, we're going to employ the [Hapi.js](#) framework for [Node.js](#) and [SQLite](#) as a database, using [Sequelize.js](#), plus other small utilities to speed up our development. We're going to build the views using [Pug](#), the templating language.

What is MVC?

Model-View-Controller (or MVC) is probably one of the most popular architectures for applications. As with a lot of other [cool things](#) in computer history, the MVC model was conceived at [PARC](#) for the Smalltalk language as a solution to the problem of organizing applications with graphical user interfaces. It was created for desktop applications, but since then, the idea has been adapted to other mediums including the web.

We can describe the MVC architecture in simple words:

- **Model:** The part of our application that will deal with the database or any data-related functionality.
- **View:** Everything the user will see. Basically the pages that we're going to send to the client.
- **Controller:** The logic of our site, and the glue between models and views. Here we call our models to get the data, then we put that data on our views to be sent to the users.

Our application will allow us to publish, see, edit and delete plain-text notes. It won't have other functionality, but because we'll have a solid architecture already defined we won't have big trouble adding things later.

You can check out the final application in the [accompanying GitHub repository](#), so you get a general overview of the application structure.

Laying out the Foundation

The first step when building any Node.js application is to create a package.json file, which is going to contain all of our dependencies and scripts. Instead of creating this file manually, npm can do the job for us using the `init` command:

```
npm init -y
```

After the process is complete will get a package.json file ready to use.

npm Commands

If you're not familiar with these commands, checkout our [Beginner's Guide to npm](#).

We're going to proceed to install Hapi.js — the framework of choice for this tutorial. It provides a good balance between simplicity, stability and feature availability that will work well for our use case (although there are other options that would also work just fine).

```
npm install --save hapi hoek
```

This command will download the latest version of Hapi.js and add it to our package.json file as a dependency. It will also download the [Hoek](#) utility library that will help us write shorter error handlers, among [other things](#).

Now we can create our entry file — the web server that will start everything. Go ahead and create a `server.js` file in your application directory and all the following code to it:

```
'use strict';

const Hapi = require('hapi');
const Hoek = require('hoek');
const Settings = require('./settings');

const server = new Hapi.Server();
server.connection({ port: Settings.port });

server.route({
```

```
method: 'GET',
path: '/',
handler: (request, reply) => {
  reply('Hello, world!');
}
});

server.start((err) => {
  Hoek.assert(!err, err);

  console.log(`Server running at: ${server.info.uri}`);
});
```

This is going to be the foundation of our application.

First, we indicate that we're going to use [strict mode](#), which is a [common practice](#) when using the Hapi.js framework.

Next, we include our dependencies and instantiate a new server object where we set the connection port to 3000 (the port can be any number [above 1023 and below 65535](#).)

Our first route for our server will work as a test to see if everything is working, so a “Hello, world!” message is enough for us. In each route, we have to define the HTTP method and path (URL) that it will respond to, and a handler, which is a function that will process the HTTP request. The handler function can take two arguments: `request` and `reply`. The first one contains information about the HTTP call, and the second will provide us with methods to handle our response to that call.

Finally, we start our server with the `server.start` method. As you can see, we can use Hoek to improve our error handling, making it shorter. This is completely optional, so feel free to omit it in your code, just be sure to handle any errors.

Storing Our Settings

It is good practice to store our configuration variables in a dedicated file. This file exports a JSON object containing our data, where each key is assigned from an environment variable — but without forgetting a fallback value.

In this file, we can also have different settings depending on our environment (e.g. development or production). For example, we can have an in-memory instance of SQLite for development purposes, but a real SQLite database file on production.

Selecting the settings depending on the current environment is quite simple. Since we also have an `env` variable in our file which will contain either `development` or `production`, we can do something like the following to get the database settings (for example):

```
const dbSettings = Settings[Settings.env].db;
```

So `dbSettings` will contain the setting of an in-memory database when the `env` variable is `development`, or will contain the path of a database file when the `env` variable is `production`.

Also, we can add support for a `.env` file, where we can store our environment variables locally for development purposes; this is accomplished using a package like [dotenv](#) for Node.js, which will read a `.env` file from the root of our project and automatically add the found values to the environment. You can find an example in the [dotenv repository](#).

If You Use a `.env` File

If you decide to also use a `.env` file, make sure you install the package with `npm install -s dotenv` and add it to `.gitignore` so you don't publish any sensitive information.

Our `settings.js` file will look like this:

```
// This will load our .env file and add the values to process.env,  
// IMPORTANT: Omit this line if you don't want to use this functiona
```

```
require('dotenv').config({silent: true});

module.exports = {
  port: process.env.PORT || 3000,
  env: process.env.ENV || 'development',

  // Environment-dependent settings
  development: {
    db: {
      dialect: 'sqlite',
      storage: ':memory:'
    }
  },
  production: {
    db: {
      dialect: 'sqlite',
      storage: 'db/database.sqlite'
    }
  }
};
```

Now we can start our application by executing the following command and navigating to `localhost:3000` in our web browser.

```
node server.js
```

Ensure Your Installation is Up-to-date

This project was tested on Node v6. If you get any errors, ensure you have an updated installation.

Defining the Routes

The definition of routes gives us an overview of the functionality supported by our application. To create our additional routes, we just have to replicate the structure of the route that we already have in our `server.js` file, changing the content of each one.

Let's start by creating a new directory called `lib` in our project. Here we're going to include all the JS components. Inside `lib`, let's create a `routes.js` file and add the following content:

```
'use strict';

module.exports = [
  // We're going to define our routes here
];
```

In this file, we'll export an array of objects that contain each route of our application. To define the first route, add the following object to the array:

```
{
  method: 'GET',
  path: '/',
  handler: (request, reply) => {
    reply('All the notes will appear here');
  },
  config: {
    description: 'Gets all the notes available'
  }
},
```

Our first route is for the home page (`/`) and since it will only return information we assign it a `GET` method. For now, it will only give us the message `All the notes will appear here`, which we're going to change later for a controller function. The `description` field in the `config` section is only for documentation purposes.

Then, we create the four routes for our notes under the `/note/` path. Since we're building a [CRUD](#) application, we will need one route for each action with the corresponding [HTTP method](#).

Add the following definitions next to the previous route:

```
{  
  method: 'POST',  
  path: '/note',  
  handler: (request, reply) => {  
    reply('New note');  
  },  
  config: {  
    description: 'Adds a new note'  
  }  
},  
{  
  method: 'GET',  
  path: '/note/{slug}',  
  handler: (request, reply) => {  
    reply('This is a note');  
  },  
  config: {  
    description: 'Gets the content of a note'  
  }  
},  
{  
  method: 'PUT',  
  path: '/note/{slug}',  
  handler: (request, reply) => {  
    reply('Edit a note');  
  },  
  config: {  
    description: 'Updates the selected note'  
  }  
},  
{  
  method: 'GET',  
  path: '/note/{slug}/delete',  
  handler: (request, reply) => {  
    reply('This note no longer exists');  
  },  
  config: {  
    description: 'Deletes the selected note'  
  }  
},
```

We've done the same as in the previous route definition, but this time we've changed the method to match the action we want to execute.

The only exception is the delete route. In this case, we're going to define it with the `GET` method rather than `DELETE` and add an extra `/delete` in the path. This way, we can call the delete action just by visiting the corresponding URL.

Strict REST

If you plan to implement a strict REST interface, then you would have to use the `DELETE` method and remove the `/delete` part of the path.

We can name parameters in the path by surrounding the word in brackets (`{ ... }`). Since we're going to identify notes by a slug, we add `{slug}` to each path, with the exception of the `PUT` route. We don't need it there, because we're not going to interact with a specific note, but to create one.

You can read more about Hapi.js routes on the [official documentation](#).

Now, we have to add our new routes to the `server.js` file. Let's import the `routes` file at the top of the file:

```
const Routes = require('./lib/routes');
```

and replace our current test route with the following:

```
server.route(Routes);
```

Building the Models

Models allow us to define the structure of the data and all the functions to work with it.

In this example, we're going to use the [SQLite](#) database with [Sequelize.js](#) which is going to provide us with a better interface using the ORM ([Object-Relational Mapping](#)) technique. It will also provide us a database-independent interface.

Setting up the database

For this section, we're going to use [Sequelize.js](#) and [SQLite](#). You can install and include them as dependencies by executing the following command:

```
npm install -s sequelize sqlite3
```

Now create a `models` directory inside `lib/` with a file called `index.js` which is going to contain the database and Sequelize.js setup, and include the following content:

```
'use strict';

const Fs = require('fs');
const Path = require('path');
const Sequelize = require('sequelize');
const Settings = require('../..../settings');

// Database settings for the current environment
const dbSettings = Settings[Settings.env].db;

const sequelize = new Sequelize(dbSettings.database, dbSettings.user,
const db = {};

// Read all the files in this directory and import them as models
Fs.readdirSync(__dirname)
.filter((file) => (file.indexOf('.') !== 0) && (file !== 'index.js'))
.forEach((file) => {
  const model = sequelize.import(Path.join(__dirname, file));
  db[model.name] = model;
});

db.sequelize = sequelize;
```

```
db.Sequelize = Sequelize;

module.exports = db;
```

First, we include the modules that we're going to use:

- Fs, to read the files inside the *models* folder, which is going to contain all the models.
- Path, to join the path of each file in the current directory.
- Sequelize, that will allow us to create a new Sequelize instance.
- Settings, which contains the data of our *settings.js* file from the root of our project.

Next, we create a new `sequelize` variable that will contain a `Sequelize` instance with our database settings for the current environment. We're going to use `sequelize` to import all the models and make them available in our `db` object.

The `db` object is going to be exported and will contain our database methods for each model; it will be available in our application when we need to do something with our data.

To load all the models, instead of defining them manually, we look for all the files inside the `models` directory (with the exception of the `index.js` file) and load them using the `import` function. The returned object will provide us with the CRUD methods, which we then add to the `db` object.

At the end, we add `sequelize` and `Sequelize` as part of our `db` object, the first one is going to be used in our `server.js` file to connect to the database before starting the server, and the second one is included for convenience if you need it in other files too.

Creating our Note model

In this section, we're going to use the [Moment.js](#) package to help with Date formatting. You can install it and include it as a dependency with the following command:

```
npm install -s moment
```

Now, we're going to create a `note.js` file inside the `models` directory, which is

going to be the only model in our application; it will provide us with all the functionality we need.

Add the following content to that file:

```
'use strict';

const Moment = require('moment');

module.exports = (sequelize, DataTypes) => {
  let Note = sequelize.define('Note', {
    date: {
      type: DataTypes.DATE,
      get: function () {
        return Moment(this.getDataValue('date')).format('MMMM Do, YYYY')
      }
    },
    title: DataTypes.STRING,
    slug: DataTypes.STRING,
    description: DataTypes.STRING,
    content: DataTypes.STRING
  });
  return Note;
};
```

We export a function that accepts a `sequelize` instance, to define the model, and a `DataTypes` object with all the types available in our database.

Next, we define the structure of our data using an object where each key corresponds to a database column and the value of the key defines the type of data that we're going to store. You can see the list of data types in the [Sequelize.js documentation](#). The tables in the database are going to be created automatically based on this information.

In the case of the date column, we also define a how Sequelize should return the value using a [getter](#) function (`get` key). We indicate that before returning the information it should be first passed through the `Moment` utility to be formatted in a more readable way (`MMMM Do, YYYY`).

Ahem, Excuse Me ...

Although we're getting a simple and easy-to-read date string, it is stored as a precise date string product of the [Date](#) object of JavaScript. So this is not a destructive operation.

Finally, we return our model.

Synchronizing the Database

Now, we have to synchronize our database before we're able to use it in our application. In `server.js`, import the models at the top of the file:

```
// Import the index.js file inside the models directory
const Models = require('./lib/models');
```

Next, replace the following code block:

```
server.start((err) => {
  Hoek.assert(!err, err);
  console.log(`Server running at: ${server.info.uri}`);
});
```

with this one:

```
Models.sequelize.sync().then(() => {
  server.start((err) => {
    Hoek.assert(!err, err);
    console.log(`Server running at: ${server.info.uri}`);
  });
});
```

This code is going to synchronize the models to our database and, once that is done, the server will be started.

Building the controllers

Controllers are functions that accept the [request](#) and [reply](#) objects from Hapi.js. The `request` object contains information about the requested resource and we use `reply` to return information to the client.

In our application, we're going to return only a JSON object for now, but we will add the views once we build them.

We can think of controllers as functions that will join our models with our views; they will communicate with our models to get the data, and then return that data inside a view.

The Home Controller

The first controller that we're going to build will handle the home page of our site. Create a `home.js` file inside the `lib/controllers` directory with the following content:

```
'use strict';

const Models = require('../models');

module.exports = (request, reply) => {
  Models.Note
    .findAll({
      order: [['date', 'DESC']]
    })
    .then((result) => {
      reply({
        data: {
          notes: result
        },
        page: 'Home—Notes Board',
        description: 'Welcome to my Notes Board'
      });
    });
};
```

First, we get all the notes in our database using the [findAll](#) method of our model. This function will return a promise and, if it resolves, we will get an

array containing all the notes in our database.

We can arrange the results in descending order, using the `order` parameter in the `options` object passed to the `findAll` method, so the last item will appear first. You can check all the available options in the [Sequelize.js documentation](#).

Once we have the home controller, we can edit our `routes.js` file. First, we import the module at the top of the file, next to the `Path` module import:

```
const Home = require('./controllers/home');
```

Then we add the controller we just made to the array:

```
{
  method: 'GET',
  path: '/',
  handler: Home,
  config: {
    description: 'Gets all the notes available'
  }
},
```

Boilerplate of the Note Controller

Since we're going to identify our notes with a slug, we can generate one using the title of the note and the [slug](#) library, so let's install it and include it as a dependency with the following command:

```
npm install -s slug
```

The last controller that we have to define in our application will allow us to create, read, update, and delete notes.

We can proceed to create a `note.js` file inside the `lib/controllers` directory and add the following content:

```
'use strict';

const Models = require('../models/');
const Slugify = require('slug');
const Path = require('path');

module.exports = {
```

```
// Here we're going to include our functions that will handle each
};
```

The “create” function

To add a note to our database, we’re going to write a `create` function that is going to wrap the `create` method on our model using the data contained in the payload object.

Add the following inside the object that we’re exporting:

```
create: (request, reply) => {
  Models.Note
    .create({
      date: new Date(),
      title: request.payload.noteTitle,
      slug: Slugify(request.payload.noteTitle, {lower: true}),
      description: request.payload.noteDescription,
      content: request.payload.noteContent
    })
    .then((result) => {
      // We're going to generate a view later, but for now lets just
      reply(result);
    });
},
```

Once the note is created, we will get back the note data and send it to the client as JSON using the `reply` function.

For now, we just return the result, but once we build the views in the next section, we will be able to generate the HTML with the new note and add it dynamically on the client. Although this is not completely necessary and will depend on how you are going to handle your front-end logic, we’re going to return an HTML block to simplify the logic on the client.

Also, note that the date is being generated on the fly when we execute the function, using `new Date()`.

The “read” function

To search just one element we use the `findOne` method on our model. Since we identify notes by their slug, the `where` filter must contain the slug provided by

the client in the URL (`http://localhost:3000/note/:slug:`).

```
read: (request, reply) => {
  Models.Note
    .findOne({
      where: {
        slug: request.params.slug
      }
    })
    .then((result) => {
      reply(result);
    });
},
```

As in the previous function, we will just return the result, which is going to be an object containing the note information. The views are going to be used once we build them in the [Building the Views](#) section.

The “update” function

To update a note, we use the `update` method on our model. It takes two objects, the new values that we’re going to replace and the options containing a `where` filter with the note slug, which is the note that we’re going to update.

```
update: (request, reply) => {
  const values = {
    title: request.payload.noteTitle,
    description: request.payload.noteDescription,
    content: request.payload.noteContent
  };

  const options = {
    where: {
      slug: request.params.slug
    }
  };

  Models.Note
    .update(values, options)
    .then(() => {
      Models.Note
        .findOne(options)
        .then((result) => {
          reply(result);
        });
    });
},
```

```
        });
    });
},
```

After updating our data, since our database won't return the updated note, we can find the modified note again to return it to the client, so we can show the updated version as soon as the changes are made.

The “delete” function

The delete controller will remove the note by providing the slug to the `destroy` function of our model. Then, once the note is deleted, we redirect to the home page. To accomplish this, we use the `redirect` function of the Hapi.js reply object.

```
delete: (request, reply) => {
  Models.Note
    .destroy({
      where: {
        slug: request.params.slug
      }
    })
    .then(() => reply.redirect('/'));
}
```

Using the Note controller in our routes

At this point, we should have our note controller file ready with all the CRUD actions. But to use them, we have to include it in our routes file.

First, let's import our controller at the top of the `routes.js` file:

```
const Note = require('./controllers/note');
```

We have to replace each handler with our new functions, so we should have our routes file as follows:

```
{
  method: 'POST',
  path: '/note',
  handler: Note.create,
  config: {
    description: 'Adds a new note'
  }
```

```
},
{
  method: 'GET',
  path: '/note/{slug}',
  handler: Note.read,
  config: {
    description: 'Gets the content of a note'
  }
},
{
  method: 'PUT',
  path: '/note/{slug}',
  handler: Note.update,
  config: {
    description: 'Updates the selected note'
  }
},
{
  method: 'GET',
  path: '/note/{slug}/delete',
  handler: Note.delete,
  config: {
    description: 'Deletes the selected note'
  }
},
```

Referencing Functions, Not Calling Them

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

Building the Views

At this point, our site is receiving HTTP calls and responding with JSON objects. To make it useful to everybody we have to create the pages that render our information in a nice way.

In this example, we're going to use the Pug templating language, although this is not mandatory and we can use [other languages](#) with Hapi.js. Also, we're going to use the [Vision](#) plugin to enable the view functionality in our server.

Pug

If you're not familiar with Pug (formerly Jade), see our [Jade Tutorial for Beginners](#).

You can install the packages with the following command:

```
npm install -s vision pug
```

The note component

First, we're going to build our note component that is going to be reused across our views. Also, we're going to use this component in some of our controller functions to build a note on the fly in the back-end to simplify the logic on the client.

Create a file in `lib/views/components` called `note.pug` with the following content:

```
article
  h2: a(href=`/note/${note.slug}`)= note.title
  small Published on #{note.date}
  p= note.content
```

It is composed of the title of the note, the publication date and the content of the note.

The base layout

The base layout contains the common elements of our pages; or in other words, for our example, everything that is not content. Create a file in `lib/views/` called `layout.pug` with the following content:

```
doctype html
html(lang='en')
  head
    meta(charset='utf-8')
    meta(http-equiv='x-ua-compatible' content='ie=edge')
    title= page
    meta(name='description' content=description)
    meta(name='viewport' content='width=device-width, initial-scale=1')

    link(href='https://fonts.googleapis.com/css?family=Gentium+Book+Pro')
    link(rel='stylesheet' href='/styles/main.css')
  body
    block content

    script(src='https://code.jquery.com/jquery-3.1.1.min.js' integrity='sha256-hVVnYaiADRBBfFzzB7ewgZJ07Bq46dIvbsVeliU=vK')
    script(src='/scripts/jquery.modal.js')
    script(src='/scripts/main.js')
```

The content of the other pages will be loaded in place of `block content`. Also, note that we will display a `page` variable in the `title` element, and a `description` variable in the `meta(name='description')` element. We will create those variables in our routes later.

We also include, at the bottom of the page, three JS files, [jQuery](#), [jQuery Modal](#) and a `main.js` file which will contain all of our custom JS code for the front-end. Be sure to download those packages and put them in a `static/public/scripts/` directory. We're going to make them public in the [Serving Static Files](#) section.

The home view

On our home page, we will show a list containing all the notes in our database and a button that will show a modal window with a form that allows us to create a new note via Ajax.

Create a file in `lib/views` called `home.pug` with the following content:

```
extends layout
```

```

block content
  header(container)
    h1 Notes Board

  nav
    ul
      // This will show a modal window with a form to send new note
      li: a(href='#note-form' rel='modal:open') Publish

  main(container).notes-list
    // We loop over all the notes received from our controller rendered
    each note in data.notes
      include components/note

    // Form to add a new note, this is used by our controller `create`
    form(action='/note' method='POST').note-form#note-form
      p: input(name='noteTitle' type='text' placeholder='Title...')
      p: input(name='noteDescription' type='text' placeholder='Short description')
      p: textarea(name='noteContent') Write here the content of the note
      p._text-right: input(type='submit' value='Submit')

```

The note view

The note page is pretty similar to the home page, but in this case, we show a menu with options specific to the current note, the content of the note and the same form as in the home page but with the current note information already filled, so it's there when we update it.

Create a file in `lib/views` called `note.pug` with the following content:

```

extends layout

block content
  header(container)
    h1 Notes Board

  nav
    ul
      li: a(href='/') Home
      li: a(href='#note-form' rel='modal:open') Update
      li: a(href=`/note/${note.slug}/delete`) Delete

  main(container).note-content
    include components/note

```

```
form(action=`/note/${note.slug}` method='PUT').note-form#note-form
p: input(name='noteTitle' type='text' value=note.title)
p: input(name='noteDescription' type='text' value=note.description)
p: textarea(name='noteContent')= note.content
p._text-right: input(type='submit' value='Update')
```

The JavaScript on the client

To create and update notes we use the Ajax functionality of jQuery. Although this is not strictly necessary, I feel it provides a better experience for the user.

This is the content of our `main.js` file in the `static/public/scripts/` directory:

```
$('#note-form').submit(function (e) {
  e.preventDefault();

  var form = {
    url: $(this).attr('action'),
    type: $(this).attr('method')
  };

  $.ajax({
    url: form.url,
    type: form.type,
    data: $(this).serialize(),
    success: function (result) {
      $.modal.close();

      if (form.type === 'POST') {
        $('.notes-list').prepend(result);
      } else if (form.type === 'PUT') {
        $('.note-content').html(result);
      }
    }
  });
});
```

Every time the user submits the form in the modal window, we get the information from the form elements and send it to our back-end, depending on the action URL and the method (POST or PUT). Then, we will get the result as a block of HTML containing our new note data. When we add a note, we will just add it on top of the list on the home page, and when we update a note we replace

the content for the new one in the note view.

Adding support for views on the server

To make use of our views, we have to include them in our controllers and add the required settings.

In our `server.js` file, let's import the Node [Path](#) utility at the top of the file, since we're using it in our code to indicate the path of our views.

```
const Path = require('path');
```

Now, replace the `server.route(Routes);` line with the following code block:

```
server.register([
  require('vision')
], (err) => {
  Hoek.assert(!err, err);

  // View settings
  server.views({
    engines: { pug: require('pug') },
    path: Path.join(__dirname, 'lib/views'),
    compileOptions: {
      pretty: false
    },
    isCached: Settings.env === 'production'
  });

  // Add routes
  server.route(Routes);
});
```

In the code we have added, we first register the [Vision](#) plugin with our Hapi.js server, which is going to provide the view functionality. Then, we add the settings for our views — like the engine that we're going to use and the path where the views are located. At the end of the code block, we add again our routes.

This will make work our views on the server, but we still have to declare the view that we're going to use for each route.

Setting the home view

Open the lib/controllers/home.js file and replace the `reply(result);` line with the following:

```
reply.view('home', {
  data: {
    notes: result
  },
  page: 'Home–Notes Board',
  description: 'Welcome to my Notes Board'
});
```

After registering the Vision plugin, we now have a `view` method available on the `reply` object, we're going to use it to select the `home` view in our `views` directory and to send the data that is going to be used when rendering the views.

In the data that we provide to the view, we also include the page title and a meta description for search engines.

Setting the note view: create function

Right now, every time we create a note we get a JSON object from the server to the client. But since we're doing this process with Ajax, we can send the new note as HTML ready to be added to the page. To do this, we render the `note` component with the data we have. Replace the line `reply(result);` with the following code block:

```
// Generate a new note with the 'result' data
const newNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
    note: result
  }
);

reply(newNote);
```

We use the `renderFile` method from Pug to render the note template with the data we just received from our model.

Setting the note view: read function

When we enter a note page, we should get the note template with the content of our note. To do this, we have to replace the `reply(result);` line with this:

```
reply.view('note', {
  note: result,
  page: `${result.title}–Notes Board`,
  description: result.description
});
```

As with the home page, we select a view as the first parameter and the data that we're going to use as the second one.

Setting the note view: update function

Every time we update a note, we will reply similarly to when we create new notes. Replace the `reply(result);` line with the following code:

```
// Generate a new note with the updated data
const updatedNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
    note: result
  }
);

reply(updatedNote);
```

No View for the Delete Function

The delete function doesn't need a view, since it will just redirect to the home page once the note is deleted.

Serving Static Files

The JavaScript and CSS files that we're using on the client side are provided by Hapi.js from the `static/public/` directory. But it won't happen automatically; we have to indicate to the server that we want to define this folder as public. This is done using the [Inert](#) package, which you can install with the following command:

```
npm install -s inert
```

In the `server.register` function inside the `server.js` file, import the Inert plugin and register it with Hapi like this:

```
server.register([
  require('vision'),
  require('inert')
], (err) => {
```

Now we have to define the route where we're going to provide the static files, and their location on our server's filesystem. Add the following entry at the end of the exported object in `routes.js`:

```
{
  // Static files
  method: 'GET',
  path: '/{param*}',
  handler: {
    directory: {
      path: Path.join(__dirname, '../static/public')
    }
  },
  config: {
    description: 'Provides static resources'
  }
}
```

This route will use the `GET` method and we have replaced the `handler` function with an object containing the `directory` that we want to make public.

You can find more information about serving static content in the [Hapi.js documentation](#).

More Files in the Repo

Check the [Github repository](#) for the rest of the [static files](#), like the [main stylesheet](#).

Conclusion

At this point, we have a very basic Hapi.js application using the MVC architecture. Although there are still things that we should take care of before putting our application in production (e.g. input validation, error handling, error pages, etc.) this should work as a foundation to learn and build your own applications.

If you would like to take this example a bit further, after finishing all the small details (not related the architecture) to make this a robust application, you could implement an authentication system so only registered users are able to publish and edit notes. But your imagination is the limit, so feel free to fork the [application repository](#) and experiment with your ideas.

Chapter 4: User Authentication with the MEAN Stack

by Simon Holmes & Jeremy Wilken

In this article, we're going to look at managing user authentication in the MEAN stack. We'll use the most common MEAN architecture of having an Angular single-page app using a REST API built with Node, Express and MongoDB.

When thinking about user authentication, we need to tackle the following things:

1. let a user register
2. save their data, but never directly store their password
3. let a returning user log in
4. keep a logged in user's session alive between page visits
5. have some pages that can only been seen by logged in users
6. change output to the screen depending on logged in status (e.g. a “login” button or a “my profile” button).

Before we dive into the code, let's take a few minutes for a high-level look at how authentication is going to work in the MEAN stack.

The MEAN Stack Authentication Flow

So what does authentication look like in the MEAN stack?

Still keeping this at a high level, these are the components of the flow:

- user data is stored in MongoDB, with the passwords hashed
- CRUD functions are built in an Express API — Create (register), Read (login, get profile), Update, Delete
- an Angular application calls the API and deals with the responses
- the Express API generates a JSON Web Token (JWT, pronounced “Jot”) upon registration or login, and passes this to the Angular application
- the Angular application stores the JWT in order to maintain the user’s session
- the Angular application checks the validity of the JWT when displaying protected views
- the Angular application passes the JWT back to Express when calling protected API routes.

JWTs are preferred over cookies for maintaining the session state in the browser. Cookies are better for maintaining state when using a server-side application.

The Example Application

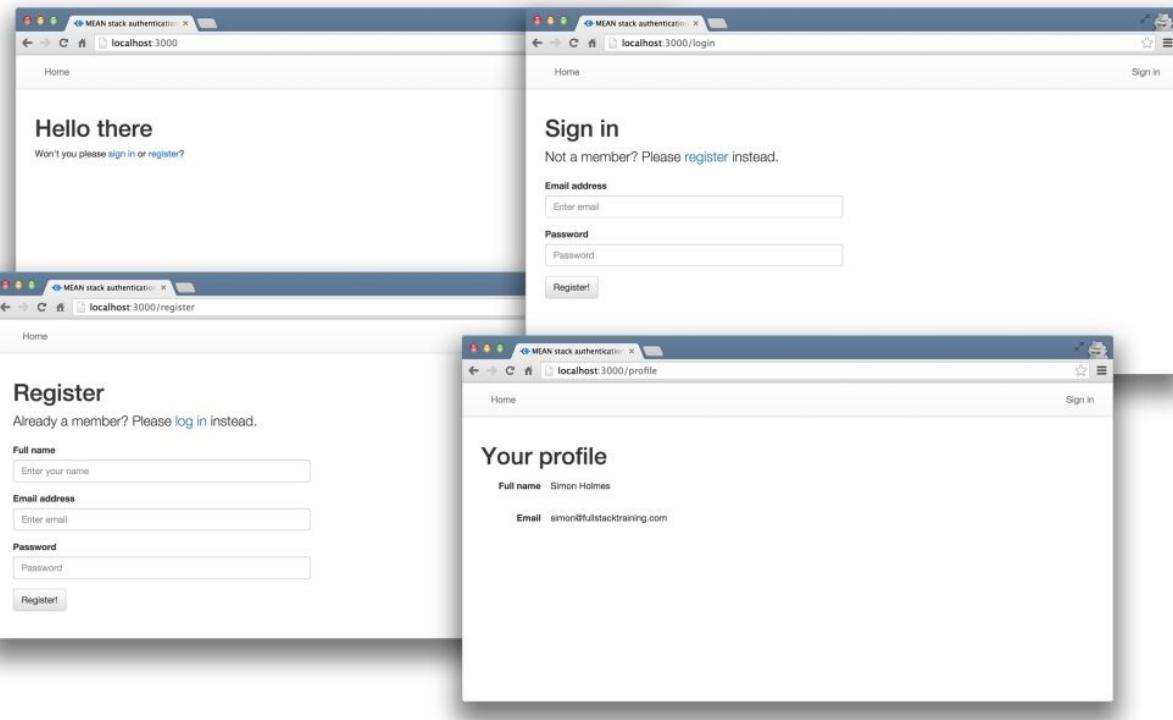
The code for this article is available on [GitHub](#). To run the application, you'll need to have [Node.js installed](#), along with MongoDB. (For instructions on how to install, please refer to [Mongo's official documentation — Windows, Linux, macOS](#)).

The Angular App

To keep the example in this article simple, we'll start with an Angular app with four pages:

1. home page
2. register page
3. login page
4. profile page

The pages are pretty basic and look like this to start with:



The profile page will only be accessible to authenticated users. All the files for the Angular app are in a folder inside the Angular CLI app called `/client`.

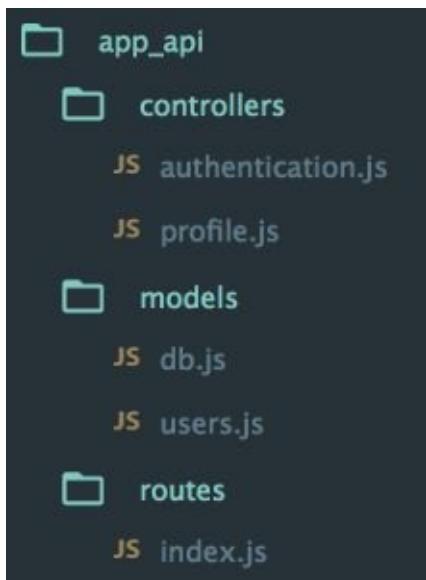
We'll use the Angular CLI for building and running the local server. If you're unfamiliar with the Angular CLI, refer to the [Angular 2 Tutorial: Create a CRUD App with Angular CLI](#) to get started.

The REST API

We'll also start off with the skeleton of a REST API built with Node, Express and MongoDB, using [Mongoose](#) to manage the schemas. This API has three routes:

1. `/api/register` (POST) — to handle new users registering
2. `/api/login` (POST) — to handle returning users logging in
3. `/api/profile/USERID` (GET) — to return profile details when given a `USERID`.

The code for the API is all held in another folder inside the Express app, called `api`. This holds the routes, controllers and model, and is organized like this:



At this starting point, each of the controllers simply responds with a confirmation, like this:

```
module.exports.register = function(req, res) {  
  console.log("Registering user: " + req.body.email);
```

```
res.status(200);
res.json({
  "message" : "User registered: " + req.body.email
});
};
```

Okay, let's get on with the code, starting with the database.

Creating the MongoDB Data Schema with Mongoose

There's a simple user schema defined in [/api/models/users.js](#). It defines the need for an email address, a name, a hash and a salt. The hash and salt will be used instead of saving a password. The `email` is set to unique as we'll use it for the login credentials. Here's the schema:

```
var userSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true
  },
  name: {
    type: String,
    required: true
  },
  hash: String,
  salt: String
});
```

Managing the Password without Saving It

Saving user passwords is a big no-no. Should a hacker get a copy of your database, you want to make sure they can't use it to log in to accounts. This is where the hash and salt come in.

The salt is a string of characters unique to each user. The hash is created by combining the password provided by the user and the salt, and then applying one-way encryption. As the hash can't be decrypted, the only way to authenticate a user is to take the password, combine it with the salt and encrypt it again. If the output of this matches the hash, the password must have been correct.

To do the setting and the checking of the password, we can use Mongoose schema methods. These are essentially functions that you add to the schema. They'll both make use of the Node.js [crypto](#) module.

At the top of the `users.js` model file, require crypto so that we can use it:

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

Nothing needs installing, as crypto ships as part of Node. Crypto itself has several methods; we're interested in `randomBytes` to create the random salt and `pbkdf2Sync` to create the hash (there's much more about Crypto in the [Node.js API docs](#)).

Setting the Password

To save the reference to the password, we can create a new method called `setPassword` on the `userSchema` schema that accepts a `password` parameter. The method will then use `crypto.randomBytes` to set the salt, and `crypto.pbkdf2Sync` to set the hash:

```
userSchema.methods.setPassword = function(password){
  this.salt = crypto.randomBytes(16).toString('hex');
  this.hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64, 'sha512');
};
```

We'll use this method when creating a user. Instead of saving the password to a `password` path, we'll be able to pass it to the `setPassword` function to set the salt and hash paths in the user document.

Checking the Password

Checking the password is a similar process, but we already have the salt from the Mongoose model. This time we just want to encrypt the salt and the password and see if the output matches the stored hash.

Add another new method to the `users.js` model file, called `validPassword`:

```
userSchema.methods.validPassword = function(password) {
  var hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64, 'sha512');
  return this.hash === hash;
};
```

Generating a JSON Web Token (JWT)

One more thing the Mongoose model needs to be able to do is generate a [JWT](#), so that the API can send it out as a response. A Mongoose method is ideal here too, as it means we can keep the code in one place and call it whenever needed. We'll need to call it when a user registers and when a user logs in.

To create the JWT, we'll use a module called [jsonwebtoken](#) which needs to be installed in the application, so run this on the command line:

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

Then require this in the `users.js` model file:

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

This module exposes a `sign` method that we can use to create a JWT, simply passing it the data we want to include in the token, plus a secret that the hashing algorithm will use. The data should be sent as a JavaScript object, and include an expiry date in an `exp` property.

Adding a `generateJwt` method to `userSchema` in order to return a JWT looks like this:

```
userSchema.methods.generateJwt = function() {
  var expiry = new Date();
  expiry.setDate(expiry.getDate() + 7);

  return jwt.sign({
    _id: this._id,
    email: this.email,
    name: this.name,
    exp: parseInt(expiry.getTime() / 1000),
  }, "MY_SECRET"); // DO NOT KEEP YOUR SECRET IN THE CODE!
};
```

Keep the Secret Safe!

It's important that your secret is kept safe: only the originating server should know what it is. It's best practice to set the secret as an environment variable, and not have it in the source code, especially if your code is stored in version control somewhere.

That's everything we need to do with the database.

Set Up Passport to Handle the Express Authentication

Passport is a Node module that simplifies the process of handling authentication in Express. It provides a common gateway to work with many different authentication “strategies”, such as logging in with Facebook, Twitter or Oauth. The strategy we’ll use is called “local”, as it uses a username and password stored locally.

To use Passport, first install it and the strategy, saving them in package.json:

```
npm install passport --save
npm install passport-local --save
```

Configure Passport

Inside the api folder, create a new folder config and create a file in there called `passport.js`. This is where we define the strategy.

Before defining the strategy, this file needs to require Passport, the strategy, Mongoose and the User model:

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var mongoose = require('mongoose');
var User = mongoose.model('User');
```

For a local strategy, we essentially just need to write a Mongoose query on the User model. This query should find a user with the email address specified, and then call the `validPassword` method to see if the hashes match. Pretty simple.

There’s just one curiosity of Passport to deal with. Internally, the local strategy for Passport expects two pieces of data called `username` and `password`. However, we’re using `email` as our unique identifier, not `username`. This can be configured in an options object with a `usernameField` property in the strategy definition. After that, it’s over to the Mongoose query.

So all in, the strategy definition will look like this:

```

passport.use(new LocalStrategy({
    usernameField: 'email'
},
function(username, password, done) {
    User.findOne({ email: username }, function (err, user) {
        if (err) { return done(err); }
        // Return if user not found in database
        if (!user) {
            return done(null, false, {
                message: 'User not found'
            });
        }
        // Return if password is wrong
        if (!user.validPassword(password)) {
            return done(null, false, {
                message: 'Password is wrong'
            });
        }
        // If credentials are correct, return the user object
        return done(null, user);
    });
});
));

```

Note how the `validPassword` schema method is called directly on the `user` instance.

Now Passport just needs to be added to the application. So in `app.js` we need to require the Passport module, require the Passport config and initialize Passport as middleware. The placement of all of these items inside `app.js` is quite important, as they need to fit into a certain sequence.

The Passport module should be required at the top of the file with the other general `require` statements:

```

var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var passport = require('passport');

```

The config should be required *after* the model is required, as the config references the model.

