



# SMASHING Node.js

JavaScript Everywhere

#### CHAPTER

## THE SETUP

INSTALLING NODE.JS IS a painless process. Since its conception, one of its goals has been maintaining a small number of dependencies that would make the compilation or installation of the project very seamless.

This chapter describes the installation process for Windows, OS X, and Linux systems. For the latter, you're going to ensure that you have the correct dependencies and compile it from the source.

Note: When you see lines prefixed with \$\\$\$ in the code snippets in the book, you should type these expressions into your OS shell.

#### INSTALLING ON WINDOWS

On Windows, go to http://nodejs.org and download the MSI installer. Every release of node has a corresponding MSI installer that you need to download and execute.

The filename follows the format node-v?.?.?.msi. Upon executing it, simply follow the instructions in the setup wizard shown in Figure 1-1.

To ensure that the installation worked, open the shell or command prompt by running cmd. exe and typing \$ node -version.

The version name of the package you just installed should display.



Figure 1-1: The Node.JS setup wizard.

#### INSTALLING ON OS X

On the Mac, similarly to Windows, you can leverage an installer package. From the Node.JS website, download the PKG file that follows the format node-v?.?.?.pkg. If you want to compile it instead, ensure you have XCode installed and follow the Compilation instructions for Linux.

Run the downloaded package and follow the simple steps (see Figure 1-2).

To ensure installation was successful, open the shell or terminal by running Terminal. app (you can type in "Terminal" in Spotlight to locate it) and type in \$ node -version.

The version of Node you just installed should be outputted.



Figure 1-2: The Node.JS package installer.

#### INSTALLING ON LINUX

Compiling Node.JS is almost just as easy as installing binaries. To compile it in most \*nix systems, simply make sure a C/C++ compiler and the OpenSSL libraries are available.

Most Linux distributions come with a package manager that allows for the easy installation of these.

For example, for Amazon Linux, you use

```
> sudo yum install gcc gcc-c++ openssl-devel curl
```

On Ubuntu, the installation is slightly different; you use

```
> sudo apt-get install g++ libssl-dev apache2-utils curl
```

#### COMPILING

From your OS terminal, execute the following commands:

*Note: Replace?* with the latest available version of node in the following example.

```
$ curl -0 http://nodejs.org/dist/node-v?.?.tar.gz
$ tar -xzvf node-v?.?.tar.gz
$ cd node-v?.?.?
$ ./configure
$ make
$ make test
$ make install
```

If the make test command aborts with errors, I recommend you stop the installation and post a log of the ./configure, make, and make test commands to the Node.JS mailing list.

#### ENSURING THAT IT WORKS

Launch a terminal or equivalent, such as XTerm, and type in \$ node -version.

The version of Node you just installed should be outputted.

#### THE NODE REPL

To run the Node REPL, simply type node.

Try running some JavaScript expressions. For example:

```
> Object.keys(global)
```

Note: When you see lines prefixed with > in the code snippets in the book, you should run these expressions in the REPL.

The REPL is one of my favorite tools for quickly verifying that different Node or vanilla JavaScript APIs work as expected. While developing larger modules, it's often useful to check a certain API works exactly the way you remember it when unsure. To that end, opening a separate terminal tab and quickly evaluating some JavaScript primitives in a REPL helps immensely.

#### EXECUTING A FILE

Like most scripted programming languages, Node can interpret the contents of a file by appending a path to the node command.

With your favorite text editor, create a file called my-web-server.js, with the following contents:

```
var http = require('http');
var serv = http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<marquee>Smashing Node!</marquee>');
});
serv.listen(3000);
```

#### Run the file:

```
$ node my-web-server.js
```

Then, as shown in Figure 1-3, point your web browser to http://localhost:3000.

In this code snippet, you're leveraging the power of Node to script a fully compliant HTTP server that serves a basic HTML document. This is the traditional example used whenever Node.JS is being discussed, because it demonstrates the power of creating a web server just like Apache or IIS with only a few lines of JavaScript.

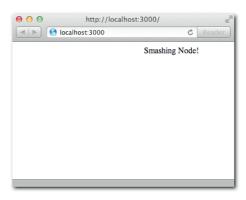


Figure 1-3: Serving a basic HTML document in Node.

#### NPM

The Node Package Manager (NPM) allows you to easily manage modules in projects by downloading packages, resolving dependencies, running tests, and installing command-line utilities.

Even though doing so is not essential to the core functionality of the project, you truly need to work efficiently on projects that rely on other pre-existing modules released by third parties.

NPM is a program written in Node.JS and shipped with the binary packages (the MSI Windows installer, and the PKG for the Mac). If you compiled node from the source files, you want to install NPM as follows:

```
$ curl http://npmjs.org/install.sh | sh
```

To ensure successful installation, issue the following command:

```
$ npm --version
```

The NPM version should be displayed.

#### INSTALLING MODULES

To illustrate the installation of a module with NPM, install the colors library in the directory my-project and then create an index.js file:

```
$ mkdir my-project/
$ cd my-project/
$ npm install colors
```

Verify that the project was installed by ensuring the path node\_modules/colors was created.

