Community Experience Distilled

Node.js Essentials

From client to server, learn how Node.js can help you use JavaScript more effectively to develop faster and more scalable applications with ease

Fabian Cook

[PACKT] open source*
PUBLISHING
community experience distilled

Node.js Essentials

Table of Contents

Node.js Essentials

Node.js Essentials

First published: November 2015

Production reference: 1301015

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-492-2

www.packtpub.com

Credits

Author

Fabian Cook

Reviewers

Shoubhik Bose

Glenn Geenen

Commissioning Editor

Edward Gordan

Acquisition Editor

Divya Poojari

Content Development Editor

Athira Laji

Technical Editor

Naveenkumar Jain

Copy Editor

Sneha Singh

Project Coordinator

Harshal Ved

Proofreader

Safis Editing

Indexer

Hemangini Bari

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

About the Author

Fabian Cook is an experienced JavaScript developer who lives in Hawkes Bay, New Zealand. He began working with Java and C# very early in his life, which lead to using Node.js in an open source context. He is now currently working for a New Zealand ISP, known as NOW NZ where they are utilizing the full power of Node.js, Docker and CoreOS.

About the Reviewer

Glenn Geenen is a Node.js developer with a background in game and mobile development. He has mostly worked as an iOS consultant before becoming a Node.js consultant for his company, GeenenTijd.

Support files, eBooks, discount offers, and more

Preface

Node.js is simply a tool that lets you use JavaScript on the server side. However, it actually does much more than that – by extending JavaScript, it allows for a much more integrated and efficient approach to development. It comes as no surprise that it's a fundamental tool for full-stack JavaScript developers. Whether you work on the backend or frontend, you adopt a much more collaborative and agile way of working using Node.js, so that you and your team can focus on delivering a quality end product. This will ensure that you're ready to take on any new challenge that gets thrown at you.

This book will be fast paced and cover dependency management, running your own HTTP server, real time communication, and everything in between that is needed to get up and running with Node.js.

What this book covers

Chapter 1, *Getting Started*, covers the setup of Node.js. You will also cover how to utilize and manage dependencies.

Chapter 2, *Simple HTTP*, covers how to run a simple HTTP server and helps you understand routing and utilization of middleware.

Chapter 3, *Authentication*, covers the utilization of middleware and JSON Web Token to authenticate users.

Chapter 4, *Debugging*, covers the integration of post-mortem techniques in your development tasks and how to debug your Node.js programs.

Chapter 5, *Configuration*, covers the configuration and maintenance of your software using centralized configuration options, arguments, and environmental variables.

Chapter 6, *LevelDB and NoSQL*, covers the introduction of NoSQL databases, such as LevelDB and MongoDB. It also covers the use of the simple key/value store and a more complete document database.

Chapter 7, *Socket.IO*, explores the real-time communication between clients, servers, and back again and also how it authenticates and notifies the users.

Chapter 8, *Creating and Deploying Packages*, focuses on sharing the modules and contributing to the eco-system

Chapter 9, *Unit Testing*, tests your code using Mocha, Sinon, and Chance and also covers how to use mocks with functions and generate random values to test your code

Chapter 10, *Using More Than JavaScript*, explains the usage of CoffeeScript with Node.js to expand language capabilities.

What you need for this book

You will need a computer that runs Unix (Macintosh), Linux or Windows, along with your preferred Integrated Development Environment. If you don’t have an IDE then you have a few options, such as:

- Atom: https://atom.io/
- Sublime: http://www.sublimetext.com/
- Cloud 9: https://c9.io/

Who this book is for

The book will be helpful to anybody who wants to have knowledge of Node.js (what Node.js is about, how to use it, where it’s useful and when to use it). Familiarity with server-side and Node.js is a prerequisite.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<script type='application/javascript' src='script_a.js'></script>
<script type='application/javascript' src='script_b.js'></script>
```

Any command-line input or output is written as follows:

```
[~]$ npm install -g n
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If the user hasn't passed both the username and password the server will return **500 Bad Request**".

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <`feedback@packtpub.com`>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started

Every Web developer must have come across it every once in a while, even if they just dabble in simple Web pages. Whenever you want to make your Web page a little more interactive, you grab your trustworthy friends, such as JavaScript and jQuery, and hack together something new. You might have developed some exciting frontend applications using AngularJS or Backbone and want to learn more about what else you can do with JavaScript.

While testing your website on multiple browsers you must have come across Google Chrome at some point and you might have noticed that it is a great platform for JavaScript applications.

Google Chrome and Node.js have something very big in common: they both work on Google’s high-performance V8 JavaScript engine, this gives us the same engine in the browser that we will be using in the backend, pretty cool, right?

Setting up

In order to get started and use Node.js, we need to download and install Node.js. The best way to install it will be to head over to https://nodejs.org/ and download the installer.

At the time of writing, the current version of Node.js is 4.2.1.

To ensure consistency, we are going to use a `npm` package to install the correct version of Node.JS and, for this, we are going to use the `n` package described at https://www.npmjs.com/package/n.

Currently, this package has support only for `*nix` machines. For Windows. see nvm-windows or download the binary for 4.2.1 from https://nodejs.org/dist/v4.2.1/.

Once you have Node.js installed, open a terminal and run:

```
[~]$ npm install -g n
```

The `-g` argument will install the package globally so we can use the package anywhere.

Linux users may need to run commands that install global packages as `sudo`.

Using the recently install package, run:

```
[~]$ n
```

This will display a screen with the following packages:

```
  node/0.10.38
  node/0.11.16
  node/0.12.0
  node/0.12.7
  node/4.2.1
```

If `node/4.2.1` isn't marked we can simply run the following packages; this will ensure that `node/4.2.1` gets installed:

```
[~]$ sudo n 4.2.1
```

To ensure that the `node` is good-to-go, lets create and run a simple `hello world` example:

```
[~/src/examples/example-1]$ touch example.js
[~/src/examples/example-1]$ echo "console.log(\"Hello world\")" >
example.js
[~/src/examples/example-1]$ node example.js
Hello World
```

Cool, it works; now let's get down to business.

Hello require

In the preceding example, we just logged a simple message, nothing interesting, so let’s dive a bit deeper in this section.

When using multiple scripts in the browser, we usually just include another script tag such as:

```
<script type='application/javascript' src='script_a.js'></script>
<script type='application/javascript' src='script_b.js'></script>
```

Both these scripts share the same global scope, this usually leads to some unusual conflicts when people want to give variables the same name.

```
//script_a.js
function run( ) {
    console.log( "I'm running from script_a.js!" );
}
$( run );

//script_b.js
function run( ) {
    console.log( "I'm running from script_b.js!" );
}
$( run );
```

This can lead to confusion, and when many files are minified and crammed together it causes a problem; `script_a` declares a global variable, which is then declared again in `script_b` and, on running the code, we see the following on the console:

```
> I'm running from script_b.js!
> I'm running from script_b.js!
```

The most common method to get around this and to limit the pollution of the global scope is to wrap our files with an anonymous function, as shown:

```
//script_a.js
(function( $, undefined ) {
    function run( ) {
        console.log( "I'm running from script_a.js!" );
    }
    $( run );
})( jQuery );

//script_b.js
(function( $, undefined ) {
    function run( ) {
        console.log( "I'm running from script_b.js!" );
    }
    $( run );
})( jQuery );
```

Now when we run this, it works as expected:

```
> I'm running from script_a.js!
> I'm running from script_b.js!
```

This is good for code that isn't depended upon externally, but what do we do for the code that is? We just *export* it, right?

Something similar to the following code will do:

```
(function( undefined ) {
    function Logger(){
    }
    Logger.prototype.log = function( message /*...*/ ){
        console.log.apply( console, arguments );
    }
    this.Logger = Logger;
})( )
```

Now, when we run this script, we can access Logger from the global scope:

```
var logger = new Logger( );
logger.log( "This", "is", "pretty", "cool" )
> This is pretty cool
```

So now we can share our libraries and everything is good; But what if someone else already has a library that exposes the same `Logger` class.

What does `node` do to solve this issue? Hello require!

Node.js has a simple way to bring in scripts and modules from external sources, comparable to require in PHP.

Lets create a few files in this structure:

```
/example-2
    /util
        index.js
        logger.js
    main.js

/* util/index.js */
var logger = new Logger( )
var util = {
    logger: logger
};

/* util/logger.js */

function Logger(){
}
Logger.prototype.log = function( message /*...*/ ){
    console.log.apply( console, arguments );
};

/* main.js */
util.logger.log( "This is pretty cool" );
```

We can see that `main.js`. is dependent on `util/index.js`, which is in turn dependent on `util/logger.js`.

This should just work right? Maybe not. Let's run the command:

```
[~/src/examples/example-2]$ node main.js
ReferenceError: logger is not defined
    at Object.<anonymous> (/Users/fabian/examples/example-2/main.js:1:63)
    /* Removed for simplicity */
    at Node.js:814:3
```

So why is this? Shouldn't they be sharing the same global scope? Well, in Node.js the story is a bit different. Remember those anonymous functions that we were wrapping our files in earlier? Node.js wraps our scripts in them automatically and this is where require fits in.

Lets fix our files, as shown:

```
/* util/index.js */
Logger = require( "./logger" )

/* main.js */
util = require( "./util" );
```

If you notice, I didn't use `index.js` when requiring `util/index.js`; the reason for this is that when you a require a folder rather than a file you can specify an index file that can represent that folder's code. This can be handy for something such as a model folder where you expose all your models in one require rather than having a separate require for each model.

So now, we have required our files. But what do we get back?

```
[~/src/examples/example-2]$ node
> var util = require( "./util" );
> console.log( util );
{}
```

Still, there is no logger. We have missed an important step; we haven't told Node.js what we want to expose in our files.

To expose something in Node.js, we use an object called `module.exports`. There is a shorthand reference to it that is just *exports*. When our file is wrapped in an anonymous function, both *module* and *exports* are passed as a parameter, as shown in the following example:

```
function Module( ) {
    this.exports = { };
}

function require( file ) {
    // .....
    returns module.exports;
}

var module = new Module( );
var exports = module.exports;

(function( exports, require, module ) {
    exports = "Value a"
    module.exports = "Value b"
})( exports, require, module );
console.log( module.exports );
// Value b
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

The example shows that *exports* is initially just a reference to `module.exports`. This means that, if you use `exports = { }`, the value you set it as won't be accessible outside the function's scope. However, when you add properties to an *exports* object, you are actually adding properties to the `module.exports` object as they are both the same value. Assigning a value to `module.exports` will export that value because it is accessible outside the function's scope through the module.

With this knowledge, we can finally run our script in the following manner:

```
/* util/index.js */
Logger = require( "./logger.js" );
exports.logger = new Logger( );

/* util/logger.js */
function Logger( ){
}
Logger.prototype.log = ( message /*...*/ ) {
    console.log.apply( console, arguments );
};
module.exports = Logger;

/* main.js */
util = require( "./utils" );
util.logger.log( "This is pretty cool" );
```

Running `main.js`:

```
[~/src/examples/example-2]$ node main.js
This is pretty cool
```

Require can also be used to include modules in our code. When requiring modules, we don't need to use a file path, we just need the name of the `node` module that we want.

Node.js includes many prebuilt core modules, one of which is the `util` module. You can find details on the `util` module at https://nodejs.org/api/util.html.

Let's see the `util` module command:

```
[~]$ node
> var util = require( "util" )
> util.log( 'This is pretty cool as well' )
01 Jan 00:00:00 - This is pretty cool as well
```

Hello npm

Along with internal modules there is also an entire ecosystem of packages; the most common package manager for Node.js is `npm`. At the time of writing, there are a total of 192,875 packages available.

We can use `npm` to access packages that do many things for us, from routing HTTP requests to building our projects. You can also browse the packages available at https://www.npmjs.com/.

Using a package manager you can bring in other modules, which is great as you can spend more time working on your business logic rather than reinventing the wheel.

Let’s download the following package to make our log messages colorful:

```
[~/src/examples/example-3]$ npm install chalk
```

Now, to use it, create a file and require it:

```
[~/src/examples/example-3]$ touch index.js
/* index.js */
var chalk = require( "chalk" );
console.log( "I am just normal text" )
console.log( chalk.blue( "I am blue text!" ) )
```

On running this code, you will see the first message in a default color and the second message in blue. Let’s look at the command:.

```
[~/src/examples/example-3]$ node index.js
I am just normal text
I am blue text!
```

Having the ability to download existing packages comes in handy when you require something that someone else has already implemented. As we said earlier, there are many packages out there to choose from.

We need to keep track of these dependencies and there is a simple solution to that: `package.json`.

Using `package.json` we can define things, such as the name of our project, what the main script is, how to run tests, our dependencies, and so on. You can find a full list of properties at https://docs.npmjs.com/files/package.json.

`npm` provides a handy command to create these files and it will ask you the relevant questions needed to create your `package.json` file:

```
[~/src/examples/example-3]$ npm init
```

The preceding utility will walk you through the creation of a `package.json` file.

It only covers the most common items and tries to guess valid defaults.

Run the `npm help json` command for definitive documentation on these fields and to know what they do exactly.

Afterwards, use `npm` and install `<pkg> --save` to install a package and save it as a dependency in the `package.json` file.

Press `^C` to quit at any time:

```
name: (example-3)
version: (1.0.0)
description:
entry point: (main.js)
test command:
git repository:
keywords:
license: (ISC)
About to write to /examples/example-3/package.json:
{
  "name": "example-3",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "....",
  "license": "ISC"
}
Is this ok? (yes)
```

The utility will provide you with default values, so it is easier to just skip through them using the *Enter* key.

Now when installing our package we can use the `--save` option to save `chalk` as a dependency, as shown:

```
[~/src/examples/example-3]$ npm install --save chalk
```

We can see chalk has been added:

```
[~/examples/example-3]$ cat package.json
{
  "name": "example-3",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "...",
  "license": "ISC",
  "dependencies": {
    "chalk": "^1.0.0"
  }
}
```

We can add these dependencies manually by modifying `package.json`; this is the most common method to save dependencies on installation.

You can read more about the package file at: https://docs.npmjs.com/files/package.json.

If you are creating a server or an application rather than a module, you most likely want to find a way to start your process without having to give a path to your main file all the time; this is where the script object in your `package.json` file comes into play.

To set your start up script, you just need to set the `start` property in the `scripts` object, as shown:

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
}
```

Now, all we need to do is run `npm` start and then `npm` will run the start script we have already specified.