```
require('./api/models/db');
require('./api/config/passport');
```

Finally, Passport should be initialized as Express middleware just before the API routes are added, as these routes are the first time that Passport will be used.

```
app.use(passport.initialize());
app.use('/api', routesApi);
```

We've now got the schema and Passport set up. Next, it's time to put these to use in the routes and controllers of the API.

Configure API Endpoints

With the API we've got two things to do:

1. make the controllers functional
2. secure the /api/profile route so that only authenticated users can access it.

Code the Register and Login API Controllers

In the example app the register and login controllers are in [`/api/controllers/authentication.js`](#). In order for the controllers to work, the file needs to require Passport, Mongoose and the user model:

```
var passport = require('passport');
var mongoose = require('mongoose');
var User = mongoose.model('User');
```

The Register API Controller

The register controller needs to do the following:

1. take the data from the submitted form and create a new Mongoose model instance
2. call the setPassword method we created earlier to add the salt and the hash to the instance
3. save the instance as a record to the database
4. generate a JWT
5. send the JWT inside the JSON response.

In code, all of that looks like this:

```
module.exports.register = function(req, res) {
  var user = new User();

  user.name = req.body.name;
  user.email = req.body.email;

  user.setPassword(req.body.password);
```

```
user.save(function(err) {
  var token;
  token = user.generateJwt();
  res.status(200);
  res.json({
    "token" : token
  });
});
```

This makes use of the `setPassword` and `generateJwt` methods we created in the Mongoose schema definition. See how having that code in the schema makes this controller really easy to read and understand.

Don't forget that, in reality, this code would have a number of error traps, validating form inputs and catching errors in the `save` function. They're omitted here to highlight the main functionality of the code.

The Login API Controller

The login controller hands over pretty much all control to Passport, although you could (and should) add some validation beforehand to check that the required fields have been sent.

For Passport to do its magic and run the strategy defined in the config, we need to call the `authenticate` method as shown below. This method will call a callback with three possible parameters `err`, `user` and `info`. If `user` is defined, it can be used to generate a JWT to be returned to the browser:

```
module.exports.login = function(req, res) {

  passport.authenticate('local', function(err, user, info){
    var token;

    // If Passport throws/catches an error
    if (err) {
      res.status(404).json(err);
      return;
    }

    // If a user is found
    if(user){
```

```
        token = user.generateJwt();
        res.status(200);
        res.json({
            "token" : token
        });
    } else {
        // If user is not found
        res.status(401).json(info);
    }
})(req, res);

};
```

Securing an API Route

The final thing to do in the back end is make sure that only authenticated users can access the /api/profile route. The way to validate a request is to ensure that the JWT sent with it is genuine, by using the secret again. This is why you should keep it a secret and not in the code.

Configuring the Route Authentication

First we need to install a piece of middleware called express-jwt:

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

Then we need to require it and configure it in the file where the routes are defined. In the sample application, this is [/api/routes/index.js](#). Configuration is a case of telling it the secret, and — optionally — the name of the property to create on the req object that will hold the JWT. We'll be able to use this property inside the controller associated with the route. The default name for the property is user, but this is the name of an instance of our Mongoose User model, so we'll set it to payload to avoid confusion:

```
var jwt = require('express-jwt');
var auth = jwt({
    secret: 'MY_SECRET',
    userProperty: 'payload'
});
```

Again, *don't keep the secret in the code!*

Applying the Route Authentication

To apply this middleware, simply reference the function in the middle of the route to be protected, like this:

```
router.get('/profile', auth, ctrlProfile.profileRead);
```

If someone tries to access that route now without a valid JWT, the middleware will throw an error. To make sure our API plays nicely, catch this error and return a 401 response by adding the following into the error handlers section of the main app.js file:

```
// error handlers
// Catch unauthorised errors
app.use(function (err, req, res, next) {
  if (err.name === 'UnauthorizedError') {
    res.status(401);
    res.json({"message" : err.name + ": " + err.message});
  }
});
```

Using the Route Authentication

In this example, we only want people to be able to view their own profiles, so we get the user ID from the JWT and use it in a Mongoose query.

The controller for this route is in [/api/controllers/profile.js](#). The entire contents of this file look like this:

```
var mongoose = require('mongoose');
var User = mongoose.model('User');

module.exports.profileRead = function(req, res) {

  // If no user ID exists in the JWT return a 401
  if (!req.payload._id) {
    res.status(401).json({
      "message" : "UnauthorizedError: private profile"
    });
  } else {
    // Otherwise continue
    User
      .findById(req.payload._id)
```

```
    .exec(function(err, user) {
      res.status(200).json(user);
    });
};

};
```

Naturally, this should be fleshed out with some more error trapping — for example, if the user isn't found — but this snippet is kept brief to demonstrate the key points of the approach.

That's it for the back end. The database is configured, we have API endpoints for registering and logging in that generate and return a JWT, and also a protected route. On to the front end!

Create Angular Authentication Service

Most of the work in the front end can be put into an Angular service, creating methods to manage:

- saving the JWT in local storage
- reading the JWT from local storage
- deleting the JWT from local storage
- calling the register and login API endpoints
- checking whether a user is currently logged in
- getting the details of the logged-in user from the JWT.

We'll need to create a new service called `AuthenticationService`. With the CLI, this can be done by running `ng generate service authentication`, and making sure it's listed in the app module providers. In the example app, this is in the file [`/client/src/app/authentication.service.ts`](#).

Local Storage: Saving, Reading and Deleting a JWT

To keep a user logged in between visits, we use `localStorage` in the browser to save the JWT. An alternative is to use `sessionStorage`, which will only keep the token during the current browser session.

First, we want to create a few interfaces to handle the data types. This is useful for type checking our application. The profile returns an object formatted as `UserDetails`, and the login and register endpoints expect a `TokenPayload` during the request and return a `TokenResponse` object:

```
export interface UserDetails {
  _id: string;
  email: string;
  name: string;
  exp: number;
  iat: number;
}

interface TokenResponse {
  token: string;
}
```

```
export interface TokenPayload {  
  email: string;  
  password: string;  
  name?: string;  
}
```

This service uses the `HttpClient` service from Angular to make HTTP requests to our server application (which we'll use in a moment) and the `Router` service to navigate programmatically. We must inject them into our service constructor.

Then we define four methods that interact with the JWT token. We implement `saveToken` to handle storing the token into `localStorage` and onto the `token` property, a `getToken` method to retrieve the token from `localStorage` or from the `token` property, and a `logout` function that removes the JWT token from memory and redirects to the home page.

It's important to note that this code doesn't run if you're using server-side rendering, because APIs like `localStorage` and `window.atob` aren't available, and there are details about solutions to address server-side rendering in the Angular documentation.

```
import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { Observable } from 'rxjs/Observable';  
import { map } from 'rxjs/operators/map';  
import { Router } from '@angular/router';  
  
// Interfaces here  
  
@Injectable()  
export class AuthenticationService {  
  private token: string;  
  
  constructor(private http: HttpClient, private router: Router) {}  
  
  private saveToken(token: string): void {  
    localStorage.setItem('mean-token', token);  
    this.token = token;  
  }  
  
  private getToken(): string {  
    if (!this.token) {  
      this.token = localStorage.getItem('mean-token');  
    }  
  }
```

```
    return this.token;
}

public logout(): void {
  this.token = '';
  window.localStorage.removeItem('mean-token');
  this.router.navigateByUrl('/');
}
}
```

Now let's add a method to check for this token — and the validity of the token — to find out if the visitor is logged in.

Getting Data from a JWT

When we set the data for the JWT (in the `generateJwt` Mongoose method) we included the expiry date in an `exp` property. But if you look at a JWT, it seems to be a random string, like this following example:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfaWQiOiI1NWQ0MjNjMTUxMzcxMmN
```

So how do you read a JWT?

A JWT is actually made up of three separate strings, separated by a dot .. These three parts are:

1. *Header* — an encoded JSON object containing the type and the hashing algorithm used
2. *Payload* — an encoded JSON object containing the data, the real body of the token
3. *Signature* — an encrypted hash of the header and payload, using the “secret” set on the server.

It's the second part we're interested in here — the payload. Note that this is *encoded* rather than encrypted, meaning that we can *decode* it.

There's a function called `atob()` that's native to modern browsers, and which will decode a Base64 string like this.

So we need to get the second part of the token, decode it and parse it as JSON. Then we can check that the expiry date hasn't passed.

At the end of it, the `getUserDetails` function should return an object of the `UserDetails` type or `null`, depending on whether a valid token is found or not. Put together, it looks like this:

```
public getUserDetails(): UserDetails {
  const token = this.getToken();
  let payload;
  if (token) {
    payload = token.split('.')[1];
    payload = window.atob(payload);
    return JSON.parse(payload);
  } else {
    return null;
  }
}
```

The user details that are provided include the information about the user's name, email, and the expiration of the token, which we'll use to check if the user session is valid.

Check Whether a User Is Logged In

Add a new method called `isLoggedIn` to the service. It uses the `getUserDetails` method to get the token details from the JWT token and checks the expiration hasn't passed yet:

```
public isLoggedIn(): boolean {
  const user = this.getUserDetails();
  if (user) {
    return user.exp > Date.now() / 1000;
  } else {
    return false;
  }
}
```

If the token exists, the method will return if the user is logged in as a boolean value. Now we can construct our HTTP requests to load data, using the token for authorization.

Structuring the API Calls

To facilitate making API calls, add the `request` method to the

AuthenticationService, which is able to construct and return the proper HTTP request observable depending on the specific type of request. It's a private method, since it's only used by this service, and exists just to reduce code duplication. This will use the Angular HttpClient service; remember to inject this into the AuthenticationService if it's not already there:

```
private request(method: 'post' | 'get', type: 'login' | 'register' | 'profile': string, user: TokenPayload): Observable<any> {
  let base;

  if (method === 'post') {
    base = this.http.post(`api/${type}`, user);
  } else {
    base = this.http.get(`api/${type}`, { headers: { Authorization: `Bearer ${user.token}` } });
  }

  const request = base.pipe(
    map((data: TokenResponse) => {
      if (data.token) {
        this.saveToken(data.token);
      }
      return data;
    })
  );
  return request;
}
```

It does require the `map` operator from RxJS in order to intercept and store the token in the service if it's returned by an API login or register call. Now we can implement the public methods to call the API.

Calling the Register and Login API Endpoints

Just three methods to add. We'll need an interface between the Angular app and the API, to call the login and register endpoints and save the returned token, or the profile endpoint to get the user details:

```
public register(user: TokenPayload): Observable<any> {
  return this.request('post', 'register', user);
}

public login(user: TokenPayload): Observable<any> {
  return this.request('post', 'login', user);
}
```

```
public profile(): Observable<any> {
  return this.request('get', 'profile');
}
```

Each method returns an observable that will handle the HTTP request for one of the API calls we need to make. That finalizes the service; now to tie everything together in the Angular app.

Apply Authentication to Angular App

We can use the AuthenticationService inside the Angular app in a number of ways to give the experience we're after:

1. wire up the register and sign-in forms
2. update the navigation to reflect the user's status
3. only allow logged-in users access to the /profile route
4. call the protected /api/profile API route.

Connect the Register and Login Controllers

We'll begin by looking at the register and login forms.

The Register Page

The HTML for the registration form already exists and has NgModel directives attached to the fields, all bound to properties set on the credentials controller property. The form also has a (submit) event binding to handle the submission. In the example application, it's in

</client/src/app/register/register.component.html> and looks like this:

```
<form (submit)="register()">
  <div class="form-group">
    <label for="name">Full name</label>
    <input type="text" class="form-control" name="name" placeholder=</div>
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" name="email" placeholder=</div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" name="password" placeholder=</div>
  <button type="submit" class="btn btn-default">Register!</button>
</form>
```

The first task in the controller is to ensure our AuthenticationService and the Router are injected and available through the constructor. Next, inside the

register handler for the form submit, make a call to auth.register, passing it the credentials from the form.

The register method returns an observable, which we need to subscribe to in order to trigger the request. The observable will emit success or failure, and if someone has successfully registered, we'll set the application to redirect them to the profile page or log the error in the console.

In the sample application, the controller is in

[`/client/src/app/register/register.component.ts`](#) and looks like this:

```
import { Component } from '@angular/core';
import { AuthenticationService, TokenPayload } from '../authentication';
import { Router } from '@angular/router';

@Component({
  templateUrl: './register.component.html'
})
export class RegisterComponent {
  credentials: TokenPayload = {
    email: '',
    name: '',
    password: ''
  };

  constructor(private auth: AuthenticationService, private router: Router) {}

  register() {
    this.auth.register(this.credentials).subscribe(() => {
      this.router.navigateByUrl('/profile');
    }, (err) => {
      console.error(err);
    });
  }
}
```

The Login Page

The login page is very similar in nature to the register page, but in this form we don't ask for the name, just email and password. In the sample application, it's in [`/client/src/app/login/login.component.html`](#) and looks like this:

```
<form (submit)="login()">
```

```

<div class="form-group">
  <label for="email">Email address</label>
  <input type="email" class="form-control" name="email" placeholder="Email address" ngModel>
</div>
<div class="form-group">
  <label for="password">Password</label>
  <input type="password" class="form-control" name="password" placeholder="Password" ngModel>
</div>
<button type="submit" class="btn btn-default">Sign in!</button>
</form>

```

Once again, we have the form submit handler, and `NgModel` attributes for each of the inputs. In the controller, we want the same functionality as the register controller, but this time called the `login` method of the `AuthenticationService`.

In the sample application, the controller is in

[`/client/src/app/login/login.controller.ts`](#) and look like this:

```

import { Component } from '@angular/core';
import { AuthenticationService, TokenPayload } from '../authentication';
import { Router } from '@angular/router';

@Component({
  templateUrl: './login.component.html'
})
export class LoginComponent {
  credentials: TokenPayload = {
    email: '',
    password: ''
  };

  constructor(private auth: AuthenticationService, private router: Router) {}

  login() {
    this.auth.login(this.credentials).subscribe(() => {
      this.router.navigateByUrl('/profile');
    }, (err) => {
      console.error(err);
    });
  }
}

```

Now users can register and sign in to the application. Note that, again, there should be more validation in the forms to ensure that all required fields are filled before submitting. These examples are kept to the bare minimum to highlight the

main functionality.

Change Content Based on User Status

In the navigation, we want to show the *Sign in* link if a user isn't logged in, and their username with a link to the profile page if they are logged in. The navbar is found in the App component.

First, we'll look at the App component controller. We can inject the AuthenticationService into the component and call it directly in our template. In the sample app, the file is in [/client/src/app/app.component.ts](#) and looks like this:

```
import { Component } from '@angular/core';
import { AuthenticationService } from './authentication.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  constructor(public auth: AuthenticationService) {}
}
```

That's pretty simple, right? Now, in the associated template we can use `auth.isLoggedIn()` to determine whether to display the sign-in link or the profile link. To add the user's name to the profile link, we can access the `name` property of `auth.getUserDetails()?.name`. Remember that this is getting the data from the JWT. The `?.` operator is a special way to access a property on an object that may be undefined, without throwing an error.

In the sample app, the file is in [/client/src/app/app.component.html](#) and the updated part looks like this:

```
<ul class="nav navbar-nav navbar-right">
  <li *ngIf="!auth.isLoggedIn()"><a routerLink="/login">Sign in</a><
  <li *ngIf="auth.isLoggedIn()"><a routerLink="/profile">{{ auth.get
    <li *ngIf="auth.isLoggedIn()"><a (click)="auth.logout()">Logout</a
  </ul>
```

Protect a Route for Logged in Users Only

In this step, we'll see how to make a route accessible only to logged-in users, by protecting the /profile path.

Angular allows you to define a route guard, which can run a check at several points of the routing life cycle to determine if the route can be loaded. We'll use the `CanActivate` hook to tell Angular to load the profile route only if the user is logged in.

To do, this we need to create a route guard service, `ng generate service auth-guard`. It must implement the `CanActivate` interface, and the associated `canActivate` method. This method returns a boolean value from the `AuthenticationService.isLoggedIn` method (basically checks if the token is found, and still valid), and if the user is not valid also redirects them to the home page:

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthenticationService } from './authentication.service';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(private auth: AuthenticationService, private router: Router) {}

  canActivate() {
    if (!this.auth.isLoggedIn()) {
      this.router.navigateByUrl('/');
      return false;
    }
    return true;
  }
}
```

To enable this guard, we have to declare it on the route configuration. There's a property called `canActivate`, which takes an array of services that should be called before activating the route. Ensure you also declare these services in the App NgModule's providers array. The routes are defined in the [App module](#), which contains the routes like you see here:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
```

```
{ path: 'profile', component: ProfileComponent, canActivate: [AuthGuard]};
```

With that route guard in place, now if an unauthenticated user tries to visit the profile page, Angular will cancel the route change and redirect to the home page, thus protecting it from unauthenticated users.

Call a Protected API Route

The `/api/profile` route has been set up to check for a JWT in the request. Otherwise, it will return a 401 unauthorized error.

To pass the token to the API, it needs to be sent through as a header on the request, called `Authorization`. The following snippet shows the main data service function, and the format required to send the token. The `AuthenticationService` already handles this, but you can find this in [`/client/src/app/authentication.service.ts`](#).

```
base = this.http.get(`/api/${type}`, { headers: { Authorization: `Be
```

Remember that the back-end code is validating that the token is genuine when the request is made, by using the secret known only to the issuing server.

To make use of this in the profile page, we just need to update the controller, in [`/client/src/app/profile/profile.component.ts`](#) in the sample app. This will populate a `details` property when the API returns some data, which should match the `UserDetails` interface.

```
import { Component } from '@angular/core';
import { AuthenticationService, UserDetails } from '../authentication.service';

@Component({
  templateUrl: './profile.component.html'
})
export class ProfileComponent {
  details: UserDetails;

  constructor(private auth: AuthenticationService) {}

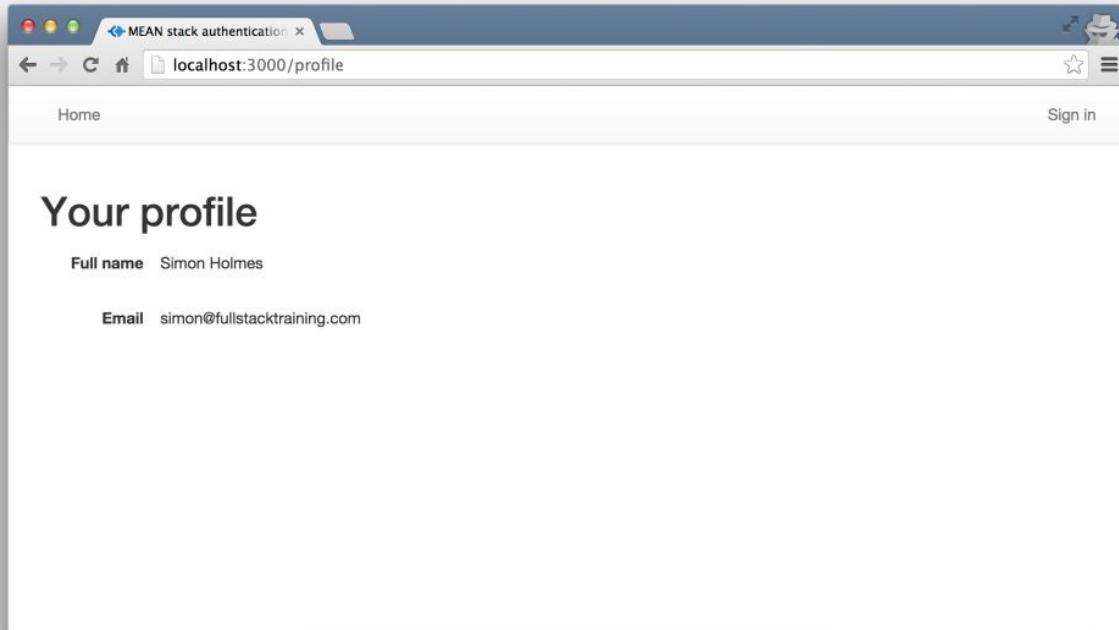
  ngOnInit() {
    this.auth.profile().subscribe(user => {
      this.details = user;
```

```
    }, (err) => {
      console.error(err);
    });
}
}
```

Then, of course, it's just a case of updating the bindings in the view ([/client/src/app/profile/profile.component.html](#)). Again, the `?.` is a safety operator for binding properties that don't exist on first render (since data has to load first).

```
<div class="form-horizontal">
  <div class="form-group">
    <label class="col-sm-3 control-label">Full name</label>
    <p class="form-control-static">{{ details?.name }}</p>
  </div>
  <div class="form-group">
    <label class="col-sm-3 control-label">Email</label>
    <p class="form-control-static">{{ details?.email }}</p>
  </div>
</div>
```

And here's the final profile page, when logged in:



That's how to manage authentication in the MEAN stack, from securing API routes and managing user details to working with JWTs and protecting routes.

Chapter 5: Build a JavaScript Command Line Interface (CLI) with Node.js

by Lukas White & Michael Wanyoike

As great as Node.js is for “traditional” web applications, its potential uses are far broader. Microservices, REST APIs, tooling, working with the Internet of Things and even desktop applications: it’s got your back.

Another area where Node.js is really useful is for building command-line applications — and that’s what we’re going to be doing in this article. We’re going to start by looking at a number of third-party packages designed to help work with the command line, then build a real-world example from scratch.

What we’re going to build is a tool for initializing a Git repository. Sure, it’ll run `git init` under the hood, but it’ll do more than just that. It will also create a remote repository on GitHub right from the command line, allow the user to interactively create a `.gitignore` file, and finally perform an initial commit and push.

As ever, the code accompanying this tutorial can be found on our [GitHub repo](#).

Why Build a Command-line Tool with Node.js?

Before we dive in and start building, it's worth looking at why we might choose Node.js to build a command-line application.

The most obvious advantage is that, if you're reading this, you're probably already familiar with it — and, indeed, with JavaScript.

Another key advantage, as we'll see as we go along, is that the strong Node.js ecosystem means that among the hundreds of thousands of packages available for all manner of purposes, there are a number which are specifically designed to help build powerful command-line tools.

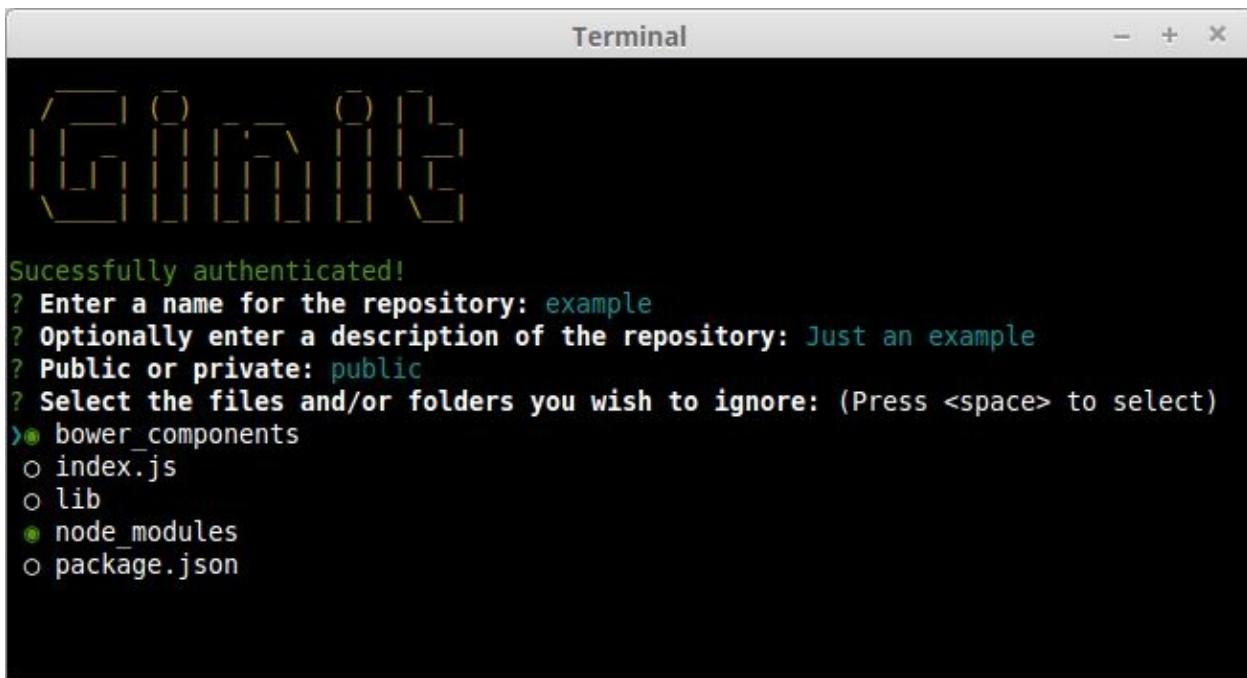
Finally, we can use `npm` to manage any dependencies, rather than have to worry about OS-specific package managers such as Aptitude, Yum or Homebrew.

Although Your Tool May Have Other Dependencies

That isn't necessarily true, in that your command-line tool may have other external dependencies.

Tip: that isn't necessarily true, in that your command-line tool may have other external dependencies.

What We're Going to Build: ginit



A screenshot of a terminal window titled "Terminal". The window shows a series of ASCII art icons representing a user's profile picture. Below the icons, the text "Successfully authenticated!" is displayed in green. The terminal then prompts the user to enter a repository name ("? Enter a name for the repository: example"), a description ("? Optionally enter a description of the repository: Just an example"), and a visibility setting ("? Public or private: public"). Finally, it asks the user to select files to ignore, listing "bower_components", "index.js", "lib", "node_modules" (which is highlighted with a green dot), and "package.json".

For this tutorial, We're going to create a command-line utility which I'm calling **ginit**. It's `git init`, but on steroids.

You're probably wondering what on earth that means.

As you no doubt already know, `git init` initializes a git repository in the current folder. However, that's usually only one of a number of repetitive steps involved in the process of hooking up a new or existing project to Git. For example, as part of a typical workflow, you may well:

1. initialise the local repository by running `git init`
2. create a remote repository, for example on GitHub or Bitbucket — typically by leaving the command line and firing up a web browser
3. add the remote
4. create a `.gitignore` file
5. add your project files
6. commit the initial set of files
7. push up to the remote repository.

There are often more steps involved, but we'll stick to those for the purposes of

our app. Nevertheless, these steps are pretty repetitive. Wouldn't it be better if we could do all this from the command line, with no copy-pasting of Git URLs and such like?

So what ginit will do is create a Git repository in the current folder, create a remote repository — we'll be using GitHub for this — and then add it as a remote. Then it will provide a simple interactive “wizard” for creating a `.gitignore` file, add the contents of the folder and push it up to the remote repository. It might not save you hours, but it'll remove some of the initial friction when starting a new project.

With that in mind, let's get started.

The Application Dependencies

One thing is for certain: in terms of appearance, the console will never have the sophistication of a graphical user interface. Nevertheless, that doesn't mean it has to be plain, ugly, monochrome text. You might be surprised by just how much you can do visually, while at the same time keeping it functional. We'll be looking at a couple of libraries for enhancing the display: [chalk](#) for colorizing the output and [clui](#) to add some additional visual components. Just for fun, we'll use [figlet](#) to create a fancy ASCII-based banner and we'll also use [clear](#) to clear the console.

In terms of input and output, the low-level [Readline](#) Node.js module could be used to prompt the user and request input, and in simple cases is more than adequate. But we're going to take advantage of a third-party package which adds a greater degree of sophistication — [Inquirer](#). As well as providing a mechanism for asking questions, it also implements simple input controls: think radio buttons and checkboxes, but in the console.

We'll also be using [minimist](#) to parse command-line arguments.

Here's a complete list of the packages we'll use specifically for developing on the command line:

- [chalk](#) — colorizes the output
- [clear](#) — clears the terminal screen
- [clui](#) — draws command-line tables, gauges and spinners
- [figlet](#) — creates ASCII art from text
- [inquirer](#) — creates interactive command-line user interface
- [minimist](#) — parses argument options
- [configstore](#) — easily loads and saves config without you having to think about where and how.

Additionally, we'll also be using the following:

- [@octokit/rest](#) — a GitHub REST API client for Node.js
- [lodash](#) — a JavaScript utility library
- [simple-git](#) — a tool for running Git commands in a Node.js application
- [touch](#) — a tool for implementing the Unix touch command.

Getting Started

Although we're going to create the application from scratch, don't forget that you can also grab a copy of the code from the [repository which accompanies this article](#).

Create a new directory for the project. You don't have to call it ginit, of course:

```
mkdir ginit  
cd ginit
```

Create a new package.json file:

```
npm init
```

Follow the simple wizard, for example:

```
name: (ginit)  
version: (1.0.0)  
description: "git init" on steroids  
entry point: (index.js)  
test command:  
git repository:  
keywords: Git CLI  
author: [YOUR NAME]  
license: (ISC)
```

Now install the dependencies:

```
npm install chalk clear clui figlet inquirer minimist configstore @c
```

Alternatively, simply copy-paste the following package.json file — modifying the author appropriately — or grab it from the [repository which accompanies this article](#):

```
{  
  "name": "ginit",  
  "version": "1.0.0",  
  "description": "\"git init\" on steroids",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
}
```

```
},
"keywords": [
  "Git",
  "CLI"
],
"author": "Lukas White <hello@lukaswhite.com>",
"license": "ISC",
"bin": {
  "ginit": "./index.js"
},
"dependencies": {
  "@octokit/rest": "^14.0.5",
  "chalk": "^2.3.0",
  "clear": "0.0.1",
  "clui": "^0.3.6",
  "configstore": "^3.1.1",
  "figlet": "^1.2.0",
  "inquirer": "^5.0.1",
  "lodash": "^4.17.4",
  "minimist": "^1.2.0",
  "simple-git": "^1.89.0",
  "touch": "^3.1.0"
}
}
```

Now create an `index.js` file in the same folder and require the following dependencies:

```
const chalk      = require('chalk');
const clear      = require('clear');
const figlet      = require('figlet');
```

Adding Some Helper Methods

We're going to create a `lib` folder where we'll split our helper code into modules:

- `files.js` — basic file management
- `inquirer.js` — command-line user interaction
- `github.js` — access token management
- `repo.js` — Git repository management.

Let's start with `lib/files.js`. Here, we need to:

- get the current directory (to get a default repo name)
- check whether a directory exists (to determine whether the current folder is already a Git repository by looking for a folder named `.git`).

This sounds straightforward, but there are a couple of gotchas to take into consideration.

Firstly, you might be tempted to use the `fs` module's [`realpathSync`](#) method to get the current directory:

```
path.basename(path.dirname(fs.realpathSync(__filename)));
```

This will work when we're calling the application from the same directory (e.g. using `node index.js`), but bear in mind that we're going to be making our console application available globally. This means we'll want the name of the directory we're working in, not the directory where the application resides. For this purpose, it's better to use [`process.cwd`](#):

```
path.basename(process.cwd());
```

Secondly, the preferred method of checking whether a file or directory exists [keeps changing](#). The current way is to use `fs.stat` or `fs.statSync`. These throw an error if there's no file, so we need to use a `try ... catch` block.

Finally, it's worth noting that when you're writing a command-line application, using the synchronous version of these sorts of methods is just fine.

Putting that all together, let's create a utility package in lib/files.js:

```
const fs = require('fs');
const path = require('path');

module.exports = {
  getCurrentDirectoryBase : () => {
    return path.basename(process.cwd());
  },

  directoryExists : (filePath) => {
    try {
      return fs.statSync(filePath).isDirectory();
    } catch (err) {
      return false;
    }
  }
};
```

Go back to index.js and ensure you require the new file:

```
const files = require('./lib/files');
```

With this in place, we can start developing the application.

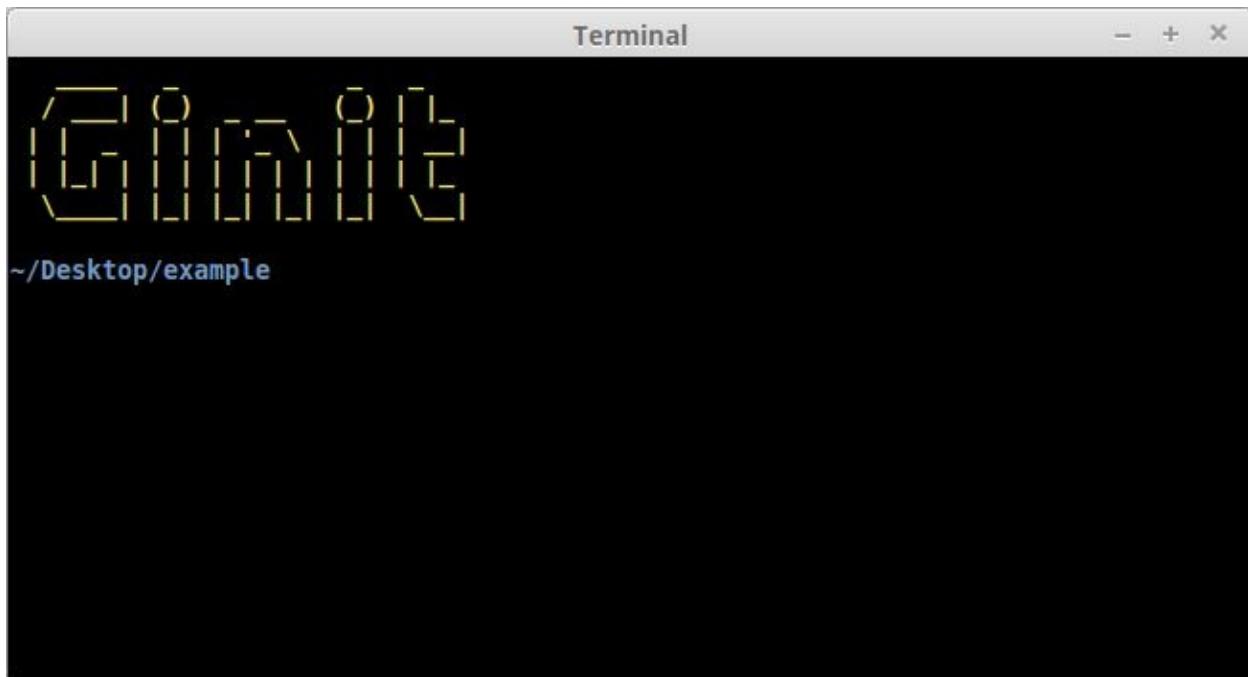
Initializing the Node CLI

Now let's implement the start-up phase of our console application.

In order to demonstrate some of the packages we've installed to enhance the console output, let's clear the screen and then display a banner:

```
clear();
console.log(
  chalk.yellow(
    figlet.textSync('Ginit', { horizontalLayout: 'full' })
  )
);
```

The output from this is shown below.



Next up, let's run a simple check to ensure that the current folder isn't already a Git repository. That's easy: we just check for the existence of a .git folder using the utility method we just created:

```
if (files.existsSync('.git')) {
  console.log(chalk.red('Already a git repository!'));
  process.exit();
```

}

Coloring the Message

Notice we're using the [chalk module](#) to show a red-colored message.

Prompting the User for Input

The next thing we need to do is create a function that will prompt the user for their GitHub credentials.

We can use [Inquirer](#) for this. The module includes a number of methods for various types of prompts, which are roughly analogous to HTML form controls. In order to collect the user's GitHub username and password, we're going to use the `input` and `password` types respectively.

First, create `lib/inquirer.js` and insert this code:

```
const inquirer = require('inquirer');
const files = require('./files');

module.exports = {

  askGithubCredentials: () => {
    const questions = [
      {
        name: 'username',
        type: 'input',
        message: 'Enter your GitHub username or e-mail address:',
        validate: function( value ) {
          if (value.length) {
            return true;
          } else {
            return 'Please enter your username or e-mail address.';
          }
        }
      },
      {
        name: 'password',
        type: 'password',
        message: 'Enter your password:',
        validate: function(value) {
          if (value.length) {
            return true;
          } else {
            return 'Please enter your password.';
          }
        }
      }
    ]
  }
}
```

```
];
    return inquirer.prompt(questions);
},
}
```

As you can see, `inquirer.prompt()` asks the user a series of questions, provided in the form of an array as the first argument. Each question is made up of an object which defines the name of the field, the type (we're just using `input` and `password` respectively here, but later we'll look at a more advanced example), and the prompt (`message`) to display.

The input the user provides will be passed in to the calling function as a promise. If successful, we'll end up with a simple object with two properties; `username` and `password`.

You can test all of this out by adding the following to `index.js`:

```
const inquirer = require('./lib/inquirer');

const run = async () => {
  const credentials = await inquirer.askGithubCredentials();
  console.log(credentials);
}

run();
```

Then run the script using `node index.js`.

Terminal

```
[~] ? Enter your Github username or e-mail address: mattdamon
? Enter your password: *****
{ '0': { username: 'mattdamon', password: 'secret' } }
~/Desktop/example
```

Dealing With GitHub Authentication

The next step is to create a function to retrieve an OAuth token for the GitHub API. Essentially, we're going to "exchange" the username and password for a token.

Of course, we don't want users to have to enter their credentials every time they use the tool. Instead, we'll store the OAuth token for subsequent requests. This is where the [configstore](#) package comes in.

Storing Config

Storing config is outwardly quite straightforward: you can simply read and write to/from a JSON file without the need for a third-party package. However, the configstore package provides a few key advantages:

1. It determines the most appropriate location for the file for you, taking into account your operating system and the current user.
2. There's no need to explicitly read or write to the file. You simply modify a configstore object and that's taken care of for you in the background.