Then edit index.js with your favorite editor:

```
$ vim index.js
```

And add the following contents:

```
require('colors');
console.log('smashing node'.rainbow);
```

The result should look like Figure 1-4.

```
for the second s
```

Figure 1-4: The result of installing a module

#### DEFINING YOUR OWN MODULE

To define your own module, you need to create a package. json file. Defining your own module has three fundamental benefits:

- Allows you to easily share the dependencies of your application with others, without sending along the node\_modules directory. Because npm install takes care of fetching everything, distributing this directory wouldn't make sense. This is especially important in SCM systems like Git.
- Allows you to easily track the versions of the modules you depend on that you know work. For example, when you wrote a particular project, you ran npm install colors and that installed colors 0.5.0. A year later, due to API changes, perhaps the latest colors are no longer compatible with your project, and if you were to run npm install without specifying the version, your project would break.
- Makes redistribution possible. Did your project turn out fine and you want to share it with others? Because you have a package.json, the command npm publish. publishes it to the NPM registry for everyone to install.

In the directory created earlier (my-project), remove the node\_modules directory and create a package.json file:

```
$ rm -r node_modules
$ vim package.json
```

Then add the following contents:

```
{
    "name": "my-colors-project"
, "version": "0.0.1"
, "dependencies": {
        "colors": "0.5.0"
}
```

Note: The contents of this file must be valid JSON. Valid JavaScript is not enough. This means that you must make sure, for example, to use double quotes for all strings, including property names.

The package.json file is the file that describes your project to both Node.JS and NPM. The only required fields are name and version. Normally, modules have dependencies, which is an object that references other projects by the name and version they defined in their package.json files.

Save the file, install the local project, and run index.js again:

```
$ npm install
$ node index  # notice that you don't need to include ".js"!
```

In this case, the intention is to create a module for internal use. If you wanted, NPM makes it really easy to publish a module by running:

```
$ npm publish
```

To tell Node which file to look for when someone calls require ('my-colors-project') we can specify the main property in the package.json:

```
{
    "name": "my-colors-project"
, "version": "0.0.1"
, "main": "./index"
, "dependencies": {
        "colors": "0.5.0"
}
```

When you learn how to make modules export APIs, the main property will become a lot more important, because you will need it to define the entry point of your modules (which sometimes are comprised of multiple files).

To learn about all the possible properties for the package. json file, run:

```
$ npm help json
```

Tip: If you never intend to publish a certain project, add "private": "true" to your package. json. This prevents accidental publication.

#### INSTALLING BINARY UTILITIES

Some projects distribute command-line tools that were written in Node. When that's the case, you need to install them with the -g flag.

For example, the web framework you're going to learn in this book called express contains an executable utility to create projects.

```
$ npm install -g express
```

Then try it out by creating a directory and running "express" inside:

```
$ mkdir my-site
$ cd mysite
$ express
```

Tip: If you want to distribute a script like this, include a flag "bin": "./path/to/script" pointing to your executable script or binary when publishing.

#### EXPLORING THE NPM REGISTRY

Once you get comfortable with the Node.JS module system in Chapter 4, you should be able to write programs that leverage any module in the ecosystem.

NPM has a rich registry that contains thousands of modules. Two commands are instrumental in your exploration of the registry: search and view.

If you want to search for plugins related to realtime, for example, you would execute the following:

```
$ npm search realtime
```

This will search all the published modules that contain MySQL in their name, tags, and description fields.

Once you find a package that interests you, you can see its package.json and other properties related to the NPM registry by running npm view followed by the module name. For example:

```
$ npm view socket.io
```

Tip: If you want to learn more about a certain NPM command, type "npm help <command>." For example, "npm help publish" will teach you more about how to publish modules.

#### SUMMARY

After this chapter, you should now have a working Node.JS + NPM environment.

In addition to being able to run the node and npm commands, you should now have a basic understanding of how to execute simple scripts, but also how to put together modules with dependencies.

You now know that an important keyword in Node.JS is require, which allows for module and API interoperability, and which will be an important subject in Chapter 4, after quickly reviewing the language basics.

You also are now aware of the NPM registry, which is the gateway to the Node.JS module ecosystem. Node.JS is an open source project, and as a result many of the programs that are written with it are also open source and available for you to reuse, a few keystrokes away.

#### CHAPTER

## JAVASCRIPT: AN OVERVIEW

#### INTRODUCTION

JAVASCRIPT IS A prototype-based, objectoriented, loosely-typed dynamic scripting language. It has powerful features from the functional world, such as *closures* and *higherorder functions*, that are of special interest here.

JavaScript is technically an implementation of the ECMAScript language standard. It's important to know that with Node, because of v8, you'll be primarily dealing with an implementation that gets close to the standard, with the exception of a few extra features. This means that the JavaScript you're going to be dealing with has some important differences with the one that earned the language its bad reputation in the browser world.

In addition, most of the code you'll write is in compliance with the "good parts" of JavaScript that Douglas Crockford enounced in his famous book, *JavaScript: The Good Parts*.

This chapter is divided into two parts:

- Basic JavaScript. The fundamentals of the language. They apply everywhere: node, browser, and standards committee.
- v8 JavaScript. Some features used in v8 are not available in all browsers, especially Internet Explorer, because they've recently been standardized. Others are nonstandard, but you still use them because they solve fundamental problems.

In addition, the next chapter covers the language extensions and features exclusively available in Node.

#### BASIC JAVASCRIPT

This chapter assumes that you're somewhat familiar with JavaScript and its syntax. It goes over some fundamental concepts you must understand if you want to work with Node.js.