We can define more scripts, for example if we want a start script for the development environment we can also define a development property; with non-standard script names however, instead of just using `npm <script>`, we need to use `npm run <script>`. For example, if we want to run our new development script we will have to use `npm run development`.

`npm` has scripts that are triggered at different times. We can define a `postinstall` script that runs after we run `npm install`; we can use this if we want to trigger a package manager to install the modules (for example, bower)

You can read more about the scripts object here: https://docs.npmjs.com/misc/scripts.

You need to define a package if you are working in a team of developers where the project is to be installed on different machines. If you are using a source control tool such as **git**, it is recommended that you add the `node_modules` directory into your ignore file, as shown:

```
[~/examples/example-3]$ echo "node_modules" > .gitignore
[~/examples/example-3]$ cat .gitignore
node_modules
```

Summary

That was quick, wasn’t it? We have covered the fundamentals of Node.js, which we need to continue on our journey.

We have covered how easy it is to expose and protect public and private code compared to regular JavaScript code in the browser, where the global scope can get very polluted.

We also know how to include packages and code from external sources and how to ensure that the packages included are consistent.

As you can see there is a huge ecosystem of packages in one of the many package managers, such as `npm`, just waiting for us to use and consume.

In the next chapter, we will focus on creating a simple server to route, authenticate, and consume requests.

Chapter 2. Simple HTTP

Now that we have understood the basics, we can move on to something a bit more useful. In this chapter, we will look at creating an HTTP server and routing requests. While working with Node.js you will come across HTTP very often, as server side scripting is one of the common uses of Node.js.

Node.js comes with a built in HTTP server; all you need to do is require the included `http` package and create a server. You can read more about the package at https://nodejs.org/api/http.html.

```
var Http = require( 'http' );

var server = Http.createServer( );
```

This will create your very own HTTP server that is ready to roll. In this state, though, it won't be listening for any requests. We can start listening on any port or socket we wish, as long as it is available, as shown:

```
var Http = require( 'http' );

var server = Http.createServer( );
server.listen( 8080, function( ) {
    console.log( 'Listening on port 8080' );
});
```

Let's save the preceding code to `server.js` and run it:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
```

By navigating to `http://localhost:8080/` on your browser you will see that the request has been accepted but the server isn't responding; this is because we haven't handled the requests yet, we are just listening for them.

When we create the server we can pass a callback that will be called each time there is a request. The parameters passed will be: `request`, `response`.

```
function requestHandler( request, response ) {
}
var server = Http.createServer( requestHandler );
```

Now each time we get a request we can do something:

```
var count = 0;
function requestHandler( request, response ) {
    var message;
    count += 1;
    response.writeHead( 201, {
        'Content-Type': 'text/plain'
    });

    message = 'Visitor count: ' + count;
    console.log( message );
    response.end( message );
}
```

Let's run the script and request the page from the browser; you should see `Visitor count: 1` returned to the browser:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
Visitor count: 1
Visitor count: 2
```

Something weird has happened though: an extra request gets generated. Who is visitor 2?

The `http.IncomingMessage` (the parameter `request`) *exposes* a few properties that can be used to figure this out. The property we are most interested in right now is `url`. We are expecting just `/` to be requested, so let's add this to our message:

```
message = 'Visitor count: ' + count + ', path: ' + request.url;
```

Now you can run the code and see what's going on. You will notice that `/favicon.ico` has been requested as well. If you are not able to see this then you must be wondering what I have been going on about or if your browser has been to `http://localhost:8080` recently and has a cached icon already. If this is the case, then you can request the icon manually, for example from `http://localhost:8080/favicon.ico`:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
Visitor count: 1, path: /
Visitor count: 2, path: /favicon.ico
```

We can also see that if we request any other page we will get the correct path, as shown:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
Visitor count: 1, path: /
Visitor count: 2, path: /favicon.ico
Visitor count: 3, path: /test
Visitor count: 4, path: /favicon.ico
Visitor count: 5, path: /foo
Visitor count: 6, path: /favicon.ico
Visitor count: 7, path: /bar
Visitor count: 8, path: /favicon.ico
Visitor count: 9, path: /foo/bar/baz/qux/norf
Visitor count: 10, path: /favicon.ico
```

This isn't the desired outcome though, for everything but a few routes we want to return `404: Not Found`.

Introducing routing

Routing is essential for almost all Node.js servers. First, we will implement our own simple version and then move on to the more complex rounting.

We can implement our own simple router using a `switch` statement, such as:

```
function requestHandler( request, response ) {
    var message,
        status = 200;

    count += 1;


    switch( request.url ) {
        case '/count':
            message = count.toString( );
            break;
        case '/hello':
            message = 'World';
            break;
        default:
            status = 404;
            message = 'Not Found';
            break;
    }

    response.writeHead( 201, {
        'Content-Type': 'text/plain'
    });
    console.log( request.url, status, message );
    response.end( message );
}
```

Let’s run the following example:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
/foo 404 Not Found
/bar 404 Not Found
/world 404 Not Found
/count 200 4
/hello 200 World
/count 200 6
```

You can see the count increasing with each request; however, it isn’t returned each time. If we haven’t defined a case specifically for that route, we return `404: Not Found`.

For services that implement a RESTful interface, we want to be able to route requests based on the HTTP method as well. The request object exposes this using the `method` property.

Adding this to the log we can see this:

```
console.log( request.method, request.url, status, message );
```

Run the example and execute your requests, you can use a REST client to invoke a POST request:

```
[~/examples/example-4]$ node server.js
Listening on port 8080
GET /count 200 1
POST /count 200 2
PUT /count 200 3
DELETE /count 200 4
```

We can implement a router to route based on a method, but there are packages that do this for us already out there. For now we will use a simple package called `router`:

```
[~/examples/example-5]$ npm install router
```

Now, we can do some more complex routing of our requests:

Let’s create a simple RESTful interface.

First, we need to create the server, as shown:

```
/* server.js */
var Http = require( 'http' ),
    Router = require( 'router' ),
    server,
    router;

router = new Router( );

server = Http.createServer( function( request, response ) {
    router( request, response, function( error ) {
        if( !error ) {
            response.writeHead( 404 );
        } else {
            //Handle errors
            console.log( error.message, error.stack );
            response.writeHead( 400 );
        }
        response.end( '\n' );
    });
});

server.listen( 8080, function( ) {
    console.log( 'Listening on port 8080' );
});
```

Running the server should show that the server is listening.

```
[~/examples/example-5]$ node server.js
Listening on port 8080
```

We want to define a simple interface to read, save, and delete messages. We might want to read individual messages as well as a list of messages; this essentially defines a set of RESTful endpoints.

REST stands for **R**epresentational **S**tate **T**ransfer; it is a very simple and common style used by many HTTP programming interfaces.

The endpoints we want to define are:

HTTP Method	**Endpoint**	**Used to**
`POST`	`/message`	Create message
`GET`	`/message/:id`	Read message
`DELETE`	`/message/:id`	Delete message
`GET`	`/message`	Read multiple messages

For each HTTP method, the router has a method to use for mapping a route. This interface is in the form of:

```
router.<HTTP method>( <path>, [ ... <handler> ] )
```

We can define multiple handlers for each route, but we will come back to that in a moment.

We will go through each route, create an implementation, and append the code to the end of `server.js`.

We want to store our messages somewhere, and in the real world we will store them in a database; however, for simplicity we will use an array with a simple counter, as shown:

```
var counter = 0,
    messages = { };
```

Our first route will be used to create messages:

```
function createMessage( request, response ) {
    var id = counter += 1;
    console.log( 'Create message', id );
    response.writeHead( 201, {
        'Content-Type': 'text/plain'
    });
    response.end( 'Message ' + id );
}
router.post( '/message', createMessage );
```

We can ensure that this route works by running the server and doing a POST request to `http://localhost:8000/message`.

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Create message 1
Create message 2
Create message 3
```

We can also confirm that the counter is incrementing, as the id increases each time we make a request. We will do this to keep a track of the count of messages and to give a *unique* id to each message.

Now that this is working, we need to be able to read the message text and to do this we need to be able to read the request body that was sent by the client. This is where multiple handlers come into play. We could tackle this in two different ways, if we were reading the body in only one route or if we were doing some other action specific to a route, for instance authorization, we will add an additional handler to the route, such as:

```
router.post( '/message', parseBody, createMessage )
```

The other way we could do it is by adding a handler for all methods and routes; this will be executed first before the route handlers, these are commonly referred to as middleware. You can think of handlers as being a chain of functions where each one is calling the next, once it is finished with its tasks. With this in mind you should note that the order in which you add a handler, both middleware and route, will dictate the order of operations. This means that, if we are registering a handler that is executed for all methods, we must do this first.

The router *exposes* a function to add the following handlers:

```
router.use( function( request, response, next ) {
    console.log( 'middleware executed' );
    // Null as there were no errors
    // If there was an error then we could call `next( error );`
    next( null );
});
```

You can add this code just above your implementation of `createMessage`:

Once you have done that, run the server and make the following request:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
middleware executed
Create message 1
```

You can see that the middleware gets executed before the route handler.

Now that we know how middleware works, we can use them as follows:

```
[~/examples/example-5]$ npm install body-parser
```

Replace our custom middleware with:

```
var BodyParser = require( 'body-parser' );
router.use( BodyParser.text( ) );
```

At this stage, we just want to read all requests as plain text.

Now we can retrieve the message in `createMessage`:

```
function createMessage( request, response ) {
    var id = counter += 1,
        message = request.body;

    console.log( 'Create message', id, message );
    messages[ id ] = message;
    response.writeHead( 201, {
        'Content-Type': 'text/plain',
        'Location': '/message/' + id
    });
    response.end( message );
}
```

Run `server.js` and `POST` a couple of messages to `http://localhost:8080/message`; you will see something similar to these messages:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Create message 1 Hello foo
Create message 2 Hello bar
```

If you notice, you will see that a header returns with a new location of the message and its id, If we request `http://localhost:8080/message/1`, the content from the first message should be returned.

However, there is something different with this route; it has a key that is generated each time a message is created. We don't want to set up a new route for each new message as it will be highly inefficient. Instead, we create a route that matches a pattern, such as `/message/:id`. This is a common way to define a dynamic route in Node.js.

The `id` part of the route is called a parameter. We can define as many of these as we want in our route and refer them using the request; for example we can have a route similar to `/user/:id/profile/:attribute`.

With this in mind we can create our `readMessage` handler, as shown:

```
function readMessage( request, response ) {
    var id = request.params.id,
        message = messages[ id ];
    console.log( 'Read message', id, message );

    response.writeHead( 200, {
        'Content-Type': 'text/plain'
    });
    response.end( message );
}
router.get( '/message/:id', readMessage );
```

Now let's save the preceding code in the `server.js` file and run the server:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Create message 1 Hello foo
Read message 1 Hello foo
Create message 2 Hello bar
Read message 2 Hello bar
Read message 1 Hello foo
```

We can see it's working by sending a few requests to the server.

Deleting messages is almost the same as reading them; but we don't return anything and null out the original message value:

```
function deleteMessage( request, response ) {
    var id = request.params.id;

    console.log( 'Delete message', id );

    messages[ id ] = undefined;

    response.writeHead( 204, { } );

    response.end( '' );
}

router.delete( '/message/:id', deleteMessage )
```

First, run the server, then create, read, and delete a message, as shown:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Delete message 1
Create message 1 Hello
Read message 1 Hello
Delete message 1
Read message 1 undefined
```

That looks good; however, we have run into a problem. We shouldn't be able to read a message again after deleting it; we will return `404` in both the read and delete handlers if we can't find a message. We can do this by adding the following code to our read and delete handlers:

```
    var id = request.params.id,
        message = messages[ id ];

    if( typeof message !== 'string' ) {
        console.log( 'Message not found', id );

        response.writeHead( 404 );
        response.end( '\n' );
        return;
    }
```

Now let's save the preceding code in the `server.js` file and run the server:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Message not found 1
Create message 1 Hello
Read message 1 Hello
```

Lastly, we want to be able to read all messages and return a list of all message values:

```
function readMessages( request, response ) {
    var id,
        message,
        messageList = [ ],
        messageString;

    for( id in messages ) {
        if( !messages.hasOwnProperty( id ) ) {
            continue;
        }
        message = messages[ id ];
        // Handle deleted messages
        if( typeof message !== 'string' ) {
            continue;
        }
        messageList.push( message );
    }

    console.log( 'Read messages', JSON.stringify(
        messageList,
        null,
        '  '
    ));

    messageString = messageList.join( '\n' );

    response.writeHead( 200, {
        'Content-Type': 'text/plain'
    });

    response.end( messageString );
}
router.get( '/message', readMessages );
```

Now let's save the preceding code in the `server.js` file and run the server:

```
[~/examples/example-5]$ node server.js
Listening on port 8080
Create message 1 Hello 1
Create message 2 Hello 2
Create message 3 Hello 3
Create message 4 Hello 4
Create message 5 Hello 5
Read messages [
  "Hello 1",
  "Hello 2",
  "Hello 3",
  "Hello 4",
  "Hello 5"
]
```

Awesome; now we have a full RESTful interface to read and write messages. But, we don't want everyone to be able to read our messages; they should be secure and we also want to know who is creating the messages, we will cover this in the next chapter.

Summary

Now we have everything we need to make some pretty cool services. We can now create an HTTP from scratch, route our requests, and create a RESTful interface.

This will help you with the creation of complete Node.JS services. In the next chapter, we will cover authentication.

Chapter 3. Authentication

We can now create RESTful APIs, but we don’t want everyone to access everything we expose. We want the routes to be secure and to be able to track who is doing what.

Passport is a great module and another middleware that helps us authenticate requests.

Passport exposes a simple API for providers to expand on and create strategies to authenticate users. At the time of writing, there are 307 officially supported strategies; however, there is no reason why you can’t write your own strategy and publish it for others to use.

Basic authentication

The simplest strategy for passport is the local strategy that accepts a username and password.

We will introduce the express framework for these examples and, now that you know the basics of how it all works underneath, we can put it all together.

You can install `express`, `body-parser`, `passport`, and `passport-local`. Express is a batteries-included Web framework for Node.js, and includes routing and the ability to use middleware:

```
[~/examples/example-19]$ npm install express body-parser passport passport-local
```

For now, we can store our users in a simple object to reference later, as shown:

```
var users = {
    foo: {
        username: 'foo',
        password: 'bar',
        id: 1
    },
    bar: {
        username: 'bar',
        password: 'foo',
        id: 2
    }
}
```

Once we have a few users, we need to set up passport. When we create an instance of the local strategy, we need to provide a `verify` callback where we check the username and password, while returning a user:

```
var Passport = require( 'passport' ),
    LocalStrategy = require( 'passport-local' ).Strategy;

var localStrategy = new LocalStrategy({
    usernameField: 'username',
    passwordField: 'password'
  },
  function(username, password, done) {
    user = users[ username ];

    if ( user == null ) {
        return done( null, false, { message: 'Invalid user' } );
    }

    if ( user.password !== password ) {
        return done( null, false, { message: 'Invalid password' } );
    }

    done( null, user );
  }
)
```

The `verify` callback in this case is expecting `done` to be called with a user. It also allows us to provide information if the user was invalid or the password was wrong.

Now, that we have a strategy we can pass this to passport, which allows us to reference it later and use it to authenticate our requests, as follows:

```
Passport.use( 'local', localStrategy );
```

You can use multiple strategies per application and reference each one by the name you passed, in this case `'local'`.

Now, let's create our server, as shown here:

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

var app = Express( );
```

We will have to use the `body-parser` middleware. This will ensure that, when we post to our login route, we can read our body; we also need to initialize passport:

```
var BodyParser = require( 'body-parser' );
app.use( BodyParser.urlencoded( { extended: false } ) );
app.use( BodyParser.json( ) );
app.use( Passport.initialize( ) );
```

To login to our application, we need to create a `post` route that uses authentication as one of the handlers. The code for this is as follows:

```
app.post(
    '/login',
    Passport.authenticate( 'local', { session: false } ),
    function ( request, response ) {

    }
);
```

Now, when we send a `POST` request to `/login` the server will authenticate our requests.

Once authenticated, the `user` property will be populated on the request object, as follows:

```
app.post(
    '/login',
    Passport.authenticate( 'local', { session: false } ),
    function ( request, response ) {
        response.send( 'User Id ' + request.user.id );
    }
);
```

Lastly, we need to listen for requests, as with all the other servers:

```
app.listen( 8080, function( ) {
    console.log( 'Listening on port 8080' );
});
```

Lets run the example:

```
[~/examples/example-19]$ node server.js
Listening on port 8080
```

Now, we can authenticate users when we send a `POST` request at our server. If the user hasn't passed both the username and password the server will return `400 Bad Request`.

Tip

If you aren't familiar with `curl` you could use a tool, such as Advanced REST Client:

https://chromerestclient.appspot.com/

In the following examples I will be using the command line interface `curl`.

We can execute a login request by executing a `POST` to `/login` command:

```
[~]$ curl -X POST http://localhost:8080/login -v
< HTTP/1.1 400 Bad Request
```

If the user provides the wrong details then `401 Unauthorized` will be returned:

```
[~]$ curl -X POST http://localhost:8080/login \
        -H 'Content-Type: application/json' \
        -d '{"username":"foo","password":"foo"}' \
        -v
< HTTP/1.1 401 Unauthorized
```

If we provide the correct details then we can see our handler was called and the correct data was returned:

```
[~]$ curl -X POST http://localhost:8080/login \
        -H 'Content-Type: application/json' \
        -d '{"username":"foo","password":"bar"}'
User Id 1
[~]$ curl -X POST http://localhost:8080/login \
        -H 'Content-Type: application/json' \
        -d '{"username":"bar","password":"foo"}'
User Id 2
```

Bearer tokens

Now that we have an authenticated user, we can generate a token that can be used with the rest of our requests rather than passing our username and password everywhere. This is commonly known as a Bearer token and, conveniently, there is a passport strategy for this.

For our tokens, we will use something called a **JSON Web Token** (**JWT**). JWT allows us to encode tokens from JSON objects and then decode them and verify them. The data stored in them is open and simple to read, so passwords shouldn't be stored in them; however, it makes verifying a user very simple. We can also provide these tokens with expiry dates, which helps limit the severity of tokens being exposed.

You can read more about JWT at http://jwt.io/.

We can install JWT using the following command:

```
[~/examples/example-19]$ npm install jsonwebtoken
```

Once a user is authenticated, we can safely provide them with a token to use in future requests:

```
var JSONWebToken = require( 'jsonwebtoken' ),
    Crypto = require( 'crypto' );

var generateToken = function ( request, response ) {

    // The payload just contains the id of the user
    // and their username, we can verify whether the claim
    // is correct using JSONWebToken.verify
    var payload = {
        id: user.id,
        username: user.username
    };
    // Generate a random string
    // Usually this would be an app wide constant
    // But can be done both ways
    var secret = Crypto.randomBytes( 128 )
                       .toString( 'base64' );
    // Create the token with a payload and secret
    var token = JSONWebToken.sign( payload, secret );

    // The user is still referencing the same object
    // in users, so no need to set it again
    // If we were using a database, we would save
    // it here
    request.user.secret = secret

    return token;
}

var generateTokenHandler = function ( request, response  ) {
    var user = request.user;
    // Generate our token
    var token = generateToken( user );
    // Return the user a token to use
    response.send( token );
};

app.post(
    '/login',
    Passport.authenticate( 'local', { session: false } ),
    generateTokenHandler
);
```

Now, when the user logs in they will be presented with a token to use that we can verify.

Lets run our Node.js server:

```
[~/examples/example-19]$ node server.js
Listening on port 8080
```

When we login now we receive a token:

```
[~]$ curl -X POST http://localhost:8080/login \
            -H 'Content-Type: application/json' \
            -d '{"username":"foo","password":"bar"}'
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZC
I6MSwidXNlcm5hbWUiOiJmb28iLCJpYXQiOjE0MzcyO
TQ3OTV9.iOZO7oCIceZl6YvZqVP9WZLRx-XVvJFMF1p
pPCEsGGs
```

We can enter this into the debugger at http://jwt.io/ and see the contents, as shown:

```
{
  "id": 1,
  "username": "foo",
  "iat": 1437294795
}
```

If we had the secret we could verify that the token is correct. The signature changes every time we request a token:

```
[~]$ curl -X POST http://localhost:8080/login \
                -H 'Content-Type: application/json' \
                -d '{"username":"foo","password":"bar"}'
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZC
I6MSwidXNlcm5hbWUiOiJmb28iLCJpYXQiOjE0MzcyO
TQ5OTl9.n1eRQVOM9qORTIMUpslH-ycTNEYdLDKa9lU
pmhf44s0
```

We can authenticate a user using `passport-bearer`; it is set up very similar to `passport-local`. However, rather than accepting a username and password from the body, we accept a bearer token; this can be passed using the query string, body, or the `Authorization` header:

First we must install `passport-http-bearer`:

```
[~/examples/example-19]$ npm install passport-http-bearer
```

The let’s create our verifier. There are two steps: the first is ensuring the decoded information matches our user, this will be where we usually retrieve our user; then’ once we have a user and it’s valid, we can check whether the token is valid based on the user’s secret:

```
var BearerStrategy = require( 'passport-http-bearer' ).Strategy;

var verifyToken = function( token, done ) {
    var payload = JSONWebToken.decode( token );
    var user = users[ payload.username ];
    // If we can't find a user, or the information
    // doesn't match then return false
    if ( user == null ||
         user.id !== payload.id ||
         user.username !== payload.username ) {
        return done( null, false );
    }
    // Ensure the token is valid now we have a user
    JSONWebToken.verify( token, user.secret, function ( error, decoded ) {
        if ( error || decoded == null ) {
            return done( error, false );
        }
        return done( null, user );
    });
}
var bearerStrategy = new BearerStrategy(
    verifyToken
)
```

We can register this strategy as the bearer so we can use it later:

```
Passport.use( 'bearer', bearerStrategy );
```

We can create a simple route where we retrieve user details for an authenticated user:

```
app.get(
    '/userinfo',
    Passport.authenticate( 'bearer', { session: false } ),
    function ( request, response ) {
        var user = request.user;
        response.send( {
            id: user.id,
            username: user.username
        });
    }
);
```

Let’s run the Node.js server:

```
[~/examples/example-19]$ node server.js
Listening on port 8080
```

Once we receive a token:

```
[~]$ curl -X POST http://localhost:8080/login \
        -H 'Content-Type: application/json' \
        -d '{"username":"foo","password":"bar"}'
```

We can use the result in our requests:

```
[~]$ curl -X GET http://localhost:8080/userinfo \
        -H 'Authorization: Bearer <token>'
{"id":1,"username":"foo"}
```

OAuth

OAuth provides many advantages; for instance, it does not need to deal with the actual identification of users. We can let users login using services they trust, such as Google, Facebook, or Auth0.

For the following examples, I will be using `Auth0`. They provide a free account for you to get up-and-running: https://auth0.com/.

You will need to sign up and create an `api` (choose `AngularJS + Node.js`), then go to Settings and take down the domain, client id, and client secret. You will need these to set up `OAuth`.

We can authenticate using OAuth using `passport-oauth2`:

```
[~/examples/example-19]$ npm install --save passport-oauth2
```

As with our bearer tokens, we want to validate what the server returns, which will be a user object that has an id. We will match this with a user that is in our data or create a new user:

```
var validateOAuth = function ( accessToken, refreshToken, profile, done ) {

    var keys = Object.keys( users ), user = null;

    for( var iKey = 0; iKey < keys.length; i += 1 ) {
        user = users[ key ];
        if ( user.thirdPartyId !== profile.user_id ) { continue; }
        return done( null, user );
    }

    users[ profile.name ] = user = {
        username: profile.name,
        id: keys.length,
        thirdPartyId: profile.user_id
    }
    done( null, user );

};
```

Once we have a function to validate our users we can put together the options for our OAuth strategy:

```
var oAuthOptions = {
    authorizationURL: 'https://<domain>.auth0.com/authorize',
    tokenURL: 'https://<domain>.auth0.com/oauth/token',
    clientID: '<client id>',
    clientSecret: '<client secret>',
    callbackURL: "http://localhost:8080/oauth/callback"
}
```

Then we create our strategy, as follows:

```
var OAuth2Strategy = require( 'passport-oauth2' ).Strategy;
```

```
oAuthStrategy = new OAuth2Strategy( oAuthOptions, validateOAuth );
```

Before we use our strategy we need to duck type the strategies `userProfile` method with our own, this is so we can request the user object to use in `validateOAuth`:

```
var parseUserProfile = function ( done, error, body ) {
    if ( error ) {
        return done( new Error( 'Failed to fetch user profile' ) )
    }

    var json;
    try {
        json = JSON.parse( body );
    } catch ( error ) {
        return done( error );
    }
    done( null, json );
}

var getUserProfile = function( accessToken, done ) {
    oAuthStrategy._oauth2.get(
        "https://<domain>.auth0.com/userinfo",
        accessToken,
        parseUserProfile.bind( null, done )
    )
}
oAuthStrategy.userProfile = getUserProfile
```

We can register this strategy as `oauth` so we can use it later:

```
Passport.use( 'oauth', oAuthStrategy );
```

We need to create two routes to handle our OAuth authentication: one route to start the flow and the other for the identification server to return to:

```
app.get( '/oauth', Passport.authenticate( 'oauth', { session: false } ) );
```

We can use our `generateTokenHandler` here, as our request will have a user on it.

```
app.get( '/oauth/callback',
  Passport.authenticate( 'oauth', { session: false } ),
  generateTokenHandler
);
```

We can now start our server and request `http://localhost:8080/oauth`; the server will redirect you to `Auth0`. Once logged in, you will receive a token that you can use with `/userinfo`.

If you were using sessions, you could save the user to the session and redirect them back to your front page (or the default page set for a logged in user). For a single-page app, when using something like Angular, you may want to redirect the user with a token in the URL for the client framework to grab onto and save.

Summary

We can now authenticate users; this is great as we can now figure out who the people are and then limit the users to certain resources.

In the next chapter we will cover debugging, we may need to use it if our users aren’t being authenticated.

Chapter 4. Debugging

At some point in your journey with Node.js, it is inevitable that you will have to debug some nasty bugs. So, let’s expect them beforehand and plan for that day.

Logging

There are a few methods that we can use to debug our software; the first one we are going to look at is logging. The simplest way to log a message is to use `console`. In most of the previous examples `console` has been used to portray what is going on without needing to see the entire HTTP request and response, thus making things a lot more readable and simple.

