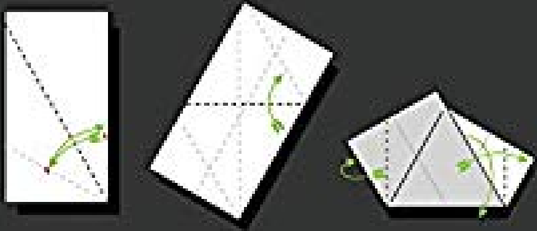
sitepoint

YOUR FIRST WEEK WITH NODE.JS



BLAZINGLY FAST WEB APPS

Your First Week With Node.js

Cover Design: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

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.

In this book, we'll offer a beginner's introduction to Node and its related technologies, and get you under way writing your first Node applications.

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.

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.

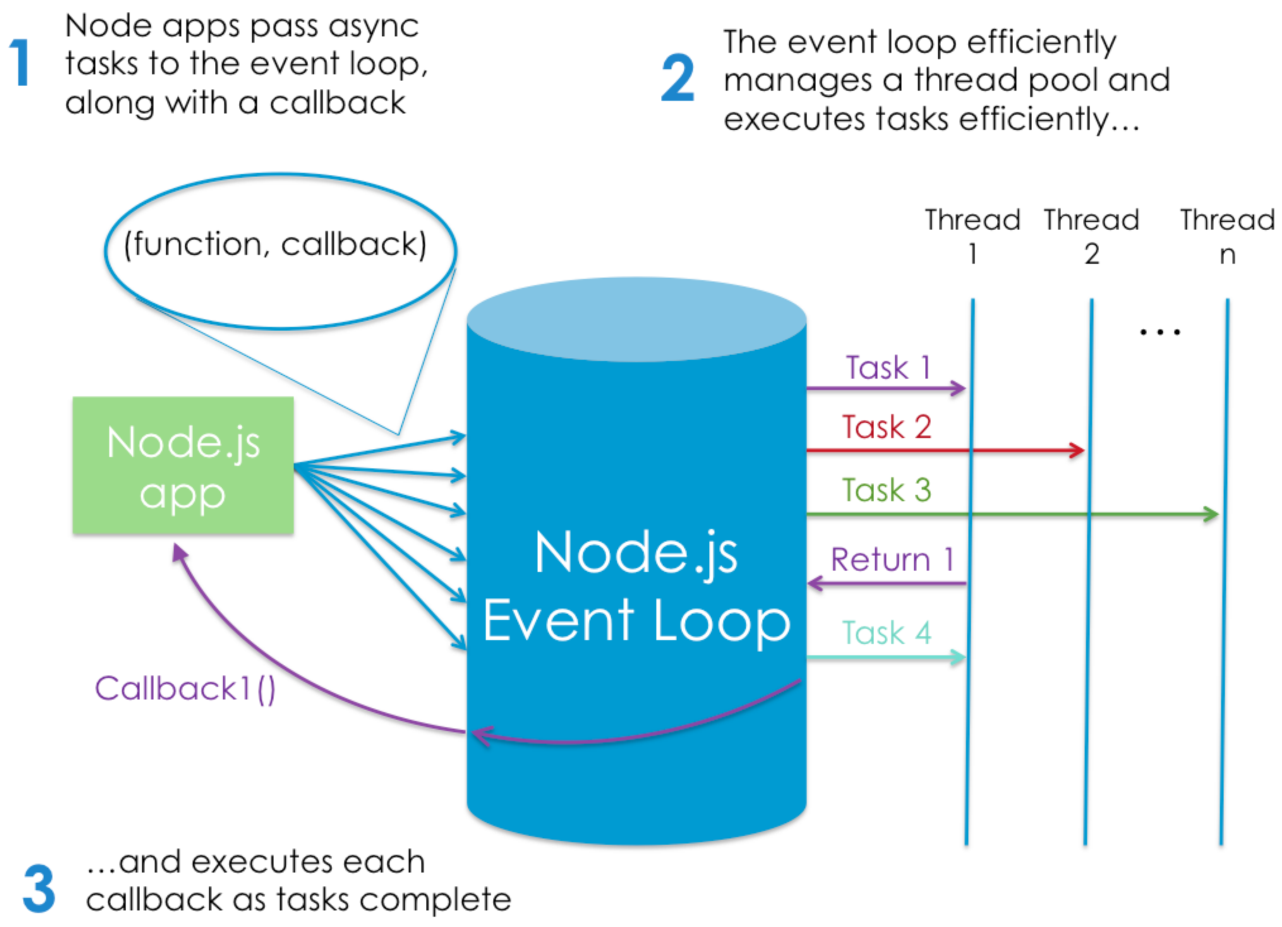
THE NODE.JS EXECUTION MODEL

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:


1 Node apps pass async tasks to the event loop, along with a callback
2 The event loop efficiently manages a thread pool and executes tasks efficiently…
(function, callback)
Node.js app
Node.js Event Loop
Thread 1
Thread 2
Thread n
…
Task 1
Task 2
Task 3
Return 1
Task 4
Callback1()
3 …and executes each callback as tasks complete

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 can 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.