To use it, simply create an instance, passing it an application identifier. For example:

```
const Configstore = require('configstore');
const conf = new Configstore('ginit');
```

If the configstore file doesn't exist, it'll return an empty object and create the file in the background. If there's already a configstore file, the contents will be decoded into JSON and made available to your application. You can now use `conf` as a simple object, getting or setting properties as required. As mentioned above, you don't need to worry about saving it afterwards. That gets taken care of for you.

On macOS/Linux

On macOS/Linux, you'll find the file in `/Users/[YOUR-USERNME]/.config/configstore/ginit.json`

Communicating with the GitHub API

Let's create a library for handling the GitHub token. Create the file `lib/github.js` and place the following code inside it:

```
const octokit      = require('@octokit/rest')();
const Configstore = require('configstore');
const pkg          = require('../package.json');
const _            = require('lodash');
const CLI          = require('clui');
const Spinner      = CLI.Spinner;
const chalk        = require('chalk');

const inquirer     = require('./inquirer');

const conf = new Configstore(pkg.name);
```

Now let's add the function that checks whether we've already got an access token. We'll also add a function that allows other libs to access `octokit`(GitHub) functions:

```
...
module.exports = {

  getInstance: () => {
    return octokit;
  },

  getStoredGithubToken : () => {
    return conf.get('github.token');
  },

  setGithubCredentials : async () => {
    ...
  },

  registerNewToken : async () => {
    ...
  }

}
```

If a `conf` object exists and it has `github.token` property, this means that there's

already a token in storage. In this case, we return the token value back to the invoking function. We'll get to that later on.

If no token is detected, we need to fetch one. Of course, getting an OAuth token involves a network request, which means a short wait for the user. This gives us an opportunity to look at the [clui](#) package which provides some enhancements for console-based applications, among them an animated spinner.

Creating a spinner is easy:

```
const status = new Spinner('Authenticating you, please wait...');  
status.start();
```

Once you're done, simply stop it and it will disappear from the screen:

```
status.stop();
```

Dynamic Captions

You can also set the caption dynamically using the `update` method. This could be useful if you have some indication of progress, for example displaying the percentage complete.

Here's the code to authenticate with GitHub:

```
...  
  setGithubCredentials : async () => {  
    const credentials = await inquirer.askGithubCredentials();  
    octokit.authenticate()  
      .extend(  
        {  
          type: 'basic',  
        },  
        credentials  
      )  
    );  
  },  
  
  registerNewToken : async () => {  
    const status = new Spinner('Authenticating you, please wait...')  
    status.start();  
  
    try {
```

```

const response = await octokit.authorization.create({
  scopes: ['user', 'public_repo', 'repo', 'repo:status'],
  note: 'ginit, the command-line tool for initializing Git rep
});
const token = response.data.token;
if(token) {
  conf.set('github.token', token);
  return token;
} else {
  throw new Error("Missing Token","GitHub token was not found
");
}
} catch (err) {
  throw err;
} finally {
  status.stop();
}
},

```

Let's step through this:

1. we prompt the user for their credentials using the `setGithubCredentials` method we defined earlier
2. we use [basic authentication](#) prior to trying to obtain an OAuth token
3. we attempt to [register a new access token for our application](#)
4. if we manage to get an access token, we set it in the configstore for next time.
5. we then return the token.

Any access tokens you create, whether manually or via the API as we're doing here, you'll be able to [see them here](#). During the course of development, you may find you need to delete ginit's access token — identifiable by the `note` parameter supplied above — so that you can re-generate it.

Working with 2FA

If you have two-factor authentication enabled on your GitHub account, the process is slightly more complicated. You'll need to request the confirmation code — for example, one sent via SMS — then supply it using the `X-GitHub-OTP` header. See [the documentation](#) for further information.

If you've been following along and would like to try out what we have so far, you can update the `run()` function in `index.js` as follows:

```
const run = async () => {
  let token = github.getStoredGithubToken();
  if(!token) {
    await github.setGithubCredentials();
    token = await github.registerNewToken();
  }
  console.log(token);
}
```

Please note you may get a `Promise` error if something goes wrong, such as inputting the wrong password. I'll show you the proper way of handling such errors later.

Creating a Repository

Once we've got an OAuth token, we can use it to create a remote repository with GitHub.

Again, we can use Inquirer to ask a series of questions. We need a name for the repo, we'll ask for an optional description, and we also need to know whether it should be public or private.

We'll use [minimist](#) to grab defaults for the name and description from optional command-line arguments. For example:

```
ginit my-repo "just a test repository"
```

This will set the default name to `my-repo` and the description to `just a test repository`.

The following line will place the arguments in an array indexed by an underscore:

```
const argv = require('minimist')(process.argv.slice(2));
// { _: [ 'my-repo', 'just a test repository' ] }
```

More to `minimist`

This only really scratches the surface of the `minimist` package. You can also use it to interpret flags, switches and name/value pairs. Check out the documentation for more information.

We'll write code to parse the command-line arguments and ask a series of questions. First, update `lib/inquirer.js` by inserting the following code right after `askGithubCredentials` function:

```
...
askRepoDetails: () => {
  const argv = require('minimist')(process.argv.slice(2));

  const questions = [
    {
      type: 'input',
```

```

        name: 'name',
        message: 'Enter a name for the repository:',
        default: argv._[0] || files.getCurrentDirectoryBase(),
        validate: function( value ) {
            if (value.length) {
                return true;
            } else {
                return 'Please enter a name for the repository.';
            }
        }
    },
{
    type: 'input',
    name: 'description',
    default: argv._[1] || null,
    message: 'Optionally enter a description of the repository:'
},
{
    type: 'list',
    name: 'visibility',
    message: 'Public or private:',
    choices: [ 'public', 'private' ],
    default: 'public'
}
];
return inquirer.prompt(questions);
},

```

Next, create the file `lib/repo.js` and add this code:

```

const _          = require('lodash');
const fs         = require('fs');
const git        = require('simple-git')();
const CLI        = require('clui')
const Spinner    = CLI.Spinner;

const inquirer   = require('./inquirer');
const gh          = require('./github');

module.exports = {
    createRemoteRepo: async () => {
        const github = gh.getInstance();
        const answers = await inquirer.askRepoDetails();

        const data = {
            name : answers.name,
            description : answers.description,

```

```
    private : (answers.visibility === 'private')
};

const status = new Spinner('Creating remote repository...');

status.start();

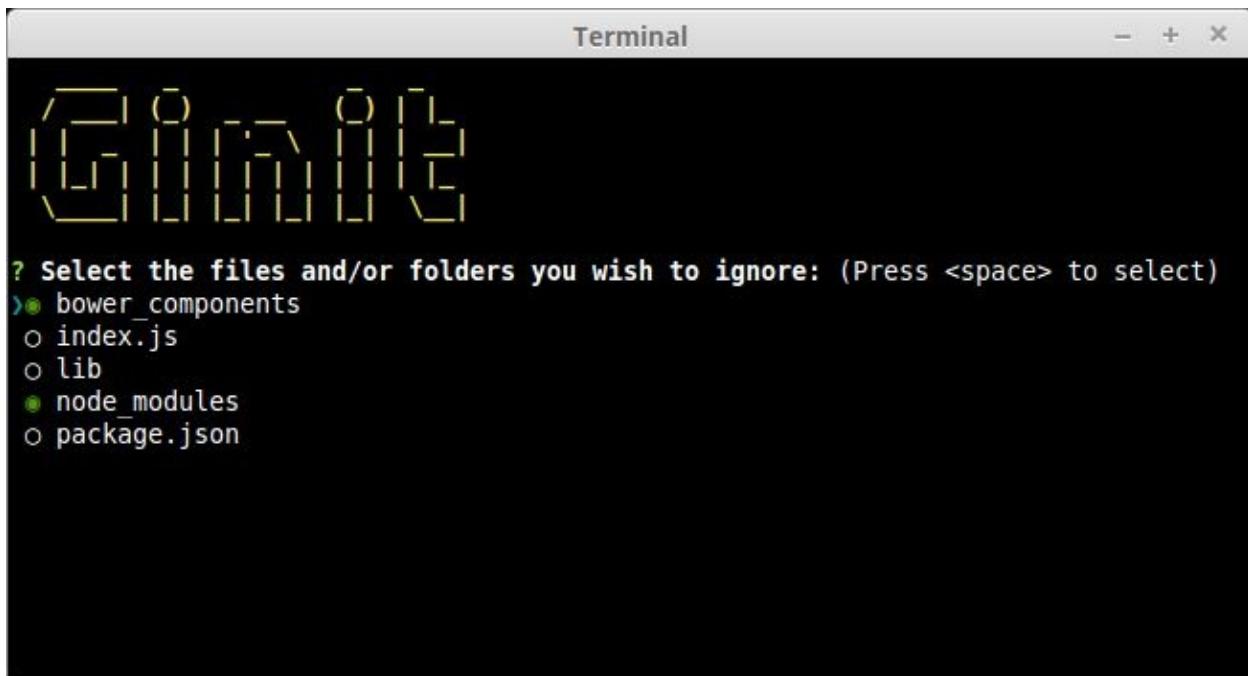
try {
  const response = await github.repos.create(data);
  return response.data.ssh_url;
} catch(err) {
  throw err;
} finally {
  status.stop();
}
},
}
```

Once we have that information, we can simply use the GitHub package to [create a repo](#), which will give us a URL for the newly created repository. We can then set that up as a remote in our local Git repository. First, however, let's interactively create a `.gitignore` file.

Creating a .gitignore File

For the next step, we'll create a simple command-line "wizard" to generate a `.gitignore` file. If the user is running our application in an existing project directory, let's show them a list of files and directories already in the current working directory, and allow them to select which ones to ignore.

The Inquirer package provides a checkbox input type for just that.



The first thing we need to do is scan the current directory, ignoring the `.git` folder and any existing `.gitignore` file (we do this by making use of lodash's [without](#) method):

```
const filelist = _.without(fs.readdirSync('.'), '.git', '.gitignore')
```

If there's nothing to add, there's no point in continuing, so let's simply touch the current `.gitignore` file and bail out of the function:

```
if (filelist.length) {
  ...
} else {
  touch('.gitignore');
}
```

Finally, let's utilize Inquirer's checkbox "widget" to list the files. Insert the following code in `lib/inquirer.js`:

```
...
askIgnoreFiles: (filelist) => {
  const questions = [
    {
      type: 'checkbox',
      name: 'ignore',
      message: 'Select the files and/or folders you wish to ignore',
      choices: filelist,
      default: ['node_modules', 'bower_components']
    }
  ];
  return inquirer.prompt(questions);
},
...
```

Notice that we can also provide a list of defaults. In this case, we're pre-selecting `node_modules` and `bower_components`, should they exist.

With the Inquirer code in place, we can now construct the `createGitignore()` function. Insert this code in `lib/repo.js`:

```
...
createGitignore: async () => {
  const filelist = _.without(fs.readdirSync('.'), '.git', '.gitignore');

  if (filelist.length) {
    const answers = await inquirer.askIgnoreFiles(filelist);
    if (answers.ignore.length) {
      fs.writeFileSync( '.gitignore', answers.ignore.join( '\n' ) )
    } else {
      touch( '.gitignore' );
    }
  } else {
    touch('.gitignore');
  }
},
```

Once "submitted", we then generate a `.gitignore` by joining up the selected list of files, separated with a newline. Our function now pretty much guarantees we've got a `.gitignore` file, so we can proceed with initializing a Git repository.

Interacting with Git from Within the App

There are a number of ways to interact with Git, but perhaps the simplest is to use the [simple-git](#) package. This provides a set of chainable methods which, behind the scenes, run the Git executable.

These are the repetitive tasks we'll use it to automate:

1. run `git init`
2. add the `.gitignore` file
3. add the remaining contents of the working directory
4. perform an initial commit
5. add the newly-created remote repository
6. push the working directory up to the remote.

Insert the following code in `lib/repo.js`:

```
...
setupRepo: async (url) => {
  const status = new Spinner('Initializing local repository and pu
  status.start();

  try {
    await git
      .init()
      .add('.gitignore')
      .add('./')
      .commit('Initial commit')
      .addRemote('origin', url)
      .push('origin', 'master');
    return true;
  } catch(err) {
    throw err;
  } finally {
    status.stop();
  }
},
...
```

Putting It All Together

First, let's set up a couple of helper functions in `lib/github.js`. We need a convenient function for accessing the stored token, and another function for setting up an oauth authentication:

```
...
githubAuth : (token) => {
  octokit.authenticate({
    type : 'oauth',
    token : token
  });
},
getStoredGithubToken : () => {
  return conf.get('github.token');
},
...
```

Next, we create a function in `index.js` for handling the logic of acquiring the token. Place this code before the `run()` function:

```
const getGithubToken = async () => {
  // Fetch token from config store
  let token = github.getStoredGithubToken();
  if(token) {
    return token;
  }

  // No token found, use credentials to access GitHub account
  await github.setGithubCredentials();

  // register new token
  token = await github.registerNewToken();
  return token;
}
```

Finally, we update the `run()` function by writing code that will handle the main logic of the app:

```
const run = async () => {
  try {
    // Retrieve & Set Authentication Token
  }
```

```
const token = await getGithubToken();
github.githubAuth(token);

// Create remote repository
const url = await repo.createRemoteRepo();

// Create .gitignore file
await repo.createGitignore();

// Set up local repository and push to remote
const done = await repo.setupRepo(url);
if(done) {
  console.log(chalk.green('All done!'));
}
} catch(err) {
  if (err) {
    switch (err.code) {
      case 401:
        console.log(chalk.red('Couldn\'t log you in. Please prov
        break;
      case 422:
        console.log(chalk.red('There already exists a remote rep
        break;
      default:
        console.log(err);
    }
  }
}
}
```

As you can see, we ensure the user is authenticated before calling all of our other functions (`createRemoteRepo()`, `createGitignore()`, `setupRepo()`) sequentially. The code also handles any errors and offers the user appropriate feedback.

You can check out the completed [index.js](#) file on our GitHub repo.

Making the ginit Command Available Globally

The one remaining thing to do is to make our command available globally. To do this, we'll need to add a [shebang](#) line to the top of `index.js`:

```
#!/usr/bin/env node
```

Next, we need to add a `bin` property to our `package.json` file. This maps the command name (`ginit`) to the name of the file to be executed (relative to `package.json`).

```
"bin": {  
  "ginit": "./index.js"  
}
```

After that, install the module globally and you'll have a working shell command:

```
npm install -g
```

Works on Windows

This will also work on Windows, as [npm will helpfully install a cmd wrapper alongside your script](#).

Taking it Further

We've got a fairly nifty, albeit simple command-line app for initializing Git repositories. But there's plenty more you could do to enhance it further.

If you're a Bitbucket user, you could adapt the program to use the Bitbucket API to create a repository. There's a [Node.js API wrapper available](#) to help you get started. You may wish to add an additional command-line option or prompt to ask the user whether they want to use GitHub or Bitbucket (Inquirer would be perfect for just that) or merely replace the GitHub-specific code with a Bitbucket alternative.

You could also provide the facility to specify your own set of defaults for the `.gitignore` file, instead of a hardcoded list. The preferences package might be suitable here, or you could provide a set of "templates" — perhaps prompting the user for the type of project. You might also want to look at integrating it with the [.gitignore.io](#) command-line tool/API.

Beyond all that, you may also want to add additional validation, provide the ability to skip certain sections, and more.

Chapter 6: Building a Real-time Chat App with Sails.js

by Michael Wanyoike

If you're a developer who currently uses frameworks such as Django, Laravel or Rails, you've probably heard about Node.js. You might already be using a popular front-end library such as Angular or React in your projects. By now, you should be thinking about doing a complete switchover to a server technology based on Node.js.

However, the big question is where to start. Today, the JavaScript world has grown at an incredibly fast pace in the last few years, and it seems to be ever expanding.

If you're afraid of losing your hard-earned programming experience in the Node universe, fear not, as we have [Sails.js](#).

Sails.js is a real-time MVC framework designed to help developers build production-ready, enterprise-grade Node.js apps in a short time. Sails.js is a pure JavaScript solution that supports multiple databases (simultaneously) and multiple front-end technologies. If you're a Rails developer, you'll be happy to learn that [Mike McNeil](#), the Sails.js founder, was inspired by Rails. You'll find a lot of similarities between Rails and Sails.js projects.

In this article, I'll teach you the fundamentals of Sails.js, by showing you how to build a simple, user-friendly chat application. The complete source code for the sails-chat project can be found in this [GitHub repo](#).

Sails Chat App

localhost:1337/chat

Chat Room Profile Logout

Members

-  John Wayne
-  Peter Quinn
-  Jane Eyre
-  Michael Wanyoike

Community Conversations

-  **Peter Quinn posted on** 2018-01-22T21:03:32.241Z
Hey Everyone! Welcome to the community!
-  **Jane Eyre posted on** 2018-01-22T21:03:32.244Z
How's it going?
-  **John Wayne posted on** 2018-01-22T21:03:32.247Z
Super excited!
-  **Michael Wanyoike posted on** 2018-01-22T21:05:09.703Z
What should we talk about?
-  **John Wayne posted on** 2018-01-22T21:07:18.229Z
Let's send more invites for people to join in first
-  **Peter Quinn posted on** 2018-01-22T21:08:27.492Z
Great idea John!
-  **Jane Eyre posted on** 2018-01-22T21:09:15.176Z
Has anyone noticed that we are all just one guy chatting to himself?

Post Message

Post

Prerequisites

Before you start, you need at least to have experience developing applications using MVC architecture. This tutorial is intended for intermediate developers. You'll also need at least to have a basic foundation in these:

- [Node.js](#)
- [Modern JavaScript syntax \(ES6+\)](#).

To make it practical and fair for everyone, this tutorial will use core libraries that are installed by default in a new Sails.js project. Integration with modern front-end libraries such as React, Vue or Angular won't be covered here. However, I highly recommend you look into them after this article. Also, we won't do database integrations. We'll instead use the default, local-disk, file-based database for development and testing.

Project Plan

The goal of this tutorial is to show you how to build a chat application similar to [Slack](#), [Gitter](#) or [Discord](#).

Not really! A lot of time and sweat went into building those wonderful platforms. The current number of features developed into them is quite huge.

Instead, we'll build a minimum viable product version of a chat application which consists of:

- single chat room
- basic authentication (passwordless)
- profile update.

I've added the *profile feature* as a bonus in order to cover a bit more ground on Sails.js features.

Installing Sails.js

Before we start installing Sails.js, we need first to set up a proper Node.js environment. At the time of writing, the latest stable version currently available is v0.12.14. Sails.js v1.0.0 is also available but is currently in beta, not recommended for production use.

The latest stable version of Node I have access to is v8.9.4. Unfortunately, Sails.js v0.12 doesn't work properly with the current latest LTS. However, I've tested with Node v.7.10 and found everything works smoothly. This is still good since we can use some new ES8 syntax in our code.

As a JavaScript developer, you'll realize working with one version of Node.js is not enough. Hence, I recommend using the [nvm tool](#) to manage multiple versions of Node.js and NPM easily. If you haven't done so, just purge your existing Node.js installation, then install nvm to help you manage multiple versions of Node.js.

Here are the basic instructions of installing Node v7 and Sails.js:

```
# Install the latest version of Node v7 LTS
nvm install v7

# Make Node v7 the default
nvm default alias v7

# Install Sails.js Global
npm install -g sails
```

If you have a good internet connection, this should only take a couple of minutes or less. Let's now go ahead and create our new application using the Sails generator command:

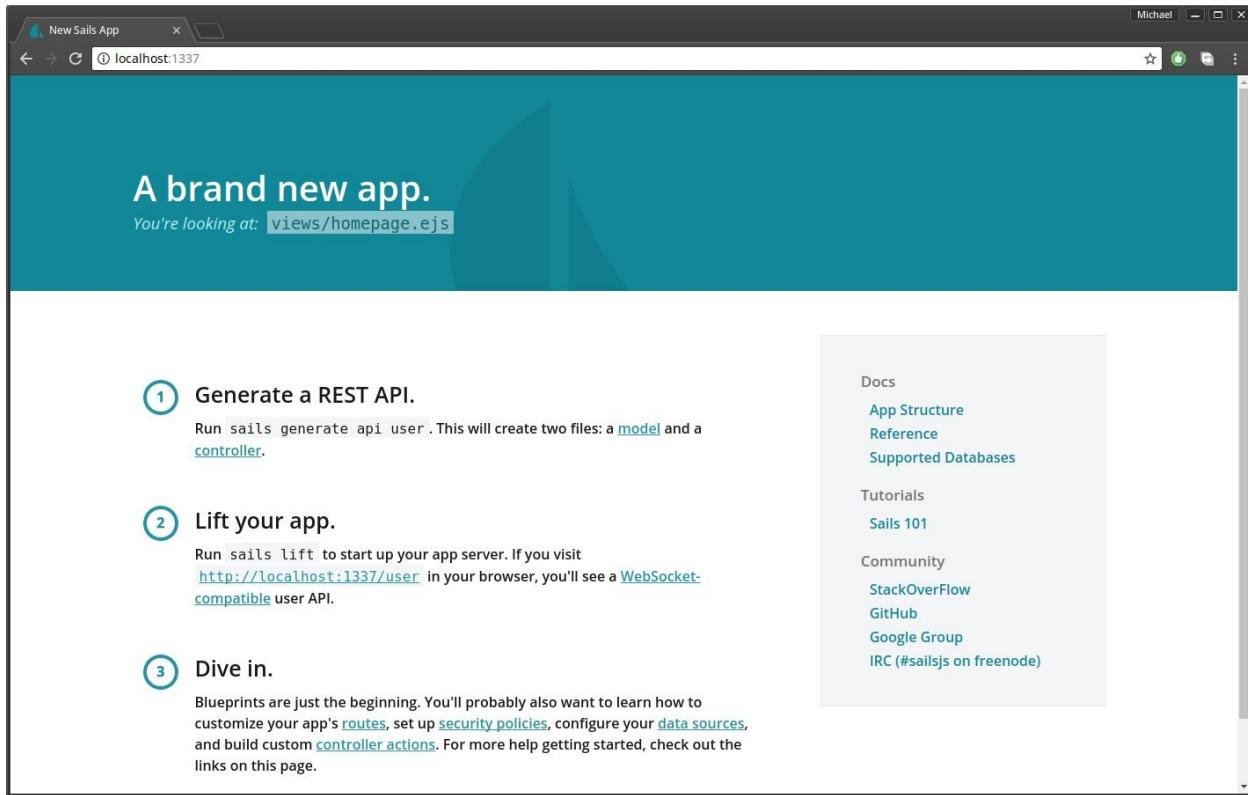
```
# Go to your projects folder
cd Projects

# Generate your new app
sails generate new chat-app

# Wait for the install to finish then navigate to the project folder
cd chat-app
```

```
# Start the app  
sails lift
```

It should take a few seconds for the app to start. You need to manually open the url <http://localhost:1337> in your browser to see your newly created web app.



Seeing this confirms that we have a running project with no errors, and that we can start working. To stop the project, just press `control + c` at the terminal. User your favorite code editor (I'm using Atom) to examine the generated project structure. Below are the main folders you should be aware of:

- `api`: controllers, models, services and policies (permissions)
- `assets`: images, fonts, JS, CSS, Less, Sass etc.
- `config`: project configuration e.g. database, routes, credentials, locales, security etc.
- `node_modules`: installed npm packages
- `tasks`: Grunt config scripts and pipeline script for compiling and injecting assets
- `views`: view pages — for example, EJS, Jade or whatever templating engine you prefer
- `.tmp`: temporary folder used by Sails to build and serve your project while in development mode.

Before we proceed, there are a couple of things we need to do:

- **Update EJS package.** If you have EJS 2.3.4 listed in `package.json`, you

need to update it by changing it to 2.5.5 immediately. It contains a serious security vulnerability. After changing the version number, do an npm install to perform the update.

- **Hot reloading.** I suggest you install [sails-hook-autoreload](#) to enable hot reloading for your Sails.js app. It's not a perfect solution but will make development easier. To install it for this current version of Sails.js, execute the following:

```
npm install sails-hook-autoreload@for-sails-0.12 --save
```

Installing Front-end Dependencies

For this tutorial, we'll spend as little time as possible building an UI. Any CSS framework you're comfortable with will do. For this tutorial, I'll go with the [Semantic UI](#) CSS library.

Sails.js doesn't have a specific guide on how to install CSS libraries. There are three or more ways you can go about it. Let's look at each.

1. Manual Download

You can download the CSS files and JS scripts yourself, along with their dependencies. After downloading, place the files inside the assets folder.

I prefer not to use this method, as it requires manual effort to keep the files updated. I like automating tasks.

2. Using Bower

This method requires you to create a file called `.bowerrc` at the root of your project. Paste the following snippet:

```
{  
  "directory" : "assets/vendor"  
}
```

This will instruct Bower to install to the `assets/vendor` folder instead of the default `bower_components` folder. Next, install Bower globally, and your front-end dependencies locally using Bower:

```
# Install bower globally via npm-  
npm install -g bower  
  
# Create bower.json file, accept default answers (except choose y for  
bower init  
  
# Install semantic-ui via bower  
bower install semantic-ui --save  
  
# Install jsrender
```

```
bower install jsrender --save
```

I'll explain the purpose of `jsrender` later. I thought it best to finish the task of installing dependencies in one go. You should take note that `jQuery` has been installed as well, since it's a dependency for `semantic-ui`.

After installing, update `assets/style/importer.less` to include this line:

```
@import '../vendor/semantic/dist/semantic.css';
```

Next include the JavaScript dependencies in `tasks/pipeline.js`:

```
var jsFilesToInject = [  
  
  // Load Sails.io before everything else  
  'js/dependencies/sails.io.js',  
  
  // Vendor dependencies  
  'vendor/jquery/dist/jquery.js',  
  'vendor/semantic/dist/semantic.js',  
  'vendor/jsrender/jsrender.js',  
  
  // Dependencies like jQuery or Angular are brought in here  
  'js/dependencies/**/*.js',  
  
  // All of the rest of your client-side JS files  
  // will be injected here in no particular order.  
  'js/**/*.js'  
];
```

When we run `sails lift`, the JavaScript files will automatically be injected into `views/layout.ejs` file as per `pipeline.js` instructions. The current `grunt` setup will take care of injecting our CSS dependencies for us.

Don't Add Vendor Dependencies

Add the word `vendor` in the `.gitignore` file. We don't want vendor dependencies saved in our repository.

3. Using npm + grunt.copy

The third method requires a little bit more effort to set up, but will result in a

lower footprint. Install the dependencies using npm as follows:

```
npm install semantic-ui-css jsrender --save
```

jQuery will be installed automatically, since it's also listed as a dependency for semantic-ui-css. Next we need to place code in tasks/config/copy.js. This code will instruct Grunt to copy the required JS and CSS files from node_modules to the assets/vendor folder for us. The entire file should look like this:

```
module.exports = function(grunt) {

  grunt.config.set('copy', {
    dev: {
      files: [{  
        expand: true,  
        cwd: './assets',  
        src: ['**/*.!(*coffee|less)*'],  
        dest: '.tmp/public'  
      },  
      //Copy JQuery  
      {  
        expand: true,  
        cwd: './node_modules/jquery/dist/',  
        src: ['jquery.min.js'],  
        dest: './assets/vendor/jquery'  
      },  
      //Copy jsrender  
      {  
        expand: true,  
        cwd: './node_modules/jsrender/',  
        src: ['jsrender.js'],  
        dest: './assets/vendor/jsrender'  
      },  
      // copy semantic-ui CSS and JS files  
      {  
        expand: true,  
        cwd: './node_modules/semantic-ui-css/',  
        src: ['semantic.css', 'semantic.js'],  
        dest: './assets/vendor/semantic-ui'  
      },  
      //copy semantic-ui icon fonts  
      {  
        expand: true,  
        cwd: './node_modules/semantic-ui-css/themes',  
        dest: './assets/vendor/semantic-ui/icon/  
      }  
    }
  })
}
```

```
        src: ["*.*", "**/*.*"],
        dest: './assets/vendor/semantic-ui/themes'
    }]
},
build: {
    files: [{
        expand: true,
        cwd: '.tmp/public',
        src: ['**/*'],
        dest: 'www'
    }]
}
});

grunt.loadNpmTasks('grunt-contrib-copy');
};
```

Add this line to assets/styles/importer.less:

```
@import '../vendor/semantic-ui/semantic.css';
```

Add the JS files to config/pipeline.js:

```
// Vendor Dependencies
'vendor/jquery/jquery.min.js',
'vendor/semantic-ui/semantic.js',
'vendor/jsrender/jsrender.js',
```

Finally, execute this command to copy the files from node_modules the assets/vendor folder. You only need to do this once for every clean install of your project:

```
grunt copy:dev
```

Remember to add vendor to your .gitignore.

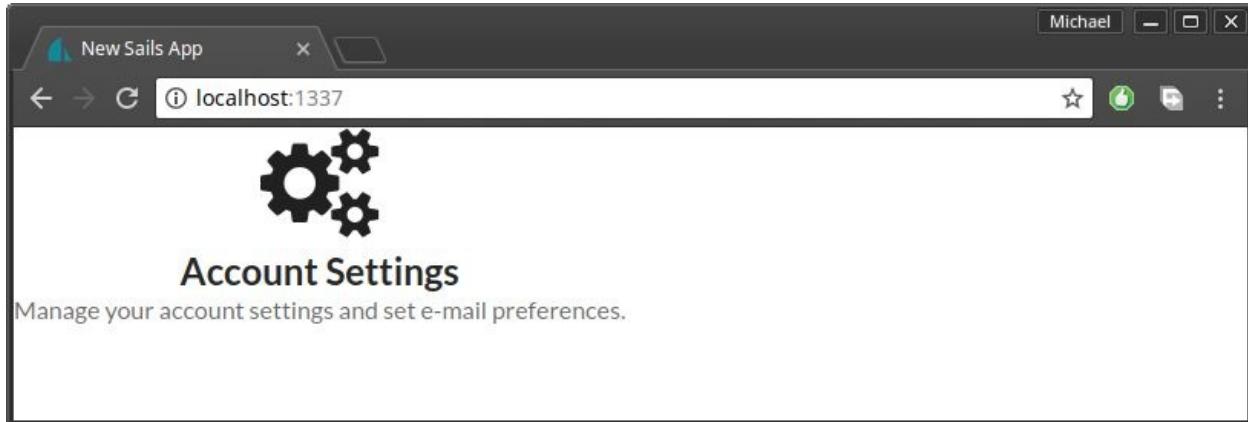
Testing Dependencies Installation

Whichever method you've chosen, you need to ensure that the required dependencies are being loaded. To do this, replace the code in view/homepage.ejs with the following:

```
<h2 class="ui icon header">
<i class="settings icon"></i>
```

```
<div class="content">
  Account Settings
    <div class="sub header">Manage your account settings and set e-mail preferences</div>
</h2>
```

After saving the file, do a `sails lift`. Your home page should now look like this:



Always do a refresh after restarting your app. If the icon is missing or the font looks off, please review the steps carefully and see what you missed. Use the browser's console to see which files are not loading. Otherwise, proceed with the next stage.

Creating Views

When it comes to project development, I like starting with the user interface. We'll use the [Embedded JavaScript Template](#) to create the views. It's a templating engine that's installed by default in every Sails.js project. However, you should be aware it has limited functionality and is no longer under development.

Open config/bootstrap.js and insert this line in order to give a proper title to our web pages. Place it right within the existing function before the cb() statement:

```
sails.config.appName = "Sails Chat App";
```

You can take a peek at views/layout.ejs to see how the title tag is set. Next, we begin to build our home page UI.

Home Page Design

Open /views/homepage.ejs and replace the existing code with this:

```
<div class="banner">
<div class="ui segment teal inverted">
  <h1 class="ui center aligned icon header">
    <i class="chat icon"></i>
    <div class="content">
      <a href="/">Sails Chat</a>
      <div class="sub header">Discuss your favorite technology with
        </div>
    </h1>
  </div>
</div>
<div class="section">
<div class="ui three column grid">
  <div class="column"></div>
  <div class="column">
    <div class="ui centered padded compact raised segment">
      <h3>Sign Up or Sign In</h3>
      <div class="ui divider"></div>
      [TODO : Login Form goes here]
    </div>
  </div>
</div>
```

```
</div>
<div class="column"></div>
</div>
</div>
```

To understand the UI elements used in the above code, please refer to the Semantic UI documentation. I've outlined the exact links below:

- [Segment](#)
- [Icon](#)
- [Header](#)
- [Grid](#)

Create a new file in assets/styles/theme.less and paste the following content:

```
.banner a {
color: #fff;
}

.centered {
margin-left: auto !important;
margin-right: auto !important;
margin-bottom: 30px !important;
}

.section {
margin-top: 30px;
}

.menu {
border-radius: 0 !important;
}

.note {
font-size: 11px;
color: #2185D0;
}

#chat-content {
height: 90%;
overflow-y: scroll;
}
```

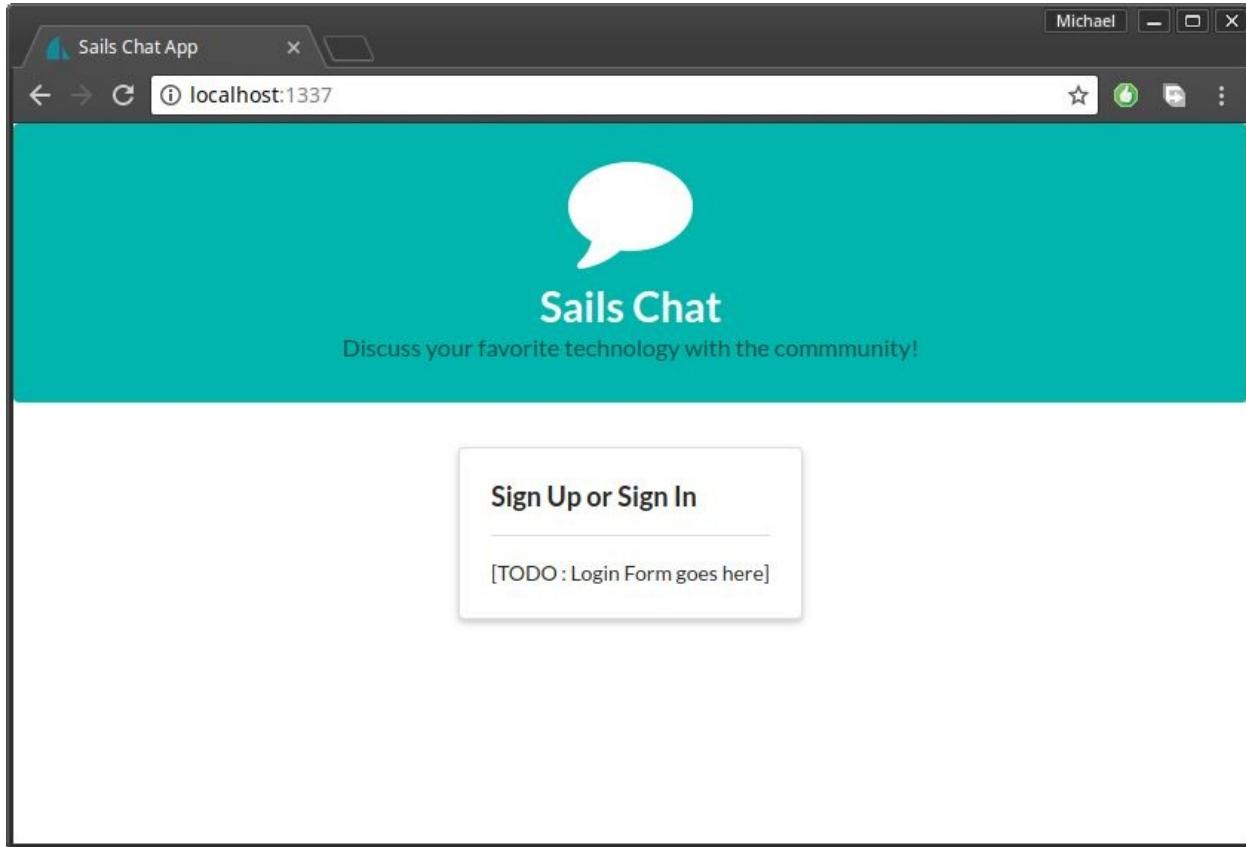
These are all the custom styles we'll use in our project. The rest of the styling

will come from the Semantic UI library.

Next, update assets/styles/importer.less to include the theme file we just created:

```
@import 'theme.less';
```

Execute sails lift. Your project should now look like this:



Next, we'll look at building the navigation menu.

Navigation Menu

This will be created as a partial since it will be shared by multiple view files. Inside the views folder, create a folder called partials. Then create the file views/partials/menu.ejs and paste the following code:

```
<div class="ui labeled icon inverted teal menu">
<a class="item" href="/chat">
  <i class="chat icon"></i>
```

```
Chat Room
</a>
<a class="item" href="/profile">
  <i class="user icon"></i>
  Profile
</a>
<div class="right menu">
  <a class="item" href="/auth/logout">
    <i class="sign out icon"></i>
    Logout
  </a>
</div>
</div>
```

To understand the above code, just refer to the [Menu](#) documentation.

If you inspect the above code, you'll notice that we've created a link for /chat, /profile and /auth/logout. Let's first create the views for profile and chat room.

Profile

Create the file `view/profile.ejs` and paste the following code:

```
<% include partials/menu %>

<div class="ui container">
<h1 class="ui centered header">Profile Updated!</h1>
<hr>
<div class="section">
  [ TODO put user-form here]
</div>
</div>
```

By now you should be familiar with header and grid UI elements if you've read the linked documentation. At the root of the document, you'll notice we have a container element. (Find out more about this in the [Container](#) documentation.

We'll build the user form later, once we've built the API. Next we'll create a layout for the chat room.