An example of this is:

```
var Http = require( 'http' );

Http.createServer( function( request, response ) {
    console.log(
        'Received request',
        request.method,
        request.url
    )

    console.log( 'Returning 200' );

    response.writeHead( 200, { 'Content-Type': 'text/plain' } );
    response.end( 'Hello World\n' );

}).listen( 8000 );

console.log( 'Server running on port 8000' );
```

Running this example will log requests and responses on the console:

```
[~/examples/example-6]$ node server.js
Server running on port 8000
Received request GET /
Returning 200
Received request GET /favicon.ico
Returning 200
Received request GET /test
Returning 200
Received request GET /favicon.ico
Returning 200
```

If we are using a framework that accepts middleware, such as express, we could use a simple `npm` package called **morgan**; you can find the package at https://www.npmjs.com/package/morgan:

```
[~/examples/example-7]$ npm install morgan
[~/examples/example-7]$ npm install router
```

We can use it by using `require` to bring it into our code and adding it as middleware:

```
var Morgan = require( 'morgan' ),
    Router = require( 'router' ),
    Http = require( 'http' );
```

```
router = new Router( );

router.use( Morgan( 'tiny' ) );

/* Simple server */
Http.createServer( function( request, response ) {
    router( request, response, function( error ) {
        if( !error ) {
            response.writeHead( 404 );
        } else {
            //Handle errors
            console.log( error.message, error.stack );
            response.writeHead( 400 );
        }
        response.end( '\n' );

    });
}).listen( 8000 );

console.log( 'Server running on port 8000' );

function getInfo ( request, response ) {
    var info = process.versions;

    info = JSON.stringify( info );
    response.writeHead( 200, { 'Content-Type': 'application/json' } );
    response.end( info );
}
router.get( '/info', getInfo );
```

When the server is running, we can see each request and response without having to add logging to each handler:

```
[~/examples/example-7]$ node server.js
Server running on port 8000
GET /test 404 - - 4.492 ms
GET /favicon.ico 404 - - 2.281 ms
GET /info 200 - - 1.120 ms
GET /info 200 - - 1.120 ms
GET /test 404 - - 0.199 ms
GET /info 200 - - 0.494 ms
GET /test 404 - - 0.162 ms
```

This kind of logging is a simple way to see what is being used on the server and how long each request is taking. Here, you can see that the first requests took the longest and then they got a lot faster. The difference is only of 3 ms; if the time was larger, it could have been a big problem.

We can increase the information that’s logged by changing the format we pass to morgan, as shown:

```
router.use( Morgan( 'combined' ) );
```

By running the server you will see more information, such as the remote user, date and time of the request, amount of content that was returned, and the client they are using.

```
[~/examples/example-7]$ node server.js
Server running on port 8000
::1 - - [07/Jun/2015:11:09:03 +0000] "GET /info HTTP/1.1" 200 - "-" "--
REMOVED---"
```

Timing is definitely an important factor as it can be helpful when sifting through the mountains of logs that you will obtain. Some bugs can be like a ticking time-bomb waiting to explode at 3 AM on a Saturday night. All these logs mean nothing to us if the process has died and the logs have disappeared. There is another popular and useful package called `bunyan`, which wraps many logging methods into one.

Bunyan brings to the table the advantage of writeable streams to write logs, whether it is a file on disk or `stdout`. This allows us to persist our logs for postmortem debugging. You can find more details about `bunyan` at https://www.npmjs.com/package/bunyan.

Now, let's install the package. We want it installed both locally and globally so that we can also use it as a command line tool:

```
 [~/examples/example-8]$ npm install –g bunyan
 [~/examples/example-8]$ npm install bunyan
```

Now, lets do some logging:

```
var Bunyan = require( 'bunyan' ),
    logger;

logger = Bunyan.createLogger( {
    name: 'example-8'
});
logger.info( 'Hello logging' );
```

Running our example:

```
[~/examples/example-8]$ node index.js
{"name":"example-
8","hostname":"macbook.local","pid":2483,"level":30,"msg":"Hello
logging","time":"2015-06-07T11:35:13.973Z","v":0}
```

This doesn't look very pretty, does it? Bunyan uses a simple structured JSON string to save messages; this makes it easy to parse, extend, and read. Bunyan comes with a CLI utility to make everything nice and pretty.

If we run the example with the utility, then we will see that the output is nicely formatted:

```
[~/examples/example-8]$ node index.js | bunyan
[2015-06-07T11:38:59.698Z]  INFO: example-8/2494 on macbook.local: Hello
logging
```

If we add a few more levels, you will see on your console that each is colored differently to help us identify them:

```
var Bunyan = require( 'bunyan' ),
    logger;
logger = Bunyan.createLogger( {
    name: 'example-8'
});
```

```
logger.trace( 'Trace' );
logger.debug( 'Debug' );
logger.info( 'Info' );
logger.warn( 'Warn' );
logger.error( 'Error' );
logger.fatal( 'Fatal' );

logger.fatal( 'We got a fatal, lets exit' );
process.exit( 1 );
```

Let’s run the example:

```
[~/examples/example-8]$ node index.js | bunyan
[2015-06-07T11:39:55.801Z]  INFO: example-8/2512 on macbook.local: Info
[2015-06-07T11:39:55.811Z]  WARN: example-8/2512 on macbook.local: Warn
[2015-06-07T11:39:55.814Z] ERROR: example-8/2512 on macbook.local: Error
[2015-06-07T11:39:55.814Z] FATAL: example-8/2512 on macbook.local: Fatal
[2015-06-07T11:39:55.814Z] FATAL: example-8/2512 on macbook.local: We got a
fatal, lets exit
```

If you notice, trace and debug weren’t outputted on the console. This is because they are used to follow the flow of the program rather than the key information and are usually very noisy.

We can change the level of logs we want to see by passing this as an option when we create the logger:

```
logger = Bunyan.createLogger( {
    name: 'example-8',
    level: Bunyan.TRACE
});
```

Now, when we run the example:

```
[~/examples/example-8]$ node index.js | bunyan
[2015-06-07T11:55:40.175Z] TRACE: example-8/2621 on macbook.local: Trace
[2015-06-07T11:55:40.177Z] DEBUG: example-8/2621 on macbook.local: Debug
[2015-06-07T11:55:40.178Z]  INFO: example-8/2621 on macbook.local: Info
[2015-06-07T11:55:40.178Z]  WARN: example-8/2621 on macbook.local: Warn
[2015-06-07T11:55:40.178Z] ERROR: example-8/2621 on macbook.local: Error
[2015-06-07T11:55:40.178Z] FATAL: example-8/2621 on macbook.local: Fatal
[2015-06-07T11:55:40.178Z] FATAL: example-8/2621 on macbook.local: We got a
fatal, lets exit
```

We usually don’t want to see logs that are lower than the info level, as any information that is useful for post-mortem debugging should have been logged using the info or higher.

Bunyan’s api is good for the function of logging errors and objects. It saves the correct structures in its JSON output, which is ready for display:

```
try {
    ref.go( );
} catch ( error ) {
    logger.error( error );
}
```

Let’s run the example:

```
[~/examples/example-9]$ node index.js | bunyan
[2015-06-07T12:00:38.700Z] ERROR: example-9/2635 on macbook.local: ref is
not defined
    ReferenceError: ref is not defined
        at Object.<anonymous> (~/examples/example-8/index.js:9:2)
        at Module._compile (module.js:460:26)
        at Object.Module._extensions..js (module.js:478:10)
        at Module.load (module.js:355:32)
        at Function.Module._load (module.js:310:12)
        at Function.Module.runMain (module.js:501:10)
        at startup (node.js:129:16)
        at node.js:814:3
```

If we look at the example and pretty-print it, we will see that they save it as an error:

```
[~/examples/example-9]$ npm install -g prettyjson
[~/examples/example-9]$ node index.js | prettyjson
name:     example-9
hostname: macbook.local
pid:      2650
level:    50
err:
  message: ref is not defined
  name:    ReferenceError
  stack:
    """
      ReferenceError: ref is not defined
          at Object.<anonymous> (~/examples/example-8/index.js:9:2)
          at Module._compile (module.js:460:26)
          at Object.Module._extensions..js (module.js:478:10)
          at Module.load (module.js:355:32)
          at Function.Module._load (module.js:310:12)
          at Function.Module.runMain (module.js:501:10)
          at startup (node.js:129:16)
          at node.js:814:3
    """
msg:      ref is not defined
time:     2015-06-07T12:02:33.875Z
v:        0
```

This is useful because, if you just log an error, you will either get an empty object if you used `JSON.stringify` or just the message if you used `toString`:

```
try {
    ref.go( );
} catch ( error ) {
    console.log( JSON.stringify( error ) );
    console.log( error );
    console.log( {
        message: error.message
        name: error.name
        stack: error.stack
    });
}
```

Let’s run the example:

```
[~/examples/example-10]$ node index.js
{}
[ReferenceError: ref is not defined]
{ message: 'ref is not defined',
  name: 'ReferenceError',
  stack: '--REMOVED--' }
```

It is to lot simpler and cleaner to use `logger.error( error )` than `logger.error( { message: error.message /*, ... */ } );`.

As mentioned earlier, `bunyan` uses the concept of streams, which means that we can write to a file, `stdout`, or any other service we wish to extend to.

To write to a file, all we need to do is add it to the options passed to `bunyan` at setup:

```
var Bunyan = require( 'bunyan' ),
    logger;

logger = Bunyan.createLogger( {
    name: 'example-11',
    streams: [
        {
            level: Bunyan.INFO,
            path: './log.log'
        }
    ]
});
logger.info( process.versions );
logger.info( 'Application started' );
```

By running the example, you won’t see any logs being outputted to the console but they will be written to file instead:

```
 [~/examples/example-11]$ node index.js
```

If you list what’s in the directory you will see a new file has been created:

```
[~/examples/example-11]$ ls
index.js     log.log      node_modules
```

If you read what’s in the file you will see that the logs have already been written:

```
[~/examples/example-11]$ cat log.log
{"name":"example-
11","hostname":"macbook.local","pid":3614,"level":30,"http_parser":"2.3","n
ode":"0.12.2","v8":"3.28.73","uv":"1.4.2-
node1","zlib":"1.2.8","modules":"14","openssl":"1.0.1m","msg":"","time":"20
15-06-07T12:29:46.606Z","v":0}
{"name":"example-
11","hostname":"macbook.local","pid":3614,"level":30,"msg":"Application
started","time":"2015-06-07T12:29:46.608Z","v":0}
```

We can run this through `bunyan` in order to print it out nicely:

```
[~/examples/example-11]$ cat log.log | bunyan
[~/examples/example-11]$ cat log.log | bunyan
[2015-06-07T12:29:46.606Z]  INFO: example-11/3614 on macbook.local:
```

```
(http_parser=2.3, node=0.12.2, v8=3.28.73, uv=1.4.2-node1, zlib=1.2.8,
modules=14, openssl=1.0.1m)
[2015-06-07T12:29:46.608Z]  INFO: example-11/3614 on macbook.local:
Application started
```

Now that we can log to a file, we also want to be able to see the messages as they are displayed. If we were just logging to a file, we could use:

```
[~/examples/example-11]$ tail -f log.log | bunyan
```

This will log to `stdout` as the file it is being written to; alternatively we could just add another stream to `bunyan`:

```
logger = Bunyan.createLogger( {
    name: 'example-11',
    streams: [
        {
            level: Bunyan.INFO,
            path: './log.log'
        },
        {
            level: Bunyan.INFO,
            stream: process.stdout
        }
    ]
});
```

Running the example will display the logs to the console:

```
[~/examples/example-11]$ node index.js | bunyan
 [2015-06-07T12:37:19.857Z] INFO: example-11/3695 on macbook.local:
(http_parser=2.3, node=0.12.2, v8=3.28.73, uv=1.4.2-node1, zlib=1.2.8,
modules=14, openssl=1.0.1m) [2015-06-07T12:37:19.860Z] INFO: example-
11/3695 on macbook.local: Application started
```

We can also see the logs have been appended to the file:

```
[~/examples/example-11]$ cat log.log | bunyan
 [2015-06-07T12:29:46.606Z]  INFO: example-11/3614 on macbook.local:
(http_parser=2.3, node=0.12.2, v8=3.28.73, uv=1.4.2-node1, zlib=1.2.8,
modules=14, openssl=1.0.1m)
[2015-06-07T12:29:46.608Z]  INFO: example-11/3614 on macbook.local:
Application started
[2015-06-07T12:37:19.857Z]  INFO: example-11/3695 on macbook.local:
(http_parser=2.3, node=0.12.2, v8=3.28.73, uv=1.4.2-node1, zlib=1.2.8,
modules=14, openssl=1.0.1m)
[2015-06-07T12:37:19.860Z]  INFO: example-11/3695 on macbook.local:
Application started
```

Great, now we have the logging down, what shall we do with it?

Well, it helps to know where our errors are occurring and it starts to get really messy when you have lots of anonymous functions around the place. If you noticed in the examples that cover an HTTP server, the majority of the functions were named. This is very helpful in tracking down errors when callbacks are involved.

Let’s look at this example:

```
try {
    a = function( callback ) {
        return function( ) {
            callback( );
        };
    };
    b = function( callback ) {
        return function( ) {
            callback( );
        }
    };
    c = function( callback ) {
        return function( ) {
            throw new Error( "I'm just messing with you" );
        };
    };
    a( b( c( ) ) )( );
} catch ( error ) {
    logger.error( error );
}
```

It might look a bit messy and that’s because it is. Let’s run the following example:

```
[~/examples/example-12]$ node index.js | bunyan
 [2015-06-07T12:51:11.665Z] ERROR: example-12/4158 on macbook.local: I'm
just messing with you
    Error: I'm just messing with you
        at /Users/fabian/examples/example-12/index.js:19:10
        at /Users/fabian/examples/example-12/index.js:14:4
        at /Users/fabian/examples/example-12/index.js:9:4
        at Object.<anonymous> (/Users/fabian/examples/example-
12/index.js:22:16)
        at Module._compile (module.js:460:26)
        at Object.Module._extensions..js (module.js:478:10)
        at Module.load (module.js:355:32)
        at Function.Module._load (module.js:310:12)
        at Function.Module.runMain (module.js:501:10)
        at startup (node.js:129:16)
```

You can see that there are no function names in our code and also there is no naming in the stack trace unlike the first few functions. In Node.js, the naming of functions will come from either the variable name or the actual function name. For example, if you use `Cls.prototype.func` then the name will be `Cls.func` but if you use the function `func` then the name will be `func`.