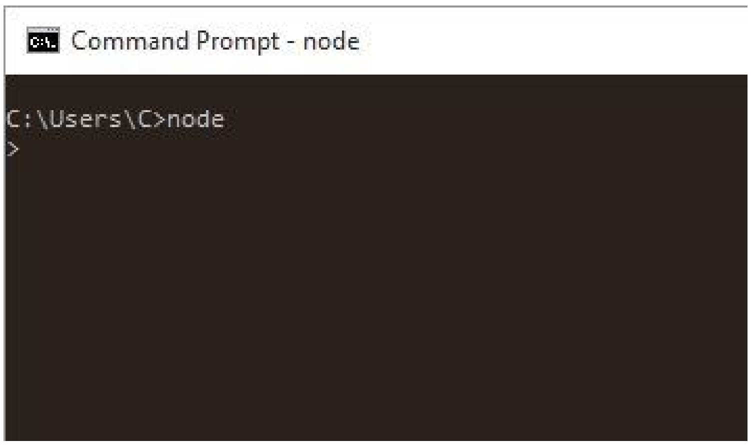
Chapter 2: A Beginner Splurge in Node.js

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.



```
C:\Users\C>node
>
```

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

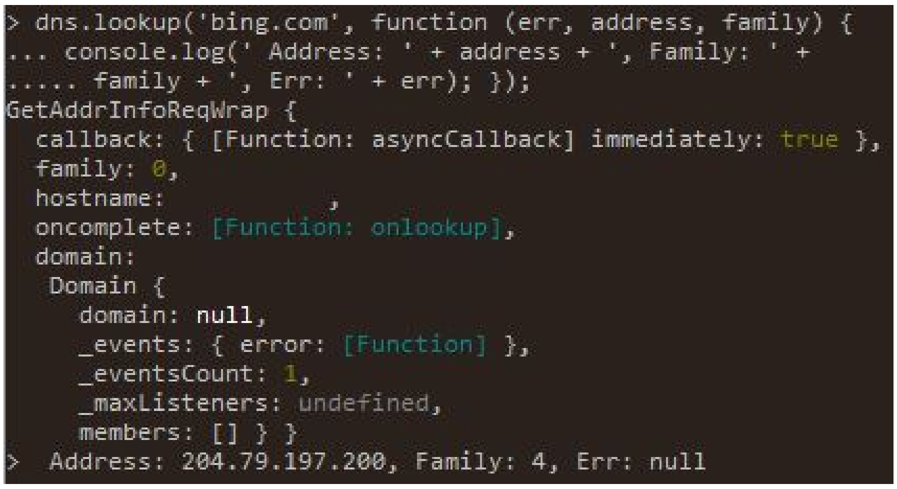
We use a `setTimeout` to delay execution in the current thread. Timeouts don't guarantee time of execution. We place a `Math.random()` to make it even more fickle, and call `doSomething()`, followed by a `console.log('test')`, to display delayed execution. You'll experience a short delay between one 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.log(' A
➥ + 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 lookupc 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.log('
➥ }); 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) { co
➥(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:

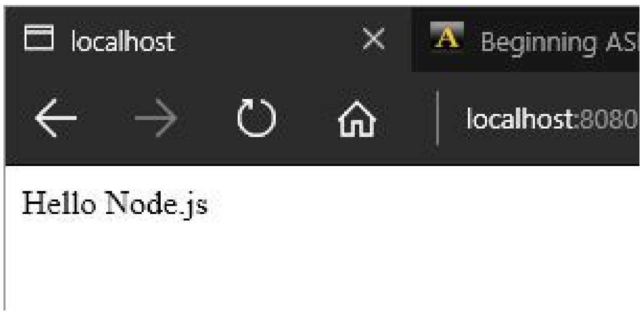
```
> var http = require('http');
> var server = http.createServer(function (request, response) { response.e
➥'Hello Node.js'); });
```

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.



Hello Node.js

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.end(r
➥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.keys(re
➥headers).forEach(function (key) { response.write(key + ': ' + request.hea
➥ + ' '); }); 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
+ 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 this
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       | VERSION
mkdir      | Directory crea…       | =joehewitt         | 2012-04-17 | 0.0.2
fs-extra   | fs-extra conta…       | =jprichardson…     | 2017-05-04 | 3.0.1
mkdirp     | Recursively mkdir,…   | =substack          | 2015-05-14 | 0.5.1
...
```

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 a 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.org
```

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 momemt 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 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, it's 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

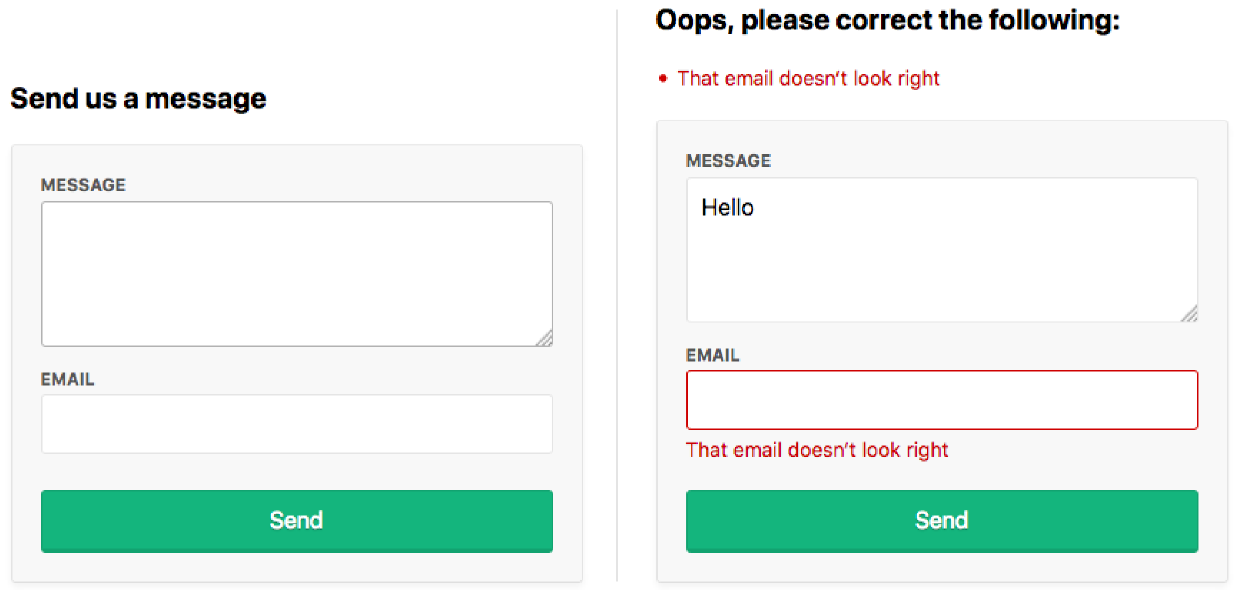
If you're building a web application, you're likely to encounter the need to build HTML forms on day one. They're a big part of the web experience, and they can be complicated.