#### **TYPES**

You can divide JavaScript types into two groups: *primitive* and *complex*. When one of the primitive types is accessed, you work directly on its value. When a complex type is accessed, you work on a reference to the value.

- The primitive types are number, boolean, string, null, and undefined.
- The complex types are array, function, and object.

#### To illustrate:

```
// primitives
var a = 5;
var b = a;
b = 6;
a; // will be 5
b; // will be 6

// complex
var a = ['hello', 'world'];
var b = a;
b[0] = 'bye';
a[0]; // will be 'bye'
b[0]; // will be 'bye'
```

In the second example, b contains the *same reference* to the value as a does. Hence, when you access the first member of the array, you alter the original, so a [0] === b[0].

#### TYPE HICCUPS

Correctly identifying the type of value a certain variable holds remains a challenge in JavaScript.

Because JavaScript has constructors for most primitives like in other languages with object-oriented features, you can create a string in these two ways:

```
var a = 'woot';
var b = new String('woot');
a + b; // 'woot woot'
```

If you use the typeof and instanceof operators on these two variables, however, things get interesting:

```
typeof a; // 'string'
typeof b; // 'object'
a instanceof String; // false
b instanceof String; // true
```

However, both are definitely strings that have the same prototypical methods:

```
a.substr == b.substr; // true
```

And they evaluate in the same way with the == operator but not with ===:

```
a == b; // true
a === b; // false
```

Considering these discrepancies, I encourage you to always define your types in the literal way, avoiding new.

It's important to remember that certain values will be evaluate to false in conditional expressions: null, undefined, '', 0:

```
var a = 0;
if (a) {
   // this will never execute
}
a == false; // true
a === false; // false
```

Also noteworthy is the fact that typeof doesn't recognize null as its own type:

```
typeof null == 'object'; // true, unfortunately
```

And the same goes for arrays, even if defined with [], as shown here:

```
typeof [] == 'object'; // true
```

You can be thankful that v8 provides a way of identifying an array without resorting to hacks. In browsers, you typically inspect the internal [[Class]] value of an object: Object. prototype.toString.call([]) == '[object Array]'. This is an immutable property of objects that has the benefit of working across different contexts (for example, browser frames), whereas instanceof Array is true only for arrays initialized within that particular context.

#### **FUNCTIONS**

Functions are of utmost importance in JavaScript.

They're *first class*: they can be stored in variables as references, and then you can pass them around as if they were any other object:

```
var a = function () {}
console.log(a); // passing the function as a parameter
```

All functions in JavaScript can be named. It's important to distinguish between the function name and the variable name:

```
var a = function a () {
  'function' == typeof a; // true
}.
```

#### THIS, FUNCTION#CALL, AND FUNCTION#APPLY

When the following function is called, the value of this is the global object. In the browser, that's window:

```
function a () {
   window == this; // true;
};
a();
```

By using the .call and .apply methods, you can change the reference of this to a different object when calling the function:

```
function a () {
   this.a == 'b'; // true
}
a.call({ a: 'b' });
```

The difference between call and apply is that call takes a list of parameters to pass to the function following, whereas apply takes an array:

```
function a (b, c) {
  b == 'first'; // true
  c == 'second'; // true
}
a.call({ a: 'b' }, 'first', 'second')
a.apply({ a: 'b' }, ['first', 'second']);
```

#### **FUNCTION ARITY**

An interesting property of a function is its *arity*, which refers to the number of arguments that the function was declared with. In JavaScript, this equates to the length property of a function:

```
var a = function (a, b, c);
a.length == 3; // true
```

Even though less common in the browser, this feature is important to us because it's leveraged by some popular Node.JS frameworks to offer different functionality depending on the number of parameters the functions you pass around take.

#### **CLOSURES**

In JavaScript, every time a function is called, a new scope is defined.

Variables defined within a scope are accessible only to that scope and inner scopes (that is, scopes defined within that scope):

```
var a = 5;
function woot () {
    a == 5; // true

    var a = 6;
    function test () {
        a == 6; // true
    }
    test();
};
woot();
```

*Self-invoked functions* are a mechanism by which you declare and call an anonymous function where your only goal is defining a new scope:

```
var a = 3;
(function () {
  var a = 5;
})();
a == 3 // true;
```

These functions are very useful when you want to declare *private variables* that shouldn't be exposed to another piece of code.

#### **CLASSES**

In JavaScript, there's no class keyword. A class is defined like a function instead:

```
function Animal () { }
```

To define a method on all the instances of Animal that you create, you set it on the prototype:

```
Animal.prototype.eat = function (food) {
   // eat method
}
```

It's worth mentioning that within functions in the prototype, this doesn't refer to the global object like regular functions, but to the class instance instead:

```
function Animal (name) {
  this.name = name;
}
Animal.prototype.getName () {
  return this.name;
};
var animal = new Animal('tobi');
a.getName() == 'tobi'; // true
```

#### INHERITANCE

JavaScript has *prototypical inheritance*. Traditionally, you simulate classical inheritance as follows.