You can see that there is a slight benefit here but this becomes very useful once you start using patterns involving `async` callbacks:

```
[~/examples/example-13]$ npm install q
```

Let’s throw an error in a callback:

```
var Q = require( 'q' );

Q( )
.then( function() {
    // Promised returned from another function
    return Q( )
    .then( function( ) {
        throw new Error( 'Hello errors' );
    });
})
.fail( function( error ) {
    logger.error( error );
});
```

Running our example gives us:

```
[~/examples/example-13]$ node index.js | bunyan
 [2015-06-07T13:03:57.047Z] ERROR: example-13/4598 on macbook.local: Hello
errors
    Error: Hello errors
        at /Users/fabian/examples/example-13/index.js:12:9
        at _fulfilled (/Users/fabian/examples/example-
13/node_modules/q/q.js:834:54)
```

This is where it starts to get difficult to read; assigning simple names to our functions can help us find where the error is coming from:

```
return Q( )
    .then( function resultFromOtherFunction( ) {
        throw new Error( 'Hello errors' );
    });
```

Running the example:

```
[~/examples/example-13]$ node index.js | bunyan
 [2015-06-07T13:04:45.598Z] ERROR: example-13/4614 on macbook.local: Hello
errors
    Error: Hello errors
        at resultFromOtherFunction (/Users/fabian/examples/example-
13/index.js:12:9)
        at _fulfilled (/Users/fabian/examples/example-
13/node_modules/q/q.js:834:54)
```

Error handling

Another aspect of debugging is handling and expecting errors beforehand. There are three ways in which we can handle our errors:

- a simple `try/catch`
- catching them at the process level
- catching errors on the domain level

A `try/catch` function will be sufficient if we expect an error to occur and we will be able to continue without knowing the result of whatever was being executed, or we could handle and return the error, as shown:

```
function parseJSONAndUse( input ) {
    var json = null;
    try {
        json = JSON.parse( input );
    } catch ( error ) {
        return Q.reject( new Error( "Couldn't parse JSON" ) );
    }
    return Q( use( json ) );
}
```

Another simple way to catch errors is to add an error handler to your process; any errors that are caught at this level are usually fatal and should be treated as such. An exit of the process should follow and you should be using a package, such as `forever` or `pm2`:

```
process.on( 'uncaughtException', function errorProcessHandler( error ) {
    logger.fatal( error );
    logger.fatal( 'Fatal error encountered, exiting now' );
    process.exit( 1 );
});
```

You should always exit the process following an uncaught error. The fact that it is uncaught means that your application is in an unknown state where anything can happen. For example, there could have been an error in your HTTP router and no more requests can be routed to the correct handlers. You can read more about this at https://nodejs.org/api/process.html#process_event_uncaughtexception.

A better way to handle errors on a global level is using `domain`. With domains you can almost *sandbox* a group of asynchronous code together.

Let's think in the context of a request to our server. We make a request, read from a database, make calls to external services, write back to a database, do some logging, do some business logic, and we expect perfect data coming from external sources all around the code. However, in the real world it isn't always so and we can't handle every error that could possibly occur; moreover, we don't want to take down our entire server just because of one error for a very specific request. That's where we need domains.

Let's look at the following example:

```
var Domain = require( 'domain' ),
    domain;

domain = Domain.create( );

domain.on( 'error', function( error ) {
    console.log( 'Domain error', error.message );
});

domain.run( function( ) {
    // Run code inside domain
    console.log( process.domain === domain );
    throw new Error( 'Error happened' );
});
```

Let’s run the code:

```
[~/examples/example-14]$ node index.js
true
Domain error Error happened
```

There is a problem with this code; however, as we are running this synchronously we are still putting the process into a broken state. This is because the error bubbled up to the node itself and then was passed to the active domain.

When we are creating the domain in an asynchronous callback, we can be sure that the process can continue. We can mimic this by using `process.nextTick`:

```
process.nextTick( function( ) {
    domain.run( function( ) {
        throw new Error( 'Error happened' );
    });
    console.log( "I won't execute" );
});

process.nextTick( function( ) {
    console.log( 'Next tick happend!' );
});

console.log( 'I happened before everything else' );
```

Running the example should display the correct logs:

```
[~/examples/example-15]$ node index.js
I happened before everything else
Domain error Error happened
Next tick happend!
```

Summary

In this chapter we have covered a few post-mortem debugging methods to help us uncover bugs including logging, naming practices, and sufficient error handling.

In the next chapter, we will cover configuration of our applications.

Chapter 5. Configuration

As our applications get larger and larger, we start to lose sight of what is configured to do what; we may also get into a situation where we have code running in 12 different places, each needing a bit of code that has to be changed to do something else, for example connecting to a different database. Then, for each of those 12 environments, we have three versions: production, staging, and development. All of a sudden, it gets very complicated. This is why we need to be able to configure our code from a higher-level so that we don’t break anything in the process.

JSON files

There are a few ways in which we can configure our application. The first way that we will look at is a simple JSON file.

If we look at the extensions require supports by default, we can see that we can import JSON right into our code, as shown:

```
[~/examples/example-16]$ node
> require.extensions
{ '.js': [Function],
'.json': [Function],
'.node': [Function: dlopen] }
```

Let’s create a simple server with a configuration file rather than a hardcoded file:

First, we have to create the configuration file:

```
{
    "host": "localhost",
    "port": 8000
}
```

With this, we can now create our server:

```
var Config = require('./config.json'),
    Http = require('http');
Http.createServer(function(request, response) {

}).listen(Config.port, Config.host, function() {
    console.log('Listening on port', Config.port, 'and host', Config.host);
});
```

Now, we can just change the `config` file instead of changing the code to change the port on which our server is running.

But our `config` file is a bit too generic; we have no idea as to what is a host or a port and what they are related to.

While configuring, the keys need to be less generic so that we know what they are being used for, unless the context is given directly by the application. For example, if the application was to serve purely static content then it may be acceptable to have more generic keys.

To make these configuration keys less generic, we can wrap them all in a server object:

```
{
    "server": {
        "host": "localhost",
        "port": 8000
    }
}
```

So now, in order to know about the port of the server we need to use the following code:

```
Config.server.port
```

An example where this will be useful could be for a server that connects to a database, as they can accept both the port and host as the parameters:

```
{
    "server": {
        "host": "localhost",
        "port": 8000
    },
    "database": {
        "host": "db1.example.com",
        "port": 27017
    }
}
```

Environmental variables

Another way in which we can configure our applications is through the use of environmental variables.

These can be defined by the environment you are running your application in or in the command that you are using to start your process with.

In Node.js, you can access the environmental variables using `process.env`. When using `env`, you don't want to be polluting this space too much and so it is a good idea to prefix the key with something related to yourself—your program or company. For example, `Config.server.host` becomes `process.env.NAME_SERVER_HOST`; the reason for this is that we can clearly see what is related to your program and what isn't.

Using environmental variables to configure our server, our code will look as follows:

```
var Http = require('http'),
    server_port,
    server_host;

server_port = parseInt(process.env.FOO_SERVER_PORT, 10);
server_host = process.env.FOO_SERVER_HOST;

Http.createServer(function(request, response) {

}).listen(server_port, server_host, function() {
    console.log('Listening on port', server_port, 'and host', server_host);
});
```

To run this code with our variables, we will use:

```
[~/examples/example-17]$ FOO_SERVER_PORT=8001 \
FOO_SERVER_HOST=localhost node server.js
Listening on port 8001 and host localhost
```

You probably noticed that I had to use `parseInt` for `FOO_SERVER_PORT`; this is because all variables passed in this manner are essentially strings. We can see this by executing `typeof process.env.FOO_ENV`:

```
[~/examples/example-17]$ FOO_ENV=1234 node
> typeof process.env.FOO_ENV
'string'
> typeof parseInt( process.env.FOO_ENV, 10 )
'number'
```

Although this kind of configuration is very simple to create and consume, it may not be the best method, as the variables are hard to keep track of if there are a lot of them and they can be dropped very easily.

Arguments

Another way in which the configuration can be done is through the use of arguments that are passed to Node.js as the process starts, you can access these using `process.argv`, with `argv` standing for argument vector.

The array that `process.argv` returns will always have a `node` at index `0`. For example, if you run node `server.js` then `process.argv` will have the value of `[ 'node', '/example/server.js' ]`.

If you pass an argument to Node.js then it will be added to the end of `process.argv`.

If you run `node server.js --port=8001`, the `process.argv` will contain `[ 'node', '/example/server.js', '--port=8001' ]`, pretty simple, right?

Even though we can have all this configuration, we should always remember that configuration can be simply excluded and we will still want our application to run when this happens. Usually, you should provide default hardcoded values as a backup when you have configuration options.

Parameters such as passwords and private keys should never have a default value but links and options that are usually standard should be given defaults. It is pretty easy to give a default value in Node.js, all you need to do is use the `OR` operator.

```
value = value || 'default';
```

Essentially, what this does is check if the value is `falsy`; if it is, then use the default value. You need to watch out for values that you know could be `falsy`, booleans and numbers definitely fall into this category.

In these cases you can use an `if` statement checking for a `null` value, as shown:

```
if ( value == null ) value = 1
```

Summary

That’s all for configuration. In this chapter you learned about the three methods that you can use to create a dynamic application. We learned that we should name our configuration keys in a way that we can identify what the values are changing to and how they will affect our application. We also learned about how we can pass simple arguments to our application using environmental variables and `argv`.

With this information, we can move forward to connecting and utilizing databases in the next chapter.

Chapter 6. Level DB and NoSQL

In this chapter, we will cover two variations of databases that can be used with Node.js; one provides a very lightweight and simple set of features, while the other gives us more flexibility and a general-purpose set of features. In this chapter, we are going to cover LevelDB and MongoDB

Level DB

One of the great things with Node.js is that we use the same language for both the front and back end and the same goes for NoSQL databases. The majority of them support JSON right off the mark; this is great for anyone using Node.js as there is no time spent in making a relational model, turning it into a JSON-like structure, passing it to the browser, doing something with it, and reversing the process.

With a database that supports JSON natively, you can get right down to business and play ball.

Google has provided us with a simple hook into a NoSQL database that can be installed and can be made ready to use with just one command:

```
[~/examples/example-18]$ npm install level
```

You will see that this will install both `LevelDOWN` and `LevelUP`.

`LevelDOWN` is the low-level binding to `LevelDB` and `LevelUP` is the simple wrapper around this.

`LevelDB` is very simple in terms of setup. Once it is installed, we just create an instance of `LevelUP` and pass it where we want our database to be stored:

```
var LevelUP = require( 'level' ),
    db = new LevelUP( './example-db');
```

Now we have a fast and simple way to store data.

As `LevelDB` is just a simple key/value store, it defaults to string keys and string values. This is useful if that's all the information you wish to store. You can also use it as a simple cache store. It has a very simple API, at this stage we are only going to focus on four methods: `put`, `get`, `del`, and `createReadStream`; it's pretty obvious what most of them do:

Method	**Used for**	**Arguments**
put	inserting pairs	key, value, callback(error)
get	fetching pairs	key, callback(error, value)
del	deleting pairs	key, callback(error)
createReadStream	fetching many pairs	

To insert data once we have created our database, all we need to do is:

```
db.put( 'key', 'value', function( error ) {
    if ( error ) return console.log( 'Error!', error )

    db.get( 'key', function( error, value ) {
        if ( error ) return console.log( 'Error!', error )

        console.log( "key =", value )
    });
});
```

If we run the code, we will see that we inserted and retrieved our value:

```
[~/examples/example-18]$ node index.js
key = value
```

This isn't our simple JSON structure; however, it's just a string. To get our store to save JSON, all we need to do is to pass the value encoding as an option to the database, as shown:

```
var LevelUP = require( 'level' ),
    db = new LevelUP( './example-db', {
        valueEncoding: 'json'
    });
```

Now we can store JSON data:

```
db.put( 'jsonKey', { inner: 'value' }, function ( error ) {
    if ( error ) return console.log( 'Error!', error )

    db.get( 'jsonKey', function( error, value ) {
        if ( error ) return console.log( 'Error!', error )

        console.log( "jsonKey =", value )
    });
});
```

However, a string can be stored as JSON and we can still pass strings as a value and also retrieve it as such.

Running this example will show the following:

```
[~/examples/example-18]$ node index.js
key = value
jsonKey = { inner: 'value' }
```

Now, we have the simple methods down and we can now move on to `createReadStream`.

This function returns an object that can be compared to Node.js built in `ReadableStream`. For each key/value pair in our database, it will emit a `data` event; it also emits other events, such as `error` and `end`. If `error` doesn't have an event listener, then it will propagate, thereby killing your entire process (or domain), as shown:

```
db.put( 'key1', { inner: 'value' }, function( error ) {
    if ( error ) return console.log( 'Error!', error )

    var stream = db.createReadStream( );

    stream
    .on( 'data', function( pair ) {
        console.log( pair.key, "=", pair.value );
    })
    .on( 'error', function( error ) {
        console.log( error );
    })
    .on( 'end', function( ) {
        console.log( 'end' );
    });
});
```

Running this example:

```
[~/examples/example-20]$ node index.js
key1 = { inner: 'value' }
end
```

If we put more data in the database we will have multiple `data` events emitted:

```
[~/examples/example-20]$ node index.js
key1 = { inner: 'value' }
key2 = { inner: 'value' }
end
```

MongoDB

As you can see, databases with Node.js can be very simple. If we want something a bit more complete we can use another NoSQL database called **MongoDB** – another very popular document-based database.