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.


Send us a message
MESSAGE
EMAIL
Send
Oops, please correct the following:
• That email doesn't look right
MESSAGE
Hello
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" autofocus
      </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" autofocus
    <%= 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 csrf = 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' : '' %>
    <label for="message">Message</label>
    <textarea class="input" id="message" name="message" rows="4" autofocus
    ➥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.

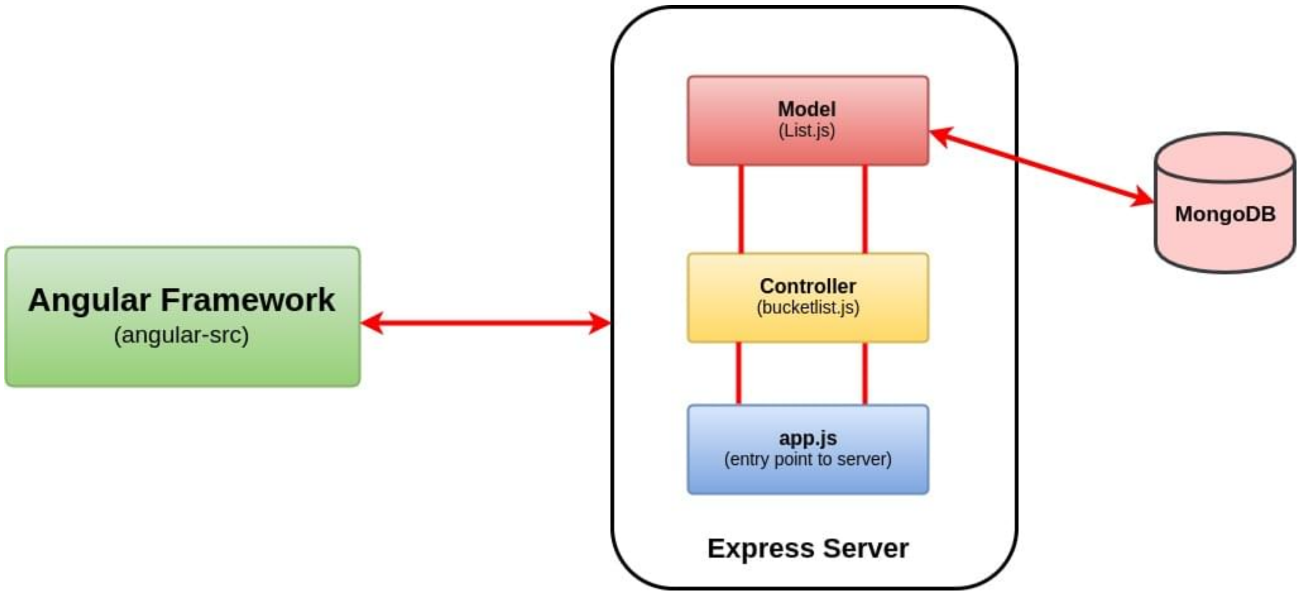
MongoDB - MongoDB is a document-oriented database program where the documents are stored in a flexible JSON-like format. Being an NoSQL database program, MongoDB relieves you from the tabular jargon of the relational database.

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?

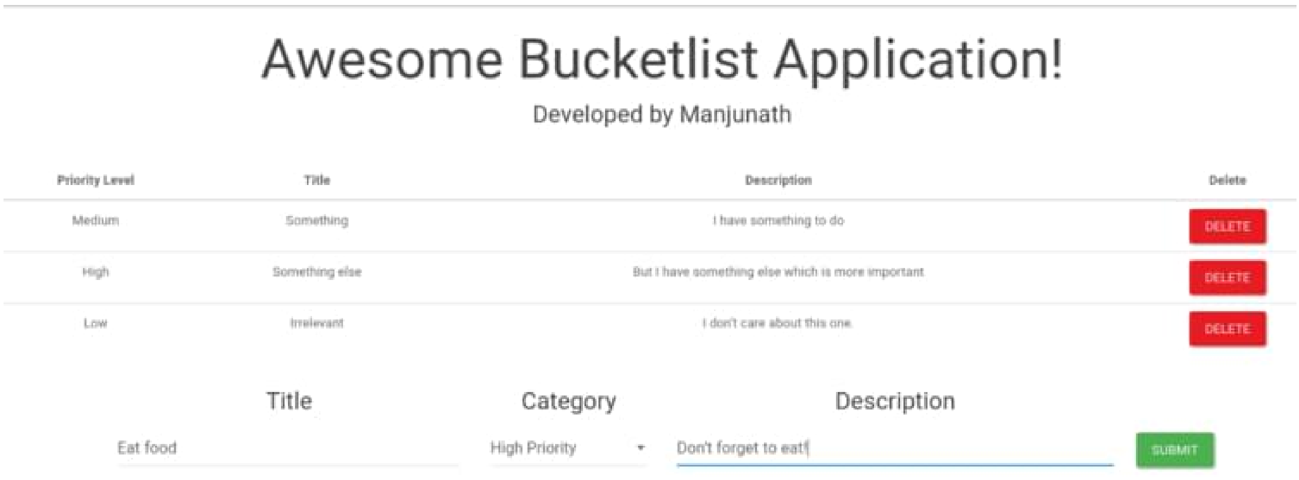
Overview

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	DELETE
High	Something else	But I have something else which is more important	DELETE
Low	Irrelevant	I don't care about this one.	DELETE

Title	Category	Description	
Eat food	High Priority	Don't forget to eat!	SUBMIT

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.

