

Node.js

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	3
Section 1.2: Hello World command line	5
Section 1.3: Hello World with Express	5
Section 1.4: Installing and Running Node.js	6
Section 1.5: Debugging Your NodeJS Application	6
Section 1.6: Hello World basic routing	7
Section 1.7: Hello World in the REPL	8
Section 1.8: Deploying your application online	8
Section 1.9: Core modules	9
Section 1.10: TLS Socket: server and client	13
Section 1.11: How to get a basic HTTPS web server up and running!	16
Chapter 2: npm	18
Section 2.1: Installing packages	18
Section 2.2: Uninstalling packages	21
Section 2.3: Setting up a package configuration	22
Section 2.4: Running scripts	23
Section 2.5: Basic semantic versioning	23
Section 2.6: Publishing a package	24
Section 2.7: Removing extraneous packages	24
Section 2.8: Listing currently installed packages	25
Section 2.9: Updating npm and packages	25
Section 2.10: Scopes and repositories	26
Section 2.11: Linking projects for faster debugging and development	26
Section 2.12: Locking modules to specific versions	27
Section 2.13: Setting up for globally installed packages	27
Chapter 3: Web Apps With Express	29
Section 3.1: Getting Started	29
Section 3.2: Basic routing	29
Section 3.3: Modular express application	31
Section 3.4: Using a Template Engine	32
Section 3.5: JSON API with ExpressJS	33
Section 3.6: Serving static files	34
Section 3.7: Adding Middleware	35
Section 3.8: Error Handling	35
Section 3.9: Getting info from the request	36
Section 3.10: Error handling in Express	36
Section 3.11: Hook: How to execute code before any req and after any res	37
Section 3.12: Setting cookies with cookie-parser	37
Section 3.13: Custom middleware in Express	38
Section 3.14: Named routes in Django-style	38
Section 3.15: Hello World	39
Section 3.16: Using middleware and the next callback	39
Section 3.17: Error handling	41
Section 3.18: Handling POST Requests	42
Chapter 4: Filesystem I/O	44

Section 4.1: Asynchronously Read from Files	44
Section 4.2: Listing Directory Contents with readdir or readdirSync	44
Section 4.3: Copying files by piping streams	45
Section 4.4: Reading from a file synchronously	46
Section 4.5: Check Permissions of a File or Directory	46
Section 4.6: Checking if a file or a directory exists	47
Section 4.7: Determining the line count of a text file	47
Section 4.8: Reading a file line by line	48
Section 4.9: Avoiding race conditions when creating or using an existing directory	48
Section 4.10: Cloning a file using streams	49
Section 4.11: Writing to a file using writeFile or writeFileSync	49
Section 4.12: Changing contents of a text file	50
Section 4.13: Deleting a file using unlink or unlinkSync	50
Section 4.14: Reading a file into a Buffer using streams	50
Chapter 5: Exporting and Consuming Modules	52
Section 5.1: Creating a hello-world.js module	52
Section 5.2: Loading and using a module	53
Section 5.3: Folder as a module	54
Section 5.4: Every module injected only once	54
Section 5.5: Module loading from node_modules	55
Section 5.6: Building your own modules	55
Section 5.7: Invalidating the module cache	56
Chapter 6: Exporting and Importing Module in node.js	57
Section 6.1: Exporting with ES6 syntax	57
Section 6.2: Using a simple module in node.js	57
Chapter 7: Installing Node.js	58
Section 7.1: Using Node Version Manager (nvm)	58
Section 7.2: Installing Node.js on Mac using package manager	59
Section 7.3: Installing Node.js on Windows	59
Section 7.4: Install Node.js on Ubuntu	60
Section 7.5: Installing Node.js with n	60
Section 7.6: Install Node.js From Source with APT package manager	61
Section 7.7: Install Node.js from source on Centos, RHEL and Fedora	61
Section 7.8: Installing with Node Version Manager under Fish Shell with Oh My Fish!	62
Section 7.9: Installing Node.js on Raspberry PI	62
Chapter 8: MySQL integration	64
Section 8.1: Connect to MySQL	64
Section 8.2: Using a connection pool	64
Section 8.3: Query a connection object with parameters	65
Section 8.4: Query a connection object without parameters	66
Section 8.5: Run a number of queries with a single connection from a pool	66
Section 8.6: Export Connection Pool	66
Section 8.7: Return the query when an error occurs	67
Chapter 9: Readline	68
Section 9.1: Line-by-line file reading	68
Section 9.2: Prompting user input via CLI	68
Chapter 10: package.json	69
Section 10.1: Exploring package.json	69
Section 10.2: Scripts	72
Section 10.3: Basic project definition	73

Section 10.4: Dependencies	73
Section 10.5: Extended project definition	74
Chapter 11: Event Emitters	75
Section 11.1: Basics	75
Section 11.2: Get the names of the events that are subscribed to	75
Section 11.3: HTTP Analytics through an Event Emitter	76
Section 11.4: Get the number of listeners registered to listen for a specific event	76
Chapter 12: Autoreload on changes	78
Section 12.1: Autoreload on source code changes using nodemon	78
Section 12.2: Browsersync	78
Chapter 13: Environment	80
Section 13.1: Accessing environment variables	80
Section 13.2: process.argv command line arguments	80
Section 13.3: Loading environment properties from a "property file"	81
Section 13.4: Using different Properties/Configuration for different environments like dev, qa, staging etc	81
Chapter 14: Callback to Promise	83
Section 14.1: Promisifying a callback	83
Section 14.2: Manually promisifying a callback	83
Section 14.3: setTimeout promisified	84
Chapter 15: Executing files or commands with Child Processes	85
Section 15.1: Spawning a new process to execute a command	85
Section 15.2: Spawning a shell to execute a command	85
Section 15.3: Spawning a process to run an executable	86
Chapter 16: Cluster Module	88
Section 16.1: Hello World	88
Section 16.2: Cluster Example	88
Chapter 17: Exception handling	90
Section 17.1: Handling Exception In Node.js	90
Section 17.2: Unhandled Exception Management	91
Section 17.3: Errors and Promises	92
Chapter 18: Keep a node application constantly running	94
Section 18.1: Use PM2 as a process manager	94
Section 18.2: Running and stopping a Forever daemon	95
Section 18.3: Continuous running with nohup	96
Chapter 19: Uninstalling Node.js	97
Section 19.1: Completely uninstall Node.js on Mac OSX	97
Section 19.2: Uninstall Node.js on Windows	97
Chapter 20: nvm - Node Version Manager	98
Section 20.1: Install NVM	98
Section 20.2: Check NVM version	98
Section 20.3: Installing an specific Node version	98
Section 20.4: Using an already installed node version	98
Section 20.5: Install nvm on Mac OSX	99
Section 20.6: Run any arbitrary command in a subshell with the desired version of node	99
Section 20.7: Setting alias for node version	100
Chapter 21: http	101
Section 21.1: http server	101
Section 21.2: http client	102

Chapter 22: Using Streams	103
Section 22.1: Read Data from TextFile with Streams	103
Section 22.2: Piping streams	103
Section 22.3: Creating your own readable/writable stream	104
Section 22.4: Why Streams?	105
Chapter 23: Deploying Node.js applications in production	107
Section 23.1: Setting NODE_ENV="production"	107
Section 23.2: Manage app with process manager	108
Section 23.3: Deployment using process manager	108
Section 23.4: Deployment using PM2	109
Section 23.5: Using different Properties/Configuration for different environments like dev, qa, staging etc	110
Section 23.6: Taking advantage of clusters	111
Chapter 24: Securing Node.js applications	112
Section 24.1: SSL/TLS in Node.js	112
Section 24.2: Preventing Cross Site Request Forgery (CSRF)	112
Section 24.3: Setting up an HTTPS server	113
Section 24.4: Using HTTPS	114
Section 24.5: Secure express.js 3 Application	115
Chapter 25: Mongoose Library	116
Section 25.1: Connect to MongoDB Using Mongoose	116
Section 25.2: Find Data in MongoDB Using Mongoose, Express.js Routes and \$text Operator	116
Section 25.3: Save Data to MongoDB using Mongoose and Express.js Routes	118
Section 25.4: Find Data in MongoDB Using Mongoose and Express.js Routes	120
Section 25.5: Useful Mongoose functions	122
Section 25.6: Indexes in models	122
Section 25.7: find data in mongodb using promises	124
Chapter 26: async.js	127
Section 26.1: Parallel : multi-tasking	127
Section 26.2: async.each(To handle array of data efficiently)	128
Section 26.3: Series : independent mono-tasking	129
Section 26.4: Waterfall : dependent mono-tasking	130
Section 26.5: async.times(To handle for loop in better way)	131
Section 26.6: async.series(To handle events one by one)	131
Chapter 27: File upload	132
Section 27.1: Single File Upload using multer	132
Section 27.2: Using formidable module	133
Chapter 28: Socket.io communication	135
Section 28.1: "Hello world!" with socket messages	135
Chapter 29: Mongodb integration	136
Section 29.1: Simple connect	136
Section 29.2: Simple connect, using promises	136
Section 29.3: Connect to MongoDB	136
Section 29.4: Insert a document	137
Section 29.5: Read a collection	137
Section 29.6: Update a document	138
Section 29.7: Delete a document	139
Section 29.8: Delete multiple documents	139
Chapter 30: Handling POST request in Node.js	141