Chat Room Layout

The chat room will be made up of three sections:

- **Chat users** — list of users
- **Chat messages** — list of messages
- **Chat post** — form for posting new messages.

Create `views/chatroom.ejs` and paste the following code:

```
<% include partials/menu %>

<div class="chat-section">
<div class="ui container grid">

    <!-- Members List Section -->
    <div class="four wide column">
        [ TODO chat-users ]
    </div>

    <div class="twelve wide column">

        <!-- Chat Messages -->
        [ TODO chat-messages ]

        <hr>

        <!-- Chat Post -->
        [ TODO chat-post ]

    </div>
</div>
</div>
```

Before we can view the pages, we need to set up routing.

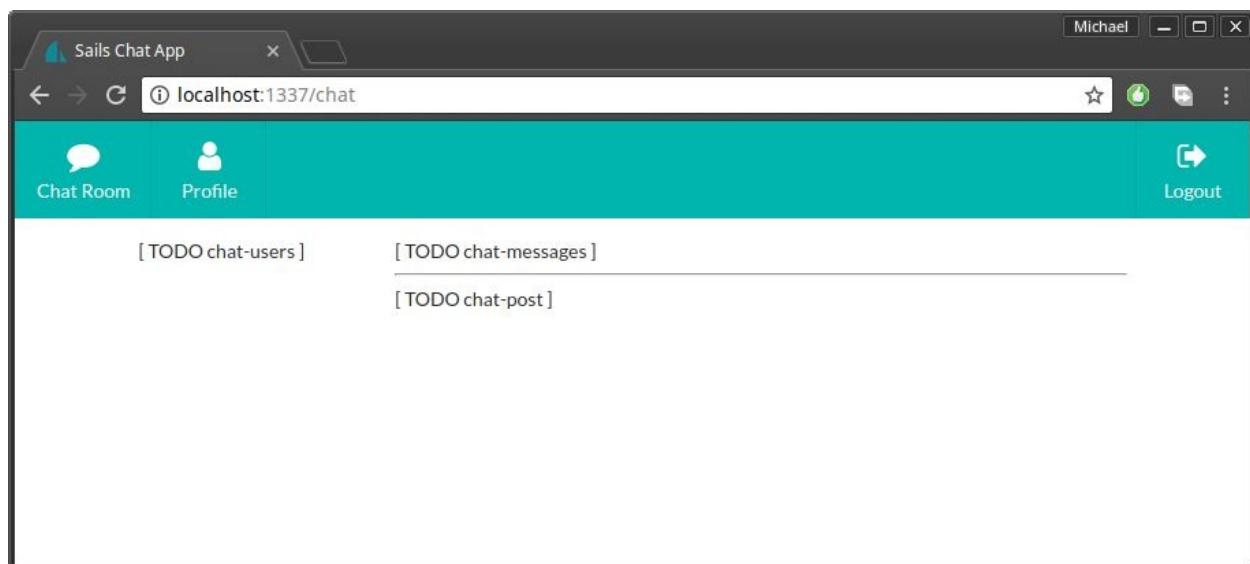
Routing

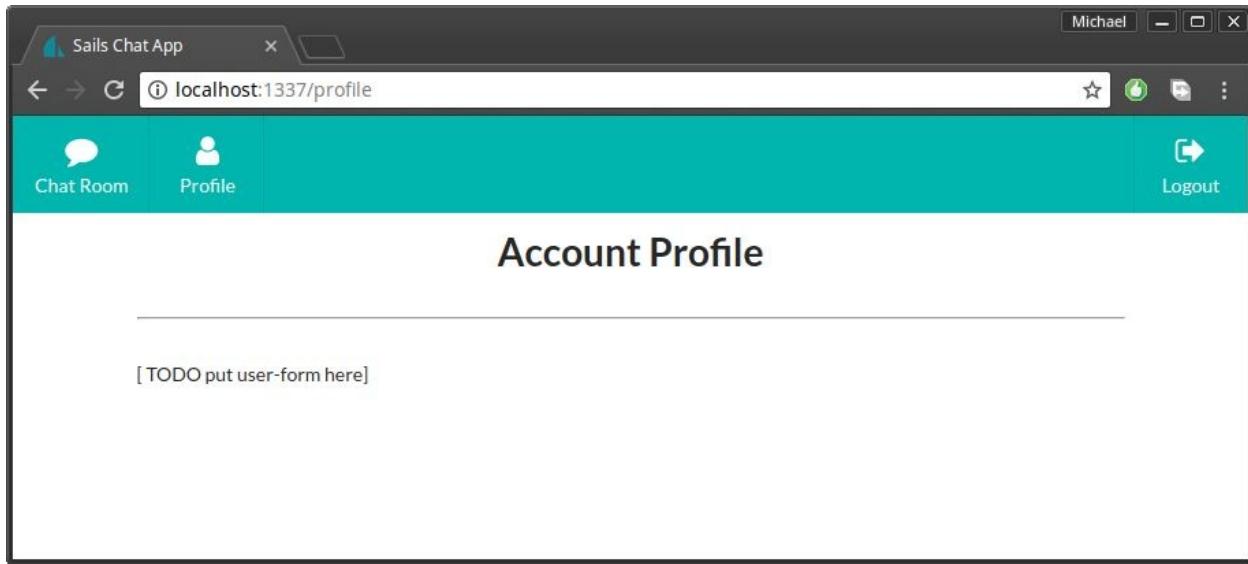
Open config/routes.js and update it like this:

```
'/' : {  
  view: 'homepage'  
},  
'/profile' : {  
  view: 'profile'  
},  
'/chat' : {  
  view: 'chatroom'  
}
```

Sails.js routing is quite flexible. There are many ways of defining routing depending on the scenario. This is the most basic version where we map a URL to a view.

Fire up your Sails app or just refresh your page if it's still running in the background. Currently there's no link between the home page and the other pages. This is intentional, as we'll later build a rudimentary authentication system that will redirect logged in users to /chat. For now, use your browser's address bar and add /chat or /profile at the end URL.





At this stage you should be having the above views. Let's go ahead and start creating the API.

Generating a User API

We're going to use the Sails.js command-line utility to generate our API. We'll need to stop the app for this step:

```
sails generate api User
```

Within a second, we get the message “Created a new api!” Basically, a `User.js` model and a `UserController.js` has just been created for us. Let’s update the `api/model/User.js` with some model attributes:

```
module.exports = {

  attributes: {

    name: {
      type: 'string',
      required: true
    },

    email: {
      type: 'string',
      required: true,
      unique: true
    },

    avatar: {
      type: 'string',
      required: true,
      defaultsTo: 'https://s.gravatar.com/avatar/e28f6f64608c970c66319'
    },

    location: {
      type: 'string',
      required: false,
      defaultsTo: ''
    },

    bio: {
      type: 'string',
      required: false,
      defaultsTo: ''
    }
}
```

```
};  
};
```

I believe the above code is self-explanatory. By default, Sails.js uses a local disk database which is basically a file located in the `.tmp` folder. In order to test our app, we need to create some users. The easiest way to do this is to install the [sails-seed package](#):

```
npm install sails-seed --save
```

After installing, you'll find that the file `config/seeds.js` has been created for you. Paste the following seed data:

```
module.exports.seeds = {  
  user: [  
    {  
      name: 'John Wayne',  
      email: 'johnnie86@gmail.com',  
      avatar: 'https://randomuser.me/api/portraits/men/83.jpg',  
      location: 'Mombasa',  
      bio: 'Spends most of my time at the beach'  
    },  
    {  
      name: 'Peter Quinn',  
      email: 'peter.quinn@live.com',  
      avatar: 'https://randomuser.me/api/portraits/men/32.jpg',  
      location: 'Langley',  
      bio: 'Rather not say'  
    },  
    {  
      name: 'Jane Eyre',  
      email: 'jane@hotmail.com',  
      avatar: 'https://randomuser.me/api/portraits/women/94.jpg',  
      location: 'London',  
      bio: 'Loves reading motivation books'  
    }  
  ]  
}
```

Now that we've generated an API, we should configure the migration policy in the file `config/models.js`:

```
migrate: 'drop'
```

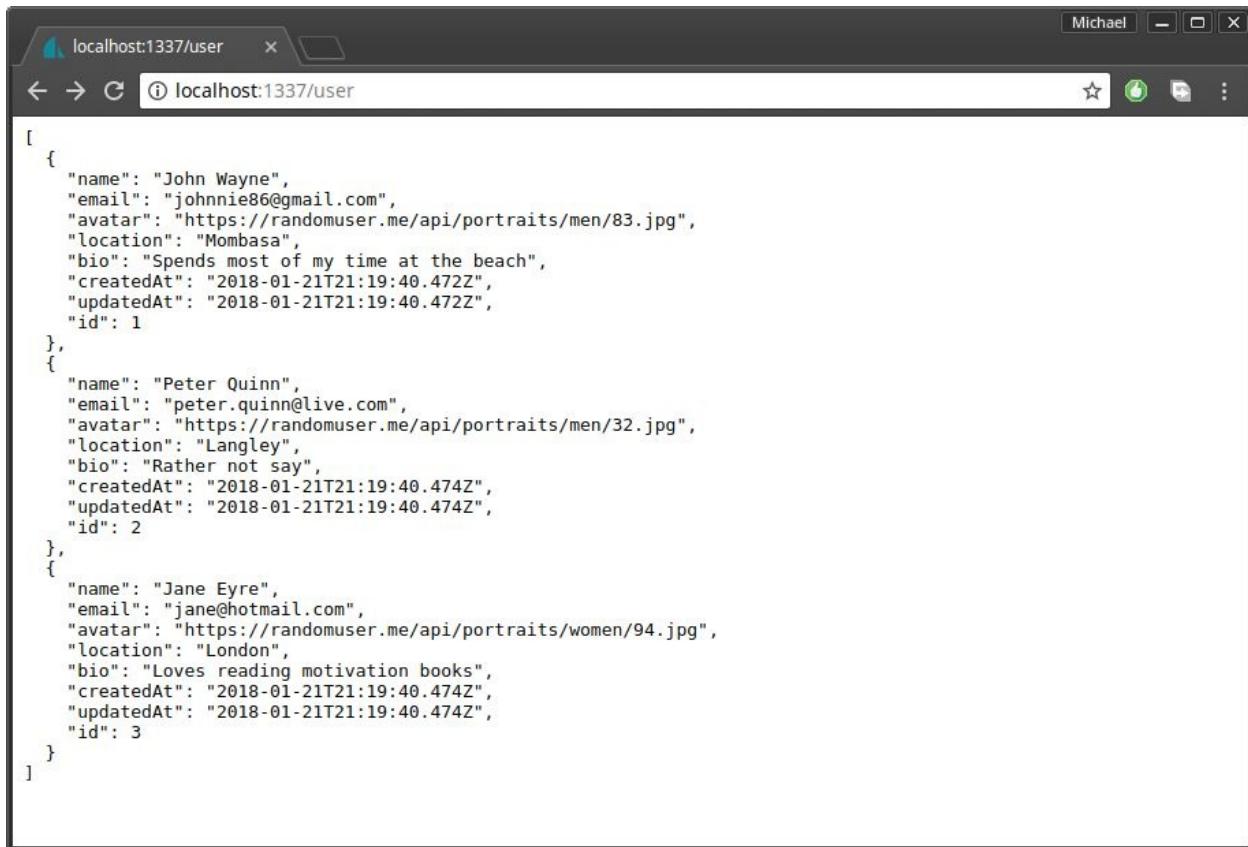
There are three migration strategies that Sails.js uses to determine how to rebuild

your database every time it's started:

- **safe** — don't migrate, I'll do it by hand
- **alter** — migrate but try to keep the existing data
- **drop** — drop all tables and rebuild everything

I prefer to use `drop` for development, as I tend to iterate a lot. You can set `alter` if you'd like to keep existing data. Nevertheless, our database will be populated by the seed data every time.

Now let me show you something cool. Fire up your Sails project and navigate to the addresses `/user` and `/user/1`.



A screenshot of a web browser window titled "localhost:1337/user". The address bar shows "localhost:1337/user". The content area displays a JSON array of three user objects. Each object has properties: name, email, avatar, location, bio, createdAt, updatedAt, and id. The users are named John Wayne, Peter Quinn, and Jane Eyre, with their respective details.

```
[{"name": "John Wayne", "email": "johnnie86@gmail.com", "avatar": "https://randomuser.me/api/portraits/men/83.jpg", "location": "Mombasa", "bio": "Spends most of my time at the beach", "createdAt": "2018-01-21T21:19:40.472Z", "updatedAt": "2018-01-21T21:19:40.472Z", "id": 1}, {"name": "Peter Quinn", "email": "peter.quinn@live.com", "avatar": "https://randomuser.me/api/portraits/men/32.jpg", "location": "Langley", "bio": "Rather not say", "createdAt": "2018-01-21T21:19:40.474Z", "updatedAt": "2018-01-21T21:19:40.474Z", "id": 2}, {"name": "Jane Eyre", "email": "jane@hotmail.com", "avatar": "https://randomuser.me/api/portraits/women/94.jpg", "location": "London", "bio": "Loves reading motivation books", "createdAt": "2018-01-21T21:19:40.474Z", "updatedAt": "2018-01-21T21:19:40.474Z", "id": 3}]
```



```
{  
  "name": "John Wayne",  
  "email": "johnnie86@gmail.com",  
  "avatar": "https://randomuser.me/api/portraits/men/83.jpg",  
  "location": "Mombasa",  
  "bio": "Spends most of my time at the beach",  
  "createdAt": "2018-01-21T21:19:40.472Z",  
  "updatedAt": "2018-01-21T21:19:40.472Z",  
  "id": 1  
}
```

Thanks to the Sails.js [Blueprints API](#), we have a fully functional CRUD API without us writing a single line of code. You can use Postman to access the User API and perform data manipulation such as creating, updating or deleting users.

Let's now proceed with building the profile form.

Profile Form

Open view/profile.ejs and replace the existing TODO line with this code:

```
  
<div class="ui grid">  
  <form action="<%= '/user/update/' + data.id %>" method="post" class="ui form">  
    <div class="field">  
      <label>Name</label>  
      <input type="text" name="name" value="<%= data.name %>">  
    </div>  
    <div class="field">  
      <label>Email</label>  
      <input type="text" name="email" value="<%= data.email %>">  
    </div>  
    <div class="field">  
      <label>Location</label>  
      <input type="text" name="location" value="<%= data.location %>">  
    </div>  
    <div class="field">  
      <label>Bio</label>  
      <textarea name="bio" rows="4" cols="40"><%= data.bio %></textarea>  
    </div>  
    <input type="hidden" name="avatar" value="<%= data.avatar %>">  
    <button class="ui right floated orange button" type="submit">Update Profile</button>  
  </form>  
</div>
```

We're using [Semantic-UI Form](#) to build the form interface. If you examine the form's action value, /user/update/' + data.id, you'll realize that I'm using a Blueprint route. This means when a user hits the Update button, the Blueprint's update action will be executed.

However, for loading the user data, I've decided to define a custom action in the User Controller. Update the api/controllers/UserController with the following code:

```
module.exports = {

  render: async (request, response) => {
    try {
      let data = await User.findOne({
        email: 'johnnie86@gmail.com'
      });
      if (!data) {
        return response.notFound('The user was NOT found!');
      }
      response.view('profile', { data });
    } catch (err) {
      response.serverError(err);
    }
  }
};
```

In this code you'll notice I'm using the `async/await` syntax to fetch the User data from the database. The alternative is to use callbacks, which for most developers is not clearly readable. I've also hardcoded the default user account to load temporarily. Later, when we set up basic authentication, we'll change it to load the currently logged-in user.

Finally, we need to change the route `/profile` to start using the newly created `UserController`. Open `config/routes` and update the `profile` route as follows:

```
...
'/profile': {
  controller: 'UserController',
  action: 'render'
},
...
```

Navigate to the URL `/profile`, and you should have the following view:

Sails Chat App

localhost:1337/profile

Michael

Chat Room Profile Logout

Account Profile



Name

Email

Location

Bio

Update

Try changing one of the form fields and hit the update button. You'll be taken to this view:



A screenshot of a web browser window titled "localhost:1337/user/up". The address bar shows "localhost:1337/user/update/1". The content area displays the following JSON object:

```
{  
  "name": "John Wayne",  
  "email": "johnnie86@gmail.com",  
  "avatar": "https://randomuser.me/api/portraits/men/83.jpg",  
  "location": "Naivasha",  
  "bio": "Spends most of my time at the beach",  
  "createdAt": "2018-01-21T21:19:40.472Z",  
  "updatedAt": "2018-01-21T22:06:34.563Z",  
  "id": 1  
}
```

You'll notice that the update has worked, but the data being displayed is in JSON format. Ideally, we should have a view-only profile page in `views/user/findOne.ejs` and an update profile page in `/views/user/update.ejs`. The Blueprint system will guess the views to use for rendering information. If it can't find the views it will just output JSON. For now, we'll simply use this neat trick. Create the file `/views/user/update.ejs` and paste the following code:

```
<script type="text/javascript">  
window.location = '/profile';  
</script>
```

Next time we perform an update, we'll be redirected to the `/profile` page. Now that we have user data, we can create the file `views/partials/chat-users.js` to be used in `views/chatroom.ejs`. After you've created the file, paste this code:

```
<div class="ui basic segment">  
<h3>Members</h3>  
<hr>  
<div id="users-content" class="ui middle aligned selection list"> </div>  
  
// jsrender template  
<script id="usersTemplate" type="text/x-jsrender">  
<div class="item">  
    
    <div class="header">{{:name}}</div>  
  </div>
```

```

</div>
</script>

<script type="text/javascript">

function loadUsers() {
    // Load existing users
    io.socket.get('/user', function(users, response) {
        renderChatUsers(users);
    });

    // Listen for new & updated users
    io.socket.on('user', function(body) {
        io.socket.get('/user', function(users, response) {
            renderChatUsers(users);
        });
    });
}

function renderChatUsers(data) {
    const template = $.templates('#usersTemplate');
    let htmlOutput = template.render(data);
    $('#users-content').html(htmlOutput);
}

</script>

```

For this view, we need a client-side rendering approach to make the page update in real time. Here, we're making use of the [jsrender](#) library, a more powerful templating engine than EJS. The beauty of jsrender is that it can either take an array or a single object literal and the template will still render correctly. If we were to do this in ejs, we'd need to combine an `if` statement and a `for` loop to handle both cases.

Let me explain the flow of our client-side JavaScript code:

1. `loadUsers()`. When the page first loads, we use the Sails.js socket library to perform a `GET` request for users. This request will be handled by the Blueprint API. We then pass on the data received to `renderChatUsers(data)` function.
2. Still within the `loadUsers()` function, we register a listener using `io.socket.on` function. We listen for events pertaining to the model user. When we get notified, we fetch the users again and replace the existing HTML output.

3. renderChatUsers(data). Here we grab a script with the id usersTemplate using a jQuery templates() function. Notice the type is text/x-jsrender. By specifying a custom type, the browser will ignore and skip over that section since it doesn't know what it is. We then use the template.render() function to merge the template with data. This process will generate an HTML output which we then take and insert it into the HTML document.

The template we wrote in `profile.ejs` was rendered on the Node server, then sent to the browser as HTML. For the case of `chat-users`, we need to perform client-side rendering. This will allow chat users to see new users joining the group without them refreshing their browser.

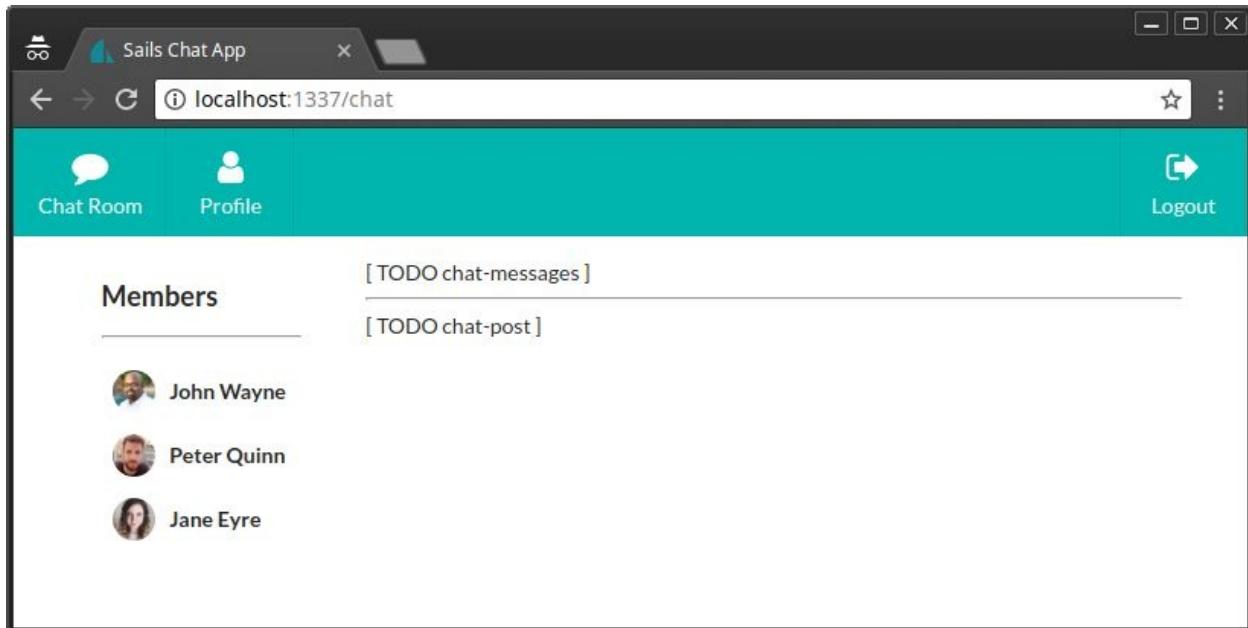
Before we test the code, we need to update `views/chatroom.ejs` to include the newly created `chat-users` partial. Replace [TODO `chat-users`] with this code:

```
...html
<% include partials/chat-users.ejs %>
...
```

Within the same file, add this script at the end:

```
<script type="text/javascript">
window.onload = function() {
  loadUsers();
}
</script>
```

This script will call the `loadUsers()` function. To confirm this is working, let's perform a `sails lift` and navigate to the `/chat` URL.



Your view should like like the image above. If it does, let's proceed with building the Chatroom API.

ChatMessage API

Same as before, we'll use Sails.js to generate the API:

```
sails generate api ChatMessage
```

Next, populate `api/models/ChatMessage.js` with these attributes:

```
module.exports = {  
  
  attributes: {  
  
    message: {  
      type: 'string',  
      required: true  
    },  
  
    createdBy : {  
      model: 'user',  
      required: true  
    }  
  }  
};
```

Notice that we've declared a one-to-one association with the `User` model through the `createdBy` attribute. Next we need to populate our disk database with a few chat messages. For that, we'll use `config/bootstrap.js`. Update the entire code as follows. We're using `async/await` syntax to simplify our code and avoid callback hell:

```
module.exports.bootstrap = async function(cb) {  
  
  sails.config.appName = "Sails Chat App";  
  
  // Generate Chat Messages  
  try {  
    let messageCount = ChatMessage.count();  
    if(messageCount > 0){  
      return; // don't repeat messages  
    }  
  
    let users = await User.find();  
    if(users.length >= 3) {  
      // Create 3 messages  
      for(let i = 0; i < 3; i++) {  
        let user = users[i];  
        let message = {  
          message:  
        };  
        ChatMessage.create(message).then(function(newMessage){  
          console.log(`Created message ${newMessage.id}`);  
        }).catch(function(error){  
          console.error(`Error creating message: ${error.message}`);  
        });  
      }  
    }  
  } catch (err) {  
    console.error(`Error bootstrapping: ${err.message}`);  
  }  
}  
};
```

```

        console.log("Generating messages...")

        let msg1 = await ChatMessage.create({
            message: 'Hey Everyone! Welcome to the community!',
            createdBy: users[1]
        });
        console.log("Created Chat Message: " + msg1.id);

        let msg2 = await ChatMessage.create({
            message: "How's it going?",
            createdBy: users[2]
        });
        console.log("Created Chat Message: " + msg2.id);

        let msg3 = await ChatMessage.create({
            message: 'Super excited!',
            createdBy: users[0]
        });
        console.log("Created Chat Message: " + msg3.id);

    } else {
        console.log('skipping message generation');
    }
} catch(err){
    console.error(err);
}

// It's very important to trigger this callback method when you're f
cb();
};

```

The great thing is that the seeds generator runs before bootstrap.js. This way, we're sure Users data has been created first so that we can use it to populate the createdBy field. Having test data will enable us to quickly iterate as we build the user interface.

Chat Messages UI

Go ahead and create a new file `views/partials/chat-messages.ejs`, then place this code:

```

<div class="ui basic segment" style="height: 70vh;">
<h3>Community Conversations</h3>
<hr>
<div id="chat-content" class="ui feed"> </div>

```

```

</div>

<script id="chatTemplate" type="text/x-jsrender">
<div class="event">
  <div class="label">
    
  </div>
  <div class="content">
    <div class="summary">
      <a href="#"> {{:createdBy.name}}</a> posted on
      <div class="date">
        {{:createdAt}}
      </div>
    </div>
    <div class="extra text">
      {{:message}}
    </div>
  </div>
</div>
</script>

<script type="text/javascript">

function loadMessages() {
  // Load existing chat messages
  io.socket.get('/chatMessage', function(messages, response) {
    renderChatMessages(messages);
  });

  // Listen for new chat messages
  io.socket.on('chatmessage', function(body) {
    renderChatMessages(body.data);
  });
}

function renderChatMessages(data) {
  const chatContent = $('#chat-content');
  const template = $.templates('#chatTemplate');
  let htmlOutput = template.render(data);
  chatContent.append(htmlOutput);
  // automatically scroll downwards
  const scrollHeight = chatContent.prop("scrollHeight");
  chatContent.animate({ scrollTop: scrollHeight }, "slow");
}

</script>

```

The logic here is very similar to `chat-users`. There's one key difference on the `listen` section. Instead of replacing the rendered output, we use `append`. Then we do a scroll animation to the bottom of the list to ensure users see the new incoming message.

Next, let's update `chatroom.ejs` to include the new `chat-messages` partial and also to update the script to call the `loadMessages()` function:

```
...
<!-- Chat Messages -->
<% include partials/chat-messages.ejs %>
...

<script type="text/javascript">
...
  loadMessages();
...
</script>
```

Your view should now look like this:

The screenshot shows a web browser window titled "Sails Chat App" at the URL "localhost:1337/chat". The interface has a teal header bar with three tabs: "Chat Room" (selected), "Profile", and "Logout". Below the header, there are two main sections: "Members" on the left and "Community Conversations" on the right. The "Members" section lists three users with their profile pictures: John Wayne, Peter Quinn, and Jane Eyre. The "Community Conversations" section shows a list of messages from different users with their timestamps and content. A vertical scrollbar is visible on the right side of the page.

Member	Message	Posted On
John Wayne	Hey Everyone! Welcome to the community!	2018-01-22T16:40:54.994Z
Peter Quinn	How's it going?	2018-01-22T16:40:54.997Z
Jane Eyre	Super excited!	2018-01-22T16:40:55.109Z

[TODO chat-post]

Let's now build a simple form that will allow users to post messages to the chat room.

Chat Post UI

Create a new file `views/partial/chat-post.ejs` and paste this code:

```
<div class="ui basic segment">
<div class="ui form">
  <div class="ui field">
    <label>Post Message</label>
    <textarea id="post-field" rows="2"></textarea>
  </div>
  <button id="post-btn" class="ui right floated large orange button">
    Post
  </button>
<div id="post-err" class="ui tiny compact negative message" style="display: none;">
  <p>Oops! Something went wrong.</p>
</div>
```

```
</div>
```

Here we're using semantic-ui elements to build the form. Next add this script to the bottom of the file:

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

function activateChat() {
    const postField = $('#post-field');
    const postButton = $('#post-btn');
    const postErr = $('#post-err');

    // Bind to click event
    postButton.click(postMessage);

    // Bind to enter key event
    postField.keypress(function(e) {
        var keycode = (e.keyCode ? e.keyCode : e.which);
        if (keycode == '13') {
            postMessage();
        }
    });
}

function postMessage() {
    if(postField.val() == "") {
        alert("Please type a message!");
    } else {
        let text = postField.val();
        io.socket.post('/postMessage', { message: text }, function(res) {
            if(jwRes.statusCode != 200) {
                postErr.html("<p>" + resData.message + "</p>")
                postErr.show();
            } else {
                postField.val(''); // clear input field
            }
        });
    }
}

</script>
```

This script is made up of two functions:

- `activateChat()`. This function binds the post button to a click event and the message box (post field) to a key press (enter) event. When either is

fired, the `postMessage()` function is called.

- `postMessage`. This function first does a quick validation to ensure the post input field is not blank. If there's a message is provided in the input field, we use the `io.socket.post()` function to send a message back to the server. Here we're using a classic callback function to handle the response from the server. If an error occurs, we display the error message. If we get a 200 status code, meaning the message was captured, we clear the post input field, ready for the next message to be typed in.

If you go back to the `chat-message` script, you'll see that we've already placed code to detect and render incoming messages. You should have also noticed that the `io.socket.post()` is sending data to the URL `/postMessage`. This is not a Blueprint route, but a custom one. Hence, we need to write code for it.

Head over to `api/controllers/UserController.js` and insert this code:

```
module.exports = {

  postMessage: async (request, response) => {
    // Make sure this is a socket request (not traditional HTTP)
    if (!request.isSocket) {
      return response.badRequest();
    }

    try {
      let user = await User.findOne({email:'johnnie86@gmail.com'})
      let msg = await ChatMessage.create({message:request.body.message})
      if(!msg.id) {
        throw new Error('Message processing failed!');
      }
      msg.createdBy = user;
      ChatMessage.publishCreate(msg);
    } catch(err) {
      return response.serverError(err);
    }

    return response.ok();
  }
};
```

Since we haven't set up basic authentication, we are hardcoding the user `johnnie86@gmail.com` for now as the author of the message. We use the

`Model.create()` Waterline ORM function to create a new record. This is a fancy way of inserting records without us writing SQL code. Next we send out a notify event to all sockets informing them that a new message has been created. We do that using the `ChatMessage.publishCreate()` function, which is defined in the Blueprints API. Before we send out the message, we make sure that the `createdBy` field is populated with a user object. This is used by `chat-messages` partial to access the avatar and the name of the user who created the message.

Next, head over to `config/routes.js` to map the `/postMessage` URL to the `postMessage` action we just defined. Insert this code:

```
...
'/chat': {
  view: 'chatroom'
}, // Add comma here
'/postMessage': {
  controller: 'ChatMessageController',
  action: 'postMessage'
}
...
```

Open `views/chatroom.js` and include the `chat-post` partial. We'll also call the `activateChat()` function right after the `loadMessages()` function:

```
...
<% include partials/chat-messages.ejs %>
...

<script type="text/javascript">
...
  activateChat();
...
</script>
```

Refresh the page and try to send several messages.

The screenshot shows a web browser window titled "Sails Chat App" at the URL "localhost:1337/chat". The interface is divided into several sections:

- Header:** Includes a back arrow, forward arrow, refresh button, and a search bar with the placeholder "localhost:1337/chat".
- Top Navigation:** Features three icons: a speech bubble for "Chat Room", a person icon for "Profile", and a right-pointing arrow for "Logout".
- Members Section:** A list of users with their profile pictures:
 - John Wayne
 - Peter Quinn
 - Jane Eyre
- Community Conversations Section:** A list of messages from different users:
 - Peter Quinn posted on 2018-01-22T16:40:54.994Z: Hey Everyone! Welcome to the community!
 - Jane Eyre posted on 2018-01-22T16:40:54.997Z: How's it going?
 - John Wayne posted on 2018-01-22T16:40:55.109Z: Super excited!
 - John Wayne posted on 2018-01-22T16:46:33.033Z: Hey! I can post now
- Post Message Form:** A text input field containing "This is awesome :)" and a "Post" button.

You should now have a functional chat system. Review the project source code in case you get stuck.

Basic Authentication

Setting up a proper authentication and authorization system is outside the scope of this tutorial. So we'll settle for a basic password-less authentication system. Let's first build the signup and login form.

Login/Sign Up Form

Create a new file `views/auth-form.ejs` and paste the following content:

```
<form method="post" action="/auth/authenticate" class="ui form">
<div class="field">
  <label>Full Names</label>
  <input type="text" name="name" placeholder="Full Names" value="<%=
</div>
<div class="required field">
  <label>Email</label>
  <input type="email" name="email" placeholder="Email" value="<%=
</div>
<button class="ui teal button" type="submit" name="action" value="si
<button class="ui blue button" type="submit" name="action" value="lo
<p class="note">*Provide email only for Login</p>
</form>
<% if(typeof error != 'undefined') { %>
<div class="ui error message">
<div class="header"><%= error.title %></div>
<p><%= error.message %></p>
</div>
<% } %>
```

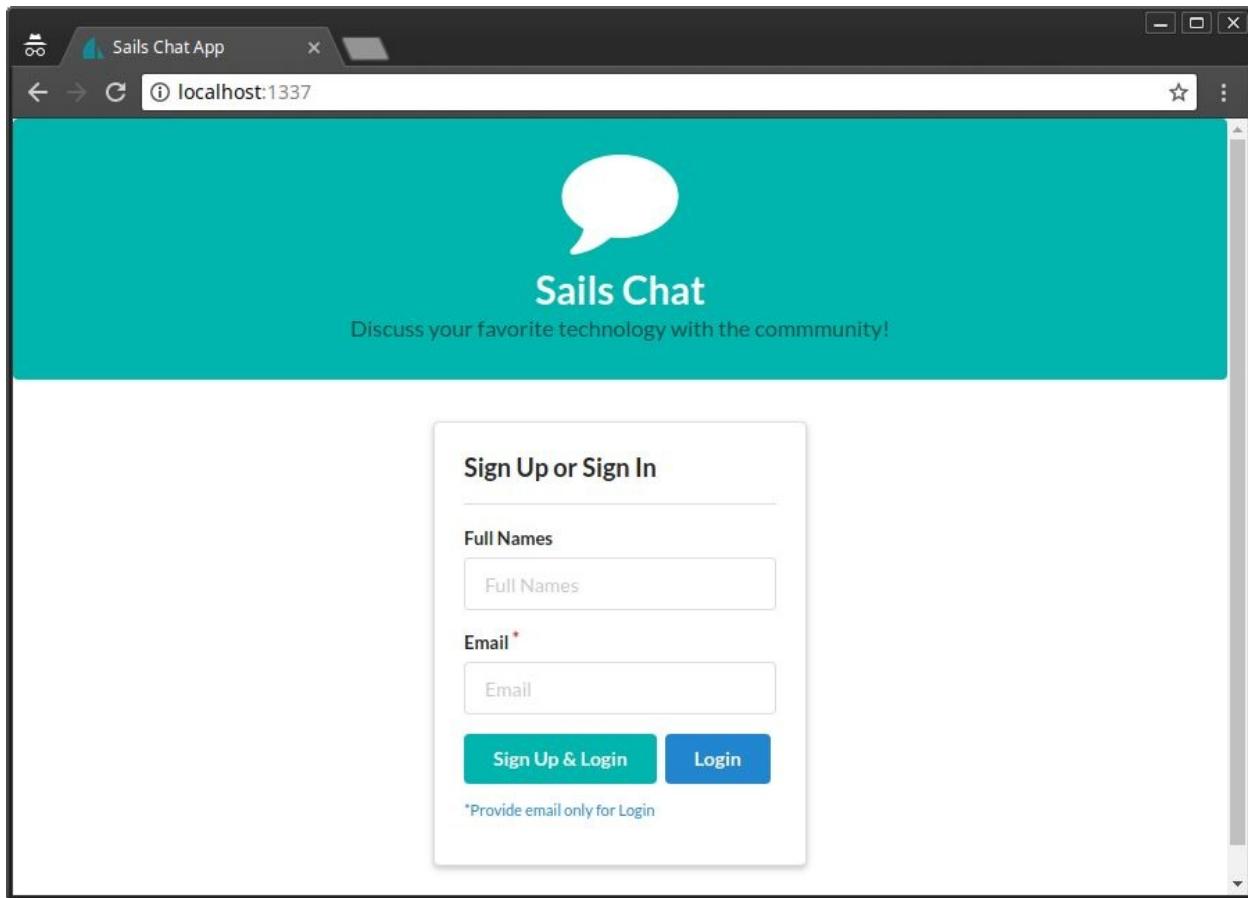
Next open `views/homepage.ejs` and replace the TODO line with this include statement:

```
...
<% include partials/auth-form.ejs %>
...
```

We've created a form that allows you to create a new account by providing an input for name and email. When you click `Signup` & `Login`, a new user record is created and you get logged in. However, if the email is already being used by another user, an error message will be displayed. If you just want to log in, just

provide the email address and click the Login button. Upon successful authentication, you'll be redirected to the /chat URL.

Right now, everything I've just said isn't working. We'll need to implement that logic. First, let's navigate to / address to confirm that the auth-form looks goods.



Policy

Now that we're setting up an authentication system, we need to protect /chat and /profile routes from public access. Only authenticated users should be allowed to access them. Open config/policies.js and insert this code:

```
ChatMessageController: {
  '*': 'sessionAuth'
},  
  
UserController: {
  '*': 'sessionAuth'
```

```
},
```

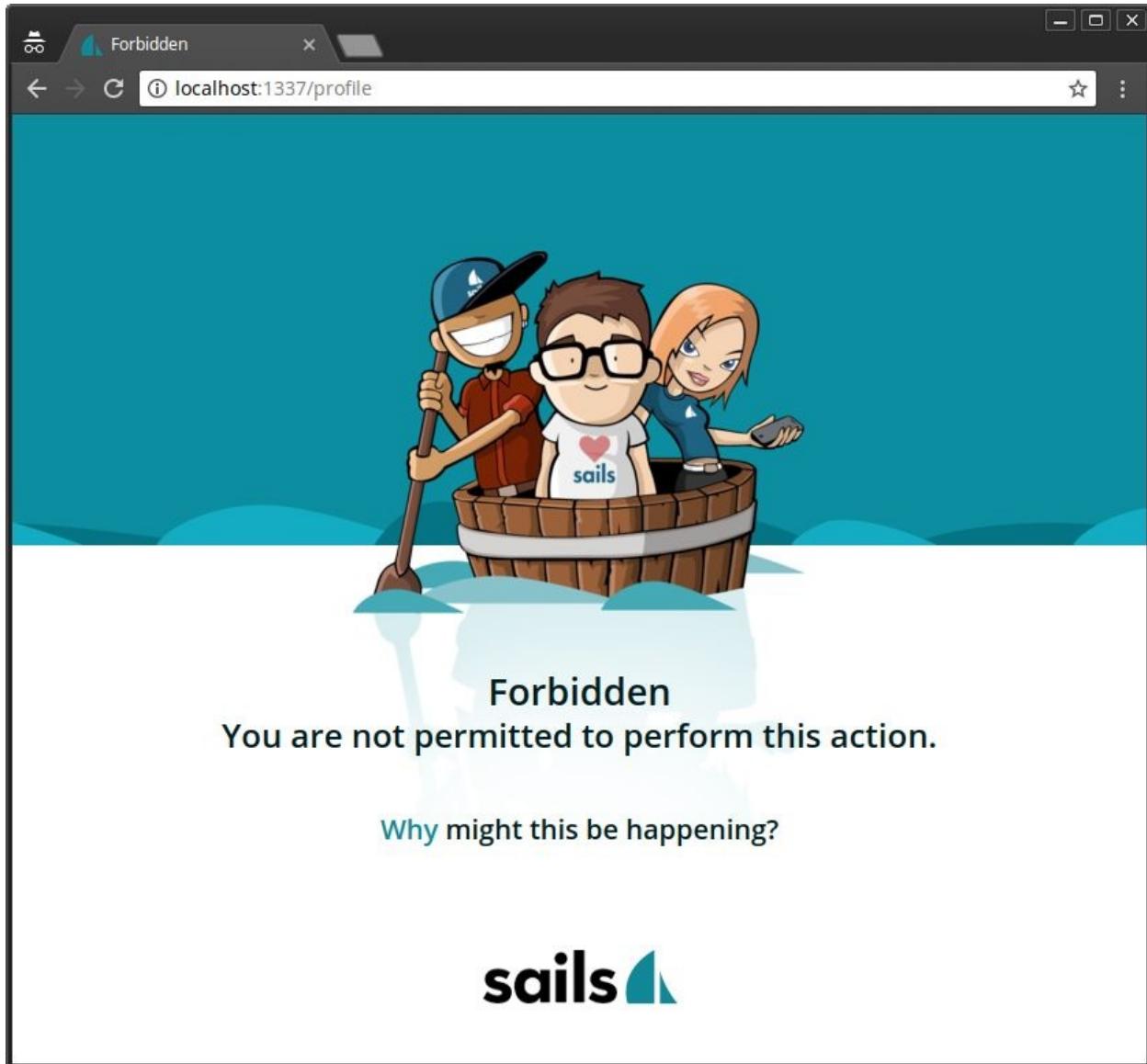
By specifying the name of the controller, we have also effectively blocked all routes provided by the Blueprint API for Users and Chat Messages. Unfortunately, policies only work with controllers. This means the route /chat can't be protected in its current state. We need to define a custom action for it. Open api/controller/ChatroomController.js and insert this code:

```
...
render: (request, response) => {
  return response.view('chatroom');
},
```

Then replace the route config for /chat with this one in config/routes.js:

```
...
'/chat': {
  controller: 'ChatMessageController',
  action: 'render'
},
...
```

The /chat route should now be protected from public access. If you restart your app and try to access /profile, /chat, /user or /chatmessage, you'll be met with the following forbidden message:



If you'd like to redirect users to the login form instead, head over to `api/policies/sessionAuth` and replace the `forbidden` call with a `redirect` call like this:

```
...
// return res.forbidden('You are not permitted to perform this action')
return res.redirect('/');
...
```

Try accessing the forbidden pages again, and you'll automatically be redirected to the home page. Let's now implement the Signup and Login code.

Auth Controller and Service

You'll need to stop Sails.js first in order to run this command:

```
sails generate controller Auth
```

This will create a blank api/controllers/AuthController for us. Open it and insert this code:

```
authenticate: async (request, response) => {

    // Sign up user
    if(request.body.action == 'signup') {
        // Validate signup form

        // Check if email is registered

        // Create new user
    }

    // Log in user
},

logout: (request, response) => {
    // Logout user
}
```

I've placed in comments explaining how the logic will flow. We can place the relevant code here. However, Sails.js recommends we keep our controller code simple and easy to follow. To achieve this, we need to write helper functions that will help us with each of the above commented tasks. To create these helper functions, we need to create a service. Do this by creating a new file api/services/AuthService.js. Insert the following code:

```
/**
* AuthService.js
*
**/

const gravatar = require('gravatar')

// Where to display auth errors
const view = 'homepage';
```

```
module.exports = {

  sendAuthError: (response, title, message, options) => {
    options = options || {};
    const { email, name} = options;
    response.view(view, { error: {title, message}, email, name });
    return false;
  },

  validateSignupForm: (request, response) => {
    if(request.body.name == '') {
      return AuthService.sendAuthError(response, 'Signup Failed!', "Yo
    } else if(request.body.email == '') {
      return AuthService.sendAuthError(response, 'Signup Failed!', "Yo
    }
    return true;
  },

  checkDuplicateRegistration: async (request, response) => {
    try {
      let existingUser = await User.findOne({email:request.body.email})
      if(existingUser) {
        const options = {email:request.body.email, name:request.body.n
        return AuthService.sendAuthError(response, 'Duplicate Registrat
      }
      return true;
    } catch (err) {
      response.serverError(err);
      return false;
    }
  },
}

registerUser: async (data, response) => {
  try {
    const {name, email} = data;
    const avatar = gravatar.url(email, {s:200}, "https");
    let newUser = await User.create({name, email, avatar});
    // Let all sockets know a new user has been created
    User.publishCreate(newUser);
    return newUser;
  } catch (err) {
    response.serverError(err);
    return false;
  }
},
}

login: async (request, response) => {
```

```

try {
    let user = await User.findOne({email:request.body.email});
    if(user) { // Login Passed
        request.session.userId = user.id;
        request.session.authenticated = true;
        return response.redirect('/chat');
    } else { // Login Failed
        return AuthService.sendAuthError(response, 'Login Failed!', "T
    }
} catch (err) {
    return response.serverError(err);
}
},
logout: (request, response) => {
    request.session.userId = null;
    request.session.authenticated = false;
    response.redirect('/');
}
}

```

Examine the code carefully. As an intermediate developer, you should be able to understand the logic. I haven't done anything fancy here. However, I would like to mention a few things:

- Gravatar. You need to install Gravatar. It's a JavaScript library for generating Gravatar URLs based on the email address.

```

```bash
npm install gravatar --save
```

```

- `User.publishCreate(newUser)`. Just like `ChatMessages`, we fire an event notifying all sockets that a new user has just been created. This will cause all logged-in clients to re-fetch the users data. Review `views/partial/chat-users.js` to see what I'm talking about.
- `request.session`. Sails.js provides us with a [session store](#) which we can use to pass data between page requests. The default Sails.js session lives in memory, meaning if you stop the server the session data gets lost. In the AuthService, we're using session to store `userId` and `authenticated` status.

With the logic in `AuthService.js` firmly in place, we can go ahead and update `api/controllers/AuthController` with the following code:

```
module.exports = {

  authenticate: async (request, response) => {
    const email = request.body.email;

    if(request.body.action == 'signup') {
      const name = request.body.name;
      // Validate signup form
      if(!AuthService.validateSignupForm(request, response)) {
        return;
      }
      // Check if email is registered
      const duplicateFound = await AuthService.checkDuplicateRegistration(email);
      if(!duplicateFound) {
        return;
      }
      // Create new user
      const newUser = await AuthService.registerUser({name, email});
      if(!newUser) {
        return;
      }
    }

    // Attempt to log in
    const success = await AuthService.login(request, response);
  },

  logout: (request, response) => {
    AuthService.logout(request, response);
  };
};
```

See how much simple and readable our controller is. Next, let's do some final touches.

Final Touches

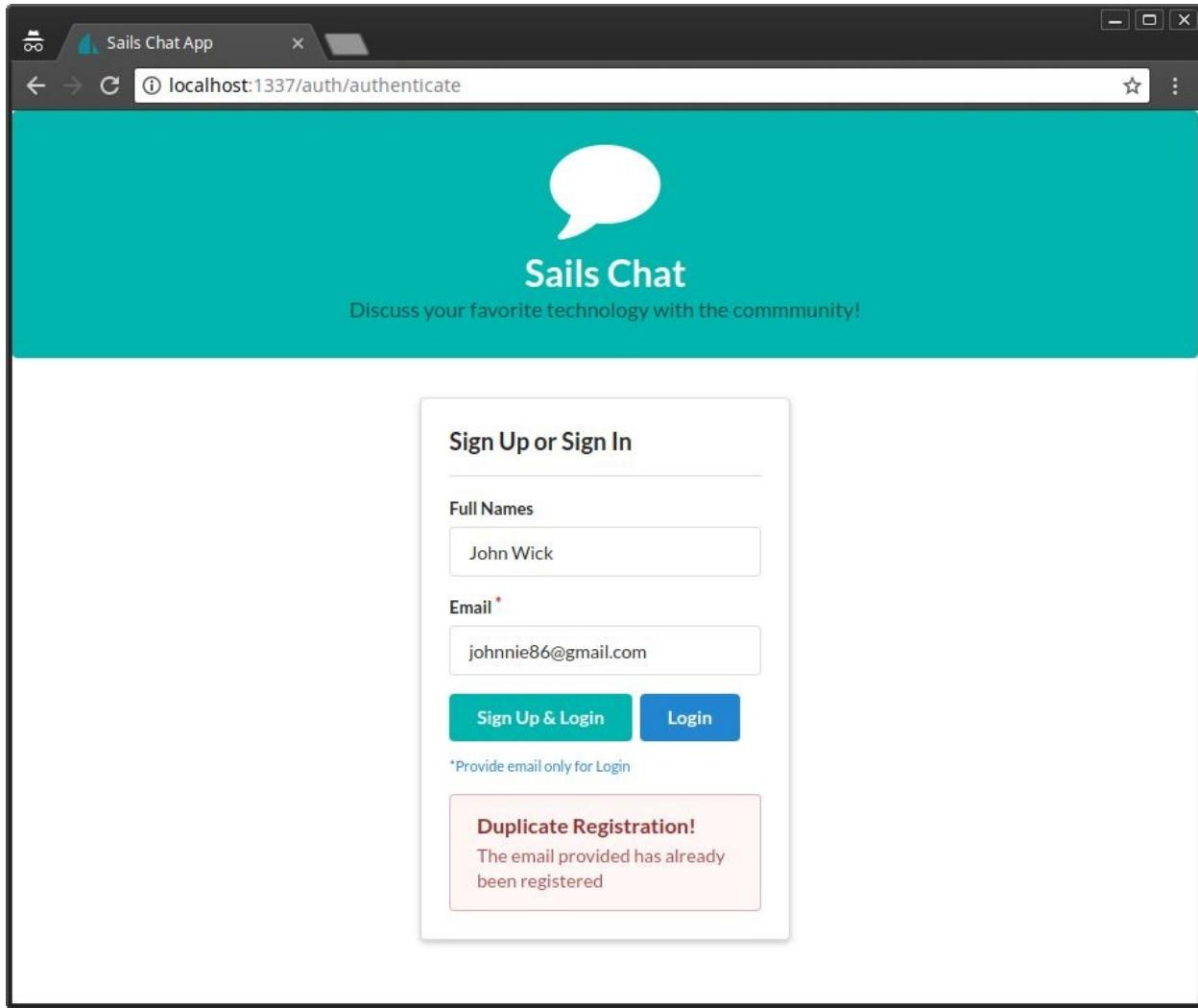
Now that we have authentication set up, we should remove the hardcoded value we placed in the `postMessage` action in `api/controllers/ChatMessageController`. Replace the email code with this one:

```
...  
let user = await User.findOne({id:request.session.userId});  
...
```

I'd like to mention something you may not have noticed, if you look at the logout URL in `views/partials/menu.ejs`, we've placed this address `/auth/logout`. If you look at `config/routes.js`, you'll notice that we haven't placed a URL for it. Surprisingly, when we run the code, it works. This is because Sails.js uses a convention to determine which controller and action is needed to resolve a particular address.

By now you should be having a functional MVP chat application. Fire up your app and test the following scenarios:

- sign up without entering anything
- sign up by filling only name
- sign up by only filling email
- sign up by filling name and a registered email — for example, `johnnie86@gmail.com` or `jane@hotmail.com`
- sign up using your name and email
- update your profile
- try posting a blank message
- post some messages
- open another browser and log in as another user, put each browser side by side and chat
- log out and create a new account.



The screenshot shows two side-by-side instances of the Sails Chat App interface. Both instances have a teal header with "Sails Chat App" and a user profile icon. The left instance shows a "Chat Room" tab selected, displaying a "Members" section with icons for John Wayne, Peter Quinn, Jane Eyre, John Wick, and Michael Wanyoike. Below this is a "Community Conversations" section with messages from Peter Quinn, Jane Eyre, John Wayne, and Michael Wanyoike. The right instance shows a "Profile" tab selected, displaying a "Members" section with the same five users. Below this is a "Community Conversations" section with the same messages. Both instances have a "Post Message" input field and a "Post" button.

Phew! That's a lot of functionality we've just implemented in one sitting and then tested. With a few more weeks, we could whip out a production ready chat system integrated with more features, such as multiple chat rooms, channel attachments, smiley icons and social accounts integration!

Summary

During this tutorial, we didn't put the name of the logged-in user somewhere at the top menu. You should be capable of fixing this yourself. If you've read the entire tutorial, you should now be proficient in building applications using Sails.js.

The goal of this tutorial is to show you that can come from a non-JavaScript MVC framework and build something awesome with relatively few lines of code. Making use of the Blueprint API will help you implement features faster. I also recommend you learn to integrate a more powerful front-end library — such as React, Angular or Vue — to create a much more interactive web application. In addition, learning how to write tests for Sails.js to automate the testing process is a great weapon in your programming arsenal.

Chapter 7: Passport Authentication for Node.js Applications

by Paul Orac

In this tutorial, we'll be implementing authentication via Facebook and GitHub in a [Node.js](#) web application. For this, we'll be using [Passport](#), an authentication middleware for Node.js. Passport supports authentication with [OpenId/OAuth](#) providers.

Express Web App

Before getting started, make sure you have [Node.js installed](#) on your machine.

We'll begin by creating the folder for our app and then accessing that folder on the terminal:

```
mkdir AuthApp  
cd AuthApp
```

To create the node app we'll use the following command:

```
npm init
```

You'll be prompted to provide some information for Node's package.json. Just hit enter until the end to leave the default configuration.

Next, we'll need an HTML file to send to the client. Create a file called auth.html in the root folder of your app, with the following contents:

```
<html>  
  <head>  
    <title>Node.js OAuth</title>  
  </head>  
  <body>  
    <a href=auth/facebook>Sign in with Facebook</a>  
    <br><br>  
    <a href=auth/github>Sign in with Github</a>  
  </body>  
</html>
```

That's all the HTML we'll need for this tutorial.

You'll also require [Express](#), a framework for building web apps that's inspired by Ruby's Sinatra. In order to install Express, from the terminal type the following command:

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

Once you've done that, it's time to write some code.

Create a file `index.js` in the root folder of your app and add the following content to it:

```
/* EXPRESS SETUP */

const express = require('express');
const app = express();

app.get('/', (req, res) => res.sendFile('auth.html', { root : __dirname });

const port = process.env.PORT || 3000;
app.listen(port, () => console.log('App listening on port ' + port))
```

In the code above, we require Express and create our Express app by calling [express\(\)](#). Then we declare the route for the home page of our app. There we send the HTML file we've created to the client accessing that route. Then, we use `process.env.PORT` to set the port to the environment port variable if it exists. Otherwise, we'll default to 3000, which is the port we'll be using locally. This gives you enough flexibility to switch from development, directly to a production environment where the port might be set by a service provider like, for instance, [Heroku](#). Right below, we call [app.listen\(\)](#) with the port variable we set up, and a simple log to let us know that it's all working fine, and on which port is the app listening.

Now we should start our app to make sure all is working correctly. Simply write the following command on the terminal:

```
node index.js
```

You should see the message: `App listening on port 3000`. If that's not the case, you probably missed a step. Go back and try again.

Moving on, let's see if our page is being served to the client. Go to your web browser and navigate to `http://localhost:3000`.

If you can see the page we created in `auth.html`, we're good to go.

Head back to the terminal and stop the app with `ctrl + c`. So remember, when I say start the app, you write `node index.js`, and when I say stop the app, you do `ctrl + c`. Clear? Good, you've just been programmed :-)

Setting up Passport

As you'll soon come to realize, Passport makes it a breeze to provide authentication for our users. Let's install Passport with the following command:

```
npm install passport --save
```

Now we have to set up Passport. Add the following code at the bottom of the `index.js` file:

```
/* PASSPORT SETUP */

const passport = require('passport');
app.use(passport.initialize());
app.use(passport.session());

app.get('/success', (req, res) => res.send("You have successfully lo
app.get('/error', (req, res) => res.send("error logging in"));

passport.serializeUser(function(user, cb) {
  cb(null, user);
});

passport.deserializeUser(function(obj, cb) {
  cb(null, obj);
});
```

Here we require Passport and initialize it along with its session authentication middleware, directly inside our Express app. Then, we set up the '`/success`' and '`/error`' routes, which will render a message telling us how the authentication went. It's the same syntax for our last route, only this time instead of using `[res.sendFile()]` (<http://expressjs.com/en/api.html#res.sendFile>) we're using `[res.send()]` (<http://expressjs.com/en/api.html#res.send>), which will render the given string as `text/html` in the browser. Then we're using `serializeUser` and `deserializeUser` callbacks. The first one will be invoked on authentication and its job is to serialize the user instance and store it in the session via a cookie. The second one will be invoked every subsequent request to deserialize the instance, providing it the unique cookie identifier as a "credential". You can read more about that in the Passport [documentation](#).

As a side note, this very simple sample app of ours will work just fine without `deserializeUser`, but it kills the purpose of keeping a session, which is something you'll need in every app that requires login.

That's all for the actual Passport setup. Now we can finally get onto business.

Implementing Facebook Authentication

The first thing we'll need to do in order to provide *Facebook Authentication* is installing the [passport-facebook](#) package. You know how it goes:

```
npm install passport-facebook --save
```

Now that everything's set up, adding *Facebook Authentication* is extremely easy. Add the following code at the bottom of your `index.js` file:

```
/* FACEBOOK AUTH */

const FacebookStrategy = require('passport-facebook').Strategy;

const FACEBOOK_APP_ID = 'your app id';
const FACEBOOK_APP_SECRET = 'your app secret';

passport.use(new FacebookStrategy({
    clientID: FACEBOOK_APP_ID,
    clientSecret: FACEBOOK_APP_SECRET,
    callbackURL: "/auth/facebook/callback"
},
function(accessToken, refreshToken, profile, cb) {
    return cb(null, profile);
}
));

app.get('/auth/facebook',
    passport.authenticate('facebook'));

app.get('/auth/facebook/callback',
    passport.authenticate('facebook', { failureRedirect: '/error' }),
    function(req, res) {
        res.redirect('/success');
    }
);
```

Let's go through this block of code step by step. First, we require the `passport-facebook` module. Then, we declare the variables in which we'll store our *app id* and *app secret* (we'll see how to get those shortly). After that, we tell Passport to use an instance of the `FacebookStrategy` we required. To instantiate said strategy we give it our *app id* and *app secret* variables and the `callbackURL` that we'll use to authenticate the user. As a second parameter, it takes a function that

will return the profile info provided by the user.

Further down, we set up the routes to provide authentication. As you can see in the `callbackURL` we redirect the user to the `/error` and `/success` routes we defined earlier. We're using [`passport.authenticate`](#), which attempts to authenticate with the given strategy on its first parameter, in this case `facebook`. You probably noticed that we're doing this twice. On the first one, it sends the request to our Facebook app. The second one is triggered by the callback URL, which Facebook will use to respond to the login request.

Now you'll need to create a Facebook app. For details on how to do that, consult Facebook's very detailed guide [`Creating a Facebook App`](#), which provides step by step instructions on how to create one.

When your app is created, go to *Settings* on the app configuration page. There you'll see your *app id* and *app secret*. Don't forget to change the variables you declared for them on the `index.js` file with their corresponding values.

Next, enter "localhost" in the *App Domains* field. Then, go to *Add platform* at the bottom of the page and choose *Website*. Use <http://localhost:3000/auth/facebook/callback> as the *Site URL*.

On the left sidebar, under the *Products* section, you should see *Facebook Login*. Click to get in there.

Lastly, set the *Valid OAuth redirect URIs* field to <http://localhost:3000/auth/facebook/callback>.

If you start the app now and click the *Sign in with Facebook* link, you should be prompted by Facebook to provide the required information, and after you've logged in, you should be redirected to the `/success` route, where you'll see the message `You have successfully logged in`.

That's it! you have just set up *Facebook Authentication*. Pretty easy, right?

Implementing GitHub Authentication

The process for adding *Github Authentication* is quite similar to what we did for Facebook. First, we'll install the [passport-github](#) module:

```
npm install passport-github --save
```

Now go to the `index.js` file and add the following lines at the bottom:

```
/* GITHUB AUTH */

const GitHubStrategy = require('passport-github').Strategy;

const GITHUB_CLIENT_ID = "your app id"
const GITHUB_CLIENT_SECRET = "your app secret";

passport.use(new GitHubStrategy({
    clientID: GITHUB_CLIENT_ID,
    clientSecret: GITHUB_CLIENT_SECRET,
    callbackURL: "/auth/github/callback"
},
function(accessToken, refreshToken, profile, cb) {
    return cb(null, profile);
}
));

app.get('/auth/github',
  passport.authenticate('github'));

app.get('/auth/github/callback',
  passport.authenticate('github', { failureRedirect: '/error' }),
  function(req, res) {
    res.redirect('/success');
}
);
```

This looks familiar! It's practically the same as before. The only difference is that we're using the *GithubStrategy* instead of *FacebookStrategy*.

So far so ... the same. In case you hadn't yet figured it out, the next step is to create our *GitHub App*. GitHub has a very simple guide, [Creating a GitHub app](#), that will guide you through the process.

When you're done, in the configuration panel you'll need to set the *Homepage URL* to `http://localhost:3000/` and the *Authorization callback URL* to `http://localhost:3000/auth/github/callback`, just like we did with Facebook.

Now, simply restart the Node server and try logging in using the GitHub link.

It works! Now you can let your users log in with GitHub.

Conclusion

In this tutorial, we saw how Passport made the task of authentication quite simple. Implementing Google and Twitter authentication follows a nearly identical pattern. I challenge you to implement these using the [passport-google](#) and [passport-twitter](#) modules. In the meantime, the code for this app is available on [GitHub](#).

Chapter 8: Local Authentication Using Passport in Node.js

by Paul Orac

In the last chapter, we talked about authentication using Passport as it relates to social login (Google, Facebook, GitHub, etc.). In this chapter, we'll see how we can use [Passport](#) for local authentication with a MongoDB back end.

All of the code from this article is available for download on [GitHub](#).

Prerequisites

- [Node.js](#) — Download and install Node.js.
- [MongoDB](#) — Download and install MongoDB Community Server. Follow the instructions for your OS. Note, if you're using Ubuntu, this [guide](#) can help you get Mongo up and running.

Creating the Project

Once all of the prerequisite software is set up, we can get started.

We'll begin by creating the folder for our app and then accessing that folder on the terminal:

```
mkdir AuthApp  
cd AuthApp
```

To create the node app, we'll use the following command:

```
npm init
```

You'll be prompted to provide some information for Node's package.json. Just hit enter until the end to leave the default configuration.

HTML

Next, we'll need a form with username and password inputs as well as a *Submit* button. Let's do that! Create a file called auth.html in the root folder of your app, with the following contents:

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

That will do just fine.

Setting up Express

Now we need to install [Express](#), of course. Go to the terminal and write this command:

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

We'll also need to install the [body-parser](#) middleware which is used to parse the request body that Passport uses to authenticate the user.

Let's do that. Run the following command:

```
npm install body-parser --save
```

When that's done, create a file `index.js` in the root folder of your app and add the following content to it:

```
/* EXPRESS SETUP */

const express = require('express');
const app = express();

const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', (req, res) => res.sendFile('auth.html', { root : __dirname });

const port = process.env.PORT || 3000;
app.listen(port, () => console.log('App listening on port ' + port))
```

We're doing almost the same as in the previous tutorial. First we require Express and create our Express app by calling `[express()]` (<http://expressjs.com/en/api.html#express>). The next line is the only difference with our previous Express setup. We'll need the [body-parser](#) middleware this time, in order for authentication to work correctly. Then we declare the route for the home page of our app. There we send the HTML file we created to the client accessing that route. Then, we use `process.env.PORT` to set the port to the environment port variable if it exists. Otherwise, we'll default to 3000, which is the port we'll be using locally. This gives you enough flexibility to switch from development, directly to a production environment where the port

might be set by a service provider like, for instance, [Heroku](#). Right below we called `[app.listen()](http://expressjs.com/en/api.html#app.listen)` with the port variable we set up and a simple log to let us know that it's all working fine and on which port is the app listening.

That's all for the Express setup. Now we have to set up Passport, exactly as we did the last time. I'll show you how to do that in case you didn't read the previous tutorial.

Setting up Passport

First, we install *passport* with the following command:

```
npm install passport --save
```

Then, add the following lines at the bottom of your `index.js` file:

```
/* PASSPORT SETUP */

const passport = require('passport');
app.use(passport.initialize());
app.use(passport.session());

app.get('/success', (req, res) => res.send("Welcome "+req.query.user));
app.get('/error', (req, res) => res.send("error logging in"));

passport.serializeUser(function(user, cb) {
  cb(null, user.id);
});

passport.deserializeUser(function(id, cb) {
  User.findById(id, function(err, user) {
    cb(err, user);
  });
});
```

Here, we require `passport` and initialize it along with its session authentication middleware, directly inside our Express app. Then, we set up the `'/success'` and `'/error'` routes which will render a message telling us how the authentication went. If it succeeds, we're going to show the `username` parameter, which we'll pass to the request. We're using the same syntax for our last route, only this time instead of using `[res.sendFile()]` (<http://expressjs.com/en/api.html#res.sendFile>) we're using `[res.send()]` (<http://expressjs.com/en/api.html#res.send>), which will render the given string as `text/html` on the browser. Then we're using `serializeUser` and `deserializeUser` callbacks. The first one will be invoked on authentication, and its job is to serialize the user instance with the information we pass to it (the user ID in this case) and store it in the session via a cookie. The second one will be invoked every subsequent request to deserialize the instance, providing it the unique cookie identifier as a “credential”. You can read more

about that in the [Passprot documentation](#).

As a side note, this very simple sample app of ours will work just fine without `deserializeUser`, but it kills the purpose of keeping a session, which is by all means something you'll need in every app that requires login.

Creating a MongoDB Data Store

Since we're assuming you've already installed Mongo, you should be able to start the mongod server using the following command:

```
sudo mongod
```

From another terminal, launch the Mongo shell:

```
mongo
```

Within the shell, issue the following commands:

```
use MyDatabase;  
db.userInfo.insert({'username': 'admin', 'password': 'admin'});
```

The first command creates a data store named `MyDatabase`. The second command creates a collection named `userInfo` and inserts a record. Let's insert a few more records:

```
db.userInfo.insert({'username': 'jay', 'password': 'jay'});  
db.userInfo.insert({'username': 'roy', 'password': 'password'});
```

Retrieving Stored Data

We can view the data we just added using the following command:

```
db.userInfo.find();
```

The resulting output is shown below:

```
{ "_id" : ObjectId("5321cd6dbb5b0e6e72d75c80"), "username" : "admin"  
{ "_id" : ObjectId("5321d3f8bb5b0e6e72d75c81"), "username" : "jay",  
{ "_id" : ObjectId("5321d406bb5b0e6e72d75c82"), "username" : "roy",
```

We can also search for a particular username and password:

```
db.userInfo.findOne({'username': 'admin', 'password': 'admin'})
```

This command would return only the admin user.

Connection Mongo to Node with Mongoose

Now that we have a database with records in it, we need a way to communicate with it from our application. We'll be using [Mongoose](#) to achieve this. Why don't we just use plain Mongo? Well, as the Mongoose devs like to say on their website:

writing MongoDB validation, casting and business logic boilerplate is a drag.

Mongoose will simply make our lives easier and our code more elegant.

Let's go ahead and install it with the following command:

```
npm install mongoose --save
```

Now we have to configure Mongoose. You know the drill: add the following code at the bottom of your `index.js` file:

```
/* MONGOOSE SETUP */

const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/MyDatabase');

const Schema = mongoose.Schema;
const UserDetail = new Schema({
    username: String,
    password: String
});
const UserDetails = mongoose.model('userInfo', UserDetail, 'userInfo')
```

Here we require the `mongoose` package. Then we connect to our database using `mongoose.connect` and give it the path to our database. Then we're making use of [Schema](#) to define our data structure. In this case, we're creating a `UserDetail` schema with `username` and `password` fields. After that, we create a [model](#) from that schema. The first parameter is the name of the collection in the database. The second one is the reference to our `Schema`, and the third one is the name we're assigning to the collection inside Mongoose.

That's all for the Mongoose setup. We can now move on to implementing our

Strategy.

Implementing Local Authentication

It's time to configure our authentication strategy using [passport-local](#). Let's go ahead and install it. Run the following command:

```
npm install passport-local --save
```

Finally, we've come to the last portion of the code! Add it to the bottom of your `index.js` file:

```
/* PASSPORT LOCAL AUTHENTICATION */

const LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
  function(username, password, done) {
    UserDetails.findOne({
      username: username
    }, function(err, user) {
      if (err) {
        return done(err);
      }