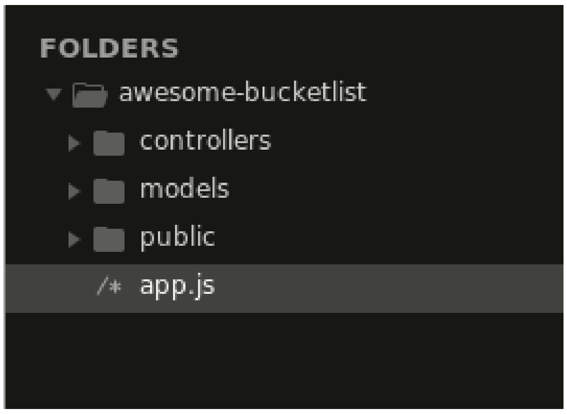
```
mkdir awesome-bucketlist
cd awesome-bucketlist
```

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.



```
FOLDERS
▾ awesome-bucketlist
  ▸ controllers
  ▸ models
  ▸ public
    app.js
```

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 breaking c

 "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
*/
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 the

*/
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', Bucketlis
```

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. E
                ➡ ${err}`});
        }
        else {
            res.write(JSON.stringify({success: true, lists:lists},null,2))
            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.

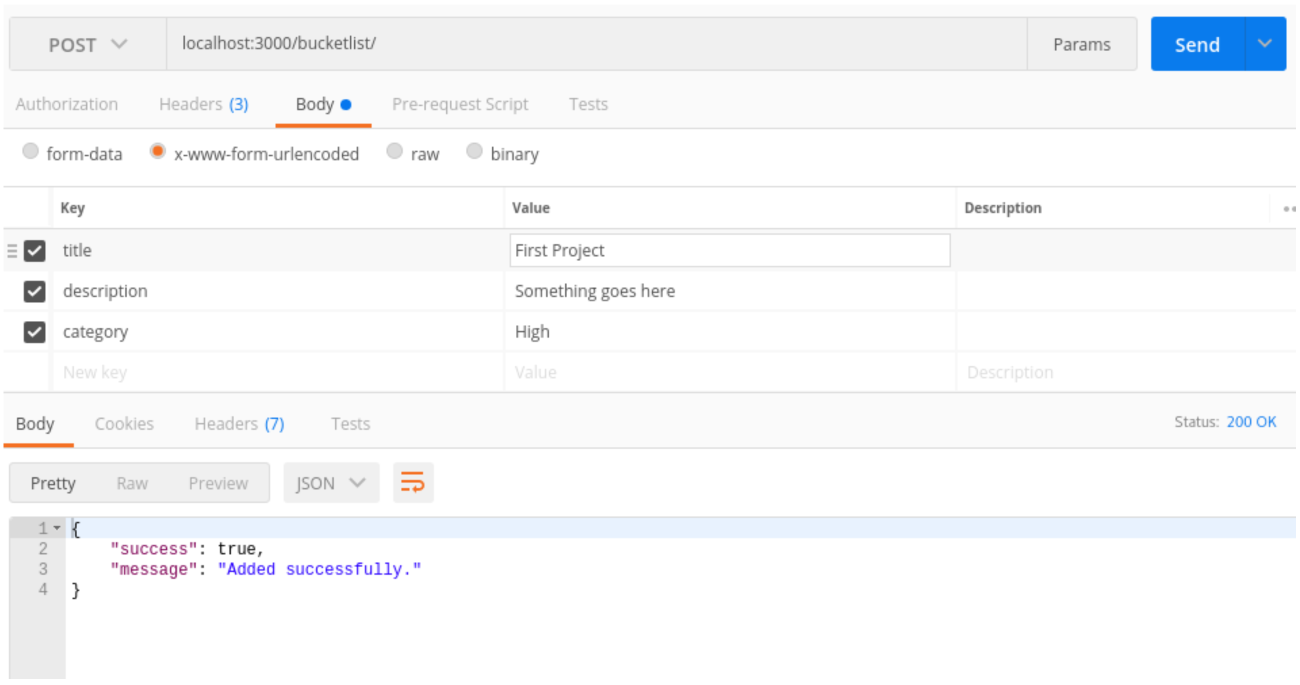
```
//newList.save is used to insert the document into MongoDB
```

```
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 new lis
                ➥ ${err}`});

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

    });
});

//DELETE HTTP method to /bucketlist. Here, we pass in a param which is the

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 list. I
                ➥ ${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.



localhost:3000/bucketlist/

Key	Value	Description
title	First Project	
description	Something goes here	
category	High	
New key	Value	Description

Status: 200 OK

```
{
    "success": true,
    "message": "Added successfully."
}
```

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.