You define another constructor that's going to inherit from Animal:

```
function Ferret () { };
```

To define the inheritance chain, you initialize an Animal object and assign it to the Ferret. prototype.

```
// you inherit
Ferret.prototype = new Animal();
```

You can then define methods and properties exclusive to your subclass:

```
// you specialize the type property for all ferrets
Ferret.prototype.type = 'domestic';
```

To override methods and call the parent, you reference the prototype:

```
Ferret.prototype.eat = function (food) {
   Animal.prototype.eat.call(this, food);
   // ferret-specific logic here
}
```

This technique is almost perfect. It's the best performing across the board (compared to the alternative functional technique) and doesn't break the instanceof operator:

```
var animal = new Animal();
animal instanceof Animal // true
animal instanceof Ferret // false

var ferret = new Ferret();
ferret instanceof Animal // true
ferret instanceof Ferret // true
```

Its major drawback is that an object is initialized when the inheritance is declared (Ferret. prototype = new Animal), which might be undesirable. A way around this problem is to include a conditional statement in the constructor:

```
function Animal (a) {
   if (false !== a) return;
   // do constructor stuff
}

Ferret.prototype = new Animal(false)
```

Another workaround is to define a new, empty constructor and override its prototype:

```
function Animal () {
    // constructor stuff
}

function f () {};
f.prototype = Animal.prototype;
Ferret.prototype = new f;
```

Fortunately, v8 has a cleaner solution for this, which is described later in this chapter.

### TRY {} CATCH {}

try/catch allows you to capture an exception. The following code throws one:

```
> var a = 5;
> a()
TypeError: Property 'a' of object #<Object> is not a function
```

When a function throws an error, execution stops:

```
function () {
  throw new Error('hi');
  console.log('hi'); // this will never execute
}
```

If you use try/catch, you can handle the error and execution continues:

```
function () {
  var a = 5;
  try {
    a();
  } catch (e) {
    e instanceof Error; // true
  }
  console.log('you got here!');
}
```

#### **V8 JAVASCRIPT**

So far you've looked at the JavaScript features that are most relevant to dealing with the language in most environments, including ancient browsers.

With the introduction of the Chrome web browser came a new JavaScript engine, v8, which has been quickly pushing the boundaries by providing us with an extremely fast execution environment that stays up-to-date and supports the latest ECMAScript features.

Some of these features address deficiencies in the language. Others were introduced thanks to the advent of client-side frameworks like jQuery and PrototypeJS, because they provided extensions or utilities that are so frequently used it's now unimaginable to consider the JavaScript language without them.

In this section you'll learn about the most useful features that you can take advantage of from v8 to write more concise and faster code that fits right it with the style of code that the most popular Node.JS frameworks and libraries adopt.

#### OBJECT#KEYS

If you wanted to obtain the keys for the following object (a and c)

```
var a = { a: 'b', c: 'd' };
```

Then normally iterate as follows:

```
for (var i in a) { }
```

By iterating over the keys, you can collect them in an array. However, if you were to extend the Object.prototype as follows:

```
Object.prototype.c = 'd';
```

To avoid getting c in the list of keys you would need to run a hasOwnProperty check:

```
for (var i in a) {
   if (a.hasOwnProperty(i)) {}
}
```

To get around that complication, to get all the own keys in an object, in v8 you can safely use

```
var a = { a: 'b', c: 'd' };
Object.keys(a); // ['a', 'c']
```

#### ARRAY#ISARRAY

Like you saw before, the typeof operator will return "object" for arrays. Most of the time, however, you want to check that an array is actually an array.

Array.isArray returns true for arrays and false for any other value:

```
Array.isArray(new Array) // true
Array.isArray([]) // true
Array.isArray(null) // false
Array.isArray(arguments) // false
```

#### ARRAY METHODS

To loop over an array, you can use for Each (similar to jQuery \$ . each):

```
// will print 1 2 and 3
[1, 2, 3].forEach(function (v) {
  console.log(v);
}):
```

To filter elements out of an array, you can use filter (similar to jQuery \$.grep)

```
[1, 2, 3].forEach(function (v) {
  return v < 3;
}); // will return [1, 2]</pre>
```

To change the value of each item, you can use map (similar to jQuery \$.map)

```
[5, 10, 15].map(function (v) {
  return v * 2;
}); // will return [10, 20, 30]
```

Also available but less commonly used are the methods reduce, reduceRight, and lastIndexOf.

#### STRING METHODS

To remove space in the beginning and ending of a string, use

```
' hello '.trim(); // 'hello'
```

#### ISON

v8 exposes JSON. stringify and JSON. parse to decode and encode JSON, respectively.

JSON is an encoding specification that closely resembles the JavaScript object literal, utilized by many web services and APIs:

```
var obj = JSON.parse('{"a":"b"}')
obj.a == 'b'; // true
```

#### FUNCTION#BIND

.bind (equivalent to jQuery's \$.proxy) allows you to change the reference of this:

```
function a () {
  this.hello == 'world'; // true
};

var b = a.bind({ hello: 'world' });
b();
```

#### FUNCTION#NAME

In v8, the nonstandard property name of a function is supported:

```
var a = function woot () {};
a.name == 'woot'; // true
```

This property is used internally by v8 in stack traces. When an error is thrown, v8 shows a *stack trace*, which is the succession of function calls it made to reach the point where the error occurred:

```
> var woot = function () { throw new Error(); };
> woot()
Error
    at [object Context]:1:32
```

In this case, v8 is not able to assign a name to the function reference. If you name it, however, v8 will be able to include it in the stack traces as shown here:

```
> var woot = function buggy () { throw new Error(); };
> woot()
Error
   at buggy ([object Context]:1:34)
```

Because naming significantly aids in debugging, I always recommend you name your functions.

