

Node.js

Notes for Professionals

Chapter 15: Executing files or commands with Child Processes

Section 15.1: Spawning a new process to execute a command

To spawn a new process in which you need unbuffered output (e.g. long-running processes) or output over a period of time rather than printing and exiting immediately, use `child_process.spawn()` which in turn provides the `stdout` and `stderr` properties. Both are instances of `stream.Readable`.

The following code is equivalent to using `running` the command `ls -lh /usr`.

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log('stdout: %s', data);
});

ls.stderr.on('data', (data) => {
  console.log('stderr: %s', data);
});

ls.on('close', (code) => {
  console.log('child process exited with code %s', code);
});
```

Another example command:

```
ssh -lrv "archie" /image.jpg
```

Might be written as:

Section 15.2: Spawning a shell to execute a command

To run a command in a shell, in which you required buffered output (i.e. it is `child_process.exec()`), for example, if you wanted to run the command `cat` that would look like this:

```
const exec = require('child_process').exec;
exec('cat file.txt', (err, stdout, stderr) => {
  if (err) {
    console.error('exec error: %s', err);
    return;
  }
  console.log('stdout: %s', stdout);
  console.log('stderr: %s', stderr);
});
```

The function accepts up to three parameters:

Chapter 21: Using Streams

Parameter	Definition
Readable Stream	type of stream where data can be read from
Writable Stream	type of stream where data can be written to
Duplex Stream	type of stream that is both readable and writable
Transform Stream	type of duplex stream that can transform data as it is being read and then written

Section 21.1: Read Data from TextFile with Streams

IO in node is asynchronous, so interacting with the disk and network involves passing callbacks to functions. You might be tempted to write code that serves up a file from disk like this:

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

var server = http.createServer(function (req, res) {
  fs.readFile(__dirname + '/data.txt', function (err, data) {
    res.end(data);
  });
});

server.listen(8080);
```

This code works but it's bulky and buffers up the entire data text file into memory for every request before writing the result back to clients. If data.txt is very large, your program could start eating a lot of memory as it serves lots of users concurrently, particularly for users on slow connections.

The user experience is poor too because users will need to wait for the whole file to be buffered into memory on your server before they can start receiving any contents.

Luckily both of the `(req, res)` arguments are streams, which means we can write this in a much better way using `fs.createReadStream()` instead of `fs.readFile()`:

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

var server = http.createServer(function (req, res) {
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});

server.listen(8080);
```

Here `pipe()` takes care of listening for `'data'` and `'end'` events from the `fs.createReadStream()`. This code is not cleaner, but now the `data.txt` file will be written to clients one chunk at a time immediately as they are received from the disk.

Section 21.2: Piping streams

Readable streams can be "piped", or connected, to writable streams. This makes data flow from the source to the destination stream without much effort.

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

var readable = fs.createReadStream('file.txt');
```

Thanks to Notes for Professionals

Chapter 57: TCP Sockets

Section 57.1: A simple TCP server

```
// Include Node.js' net module.
const net = require('net');
// The port on which the server is listening.
const port = 8080;

// Use net.createServer() in your code. This is just for illustration purpose.
// Create a new TCP server.
const server = net.createServer();
// The server listens to a socket for a client to make a connection request.
server.listen(port, function() {
  console.log('Server listening for connection requests on socket localhost:%s', port);
});

// When a client requests a connection with the server, the server creates a new
// socket dedicated to that client.
server.on('connection', function(socket) {
  console.log('A new connection has been established.');
```

```
// Now that a TCP connection has been established, the server can send data to
// the client by writing to its socket.
socket.write('Hello, client!');
```

```
// The server can also receive data from the client by reading from its socket.
socket.on('data', function(chunk) {
  console.log('Data received from client: %s', chunk.toString());
});
```

```
// When the client requests to end the TCP connection with the server, the server
// ends the connection.
socket.on('end', function() {
  console.log('Closing connection with the client.');
```

```
// Don't forget to catch error, for your own sake.
socket.on('error', function(err) {
  console.log('Error: %s', err);
});
```

Section 57.2: A simple TCP client

```
// Include Node.js' net module.
const net = require('net');
// The port number and hostname of the server.
const port = 8080;
const host = 'localhost';
```

```
// Create a new TCP client.
const client = net.Socket();
// Send a connection request to the server.
client.connect({ port: port, host: host }, function() {
  // If there is no error, the server has accepted the request and created a new
  // socket dedicated to us.
  console.log('TCP connection established with the server.');
```