Section 30.1: Sample node.js server that just handles POST requests	141
Chapter 31: Simple REST based CRUD API	142
Section 31.1: REST API for CRUD in Express 3+	142
Chapter 32: Template frameworks	143
Section 32.1: Nunjucks	143
Chapter 33: Node.js Architecture & Inner Workings	145
Section 33.1: Node.js - under the hood	145
Section 33.2: Node.js - in motion	145
Chapter 34: Debugging Node.js application	146
Section 34.1: Core node.js debugger and node inspector	146
Chapter 35: Node server without framework	149
Section 35.1: Framework-less node server	149
Section 35.2: Overcoming CORS Issues	150
Chapter 36: Node.JS with ES6	151
Section 36.1: Node ES6 Support and creating a project with Babel	151
Section 36.2: Use JS es6 on your NodeJS app	152
Chapter 37: Interacting with Console	155
Section 37.1: Logging	155
Chapter 38: Cassandra Integration	157
Section 38.1: Hello world	157
Chapter 39: Creating API's with Node.js	158
Section 39.1: GET api using Express	158
Section 39.2: POST api using Express	158
Chapter 40: Graceful Shutdown	160
Section 40.1: Graceful Shutdown - SIGTERM	160
Chapter 41: Using IISNode to host Node.js Web Apps in IIS	161
Section 41.1: Using an IIS Virtual Directory or Nested Application via <appSettings>	161
Section 41.2: Getting Started	162
Section 41.3: Basic Hello World Example using Express	162
Section 41.4: Using Socket.io with IISNode	164
Chapter 42: CLI	165
Section 42.1: Command Line Options	165
Chapter 43: NodeJS Frameworks	168
Section 43.1: Web Server Frameworks	168
Section 43.2: Command Line Interface Frameworks	168
Chapter 44: grunt	170
Section 44.1: Introduction To GruntJs	170
Section 44.2: Installing gruntplugins	171
Chapter 45: Using WebSocket's with Node.JS	172
Section 45.1: Installing WebSocket's	172
Section 45.2: Adding WebSocket's to your file's	172
Section 45.3: Using WebSocket's and WebSocket Server's	172
Section 45.4: A Simple WebSocket Server Example	172
Chapter 46: metalsmith	173
Section 46.1: Build a simple blog	173
{{ title }}	173
Chapter 47: Parsing command line arguments	174

Section 47.1: Passing action (verb) and values	174
Section 47.2: Passing boolean switches	174
Chapter 48: Client-server communication	175
Section 48.1: /w Express, jQuery and Jade	175
Chapter 49: Node.js Design Fundamental	177
Section 49.1: The Node.js philosophy	177
Chapter 50: Connect to MongoDB	178
Section 50.1: Simple example to Connect mongoDB from Node.JS	178
Section 50.2: Simple way to Connect mongoDB with core Node.JS	178
Chapter 51: Performance challenges	179
Section 51.1: Processing long running queries with Node	179
Chapter 52: Send Web Notification	183
Section 52.1: Send Web notification using GCM (Google Cloud Messaging System)	183
Chapter 53: Remote Debugging in Node.JS	185
Section 53.1: Use the proxy for debugging via port on Linux	185
Section 53.2: NodeJS run configuration	185
Section 53.3: IntelliJ/Webstorm Configuration	185
Chapter 54: Mysql Connection Pool	187
Section 54.1: Using a connection pool without database	187
Chapter 55: Database (MongoDB with Mongoose)	188
Section 55.1: Mongoose connection	188
Section 55.2: Model	188
Section 55.3: Insert data	189
Section 55.4: Read data	189
Chapter 56: Good coding style	191
Section 56.1: Basic program for signup	191
Chapter 57: Restful API Design: Best Practices	195
Section 57.1: Error Handling: GET all resources	195
Chapter 58: Deliver HTML or any other sort of file	197
Section 58.1: Deliver HTML at specified path	197
Chapter 59: TCP Sockets	198
Section 59.1: A simple TCP server	198
Section 59.2: A simple TCP client	198
Chapter 60: Hack	200
Section 60.1: Add new extensions to require()	200
Chapter 61: Bluebird Promises	201
Section 61.1: Converting nodeback library to Promises	201
Section 61.2: Functional Promises	201
Section 61.3: Coroutines (Generators)	201
Section 61.4: Automatic Resource Disposal (Promise.using)	201
Section 61.5: Executing in series	202
Chapter 62: Async/Await	203
Section 62.1: Comparison between Promises and Async/Await	203
Section 62.2: Async Functions with Try-Catch Error Handling	203
Section 62.3: Stops execution at await	204
Section 62.4: Progression from Callbacks	204
Chapter 63: Koa Framework v2	206
Section 63.1: Hello World example	206

Section 63.2: Handling errors using middleware	206
Chapter 64: Unit testing frameworks	207
Section 64.1: Mocha Asynchronous (async/await)	207
Section 64.2: Mocha synchronous	207
Section 64.3: Mocha asynchronous (callback)	207
Chapter 65: ECMAScript 2015 (ES6) with Node.js	208
Section 65.1: const/let declarations	208
Section 65.2: Arrow functions	208
Section 65.3: Arrow Function Example	208
Section 65.4: destructuring	209
Section 65.5: flow	209
Section 65.6: ES6 Class	209
Chapter 66: Routing ajax requests with Express.JS	211
Section 66.1: A simple implementation of AJAX	211
Chapter 67: Sending a file stream to client	213
Section 67.1: Using fs And pipe To Stream Static Files From The Server	213
Section 67.2: Streaming Using fluent-ffmpeg	214
Chapter 68: NodeJS with Redis	215
Section 68.1: Getting Started	215
Section 68.2: Storing Key-Value Pairs	215
Section 68.3: Some more important operations supported by node_redis	217
Chapter 69: Using Browserfy to resolve 'required' error with browsers	219
Section 69.1: Example - file.js	219
Chapter 70: Node.JS and MongoDB.	221
Section 70.1: Connecting To a Database	221
Section 70.2: Creating New Collection	221
Section 70.3: Inserting Documents	222
Section 70.4: Reading	222
Section 70.5: Updating	223
Section 70.6: Deleting	224
Chapter 71: Passport integration	225
Section 71.1: Local authentication	225
Section 71.2: Getting started	226
Section 71.3: Facebook authentication	227
Section 71.4: Simple Username-Password Authentication	228
Section 71.5: Google Passport authentication	228
Chapter 72: Dependency Injection	231
Section 72.1: Why Use Dependency Injection	231
Chapter 73: NodeJS Beginner Guide	232
Section 73.1: Hello World !	232
Chapter 74: Use Cases of Node.js	233
Section 74.1: HTTP server	233
Section 74.2: Console with command prompt	233
Chapter 75: Sequelize.js	235
Section 75.1: Defining Models	235
Section 75.2: Installation	235
Chapter 76: PostgreSQL integration	237
Section 76.1: Connect To PostgreSQL	237

Section 76.2: Query with Connection Object	237
Chapter 77: How modules are loaded	238
Section 77.1: Global Mode	238
Section 77.2: Loading modules	238
Chapter 78: Node.js with Oracle	239
Section 78.1: Connect to Oracle DB	239
Section 78.2: Using a local module for easier querying	239
Section 78.3: Query a connection object without parameters	240
Chapter 79: Synchronous vs Asynchronous programming in nodejs	242
Section 79.1: Using async	242
Chapter 80: Node.js Error Management	243
Section 80.1: try...catch block	243
Section 80.2: Creating Error object	243
Section 80.3: Throwing Error	244
Chapter 81: Node.js v6 New Features and Improvement	245
Section 81.1: Default Function Parameters	245
Section 81.2: Rest Parameters	245
Section 81.3: Arrow Functions	245
Section 81.4: "this" in Arrow Function	246
Section 81.5: Spread Operator	247
Chapter 82: Eventloop	248
Section 82.1: How the concept of event loop evolved	248
Chapter 83: Nodejs History	250
Section 83.1: Key events in each year	250
Chapter 84: passport.js	253
Section 84.1: Example of LocalStrategy in passport.js	253
Chapter 85: Asynchronous programming	254
Section 85.1: Callback functions	254
Section 85.2: Callback hell	256
Section 85.3: Native Promises	257
Section 85.4: Code example	258
Section 85.5: Async error handling	259
Chapter 86: Node.js code for STDIN and STDOUT without using any library	260
Section 86.1: Program	260
Chapter 87: MongoDB Integration for Node.js/Express.js	261
Section 87.1: Installing MongoDB	261
Section 87.2: Creating a Mongoose Model	261
Section 87.3: Querying your Mongo Database	262
Chapter 88: Lodash	263
Section 88.1: Filter a collection	263
Chapter 89: csv parser in node js	264
Section 89.1: Using FS to read in a CSV	264
Chapter 90: Loopback - REST Based connector	265
Section 90.1: Adding a web based connector	265
Chapter 91: Running node.js as a service	266
Section 91.1: Node.js as a systemd daemon	266
Chapter 92: Node.js with CORS	267
Section 92.1: Enable CORS in express.js	267

Chapter 93: Getting started with Nodes profiling	268
Section 93.1: Profiling a simple node application	268
Chapter 94: Node.js Performance	270
Section 94.1: Enable gzip	270
Section 94.2: Event Loop	270
Section 94.3: Increase maxSockets	271
Chapter 95: Yarn Package Manager	273
Section 95.1: Creating a basic package	273
Section 95.2: Yarn Installation	273
Section 95.3: Install package with Yarn	275
Chapter 96: OAuth 2.0	276
Section 96.1: OAuth 2 with Redis Implementation - grant_type: password	276
Chapter 97: Node JS Localization	282
Section 97.1: using i18n module to maintains localization in node js app	282
Chapter 98: Deploying Node.js application without downtime.	283
Section 98.1: Deployment using PM2 without downtime	283
Chapter 99: Node.js (express.js) with angular.js Sample code	285
Section 99.1: Creating our project	285
Chapter 100: NodeJs Routing	288
Section 100.1: Express Web Server Routing	288
Chapter 101: Creating a Node.js Library that Supports Both Promises and Error-First	
Callbacks	292
Section 101.1: Example Module and Corresponding Program using Bluebird	292
Chapter 102: MSSQL Intergration	295
Section 102.1: Connecting with SQL via. mssql npm module	295
Chapter 103: Project Structure	297
Section 103.1: A simple nodejs application with MVC and API	297
Chapter 104: Avoid callback hell	299
Section 104.1: Async module	299
Section 104.2: Async Module	299
Chapter 105: Arduino communication with nodeJs	301
Section 105.1: Node Js communication with Arduino via serialport	301
Chapter 106: N-API	303
Section 106.1: Hello to N-API	303
Chapter 107: Multithreading	305
Section 107.1: Cluster	305
Section 107.2: Child Process	305
Chapter 108: Windows authentication under node.js	307
Section 108.1: Using activedirectory	307
Chapter 109: Require()	308
Section 109.1: Beginning require() use with a function and file	308
Section 109.2: Beginning require() use with an NPM package	309
Chapter 110: Route-Controller-Service structure for ExpressJS	310
Section 110.1: Model-Routes-Controllers-Services Directory Structure	310
Section 110.2: Model-Routes-Controllers-Services Code Structure	310
Chapter 111: Push notifications	312
Section 111.1: Web notification	312