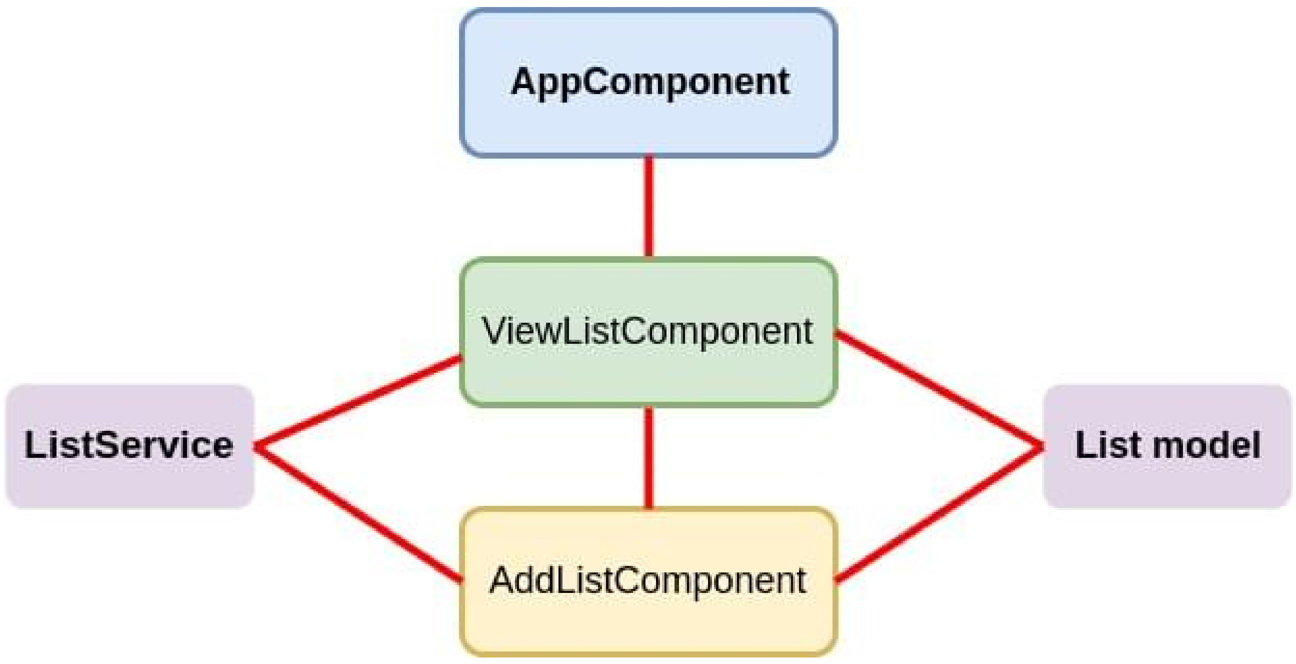
```
ng new angular-src
```

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.

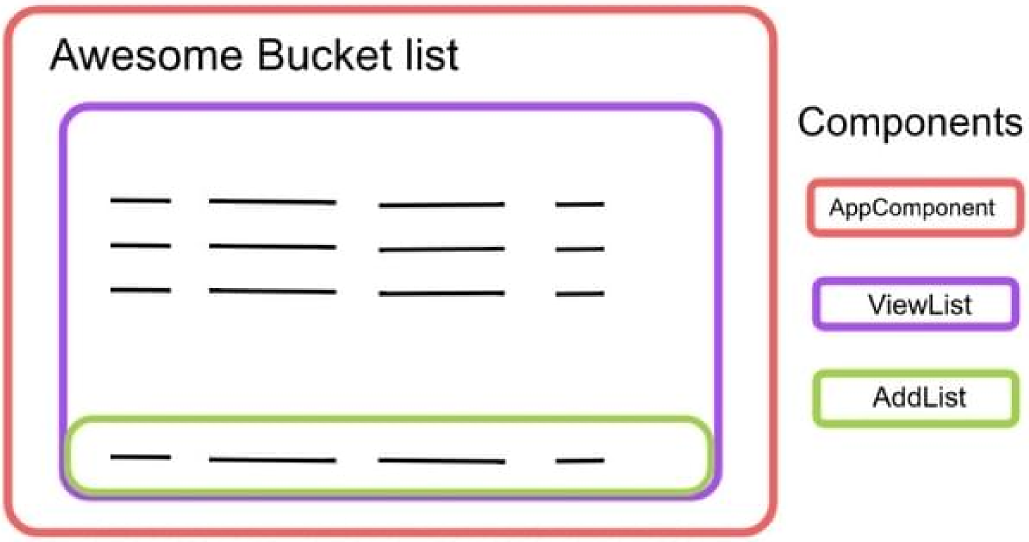
The generate command has already added our components into the `appModule`. Go ahead and import `ListService` and add it to the `providers` array. We also need to import `FormsModule` and `HTTPModule` and declare them as imports. `FormsModule` is needed to create the form for our application and `HTTPModule` 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 propoerty which is an array of List type
  private lists: List[] = [];

  constructor(private listServ: ListService) { }

  ngOnInit() {

    //Load all list on init
    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 .filter m
  public deleteList(list: List) {
    this.listServ.deleteList(list._id).subscribe(
      response =>    this.lists = this.lists.filter(lists => lists !== lis
    }

}
```

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); $event.
      ➥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" requir
          </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 Prioirty</option>

            </select>
          </div>

          <div>
            <label for="description">description</label>
            <input type="text" [(ngModel)]="newList.description" name="descrip
            ➥ 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 fadeOut class in element')
```

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.

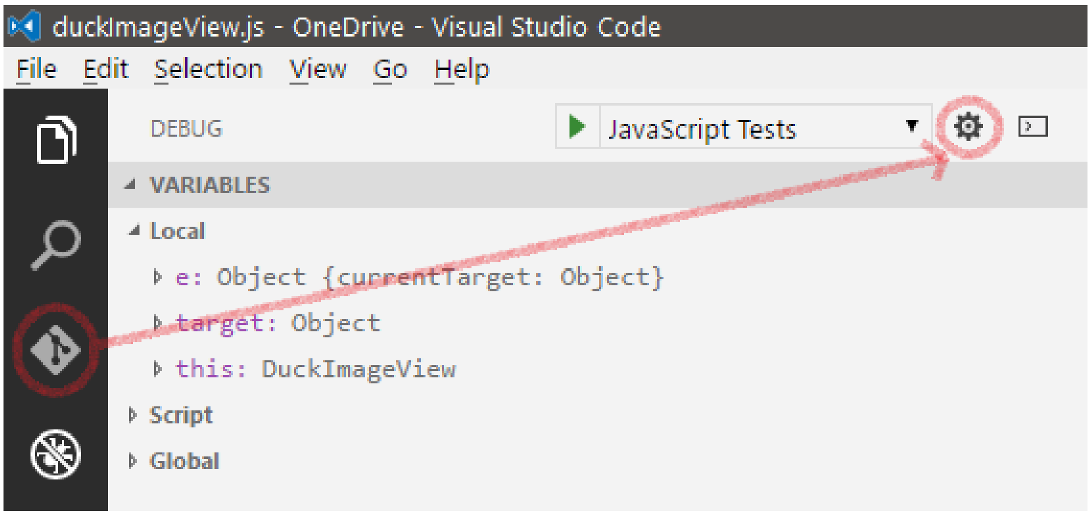
Debugging inside an Editor

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:


duckImageView.js - OneDrive - Visual Studio Code
File Edit Selection View Go Help
DEBUG
JavaScript Tests
VARIABLES
Local
Script
Global