#### \_PROTO\_ (INHERITANCE)

\_\_proto\_\_ makes it easy for you to define the inheritance chain:

```
function Animal () { }
function Ferret () { }
Ferret.prototype.__proto__ = Animal.prototype;
```

This is a very useful feature that removes the need to:

- Resort to intermediate constructors, as shown in the previous section.
- Leverage OOP toolkits or utilities. You don't need to require any third-party modules to expressively declare prototypical inheritance.

#### **ACCESSORS**

You are able to define properties that call functions when they're accessed (\_\_define Getter\_\_) or set (\_\_defineSetter\_\_).

As an example, define a property called ago that returns the time ago in words for a Date object.

Many times, especially in the software you create, you want to express time in words relative to a certain point. For example, it's easier for people to understand that something happened three seconds ago than reading the complete date.

The following example adds an ago getter to all the Date instances that will output the distance of time in words to the present. Simply accessing the property will execute the function you define, without having to explicitly call it.

```
// Based on prettyDate by John Resig (MIT license)
Date.prototype.__defineGetter__('ago', function () {
  var diff = (((new Date()).getTime() - this.getTime()) / 1000)
  , day_diff = Math.floor(diff / 86400);
```

```
return day_diff == 0 && (
    diff < 60 && "just now" ||
    diff < 120 && "1 minute ago" ||
    diff < 3600 && Math.floor( diff / 60 ) + " minutes ago" ||
    diff < 7200 && "1 hour ago" ||
    diff < 86400 && Math.floor( diff / 3600 ) + " hours ago") ||
    day_diff == 1 && "Yesterday" ||
    day_diff < 7 && day_diff + " days ago" ||
    Math.ceil( day_diff / 7 ) + " weeks ago";
});</pre>
```

Then you simply refer to the ago property. Notice that you're not executing a function, yet it's still being executed transparently for you:

```
var a = new Date('12/12/1990'); // my birth date a.ago // 1071 weeks ago
```

#### SUMMARY

Understanding this chapter is essential to getting up to speed with the quirks of the language and handicaps of most environments the language has traditionally been run in, such as old browsers.

Due to JavaScript evolving really slowly and being somewhat overlooked for years, many developers have invested significant amounts of time in developing techniques to write the most efficient and maintainable code, and have characterized what aspects of the language don't work as expected.

v8 has done a fantastic job at keeping up to date with the recent editions of ECMA, and continues to do so. The Node.JS core team of developers always ensures that when you install the latest version of Node, you always get the most recent version of v8. This opens up a new panorama for server-side development, since we can leverage APIs that are easier to understand and faster to execute.

Hopefully during this chapter you've learned some of the features that Node developers commonly use, which are those that are defining the present and future of JavaScript.

### **CONTENTS**

PART I: GETTING STARTED: SETUP AND CONCEPTS	
Chapter 1: The Setup	7
Installing on Windows	8
Installing on OS X	8
Installing on Linux	8
Compiling	9
Ensuring that it works	9
The Node REPL	9
Executing a file	10
NPM	10
Installing modules	11
Defining your own module	12
Installing binary utilities	13
Exploring the NPM registry	14
Summary	14
Chapter 2: JavaScript: An Overview	15
Introduction	15
Basic JavaScript	16
Types	16
Type hiccups	16
Functions	18
this, Function#call, and Function#apply	18
Function arity	19
Closures	19
Classes	20
Inheritance	20
try {} catch {}	21
v8 JavaScript	22
Object#keys	22
Array#isArray	23
Array methods	23
String methods	24
JSON	24
Function#bind	24

Function#name		24
_proto_ (inherit	ance)	25
Accessors		25
Summary		26
Chapter 3: Blocking and Non-block	king IO	27
With great power come	_	28
Blocking-ness	7	29
A single-threade	ed world	31
Error handling		33
Stack traces		35
Summary		37
Chapter 4: Node JavaScript		39
The global object		40
Useful globals		40
The module system		41
Absolute and rel	ative modules	41
Exposing APIs		44
Events		45
Buffers		47
Summary		48
PART II: ESSENTIAL NODE	APIS	49
Chapter 5: CLI and FS APIs: Your I		51
Chapter 5: CLI and FS APIs: Your I	First Application	<b>51</b> 52
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr	First Application	<b>51</b> 52 52
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo	First Application	<b>51</b> 52 52 53
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async?	First Application am dule	51 52 52 53 54
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s	First Application  am  dule  streams	51 52 52 53 54 55
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput	First Application  am  dule  streams	51 52 52 53 54 55 57
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring	First Application  am dule  streams	51 52 52 53 54 55
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with	First Application  am dule  streams	51 52 52 53 54 55 57
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI	First Application  am dule  streams	51 52 52 53 54 55 57 59 61
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy	First Application  am dule  streams  the fs	51 52 52 53 54 55 57 59 61 63
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI	First Application  am dule  streams  the fs	51 52 52 53 54 55 57 59 61 63 63
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director	First Application  am dule  streams  the fs	51 52 52 53 54 55 57 59 61 63 63 64
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director	First Application  am dule  streams  the fs	51 52 52 53 54 55 57 59 61 63 63 64 65
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director Environmental se	am dule streams the fs	51 52 52 53 54 55 57 59 61 63 63 64 65
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director Environmental securing Signals	am dule streams the fs	51 52 52 53 54 55 57 59 61 63 64 65 65
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director Environmental securing Signals ANSI escape coor	am dule streams the fs	51 52 52 53 54 55 57 59 61 63 64 65 65
Chapter 5: CLI and FS APIs: Your I Requirements Writing your first progr Creating the mo Sync or async? Understanding s Input and ouput Refactoring Interacting with Exploring the CLI Argy Working director Environmental s Exiting Signals ANSI escape coor Exploring the fs module	am dule streams the fs	51 52 52 53 54 55 57 59 61 63 64 65 65 65 66