For this set of examples, you can either use a hosted database using a provider such as MongoLab (they provide a free tier for development) or you can set up a database locally following the instructions at http://docs.mongodb.org/manual/installation.

We can continue once you have a database to connect to.

MongoDB has several modules that can be used with Node.js, the most popular one is Mongoose; however, we will be using the core MongoDB module:

```
[~/examples/example-21]$ npm install mongodb
```

To use our database, we first need to connect to it. We need to provide the client with a connection string, a generic URI with the protocol of `mongodb`.

If you have a local mongo database running with no credentials you will use:

```
mongodb://localhost:27017/database
```

The default port is `27017`, so you don't need to specify that; however, it is included for completeness.

If you are using MongoLab, they will provide you with a connection string; it should be in the format of:

```
mongodb://<dbuser>:<dbpassword>@<ds>.mongolab.com:<port>/<db>
```

Connecting to our database is actually pretty simple. All we need to do is provide the driver with a connection string and we get back a database:

```
var MongoDB = require('mongodb'),
    MongoClient = MongoDB.MongoClient;

connection = "mongodb://localhost:27017/database"

MongoClient.connect( connection, function( error, db ) {
    if( error ) return console.log( error );

    console.log( 'We have a connection!' );
});
```

Each set of data in MongoDB is stored in a collection. Once we have a database we can fetch a collection to run the operations on:

```
var collection = db.collection( 'collection_name' );
```

In a collection, we have a few simple methods that hold lots of power, giving us a full CRUD “API”.

Each document in MongoDB has an id, which is an instance of `ObjectId`. The property they use for this id is `_id`.

To save a document we just need to call `save`, it accepts an object or an array of objects. A single object in a collection is referred to as a document:

```
var doc = {
    key: 'value_1'
};
collection.save( doc, { w: 1 }, function( ) {
    console.log( 'Document saved' )
});
```

If we call the `save` function with a document that has an ID then the document will be updated rather than inserted:

```
var ObjectId = MongoDB.ObjectId
// This document already exists in my database
var doc_id = {
    _id: new ObjectId( "55b4b1ffa31f48c6fa33a62a" ),
    key: 'value_2'
};
collection.save( doc_id, { w: 1 }, function( ) {
    console.log( 'Document with ID saved' );
});
```

Now that we have documents in our database, we can query for them, as shown:

```
collection.find( ).toArray( function( error, result ) {
    console.log( result.length + " documents in our database!" )
});
```

If no callback is provided to `find` then it will return a cursor; this allows us to use methods such as `limit`, `sort`, and `toArray`.

You can pass a query to `find` to limit what is returned. In order to find an object by its ID we need to use something, such as:

```
collection.find(
    { _id: new ObjectId( "55b4b1ffa31f48c6fa33a62a" ) },
    function( error, documents ) {
        console.log( 'Found document', documents[ 0 ] );
    }
);
```

We can also filter it by any other property you might use:

```
collection.find(
    { key: 'value' },
    function( error, documents ) {
        console.log( 'Found', documents.length, 'documents' );
    }
);
```

If you have used SQL before, you must have noticed the lack of operators, such as `OR`, `AND`, or `NOT`. However, you don’t need to worry because mongo provides many equivalents.

You can see a complete list here:

http://docs.mongodb.org/manual/reference/operator/query/.

All operators are prefixed with the dollar sign, for example `$and`, `$or`, `$gt`, and `$lt`.

You can see the specific syntax to use these by looking at the documentation.

To use an `$or` condition, you need to include it as if it is a property:

```
collection.find(
    {
        $or: [
            { key: 'value' },
            { key: 'value_2' }
        ]
    },
    function( error, documents ) {
        console.log( 'Found', documents.length, 'documents' );
    }
);
```

Using a database such as MongoDB gives us more power to retrieve our data and create a more feature full software.

Summary

Now we have places where we can store our data. On one end we have a simple key/value store that provides us with a super-convenient way to store data and on the other end we have a feature full database that provides us with a full set of query operators.

Both these databases will help us in the next chapters as we move closer to creating our full stack application.

In the next chapter we will cover `Socket.IO`, a real-time communication framework built on top of WebSockets.

Chapter 7. Socket.IO

Simple HTTP is great for things that don’t need real-time data, but what about when we need to know about things as they happen. For example, if we were creating a website that had a chat interface or similar?

This is when something like Web sockets come into play. Web sockets are usually referred to as WebSockets and are full duplex or two-way low-latency communication channels. They are generally used by messaging applications and games where messages need to be relayed between the server and client. There is a really handy `npm` module called `socket.io`, which can add Web sockets to any Node.js application.

To install it we just need to run:

```
[~/examples/example-27] npm install socket.io
```

Socket.IO can be set up very simply to listen for connections. First, we want to be able to serve out a static html page to run client side code with:

```
var Http = require( 'http' ),
    FS = require( 'fs' );

var server = Http.createServer( handler );

server.listen( 8080 );

function handler( request, response ) {
    var index = FS.readFileSync( 'index.html' );
    index = index.toString( );

    response.writeHead(200, {
        'Content-Type': 'text/html',
        'Content-Length': Buffer.byteLength( index )
    });
    response.end( index );
}
```

Now, lets create an HTML file as well, named `index.html`, in the same directory:

```
<html>
    <head>
        <title>WS Example</title>
    </head>
    <body>
        <h2>WS Example</h2>
        <p id="output"></p>
        <!-- SocketIO Client library -->
        <script src="/socket.io/socket.io.js"></script>
        <script type="application/javascript">
            /* Our client side code will go here */
        </script>
    </body>
</html>
```

Let's run our example and ensure that we get our page, we should be able to see **WS Example** on screen. Now, to add socket support to our application we just need to require `socket.io` and specify what `http` server to listen with to `IOServer`:

```
var IOServer = require( 'socket.io' );
var io = new IOServer( server );
```

Now, whenever there is a new socket connection over `8080` we will get a `connection` event on `io`:

```
io.on( 'connection', function( socket ) {
    console.log( 'New Connection' );
});
```

Lets add some code to the client. Socket.IO provides us with a client library and they expose this through the endpoint `/socket.io/socket.io.js`. This is included in the preceding `index.html` file.

Tip

All the client side code is contained within the second `script` tag of the `index.html` file.

To create a connection with the server all we need to do is call `io.connect` and pass the location. This will return a socket for us with which we can communicate to our server.

We are using the client provided by Socket.IO here, as it will detect whether WebSockets are available, and if possible use them. Otherwise, it will utilize other methods such as polling, which makes sure that it works everywhere rather than just on evergreen browsers:

```
var socket = io.connect( 'http://localhost:8080' );
```

We will use a `p` element to log messages to the screen with. We can do that with this code, then all we need to do is call `logScreen`:

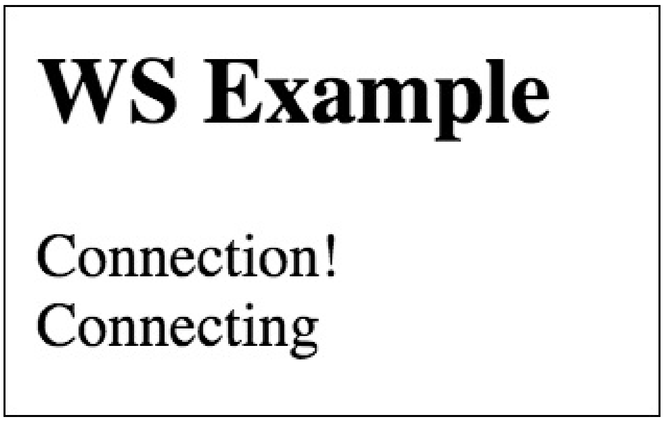
```
var output = document.getElementById( 'output' );

function logScreen( text ) {
    var date = new Date( ).toISOString( );
    line = date + " " + text + "<br/>";
    output.innerHTML =  line + output.innerHTML
}
```

Once a connection is made, just like on the server side a `connection` event is emitted, we can listen to this using `on`:

```
socket.on( 'connection', function( ){
    logScreen( 'Connection!' );
});
```

Now, we can run our server once we navigate to `http://localhost:8080`. You should be able to see **Connection!** showing up:


WS Example

Connection!
Connecting

To receive a message on server side, we just need to listen for the `message` event. For now, we will just echo the message back:

```
socket.on( 'connection', function( ){
    socket.on( 'message', function ( message ) {
        socket.send( message );
    });
});
```

On the client side, we just need to call `send` to send a message and we want to do this inside our connection event. The `api` on each side is very similar to each other, as you can see:

```
socket.send( 'Hello' );
```

On the client side, we also want to listen for messages and log them to the screen:

```
socket.on( 'message', logScreen );
```

Once we restart the server and refresh our page, we should be able to see an additional **Hello** message appear on screen.

```
[~/examples/example-27]$ node index.js
Hello
```

This happens because the server can now send the client packets of data. It also means that we can update the client at any time. For example, every second we can send an update to the client:

```
socket.on( 'connection', function( ){
    function onTimeout( ) {
        socket.send( 'Update' );
    }
    setInterval( onTimeout, 1000 );
});
```

Now, when we restart our server we should be able to see an update message every second.

You might have noticed that you didn't need to refresh your webpage for the connection to be opened again. This is because `socket.io` transparently keeps our connections "alive" as well as reconnecting if needed. This takes all the pain out of using sockets, as we have none of these troubles.

Rooms

Socket.IO also has the concept of rooms, where multiple clients can be grouped into different rooms. To emulate this, all you will need to do is navigate to `http://localhost:8080` in multiple tabs.

Once a client connects, we need to call the `join` method to tell the socket what room to be in. If we wish to do something such as a group chat with specific users only, we need have a room identifier in a database or create one. For now we will just have everyone join the same room:

```
socket.on( 'connection', function( ){
    console.log( 'New Connection' );
    var room = 'our room';
    socket.join( room, function( error ) {
        if ( error ) return console.log( error );

        console.log( 'Joined room!' );
    });
});
```

For every tab we open, we should see a message that we have joined a room:

```
[~/examples/example-27]$ node index.js
New Connection
Joined room!
New Connection
Joined room!
New Connection
Joined room
```

With this, we can broadcast a message to the entire room. Let’s do this every time someone joins. Within the join callback:

```
socket
    .to( room )
    .emit(
        'message',
        socket.id + ' joined the room!'
    );
```

If you look in your browser, with each connection the other clients get a notification that someone else has joined:

```
x3OwYOkOCSsa6Qt5AAAF joined the room!
mlx-Cy1k3szq8W8tAAAE joined the room!
Connection!
Connecting
```

This is great, we can now communicate almost directly between browsers!

If we want to leave a room, all we need to do is call `leave`, we will broadcast this before we call the function:

```
socket
```

```
    .to( room )
    .emit(
        'message',
        socket.id + ' is leaving the room'
    );
socket.leave( room );
```

While running this, you will not see any messages from another client because you are leaving right away: however, if you were to put a delay on this you might see another client come and go:

```
leave = function( ) {
    socket
        .to( room )
        .emit(
            'message',
            socket.id + ' is leaving the room'
        );
    socket.leave( room );
};

setTimeout( leave, 2000 );
```

Authentication

For authentication, we can use the same method that we used with our HTTP server and we can accept a JSON Web Token

In these examples, for simplicity we will just have a single HTTP route to login. We will sign a JWT that we will later authenticate by checking the signature

We need to install a couple of extra `npm` modules for this; we will include `chance` so that we can generate some random data.

```
[~/examples/example-27] npm install socketio-jwt jsonwebtoken chance
```

First, we are going to need a route to `login`. We will modify our handler to watch for the url `/login`:

```
if ( request.url === '/login' ) {
    return generateToken( response )
}
```

Our new function `generateToken` will create a JSON Web Token with some random data using `chance`. We will also need a secret for our tokens:

```
var JWT = require( 'jsonwebtoken' ),
    Chance = require( 'chance' ).Chance( );

var jwtSecret = 'Our secret';

function generateToken( response ) {

    var payload = {
        email: Chance.email( ),
        name: Chance.first( ) + ' ' + Chance.last( )
    }

    var token = JWT.sign( payload, jwtSecret );

    response.writeHead(200, {
        'Content-Type': 'text/plain',
        'Content-Length': Buffer.byteLength( token )
    })
    response.end(token);
}
```

Now, whenever we request `http://localhost:8080/login` we will receive a token that we can use:

```
[~]$ curl -X GET http://localhost:8080/login
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJlbW
joiR2VuZSBGbGVtaW5nIiwiaWF0IjoxNDQxMjcyMjM0
e1Y
```

We can enter this into the debugger at http://jwt.io/ and see the contents:

```
{
  "email": "jefoconeh@ewojid.io",
  "name": "Gene Fleming",
  "iat": 1441272234
}
```

Awesome, we have a token and a random user being generated for us. Now, we can use this to authenticate our users. Socket.IO has a method on the server to do this and we just need to pass a handler type function to it. This is where `socketio-jwt` comes in, we pass it our secret and it will ensure it is a real token, pretty simple:

```
var SocketIOJWT = require( 'socketio-jwt' );

io.use( SocketIOJWT.authorize({
    secret: jwtSecret,
    handshake: true }));
```

Now, when we try to connect to our server from the client it will never emit the `connect` event, as our client isn't authenticated. This is exactly what we want.

We first want to wrap up our Socket.IO code (we will call this later); we also want to give it a parameter of `token`:

```
function socketIO ( token ) {

    var socket = io.connect( 'http://localhost:8080' );

    var output = document.getElementById( 'output' );

    function logScreen( text ) {
        var date = new Date( ).toISOString( );
        line = date + " " + text + "<br/>";
        output.innerHTML =  line + output.innerHTML
    }

    logScreen( 'Connecting' );

    socket.on( 'connect', function( ){
        logScreen( 'Connection!' );
        socket.send( 'Hello' );

    });
    socket.on( 'message', logScreen );

}
```

Next, we will create a `login` function, this will request the login URL and then pass the response to the `socketIO` function, as shown:

```
function login( ) {
{
   var request = new XMLHttpRequest();
    request.onreadystatechange = function() {

            if (
            request.readyState !== 4 ||
            request.status !== 200
            ) return

            socketIO( request.responseText );
    }
    request.open( "GET", "/login", true );
    request.send( null );
}
```

Then we want to call the login function:

```
login( );
```

We can pass the token on to the server by changing the `connect` call to pass a query string:

```
var socket = io.connect( 'http://localhost:8080', {
    query: 'token=' + token
});
```

Now, when running our server and navigating to our client we should be able to connect—awesome! Since we have authenticated we can also respond with a personalized message for each user, inside our server-side `connection` event handler we will emit a message to the client.