      if (!user) {
        return done(null, false);
      }

      if (user.password != password) {
        return done(null, false);
      }
      return done(null, user);
    });
  }
));
app.post('/', passport.authenticate('local', { failureRedirect: '/error' }), function(req, res) {
  res.redirect('/success?username=' + req.user.username);
});
```

Let's go through this. First, we require the `passport-local` Strategy. Then we tell Passport to use an instance of the `LocalStrategy` we required. There we

simply use the same command that we used in the Mongo shell to find a record based on the username. If a record is found and the password matches, the above code returns the user object. Otherwise, it returns false.

Below the strategy implementation is our post, with the [passport.authenticate] (<http://www.passportjs.org/docs/authenticate/>) method which attempts to authenticate with the given strategy on its first parameter, in this case 'local'. It will redirect us to '/error' if it fails. Otherwise it'll redirect us to the '/success' route, sending the username as a parameter. That's how we get the username to show on the line req.query.username we saw earlier.

That's all we need for the app to work. We're done!

Restart your Node server and point your browser to `http://localhost:3000/` and try to log in with "admin" as username and "admin" as password. If it all goes well, you should see the message "Welcome admin!!" in the browser.

Conclusion

In this article, we learned how to implement local authentication using Passport in a Node.js application. In the process, we also learned how to connect to MongoDB using the Mongoose.

We only added the necessary modules for this app to work — nothing more, nothing less. For a production app, you'll need to add other middlewares and separate your code in modules. You can take that as a challenge to set up a clean and scalable environment and grow it into something useful.

Chapter 9: An Introduction to NodeBots

by Patrick Catanzariti

Many web developers out there would love the chance to build an incredibly cool robot that they can control via JavaScript, right? I'm here to tell you that this is already possible today! Right now.

NodeBots have been around for a while, and the community around them is growing like wildfire. In this article, I'm going to explain what NodeBots are, how they work and how you can get started tinkering away at robot creation.

What is a Microcontroller?

Before I get too far into things, we'll be mentioning microcontrollers quite frequently. A microcontroller is a tiny and very simple computer. It has a simple physical programmable circuit board that can detect various inputs and send outputs. An Arduino is a type of microcontroller. It's actually one of the most common ones for newcomers to experiment with. There are other sorts of microcontrollers too that can be powered by Node, including [Particle boards](#) (my favorite!), [BeagleBone boards](#), [Tessel boards](#) (the board itself runs on JS) and [Espruino boards](#) (also runs on JS). In this article, I'll be focusing on Arduinos, as they're the most common.

What are NodeBots?

NodeBots are (quite literally) robots of one kind or another that can be controlled via Node. They can have everything from wheels, movable arms and legs, motion detectors, cameras, LED displays, the ability to play sound and so much more. The only limits are your imagination and the components you can find and put together!

The whole idea of NodeBots evolved through the increasing capabilities of Node.js and the interest of a few developers like [Nikolai Onken](#), [Jörn Zaefferer](#), [Chris Williams](#), [Julian Gautier](#) and [Rick Waldron](#) who worked to develop the various Node modules we use in NodeBots today. The Node package called [node-serialport](#) by Chris Williams started it all, allowing access to real world devices via reading and writing to serial ports at a low level.

Julian Gautier then implemented the Firmata protocol, a protocol used to access microcontrollers like Arduinos via software on a computer, using JavaScript in his Node.js [Firmata](#) library.

Rick Waldron took it a massive step further. Using the Firmata library as a building block, he created a whole JavaScript Robotics and IoT programming framework called [Johnny-Five](#). The Johnny-Five framework makes controlling everything from LEDs to various types of sensors relatively simple and painfree. This is what many NodeBots now use to achieve some very impressive feats!

Where To Start

If you're completely new to the idea of building robots and any sort of real-world, JavaScript-controlled device, there are plenty of incredible resources for you to get started with. The very first thing I'd recommend you do is find yourself a good Arduino kit that provides a good range of components and sensors to give you a range of items to play around with. Below, I've got a list of some of the Arduino starter kits that are available from various companies. If the below list looks overwhelming, don't worry! They all contain very similar components and are all a good choice for beginners.

Starter Kits

- **SparkFun Inventors Kit.** This is the kit that started it all for me years ago! It comes with a range of standard components like colored LED lights, sensors, buttons, a motor, a tiny speaker and more. It also comes with a guide and sample projects you can use to build your skills. You can find it here: [SparkFun Inventor's Kit](#).
- **Freetronics Experimenter's Kit for Arduino.** This kit is by an Australian-based company called Freetronics. It has very similar components to the SparkFun one, with a few small differences. It also has its own guide with sample projects to try as well. For those based in Australia, these kits and other Freetronics parts are available at Jaycar. You can also order it online here: [Freetronics Experimenter's Kit](#).
- **Seeed Studio ARDX starter kit.** Seeed Studio have their own starter kit too, which is also very similar to the SparkFun and Freetronics ones. It has its own guide and such too! You can find it here: [ARDX - The starter kit for Arduino](#).
- **Adafruit ARDX Experimentation Kit for Arduino.** This kit is also very similar to the ones above with its own guide. You can find it here: [Adafruit ARDX Experimentation Kit for Arduino](#).
- **Arduino Starter Kit.** The guys at Arduino.cc have their own official kit that's available too. The starter kit is similar to the ones above but has some interesting sample projects like a "Love-O-Meter". You can find it here and often at other resellers too: [Arduino Starter Kit](#).

With all of the above kits, keep in mind that none of them are targeted towards

NodeBot development. So the examples in booklets and such are written in the simplified C++ code that Arduino uses. For examples using Node, see the resources below.

Resources for Learning NodeBots

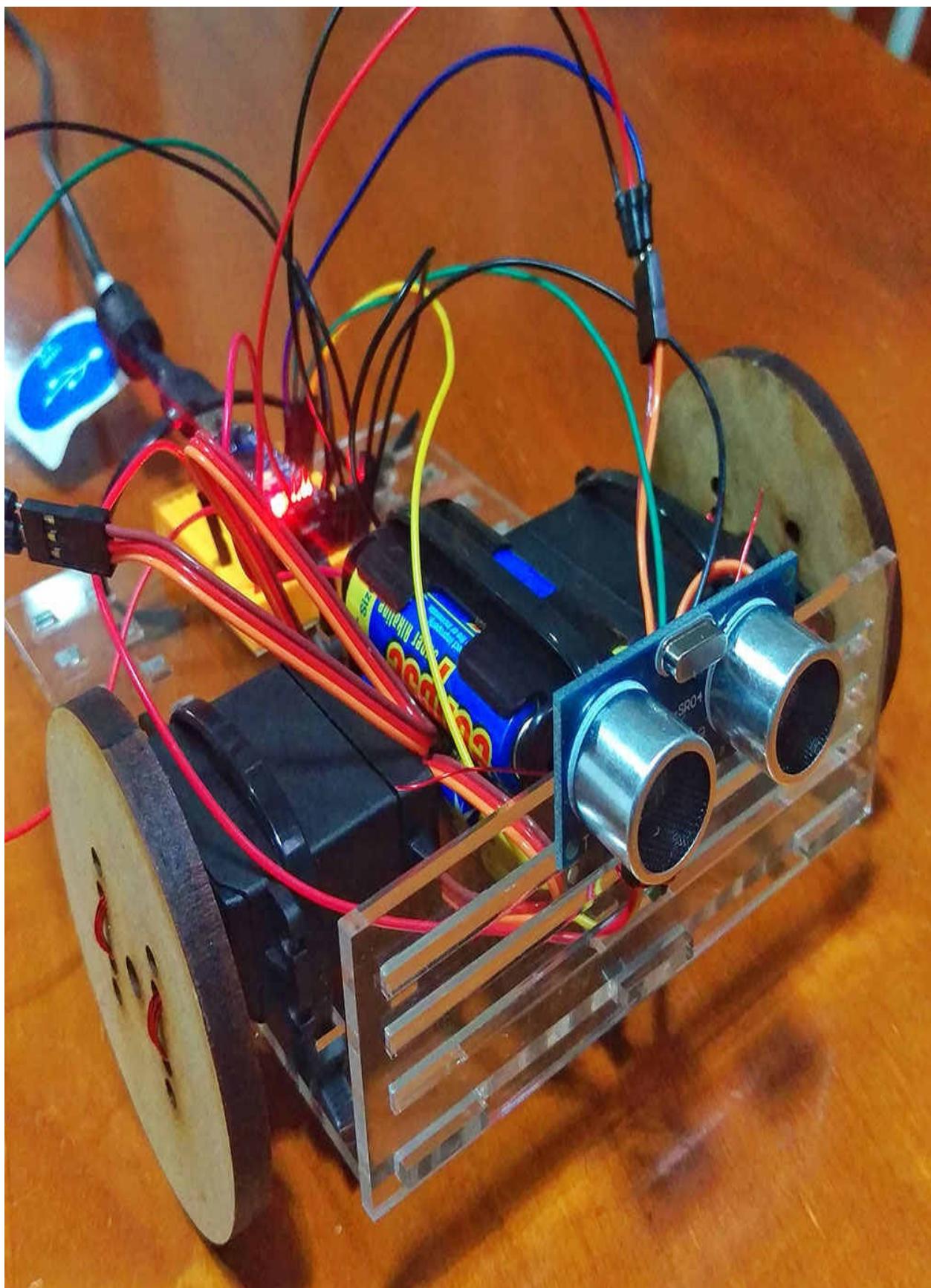
There are a few key spots where you can learn how to put together various NodeBot projects on the Web. Here are a few recommendations:

- [Controlling an Arduino with Node.js and Johnny-Five](#). This is a free SitePoint screencast I recorded a little while ago that introduces the basics of connecting up an Arduino to Node.js and using the framework to turn an LED light on and off.
- [Arduino Experimenter's Guide for NodeJS](#). An adaptation by Anna Gerber and other members of the NodeBots community from the SparkFun version of .:oomlout:.’s ARDX Guide. It shows how to do many of the examples from the kits mentioned above in Node instead of the simplified C++ code from Arduino.
- [The official Johnny-Five website](#). Not so long ago, the Johnny-Five framework had a whole new website released that has great documentation on how to use the framework on Arduino and other platforms too!
- [Make: JavaScript Robotics Book](#). A new book released by Rick Waldron and others in the NodeBot community that provides a range of JS projects using various devices. Great for those who’ve got the absolute basics down and want to explore some new projects!
- [NodeBots Official Site](#). Check this page out if you’re looking for a local NodeBots meetup near you, or to read more about NodeBots in general.
- [NodeBots - The Rise of JS Robotics](#). A great post by Chris Williams on how NodeBots came to be. It’s a good read for those interested.

The SimpleBot

Andrew Fisher, a fellow Australian NodeBot enthusiast, put together a rather simple project for people to build for their first NodeBot experience. It’s called a “SimpleBot”, and it lives up to its name. It’s a NodeBot that you can typically build in a single day. If you’re keen on getting an actual robot up and running, rather than just a basic set of sensors and lights going on and off, this is a great project choice to start with. It comes available to Australian attendees of NodeBots Day (see below) in one of the ticket types for this very reason! It’s a bot with wheels and an ultrasonic sensor to detect if it’s about to run into things. Here’s what my own finished version looks like — which I prepared as a sample

for NodeBots Day a few years ago:



A list of SimpleBot materials needed and some sample Node.js code is available at [the SimpleBot GitHub repo](#). Andrew also has a YouTube video showing [how to put the SimpleBot together](#).

Andrew also collaborated with the team at Freetronics to put together a SimpleBot Arduino shield that might also be useful to people who'd like to give it a go as a learning project without needing to solder anything: [SimpleBot Shield Kit](#).

Conclusion

That concludes a simple introduction into the world of NodeBots! If you're interested in getting involved, you've got all the info you should need to begin your NodeBot experience.

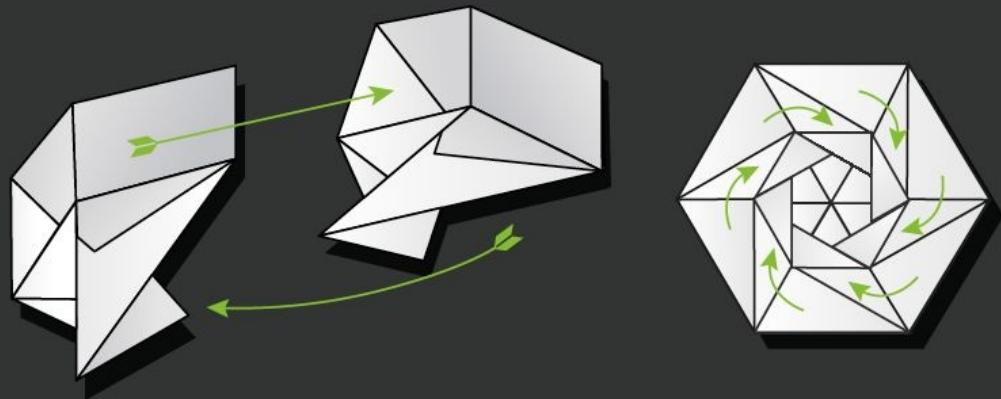
If you want to get more involved with NodeBots, keep an eye out for the annual International NodeBots Day. (It happens around July each year.) It's a day where all sorts of people get together at various events around the world to build JavaScript powered bots and have a great time.

If you build yourself a pretty neat NodeBot with any of the above resources, get in touch with me on Twitter ([@thatpatrickguy](https://twitter.com/thatpatrickguy)), I'd love to check out your JavaScript powered robot!

Book 3: Node.js: Related Tools & Skills



NODE.JS: RELATED TOOLS & SKILLS



LEVEL UP YOUR NODE KNOWLEDGE

Chapter 1: Unit Test Your JavaScript Using Mocha and Chai

by Jani Hartikainen

Have you ever made some changes to your code, and later found it caused something else to break?

I'm sure most of us have. This is almost inevitable, especially when you have a larger amount of code. One thing depends on another, and then changing it breaks something else as a result.

But what if that didn't happen? What if you had a way of knowing when something breaks as a result of some change? That would be pretty great. You could modify your code without having to worry about breaking anything, you'd have fewer bugs and you'd spend less time debugging.

That's where unit tests shine. They will *automatically* detect any problems in the code for you. Make a change, run your tests and if anything breaks, you'll immediately know what happened, where the problem is *and* what the correct behavior should be. This completely eliminates any guesswork!

In this article, I'll show you how to get started unit testing your JavaScript code. The examples and techniques shown in this article can be applied to both browser-based code and Node.js code.

The code for this tutorial is available from our [GitHub repo](#).

What Is Unit Testing

When you test your codebase, you take a piece of code — typically a function — and verify it behaves correctly in a specific situation. Unit testing is a structured and automated way of doing this. As a result, the more tests you write, the bigger the benefit you receive. You will also have a greater level of confidence in your codebase as you continue to develop it.

The core idea with unit testing is to test a function's behavior when giving it a certain set of inputs. You call a function with certain parameters, and check you got the correct result.

```
// Given 1 and 10 as inputs...
var result = Math.max(1, 10);

// ...we should receive 10 as the output
if(result !== 10) {
  throw new Error('Failed');
}
```

In practice, tests can sometimes be more complex. For example, if your function makes an Ajax request, the test needs some more set up, but the same principle of "given certain inputs, we expect a specific outcome" still applies.

Setting up the Tools

For this article, we'll be using Mocha. It's easy to get started with, can be used for both browser-based testing and Node.js testing, and it plays nicely with other testing tools.

The easiest way to install Mocha is through npm (for which we also need to install [Node.js](#)). If you're unsure about how to install either npm or Node on your system, consult our tutorial: [A Beginner's Guide to npm — the Node Package Manager](#)

With Node installed, open up a terminal or command line in your project's directory.

- If you want to test code in the browser, run `npm install mocha chai --save-dev`
- If you want to test Node.js code, in addition to the above, run `npm install -g mocha`

This installs the packages `mocha` and `chai`. [Mocha](#) is the library that allows us to run tests, and [Chai](#) contains some helpful functions that we'll use to verify our test results.

Testing on Node.js vs Testing in the Browser

The examples that follow are designed to work if running the tests in a browser. If you want to unit test your Node.js application, follow these steps.

- For Node, you don't need the test runner file.
- To include Chai, add `var chai = require('chai');` at the top of the test file.
- Run the tests using the `mocha` command, instead of opening a browser.

Setting up a Directory Structure

You should put your tests in a separate directory from your main code files. This makes it easier to structure them, for example if you want to add other types of tests in the future (such as [integration tests](#) or [functional tests](#)).

The most popular practice with JavaScript code is to have a directory called `test/` in your project's root directory. Then, each test file is placed under `test/someModuleTest.js`. Optionally, you can also use directories inside `test/`, but I recommend keeping things simple — you can always change it later if necessary.

Setting up a Test Runner

In order to run our tests in a browser, we need to set up a simple HTML page to be our *test runner* page. The page loads Mocha, the testing libraries and our actual test files. To run the tests, we'll simply open the runner in a browser.

If you're using Node.js, you can skip this step. Node.js unit tests can be run using the command `mocha`, assuming you've followed the recommended directory structure.

Below is the code we'll use for the test runner. I'll save this file as `testrunner.html`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="node_modules/mocha/mocha.css">
  </head>
  <body>
    <div id="mocha"></div>
    <script src="node_modules/mocha/mocha.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd')</script>

    <!-- load code you want to test here -->
    <!-- load your test files here -->

    <script>
      mocha.run();
    </script>
  </body>
</html>
```

The important bits in the test runner are:

- We load Mocha's CSS styles to give our test results nice formatting.
- We create a div with the ID `mocha`. This is where the test results are inserted.
- We load Mocha and Chai. They are located in subfolders of the

`node_modules` folder since we installed them via npm.

- By calling `mocha.setup`, we make Mocha's testing helpers available.
- Then, we load the code we want to test and the test files. We don't have anything here just yet.
- Last, we call `mocha.run` to run the tests. Make sure you call this *after* loading the source and test files.

The Basic Test Building Blocks

Now that we can run tests, let's start writing some.

We'll begin by creating a new file `test/arrayTest.js`. An individual test file such as this one is known as a *test case*. I'm calling it `arrayTest.js` because for this example, we'll be testing some basic array functionality.

Every test case file follows the same basic pattern. First, you have a `describe` block:

```
describe('Array', function() {  
  // Further code for tests goes here  
});
```

`describe` is used to group individual tests. The first parameter should indicate what we're testing — in this case, since we're going to test array functions, I've passed in the string '`Array`'.

Secondly, inside the `describe`, we'll have `it` blocks:

```
describe('Array', function() {  
  it('should start empty', function() {  
    // Test implementation goes here  
  });  
  
  // We can have more its here  
});
```

`it` is used to create the actual tests. The first parameter to `it` should provide a human-readable description of the test. For example, we can read the above as "it should start empty", which is a good description of how arrays should behave. The code to implement the test is then written inside the function passed to `it`.

All Mocha tests are built from these same building blocks, and they follow this same basic pattern.

- First, we use `describe` to say what we're testing - for example, "describe how array should work".
- Then, we use a number of `it` functions to create the individual tests - each

it should explain one specific behavior, such as "it should start empty" for our array case above.

Writing the Test Code

Now that we know how to structure the test case, let's jump into the fun part — implementing the test.

Since we are testing that an array should start empty, we need to create an array and then ensure it's empty. The implementation for this test is quite simple:

```
var assert = chai.assert;

describe('Array', function() {
  it('should start empty', function() {
    var arr = [];

    assert.equal(arr.length, 0);
  });
});
```

Note on the first line, we set up the `assert` variable. This is just so we don't need to keep typing `chai.assert` everywhere.

In the `it` function, we create an array and check its length. Although simple, this is a good example of how tests work.

First, you have something you're testing — this is called the *System Under Test* or *SUT*. Then, if necessary, you do something with the SUT. In this test, we're not doing anything, since we're checking the array starts as empty.

The last thing in a test should be the validation — an *assertion* which checks the result. Here, we are using `assert.equal` to do this. Most assertion functions take parameters in the same order: First the "actual" value, and then the "expected" value.

- The *actual* value is the result from your test code, so in this case `arr.length`
- The *expected* value is what the result *should* be. Since an array should begin empty, the expected value in this test is `0`

Chai also offers two different styles of writing assertions, but we're using [assert](#) to keep things simple for now. When you become more experienced with writing

tests, you might want to use the [expect assertions](#) instead, as they provide some more flexibility.

Running the Test

In order to run this test, we need to add it to the test runner file we created earlier.

If you're using Node.js, you can skip this step, and use the command `mocha` to run the test. You'll see the test results in the terminal.

Otherwise, to add this test to the runner, simply add:

```
<script src="test/arrayTest.js"></script>
```

Below:

```
<!-- load your test files here -->
```

Once you've added the script, you can then load the test runner page in your browser of choice.

The Test Results

When you run your tests, the test results will look something like this:



```
passes: 1  failures: 0  duration: 0.01s  100%
Array
  ✓ should start empty
```

Note that what we entered into the `describe` and `it` functions show up in the output — the tests are grouped under the description. Note that it's also possible to nest `describe` blocks to create further sub-groupings.

Let's take a look at what a failing test looks like.

On the line in the test that says:

```
assert.equal(arr.length, 0);
```

Replace the number `0` with `1`. This makes the test fail, as the array's length no longer matches the expected value.

If you run the tests again, you'll see the failing test in red with a description of what went wrong.

Array

X should start empty

```
AssertionError: 0 == 1  
at Context.<anonymous> (test/arrayTest.js:5:12)
```

One of the benefits of tests is that they help you find bugs quicker, however this error is not very helpful in that respect. We can fix it though.

Most of the assertion functions can also take an optional `message` parameter. This is the message that is displayed when the assertion fails. It's a good idea to use this parameter to make the error message easier to understand.

We can add a message to our assertion like so:

```
assert.equal(arr.length, 1, 'Array length was not 0');
```

If you re-run tests, the custom message will appear instead of the default.

Let's switch the assertion back to the way it was — replace `1` with `0`, and run the tests again to make sure they pass.

Putting It Together

So far we've looked at fairly simple examples. Let's put what we've learned into practice and see how we would test a more realistic piece of code.

Here's a function which adds a CSS class to an element. This should go in a new file `js/className.js`.

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) === -1) {
    el.className += newClass;
  }
}
```

To make it a bit more interesting, I made it add a new class only when that class doesn't exist in an element's `className` property — who wants to see `<div class="hello hello hello hello">` after all?

In the best case, we would write tests for this function *before* we write the code. But [test-driven development](#) is a complex topic, and for now we just want to focus on writing tests.

To get started, let's recall the basic idea behind unit tests: We give the function certain inputs and then verify the function behaves as expected. So what are the inputs and behaviors for this function?

Given an element and a class name:

- if the element's `className` property does not contain the class name, it should be added.
- if the element's `className` property does contain the class name, it should not be added.

Let's translate these cases into two tests. In the `test` directory, create a new file `classNameTest.js` and add the following:

```
describe('addClass', function() {
  it('should add class to element');
  it('should not add a class which already exists');
});
```

We changed the wording slightly to the "it should do X" form used with tests. This means that it reads a bit nicer, but is essentially still the same human-readable form we listed above. It's usually not much more difficult than this to go from idea to test.

But wait, where are the test functions? Well, when we omit the second parameter to `it`, Mocha marks these tests as *pending* in the test results. This is a convenient way to set up a number of tests — kind of like a todo list of what you intend to write.

Let's continue by implementing the first test.

```
describe('addClass', function() {
  it('should add class to element', function() {
    var element = { className: '' };

    addClass(element, 'test-class');

    assert.equal(element.className, 'test-class');
  });

  it('should not add a class which already exists');
});
```

In this test, we create an `element` variable and pass it as a parameter to the `addClass` function, along with a string `test-class` (the new class to add). Then, we check the class is included in the value using an assertion.

Again, we went from our initial idea — given an element and a class name, it should be added into the class list — and translated it into code in a fairly straightforward manner.

Although this function is designed to work with DOM elements, we're using a plain JS object here. Sometimes we can make use of JavaScript's dynamic nature in this fashion to simplify our tests. If we didn't do this, we would need to create an actual element and it would complicate our test code. As an additional benefit, since we don't use DOM, we can also run this test within Node.js if we so wish.

Running the Tests in the Browser

To run the test in the browser, you'll need to add `className.js` and `classNameTest.js` to the runner:

```
<!-- load code you want to test here -->
<script src="js/className.js"></script>

<!-- load your test files here -->
<script src="test/classNameTest.js"></script>
```

You should now see one test pass and another test show up as pending, as is demonstrated by the following CodePen. Note that the code differs slightly from the example in order to make the code work within the CodePen environment.

Live Code!

See the Pen [Unit Testing with Mocha \(1\)](#),

Next, let's implement the second test...

```
it('should not add a class which already exists', function() {
  var element = { className: 'exists' };

  addClass(element, 'exists');

  var numClasses = element.className.split(' ').length;
  assert.equal(numClasses, 1);
});
```

It's a good habit to run your tests often, so let's check what happens if we run the tests now. As expected, they should pass.

Here's another CodePen with the second test implemented.

Live Code!

See the Pen [Unit Testing with Mocha \(2\)](#),

But hang on! I actually tricked you a bit. There is a third behavior for this function which we haven't considered. There is also a bug in the function — a fairly serious one. It's only a three line function but did you notice it?

Let's write one more test for the third behavior which exposes the bug as a bonus.

```
it('should append new class after existing one', function() {
  var element = { className: 'exists' };

  addClass(element, 'new-class');

  var classes = element.className.split(' ');
  assert.equal(classes[1], 'new-class');
});
```

This time the test fails. You can see it in action in the following CodePen. The problem here is simple: CSS class names in elements should be separated by a space. However, our current implementation of `addClass` doesn't add a space!

Live Code!

See the Pen [Unit Testing with Mocha \(3\)](#).

Let's fix the function and make the test pass.

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) !== -1) {
    return;
  }

  if(el.className === '') {
    //ensure class names are separated by a space
    newClass = ' ' + newClass;
  }

  el.className += newClass;
}
```

And here's a final CodePen with the fixed function and passing tests.

Live Code!

See the Pen [Unit Testing with Mocha \(4\)](#).

Running the Tests on Node

In Node, things are only visible to other things in the same file. As `className.js` and `classNameTest.js` are in different files, we need to find a way to expose one to the other. The standard way to do this is through the use of `module.exports`. If you need a refresher, you can read all about that here: [Understanding module.exports and exports in Node.js](#)

The code essentially stays the same, but is structured slightly differently:

```
// className.js

module.exports = {
  addClass: function(el, newClass) {
    if(el.className.indexOf(newClass) !== -1) {
      return;
    }

    if(el.className !== '') {
      //ensure class names are separated by a space
      newClass = ' ' + newClass;
    }

    el.className += newClass;
  }
}
```

```
// classNameTest.js

var chai = require('chai');
var assert = chai.assert;

var className = require('../js(className.js');
var addClass = className.addClass;

// The rest of the file remains the same

describe('addClass', function() {
  ...
});
```

And as you can see, the tests pass.

Terminal

File Edit View Search Terminal Help

jim@friendly ~ \$ mocha

```
Array
  ✓ should start empty

addClass
  ✓ should add class into element
  ✓ should not add a class which already exists in element
  ✓ should append new class after existing one

4 passing (6ms)
```

jim@friendly ~ \$ █

What's Next?

As you can see, testing does not have to be complicated or difficult. Just as with other aspects of writing JavaScript apps, you have some basic patterns which repeat. Once you get familiar with those, you can keep using them again and again.

But this is just scratching the surface. There's a lot more to learn about unit testing.

- Testing more complex systems
- How to deal with Ajax, databases, and other "external" things?
- Test-Driven Development

If you want to continue learning this and more, I've created a [free JavaScript unit testing quickstart series](#). If you found this article useful, you should definitely [check it out here](#).

Alternatively, if video is more your style, you might be interested in SitePoint Premium's course: [Test-Driven Development in Node.js](#).

Chapter 2: An Introduction to Functional JavaScript

by M. David Green

You've heard that JavaScript is a functional language, or at least that it's capable of supporting functional programming. But what is functional programming? And for that matter, if you're going to start comparing programming paradigms in general, how is a functional approach different from the JavaScript that you've always written?

Well, the good news is that JavaScript isn't picky when it comes to paradigms. You can mix your imperative, object-oriented, prototypal, and functional code as you see fit, and still get the job done. But the bad news is what that means for your code. JavaScript can support a wide range of programming styles simultaneously within the same codebase, so it's up to you to make the right choices for maintainability, readability, and performance.

Functional JavaScript doesn't have to take over an entire project in order to add value. Learning a little about the functional approach can help guide some of the decisions you make as you build your projects, regardless of the way you prefer to structure your code. Learning some functional patterns and techniques can put you well on your way to writing cleaner and more elegant JavaScript regardless of your preferred approach.

Imperative JavaScript

JavaScript first gained popularity as an in-browser language, used primarily for adding simple hover and click effects to elements on a web page. For years, that's most of what people knew about it, and that contributed to the bad reputation JavaScript earned early on.

As developers struggled to match the flexibility of JavaScript against the intricacy of the browser document object model (DOM), actual JavaScript code often looked something like this in the real world:

```
var result;
function getText() {
    var someText = prompt("Give me something to capitalize");
    capWords(someText);
    alert(result.join(" "));
};

function capWords(input) {
    var counter;
    var inputArray = input.split(" ");
    var transformed = "";
    result = [];
    for (counter = 0; counter < inputArray.length; counter++) {
        transformed = [
            inputArray[counter].charAt(0).toUpperCase(),
            inputArray[counter].substring(1)
        ].join("");
        result.push(transformed);
    }
};
document.getElementById("main_button").onclick = getText;
```

So many things are going on in this little snippet of code. Variables are being defined on the global scope. Values are being passed around and modified by functions. DOM methods are being mixed with native JavaScript. The function names are not very descriptive, and that's due in part to the fact that the whole thing relies on a context that may or may not exist. But if you happened to run this in a browser inside an HTML document that defined a `<button id="main_button">`, you might get prompted for some text to work with, and then see an alert with first letter of each of the words in that text

capitalized.

Imperative code like this is written to be read and executed from top to bottom (give or take a little [variable hoisting](#)). But there are some improvements we could make to clean it up and make it more readable by taking advantage of JavaScript's object-oriented nature.

Object-Oriented JavaScript

After a few years, developers started to notice the problems with imperative coding in a shared environment like the browser. Global variables from one snippet of JavaScript clobbered global variables set by another. The order in which the code was called affected the results in ways that could be unpredictable, especially given the delays introduced by network connections and rendering times.

Eventually, some better practices emerged to help encapsulate JavaScript code and make it play better with the DOM. An updated variation of the same code above, written to an object-oriented standard, might look something like this:

```
(function() {
  "use strict";
  var SomeText = function(text) {
    this.text = text;
  };
  SomeText.prototype.capify = function(str) {
    var firstLetter = str.charAt(0);
    var remainder = str.substring(1);
    return [firstLetter.toUpperCase(), remainder].join("");
  };
  SomeText.prototype.capifyWords = function() {
    var result = [];
    var textArray = this.text.split(" ");
    for (var counter = 0; counter < textArray.length; counter++) {
      result.push(this.capify(textArray[counter]));
    }
    return result.join(" ");
  };

  document.getElementById("main_button").addEventListener("click", f
    var something = prompt("Give me something to capitalize");
    var newText = new SomeText(something);
    alert(newText.capifyWords());
  });
}());
```

In this object-oriented version, the constructor function simulates a class to model the object we want. Methods live on the new object's prototype to keep

memory use low. And all of the code is isolated in an anonymous immediately-invoked function expression so it doesn't litter the global scope. There's even a "use strict" directive to take advantage of the latest JavaScript engine, and the old-fashioned `onclick` method has been replaced with a shiny new `addEventListener`, because who uses IE8 or earlier anymore? A script like this would likely be inserted at the end of the `<body>` element on an HTML document, to make sure all the DOM had been loaded before it was processed so the `<button>` it relies on would be available.

But despite all this reconfiguration, there are still many artifacts of the same imperative style that led us here. The methods in the constructor function rely on variables that are scoped to the parent object. There's a looping construct for iterating across all the members of the array of strings. There's a counter variable that serves no purpose other than to increment the progress through the `for` loop. And there are methods that produce the side effect of modifying variables that exist outside of their own definitions. All of this makes the code more brittle, less portable, and makes it harder to test the methods outside of this narrow context.

Functional JavaScript

The object-oriented approach is much cleaner and more modular than the imperative approach we started with, but let's see if we can improve it by addressing some of the drawbacks we discussed. It would be great if we could find ways to take advantage of JavaScript's built-in ability to treat functions as first-class objects so that our code could be cleaner, more stable, and easier to repurpose.

```
(function() {
  "use strict";
  var capify = function(str) {
    return [str.charAt(0).toUpperCase(), str.substring(1)].join("");
  };
  var processWords = function(fn, str) {
    return str.split(" ").map(fn).join(" ");
  };
  document.getElementById("main_button").addEventListener("click", f
    var something = prompt("Give me something to capitalize");
    alert(processWords(capify, something));
  });
}());
```

Did you notice how much shorter this version is? We're only defining two functions: `capify` and `processWords`. Each of these functions is pure, meaning that they don't rely on the state of the code they're called from. The functions don't create side effects that alter variables outside of themselves. There is one and only one result a function returns for any given set of arguments. Because of these improvements, the new functions are very easy to test, and could be snipped right out of this code and used elsewhere without any modifications.

There might have been one keyword in there that you wouldn't recognize unless you've peeked at some functional code before. We took advantage of the new [map method](#) on `Array` to apply a function to each element of the temporary array we created when we split our string. `Map` is just one of a handful of convenience methods we were given when modern browsers and server-side JavaScript interpreters implemented the ECMAScript 5 standards. Just using `map` here, in place of a `for` loop, eliminated the counter variable and helped make our code much cleaner and easier to read.

Start Thinking Functionally

You don't have to abandon everything you know to take advantage of the functional paradigm. You can get started thinking about your JavaScript in a functional way by considering a few questions when you write your next program:

- Are my functions dependent on the context in which they are called, or are they pure and independent?
- Can I write these functions in such a way that I could depend on them always returning the same result for a given input?
- Am I sure that my functions don't modify anything outside of themselves?
- If I wanted to use these functions in another program, would I need to make changes to them?

This introduction barely scratches the surface of functional JavaScript, but I hope it whets your appetite to learn more.

Chapter 3: An Introduction to Gulp.js

by Craig Buckler

Developers spend precious little time coding. Even if we ignore irritating meetings, much of the job involves basic tasks which can sap your working day:

- generating HTML from templates and content files
- compressing new and modified images
- compiling Sass to CSS code
- removing console and debugger statements from scripts
- transpiling ES6 to cross-browser-compatible ES5 code
- code linting and validation
- concatenating and minifying CSS and JavaScript files
- deploying files to development, staging and production servers.

Tasks must be repeated every time you make a change. You may start with good intentions, but the most infallible developer will forget to compress an image or two. Over time, pre-production tasks become increasingly arduous and time-consuming; you'll dread the inevitable content and template changes. It's mind-numbing, repetitive work. Wouldn't it be better to spend your time on more profitable jobs?

If so, you need a *task runner* or *build process*.

That Sounds Scarily Complicated!

Creating a build process will take time. It's more complex than performing each task manually, but over the long term, you'll save hours of effort, reduce human error and save your sanity. Adopt a pragmatic approach:

- Automate the most frustrating tasks first.
- Try not to over-complicate your build process. An hour or two is more than enough for the initial setup.
- Choose task runner software and stick with it for a while. Don't switch to another option on a whim.

Some of the tools and concepts may be new to you, but take a deep breath and concentrate on one thing at a time.

Task Runners: the Options

Build tools such as [GNU Make](#) have been available for decades, but web-specific task runners are a relatively new phenomenon. The first to achieve critical mass was [Grunt](#) — a Node.js task runner which used plugins controlled (originally) by a JSON configuration file. Grunt was hugely successful, but there were a number of issues:

1. Grunt required plugins for basic functionality such as file watching.
2. Grunt plugins often performed multiple tasks, which made customisation more awkward.
3. JSON configuration could become unwieldy for all but the most basic tasks.
4. Tasks could run slowly because Grunt saved files between every processing step.

Many issues were addressed in later editions, but [Gulp](#) had already arrived and offered a number of improvements:

1. Features such as file watching were built in.
2. Gulp plugins were (*mostly*) designed to do a single job.
3. Gulp used JavaScript configuration code that was less verbose, easier to read, simpler to modify, and provided better flexibility.
4. Gulp was faster because it uses [Node.js streams](#) to pass data through a series of piped plugins. Files were only written at the end of the task.

Of course, Gulp itself isn't perfect, and new task runners such as [Broccoli.js](#), [Brunch](#) and [webpack](#) have also been competing for developer attention. More recently, [npm itself has been touted as a simpler option](#). All have their pros and cons, but [Gulp remains the favorite and is currently used by more than 40% of web developers](#).

Gulp requires Node.js, but while some JavaScript knowledge is beneficial, developers from all web programming faiths will find it useful.

What About Gulp 4?

This tutorial describes how to use Gulp 3 — the most recent release version at the time of writing. Gulp 4 has been in development for some time but remains a beta product. It's possible to [use or switch to Gulp 4](#), but I recommend sticking with version 3 until the final release.

Step 1: Install Node.js

Node.js can be downloaded for Windows, macOS and Linux from nodejs.org/download/. There are various options for installing from binaries, package managers and docker images, and full instructions are available.

For Windows Users

Node.js and Gulp run on Windows, but some plugins may not install or run if they depend on native Linux binaries such as image compression libraries. One option for Windows 10 users is the new [bash command-line](#), which solves many issues.

Once installed, open a command prompt and enter:

```
node -v
```

This reveals the version number. You're about to make heavy use of `npm` — the Node.js package manager which is used to install modules. Examine its version number:

```
npm -v
```

For Linux Users

Node.js modules can be installed globally so they're available throughout your system. However, most users will not have permission to write to the global directories unless `npm` commands are prefixed with `sudo`. There are a number of [options to fix npm permissions](#) and tools such as [nvm can help](#), but I often change the default directory. For example, on Ubuntu/Debian-based platforms:

```
cd ~
    mkdir .node_modules_global
    npm config set prefix=$HOME/.node_modules_global
    npm install npm -g
```

Then add the following line to the end of `~/.bashrc`:

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

Finally, update with this:

```
source ~/.bashrc
```

Step 2: Install Gulp Globally

Install Gulp command-line interface globally so the `gulp` command can be run from any project folder:

```
npm install gulp-cli -g
```

Verify Gulp has installed with this:

```
gulp -v
```

Step 3: Configure Your Project

For Node.js Projects

You can skip this step if you already have a package.json configuration file.

Note for Node.js projects: you can skip this step if you already have a package.json configuration file.

Presume you have a new or pre-existing project in the folder project1. Navigate to this folder and initialize it with npm:

```
cd project1  
npm init
```

You'll be asked a series of questions. Enter a value or hit **Return** to accept defaults. A package.json file will be created on completion which stores your npm configuration settings.

Git Users

Node.js installs modules to a node_modules folder. You should add this to your .gitignore file to ensure they're not committed to your repository. When deploying the project to another PC, you can run npm install to restore them.

For the remainder of this article, we'll presume your project folder contains the following sub-folders:

src folder: preprocessed source files

This contains further sub-folders:

- **html** - HTML source files and templates
- **images** — the original uncompressed images
- **js** — multiple preprocessed script files
- **scss** — multiple preprocessed Sass .scss files

build folder: compiled/processed files

Gulp will create files and create sub-folders as necessary:

- `html` — compiled static HTML files
- `images` — compressed images
- `js` — a single concatenated and minified JavaScript file
- `css` — a single compiled and minified CSS file

Your project will almost certainly be different but this structure is used for the examples below.

Following Along on Unix

Tip: If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,css}
```

Tip: If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,css}
```

Step 4: Install Gulp Locally

You can now install Gulp in your project folder using the command:

```
npm install gulp --save-dev
```

This installs Gulp as a development dependency and the "devDependencies" section of package.json is updated accordingly. We'll presume Gulp and all plugins are development dependencies for the remainder of this tutorial.

Alternative Deployment Options

Development dependencies are not installed when the NODE_ENV environment variable is set to production on your operating system. You would normally do this on your live server with the Mac/Linux command:

```
export NODE_ENV=production
```

Or on Windows:

```
set NODE_ENV=production
```

This tutorial presumes your assets will be compiled to the build folder and committed to your Git repository or uploaded directly to the server. However, it may be preferable to build assets on the live server if you want to change the way they are created. For example, HTML, CSS and JavaScript files are minified on production but not development environments. In that case, use the --save option for Gulp and all plugins, i.e.

```
npm install gulp --save
```

This sets Gulp as an application dependency in the "dependencies" section of package.json. It will be installed when you enter npm install and can be run wherever the project is deployed. You can remove the build folder from your repository since the files can be created on any platform when required.

Step 4: Create a Gulp Configuration File

Create a new `gulpfile.js` configuration file in the root of your project folder. Add some basic code to get started:

```
// Gulp.js configuration
var
  // modules
  gulp = require('gulp'),

  // development mode?
  devBuild = (process.env.NODE_ENV !== 'production'),

  // folders
  folder = {
    src: 'src/',
    build: 'build/'
  }
;
```

This references the Gulp module, sets a `devBuild` variable to `true` when running in development (or non-production mode) and defines the source and build folder locations.

ES6

ES5-compatible JavaScript code is provided in this tutorial. This will work for all versions of Gulp and Node.js with or without the `--harmony` flag. [Most ES6 features are supported in Node 6 and above](#) so feel free to use arrow functions, `let`, `const`, etc. if you're using a recent version.

`gulpfile.js` won't do anything yet because you need to ...

Step 5: Create Gulp Tasks

On its own, Gulp does nothing. You must:

1. install Gulp plugins, and
2. write tasks which utilize those plugins to do something useful.

It's possible to write your own plugins but, since almost 3,000 are available, it's unlikely you'll ever need to. You can search using Gulp's own directory at gulpjs.com/plugins/, on npmjs.com, or search "gulp *something*" to harness the mighty power of Google.

Gulp provides three primary task methods:

- `gulp.task` — defines a new task with a name, optional array of dependencies and a function.
- `gulp.src` — sets the folder where source files are located.
- `gulp.dest` — sets the destination folder where build files will be placed.

Any number of plugin calls are set with pipe between the `.src` and `.dest`.

Image Task

This is best demonstrated with an example, so let's create a basic task which compresses images and copies them to the appropriate build folder. Since this process could take time, we'll only compress new and modified files. Two plugins can help us: [gulp-newer](#) and [gulp-imagemin](#). Install them from the command-line:

```
npm install gulp-newer gulp-imagemin --save-dev
```

We can now reference both modules the top of `gulpfile.js`:

```
// Gulp.js configuration

var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
```

```
imagemin = require('gulp-imagemin'),
```

We can now define the image processing task itself as a function at the end of `gulpfile.js`:

```
// image processing
gulp.task('images', function() {
  var out = folder.build + 'images/';
  return gulp.src(folder.src + 'images/**/*')
    .pipe(newer(out))
    .pipe(imagemin({ optimizationLevel: 5 }))
    .pipe(gulp.dest(out));
});
```

All tasks are syntactically similar. This code:

1. Creates a new task named `images`.
2. Defines a function with a return value which ...
3. Defines an `out` folder where build files will be located.
4. Sets the Gulp `src` source folder. The `/**/*` ensures that images in sub-folders are also processed.
5. Pipes all files to the `gulp-newer` module. Source files that are newer than corresponding destination files are passed through. Everything else is removed.
6. The remaining new or changed files are piped through `gulp-imagemin` which sets an optional `optimizationLevel` argument.
7. The compressed images are output to the Gulp `dest` folder set by `out`.

Save `gulpfile.js` and place a few images in your project's `src/images` folder before running the task from the command line:

```
gulp images
```

All images are compressed accordingly and you'll see output such as:

```
Using file gulpfile.js
Running 'imagemin'...
Finished 'imagemin' in 5.71 ms
gulp-imagemin: image1.png (saved 48.7 kB)
gulp-imagemin: image2.jpg (saved 36.2 kB)
gulp-imagemin: image3.svg (saved 12.8 kB)
```

Try running `gulp images` again and nothing should happen because no newer

images exist.

HTML Task

We can now create a similar task which copies files from the source HTML folder. We can safely minify our HTML code to remove unnecessary whitespace and attributes using the [gulp-htmlclean](#) plugin:

```
npm install gulp-htmlclean --save-dev
```

This is then referenced at the top of `gulpfile.js`:

```
var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
  htmlclean = require('gulp-htmlclean'),
```

We can now create an `html` task at the end of `gulpfile.js`:

```
// HTML processing
gulp.task('html', ['images'], function() {
  var
    out = folder.build + 'html/',
    page = gulp.src(folder.src + 'html/**/*')
      .pipe(newer(out));

  // minify production code
  if (!devBuild) {
    page = page.pipe(htmlclean());
  }

  return page.pipe(gulp.dest(out));
});
```

This reuses `gulp-newer` and introduces a couple of concepts:

1. The `[images]` argument states that our `images` task must be run before processing the HTML (the HTML is likely to reference images). Any number of dependent tasks can be listed in this array and all will complete before the task function runs.
2. We only pipe the HTML through `gulp-htmlclean` if `NODE_ENV` is set to

production. Therefore, the HTML remains uncompressed during development which may be useful for debugging.