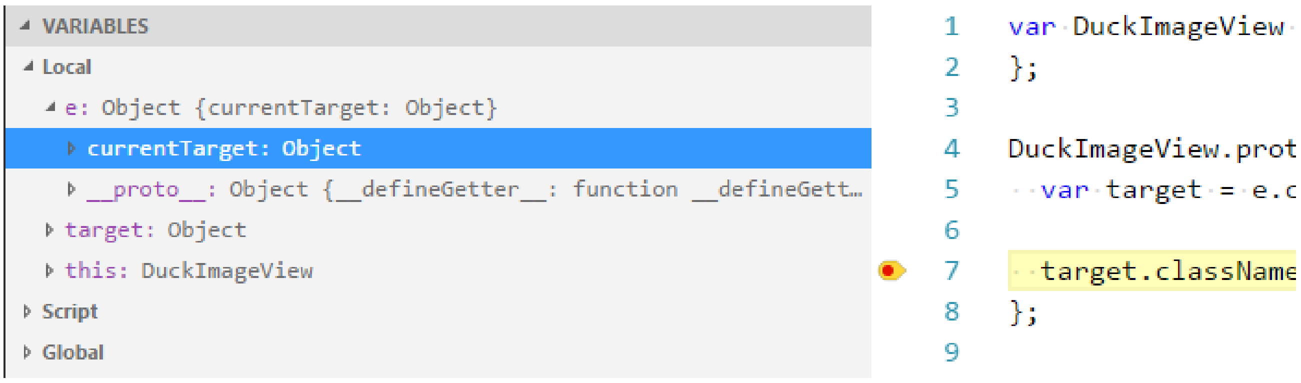
```
e: Object {currentTarget: Object}
target: Object
this: DuckImageView
```

A `launch.json` file will open up. This allows you 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:



```
VARIABLES
Local
  e: Object {currentTarget: Object}
    currentTarget: Object
    __proto__: Object {__defineGetter__: function __defineGett…
  target: Object
  this: DuckImageView
Script
Global
```

```
var DuckImageView
};

DuckImageView.prot
  var target = e.c

  target.className
};
```

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.