#### **VI** CONTENTS

Chapter 6: TCP	69
What are the characteristics of TCP?	70
Connection-oriented communication	
and same-order delivery	70
Byte orientation	70
Reliability	71
Flow control	71
Congestion control	71
Telnet	71
A TCP chat program	74
Creating the module	74
Understanding the net.server API	74
Receiving connections	76
The data event	77
State and keeping track of connections	79
Wrap up	81
An IRC Client program	83
Creating the module	83
Understanding the net#Stream API	84
Implementing part of the IRC protocol	84
Testing with a real-world IRC server	85
Summary	85
Chapter 7: HTTP	87
The structure of HTTP	88
Headers	89
Connections	93
A simple web server	94
Creating the module	95
Printing out the form	95
Methods and URLs	97
Data	99
Putting the pieces together	102
Bullet-proofing	103
A Twitter web client	104
Creating the module	104
Making a simple HTTP request	104
Sending a body of data	106
Getting tweets	107
A superagent to the rescue	110
Reloading HTTP servers with up	111
Summary	112

PART III: WEB DEVELOPMENT	113
Chapter 8: Connect	115
A simple website with HTTP	116
A simple website with Connect	119
Middleware	121
Writing reusable middleware	122
Static middleware	127
Query	128
Logger	129
Body parser	131
Cookies	134
Session	134
REDIS sessions	140
methodOverride	141
basicAuth	141
Summary	144
Chapter 9: Express	145
A simple express app	146
Creating the module	146
HTML	146
Setup	147
Defining routes	148
Search	150
Run	152
Settings	153
Template engines	154
Error handling	155
Convenience methods	155
Routes	157
Middleware	159
Organization strategies	160
Summary	162
Chapter 10: WebSocket	163
AJAX	164
HTML5 WebSocket	166
An Echo Example	167
Setting it up	167
Setting up the server	168
Setting up the client	169
Running the server	170
Mouse cursors	171
Setting up the example	171
Setting up the server	172