Thanks to Notes for Professionals

300+ pages
of professional hints and tricks

Contents

About	1
Chapter 1: Getting started with Node.js	2
Section 1.1: Hello World HTTP server	4
Section 1.2: Hello World command line	5
Section 1.3: Hello World with Express	6
Section 1.4: Installing and Running Node.js	6
Section 1.5: Debugging Your NodeJS Application	7
Section 1.6: Hello World basic routing	7
Section 1.7: Hello World in the REPL	8
Section 1.8: Deploying your application online	9
Section 1.9: Core modules	9
Section 1.10: TLS Socket: server and client	14
Section 1.11: How to get a basic HTTPS web server up and running!	16
Chapter 2: npm	19
Section 2.1: Installing packages	19
Section 2.2: Uninstalling packages	22
Section 2.3: Setting up a package configuration	23
Section 2.4: Running scripts	24
Section 2.5: Basic semantic versioning	24
Section 2.6: Publishing a package	25
Section 2.7: Removing extraneous packages	26
Section 2.8: Listing currently installed packages	26
Section 2.9: Updating npm and packages	26
Section 2.10: Scopes and repositories	27
Section 2.11: Linking projects for faster debugging and development	27
Section 2.12: Locking modules to specific versions	28
Section 2.13: Setting up for globally installed packages	28
Chapter 3: Web Apps With Express	30
Section 3.1: Getting Started	30
Section 3.2: Basic routing	31
Section 3.3: Modular express application	32
Section 3.4: Using a Template Engine	33
Section 3.5: JSON API with ExpressJS	34
Section 3.6: Serving static files	35
Section 3.7: Adding Middleware	36
Section 3.8: Error Handling	36
Section 3.9: Getting info from the request	37
Section 3.10: Error handling in Express	38
Section 3.11: Hook: How to execute code before any req and after any res	38
Section 3.12: Setting cookies with cookie-parser	39
Section 3.13: Custom middleware in Express	39
Section 3.14: Named routes in Django-style	39
Section 3.15: Hello World	40
Section 3.16: Using middleware and the next callback	40
Section 3.17: Error handling	42
Section 3.18: Handling POST Requests	43
Chapter 4: Filesystem I/O	45

Section 4.1: Asynchronously Read from Files	45
Section 4.2: Listing Directory Contents with <code>readdir</code> or <code>readdirSync</code>	45
Section 4.3: Copying files by piping streams	46
Section 4.4: Reading from a file synchronously	47
Section 4.5: Check Permissions of a File or Directory	47
Section 4.6: Checking if a file or a directory exists	48
Section 4.7: Determining the line count of a text file	49
Section 4.8: Reading a file line by line	49
Section 4.9: Avoiding race conditions when creating or using an existing directory	49
Section 4.10: Cloning a file using streams	50
Section 4.11: Writing to a file using <code>writeFile</code> or <code>writeFileSync</code>	51
Section 4.12: Changing contents of a text file	51
Section 4.13: Deleting a file using <code>unlink</code> or <code>unlinkSync</code>	52
Section 4.14: Reading a file into a Buffer using streams	52
Chapter 5: Exporting and Consuming Modules	53
Section 5.1: Creating a hello-world.js module	53
Section 5.2: Loading and using a module	54
Section 5.3: Folder as a module	55
Section 5.4: Every module injected only once	55
Section 5.5: Module loading from <code>node_modules</code>	56
Section 5.6: Building your own modules	56
Section 5.7: Invalidating the module cache	57
Chapter 6: Exporting and Importing Module in node.js	58
Section 6.1: Exporting with ES6 syntax	58
Section 6.2: Using a simple module in node.js	58
Chapter 7: How modules are loaded	59
Section 7.1: Global Mode	59
Section 7.2: Loading modules	59
Chapter 8: Cluster Module	60
Section 8.1: Hello World	60
Section 8.2: Cluster Example	60
Chapter 9: Readline	62
Section 9.1: Line-by-line file reading	62
Section 9.2: Prompting user input via CLI	62
Chapter 10: package.json	63
Section 10.1: Exploring package.json	63
Section 10.2: Scripts	66
Section 10.3: Basic project definition	67
Section 10.4: Dependencies	67
Section 10.5: Extended project definition	68
Chapter 11: Event Emitters	69
Section 11.1: Basics	69
Section 11.2: Get the names of the events that are subscribed to	69
Section 11.3: HTTP Analytics through an Event Emitter	70
Section 11.4: Get the number of listeners registered to listen for a specific event	70
Chapter 12: Autoreload on changes	72
Section 12.1: Autoreload on source code changes using <code>nodemon</code>	72
Section 12.2: <code>Browsersync</code>	72
Chapter 13: Environment	74