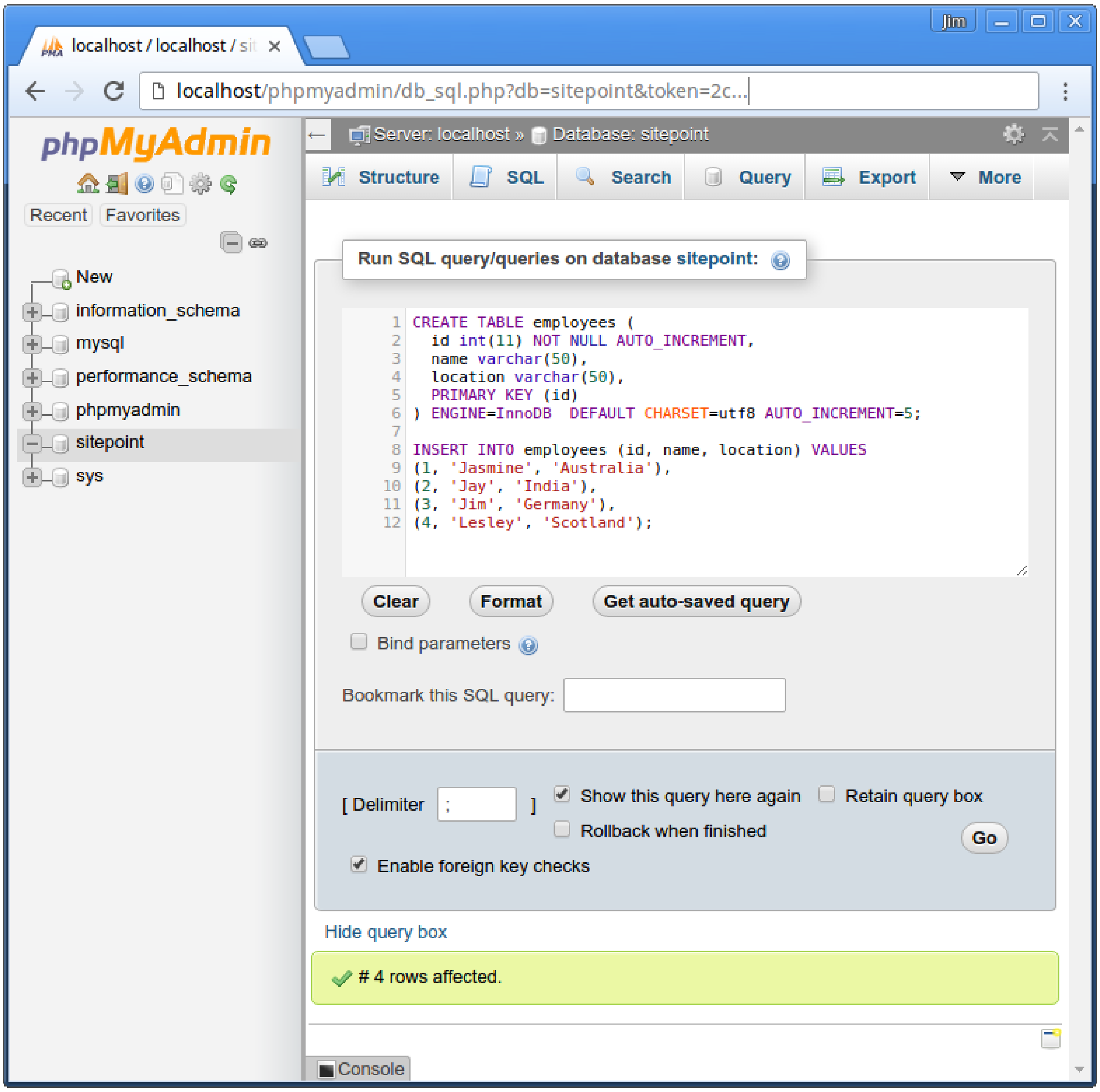
Getting Started

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');
```



```
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');
```

4 rows affected.

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-know 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 lopping 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.

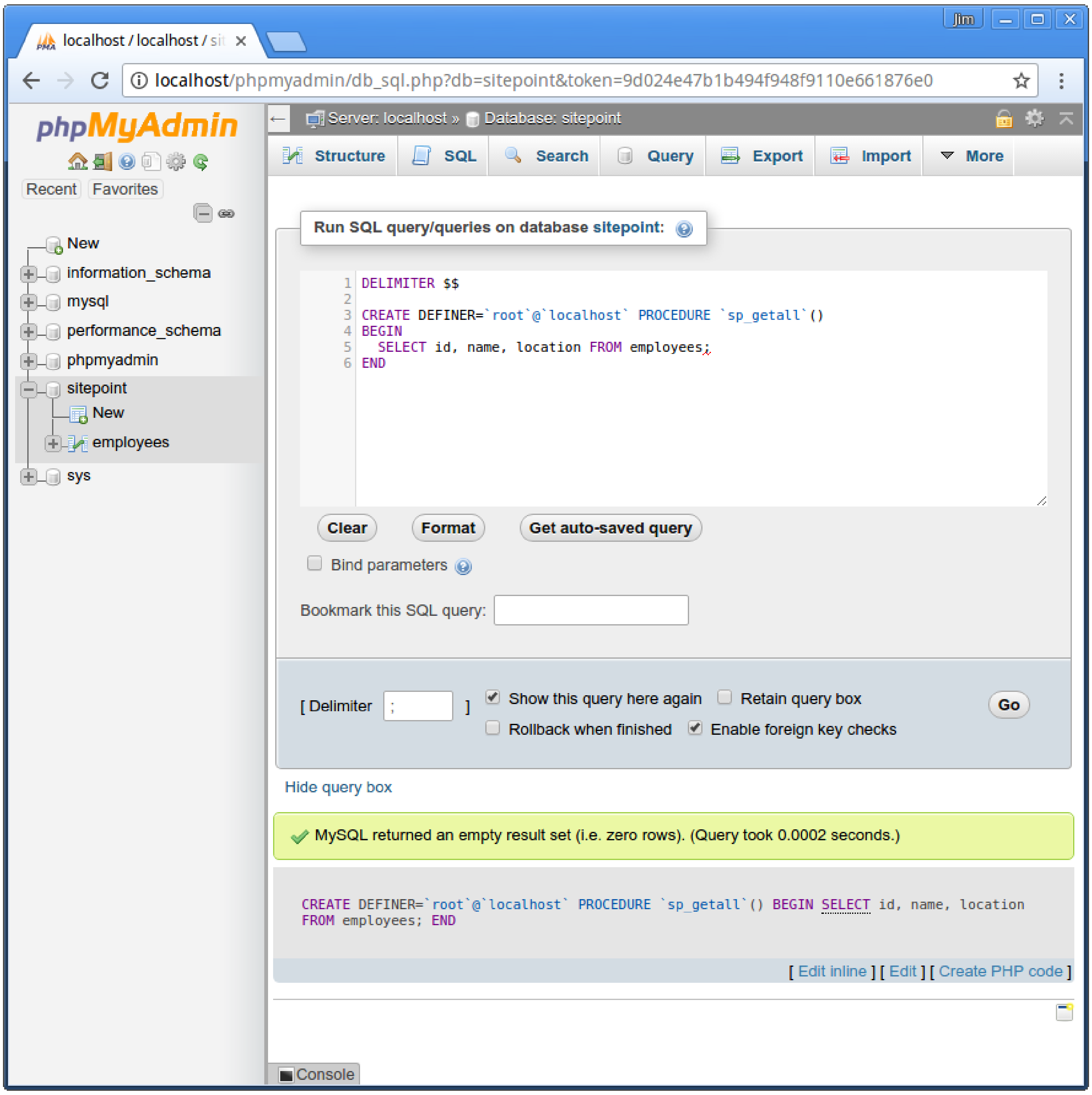
STORED PROCEDURES

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.



Run SQL query/queries on database sitepoint:

```
DELIMITER $$

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

Bind parameters

Bookmark this SQL query:

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

MySQL returned an empty result set (i.e. zero rows). (Query took 0.0002 seconds.)

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

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', 'USA
  ➥ 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

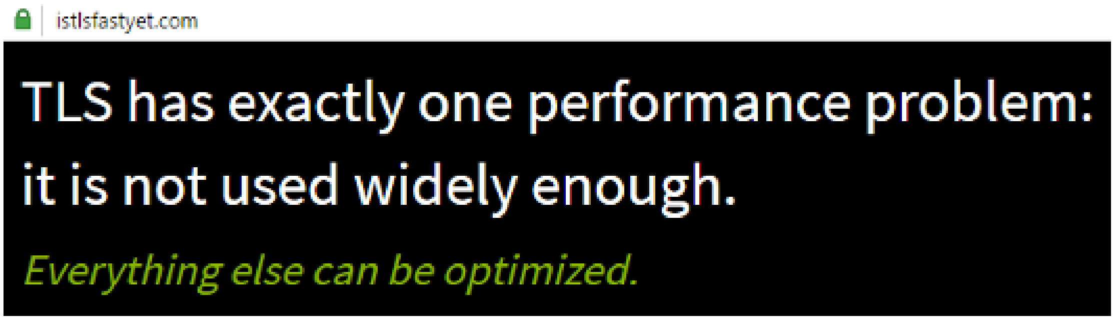
The HTTP/2 specification was published as RFC 7540 in May 2015, which means at this point it's a part of the standard. This was a major milestone. Now we can all upgrade our servers to use HTTP/2. One of the most important aspects is the backwards compatibility with HTTP 1.1 and the negotiation mechanism 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.