Section 111.2: Apple	313
Credits	314
You may also like	319

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:

<http://GoalKicker.com/NodeJSBook>

This *Node.js Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Node.js group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Node.js

Version Release Date

v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21
v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12
v6.3.1	2016-07-21
v6.3.0	2016-07-06
v6.2.2	2016-06-16
v6.2.1	2016-06-02
v6.2.0	2016-05-17
v6.1.0	2016-05-05
v6.0.0	2016-04-26
v5.12.0	2016-06-23
v5.11.1	2016-05-05
v5.11.0	2016-04-21

v5.10.1	2016-04-05
v5.10	2016-04-01
v5.9	2016-03-16
v5.8	2016-03-09
v5.7	2016-02-23
v5.6	2016-02-09
v5.5	2016-01-21
v5.4	2016-01-06
v5.3	2015-12-15
v5.2	2015-12-09
v5.1	2015-11-17
v5.0	2015-10-29
v4.4	2016-03-08
v4.3	2016-02-09
v4.2	2015-10-12
v4.1	2015-09-17
v4.0	2015-09-08
io.js v3.3	2015-09-02
io.js v3.2	2015-08-25
io.js v3.1	2015-08-19
io.js v3.0	2015-08-04
io.js v2.5	2015-07-28
io.js v2.4	2015-07-17
io.js v2.3	2015-06-13
io.js v2.2	2015-06-01
io.js v2.1	2015-05-24
io.js v2.0	2015-05-04
io.js v1.8	2015-04-21
io.js v1.7	2015-04-17
io.js v1.6	2015-03-20
io.js v1.5	2015-03-06
io.js v1.4	2015-02-27
io.js v1.3	2015-02-20
io.js v1.2	2015-02-11
io.js v1.1	2015-02-03
io.js v1.0	2015-01-14
v0.12	2016-02-09
v0.11	2013-03-28
v0.10	2013-03-11
v0.9	2012-07-20
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

Section 1.1: Hello World HTTP server

First, [install Node.js](#) for your platform.

In this example we'll create an HTTP server listening on port 1337, which sends `Hello, World!` to the browser. Note

that, instead of using port 1337, you can use any port number of your choice which is currently not in use by any other service.

The `http` module is a Node.js **core module** (a module included in Node.js's source, that does not require installing additional resources). The `http` module provides the functionality to create an HTTP server using the `http.createServer()` method. To create the application, create a file containing the following JavaScript code.

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

  // 1. Tell the browser everything is OK (Status code 200), and the data is in plain text
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });

  // 2. Write the announced text to the body of the page
  response.write('Hello, World!\n');

  // 3. Tell the server that all of the response headers and body have been sent
  response.end();

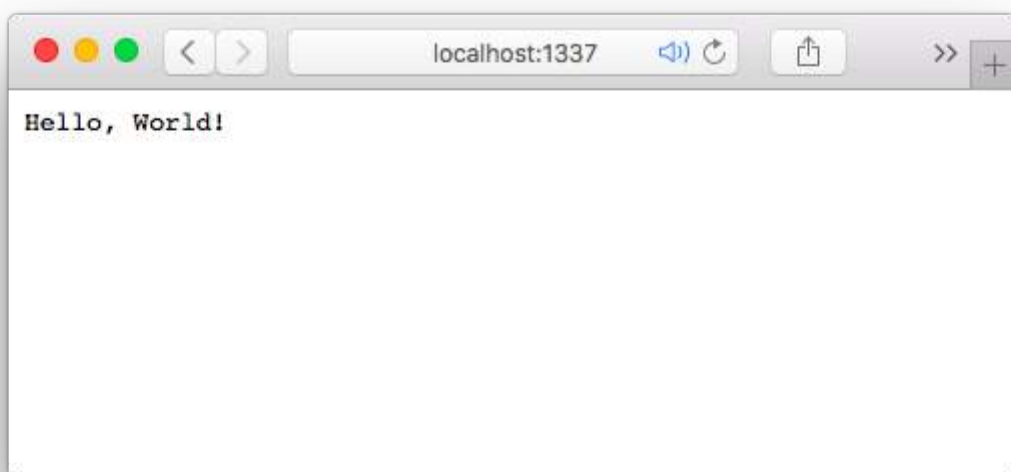
}).listen(1337); // 4. Tells the server what port to be on
```

Save the file with any file name. In this case, if we name it `hello.js` we can run the application by going to the directory the file is in and using the following command:

```
node hello.js
```

The created server can then be accessed with the URL <http://localhost:1337> or <http://127.0.0.1:1337> in the browser.

A simple web page will appear with a “Hello, World!” text at the top, as shown in the screenshot below.



[Editable online example.](#)

Section 1.2: Hello World command line

Node.js can also be used to create command line utilities. The example below reads the first argument from the command line and prints a Hello message.

To run this code on an Unix System:

1. Create a new file and paste the code below. The filename is irrelevant.
2. Make this file executable with `chmod 700 FILE_NAME`
3. Run the app with `./APP_NAME David`

On Windows you do step 1 and run it with `node APP_NAME David`

```
#!/usr/bin/env node

'use strict';

/*
  The command line arguments are stored in the `process.argv` array,
  which has the following structure:
  [0] The path of the executable that started the Node.js process
  [1] The path to this application
  [2-n] the command line arguments

  Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
  src: https://nodejs.org/api/process.html#process_process_argv
*/

// Store the first argument as username.
var username = process.argv[2];

// Check if the username hasn't been provided.
if (!username) {

  // Extract the filename
  var appName = process.argv[1].split(require('path').sep).pop();

  // Give the user an example on how to use the app.
  console.error('Missing argument! Example: %s YOUR_NAME', appName);

  // Exit the app (success: 0, error: 1).
  // An error will stop the execution chain. For example:
  //   ./app.js && ls      -> won't execute ls
  //   ./app.js David && ls -> will execute ls
  process.exit(1);
}

// Print the message to the console.
console.log('Hello %s!', username);
```

Section 1.3: Hello World with Express

The following example uses Express to create an HTTP server listening on port 3000, which responds with "Hello, World!". Express is a commonly-used web framework that is useful for creating HTTP APIs.

First, create a new folder, e.g. myApp. Go into myApp and make a new JavaScript file containing the following code (let's name it `hello.js` for example). Then install the express module using `npm install --save express` from the command line. Refer to [this documentation](#) for more information on how to install packages.

```
// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback function
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// Make the app listen on port 3000
app.listen(port, function() {
  console.log('Server listening on http://localhost:' + port);
});
```

From the command line, run the following command:

```
node hello.js
```

Open your browser and navigate to `http://localhost:3000` or `http://127.0.0.1:3000` to see the response.

For more information about the Express framework, you can check the [Web Apps With Express](#) section

Section 1.4: Installing and Running Node.js

To begin, install Node.js on your development computer.

Windows: Navigate to the [download page](#) and download/run the installer.

Mac: Navigate to the [download page](#) and download/run the installer. Alternatively, you can install Node via Homebrew using `brew install node`. Homebrew is a command-line package manager for Macintosh, and more information about it can be found on the [Homebrew website](#).

Linux: Follow the instructions for your distro on the [command line installation page](#).

Running a Node Program

To run a Node.js program, simply run `node app.js` or `nodejs app.js`, where `app.js` is the filename of your node app source code. You do not need to include the `.js` suffix for Node to find the script you'd like to run.

Alternatively under UNIX-based operating systems, a Node program may be executed as a terminal script. To do so, it needs to begin with a shebang pointing to the Node interpreter, such as `#!/usr/bin/env node`. The file also has to be set as executable, which can be done using `chmod`. Now the script can be directly run from the command line.

Section 1.5: Debugging Your NodeJS Application

You can use the node-inspector. Run this command to install it via npm:

```
npm install -g node-inspector
```

Then you can debug your application using

```
node-debug app.js
```

The Github repository can be found here: <https://github.com/node-inspector/node-inspector>

Debugging natively

You can also debug node.js natively by starting it like this:

```
node debug your-script.js
```

To breakpoint your debugger exactly in a code line you want, use this:

```
debugger;
```

For more information see [here](#).

In node.js 8 use the following command:

```
node --inspect-brk your-script.js
```

Then open about `://inspect` in a recent version of Google Chrome and select your Node script to get the debugging experience of Chrome's DevTools.

Section 1.6: Hello World basic routing

Once you understand how to create an [HTTP Server](#) with node, it's important to understand how to make it "do" things based on the path that a user has navigated to. This phenomenon is called, "routing".

The most basic example of this would be to check `if (request.url === 'some/path/here')`, and then call a function that responds with a new file.

An example of this can be seen here:

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

function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
}

http.createServer(function (request, response) {

  if (request.url === '/') {
    return index(request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

If you continue to define your "routes" like this, though, you'll end up with one massive callback function, and we don't want a giant mess like that, so let's see if we can clean this up.

First, let's store all of our routes in an object:

```
var routes = {
  '/': function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
  },
  '/foo': function foo (request, response) {
    response.writeHead(200);
    response.end('You are now viewing "foo"');
  }
}
```

Now that we've stored 2 routes in an object, we can now check for them in our main callback:

```
http.createServer(function (request, response) {

  if (request.url in routes) {
    return routes[request.url](request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Now every time you try to navigate your website, it will check for the existence of that path in your routes, and it will call the respective function. If no route is found, the server will respond with a 404 (Not Found).