#### VIII CONTENTS

The Challenges Ahead  Close doesn't mean disconnect  JSON  Reconnections  Broadcasting  WebSockets are HTML5: Older browsers don't support them  The solution  Summary  Chapter 11: Socket.IO  Transports  Disconnected versus closed  Events  Namespaces  A chat program  Setting up the program  Setting up the server  Setting up the client  Events and Broadcasting  Ensuring reception  A DJ-by-turns application  Extending the chat  Integrating with the Grooveshark API  Playing  Summary  PART IV: DATABASES  Chapter 12: MongoDB  Installation  Accessing MongoDB: A user authentication example  Setting up the application  Creating the Express app  Connecting to MongoDB  Creating documents  Authentication middleware  Validation  Atomicity  Safe mode  Introducing Mongoose  Defining a model  Defining nested keys  2022		Setting up the client	174
Close doesn't mean disconnect		Running the server	176
JSON   177   Reconnections   177   Broadcasting   177   WebSockets are HTML5: Older browsers don't support them   178   The solution   178   Summary   178   Chapter 11: Socket.JO   179   Transports   180   Disconnected versus closed   180   Events   180   Events   180   Namespaces   181   A chat program   182   Setting up the program   182   Setting up the server   182   Setting up the client   183   Ensuring reception   190   A DJ-by-turns application   191   Extending the chat   191   Integrating with the Grooveshark API   193   Playing   196   Summary   201  PART IV: DATABASES   203    Chapter 12: MongoDB   205   Creating up the application   206   Creating the Express app   208   Setting up the application   206   Creating documents   214   Finding documents   214   Finding documents   215   Authentication middleware   217   Validation   218   Atomicity   326   Defining a model   220   Defining nested keys   222		The Challenges Ahead	177
Reconnections   177   Broadcasting   177   WebSockets are HTML5: Older browsers don't support them   177   The solution   178   Summary   178   Chapter 11: Socket.IO   179   Transports   186   Disconnected versus closed   186   Events   188   Namespaces   188   A chat program   182   Setting up the program   182   Setting up the server   182   Setting up the client   183   Events and Broadcasting   185   Ensuring reception   190   A DJ-by-turns application   191   A Extending the chat   191   Integrating with the Grooveshark API   193   Playing   196   Summary   201  PART IV: DATABASES   203    Chapter 12: MongoDB   205   Installation   207   Accessing MongoDB: A user authentication example   208   Setting up the application   208   Creating the Express app   208   Connecting to MongoDB   212   Creating documents   214   Finding documents   215   Authentication middleware   217   Validation   218   Authentication middleware   216   Atomicity   326   Safe mode   215   Introducing Mongoose   226   Defining a model   226   Defining nested keys   222		Close doesn't mean disconnect	177
Broadcasting WebSockets are HTML5: Older browsers don't support them The solution Summary 178 Chapter 11: Socket.IO Transports Disconnected versus closed Events Namespaces 188 A chat program 182 Setting up the program 182 Setting up the server 183 Setting up the server 183 Events and Broadcasting Ensuring reception 190 A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation 215 Validation 216 Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 222		JSON	177
WebSockets are HTML5: Older browsers don't support them         177           The solution         178           Summary         178           Chapter 11: Socket.IO         179           Transports         186           Disconnected versus closed         180           Events         180           Namespaces         181           A chat program         182           Setting up the program         183           Setting up the server         182           Setting up the client         183           Ensuring reception         190           A DJ-by-turns application         191           Extending the chat         191           Integrating with the Grooveshark API         193           Playing         196           Summary         201           PART IV: DATABASES         203           Chapter 12: MongoDB         205           Installation         207           Accessing MongoDB: A user authentication example         208           Setting up the application         208           Creating the Express app         208           Connecting to MongoDB         212           Creating documents         214		Reconnections	177
The solution   178   Summary   178   Summary   178   Chapter 11: Socket.IO   179		Broadcasting	177
The solution   178   Summary   178   Summary   178   Chapter 11: Socket.IO   179		WebSockets are HTML5: Older browsers don't support them	177
Chapter 11: Socket.IO  Transports  Disconnected versus closed Events  Namespaces  A chat program  Setting up the program  Setting up the server  Setting up the client Events and Broadcasting Ensuring reception  A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Setting up the application  Setting up the application  Creating the Express app Connecting to MongoDB Creating to MongoDB Setting up the application Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys  222			178
Transports  Disconnected versus closed Events Namespaces 188 A chat program 182 Setting up the program 183 Setting up the server 184 Setting up the client 185 Events and Broadcasting Ensuring reception 190 A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 222		Summary	178
Transports  Disconnected versus closed Events Namespaces 188 A chat program 182 Setting up the program 183 Setting up the server 184 Setting up the client 185 Events and Broadcasting Ensuring reception 190 A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 222	Chapter 11	: Socket.IO	179
Disconnected versus closed Events Namespaces 188 Namespaces 181 A chat program 182 Setting up the program 182 Setting up the server 183 Setting up the client 183 Events and Broadcasting Ensuring reception 190 A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 222	•		180
Namespaces       181         A chat program       182         Setting up the program       182         Setting up the server       183         Setting up the client       183         Events and Broadcasting       185         Ensuring reception       190         A DJ-by-turns application       191         Extending the chat       191         Integrating with the Grooveshark API       193         Playing       196         Summary       201         PART IV: DATABASES         Chapter 12: MongoDB       205         Installation       207         Accessing MongoDB: A user authentication example       208         Setting up the application       208         Creating the Express app       208         Connecting to MongoDB       212         Creating documents       212         Finding documents       214         Finding documents       215         Authentication middleware       217         Validation       218         Atomicity       218         Safe mode       215         Introducing Mongoose       220         Defining nested keys       2			180
A chat program Setting up the program Setting up the server Setting up the server Setting up the client Events and Broadcasting Ensuring reception A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys		Events	180
A chat program  Setting up the program  Setting up the server  Setting up the client  Events and Broadcasting  Ensuring reception  A DJ-by-turns application  Extending the chat  Integrating with the Grooveshark API  Playing  Summary  PART IV: DATABASES  Chapter 12: MongoDB  Installation  Accessing MongoDB: A user authentication example  Setting up the application  Creating the Express app  Connecting to MongoDB  Creating documents  Finding documents  Authentication middleware  Validation  Atomicity  Safe mode  Introducing Mongoose  Defining a model  Defining nested keys  188  182  182  283  284  285  285  287  287  288  298  209  209  209  209  209  209		Namespaces	181
Setting up the program  Setting up the server  Setting up the server  Setting up the client  Events and Broadcasting  Ensuring reception  A DJ-by-turns application  Extending the chat  Integrating with the Grooveshark API  Playing  Summary  PART IV: DATABASES  Chapter 12: MongoDB  Installation  Accessing MongoDB: A user authentication example  Setting up the application  Creating the Express app  Connecting to MongoDB  Creating documents  Finding documents  Authentication middleware  Validation  Atomicity  Safe mode  Introducing Mongoose  Defining a model  Defining nested keys  222		A chat program	182
Setting up the server Setting up the client Events and Broadcasting Ensuring reception A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys  182 193 185 185 185 185 185 185 185 185 185 185			182
Events and Broadcasting Ensuring reception A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 219 197 197 298 209 209 209 201 209 209 209 209 209 209 209 209 209 209			182
Ensuring reception A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys		Setting up the client	183
A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys		Events and Broadcasting	185
A DJ-by-turns application Extending the chat Integrating with the Grooveshark API Playing Summary  PART IV: DATABASES  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys		Ensuring reception	190
Integrating with the Grooveshark API Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys 203		A DJ-by-turns application	191
Playing Summary 201  PART IV: DATABASES 203  Chapter 12: MongoDB 205  Installation 207  Accessing MongoDB: A user authentication example 208  Setting up the application 208  Creating the Express app 208  Connecting to MongoDB 212  Creating documents 214  Finding documents 215  Authentication middleware 217  Validation 218  Atomicity 219  Safe mode 219  Introducing Mongoose 220  Defining a model 220  Defining nested keys 222		Extending the chat	191
Summary  PART IV: DATABASES  Chapter 12: MongoDB  Installation  Accessing MongoDB: A user authentication example  Setting up the application  Creating the Express app  Connecting to MongoDB  Creating documents  Finding documents  Finding documents  Authentication middleware  Validation  Atomicity  Safe mode  Introducing Mongoose  Defining a model  Defining nested keys  203  205  207  Accessing MongoDB  208  208  208  208  208  208  208  20		Integrating with the Grooveshark API	193
PART IV: DATABASES  Chapter 12: MongoDB  Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys  203 205 207 207 208 208 208 208 208 208 208 208 208 209 209 209 209 209 209 200 200 200 200		Playing	196
Chapter 12: MongoDB Installation Accessing MongoDB: A user authentication example Setting up the application Creating the Express app Connecting to MongoDB Creating documents Finding documents Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys  205 207 208 208 209 208 209 209 200 200 200 200 200 200 200 200		Summary	201
Installation 207 Accessing MongoDB: A user authentication example 208 Setting up the application 208 Creating the Express app 208 Connecting to MongoDB 212 Creating documents 214 Finding documents 215 Authentication middleware 217 Validation 218 Atomicity 219 Safe mode 219 Introducing Mongoose 220 Defining a model 220 Defining nested keys 222	PART IV	: DATABASES	203
Installation 207 Accessing MongoDB: A user authentication example 208 Setting up the application 208 Creating the Express app 208 Connecting to MongoDB 212 Creating documents 214 Finding documents 215 Authentication middleware 217 Validation 218 Atomicity 219 Safe mode 219 Introducing Mongoose 220 Defining a model 220 Defining nested keys 222	Chapter 12	: MongoDB	205
Setting up the application       208         Creating the Express app       208         Connecting to MongoDB       212         Creating documents       214         Finding documents       215         Authentication middleware       217         Validation       218         Atomicity       219         Safe mode       219         Introducing Mongoose       220         Defining a model       220         Defining nested keys       222	•	I.	207
Setting up the application       208         Creating the Express app       208         Connecting to MongoDB       212         Creating documents       214         Finding documents       215         Authentication middleware       217         Validation       218         Atomicity       219         Safe mode       219         Introducing Mongoose       220         Defining a model       220         Defining nested keys       222		Accessing MongoDB: A user authentication example	208
Connecting to MongoDB       212         Creating documents       214         Finding documents       215         Authentication middleware       217         Validation       218         Atomicity       219         Safe mode       219         Introducing Mongoose       220         Defining a model       220         Defining nested keys       222			208
Creating documents       214         Finding documents       215         Authentication middleware       217         Validation       218         Atomicity       219         Safe mode       219         Introducing Mongoose       220         Defining a model       220         Defining nested keys       222		Creating the Express app	208
Finding documents Authentication middleware Validation Atomicity Safe mode Introducing Mongoose Defining a model Defining nested keys  215 226 227 228 229 230 240 250 260 260 260 260 260 260 260 260 260 26		Connecting to MongoDB	212
Authentication middleware  Validation  Atomicity  Safe mode  Introducing Mongoose  Defining a model Defining nested keys  218  220  220  221  222		Creating documents	214
Validation 218 Atomicity 219 Safe mode 219 Introducing Mongoose 220 Defining a model 220 Defining nested keys 222		Finding documents	215
Atomicity 219 Safe mode 219 Introducing Mongoose 220 Defining a model 220 Defining nested keys 222		Authentication middleware	217
Safe mode 219 Introducing Mongoose 220 Defining a model 220 Defining nested keys 222		Validation	218
Introducing Mongoose 220 Defining a model 220 Defining nested keys 222			219
Defining a model 220 Defining nested keys 222			219
Defining nested keys 222			220
		Defining a model	220
Defining embedded documents 222			222
		Defining embedded documents	222