Our socket will have a new property called `decoded_token`; using this we will be able to view the contents of our token:

```
var payload = socket.decoded_token;
var name = payload.name;

socket.emit( 'message', 'Hello ' + name + '!' );
```

Once we join our room, we can tell the rest of the clients who have also joined:

```
socket
    .to( room )
    .emit(
        'message',
        name + ' joined the room!'
    );
```

Summary

Socket.IO brings amazing capabilities to our applications. We can now instantly communicate with others, either individually or by broadcasting in a room. With the ability to identify users, we can record messages or the history of that user, ready to be served up by a RESTful API.

We are now ready to build real-time applications!

Chapter 8. Creating and Deploying Packages

Now that we have all of the pieces that are needed to create Node.js applications and servers, we will now focus more on sharing our modules and contributing to the eco-system.

All the packages on npm have been uploaded, maintained, and contributed by someone in the community, so let’s have a look at how we can do the same ourselves.

Creating npm packages

We can start with the following steps:

First we need to create a user:

```
[~]$ npm add user
Username: <username>
Password:
Email: (this IS public) <email>
```

Once we have a user we have opened the gates to npm.

Now, let's create a package:

```
[~/examples/example-22]$ npm init
{
  "name": "njs-e-example-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

To publish this package all we need to do is run `npm publish`:

```
[~/examples/example-22]$ npm publish
+ njs-e-example-package@1.0.0
```

You can see that we have published our package successfully, you can view the one I published at:

https://www.npmjs.com/package/njs-e-example-package

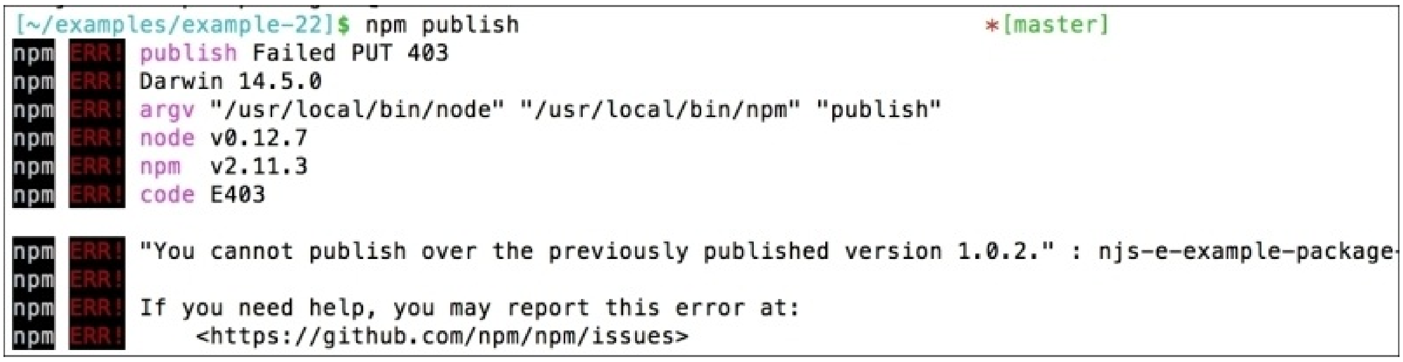
You will have to name your package something else in order to publish it; otherwise, we will have a conflict.

Now we can run the following command:

```
[~/examples/example-21]$ npm install njs-e-example-package
njs-e-example-package@1.0.0 node_modules/njs-e-example-package
```

Then we will have the package! Isn't that pretty cool?

If we try to publish again, we will get an error because version `1.0.2` is already published, as shown in the following screenshot:



```
[~/examples/example-22]$ npm publish                                  *[master]
npm ERR! publish Failed PUT 403
npm ERR! Darwin 14.5.0
npm ERR! argv "/usr/local/bin/node" "/usr/local/bin/npm" "publish"
npm ERR! node v0.12.7
npm ERR! npm  v2.11.3
npm ERR! code E403

npm ERR! "You cannot publish over the previously published version 1.0.2." : njs-e-example-package
npm ERR!
npm ERR! If you need help, you may report this error at:
npm ERR!     <https://github.com/npm/npm/issues>
```

To increment our package version, all we need to do is execute:

```
[~/examples/example-22]$ npm version patch
v1.0.1
```

Now we can publish again:

```
[~/examples/example-22]$ npm publish
+ njs-e-example-package@1.0.1
```

You can go to your packages page on npm and you will see that the version number and release count has been updated.

Versioning in Node.js follows the `semver` schema, which is made up of major, minor, and patch versions. When the patch version is incremented, it means that the API has stayed the same however something has been fixed behind the scenes. If the minor version has been incremented, it means that a non-breaking API change has occurred, such as a method has been added. If the major version is updated, it means that there has been a breaking API change; for example a method has been deleted or a method signature has changed.

Sometimes, there are things in your project that you don't want to be pushed out for other people to have. This could be an original source, some certificates, or maybe some keys for development. Just like when using `git`, we have an ignore file called `.npmignore`.

By default, if there is no `.npmignore` but there is a `.gitignore`, npm will ignore what is matched by the `.gitignore` file. If you don't like this behavior then you can just create an empty `.npmignore` file.

The `.npmignore` file follows the same rules as `.gitignore`, which are as follows:

- Blank lines or lines starting with `#` are ignored
- Standard glob patterns work
- You can end patterns with a forward slash `/` to specify a directory
- You can negate a pattern by starting it with an exclamation point `!`

For example, if we had a directory of certificates in which we had a key:

```
[~/examples/example-22]$ mkdir certificates
[~/examples/example-22]$ touch certifticates/key.key
```

We probably don't want this to be published, so in our ignore file we will have:

```
certificates/
```

We also don’t want any `key` files that we have lying around, so we add this as well:

```
*.key
```

Now, let’s publish:

```
[~/examples/example-22]$ npm version patch
v1.0.2
[~/examples/example-22]$ npm publish
+ njs-e-example-package@1.0.2
```

Now, let’s install our package:

```
[~/examples/example-23]$ npm install njs-e-example-package@1.0.2
```

Now, when we list what’s in the directory, we don’t see all our certificates being passed around:

```
[~/examples/example-23]$ ls node_modules/njs-e-example-package
package.json
```

This is great, but what if we want to protect our entire package and not just a few certificates?

All we need to do is set `private` to `true` in our `package.json` file and it will prevent npm from publishing the module when we run `npm publish`:

Our `package.json` should look something like:

```
{
  "name": "example-23",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "UNLICENSED",
  "dependencies": {
    "njs-e-example-package": "^1.0.2"
  },
  "private": true
}
```

Now, when we run `npm publish`:

```
[~/examples/example-23]$ npm publish
npm ERR! This package has been marked as private
```

Awesome, that’s exactly what we wanted to see.

Summary

Looks like we are getting closer to being ready with all things about Node.js. We know now how to set up, debug, develop, and distribute our software.

In the next chapter, we will cover one more concept we need to know about: unit testing.

Chapter 9. Unit Testing

We have come this far but haven’t done any testing! That’s not very good, is it? Usually, if not always, testing is a major concern in software development. In this chapter, we will cover unit testing concepts with Node.

There are many testing frameworks for Node.js and in this chapter we will be covering Mocha.

Installing mocha

To ensure that `mocha` gets installed everywhere, we need to install it globally. This can be done using the `-g` flag with `npm install`:

```
[~/examples/example-24]$ npm install -g mocha
```

Now, we can use Mocha through the terminal console.

Typically, we will contain all our testing code in a `test` sub-directory within the project. All we need to do to get our code running is run `mocha`, assuming we have written some tests first.

As with many (if not all) unit testing frameworks, Mocha uses assertions to ensure that a test runs correctly. If an error is thrown and is not handled then the test is considered to have failed. What assertion libraries do is throw errors when an unexpected value is passed, so this works well.

Node.js provides a simple assertion module, let's have a look at the following:

```
[~/examples/example-24]$ node
> assert = require( 'assert' )
> expected = 1
> actual = 1
> assert.equal( actual, expected )
> actual = 1
> assert.equal( actual, expected )
AssertionError: 2 == 1
```

As we can see, an error is thrown if the assertion doesn't pass. However, the error message provided wasn't very handy; to fix this we can pass an error message as well:

```
> assert.equal( actual, expected, 'Expected 1' )
AssertionError: Expected 1
```

With this we can create a test.

Mocha provides many ways of creating tests, these are called *interfaces* and the default is called BDD.

You can view all interfaces at http://mochajs.org/#interfaces.

The **BDD** (**Behavior Driven Development**) interface can be compared to Gherkin where we specify a feature and a set of scenarios. It provides methods to help define these sets, `describe` or `context` is used to define a feature, and the `it` or `specify` functions are used to define a scenario.

For example, if we were to have a function that joins someone's first and last name, the tests might look something like the following:

```
var GetFullName = require( '../lib/get-full-name' ),
    assert = require( 'assert' );

describe( 'Fetch full name', function( ) {
    it( 'should return both a first and last name', function( ) {
        var result = GetFullName( { first: 'Node', last: 'JS' } )
        assert.equal( result, 'Node JS' );
    })
})
```

We can also add a few more tests for this; for example, if it threw an error in case of no object being passed:

```
it( 'should throw an error when an object was not passed', function( ) {
    assert.throws(
        function( ) {
            GetFullName( null );
        },
        /Object expected/
    )
})
```

You can explore many more mocha-specific features at http://mochajs.org/.

Chai

Along with the many testing frameworks, there are also many assertion frameworks, one of which is called **Chai**. Complete documentation can be found at http://chaijs.com/.

Instead of just using the built-in assertion module provided by Node.js, we may want to use a module such as Chai to extend our possibilities.

Chai has three sets of interfaces, should, expect, and assert. In this chapter, we will be covering expect.

When using expect, you are using natural language to describe what you want; for example, if you want something to exist, you can say, `expect( x ).to.exist` rather than `assert( !!x )`:

```
var Expect = require( 'chai' ).expect
var Assert = require( 'assert' )

var value = 1

Expect( value ).to.exist
assert( !!value )
```

Using natural language makes things a lot clearer for people reading your tests.

This language can be linked together; we have `to`, `be`, `been`, `is`, `that`, `which`, `and`, `has`, `have`, `with`, `at`, `of`, and `same`, which can help us to build sentences like:

```
Expect( value ).to.be.ok.and.to.equal( 1 )
```

However, these words are only for reliability and they don't modify the result. There are a lot of other words that can be used to assert things, such as `not`, `exists`, `ok`, and many more. You can view them all at http://chaijs.com/api/bdd/.

Some examples of chai in use are:

```
Expect( true ).to.be.ok
Expect( false ).to.not.be.ok
Expect( 1 ).to.exists
Expect( [ ] ).to.be.empty
Expect( 'hi' ).to.equal( 'hi' )
Expect( 4 ).to.be.below( 5 )
Expect( 5 ).to.be.above( 4 )
Expect( function() {} ).to.be.instanceOf( Function )
```

Stubbing methods

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

As you write your tests you want to be only be testing `units` of code. Generally this will be a method, provide it some input, and expect an output of some kind, or if it is a `void` function, expect nothing to be returned.

With this in mind, you have to think of your application as being in a sandboxed state where it can't talk to the outside world. For example, it might not be able to talk to a database or make any kind of external request. Having this assumption is great if you are going to (and you usually should) implement continuous integration and deployment. It also means that there are no external requirements for the machine you are testing on except for Node.js and the testing framework, which could just be a part of your package anyway.

Unless the method you are testing is rather simple and doesn't have any external dependencies, you will probably want to `mock` the methods that you know it is going to execute. A great module to do this is called Sinon.js; it allows you to create `stubs` and `spies` to make sure that the correct data returns from other methods and to ensure that they were called in the first place.

`sinon` provides many helpers, as mentioned before and one of these is called a **spy**. A spy is used mainly to just wrap a function to see what its input and output was. Once a spy has been applied to a function, to the outside world it behaves exactly the same.

```
var Sinon = require( 'sinon' );

var returnOriginal = function( value ) {
    return value;
}

var spy = Sinon.spy( returnOriginal );

result = spy( 1 );
console.log( result ); // Logs 1
```

We can use a spy to check if a function was called:

```
assert( spy.called )
```

Or what arguments were passed for each call:

```
assert.equal( spy.args[ 0 ][ 0 ], 1 )
```

If we provided `spy` with an object and a method to replace then we can restore the original once we are finished. We will usually do this in the `tear down` of our test:

```
var object = {
    spyOnMe: function( value ) {
        return value;
    }
}
Sinon.spy( object, 'spyOnMe' )

var result = object.spyOnMe( 1 )
assert( result.called )
assert.equal( result.args[ 0 ][ 0 ], 1 )

object.spyOnMe.restore( )
```

We also have a `stub` function, which inherits all the functionality of `spy` but instead of calling the original function it completely replaces it.

This is so we can define the behavior, for example, what it returns:

```
var stub = Sinon.stub( ).returns( 42 )
console.log( stub( ) ) // logs 42
```

We can also define a return value for a set of arguments passed:

```
var stub = Sinon.stub( )
stub.withArgs( 1, 2, 3 ).returns( 42 )
stub.withArgs( 3, 4, 5 ).returns( 43 )

console.log( stub( 1, 2, 3 ) ) // logs 42
console.log( stub( 3, 4, 5 ) ) // logs 43
```

Let’s say we had this set of methods:

```
function Users( ) {

}
Users.prototype.getUser = function( id ) {
    return Database.findUser( id );
}
Users.prototype.getNameForUser = function( id ) {
    var user = this.getUser( id );
    return user.name;
}
module.exports = Users
```

Now, we only care about the scenario where a user is returned, as the `getUser` function will throw an error if it can’t find it. Knowing this, we just want to test that when a user is found it returns their name.

This is a perfect example of when we want to `stub` a method:

```
var Sinon = require( 'sinon' );
var Users = require( '../lib/users' );
var Assert = require( 'assert' );

it( 'should return a users name', function( ) {

    var name = 'NodeJS';
    var user = { name: name };

    var stub = Sinon.stub( ).returns( user );
    var users = new Users( );
    users.getUser = stub;

    var result = users.getNameForUser( 1 );

    assert.equal( result, name, 'Name not returned' );
});
```

Instead of replacing the function we can also pass the function using the scope, replacing this with the object we passed; either way is sufficient.