Save `gulpfile.js` and run `gulp html` from the command line. Both the `html` and `images` tasks will run.

JavaScript Task

Too easy for you? Let's process all our JavaScript files by building a basic module bundler. It will:

1. Ensure dependencies are loaded first using the [gulp-deporder](#) plugin. This analyses comments at the top of each script to ensure correct ordering. For example, `// requires: defaults.js lib.js`.
2. Concatenate all script files into a single `main.js` file using [gulp-concat](#).
3. Remove all `console` and debugging statements with [gulp-strip-debug](#) and minimize code with [gulp-uglify](#). This step will only occur when running in production mode.

Install the plugins:

```
npm install gulp-deporder gulp-concat gulp-strip-debug gulp-uglify -
```

Reference them at the top of `gulpfile.js`:

```
var  
  ...  
concat = require('gulp-concat'),  
deporder = require('gulp-deporder'),  
stripdebug = require('gulp-strip-debug'),  
uglify = require('gulp-uglify'),
```

Then add a new js task:

```
// JavaScript processing  
gulp.task('js', function() {  
  
  var jsbuild = gulp.src(folder.src + 'js/**/*')  
    .pipe(deporder())  
    .pipe(concat('main.js'));  
  
  if (!devBuild) {
```

```

        jsbuild = jsbuild
          .pipe(stripdebug())
          .pipe(uglify());
    }

    return jsbuild.pipe(gulp.dest(folder.build + 'js/'));
});


```

Save then run `gulp js` to watch the magic happen!

CSS Task

Finally, let's create a CSS task which compiles Sass `.scss` files to a single `.css` file using [gulp-sass](#). This is a Gulp plugin for [node-sass](#) which binds to the superfast [LibSass C/C++ port of the Sass engine](#) (*you won't need to install Ruby*). We'll presume your primary Sass file `scss/main.scss` is responsible for loading all partials.

Our task will also utilize the fabulous [PostCSS](#) via the [gulp-postcss](#) plugin. PostCSS requires its own set of plugins and we'll install these:

- [postcss-assets](#) to manage assets. This allows us to use properties such as `background: resolve('image.png');` to resolve file paths or `background: inline('image.png');` to inline data-encoded images.
- [autoprefixer](#) to automatically add vendor prefixes to CSS properties.
- [css-mqpacker](#) to pack multiple references to the same CSS media query into a single rule.
- [cssnano](#) to minify the CSS code when running in production mode.

First, install all the modules:

```
npm install gulp-sass gulp-postcss postcss-assets autoprefixer css-mqpacker cssnano
```

Then reference them at the top of `gulpfile.js`:

```

var
  ...
  sass = require('gulp-sass'),
  postcss = require('gulp-postcss'),
  assets = require('postcss-assets'),
  autoprefixer = require('autoprefixer'),

```

```
mqpacker = require('css-mqpacker'),  
cssnano = require('cssnano'),
```

We can now create a new css task at the end of `gulpfile.js`. Note the `images` task is set as a dependency because the `postcss-assets` plugin can reference images during the build process. In addition, most plugins can be passed arguments (refer to their documentation for more information):

```
// CSS processing  
gulp.task('css', ['images'], function() {  
  
  var postCssOpts = [  
    assets({ loadPaths: ['images/] }),  
    autoprefixer({ browsers: ['last 2 versions', '> 2%'] }),  
    mqpacker  
  ];  
  
  if (!devBuild) {  
    postCssOpts.push(cssnano);  
  }  
  
  return gulp.src(folder.src + 'scss/main.scss')  
    .pipe(sass({  
      outputStyle: 'nested',  
      imagePath: 'images/',  
      precision: 3,  
      errLogToConsole: true  
    }))  
    .pipe(postcss(postCssOpts))  
    .pipe(gulp.dest(folder.build + 'css/'));  
  
});
```

Save the file and run the task from the command line:

```
gulp css
```

Step 6: Automate Tasks

We've been running one task at a time. We can run them all in one command by adding a new `run` task to `gulpfile.js`:

```
// run all tasks
gulp.task('run', ['html', 'css', 'js']);
```

Save and enter `gulp run` at the command line to execute all tasks. Note that I omitted the `images` task because it's already set as a dependency for the `html` and `css` tasks.

Is this still too much hard work? Gulp offers another method — `gulp.watch` — which can monitor your source files and run an appropriate task whenever a file is changed. The method is passed a folder and a list of tasks to execute when a change occurs. Let's create a new `watch` task at the end of `gulpfile.js`:

```
// watch for changes
gulp.task('watch', function() {

    // image changes
    gulp.watch(folder.src + 'images/**/*', ['images']);

    // html changes
    gulp.watch(folder.src + 'html/**/*', ['html']);

    // javascript changes
    gulp.watch(folder.src + 'js/**/*', ['js']);

    // css changes
    gulp.watch(folder.src + 'scss/**/*', ['css']);

});
```

Rather than running `gulp watch` immediately, let's add a default task:

```
// default task
gulp.task('default', ['run', 'watch']);
```

Save `gulpfile.js` and enter `gulp` at the command line. Your images, HTML, CSS and JavaScript will all be processed, then Gulp will remain active watching for updates and re-running tasks as necessary. Hit `Ctrl/Cmd + C` to abort

monitoring and return to the command line.

Step 7: Profit!

Other plugins you may find useful:

- [gulp-load-plugins](#) ± load all Gulp plugin modules without require declarations
- [gulp-preprocess](#) — a simple HTML and JavaScript [preprocessor](#)
- [gulp-less](#) — the [Less CSS preprocessor](#) plugin
- [gulp-stylus](#) — the [Stylus CSS preprocessor](#) plugin
- [gulp-sequence](#) — run a series of gulp tasks in a specific order
- [gulp-plumber](#) — error handling which prevents Gulp stopping on failures
- [gulp-size](#) — displays file sizes and savings
- [gulp-nodemon](#) — uses [nodemon](#) to automatically restart Node.js applications when changes occur.
- [gulp-util](#) — utility functions including logging and color coding.

One useful method in `gulp-util` is `.noop()` which passes data straight through without performing any action. This could be used for cleaner development/production processing code. For example:

```
var gutil = require('gulp-util');

// HTML processing
gulp.task('html', ['images'], function() {
  var out = folder.src + 'html/**/*';

  return gulp.src(folder.src + 'html/**/*')
    .pipe(newer(out))
    .pipe(devBuild ? gutil.noop() : htmlclean())
    .pipe(gulp.dest(out));

});
```

Gulp can also call other Node.js modules, and they don't necessarily need to be plugins. For example:

- [browser-sync](#) — automatically reload assets or refresh your browser when changes occur
- [del](#) — delete files and folders (perhaps clean your build folder at the start of every run).

Invest a little time and Gulp could save many hours of development frustration.
The advantages:

- [plugins are plentiful](#)
- configuration using pipes is readable and easy to follow
- `gulpfile.js` can be adapted and reused in other projects
- your total page weight can be reduced to improve performance
- you can simplify your deployment.

Useful links:

- [Gulp home page](#)
- [Gulp plugins](#)
- [npm home page](#)

Applying the processes above to a simple website reduced the total weight by more than 50%. You can test your own results using [page weight analysis tools](#) or a service such as [New Relic](#), which provides a range of sophisticated application performance monitoring tools.

Gulp can revolutionize your workflow. I hope you found this tutorial useful and consider Gulp for your production process.

Chapter 4: A Side-by-side Comparison of Express, Koa and Hapi.js

by Olayinka Omole

If you're a Node.js developer, chances are you have, at some point, used Express.js to create your applications or APIs. Express.js is a very popular Node.js framework, and even has some other frameworks built on top of it such as [Sails.js](#), [kraken.js](#), [KeystoneJS](#) and [many others](#). However, amidst this popularity, a bunch of other frameworks have been gaining attention in the JavaScript world, such as Koa and hapi.

In this article, we'll examine Express.js, Koa and hapi.js — their similarities, differences and use cases.

Background

Let's firstly introduce each of these frameworks separately.

Express.js

[Express.js](#) is described as the standard server framework for Node.js. It was created by [TJ Holowaychuk](#), acquired by StrongLoop in 2014, and is currently maintained by the [Node.js Foundation incubator](#). With about [170+ million downloads in the last year](#), it's currently beyond doubt that it's the most popular Node.js framework.

Koa

Development began on [Koa](#) in late 2013 by the same guys at Express. It's referred to as the future of Express. Koa is also described as a much more modern, modular and minimalistic version of the Express framework.

Hapi.js

[Hapi.js](#) was developed by the team at Walmart Labs (led by [Eran Hammer](#)) after they tried Express and discovered that it didn't work for their requirements. It was originally developed on top of Express, but as time went by, it grew into a full-fledged framework.

Fun Fact: hapi is short for Http API server.

Philosophy

Now that we have some background on the frameworks and how they were created, let's compare each of them based on important concepts, such as their philosophy, routing, and so on.

Compatibility

All code examples are in ES6 and make use of version 4 of Express.js, 2.4 of Koa, and 17 for hapi.js.

Express.js

Express was built to be a simple, unopinionated web framework. From its [GitHub README](#):

The Express philosophy is to provide small, robust tooling for HTTP servers, making it a great solution for single page applications, web sites, hybrids, or public HTTP APIs.

Express.js is minimal and doesn't possess many features out of the box. It doesn't force things like file structure, ORM or templating engine.

Koa

While Express.js is minimal, Koa can boast a much more minimalistic code footprint —around 2k LOC. Its aim is to allow developers be even more expressive. Like Express.js, it can easily be extended by using existing or custom plugins and middleware. It's more futuristic in its approach, in that it relies heavily on the relatively new JavaScript features like [generators](#) and [async/await](#).

Hapi.js

Hapi.js focuses more on configuration and provides a lot more features out of the box than Koa and Express.js. [Eran Hammer](#), one of the creators of hapi, described the reason for building the framework properly in [his blog post](#):

hapi was created around the idea that configuration is better than code, that business logic must be isolated from the transport layer, and that native node constructs like buffers and stream should be supported as first class objects.

Starting a Server

Starting a server is one of the basic things we'd need to do in our projects. Let's examine how it can be done in the different frameworks. We'll start a server and listen on port 3000 in each example.

Express.js

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

app.listen(3000, () => console.log('App is listening on port 3000!'))
```

Starting a server in Express.js is as simple as requiring the `express` package, initializing the `express` app to the `app` variable and calling the `app.listen()` method, which is just a wrapper around the native Node.js [http.createServer\(\)](#) method.

Koa

Starting a server in Koa is quite similar to Express.js:

```
const Koa = require('koa');
const app = new Koa();

app.listen(3000, () => console.log('App is listening on port 3000!'))
```

The `app.listen()` method in Koa is also a wrapper around the `http.createServer()` method.

Hapi.js

Starting a server in hapi.js is quite a departure from what many of us may be used to from Express:

```
const Hapi = require('hapi');

const server = Hapi.server({
  host: 'localhost',
  port: 3000
```

```
});

async function start() {
  try {
    await server.start();
  }
  catch (err) {
    console.log(err);
    process.exit(1);
  }
  console.log('Server running at:', server.info.uri);
};

start();
```

In the code block above, first we require the `hapi` package, then instantiate a server with `Hapi.server()`, which has a single config object argument containing the host and port parameters. Then we start the server with the asynchronous `server.start()` function.

Unlike in Express.js and Koa, the `server.start()` function in hapi is not a wrapper around the native `http.createServer()` method. It instead implements its own custom logic.

The above code example is from the [hapi.js](#) website, and shows the importance the creators of hapi.js place on configuration and error handling.

Routing

Routing is another key aspect of modern web applications. Let's define a /hello route for a simple Hello World app in each framework to have a feel of how routing works for them.

Express.js

```
app.get('/hello', (req, res) => res.send('Hello World!'));
```

Creating routes in Express is as simple as calling the app object with the required HTTP method. The syntax is `app.METHOD(PATH, HANDLER)`, where PATH is the path on the server and HANDLER is function which is called when the path is matched.

Koa

Koa doesn't have its own router bundled with it, so we'll have to use a router middleware to handle routing on Koa apps. Two common routing options are [koa-route](#) and [koa-router](#). Here's an example using koa-route:

```
const route = require('koa-route');

app.use(route.get('/hello', ctx => {
  ctx.body = 'Hello World!';
}));
```

We can see immediately that Koa needs each route to be defined as a middleware on the app. The ctx is a context object that contains Node's request and response objects. `ctx.body` is a method in the response object and can be used to set the response body to either a string, Buffer, Stream, Object or null. The second parameter for the route method can be an async or generator function, so the use of callbacks is reduced.

Hapi.js

```
server.route({
  method: 'GET',
  path: '/hello',
```

```
handler: function (request, h) {
  return 'Hello world!';
}
});
```

The `server.route()` method in hapi takes a single config object with the following parameters: `method`, `path` and `handler`. You can see the documentation on routing in hapi [here](#).

The `request` parameter in the handler function is an object which contains the user's request details, while the `h` parameter is described as a response toolkit.

Middleware

One of the major concepts Node developers are used to is working with middleware. **Middleware functions** are functions that sit in between requests and responses. They have access to the request and response objects and can run the next middleware after they're processed. Let's take a look at how they're defined in the different frameworks by implementing a simple function that logs the time a request is made to the server.

Express.js

```
app.use((req, res, next) => {
  console.log(`Time: ${Date.now()}`);
  next();
})
```

Registering middleware in Express.js is as simple as binding the middleware to the [app object](#) by using the `app.use()` function. You can read more on middleware in Express.js [here](#).

Koa

```
app.use(async (ctx, next) => {
  console.log(`Time: ${Date.now()}`);
  await next();
});
```

Middleware registration in Koa is similar to Express.js. The major differences are that the [context object](#) (`ctx`) is used in place of the `request` and `response` objects in Express.js and Koa embraces the modern `async/await` paradigm for defining the middleware function.

Hapi.js

```
server.ext('onRequest', (request, h) => {
  console.log(`Time: ${Date.now()}`);
  return h.continue;
});
```

In hapi.js there are certain extension points in the [request lifecycle](#). The `server.ext()` method registers an extension function to be called at a certain point in the request life cycle. You can read more about it [here](#). We make use of the `onRequest` extension point in the example above to register a middleware (or extension) function.

Usage

From the comparisons and code examples we've seen above, it's clear that Express and Koa are the most similar, with hapi.js being the framework to deviate from the norm that Node.js devs are used to. Hence hapi.js may not be the best choice when trying to build a quick and easy app, as it will take a bit of time to get used to.

In my opinion, Express is still a great choice when building small- to medium-sized applications. It can become a bit complicated to manage for very large applications, as it doesn't possess the modularity hapi.js has built into it, with support for [custom plugins](#) and its unique [routing method](#). However, there's been some speculation in recent times regarding the future of Express.js [as TJ announced he's no longer working on it](#) and the [reduced rate at which updates are shipped](#). Bit it's pretty stable and willn't be going away any time soon. It also has a large community of developers building various extensions and plugins for it.

Like Express.js, Koa is well suited for many simple Node.js projects. It only consists of the bare minimum (it has zero in-built middleware) and encourages developers to add what they need to it by building or making use of available external middleware. It makes use of modern JavaScript generator functions and `async/await` heavily, which makes it sort of futuristic in its approach. Its [middleware cascading pattern](#) is also great, as it makes implementing and understanding the flow of middleware in your applications very easy. Koa probably won't be a great choice for you if you aren't yet ready to embrace new shiny things like generator functions, or if you're not willing to spend some time building out all the middleware you need. The community support for Koa is rapidly growing, as it has a good amount of external middleware already built for it (some by the core Koa team) for common tasks such as routing, logging and so on.

Hapi.js is the definite choice if you and your team prefer to spend more time configuring than actually coding out features. It was built to be modular and for large applications with large teams. It encourages the micro-service architecture, as various parts of your app can be built as plugins and registered in your server before starting it up. Hapi.js is backed by large companies such as Auth0 and Lob, so it has a pretty good future ahead of it and won't be going

away anytime soon. It's also trusted by some big names, as seen on [their community page](#).

Hapi.js has a whole lot more features out of the box than Koa and Express.js, such as support for authentication, caching, logging, validation and so on, which makes it feel more like a full-fledged framework. You can check out their [tutorials page](#) to get a good feel of the features they provide. There aren't yet very many open-source projects and plugins built on and for hapi.js, so a lot of work might need to be done by developers using it if they plan to extend its core functionality.

Conclusion

All three frameworks are great choices when starting up new projects, but ultimately your choice will be based on the project requirements, your team members and the level of flexibility you're looking for.

Chapter 5: An Introduction to Sails.js

by Ahmed Bouchefra

[Sails.js](#) is a Node.js MVC (model–view–controller) framework that follows the “convention over configuration” principle. It’s inspired by the popular Ruby on Rails web framework, and allows you to quickly build REST APIs, single-page apps and real-time (WebSockets-based) apps. It makes extensive use of code generators that allow you to build your application with less writing of code—particularly of common code that can be otherwise scaffolded.

The framework is built on top of Express.js, one of the most popular Node.js libraries, and Socket.io, a JavaScript library/engine for adding real-time, bidirectional, event-based communication to applications. At the time of writing, the official stable version of Sails.js is *0.12.14*, which is available from [npm](#). Sails.js version 1.0 has not officially been released, but [according to Sails.js creators](#), version **1.0** is already used in some production applications, and they even recommend using it when starting new projects.

Main Features

Sails.js has many great features:

- it's built on Express.js
- it has real-time support with WebSockets
- it takes a “convention over configuration” approach
- it has powerful code generation, thanks to Blueprints
- it's database agnostic thanks to its powerful Waterline ORM/ODM
- it supports multiple data stores in the same project
- it has good documentation.

There are currently a few important cons, such as:

- no support for JOIN query in Waterline
- no support for SQL transactions until Sails v1.0 (in beta at the time of writing)
- until version 1.0, it still uses Express.js v3, which is EOL (end of life)
- development is very slow.

Sails.js vs Express.js

Software development is all about building abstractions. Sails.js is a high-level abstraction layer on top of Express.js (which itself is an abstraction over Node's HTTP modules) that provides routing, middleware, file serving and so on. It also adds a powerful ORM/ODM, the MVC architectural pattern, and a powerful generator CLI (among other features).

You can build web applications using Node's low-level HTTP service and other utility modules (such as the filesystem module) but it's not recommended except for the sake of learning the Node.js platform. You can also take a step up and use Express.js, which is a popular, lightweight framework for building web apps.

You'll have routing and other useful constructs for web apps, but you'll need to take care of pretty much everything from configuration, file structure and code organization to working with databases.

Express doesn't offer any built-in tool to help you with database access, so you'll need to bring together the required technologies to build a complete web application. This is what's called a stack. Web developers, using JavaScript, mostly use the popular [MEAN stack](#), which stands for MongoDB, ExpressJS, AngularJS and Node.js.

MongoDB is the preferred database system among Node/Express developers, but you can use any database you want. The most important point here is that Express doesn't provide any built-in APIs when it comes to databases.

The Waterline ORM/ODM

One key feature of Sails.js is Waterline, a powerful ORM (object relational mapper) for SQL-based databases and ODM (object document mapper) for NoSQL document-based databases. Waterline abstracts away all the complexities when working with databases and, most importantly, with Waterline you don't have to make the decision of choosing a database system when you're just starting development. It also doesn't intimidate you when your client hasn't yet decided on the database technology to use.

You can start building your application without a single line of configuration. In fact, you don't have to install a database system at all initially. Thanks to the built-in `sails-disk` NeDB-based file database, you can transparently use the file system to store and retrieve data for testing your application functionality.

Once you're ready and you have decided on the convenient database system you want to use for your project, you can then simply switch the database by installing the relevant adapter for your database system. Waterline has [official adapters](#) for popular relational database systems such as MySQL and PostgreSQL and the NoSQL databases, such as MongoDB and Redis, and the community has also built numerous adapters for the other popular database systems such as Oracle, MSSQL, DB2, SQLite, CouchDB and neo4j. In case when you can't find an adapter for the database system you want to use, you can [develop your own custom adapter](#).

Waterline abstracts away the differences between different database systems and allows you to have a normalized interface for your application to communicate with any supported database system. You don't have to work with SQL or any low-level API (for NoSQL databases) but that doesn't mean you can't (at least for SQL-based databases and MongoDB).

There are situations when you need to write custom SQL, for example, for performance reasons, for working with complex database requirements, or for accessing database-specific features. In this case, you can use the `.query()` method available only on the Waterline models that are configured to use SQL systems (you can find more information about `query()` from the [docs](#)).

Since different database systems have common and database-specific features,

the Waterline ORM/ODM can only be good for you as long as you only constrain yourself to use the common features. Also, if you use raw SQL or native MongoDB APIs, you'll lose many of the features of Waterline, including the ability to switch between different databases.

Getting Started with Sails.js

Now that we've covered the basic concepts and features of Sails.js, let's see how you can quickly get started using Sails.js to create new projects and lift them.

Prerequisites

Before you can use Sails.js, you need to have a development environment with Node.js (and npm) installed. You can install both of them by heading to the [official Node.js website](#) and downloading the right installer for your operating system.

Downloads

Latest LTS Version: 8.9.4 (includes npm 5.6.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features
 Windows Installer <small>node-v8.9.4-x86.msi</small>	 macOS Installer <small>node-v8.9.4.pkg</small>
	 Source Code <small>node-v8.9.4.tar.gz</small>
Windows Installer (.msi)	32-bit 64-bit
Windows Binary (.zip)	32-bit 64-bit
macOS Installer (.pkg)	64-bit
macOS Binaries (.tar.gz)	64-bit
Linux Binaries (x86/x64)	32-bit 64-bit
Linux Binaries (ARM)	ARMv6 ARMv7 ARMv8
Source Code	node-v8.9.4.tar.gz

Additional Platforms

SunOS Binaries
Docker Image
Linux on Power Systems
Linux on System z
AIX on Power Systems

32-bit	64-bit
Official Node.js Docker Image	
64-bit le	
64-bit	
64-bit	

- [Signed SHASUMS for release files](#)

Make sure, also, to install whatever database management system you want to use with Sails.js (either a relational or a NoSQL database). If you're not interested by using a full-fledged database system, at this point, you can still work with Sails.js thanks to `sails-disk`, which allows you to have a file-based database out of the box.

Installing the Sails.js CLI

After satisfying the working development requirements, you can head over to your terminal (Linux and macOS) or command prompt (Windows) and install the Sails.js Command Line Utility, globally, from npm:

```
sudo npm install sails -g
```

If you want to install the latest *1.0* version to try the new features, you need to use the beta version:

```
npm install sails@beta -g
```

You may or may not need *sudo* to install packages globally depending on your [npm configuration](#).

Scaffolding a Sails.js Project

After installing the Sails.js CLI, you can go ahead and scaffold a new project with one command:

```
sails new sailsdemo
```

This will create a new folder for your project named *sailsdemo* on your current directory. You can also scaffold your project files inside an existing folder with this:

```
sails new .
```

You can scaffold a new Sails.js project without a front end with this:

```
sails new sailsdemo --no-frontend
```

Find more information about the features of the CLI from the [docs](#).

The Anatomy of a Sails.js Project

Here's a screenshot of a project generated using the Sails.js CLI:

```
▲ sailsdemo
  ▲ .tmp
    ▶ public
    ≡ localDiskDb.db
  ▲ api
    ▶ controllers
    ▶ models
    ▶ policies
    ▶ responses
    ▶ services
  ▶ assets
  ▶ config
  ▶ node_modules
  ▶ tasks
  ▶ views
  ⚙ .editorconfig
  ⚙ .gitignore
  ≡ .sailsrc
  JS app.js
  📄 Gruntfile.js
  {} package-lock.json
  {} package.json
  ⓘ README.md
```

A Sails.js project is a Node.js module with a `package.json` and a `node_modules` folder. You may also notice the presence of `Gruntfile.js`. Sails.js uses Grunt as a build tool for building front-end assets.

If you're building an app for the browser, you're in luck. Sails ships with Grunt — which means your entire front-end asset workflow is completely customizable, and comes with support for all of the great Grunt modules which are already out there. That includes support for Less, Sass, Stylus, CoffeeScript, JST, Jade, Handlebars, Dust, and many more. When you're ready to go into production, your assets are minified and gzipped automatically. You can even compile your static assets and push them out to a CDN like CloudFront to make your app load even faster. (You can read more about these points [on the Sails.js website](#).)

You can also use Gulp or Webpack as your build system instead of Grunt, with custom generators. See the [`sails-generate-new-gulp`](#) and [`sails-webpack`](#) projects

on GitHub.

For more community generators, see [this documentation page](#) on the Sails.js site.

The project contains many configuration files and folders. Most of them are self explanatory, but let's go over the ones you'll be working with most of the time:

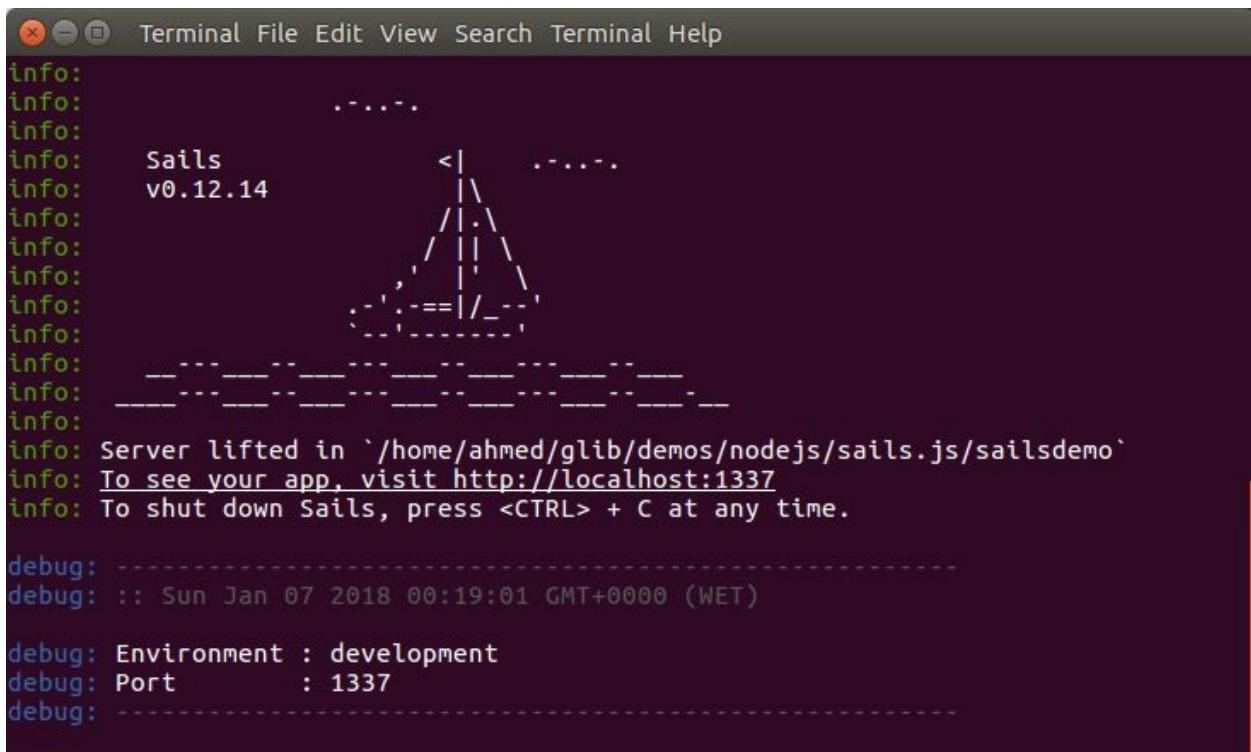
- `api/controllers`: this is the folder where controllers live. Controllers correspond to the *C* part in *MVC*. It's where the business logic for your application exists.
- `api/models`: the folder where models exist. Models correspond to the *M* part of *MVC* architecture. This is where you need to put classes or objects that map to your SQL/NoSQL data.
- `api/policies`: this is the folder where you need to put policies for your application
- `api/responses`: this folder contains server response logic such as functions to handle the 404 and 500 responses, etc.
- `api/services`: this where your app-wide services live. A service is a global class encapsulating common logic that can be used throughout many controllers.
- `./views`: this folder contains templates used for displaying views. By default, this folder contains the `ejs` engine templates, but you can configure any Express-supported engine such as EJS, Jade, Handlebars, Mustache and Underscore etc.
- `./config`: this folder contains many configuration files that enable you to configure every detail of your application, such as CORS, CSRF protection, i18n, http, settings for models, views, logging and policies etc. One important file that you'll frequently use is `config/routes.js`, where you can create your application routes and map them to actual actions in the controllers or to views directly.
- `./assets`: this is the folder where you can place any static files (CSS, JavaScript and images etc.) for your application.

Running your Sails.js Project

You can start the development server by running the following command from your project's root:

```
sails lift
```

This will prompt you to choose a migration strategy, and then will launch the dev server.



A screenshot of a terminal window titled "Terminal". The window shows the startup logs for a Sails.js application. The logs include:

```
info: .....  
info: Sails      <|      ..  
info: v0.12.14  
info: .....  
info: ;' ;==| /_--'  
info: -----  
info: Server lifted in `/home/ahmed/glib/demos/nodejs/sails.js/sailsdemo`  
info: To see your app, visit http://localhost:1337  
info: To shut down Sails, press <CTRL> + C at any time.  
  
debug: -----  
debug: :: Sun Jan 07 2018 00:19:01 GMT+0000 (WET)  
  
debug: Environment : development  
debug: Port       : 1337  
debug: -----
```

You can then use your web browser to navigate to [<http://localhost:1337/>] (<http://localhost:1337/>). If you've generated a Sails.js project with a front end (i.e without using the `--no-frontend` option) you'll see this home page:

A brand new app.

You're looking at: `views/homepage.ejs`

1 Generate a REST API.

Run `sails generate api user`. This will create two files: a [model](#) and a [controller](#).

2 Lift your app.

Run `sails lift` to start up your app server. If you visit <http://localhost:1337/user> in your browser, you'll see a [WebSocket-compatible user API](#).

3 Dive in.

Blueprints are just the beginning. You'll probably also want to learn how to customize your app's [routes](#), set up [security policies](#), configure your [data sources](#), and build custom [controller actions](#). For more help getting started, check out the links on this page.

Docs

[App Structure](#)

[Reference](#)

[Supported Databases](#)

Tutorials

[Sails 101](#)

Community

[StackOverFlow](#)

[GitHub](#)

[Google Group](#)

[IRC \(#sailsjs on freenode\)](#)

Creating Waterline Models

A **model** is an abstraction, usually represented by an object or a class in a general-purpose programming language, and it refers/maps either to an SQL table in a relational database or a document (or key-value pairs) in a NoSQL database.

You can create models using the Sails.js CLI:

```
sails generate model product
```

This will create a `Product.js` model in `api/models` with the following content:

```
/**  
 * Product.js  
 *  
 * @description :: TODO: You might write a short summary of how this  
 * @docs          :: http://sailsjs.org/documentation/concepts/models-
```

```
/*
module.exports = {

  attributes: {

  }
};
```

You can then augment your model with attributes. For example:

```
module.exports = {

  attributes: {
    name: {
      type: 'string',
      defaultsTo: '',
      required: 'true'
    },
    description: {
      type: 'string',
      defaultsTo: ''
    },
    quantity: {
      type: 'integer'
    },
    user: { model: 'User' }
  }
};
```

Notice how we can define the association (one-to-many or belongsTo relationship) with the model *User*. You can see all supported associations and how to create them via this [Sails.js associations page](#).

For more information about the available model attributes, see the [Sails.js attributes page](#).

You can also add configurations per model or model settings by adding top-level properties in the model definition, which will override the global models settings in config/models.js. You can override settings related to the model's attributes, database connections etc.

For example, let's specify a different datastore for the *product* model other than the global one used throughout the project:

```
module.exports = {
  connection: 'mysqlcon'
  attributes: { /*...*/}
}
```

This will instruct Sails.js to use a connection named *mysqlcon* to store this model data. Make sure you add the *mysqlcon* connection to the *connections* object in config/connections.js:

```
module.exports.connections = {
  // sails-disk is installed by default.
  localDiskDb: {
    adapter: 'sails-disk'
  },
  mysqlcon: {
    adapter: 'sails-mysql',
    host: 'YOUR_MYSQL_HOST',
    user: 'YOUR_MYSQL_USER',
    password: 'YOUR_MYSQL_PASSWORD',
    database: 'YOUR_MYSQL_DB'
  }
};
```

You also need to install the *sails-mysql* adapter from npm:

```
npm install sails-mysql@for-sails-0.12
```

You can find the available model settings that you can specify from the [Sails.js model settings page](#).

Sails.js Controllers

Controllers hold your app's business logic. They live in api/controllers and provide a layer which glues your app's models and views. Controllers contain actions that are bound to routes and respond to HTTP requests from web/mobile clients.

A controller is a JavaScript object that contains methods called the *controller actions*, which take two parameters: a request and a response.

You can find more information about controllers on the [Sails.js controllers page](#).

You can generate a controller using the Sails.js CLI:

```
sails generate controller product
```

This command will generate a controller named `api/controllers/ProductController.js`, with the following content:

```
/**  
 * ProductController  
 *  
 * @description :: Server-side logic for managing products  
 * @help          :: See http://sailsjs.org/#!/documentation/concepts/  
 */  
  
module.exports = {  
  
};
```

The code exports an empty JavaScript object where you can add new actions or override the default (automatically added) controller actions.

At this point, you can actually execute CRUD operations against your server without further adding any code. Since Sails.js follows convention over configuration, it wires your controllers to their corresponding routes and provides default actions for handling the common HTTP POST, GET, PUT and DELETE requests etc.

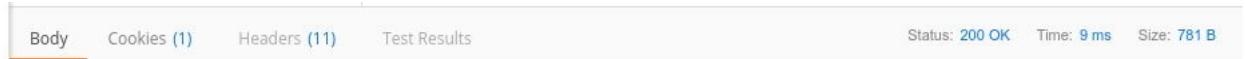
Testing with Postman

Using Postman, you can send POST, GET and other requests to test your API, so go ahead and [grab the Postman version for your operating system](#). Next, enter the product endpoint URL `http://localhost:1337/product`. Then choose the HTTP method to send — POST in this case, because we want to create a Product. Next, you need to provide data, so click on the *Body tab*, select the *Raw* option, then enter the following:

```
{  
  "name": "Product 1",  
  "description": "This is product 1",  
  "quantity": 100  
}
```

Then hit the Send button.

You should pay attention to the returned status code: 200 OK means the product was successfully created.

A screenshot of the Postman application interface. The 'Body' tab is selected. The request method is set to 'POST' and the URL is 'http://localhost:1337/product'. The 'Body' section contains the following JSON data:

```
1 [
2   {
3     "name": "Product 1",
4     "description": "This is product 1",
5     "quantity": 100
6   }
7 ]
```

A screenshot of the Postman application interface. The 'Body' tab is selected. The status bar at the top shows 'Status: 201 Created', 'Time: 33 ms', and 'Size: 598 B'. The response body is displayed in 'Pretty' format:

```
1 [
2   {
3     "name": "Product 1",
4     "description": "This is product 1",
5     "quantity": 100,
6     "createdAt": "2018-01-07T13:54:15.753Z",
7     "updatedAt": "2018-01-07T13:54:15.753Z",
8     "id": 3
9   }
10 ]
```

You can also update a product by its id by sending a PUT request:

The screenshot shows the Postman application interface. At the top, there are tabs for 'New Tab' and 'No Environment'. Below the header, the method is set to 'PUT' and the URL is 'http://localhost:1337/product/1'. There are buttons for 'Params', 'Send', and 'Save'. The 'Body' tab is selected, showing the following JSON payload:

```
1 {  
2   "name": "Product 100",  
3   "description": "This is product 100",  
4   "quantity": 1000  
5 }
```

Below the body, the 'Body' tab is selected again, showing the response from the server:

```
1 {  
2   "name": "Product 100",  
3   "description": "This is product 100",  
4   "createdAt": "2018-01-07T13:10:58.642Z",  
5   "updatedAt": "2018-01-07T14:22:05.693Z",  
6   "id": 1,  
7   "quantity": 1000  
8 }
```

The status bar at the bottom indicates 'Status: 200 OK', 'Time: 30 ms', and 'Size: 568 B'.

Finally, you can delete a product by its id by sending a DELETE request:

The screenshot shows the Postman application interface. At the top, there are tabs for 'Authorization', 'Headers', 'Body' (which is selected), 'Pre-request Script', and 'Tests'. Below the tabs, there's a note about inheritance of authorization. The main area displays a JSON response with three products:

```

1 [
2   {
3     "name": "Product 100",
4     "description": "This is product 100",
5     "createdAt": "2018-01-07T13:10:58.642Z",
6     "updatedAt": "2018-01-07T14:22:05.693Z",
7     "id": 1,
8     "quantity": 1000
9   },
10  {
11    "name": "Product 1",
12    "description": "This is product 1",
13    "quantity": 100,
14    "createdAt": "2018-01-07T13:12:06.336Z",
15    "updatedAt": "2018-01-07T14:21:12.707Z",
16    "id": 2
17  },
18  {
19    "name": "Product 1",
20    "description": "This is product 1",
21    "quantity": 100,
22    "createdAt": "2018-01-07T13:54:15.753Z",
23    "updatedAt": "2018-01-07T13:54:15.753Z",
24    "id": 3
25  }
26 ]

```

For custom logic, you can also override these actions and implement your own.

When you create an API (i.e. a controller and a model) Sails.js automatically adds eight default actions, which are:

- add to
- create
- destroy
- find one
- find where
- populate where
- remove from
- update

`Find` where and `find one`, `create`, `update` and `destroy` are normal CRUD actions that needs to be present in most APIs. The others are related to foreign records:

- `add to:` used to add a foreign record to another record collection (e.g a product to a user's products).
- `populate where:` used to populate and return foreign record(s) for the given association of another record. [Read more information here](#).
- `remove from:` used to remove a foreign record (e.g. a product) from one of a related record collection association (e.g. user's products). See [more information and examples here](#).

For customizing the behavior of the default actions, you can do either of these:

- Override the action in a specified controller. That is, create an action with the same name as one of these actions: `find`, `findOne`, `create`, `update`, `destroy`, `populate`, `add` or `remove`.
- Override the default action for all controllers. You can do this by creating an `api/blueprints` folder, where you need to add files with lowercase names for a default action (e.g. `find.js`, `findone.js`, `create.js`, etc.). You can find more information about blueprints by checking the [Sails.js Blueprint API docs](#).

Routing in Sails.js

[Routes](#) allow you to map URLs to controllers or views. Just like the default actions, Sails.js automatically adds default routes for the default actions, allowing you to have an automatic API by just creating models and controllers.

You can also add custom routes for your custom actions or views. To add a route, open the `config/routes.js` file and add this:

```
module.exports.routes = {  
  '/products': {  
    view: 'products'  
  }  
};
```

This maps `/products` to the template named `products` in views folder.

You can optionally add an HTTP verb to the URL. For example:

```
module.exports.routes = {
  'get /': {
    view: 'homepage'
  }
};
```

You can also specify a controller action for a route. For example:

```
module.exports.routes = {
  'post /product': 'ProductController.create',
};
```

This tells Sails.js to call the *create* action of the *ProductController* controller when a client sends a POST request to the /product endpoint.

Conclusion

In this chapter, you were introduced to Sails.js. We looked at the basic concepts of Sails.js, and how to generate a new Sails.js project, and then created an API by just generating models, adding some attributes then generate controllers. Sails.js has other advanced concepts such as [services](#), [policies](#), [blueprints](#) and [hooks](#). These you can further discover on your own, once you grasp and get familiar with the basic concepts in this introduction.

Chapter 6: Building Apps and Services with the Hapi.js Framework

by Mark Brown

[Hapi.js](#) is described as “a rich framework for building applications and services”. Hapi’s smart defaults make it a breeze to create JSON APIs, and its modular design and plugin system allow you to easily extend or modify its behavior.

The recent release of [version 17.0](#) has fully embraced `async` and `await`, so you’ll be writing code that appears synchronous but is non-blocking *and* avoids callback hell. Win-win.

The Project

In this article, we'll be building the following API for a typical blog from scratch:

```
# RESTful actions for fetching, creating, updating and deleting articles
GET    /articles                         articles#index
GET    /articles/:id                      articles#show
POST   /articles                          articles#create
PUT    /articles/:id                     articles#update
DELETE /articles/:id                    articles#destroy

# Nested routes for creating and deleting comments
POST   /articles/:id/comments          comments#create
DELETE /articles/:id/comments          comments#destroy

# Authentication with JSON Web Tokens (JWT)
POST   /authentications                 authentications#create
```

The article will cover:

- Hapi's core API: routing, request and response
- models and persistence in a relational database
- routes and actions for Articles and Comments
- testing a REST API with HTTPie
- authentication with JWT and securing routes
- validation
- an HTML View and Layout for the root route /.

The Starting Point

Make sure you've got a recent version of Node.js installed; `node -v` should return `8.9.0` or higher.

Download the starting code from here with git:

```
git clone https://github.com/markbrown4/hapi-api.git  
cd hapi-api  
npm install
```

Open up `package.json` and you'll see that the "start" script runs `server.js` with `nodemon`. This will take care of restarting the server for us when we change a file.

Run `npm start` and open `http://localhost:3000/`:

```
[{ "so": "hapi!" }]
```

Let's look at the source:

```
// server.js  
const Hapi = require('hapi')  
  
// Configure the server instance  
const server = Hapi.server({  
  host: 'localhost',  
  port: 3000  
})  
  
// Add routes  
server.route({  
  method: 'GET',  
  path: '/',  
  handler: () => {  
    return [{ so: 'hapi!' }]  
  }
})  
  
// Go!  
server.start().then(() => {  
  console.log('Server running at:', server.info.uri)
}).catch(err => {
```

```
    console.log(err)
    process.exit(1)
})
```

The Route Handler

The route handler is the most interesting part of this code. Replace it with the code below, comment out the return lines one by one, and test the response in your browser.