And there you have it--routing with the HTTP Server API is very simple.

Section 1.7: Hello World in the REPL

When called without arguments, Node.js starts a REPL (Read-Eval-Print-Loop) also known as the *"Node shell"*.

At a command prompt type node.

```
$ node
>
```

At the Node shell prompt `>` type "Hello World!"

```
$ node
> "Hello World!"
'Hello World!'
```

Section 1.8: Deploying your application online

When you deploy your app to a (Node.js-specific) hosted environment, this environment usually offers a PORT-environment variable that you can use to run your server on. Changing the port number to `process.env.PORT` allows you to access the application.

For example,

```
http.createServer(function(request, response) {
  // your server code
}).listen(process.env.PORT);
```

Also, if you would like to access this offline while debugging, you can use this:

```
http.createServer(function(request, response) {  
  // your server code  
}).listen(process.env.PORT || 3000);
```

where 3000 is the offline port number.

Section 1.9: Core modules

Node.js is a Javascript engine (Google's V8 engine for Chrome, written in C++) that allows to run Javascript outside the browser. While numerous libraries are available for extending Node's functionalities, the engine comes with a set of *core modules* implementing basic functionalities.

There's currently 34 core modules included in Node:

```
[ 'assert',  
  'buffer',  
  'c/c++_addons',  
  'child_process',  
  'cluster',  
  'console',  
  'crypto',  
  'deprecated_apis',  
  'dns',  
  'domain',  
  'Events',  
  'fs',  
  'http',  
  'https',  
  'module',  
  'net',  
  'os',  
  'path',  
  'punycode',  
  'querystring',  
  'readline',  
  'repl',  
  'stream',  
  'string_decoder',  
  'timers',  
  'tls_(ssl)',  
  'tracing',  
  'tty',  
  'dgram',  
  'url',  
  'util',  
  'v8',  
  'vm',  
  'zlib' ]
```

This list was obtained from the Node documentation API <https://nodejs.org/api/all.html> (JSON file: <https://nodejs.org/api/all.json>).

All core modules at-a-glance

assert

The assert module provides a simple set of assertion tests that can be used to test invariants.

buffer

Prior to the introduction of [TypedArray](#) in ECMAScript 2015 (ES6), the JavaScript language had no mechanism for reading or manipulating streams of binary data. The Buffer class was introduced as part of the Node.js API to make it possible to interact with octet streams in the context of things like TCP streams and file system operations.

Now that [TypedArray](#) has been added in ES6, the Buffer class implements the

```
Uint8Array
```

API in a manner that is more optimized and suitable for Node.js' use cases.

c/c++_addons

Node.js Addons are dynamically-linked shared objects, written in C or C++, that can be loaded into Node.js using the `require()` function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

child_process

The `child_process` module provides the ability to spawn child processes in a manner that is similar, but not identical, to `popen(3)`.

cluster

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load. The cluster module allows you to easily create child processes that all share server ports.

console

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

crypto

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

deprecated_apis

Node.js may deprecate APIs when either: (a) use of the API is considered to be unsafe, (b) an improved alternative API has been made available, or (c) breaking changes to the API are expected in a future major release.

dns

The `dns` module contains functions belonging to two different categories:

1. Functions that use the underlying operating system facilities to perform name resolution, and that do not

necessarily perform any network communication. This category contains only one function: `dns.lookup()`.

2. Functions that connect to an actual DNS server to perform name resolution, and that *always* use the network to perform DNS queries. This category contains all functions in the `dns` module *except* `dns.lookup()`.

domain

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most end users should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Events

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called.

fs

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

http

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses--the user is able to stream data.

https

HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module.

module

Node.js has a simple module loading system. In Node.js, files and modules are in one-to-one correspondence (each file is treated as a separate module).

net

The `net` module provides you with an asynchronous network wrapper. It contains functions for creating both servers and clients (called streams). You can include this module with `require('net');`.

os

The `os` module provides a number of operating system-related utility methods.

path

The `path` module provides utilities for working with file and directory paths.

punycode

The version of the punycode module bundled in Node.js is being deprecated.

querystring

The `querystring` module provides utilities for parsing and formatting URL query strings.

readline

The `readline` module provides an interface for reading data from a Readable stream (such as `process.stdin`) one line at a time.

repl

The `repl` module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications.

stream

A stream is an abstract interface for working with streaming data in Node.js. The `stream` module provides a base API that makes it easy to build objects that implement the stream interface.

There are many stream objects provided by Node.js. For instance, a request to an HTTP server and `process.stdout` are both stream instances.

string_decoder

The `string_decoder` module provides an API for decoding `Buffer` objects into strings in a manner that preserves encoded multi-byte UTF-8 and UTF-16 characters.

timers

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around [the Node.js Event Loop](#).

tls_(ssl)

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL.

tracing

Trace Event provides a mechanism to centralize tracing information generated by V8, Node core, and userspace code.

Tracing can be enabled by passing the `--trace-events-enabled` flag when starting a Node.js application.

tty

The `tty` module provides the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, it will not be necessary or possible to use this module directly.

dgram

The `dgram` module provides an implementation of UDP Datagram sockets.

url

The `url` module provides utilities for URL resolution and parsing.

util

The `util` module is primarily designed to support the needs of Node.js' own internal APIs. However, many of the utilities are useful for application and module developers as well.

v8

The `v8` module exposes APIs that are specific to the version of [V8](#) built into the Node.js binary.

Note: The APIs and implementation are subject to change at any time.

vm

The `vm` module provides APIs for compiling and running code within V8 Virtual Machine contexts. JavaScript code can be compiled and run immediately or compiled, saved, and run later.

Note: The `vm` module is not a security mechanism. ***Do not use it to run untrusted code.***

zlib

The `zlib` module provides compression functionality implemented using Gzip and Deflate/Inflate.

Section 1.10: TLS Socket: server and client

The only major differences between this and a regular TCP connection are the private Key and the public certificate that you'll have to set into an option object.

How to Create a Key and Certificate

The first step in this security process is the creation of a private Key. And what is this private key? Basically, it's a set of random noise that's used to encrypt information. In theory, you could create one key, and use it to encrypt whatever you want. But it is best practice to have different keys for specific things. Because if someone steals your private key, it's similar to having someone steal your house keys. Imagine if you used the same key to lock your car, garage, office, etc.

```
openssl genrsa -out private-key.pem 1024
```

Once we have our private key, we can create a CSR (certificate signing request), which is our request to have the private key signed by a fancy authority. That is why you have to input information related to your company. This information will be seen by the signing authority, and used to verify you. In our case, it doesn't matter what you type, since in the next step we're going to sign our certificate ourselves.

```
openssl req -new -key private-key.pem -out csr.pem
```

Now that we have our paper work filled out, it's time to pretend that we're a cool signing authority.

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

Now that you have the private key and the public cert, you can establish a secure connection between two NodeJS apps. And, as you can see in the example code, it is a very simple process.

Important!

Since we created the public cert ourselves, in all honesty, our certificate is worthless, because we are nobodies. The NodeJS server won't trust such a certificate by default, and that is why we need to tell it to actually trust our cert with the following option `rejectUnauthorized: false`. **Very important:** never set this variable to true in a production environment.

TLS Socket Server

```
'use strict';

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

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

  // Send a friendly message
  socket.write("I am the server sending you a message.");

  // Print the data that we received
  socket.on('data', function(data) {

    console.log('Received: %s [it is %d bytes long]',
      data.toString().replace(/\n/gm, ""),
      data.length);

  });

  // Let us know when the transmission is over
  socket.on('end', function() {

    console.log('EOT (End Of Transmission)');

  });

});

// Start listening on a specific port and address
server.listen(PORT, HOST, function() {

  console.log("I'm listening at %s, on port %s", HOST, PORT);

});

// When an error occurs, show it.
server.on('error', function(error) {

  console.error(error);

  // Close the connection after the error occurred.
```

```
server.destroy();

});
```

TLS Socket Client

```
'use strict';

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

const PORT = 1337;
const HOST = '127.0.0.1'

// Pass the certs to the server and let it know to process even unauthorized certs.
var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem'),
  rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

  // Check if the authorization worked
  if (client.authorized) {
    console.log("Connection authorized by a Certificate Authority.");
  } else {
    console.log("Connection not authorized: " + client.authorizationError)
  }

  // Send a friendly message
  client.write("I am the client sending you a message.");

});

client.on("data", function(data) {

  console.log('Received: %s [it is %d bytes long]',
    data.toString().replace(/\n/gm, ""),
    data.length);

  // Close the connection after receiving the message
  client.end();

});

client.on('close', function() {

  console.log("Connection closed");

});

// When an error occurs, show it.
client.on('error', function(error) {

  console.error(error);

  // Close the connection after the error occurred.
  client.destroy();

});
```


Section 1.11: How to get a basic HTTPS web server up and running!

Once you have node.js installed on your system, you can just follow the procedure below to get a basic web server running with support for both HTTP and HTTPS!

Step 1 : Build a Certificate Authority

1. create the folder where you want to store your key & certificate :

```
mkdir conf
```

2. go to that directory :

```
cd conf
```

3. grab this ca.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. create a new certificate authority using this configuration :

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. now that we have our certificate authority in ca-key.pem and ca-cert.pem, let's generate a private key for the server :

```
openssl genrsa -out key.pem 4096
```

6. grab this server.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generate the certificate signing request using this configuration :

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. sign the request :

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Step 2 : Install your certificate as a root certificate

1. copy your certificate to your root certificates' folder :

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. update CA store :

```
sudo update-ca-certificates
```

Step 3 : Starting your node server

First, you want to create a `server.js` file that contains your actual server code.