Section 13.1: Accessing environment variables	74
Section 13.2: process.argv command line arguments	74
Section 13.3: Loading environment properties from a "property file"	75
Section 13.4: Using different Properties/Configuration for different environments like dev, qa, staging etc	75
Chapter 14: Callback to Promise	77
Section 14.1: Promisifying a callback	77
Section 14.2: Manually promisifying a callback	77
Section 14.3: setTimeout promisified	78
Chapter 15: Executing files or commands with Child Processes	79
Section 15.1: Spawning a new process to execute a command	79
Section 15.2: Spawning a shell to execute a command	79
Section 15.3: Spawning a process to run an executable	80
Chapter 16: Exception handling	82
Section 16.1: Handling Exception In Node.js	82
Section 16.2: Unhandled Exception Management	83
Section 16.3: Errors and Promises	84
Chapter 17: Keep a node application constantly running	86
Section 17.1: Use PM2 as a process manager	86
Section 17.2: Running and stopping a Forever daemon	87
Section 17.3: Continuous running with nohup	88
Chapter 18: Uninstalling Node.js	89
Section 18.1: Completely uninstall Node.js on Mac OSX	89
Section 18.2: Uninstall Node.js on Windows	89
Chapter 19: nvm - Node Version Manager	90
Section 19.1: Install NVM	90
Section 19.2: Check NVM version	90
Section 19.3: Installing an specific Node version	90
Section 19.4: Using an already installed node version	90
Section 19.5: Install nvm on Mac OSX	91
Section 19.6: Run any arbitrary command in a subshell with the desired version of node	91
Section 19.7: Setting alias for node version	92
Chapter 20: http	93
Section 20.1: http server	93
Section 20.2: http client	94
Chapter 21: Using Streams	95
Section 21.1: Read Data from TextFile with Streams	95
Section 21.2: Piping streams	95
Section 21.3: Creating your own readable/writable stream	96
Section 21.4: Why Streams?	97
Chapter 22: Deploying Node.js applications in production	99
Section 22.1: Setting NODE_ENV="production"	99
Section 22.2: Manage app with process manager	100
Section 22.3: Deployment using process manager	100
Section 22.4: Deployment using PM2	101
Section 22.5: Using different Properties/Configuration for different environments like dev, qa, staging etc	102
Section 22.6: Taking advantage of clusters	103
Chapter 23: Securing Node.js applications	104

Section 23.1: SSL/TLS in Node.js	104
Section 23.2: Preventing Cross Site Request Forgery (CSRF)	104
Section 23.3: Setting up an HTTPS server	105
Section 23.4: Using HTTPS	107
Section 23.5: Secure express.js 3 Application	107
Chapter 24: Mongoose Library	109
Section 24.1: Connect to MongoDB Using Mongoose	109
Section 24.2: Find Data in MongoDB Using Mongoose, Express.js Routes and \$text Operator	109
Section 24.3: Save Data to MongoDB using Mongoose and Express.js Routes	111
Section 24.4: Find Data in MongoDB Using Mongoose and Express.js Routes	113
Section 24.5: Useful Mongoose functions	115
Section 24.6: Indexes in models	115
Section 24.7: find data in mongodb using promises	117
Chapter 25: async.js	120
Section 25.1: Parallel : multi-tasking	120
Section 25.2: async.each(To handle array of data efficiently)	121
Section 25.3: Series : independent mono-tasking	122
Section 25.4: Waterfall : dependent mono-tasking	123
Section 25.5: async.times(To handle for loop in better way)	124
Section 25.6: async.series(To handle events one by one)	124
Chapter 26: File upload	125
Section 26.1: Single File Upload using multer	125
Section 26.2: Using formidable module	126
Chapter 27: Socket.io communication	128
Section 27.1: "Hello world!" with socket messages	128
Chapter 28: Mongodb integration	129
Section 28.1: Simple connect	129
Section 28.2: Simple connect, using promises	129
Section 28.3: Connect to MongoDB	129
Section 28.4: Insert a document	130
Section 28.5: Read a collection	131
Section 28.6: Update a document	131
Section 28.7: Delete a document	132
Section 28.8: Delete multiple documents	132
Chapter 29: Handling POST request in Node.js	134
Section 29.1: Sample node.js server that just handles POST requests	134
Chapter 30: Simple REST based CRUD API	135
Section 30.1: REST API for CRUD in Express 3+	135
Chapter 31: Template frameworks	136
Section 31.1: Nunjucks	136
Chapter 32: Node.js Architecture & Inner Workings	138
Section 32.1: Node.js - under the hood	138
Section 32.2: Node.js - in motion	138
Chapter 33: Debugging Node.js application	139
Section 33.1: Core node.js debugger and node inspector	139
Chapter 34: Node server without framework	142
Section 34.1: Framework-less node server	142
Section 34.2: Overcoming CORS Issues	143

[Click here to download full PDF material](#)