```
server.route({
  method: 'GET',
  path: '/',
  handler: () => {
    // return [{ so: 'hapi!' }]
    return 123
    return `<h1><marquee>HTML <em>rules!</em></h1>`
    return null
    return new Error('Boom')
    return Promise.resolve({ whoa: true })
    return require('fs').createReadStream('index.html')
  }
})
```

To send a response, you simply `return` a value and Hapi will send the appropriate body and headers.

- An Object will respond with stringified JSON and Content-Type: `application/json`
- String values will be Content-Type: `text/html`
- You can also return a Promise or Stream.

The handler function is often made `async` for cleaner control flow with Promises:

```
server.route({
  method: 'GET',
  path: '/',
  handler: async () => {
    let html = await Promise.resolve(`<h1>Google</h1>`)
    html = html.replace('Google', 'Hapi')

    return html
  }
})
```

It's not *always* cleaner with `async` though. Sometimes returning a Promise is simpler:

```
handler: () => {
  return Promise.resolve(`<h1>Google</h1>`)
    .then(html => html.replace('Google', 'Hapi'))
}
```

We'll see better examples of how `async` helps us out when we start interacting with the database.

The Model Layer

Like the popular [Express.js](#) framework, Hapi is a minimal framework that doesn't provide any recommendations for the Model layer or persistence. You can choose any database and ORM that you'd like, or none — it's up to you. We'll be using [SQLite](#) and the [Sequelize ORM](#) in this tutorial to provide a clean API for interacting with the database.

SQLite comes pre-installed on macOS and most Linux distributions. You can check if it's installed with `sqlite -v`. If not, you can find installation instructions at the [SQLite website](#).

Sequelize works with many popular relational databases like Postgres or MySQL, so you'll need to install both `sequelize` and the `sqlite3` adapter:

```
npm install --save sequelize sqlite3
```

Let's connect to our database and write our first table definition for `articles`:

```
// models.js
const path = require('path')
const Sequelize = require('sequelize')

// configure connection to db host, user, pass - not required for SQ
const sequelize = new Sequelize(null, null, null, {
  dialect: 'sqlite',
  storage: path.join('tmp', 'db.sqlite') // SQLite persists its data
})

// Here we define our Article model with a title attribute of type s
const Article = sequelize.define('article', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT
})

// Create table
Article.sync()

module.exports = {
  Article
}
```

Let's test out our new model by importing it and replacing our route handler with the following:

```
// server.js
const { Article } = require('./models')

server.route({
  method: 'GET',
  path: '/',
  handler: () => {
    // try commenting these lines out one at a time
    return Article.findAll()
    return Article.create({ title: 'Welcome to my blog', body: 'The
    return Article.findById(1)
    return Article.update({ title: 'Learning Hapi', body: `JSON API`})
    return Article.findAll()
    return Article.destroy({ where: { id: 1 } })
    return Article.findAll()
  }
})
```

If you're familiar with SQL or other ORM's, the [Sequelize API](#) should be self explanatory, It's built with Promises so it works great with Hapi's async handlers too.

Note: using Article.sync() to create the tables or Article.sync({ force: true }) to drop and create are fine for the purposes of this demo. If you're wanting to use this in production you should check out [sequelize-cli](#) and write [Migrations](#) for any schema changes.

Our RESTful Actions

Let's build the following routes:

```
GET      /articles           fetch all articles
GET      /articles/:id        fetch article by id
POST     /articles           create article with '{ title, body }` param
PUT      /articles/:id        update article with '{ title, body }` param
DELETE   /articles/:id        delete article by id
```

Add a new file, `routes.js`, to separate the server config from the application logic:

```
// routes.js
const { Article } = require('./models')

exports.configureRoutes = (server) => {
  // server.route accepts an object or an array
  return server.route([
    {
      method: 'GET',
      path: '/articles',
      handler: () => {
        return Article.findAll()
      }
    },
    {
      method: 'GET',
      // The curly braces are how we define params (variable path segments)
      path: '/articles/{id}',
      handler: (request) => {
        return Article.findById(request.params.id)
      }
    },
    {
      method: 'POST',
      path: '/articles',
      handler: (request) => {
        const article = Article.build(request.payload.article)

        return article.save()
      }
    },
    {
      // method can be an array
      method: ['PUT', 'PATCH'],
      path: '/articles/{id}',
      handler: async (request) => {
```

```

        const article = await Article.findById(request.params.id)
        article.update(request.payload.article)

        return article.save()
    }
}, {
    method: 'DELETE',
    path: '/articles/{id}',
    handler: async (request) => {
        const article = await Article.findById(request.params.id)

        return article.destroy()
    }
}])
}

```

Import and configure our routes before we start the server:

```

// server.js
const Hapi = require('hapi')
const { configureRoutes } = require('./routes')

const server = Hapi.server({
    host: 'localhost',
    port: 3000
})

// This function will allow us to easily extend it later
const main = async () => {
    await configureRoutes(server)
    await server.start()

    return server
}

main().then(server => {
    console.log('Server running at:', server.info.uri)
}).catch(err => {
    console.log(err)
    process.exit(1)
})

```

Testing Our API Is as Easy as HTTPie

[HTTPie](#) is great little command-line HTTP client that works on all operating systems. Follow the installation instructions in the [documentation](#) and then try hitting the API from the terminal:

```
http GET http://localhost:3000/articles
http POST http://localhost:3000/articles article='{"title": "Welcome to my blog!"}'
http POST http://localhost:3000/articles article='{"title": "Learn how to build a RESTful API with Node.js and Express"}'
http GET http://localhost:3000/articles
http GET http://localhost:3000/articles/2
http PUT http://localhost:3000/articles/2 article='{"title": "True beginners guide to Node.js"}'
http GET http://localhost:3000/articles/2
http DELETE http://localhost:3000/articles/2
http GET http://localhost:3000/articles
```

Okay, everything seems to be working well. Let's try a few more:

```
http GET http://localhost:3000/articles/12345
http DELETE http://localhost:3000/articles/12345
```

Yikes! When we try to fetch an article that doesn't exist, we get a 200 with an empty body and our destroy handler throws an Error which results in a 500. This is happening because `findById` returns `null` by default when it can't find a record. We want our API to respond with a 404 in both of these cases. There's a few ways we can achieve this.

Defensively Check for `null` Values and Return an Error

There's a package called `boom` which helps create standard error response objects:

```
npm install --save boom
```

Import it and modify `GET /articles/:id` route:

```
// routes.js
const Boom = require('boom')

{
  method: 'GET',
```

```
path: '/articles/{id}',  
handler: async (request) => {  
  const article = await Article.findById(request.params.id)  
  if (article === null) return Boom.notFound()  
  
  return article  
}  
}
```

Extend Sequelize.Model to throw an Error

`Sequelize.Model` is a reference to the prototype that all of our Models inherit from, so we can easily add a new method `find` to `findById` and throw an error if it returns null:

```
// models.js  
const Boom = require('boom')  
  
Sequelize.Model.find = async function (...args) {  
  const obj = await this.findById(...args)  
  if (obj === null) throw Boom.notFound()  
  
  return obj  
}
```

We can then revert the handler to its former glory and replace occurrences of `findById` with `find`:

```
{  
  method: 'GET',  
  path: '/articles/{id}',  
  handler: (request) => {  
    return Article.find(request.params.id)  
  }  
}
```

```
http GET http://localhost:3000/articles/12345  
http DELETE http://localhost:3000/articles/12345
```

Boom. We now get a *404 Not Found* error whenever we try to fetch something from the database that doesn't exist. We've replaced our custom error checks with an easy-to-understand convention that keeps our code clean.

Note: another popular tool for making requests to REST APIs is [Postman](#). If you

prefer a UI and ability to save common requests, this is a great option.

Path Parameters

The routing in Hapi is a little different from other frameworks. The route is selected on the *specificity* of the path, so the order you define them in doesn't matter.

- `/hello/{name}` matches `/hello/bob` and passes 'bob' as the *name* param
- `/hello/{name?}` — the ? makes name optional and matches both `/hello` and `/hello/bob`
- `/hello/{name*2}` — the * denotes multiple segments, matching `/hello/bob/marley` by passing 'bob/marley' as the *name* param
- `/{args*}` matches `/any/route/imaginable` and has the lowest specificity.

The Request Object

The [request object](#) that's passed to the route handler has the following useful properties:

- `request.params` — path params
- `request.query` — query string params
- `request.payload` — request body for JSON or form params
- `request.state` — cookies
- `request.headers`
- `request.url`

Adding a Second Model

Our second model will handle comments on articles. Here's the complete file:

```
// models.js
const path = require('path')
const Sequelize = require('sequelize')
const Boom = require('boom')

Sequelize.Model.find = async function (...args) {
  const obj = await this.findById(...args)
  if (obj === null) throw Boom.notFound()

  return obj
}

const sequelize = new Sequelize(null, null, null, {
  dialect: 'sqlite',
  storage: path.join('tmp', 'db.sqlite')
})

const Article = sequelize.define('article', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT
})

const Comment = sequelize.define('comment', {
  commenter: Sequelize.STRING,
  body: Sequelize.TEXT
})

// These associations add an articleId foreign key to our comments table
// They add helpful methods like article.getComments() and article.comments()
ArticlehasMany(Comment)
Comment.belongsTo(Article)

// Create tables
Article.sync()
Comment.sync()

module.exports = {
  Article,
  Comment
}
```

For creating and deleting comments we can add nested routes under the article's path:

```
// routes.js
const { Article, Comment } = require('./models')

{
  method: 'POST',
  path: '/articles/{id}/comments',
  handler: async (request) => {
    const article = await Article.find(request.params.id)

    return article.createComment(request.payload.comment)
  }
}, {
  method: 'DELETE',
  path: '/articles/{articleId}/comments/{id}',
  handler: async (request) => {
    const { id, articleId } = request.params
    // You can pass options to findById as a second argument
    const comment = await Comment.find(id, { where: { articleId } })

    return comment.destroy()
  }
}
```

Lastly, we can extend `GET /articles/:id` to return both the article *and* its comments:

```
{
  method: 'GET',
  path: '/articles/{id}',
  handler: async (request) => {
    const article = await Article.find(request.params.id)
    const comments = await article.getComments()

    return { ...article.get(), comments }
  }
}
```

`article` here is the *Model* object; `article.get()` returns a plain object with the model's values, on which we can use the spread operator to combine with our comments. Let's test it out:

```
http POST http://localhost:3000/articles/3/comments comment:='{"com
```

```
http POST http://localhost:3000/articles/3/comments comment:='{"comment": "Great article!"}'  
http GET http://localhost:3000/articles/3  
http DELETE http://localhost:3000/articles/3/comments/2  
http GET http://localhost:3000/articles/3
```

Our blog API is almost ready to ship to production, just needing a couple of finishing touches.

Authentication with JWT

[JSON Web Tokens](#) are a common authentication mechanism for APIs. There's a plugin `hapi-auth-jwt2` for setting it up, but it hasn't yet been updated for Hapi 17.0, so we'll need to install a fork for now:

```
npm install --save salzhrani/hapi-auth-jwt2#v-17
```

The code below registers the `hapi-auth-jwt2` plugin and sets up a *strategy* named `admin` using the `jwt` *scheme*. If a valid JWT token is sent in a header, query string or cookie, it will call our `validate` function to verify that we're happy to grant those credentials access:

```
// auth.js
const jwtPlugin = require('hapi-auth-jwt2').plugin
// This would be in an environment variable in production
const JWT_KEY = 'NeverShareYourSecret'

var validate = function (credentials) {
  // Run any checks here to confirm we want to grant these credentials
  return {
    isValid: true,
    credentials // request.auth.credentials
  }
}

exports.configureAuth = async (server) => {
  await server.register(jwtPlugin)
  server.auth.strategy('admin', 'jwt', {
    key: JWT_KEY,
    validate,
    verifyOptions: { algorithms: [ 'HS256' ] }
  })

  // Default all routes to require JWT and opt out for public routes
  server.auth.default('admin')
}
```

Next, import and configure our auth strategy before starting the server:

```
// server.js
const { configureAuth } = require('./auth')
```

```
const main = async () => {
  await configureAuth(server)
  await configureRoutes(server)
  await server.start()

  return server
}
```

Now all routes will require our admin auth strategy. Try these three:

```
http GET localhost:3000/articles
http GET localhost:3000/articles Authorization:yep
http GET localhost:3000/articles Authorization:eyJ0eXAiOiJKV1QiLCJhb
```

The last one should contain a valid token and return the articles from the database. To make a route public, we just need to add `config: { auth: false }` to the route object. For example:

```
{
  method: 'GET',
  path: '/articles',
  handler: (request) => {
    return Article.findAll()
  },
  config: { auth: false }
}
```

Make these three routes public so that anyone can read articles and post comments:

GET	/articles	articles#index
GET	/articles/:id	articles#show
POST	/articles/:id/comments	comments#create

Generating a JWT

There's a package named `jsonwebtoken` for signing and verifying JWT:

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

Our final route will take an email / password and generate a JWT. Let's define our login function in `auth.js` to keep all auth logic in a single place:

```
// auth.js
const jwt = require('jsonwebtoken')
const Boom = require('boom')

exports.login = (email, password) => {
  if (!(email === 'mb4@gmail.com' && password === 'bears'))

    const credentials = { email }
    const token = jwt.sign(credentials, JWT_KEY, { algorithm: 'HS256' },
      return { token }
}

// routes.js
const { login } = require('./auth')

{
  method: 'POST',
  path: '/authentications',
  handler: async (request) => {
    const { email, password } = request.payload.login

    return login(email, password)
  },
  config: { auth: false }
}

http POST localhost:3000/authentications login:='{"email": "mb4@gmai
```

Try using the returned token in your requests to the secure routes!

Validation with joi

You can validate request params by adding config to the route object. The code below ensures that the submitted article has a body and title between three and ten characters in length. If a validation fails, Hapi will respond with a 400 error:

```
const Joi = require('joi')

{
  method: 'POST',
  path: '/articles',
  handler: (request) => {
    const article = Article.build(request.payload.article)

    return article.save()
  },
  config: {
    validate: {
      payload: {
        article: {
          title: Joi.string().min(3).max(10),
          body: Joi.string().required()
        }
      }
    }
  }
}
```

In addition to payload, you can also add validations to path, query and headers. Learn more about validation in the [docs](#).

Who Is Consuming this API?

We could serve a single-page app from `/`. We've already seen — at the start of the tutorial — one example of how to serve an HTML file with streams. There are much better ways of working with Views and Layouts in Hapi, though. See [Serving Static Content](#) and [Views and Layouts](#) for more on how to render dynamic views:

```
{  
  method: 'GET',  
  path: '/',  
  handler: () => {  
    return require('fs').createReadStream('index.html')  
  },  
  config: { auth: false }  
}
```

If the front end and API are on the same domain, you'll have no problems making requests: `client -> hapi-api`.

If you're serving the front end from a *different* domain and want to make requests to the API directly from the client, you'll need to enable CORS. This is super easy in Hapi:

```
const server = Hapi.server({  
  host: 'localhost',  
  port: 3000,  
  routes: {  
    cors: {  
      credentials: true  
      // See options at https://hapijs.com/api/17.0.0#-routeoptions  
    }  
  }  
)
```

You could also create a *new* application in between the two. If you go down this route, you won't need to bother with CORS, as the client will only be making requests to the front-end app, and it can then make requests to the API on the server without any cross-domain restrictions: `client -> hapi-front-end -> hapi-api`.

Whether that front end is another Hapi application, or Next, or Nuxt ... I'll leave that for you to decide!

Chapter 7: Create New Express.js Apps in Minutes with Express Generator

by Paul Sauve

[Express.js](#) is a Node.js web framework that has gained immense popularity due to its simplicity. It has easy-to-use routing and simple support for view engines, putting it far ahead of the basic Node HTTP server.

However, starting a new Express application requires a certain amount of boilerplate code: starting a new server instance, configuring a view engine, setting up error handling.

Although there are various starter projects and boilerplates available, Express has its own command-line tool that makes it easy to start new apps, called the [express-generator](#).

What is Express?

Express has a lot of features built in, and a lot more features you can get from other packages that integrate seamlessly, but there are three main things it does for you out of the box:

1. **Routing.** This is how `/home` `/blog` and `/about` all give you different pages. Express makes it easy for you to modularize this code by allowing you to put different routes in different files.
2. **Middleware.** If you're new to the term, basically middleware is "software glue". It accesses requests before your routes get them, allowing them to handle hard-to-do stuff like cookie parsing, file uploads, errors, and more.
3. **Views.** Views are how HTML pages are rendered with custom content. You pass in the data you want to be rendered and Express will render it with your given view engine.

Getting Started

Starting a new project with the Express generator is as simple as running a few commands:

```
npm install express-generator -g
```

This installs the Express generator as a global package, allowing you to run the `express` command in your terminal:

```
express myapp
```

This creates a new Express project called `myapp` which is then placed inside of the `myapp` directory.

```
cd myapp
```

If you're unfamiliar with terminal commands, this one puts you inside of the `myapp` directory.

```
npm install
```

npm is the default Node.js package manager. Running `npm install` installs all dependencies for the project. By default, the `express-generator` includes several packages that are commonly used with an Express server.

Options

The generator CLI takes half a dozen [arguments](#), but the two most useful ones are the following:

- **-v ejss|hbs|hjs|jade|pug|twig|vash.** This lets you select a view engine to install. The default is `jade`. Although this still works, it has been deprecated in favor of `pug`.
- **-c less|stylus|compass|sass.** By default, the generator creates a very basic CSS file for you, but selecting a CSS engine will configure your new app with middleware to compile any of the above options.

Now that we've got our project set up and dependencies installed, we can start

the new server by running the following:

```
npm start
```

Then browse to <http://localhost:3000> in your browser.

Application Structure

The generated Express application starts off with four folders.

bin

The `bin` folder contains the executable file that starts your app. It starts the server (on port 3000, if no alternative is supplied) and sets up some basic error handling. You don't really need to worry about this file because `npm start` will run it for you.

public

The `public` folder is one of the important ones: *everything* in this folder is accessible to people connecting to your application. In this folder, you'll put JavaScript, CSS, images, and other assets that people need when they connect to your website.

routes

The `routes` folder is where you'll put your router files. The generator creates two files, `index.js` and `users.js`, which serve as examples of how to separate out your application's route configuration.

Usually, here you'll have a different file for each major route on your website. So you might have files called `blog.js`, `home.js`, and/or `about.js` in this folder.

views

The `views` folder is where you have the files used by your templating engine. The generator will configure Express to look in here for a matching view when you call the `render` method.

Outside of these folders, there's one file that you should know well.

app.js

The `app.js` file is special because it sets up your Express application and glues all of the different parts together. Let's walk through what it does. Here's how the file starts:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
```

These first six lines of the file are requires. If you're new to Node, be sure to read [Understanding module.exports and exports in Node.js](#).

```
var routes = require('./routes/index');
var users = require('./routes/users');
```

The next two lines of code are requiring the different route files that the Express generator sets up by default: routes and users.

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

After that, we create a new app by calling `express()`. The `app` variable contains all of the settings and routes for your application. This object glues together your application.

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

Once the `app` instance is created, the templating engine is set up for rendering views. This is where you'd change the path to your view files if you wanted.

After this, you'll see Express being configured to use middleware. The generator installs several common pieces of middleware that you're likely to use in a web application.

```
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
```

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

- **favicon.** This one is pretty self-explanatory: it just serves your `favicon.ico` out of your public directory.
- **logger.** When you run your app, you might notice that all the routes that are requested are logged to console. If you want to disable this, you can just comment out this middleware.
- **bodyParser.** You might notice that there are two lines for parsing the body of incoming HTTP requests. The first line handles when JSON is sent via POST request and it puts this data in `request.body`. The second line parses query string data in the URL (e.g. `/profile?id=5`) and puts this in `request.query`.
- **cookieParser.** This takes all the cookies the client sends and puts them in `request.cookies`. It also allows you to modify them before sending them back to the client, by changing `response.cookies`.
- **express.static.** This middleware serves static assets from your `public` folder. If you wanted to rename or move the public folder, you can change the path here.

Next up is the routing:

```
app.use('/', routes);
app.use('/users', users);
```

Here the example route files that were required are attached to our app. If you need to add additional routes, you'd do it here.

All the code after this is used for error handling. You usually won't have to modify this code unless you want to change the way Express handles errors. By default, it's set up to show the error that occurred in the route when you're in development mode.

A Useful Tool

Hopefully you now have a clear idea of how the express-generator tool can save you time writing repetitive boilerplate when starting new Express-based projects.

By providing a sensible default file structure, and installing and wiring up commonly needed middleware, the generator creates a solid foundation for new applications with just a couple of commands.

Chapter 8: An Introduction to MongoDB

by Manjunath M

MongoDB is an open-source, document-oriented, NoSQL database program. If you've been involved with the traditional, relational databases for long, the idea of a document-oriented, NoSQL database might indeed sound peculiar. "How can a database not have tables?", you might wonder. This tutorial introduces you to some of the basic concepts of MongoDB and should help you get started even if you have very limited experience with a database management system.

What's MongoDB?

Here's what the [official documentation](#) has to say about MongoDB.

MongoDB is an open-source database developed by MongoDB Inc. that's scalable and flexible. MongoDB stores data in JSON-like documents that can vary in structure.

Obviously, MongoDB is a database system. But why do we need another database option when the existing solutions are inexpensive and successful? A relational database system like MySQL or Oracle has tables, and each table has a number of rows and columns. MongoDB, being a document-oriented database system, doesn't have them. Instead, it has a JSON-like document structure that is flexible and easy to work with. Here's an example of what a MongoDB document looks like:

```
{  
    "_id": ObjectId(3da252d3902a),  
    "type": "Article",  
    "title": "MongoDB Tutorial Part One",  
    "description": "First document",  
    "author": "Manjunath",  
    "content": "This is an..."  
}  
{  
    "_id": ObjectId(8da21ea3902a),  
    "type": "Article",  
    "title": "MongoDB Tutorial Part Two",  
    "description": "Second document",  
    "author": "Manjunath",  
    "content": "In the second..."  
}
```

A document in a NoSQL database corresponds to a row in an SQL database. A group of documents together is known as a **collection**, which is roughly synonymous with a table in a relational database. For an in-depth overview of both NoSQL and SQL databases and their differences, we have that covered in the [SQL vs NoSQL tutorial](#).

Apart from being a NoSQL database, MongoDB has a few qualities of its own. I've listed some of its key features below:

- it's easy to install and set up
- it uses a BSON (a JSON-like format) to store data
- it's easy to map the document objects to your application code
- it claims to be highly scalable and available, and includes support for out-of-the-box replication
- it supports MapReduce operations for condensing a large volume of data into useful aggregated results
- it's free and open source.

MongoDB appears to overcome the limitations of the age-old relational databases and NoSQL databases. If you aren't yet convinced about why it's useful, check out our article on [Choosing Between NoSQL and SQL](#).

Let's go ahead and install MongoDB.

Installing MongoDB

Installing MongoDB is easy and it doesn't require much configuration or setup. MongoDB is supported by all major platforms and distributions, and you can check out their documentation if you're in doubt.

I've covered the instructions for installing MongoDB on macOS below. For Linux, MongoDB recommends installing the software with the help of your [distribution's package manager](#). If you're on Windows, it's as easy as [downloading the latest release](#) from the MongoDB website and running the interactive installer.

Installing MongoDB on macOS using Homebrew

Homebrew is a package manager for macOS, and this tutorial assumes you've already installed Homebrew on your machine.

1. Open the terminal application and run

```
$ brew update
```

1. Once brew is updated, install the mongodb package as follows

```
$ brew install mongodb
```

1. Create a data directory so that the mongod process will be able to write data. By default, the directory is /data/db:

```
$ mkdir -p /data/db
```

2. Make sure your user account has permission to read and write data into that directory:

```
$ sudo chown -R `id -un` /data/db  
&gt; # Enter your password
```

3. Initiate the mongo server. Run the mongod command to start the server.
4. Start the Mongo shell. Open up another terminal window and run mongo. The mongo shell will attempt to connect to the mongo daemon that we initiated earlier. Once you're successfully connected, you can use the shell as a playground to safely run all your commands.
5. Exit the Mongo shell by running quit() and the mongo daemon by pressing control + c.

Now that we have the mongo daemon up and running, let's get acquainted with the MongoDB basics.

Basic Database Operations

Enter the Mongo shell if you haven't already:

```
[mj@localhost ~]$ mongo
MongoDB shell version v3.6.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.1

connecting to: test
```

You can see my version of MongoDB shell when you log in. By default, you're connected to a test database. You can verify the name of the current database by running db:

```
&gt; db
test
```

That's awesome, but what if we want a new database? To create a database, MongoDB has a use DATABASE_NAME command:

```
&gt;use exampledb
switched to db exampledb
```

To display all the existing databases, try show dbs:

```
&gt; show dbs
local      0.078GB
prototype  0.078GB
test       0.078GB
```

The exampledb isn't in the list because we need to insert at least one document into the database. To insert a document, you can do

db.COLLECTION_NAME.insertOne({ "key": "value" }). Here's an example:

```
&gt; db.users.insertOne({ 'name': 'Bob' })
{
  "acknowledged" : true,
  "insertedId"   : ObjectId("5a52c53b223039ee9c2daaec")
}
```

MongoDB automatically creates a new users collection and inserts a document with the key-value pair 'name' : 'Bob'. The ObjectId returned is the id of document inserted. MongoDB creates a unique ObjectId for each document on creation and it becomes the default value of the `_id` field:

```
&gt; show dbs
exampledb 0.078GB
local      0.078GB
prototype  0.078GB
test       0.078GB
```

Similarly, you can confirm that the collection was created using the `show collections` command:

```
&gt; show collections
users
```

We've created a database, added a collection named `users` and inserted a document into it. Now let's try dropping it. To drop an existing database, use the `dropDatabase()` command as exemplified below:

```
&gt; db.dropDatabase()
{ "ok" : 1 }
```

`show dbs` confirms that the database was indeed dropped.

```
> show dbs
local      0.078GB
prototype  0.078GB
test       0.078GB
```

For more database operations, see MongoDB reference page on [Database Commands](#).

MongoDB CRUD Operations

As you might already know, the CRUD acronym stands for Create, Read, Update, and Delete. These are the four basic database operations that you can't avoid while building an application. For instance, any modern application will have the ability to create a new user, read the user data, update the user information and, if needed, delete the user account. Let's accomplish this at the database level using MongoDB.

Create Operation

Creation is the same as [inserting a document](#) into a collection. In the previous section, we had inserted a single document using the `db.collection.insertOne()` syntax. There's another method called `db.collection.insertMany()` that lets you insert multiple documents at once. Here's the syntax:

```
&gt; db.COLLECTION_NAME.insertMany(  
  [ &lt;document 1&gt;, &lt;document 2&gt;, ... ]  
)
```

Let's create a `users` collection and populate it with some actual users:

```
&gt; db.users.insertMany([  
  { "name": "Tom", "age": 33, "email": "tom@example.com" },  
  { "name": "Bob", "age": 35, "email": "bob@example.com" },  
  { "name": "Kate", "age": 27, "email": "kate@example.com" },  
  { "name": "Watson", "age": 15, "email": "watson@example.com" }  
)  
  
{  
  "acknowledged" : true,  
  "insertedIds" : [  
    ObjectId("5a52cb2451dd8b08d5a22cf5"),  
    ObjectId("5a52cb2451dd8b08d5a22cf6"),  
    ObjectId("5a52cb2451dd8b08d5a22cf7"),  
    ObjectId("5a52cb2451dd8b08d5a22cf8")  
  ]  
}
```

The `insertMany` method accepts an array of objects and, in return, we get an

array of ObjectIDs.

Read Operation

Read operation is used to [retrieve a document](#) or multiple documents from a collection. The syntax for the read operation is as follows:

```
&gt; db.collection.find(query, projection)
```

To retrieve all user documents, you can do this:

```
&gt; db.users.find().pretty()
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf5"),
  "name" : "Tom",
  "age" : 33,
  "email" : "tom@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf6"),
  "name" : "Bob",
  "age" : 35,
  "email" : "bob@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf7"),
  "name" : "Kate",
  "age" : 27,
  "email" : "kate@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf8"),
  "name" : "Watson",
  "age" : 15,
  "email" : "watson@example.com"
}
```

This corresponds to the `SELECT * FROM USERS` query from an SQL database.

The `pretty` method is a cursor method, and there are many others too. You can chain these methods to modify your query and the documents that are returned by the query.

What if you need to filter queries and return a subset of the collection? Say, find

all users who are below 30. You can modify the query like this:

```
&gt; db.users.find({ "age": { $lt: 30 } })
{ "_id" : ObjectId("5a52cb2451dd8b08d5a22cf7"), "name" : "Kate", "ag
{ "_id" : ObjectId("5a52cb2451dd8b08d5a22cf8"), "name" : "Watson", "
```

`$lt` is a query filter operator that selects documents whose `age` field value is less than 30. There are many comparison and logical query filters available, and you see the entire list in the [Query Selector documentation](#).

Update Operation

The [update operation](#) modifies the document in a collection. Similar to the create operation, MongoDB offers two methods for updating a document. They are:

1. `db.collection.updateMany(filter, update, options)`
2. `db.collection.updateMany(filter, update, options).`

If you need to add an extra field, say `registration`, to all the existing documents in a collection, you can do something like this:

```
&gt; db.users.updateMany({}, {$set: { 'registration': 'incomplete'}}
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4 }
```

The first argument is an empty object because we want to update all documents in the collection. The `$set` is an update operator that sets the value of a field with the specified value. You can verify that the extra field was added using `db.users.find()`.

To update the value of documents that match certain criteria, `updateMany()` accepts a filter object as the first argument. For instance, you might want to overwrite the value of `registration` to `complete` for all users who are aged 18+. Here is what you can do:

```
&gt; db.users.updateMany(
  {'age':{ $gt: 18} },
  {$set: { 'registration': 'complete'}
)
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

To update the registration details of a single user, you can do this:

```
&gt; db.users.updateOne(  
  {'email': 'watson@example.com' },  
  {$set: { 'registration': 'complete'}  
)  
  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Delete Operation

[Delete operation](#) removes a document from the collection. To delete a document, you can use the `db.collection.deleteOne(filter, options)` method, and to delete multiple documents, you can use the `db.collection.deleteMany(filter, options)` method.

To delete documents based on certain criteria, you can use the filter operators that we used for the read and update operation:

```
db.users.deleteMany( { status: { $in: [ "dormant", "inactive" ] } }  
  
{ "acknowledged" : true, "deletedCount" : 1 }
```

This deletes all documents with a status of “dormant” or “inactive”.

That's it.

An Overview of MongoDB Drivers

For an application to communicate with the mongodb-server, you have to use a client-side library called a **driver**. The driver sits on top of the database server and lets you interact with the database using the driver API. MongoDB has official and third-party drivers for all popular languages and environments.

In this tutorial, we're going to look at our options for drivers for Node.js. Some drivers add lots of good features, like schema support and validation of business logic, and you can choose the one that matches your style. The popular drivers for Node.js include the native MongoDB driver and Mongoose. I'll briefly discuss their features here.

MongoDB Node.js Driver

This is the official [MongoDB driver for Node.js](#). The driver can interact with the database using either promises or callbacks, and also supports the ES6 `async/await` functions. The example below demonstrates connecting the driver to the server:

```
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');

// Connection URL
const url = 'mongodb://localhost:27017/exampledbs';
// Database Name
const dbName = 'exampledbs';

(async function() {

  let client;

  try {
    // Attempts to connect to the server
    client = await MongoClient.connect(url);

    console.log("Successfully connected to the server.");

    const db = client.db(dbName);
  } catch (err) {
    // Log errors to console
  }
})()
```

```
    console.log(err.stack);
}

if (client) {
  client.close();
}
})();
```

The `MongoClient.connect` returns a promise, and any error is caught by the `try ... catch` block. You'll be implementing the CRUD actions inside the `try ... catch` block, and the API for that is similar to what we've covered in this tutorial. You can read more about it in the official documentation page.

Mongoose Driver

Another popular Node.js driver for MongoDB is [Mongoose](#). Mongoose is built on top of the official MongoDB driver. Back when Mongoose was released, it had tons of features that the native MongoDB driver didn't have. One prominent feature was the ability to define a schema structure that would get mapped onto the database's collection. However, the latest versions of MongoDB have adopted some of these features in the form of JSON schema and schema validation.

Apart from schema, other fancy features of Mongoose include Models, Validators and middlewares, the `populate` method, plugins and so on. You can read more about these in the Mongoose docs. Here's an example to demonstrate `mongoose.connect()` for connecting to the database:

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

// Connection URL
const url = 'mongodb://localhost:27017/exampledbs';

mongoose.connect(url);

var db = mongoose.connection;

db.on('error',
  console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

Conclusion

MongoDB is a popular NoSQL database solution that suits modern development requirements. In this tutorial, we've covered the basics of MongoDB, the Mongo shell and some of the popular drivers available. We've also explored the common database operations and CRUD actions within the Mongo shell. Now it's time for you to head out and try what we've covered here and more. If you want to learn more, I recommend creating a REST API with MongoDB and Node to acquaint yourself with the common database operations and methods.