The minimal setup for an HTTPS server in Node.js would be something like this :

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

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

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

https.createServer(httpsOptions, app).listen(4433);
```

If you also want to support http requests, you need to make just this small modification :

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

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

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

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. go to the directory where your `server.js` is located :

```
cd /path/to
```

2. run `server.js` :

```
node server.js
```

Chapter 2: npm

Parameter	Example
access	npm publish --access=public
bin	npm bin -g
edit	npm edit connect
help	npm help init
init	npm init
install	npm install
link	npm link
prune	npm prune
publish	npm publish ./
restart	npm restart
start	npm start
stop	npm start
update	npm update
version	npm version

Node Package Manager (npm) provides following two main functionalities: Online repositories for node.js packages/modules which are searchable on search.nodejs.org. Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

Section 2.1: Installing packages

Introduction

Package is a term used by npm to denote tools that developers can use for their projects. This includes everything from libraries and frameworks such as jQuery and AngularJS to task runners such as Gulp.js. The packages will come in a folder typically called `node_modules`, which will also contain a `package.json` file. This file contains information regarding all the packages including any dependencies, which are additional modules needed to use a particular package.

Npm uses the command line to both install and manage packages, so users attempting to use npm should be familiar with basic commands on their operating system i.e.: traversing directories as well as being able to see the contents of directories.

Installing NPM

Note that in order to install packages, you must have NPM installed.

The recommended way to install NPM is to use one of the installers from the [Node.js download page](#). You can check to see if you already have node.js installed by running either the `npm -v` or the `npm version` command.

After installing NPM via the Node.js installer, be sure to check for updates. This is because NPM gets updated more frequently than the Node.js installer. To check for updates run the following command:

```
npm install npm@latest -g
```

How to install packages

To install one or more packages use the following:

```
npm install <package-name>
# or
npm i <package-name>...
```

```
# e.g. to install lodash and express
npm install lodash express
```

Note: This will install the package in the directory that the command line is currently in, thus it is important to check whether the appropriate directory has been chosen

If you already have a package. `json` file in your current working directory and dependencies are defined in it, then `npm install` will automatically resolve and install all dependencies listed in the file. You can also use the shorthand version of the `npm install` command which is: `npm i`

If you want to install a specific version of a package use:

```
npm install <name>@<version>

# e.g. to install version 4.11.1 of the package lodash
npm install lodash@4.11.1
```

If you want to install a version which matches a specific version range use:

```
npm install <name>@<version range>

# e.g. to install a version which matches "version >= 4.10.1" and "version < 4.11.1"
# of the package lodash
npm install lodash@">=4.10.1 <4.11.1"
```

If you want to install the latest version use:

```
npm install <name>@latest
```

The above commands will search for packages in the central npm repository at [npmjs.com](https://www.npmjs.com). If you are not looking to install from the npm registry, other options are supported, such as:

```
# packages distributed as a tarball
npm install <tarball file>
npm install <tarball url>

# packages available locally
npm install <local path>

# packages available as a git repository
npm install <git remote url>

# packages available on GitHub
npm install <username>/<repository>

# packages available as gist (need a package.json)
npm install gist:<gist-id>

# packages from a specific repository
npm install --registry=http://myreg.mycompany.com <package name>

# packages from a related group of packages
# See npm scope
npm install @<scope>/<name>(@<version>)

# Scoping is useful for separating private packages hosted on private registry from
# public ones by setting registry for specific scope
```

```
npm config set @mycompany:registry http://myreg.mycompany.com
npm install @mycompany/<package name>
```

Usually, modules will be installed locally in a folder named `node_modules`, which can be found in your current working directory. This is the directory `require()` will use to load modules in order to make them available to you.

If you already created a `package.json` file, you can use the `--save` (shorthand `-S`) option or one of its variants to automatically add the installed package to your `package.json` as a dependency. If someone else installs your package, npm will automatically read dependencies from the `package.json` file and install the listed versions. Note that you can still add and manage your dependencies by editing the file later, so it's usually a good idea to keep track of dependencies, for example using:

```
npm install --save <name> # Install dependencies
# or
npm install -S <name> # shortcut version --save
# or
npm i -S <name>
```

In order to install packages and save them only if they are needed for development, not for running them, not if they are needed for the application to run, follow the following command:

```
npm install --save-dev <name> # Install dependencies for development purposes
# or
npm install -D <name> # shortcut version --save-dev
# or
npm i -D <name>
```

Installing dependencies

Some modules do not only provide a library for you to use, but they also provide one or more binaries which are intended to be used via the command line. Although you can still install those packages locally, it is often preferred to install them globally so the command-line tools can be enabled. In that case, npm will automatically link the binaries to appropriate paths (e.g. `/usr/local/bin/<name>`) so they can be used from the command line. To install a package globally, use:

```
npm install --global <name>
# or
npm install -g <name>
# or
npm i -g <name>

# e.g. to install the grunt command line tool
npm install -g grunt-cli
```

If you want to see a list of all the installed packages and their associated versions in the current workspace, use:

```
npm list
npm list <name>
```

Adding an optional name argument can check the version of a specific package.

Note: If you run into permission issues while trying to install an npm module globally, resist the temptation to issue a `sudo npm install -g ...` to overcome the issue. Granting third-party scripts to run on your system with elevated privileges is dangerous. The permission issue might mean that you have an issue with the way npm itself was installed. If you're interested in installing Node in sandboxed user environments, you might want to try using [nvm](#).

If you have build tools, or other development-only dependencies (e.g. Grunt), you might not want to have them bundled with the application you deploy. If that's the case, you'll want to have it as a development dependency, which is listed in the package.json under devDependencies. To install a package as a development-only dependency, use `--save-dev` (or `-D`).

```
npm install --save-dev <name> // Install development dependencies which is not included in
production
# or
npm install -D <name>
```

You will see that the package is then added to the devDependencies of your package.json.

To install dependencies of a downloaded/cloned node.js project, you can simply use

```
npm install
# or
npm i
```

npm will automatically read the dependencies from package.json and install them.

NPM Behind A Proxy Server

If your internet access is through a proxy server, you might need to modify npm install commands that access remote repositories. npm uses a configuration file which can be updated via command line:

```
npm config set
```

You can locate your proxy settings from your browser's settings panel. Once you have obtained the proxy settings (server URL, port, username and password); you need to configure your npm configurations as follows.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

username, password, port fields are optional. Once you have set these, your npm install, npm i -g etc. would work properly.

Section 2.2: Uninstalling packages

To uninstall one or more locally installed packages, use:

```
npm uninstall <package name>
```

The uninstall command for npm has five aliases that can also be used:

```
npm remove <package name>
npm rm <package name>
npm r <package name>

npm unlink <package name>
npm un <package name>
```

If you would like to remove the package from the package.json file as part of the uninstallation, use the `--save` flag (shorthand: `-S`):

```
npm uninstall --save <package name>
```



```
npm uninstall -S <package name>
```

For a development dependency, use the `--save-dev` flag (shorthand: `-D`):

```
npm uninstall --save-dev <package name>
npm uninstall -D <package name>
```

For an optional dependency, use the `--save-optional` flag (shorthand: `-O`):

```
npm uninstall --save-optional <package name>
npm uninstall -O <package name>
```

For packages that are installed globally use the `--global` flag (shorthand: `-g`):

```
npm uninstall -g <package name>
```

Section 2.3: Setting up a package configuration

Node.js package configurations are contained in a file called `package.json` that you can find at the root of each project. You can setup a brand new configuration file by calling:

```
npm init
```

That will try to read the current working directory for Git repository information (if it exists) and environment variables to try and autocomplete some of the placeholder values for you. Otherwise, it will provide an input dialog for the basic options.

If you'd like to create a `package.json` with default values use:

```
npm init --yes
# or
npm init -y
```

If you're creating a `package.json` for a project that you are not going to be publishing as an npm package (i.e. solely for the purpose of rounding up your dependencies), you can convey this intent in your `package.json` file:

1. Optionally set the `private` property to `true` to prevent accidental publishing.
2. Optionally set the `license` property to `"UNLICENSED"` to deny others the right to use your package.

To install a package and automatically save it to your `package.json`, use:

```
npm install --save <package>
```

The package and associated metadata (such as the package version) will appear in your dependencies. If you save it as a development dependency (using `--save-dev`), the package will instead appear in your `devDependencies`.

With this bare-bones `package.json`, you will encounter warning messages when installing or upgrading packages, telling you that you are missing a description and the repository field. While it is safe to ignore these messages, you can get rid of them by opening the `package.json` in any text editor and adding the following lines to the JSON object:

```
[...]
"description": "No description",
"repository": {
  "private": true
},
```

[...]

Section 2.4: Running scripts

You may define scripts in your `package.json`, for example:

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

To run the echo script, run `npm run echo` from the command line. Arbitrary scripts, such as echo above, have to be run with `npm run <script name>`. npm also has a number of official scripts that it runs at certain stages of the package's life (like `preinstall`). See [here](#) for the entire overview of how npm handles script fields.

npm scripts are used most often for things like starting a server, building the project, and running tests. Here's a more realistic example:

```
"scripts": {
  "test": "mocha tests",
  "start": "pm2 start index.js"
}
```

In the `scripts` entries, command-line programs like `mocha` will work when installed either globally or locally. If the command-line entry does not exist in the system `PATH`, npm will also check your locally installed packages.

If your scripts become very long, they can be split into parts, like this:

```
"scripts": {
  "very-complex-command": "npm run chain-1 && npm run chain-2",
  "chain-1": "webpack",
  "chain-2": "node app.js"
}
```

Section 2.5: Basic semantic versioning

Before publishing a package you have to version it. npm supports [semantic versioning](#), this means there are **patch**, **minor** and **major** releases.

For example, if your package is at version 1.2.3 to change version you have to:

1. patch release: `npm version patch => 1.2.4`
2. minor release: `npm version minor => 1.3.0`
3. major release: `npm version major => 2.0.0`

You can also specify a version directly with:

```
npm version 3.1.4 => 3.1.4
```

When you set a package version using one of the npm commands above, npm will modify the version field of the package.json file, commit it, and also create a new Git tag with the version prefixed with a "v", as if you've issued the command:

```
git tag v3.1.4
```

Unlike other package managers like Bower, the npm registry doesn't rely on Git tags being created for every version. But, if you like using tags, you should remember to push the newly created tag after bumping the package version:

```
git push origin master (to push the change to package.json)
```

```
git push origin v3.1.4 (to push the new tag)
```

Or you can do this in one swoop with:

```
git push origin master --tags
```

Section 2.6: Publishing a package

First, make sure that you have configured your package (as said in [Setting up a package configuration?](#)). Then, you have to be logged in to npmjs.

If you already have a npm user

```
npm login
```

If you don't have a user

```
npm adduser
```

To check that your user is registered in the current client

```
npm config ls
```

After that, when your package is ready to be published use

```
npm publish
```

And you are done.

If you need to publish a new version, ensure that you update your package version, as stated in [Basic semantic versioning](#). Otherwise, npm will not let you publish the package.