```
var result = users.getNameForUser.call(
    {
        getUser: stub
    },
    1
);
```

Summary

Everything we need to create a Node.js application is now at our fingertips. Testing is just one of those things that are essential to any successful software. We covered using mocha as a testing framework and chai as an assertion framework.

In the next chapter, we will cover using another language with Node.js, CoffeeScript!

Chapter 10. Using More Than JavaScript

Throughout this book we have used only JavaScript. Well, it’s called Node.js isn’t it?

However, that doesn’t mean that we can’t use other languages with it. We can and as long as it compiles to JavaScript you are good to go.

There is a big list of common languages that are available at: https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-JS.

If you are missing your strongly typed language or just want a slightly different syntax, then there will surely be one option out there for you somewhere.

A couple of common languages include `CoffeeScript` and `TypeScript`, they work great with Node.js as they both compile to JavaScript. In this chapter, we will cover the usage of `CoffeeScript`. `TypeScript` is similar in usage; however, the syntax follows a similar path to C# and Java.

CoffeeScript

It’s very simple to install and start using additional languages. Let’s have a look at CoffeeScript:

We need to install CoffeeScript globally, so that we can use a command similar to `node`:

```
[~] npm install -g coffee-script
```

Now we can run `coffee`:

```
[~] coffee
>
```

The syntax is very similar to JavaScript:

```
[~] coffee
> 1 + 1
2
> console.log( 'Hello' )
Hello
```

Instead of using the `.js` extension, we use `.coffee`.

First, we will create a CoffeeScript file:

```
/* index.coffee */
console.log( 'Hello CoffeeScript!' )
```

Then to run it, all we need to do is use the `coffee` command, similar to the `node` command:

```
[~/examples/example-25] coffee index.coffee
Hello CoffeScript!
```

To compile our `.coffee` files into `.js`, we can use `-c`. Once compiled, we can run them directly with Node.js:

```
[~/examples/example-25] coffee -c index.coffee
[~/examples/example-25] node index.js
Hello CoffeeScript!
```

If we have a bunch of CoffeeScript that we want to compile to JavaScript all at once, we can use `coffee -c -o ./lib ./src`. This will take all `.coffee` files from `./src`, compile them to `.js` and then output them to `./lib`.

You will need to compile all your files for other users to use our CoffeeScript code along side their JavaScript code. The alternative is to include CoffeeScript as a dependency and `require` the register file into your application, as shown:

```
/* index.js */
require( 'coffee-script/register' );
require( './other.coffee' );
```

You may need to do this if do not you wish to compile your CoffeeScript, or if you are using a tool that requires a JavaScript file such as Gulp or Grunt.

Tip

To see the equivalents between JavaScript and CoffeeScript you can use the site http://js2.coffee/, it provides a simple way to compare the two on the fly.

CoffeeScript is basically just JavaScript; however, it has targeted readability and simplicity. With simplicity it also tries to limit the use of the bad parts of JavaScript and exposes the good parts.

Using CoffeeScript is usually great for beginners, (and for experts), as it uses English language rather than computer language. For example, instead of using `===` (triple equals) to check if two values equal, we can just use the English word `is`. So, `x === y` becomes `x is y`, which means that there is no translating required when reading.

Along with `is`, there are other keywords, such as `isnt`, `not`, `or`, `and`, `yes` and `no`.

Using these keywords instead of symbol operators gives clarity to the readers and programmers. The CoffeeScript has similar formatting to Python in the way functions and code blocks are declared; the indentation indicates when the block ends and begins

Code blocks and functions

In JavaScript you will usually group together blocks using curly braces, as shown in the following example:

```
if ( true )
{
  console.log( 'It was true!' )
}
```

Where as in CoffeeScript you will leave out all the curly braces, in fact all the braces are left out:

```
if true
  console.log( 'It was true!' )
```

The same is true when declaring a function, notice that we are using an *arrow* rather than the keyword `function`. The parameter list is only required if you want named arguments:

```
func = ->
  console.log( 'I executed' )
```

CoffeeScript tries to assume as much as possible while still giving the programmer enough control.

You may have also noticed that I didn't use the `var` keyword when declaring a function. This is because it is implicitly declared, as you can see by compiling the above code to JavaScript:

```
var func;
func = function()
{
  return console.log('I executed');
};
```

You can see in this compiled code that the last statement in the function is the return value, this means that we don't need to declare the return value and just assume that the last value is returned. This makes it very simple to create one line functions, such as:

```
add = ( a, b ) -> a + b
```

Unlike JavaScript, you may provide default arguments for a function and this can be compared to C#; however, it's not limited to only constants as it essentially executes the statement within the function:

```
keys = { }
func = ( key, date = new Date ) ->
  keys[ key ] = date
```

You can see this by compiling the above function as:

```
var func, keys;
keys = {};
func = function(key, date)
{
  if (date == null)
  {
    date = new Date();
  }
  return keys[key] = date;
};
```

Essentially, all CoffeeScript does is check if the value is `null` or `undefined`.

The existential operator

You can check to see if a value is `null` or `undefined` using the existential operator, which checks to see if the value *exists*. This is indicated by using the question mark symbol after a variable; the statement will be true if the value exists and otherwise false.

To use this in an expression:

```
date = null
if not date?
  date = new Date( )
console.log( date )
```

You can use this as a shorthand operator as well, for example:

```
date ?= new Date( )
console.log( date )
```

The above two examples of code will behave exactly the same and will actually compile to give the same code:

```
var date;
date = null;
if (date == null)
{
  date = new Date();
}
```

You may also use the existential operator to ensure a value exists before accessing a property of it. For example, if you want to get the time from a date, or `-1` if the date doesn't exist:

```
getTime = ( date = null ) -> date?.getTime( ) ? -1
```

Giving `date` the `null` value shows that we don't mind if no value is passed:

When an object doesn't exist and the operator is used then the returned value is `undefined`, this means that we can use the same operator again to return a default value.

Objects and arrays

Along with all the assumptions that CoffeeScript tries to make, it surely does try to remove all the un-needed syntax plain JavaScript requires. Another instance of this can be seen while defining arrays and objects in which the use of a new line declares a new item. For example, you will usually define an array as:

```
array = [
  1,
  2,
  3
]
```

This still works; however, with CoffeeScript you can leave out the commas separating each item:

```
array = [
  1
  2
  3
]
```

You can also mix the two styles together:

```
array = [
  'a', 'b', 'c'
  1, 2, 3
  true, false
]
```

You can do the same with objects, such as:

```
object = {
  foo: 1
  bar: 2
}
```

With objects you can even leave out the curly braces and use indentation to show the differences in the object:

```
object =
  foo: 1
  bar: 2
  foobar:
    another: 3
    key: 4
```

To loop an array in CoffeeScript, all you need to do is use the `for…in` loop, such as:

```
for value, index in array
  console.log( value, index )
  continue if typeof value is 'string'
  console.log( 'Value was not a string' )
```

If you do not wish to use the index of your item, you simply don’t ask for it:

```
for value in array
  console.log( value )
```

As with JavaScript loops, you can use `break` and `continue` to control the flow.

To loop an object in CoffeeScript you can use the `for…of` loop, this is a bit different from the `for…of` loop provided by JavaScript:

```
for key, value of object
  console.log( key, value )
```

As with the `for…in` loop, if you don't want the value, exclude it:

```
for key of object
  console.log( key )
```

For both types of loops, the naming is irrelevant:

```
for key, value of object
    # Note that this will let dates and arrays through ( etc )
    continue unless value instanceof Object
    for nestedKey, nestedValue of value
      console.log(nestedKey, nestedValue )
```

Classes

Unlike JavaScript, CoffeeScript provides a natural way to declare classes and inheritance.

To define a class in JavaScript, you need to declare a function first:

```
function User( username ) {
  this.username = username;
}
```

Then you will declare the `prototype` methods:

```
User.prototype.getUsername = function( ) {
  return this.username;
}
```

If you have a `static` method, you can define this on the function rather than the prototype:

```
User.createUser = function( username ) {
  return new User( username );
}
```

In CoffeeScript you can use the `class` keyword and give the class a name. You can then declare the constructor, static, and instance (prototype) methods:

```
class User
  @createUser: ( username ) ->
    return new User( username )

  constructor: ( username ) ->
    this.username = username
  getUsername: ->
    return this.username
```

Usually, you place all your `static` methods above your constructor so that they stay separate from your instance methods. This avoids confusion, you may have noticed that I declared the static method `createUser` with a `@` prefix, this is how you define a static method in CoffeeScript. However, you can also use the traditional JavaScript method of `User.createUser = ->`, either way will work here.

The code that is run when the instance is being created or *constructed* is called the constructor. This is the same terminology that is used in many other languages so it should be familiar. A constructor is essentially a function.

All the instance methods are declared similarly to properties of an object.

With classes comes another symbol, the `@` symbol. When used on an instance, you can use it to refer to the `this` keyword. For example, the `getUsername` method can be written as:

```
getUsername: ->
  return @username
```

Or, if we want to drop the return statement and make it a one liner:

```
getUsername: -> @username
```

The `@` symbol can also be used in parameter lists to declare that we want the instance property to be set as the passed value. For example, if we had a `setUsername` method we can either do:

```
setUsername: ( username ) ->
  @username = username
```

Or we can do:

```
setUsername: ( @username ) ->
```

Both the methods will compile to the same JavaScript code.

Given the fact that we can use the `@` symbol in our parameter list, we can refactor our constructor function to:

```
constructor: ( @username ) ->
```

Another advantage of using CoffeeScript class is that we can define inheritance. To do so, all we need to do is use the `extends` keyword, this is similar to other languages.

In these examples, we want to have two *classes*, `Person` and `Robot` that extend the base `User` class.

For our person, we want to be able to give them a name and an age along with the username that the `User` class requires.

First, we need to declare our class:

```
class Person extends User
```

Then declare our `constructor`. In our `constructor`, we will call the `super` function, this will execute the constructor of the parent class `User` and we want to pass the username to it, as shown:

```
  constructor: ( username, @name, @age ) ->
    super( username )
```

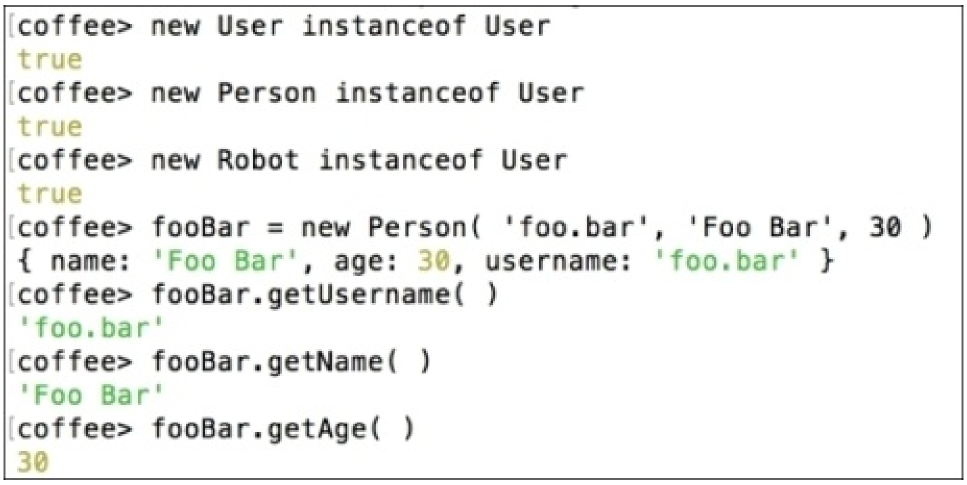
We then add two methods, `getName` and `getAge`:

```
  getName: -> @name
  getAge: -> @age
```

Next, we will do the same for `Robot`, except this time we only want a `username` and `@usage`:

```
class Robot extends User
  constructor: ( username, @usage ) –>
    super( username )
  getUsage: -> @usage
```

We can now create instances of our classes and compare them, as shown here:



```
coffee> new User instanceof User
true
coffee> new Person instanceof User
true
coffee> new Robot instanceof User
true
coffee> fooBar = new Person( 'foo.bar', 'Foo Bar', 30 )
{ name: 'Foo Bar', age: 30, username: 'foo.bar' }
coffee> fooBar.getUsername( )
'foo.bar'
coffee> fooBar.getName( )
'Foo Bar'
coffee> fooBar.getAge( )
30
```

Summary

CoffeeScript tries to make *good* assumptions with your code. This helps to remove some problems that JavaScript developers come across. For example, the difference between == and ===.

You can learn more about the specific syntax of CoffeeScript at http://coffeescript.org/.

In this chapter we have covered utilizing another language. This can help alleviate the struggles with JavaScript's style or syntax for beginners. For people who are used to more language features, this is a big advantage as it helps remove the pitfalls that people usually come across.

Index