istlsfastyet.com

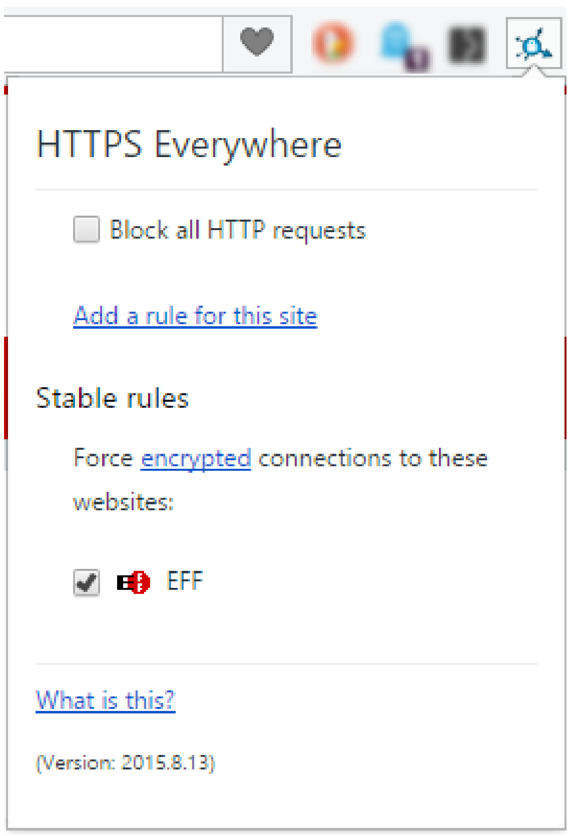
TLS has exactly one performance problem: it is not used widely enough.

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.


HTTPS Everywhere

☐ Block all HTTP requests

Add a rule for this site

Stable rules

Force encrypted connections to these websites:

☑ EFF

What is this?

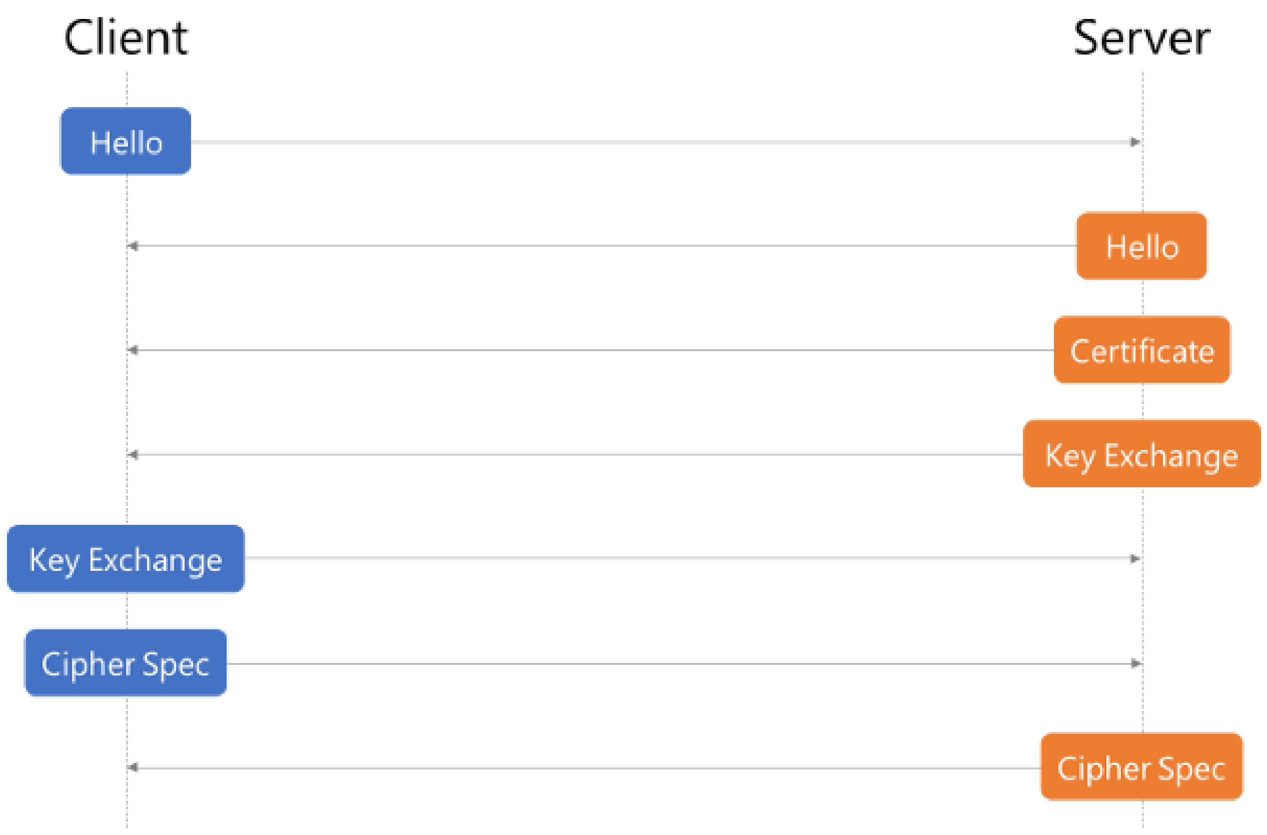
(Version: 2015.8.13)

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 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 functionallity 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 -d examp
```

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") // these
  ➥might differ for you, make sure to copy from the certbot output
};

// ...
```

Tighetning 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"), // thes
  ➥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.