```
{  
  name: "package-name",  
  version: "1.0.4"  
}
```

Section 2.7: Removing extraneous packages

To remove extraneous packages (packages that are installed but not in dependency list) run the following command:

```
npm prune
```

To remove all dev packages add `--production` flag:

```
npm prune --production
```

[More on it](#)

Section 2.8: Listing currently installed packages

To generate a list (tree view) of currently installed packages, use

```
npm list
```

ls, **la** and **ll** are aliases of **list** command. **la** and **ll** commands shows extended information like description and repository.

Options

The response format can be changed by passing options.

```
npm list --json
```

- **json** - Shows information in json format
- **long** - Shows extended information
- **parseable** - Shows parseable list instead of tree
- **global** - Shows globally installed packages
- **depth** - Maximum display depth of dependency tree
- **dev/development** - Shows devDependencies
- **prod/production** - Shows dependencies

If you want, you can also go to the package's home page.

```
npm home <package name>
```

Section 2.9: Updating npm and packages

Since npm itself is a Node.js module, it can be updated using itself.

If OS is Windows must be running command prompt as Admin

```
npm install -g npm@latest
```

If you want to check for updated versions you can do:

```
npm outdated
```

In order to update a specific package:

```
npm update <package name>
```

This will update the package to the latest version according to the restrictions in package.json

In case you also want to lock the updated version in package.json:

```
npm update <package name> --save
```

Section 2.10: Scopes and repositories

```
# Set the repository for the scope "myscope"
npm config set @myscope:registry http://registry.corporation.com

# Login at a repository and associate it with the scope "myscope"
npm adduser --registry=http://registry.corporation.com --scope=@myscope

# Install a package "mylib" from the scope "myscope"
npm install @myscope/mylib
```

If the name of your own package starts with `@myscope` and the scope "myscope" is associated with a different repository, `npm publish` will upload your package to that repository instead.

You can also persist these settings in a `.npmrc` file:

```
@myscope:registry=http://registry.corporation.com
//registry.corporation.com/:_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx
```

This is useful when automating the build on a CI server f.e.

Section 2.11: Linking projects for faster debugging and development

Building project dependencies can sometimes be a tedious task. Instead of publishing a package version to NPM and installing the dependency to test the changes, use `npm link`. `npm link` creates a symlink so the latest code can be tested in a local environment. This makes testing global tools and project dependencies easier by allowing the latest code run before making a published version.

Help text

```
NAME
  npm-link - Symlink a package folder

SYNOPSIS
  npm link (in package dir)
  npm link [<@scope>/]<pkg>[@<version>]

  alias: npm ln
```

Steps for linking project dependencies

When creating the dependency link, note that the package name is what is going to be referenced in the parent project.

1. CD into a dependency directory (ex: `cd ../my-dep`)
2. `npm link`
3. CD into the project that is going to use the dependency
4. `npm link my-dep` or if namespaced `npm link @namespace/my-dep`

Steps for linking a global tool

1. CD into the project directory (ex: `cd eslint-watch`)

2. `npm link`
3. Use the tool
4. `esw --quiet`

Problems that may arise

Linking projects can sometimes cause issues if the dependency or global tool is already installed. `npm uninstall (-g) <pkg>` and then running `npm link` normally resolves any issues that may arise.

Section 2.12: Locking modules to specific versions

By default, npm installs the latest available version of modules according to each dependencies' [semantic version](#). This can be problematic if a module author doesn't adhere to semver and introduces breaking changes in a module update, for example.

To lock down each dependencies' version (and the versions of their dependencies, etc) to the specific version installed locally in the `node_modules` folder, use

```
npm shrinkwrap
```

This will then create a `npm-shrinkwrap.json` alongside your `package.json` which lists the specific versions of dependencies.

Section 2.13: Setting up for globally installed packages

You can use `npm install -g` to install a package "globally." This is typically done to install an executable that you can add to your path to run. For example:

```
npm install -g gulp-cli
```

If you update your path, you can call `gulp` directly.

On many OSes, `npm install -g` will attempt to write to a directory that your user may not be able to write to such as `/usr/bin`. You should **not** use `sudo npm install` in this case since there is a possible security risk of running arbitrary scripts with `sudo` and the root user may create directories in your home that you cannot write to which makes future installations more difficult.

You can tell npm where to install global modules to via your configuration file, `~/.npmrc`. This is called the prefix which you can view with `npm prefix`.

```
prefix=~/.npm-global-modules
```

This will use the prefix whenever you run `npm install -g`. You can also use `npm install --prefix ~/.npm-global-modules` to set the prefix when you install. If the prefix is the same as your configuration, you don't need to use `-g`.

In order to use the globally installed module, it needs to be on your path:

```
export PATH=$PATH:~/.npm-global-modules/bin
```

Now when you run `npm install -g gulp-cli` you will be able to use `gulp`.

Note: When you `npm install` (without `-g`) the prefix will be the directory with `package.json` or the current directory if none is found in the hierarchy. This also creates a directory `node_modules/.bin` that has the

executables. If you want to use an executable that is specific to a project, it's not necessary to use `npm install -g`. You can use the one in `node_modules/.bin`.

Chapter 3: Web Apps With Express

Parameter	Details
path	Specifies the path portion or the URL that the given callback will handle.
middleware	One or more functions which will be called before the callback. Essentially a chaining of multiple callback functions. Useful for more specific handling for example authorization or error handling.
callback	A function that will be used to handle requests to the specified path. It will be called like <code>callback(request, response, next)</code> , where request, response, and next are described below.
<i>callback</i> request	An object encapsulating details about the HTTP request that the callback is being called to handle.
response	An object that is used to specify how the server should respond to the request.
next	A callback that passes control on to the next matching route. It accepts an optional error object.

Express is a minimal and flexible Node.js web application framework, providing a robust set of features for building web applications.

The official website of Express is expressjs.com. The source can be found [on GitHub](#).

Section 3.1: Getting Started

You will first need to create a directory, access it in your shell and install Express using [npm](#) by running `npm install express --save`

Create a file and name it `app.js` and add the following code which creates a new Express server and adds one endpoint to it (`/ping`) with the `app.get` method:

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

const app = express();

app.get('/ping', (request, response) => {
  response.send('pong');
});

app.listen(8080, 'localhost');
```

To run your script use the following command in your shell:

```
> node app.js
```

Your application will accept connections on localhost port 8080. If the hostname argument to `app.listen` is omitted, then server will accept connections on the machine's IP address as well as localhost. If port value is 0, the operating system will assign an available port.

Once your script is running, you can test it in a shell to confirm that you get the expected response, "pong", from the server:

```
> curl http://localhost:8080/ping
pong
```

You can also open a web browser, navigate to the url <http://localhost:8080/ping> to view the output

Section 3.2: Basic routing

First create an express app:


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

Then you can define routes like this:

```
app.get('/someUri', function (req, res, next) {})
```

That structure works for all HTTP methods, and expects a path as the first argument, and a handler for that path, which receives the request and response objects. So, for the basic HTTP methods, these are the routes

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

You can check the complete list of supported verbs [here](#). If you want to define the same behavior for a route and all HTTP methods, you can use:

```
app.all('/myPath', function (req, res, next) {})
```

or

```
app.use('/myPath', function (req, res, next) {})
```

or

```
app.use('*', function (req, res, next) {})

// * wildcard will route for all paths
```

You can chain your route definitions for a single path

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

You can also add functions to any HTTP method. They will run before the final callback and take the parameters (req, res, next) as arguments.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Your final callbacks can be stored in an external file to avoid putting too much code in one file:

```
// other.js
exports.doSomething = function(req, res, next) { /* do some stuff */};
```

And then in the file containing your routes:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

This will make your code much cleaner.

Section 3.3: Modular express application

To make express web application modular use router factories:

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();

  router.get('/greet', (req, res, next) => {
    res.end(options.greeting);
  });

  return router;
};
```

Application:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
  .use('/api/v1/', greetMiddleware({ greeting: 'Hello world' }))
  .listen(8080);
```

This will make your application modular, customisable and your code reusable.

When accessing `http://<hostname>:8080/api/v1/greet` the output will be Hello world

More complicated example

Example with services that shows middleware factory advantages.

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();
  // Get controller
  const {service} = options;

  router.get('/greet', (req, res, next) => {
    res.end(
      service.createGreeting(req.query.name || 'Stranger')
    );
  });
};
```

```
    return router;
};
```

Application:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
  constructor(greeting = 'Hello') {
    this.greeting = greeting;
  }

  createGreeting(name) {
    return `${this.greeting}, ${name}!`;
  }
}

express()
  .use('/api/v1/service1', greetMiddleware({
    service: new GreetingService('Hello'),
  }))
  .use('/api/v1/service2', greetMiddleware({
    service: new GreetingService('Hi'),
  }))
  .listen(8080);
```

When accessing `http://<hostname>:8080/api/v1/service1/greet?name=World` the output will be `Hello, World` and accessing `http://<hostname>:8080/api/v1/service2/greet?name=World` the output will be `Hi, World`.