	Setting up indexes	222
	Middleware	223
	Inspecting the state of the model	223
	Querying	224
	Extending queries	224
	Sorting	224
	Making Selections	224
	Limiting	225
	Skipping	225
	Populating keys automatically	225
	Casting	225
	A mongoose example	226
	Setting up the application	226
	Refactoring	226
	Setting up models	227
	Summary	229
Chapter 13	•	231
	node-mysql	232
	Setting it up	232
	The Express app	232
	Connecting to MySQL	234
	Initializing the script	234
	Creating data	238
	Fetching data	242
	sequelize	244
	Setting up sequelize	245
	Setting up the Express app	245
	Connecting sequelize	248
	Defining models and synchronizing	249
	Creating data	250
	Retrieving data	253
	Removing data	254
	Wrapping up	256
	Summary	257
Chapter 14	: Redis	259
	Installing Redis	261
	The Redis query language	261
	Data types	262
	Strings	263
	Hashes	263
	Lists	265
	Sets	265
	Sorted sets	266
	Redis and Node	266
	Implementing a social graph with node-redis	267
	Summary	276

#### X CONTENTS

PART V: TESTING	277
Chapter 15: Code Sharing	279
What can be shared?	280
Writing compatible JavaScript	280
Exposing modules	280
Shimming ECMA APIs	282
Shimming Node APIs	283
Shimming browser APIs	284
Cross-browser inheritance	284
Putting it all together: browserbuild	285
A basic example	286
Summary	288
Chapter 16: Testing	289
Simple testing	290
The test subject	290
The test strategy	290
The test program	291
Expect.JS	292
API overview	292
Mocha	294
Testing asynchronous code	295
BDD style	297
TDD style	298
Exports style	298
Taking Mocha to the browser	299
Summary	300