Section 3.4: Using a Template Engine

Using a Template Engine

The following code will setup Jade as template engine. (Note: Jade has been renamed to pug as of December 2015.)

```
const express = require('express'); //Imports the express module
const app = express(); //Creates an instance of the express module

const PORT = 3000; //Randomly chosen port

app.set('view engine', 'jade'); //Sets jade as the View Engine / Template Engine
app.set('views', 'src/views'); //Sets the directory where all the views (.jade files) are stored.

//Creates a Root Route
app.get('/', function(req, res){
  res.render('index'); //renders the index.jade file into html and returns as a response. The
  render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
app.listen(PORT, function(err) {
  if (!err) {
    console.log('Server is running at port', PORT);
  } else {
    console.log(JSON.stringify(err));
  }
});
```

Similarly, other Template Engines could be used too such as Handlebars(hbs) or ejs. Remember to npm **install** the Template Engine too. For Handlebars we use hbs package, for Jade we have a jade package and for EJS, we have an ejs package.

EJS Template Example

With EJS (like other express templates), you can run server code and access your server variables from you HTML. In EJS it's done using "<%" as start tag and "%>" as end tag, variables passed as the render params can be accessed using `<%=var_name%>`

For instance, if you have supplies array in your server code you can loop over it using

```
<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) { %>
  <li>
    <a href='supplies/<%= supplies[i] %>'>
      <%= supplies[i] %>
    </a>
  </li>
<% } %>
```

As you can see in the example every time you switch between server side code and HTML you need to close the current EJS tag and open a new one later, here we wanted to create li inside the **for** command so we needed to close our EJS tag at the end of the **for** and create new tag just for the curly brackets

another example

if we want to put input default version to be a variable from the server side we use `<%=` for example:

```
Message:<br>
<input type="text" value="<%= message %>" name="message" required>
```

Here the message variable passed from your server side will be the default value of your input, please be noticed that if you didn't pass message variable from your server side, EJS will throw an exception. You can pass parameters using `res.render('index', {message: message});` (for ejs file called index.ejs).

In the EJS tags you can also use **if**, while or any other javascript command you want.

Section 3.5: JSON API with ExpressJS

```
var express = require('express');
var cors = require('cors'); // Use cors module for enable Cross-origin resource sharing

var app = express();
app.use(cors()); // for all routes

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
  var info = {
    'string_value': 'StackOverflow',
    'number_value': 8476
  }
  res.json(info);

  // or
```

```

    /* res.send(JSON.stringify({
      string_value: 'StackOverflow',
      number_value: 8476
    }))) */

    //you can add a status code to the json response
    /* res.status(200).json(info) */
  })

  app.listen(port, function() {
    console.log('Node.js listening on port ' + port)
  })

```

On `http://localhost:8080/` output object

```

{
  string_value: "StackOverflow",
  number_value: 8476
}

```

Section 3.6: Serving static files

When building a webserver with Express it's often required to serve a combination of dynamic content and static files.

For example, you may have `index.html` and `script.js` which are static files kept in the file system.

It is common to use folder named 'public' to have static files. In this case the folder structure may look like:

```

project root
??? server.js
??? package.json
??? public
    ??? index.html
    ??? script.js

```

This is how to configure Express to serve static files:

```

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

app.use(express.static('public'));

```

Note: once the folder is configured, `index.html`, `script.js` and all the files in the "public" folder will be available in at the root path (you must not specify `/public/` in the url). This is because, express looks up for the files relative to the static folder configured. You can specify *virtual path prefix* as shown below:

```

app.use('/static', express.static('public'));

```

will make the resources available under the `/static/` prefix.

Multiple folders

It is possible to define multiple folders at the same time:

```

app.use(express.static('public'));
app.use(express.static('images'));

```

```
app.use(express.static('files'));
```

When serving the resources Express will examine the folder in definition order. In case of files with the same name, the one in the first matching folder will be served.

Section 3.7: Adding Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.

Middleware functions can execute any code, make changes to res and req objects, end response cycle and call next middleware.

Very common example of middleware is cors module. To add CORS support, simply install it, require it and put this line:

```
app.use(cors());
```

before any routers or routing functions.

Section 3.8: Error Handling

Basic Error Handling

By default, Express will look for an 'error' view in the `/views` directory to render. Simply create the 'error' view and place it in the views directory to handle errors. Errors are written with the error message, status and stack trace, for example:

views/error.pug

```
html
  body
    h1= message
    h2= error.status
    p= error.stack
```

Advanced Error Handling

Define your error-handling middleware functions at the very end of the middleware function stack. These have four arguments instead of three (`err, req, res, next`) for example:

app.js

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;

  //pass error to the next matching route.
  next(err);
});

// handle error, print stacktrace
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
```

```
res.render('error', {
  message: err.message,
  error: err
});
});
```

You can define several error-handling middleware functions, just as you would with regular middleware functions.

Section 3.9: Getting info from the request

To get info from the requesting url (notice that req is the request object in the handler function of routes). Consider this route definition `/settings/:user_id` and this particular example `/settings/32135?field=name`

```
// get the full path
req.originalUrl // => /settings/32135?field=name

// get the user_id param
req.params.user_id // => 32135

// get the query value of the field
req.query.field // => 'name'
```

You can also get headers of the request, like this

```
req.get('Content-Type')
// "text/plain"
```

To simplify getting other info you can use middlewares. For example, to get the body info of the request, you can use the [body-parser](#) middleware, which will transform raw request body into usable format.

```
var app = require('express')();
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded
```

Now suppose a request like this

```
PUT /settings/32135
{
  "name": "Peter"
}
```

You can access the posted name like this

```
req.body.name
// "Peter"
```

In a similar way, you can access cookies from the request, you also need a middleware like [cookie-parser](#)

```
req.cookies.name
```

Section 3.10: Error handling in Express

In Express, you can define unified error handler for handling errors occurred in application. Define then handler at the end of all routes and logic code.

Example

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

//GET /names/john
app.get('/names/:name', function(req, res, next){
  if (req.params.name == 'john'){
    return res.send('Valid Name');
  } else{
    next(new Error('Not valid name'));    //pass to error handler
  }
});

//error handler
app.use(function(err, req, res, next){
  console.log(err.stack);    // e.g., Not valid name
  return res.status(500).send('Internal Server Occured');
});

app.listen(3000);
```

Section 3.11: Hook: How to execute code before any req and after any res

`app.use()` and middleware can be used for "before" and a combination of the [close](#) and [finish](#) events can be used for "after".

```
app.use(function (req, res, next) {
  function afterResponse() {
    res.removeListener('finish', afterResponse);
    res.removeListener('close', afterResponse);

    // actions after response
  }
  res.on('finish', afterResponse);
  res.on('close', afterResponse);

  // action before request
  // eventually calling `next()`
  next();
});
...
app.use(app.router);
```

An example of this is the [logger](#) middleware, which will append to the log after the response by default.

Just make sure this "middleware" is used before `app.router` as order does matter.

Original post is [here](#)

Section 3.12: Setting cookies with cookie-parser

The following is an example for setting and reading cookies using the [cookie-parser](#) module:

```
var express = require('express');
var cookieParser = require('cookie-parser'); // module for parsing cookies
var app = express();
```



```

app.use(cookieParser());

app.get('/setcookie', function(req, res){
  // setting cookies
  res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
  return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
  var username = req.cookies['username'];
  if (username) {
    return res.send(username);
  }

  return res.send('No cookie found');
});

app.listen(3000);

```

Section 3.13: Custom middleware in Express

In Express, you can define middlewares that can be used for checking requests or setting some headers in response.

```

app.use(function(req, res, next){ }); // signature

```

Example

The following code adds user to the request object and pass the control to the next matching route.

```

var express = require('express');
var app = express();

//each request will pass through it
app.use(function(req, res, next){
  req.user = 'testuser';
  next(); // it will pass the control to next matching route
});

app.get('/', function(req, res){
  var user = req.user;
  console.log(user); // testuser
  return res.send(user);
});

app.listen(3000);

```

Section 3.14: Named routes in Django-style

One big problem is that valuable named routes is not supported by Express out of the box. Solution is to install supported third-party package, for example [express-reverse](#):

```

npm install express-reverse

```

Plug it in your project:

```

var app = require('express')();

```

```
require('express-reverse')(app);
```

Then use it like:

```
app.get('test', '/hello', function(req, res) {  
  res.end('hello');  
});
```

The downside of this approach is that you can't use the route Express module as shown in [Advanced router usage](#). The workaround is to pass your app as a parameter to your router factory:

```
require('./middlewares/routing')(app);
```

And use it like:

```
module.exports = (app) => {  
  app.get('test', '/hello', function(req, res) {  
    res.end('hello');  
  });  
};
```

You can figure it out from now on, how to define functions to merge it with specified custom namespaces and point at appropriate controllers.

Section 3.15: Hello World

Here we create a basic hello world server using Express. Routes:

- '/'
- '/wiki'

And for rest will give "404", i.e. page not found.

```
'use strict';  
  
const port = process.env.PORT || 3000;  
  
var app = require('express')();  
app.listen(port);  
  
app.get('/', (req, res) => res.send('HelloWorld!'));  
app.get('/wiki', (req, res) => res.send('This is wiki page.'));  
app.use((req, res) => res.send('404-PageNotFound'));
```

Note: We have put 404 route as the last route as Express stacks routes in order and processes them for each request sequentially.

Section 3.16: Using middleware and the next callback

Express passes a `next` callback to every route handler and middleware function that can be used to break logic for single routes across multiple handlers. Calling `next()` with no arguments tells express to continue to the next matching middleware or route handler. Calling `next(err)` with an error will trigger any error handler middleware. Calling `next('route')` will bypass any subsequent middleware on the current route and jump to the next matching route. This allows domain logic to be decoupled into reusable components that are self-contained, simpler to test, and easier to maintain and change.

Multiple matching routes

Requests to `/api/foo` or to `/api/bar` will run the initial handler to look up the member and then pass control to the actual handler for each route.

```
app.get('/api', function(req, res, next) {
  // Both /api/foo and /api/bar will run this
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // Only /api/foo will run this
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // Only /api/bar will run this
  doSomethingDifferentWithMember(req.member);
});
```

Error handler

Error handlers are middleware with the signature `function(err, req, res, next)`. They could be set up per route (e.g. `app.get('/foo', function(err, req, res, next))`) but typically, a single error handler that renders an error page is sufficient.

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
    if (err) return next(err);
    renderPage(data);
  });
});

// In the case that doSomethingAsync return an error, this special
// error handler middleware will be called with the error as the
// first parameter.
app.use(function(err, req, res, next) {
  renderErrorPage(err);
});
```

Middleware

Each of the functions above is actually a middleware function that is run whenever a request matches the route defined, but any number of middleware functions can be defined on a single route. This allows middleware to be defined in separate files and common logic to be reused across multiple routes.

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
    if (err) return next(err);
    // If there's no member, don't try to look
    // up data. Just go render the page now.
    if (!member) return next('route');
    // Otherwise, call the next middleware and fetch
    // the member's data.
    req.member = member;
  });
});
```

```

    next();
  });
}, function(req, res, next) {
  getMemberData(req.member, function(err, data) {
    if (err) return next(err);
    // If this member has no data, don't bother
    // parsing it. Just go render the page now.
    if (!data) return next('route');
    // Otherwise, call the next middleware and parse
    // the member's data. THEN render the page.
    req.member.data = data;
    next();
  });
}, function(req, res, next) {
  req.member.parsedData = parseMemberData(req.member.data);
  next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});

```

In this example, each middleware function would be either in it's own file or in a variable elsewhere in the file so that it could be reused in other routes.

Section 3.17: Error handling

Basic docs can be found [here](#)

```

app.get('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed
  if (req.params.id == 0) // validate param
    return next(new Error('Id is 0')); // go to first Error handler, see below

  // Catch error on sync operation
  var data;
  try {
    data = JSON.parse('/file.json');
  } catch (err) {
    return next(err);
  }

  // If some critical error then stop application
  if (!data)
    throw new Error('Smth wrong');

  // If you need send extra info to Error handler
  // then send custom error (see Appendix B)
  if (smth)
    next(new MyError('smth wrong', arg1, arg2))

  // Finish request by res.render or res.end
  res.status(200).end('OK');
});

// Be sure: order of app.use have matter
// Error handler
app.use(function(err, req, res, next) {
  if (smth-check, e.g. req.url != 'POST')
    return next(err); // go-to Error handler 2.

```

```

    console.log(req.url, err.message);

    if (req.xhr) // if req via ajax then send json else render error-page
      res.json(err);
    else
      res.render('error.html', {error: err.message});
  });

  // Error handler 2
  app.use(function(err, req, res, next) {
    // do smth here e.g. check that error is MyError
    if (err instanceof MyError) {
      console.log(err.message, err.arg1, err.arg2);
    }
    ...
    res.end();
  });

```

Appendix A

```

// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
app.use(function(req, res, next) {
  next(new Error(404));
});

```

Appendix B

```

// How to define custom error
var util = require('util');
...
function MyError(message, arg1, arg2) {
  this.message = message;
  this.arg1 = arg1;
  this.arg2 = arg2;
  Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';

```

Section 3.18: Handling POST Requests

Just like you handle get requests in Express with `app.get` method, you can use `app.post` method to handle post requests.

But before you can handle POST requests, you will need to use the `body-parser` middleware. It simply parses the body of POST, PUT, DELETE and other requests.

`Body-Parser` middleware parses the body of the request and turns it into an object available in `req.body`

```

var bodyParser = require('body-parser');

const express = require('express');

const app = express();

// Parses the body for POST, PUT, DELETE, etc.
app.use(bodyParser.json());

```

```
app.use(bodyParser.urlencoded({ extended: true }));  
app.post('/post-data-here', function(req, res, next){  
    console.log(req.body); // req.body contains the parsed body of the request.  
});  
app.listen(8080, 'localhost');
```

Chapter 4: Filesystem I/O

Section 4.1: Asynchronously Read from Files

Use the filesystem module for all file operations:

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

With Encoding

In this example, read `hello.txt` from the directory `/tmp`. This operation will be completed in the background and the callback occurs on completion or failure:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {  
  // If an error occurred, output it and return  
  if(err) return console.error(err);  
  
  // No error occurred, content is a string  
  console.log(content);  
});
```

Without Encoding

Read the binary file `binary.txt` from the current directory, asynchronously in the background. Note that we do not set the 'encoding' option - this prevents Node.js from decoding the contents into a string:

```
fs.readFile('binary', (err, binaryContent) => {  
  // If an error occurred, output it and return  
  if(err) return console.error(err);  
  
  // No error occurred, content is a Buffer, output it in  
  // hexadecimal representation.  
  console.log(content.toString('hex'));  
});
```

Relative paths

Keep in mind that, in general case, your script could be run with an arbitrary current working directory. To address a file relative to the current script, use `__dirname` or `__filename`:

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {  
  //Rest of Function  
}
```

Section 4.2: Listing Directory Contents with `readdir` or `readdirSync`

```
const fs = require('fs');  
  
// Read the contents of the directory /usr/local/bin asynchronously.  
// The callback will be invoked once the operation has either completed  
// or failed.  
fs.readdir('/usr/local/bin', (err, files) => {  
  // On error, show it and return  
  if(err) return console.error(err);  
  
  // files is an array containing the names of all entries  
  // in the directory, excluding '.' (the directory itself)
```

```
// and '..' (the parent directory).

// Display directory entries
console.log(files.join(' '));
});
```

A synchronous variant is available as `readdirSync` which blocks the main thread and therefore prevents execution of asynchronous code at the same time. Most developers avoid synchronous IO functions in order to improve performance.

```
let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Using a generator

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

// Iterate through all items obtained via
// 'yield' statements
// A callback is passed to the generator function because it is required by
// the 'readdir' method
function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }

    return iter.next(data);
  });

  iter.next();
}

const dirPath = '/usr/local/bin';

// Execute the generator function
run(function* (resume) {
  // Emit the list of files in the directory from the generator
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});
```

Section 4.3: Copying files by piping streams

This program copies a file using readable and a writable stream with the `pipe()` function provided by the stream class

```
// require the file system module
var fs = require('fs');

/*
  Create readable stream to file in current directory named 'node.txt'
  Use utf8 encoding
  Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });
```



```
// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// use pipe to copy readable to writable
readable.pipe(writable);
```

Section 4.4: Reading from a file synchronously

For any file operations, you will need the filesystem module:

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

Reading a String

`fs.readFileSync` behaves similarly to `fs.readFile`, but does not take a callback as it completes synchronously and therefore blocks the main thread. Most node.js developers prefer the asynchronous variants which will cause virtually no delay in the program execution.

If an encoding option is specified, a string will be returned, otherwise a Buffer will be returned.

```
// Read a string from another file synchronously
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Section 4.5: Check Permissions of a File or Directory

`fs.access()` determines whether a path exists and what permissions a user has to the file or directory at that path. `fs.access` doesn't return a result rather, if it doesn't return an error, the path exists and the user has the desired permissions.

The permission modes are available as a property on the fs object, `fs.constants`

- `fs.constants.F_OK` - Has read/write/execute permissions (If no mode is provided, this is the default)
- `fs.constants.R_OK` - Has read permissions
- `fs.constants.W_OK` - Has write permissions
- `fs.constants.X_OK` - Has execute permissions (Works the same as `fs.constants.F_OK` on Windows)

Asynchronously

```
var fs = require('fs');
var path = '/path/to/check';

// checks execute permission
fs.access(path, fs.constants.X_OK, (err) => {
  if (err) {
    console.log('%s doesn't exist', path);
  } else {
    console.log('can execute %s', path);
  }
});

// Check if we have read/write permissions
// When specifying multiple permission modes
// each mode is separated by a pipe : '|'
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
```

```

    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can read/write %s', path);
    }
});

```

Synchronously

`fs.access` also has a synchronous version `fs.accessSync`. When using `fs.accessSync` you must enclose it within a try/catch block.

```

// Check write permission
try {
    fs.accessSync(path, fs.constants.W_OK);
    console.log('can write %s', path);
}
catch (err) {
    console.log("%s doesn't exist", path);
}

```

Section 4.6: Checking if a file or a directory exists

Asynchronously

```

var fs = require('fs');

fs.stat('path/to/file', function(err) {
    if (!err) {
        console.log('file or directory exists');
    }
    else if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
});

```

Synchronously

here, we must wrap the function call in a **try/catch** block to handle error.

```

var fs = require('fs');

try {
    fs.statSync('path/to/file');
    console.log('file or directory exists');
}
catch (err) {
    if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
}

```

Section 4.7: Determining the line count of a text file

app.js

```

const readline = require('readline');
const fs = require('fs');

var file = 'path.to.file';
var linesCount = 0;

```

```
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});
rl.on('line', function (line) {
  linesCount++; // on each linebreak, add +1 to 'linesCount'
});
rl.on('close', function () {
  console.log(linesCount); // print the result when the 'close' event is called
});
```

Usage:

```
node app
```

Section 4.8: Reading a file line by line

app.js

```
const readline = require('readline');
const fs = require('fs');

var file = 'path.to.file';
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});

rl.on('line', function (line) {
  console.log(line) // print the content of the line on each linebreak
});
```

Usage:

```
node app
```

Section 4.9: Avoiding race conditions when creating or using an existing directory

Due to Node's asynchronous nature, creating or using a directory by first:

1. checking for its existence with `fs.stat()`, then
2. creating or using it depending of the results of the existence check,

can lead to a [race condition](#) if the folder is created between the time of the check and the time of the creation. The method below wraps `fs.mkdir()` and `fs.mkdirSync()` in error-catching wrappers that let the exception pass if its code is `EEXIST` (already exists). If the error is something else, like `EPERM` (permission denied), throw or pass an error like the native functions do.

Asynchronous version with `fs.mkdir()`

```
var fs = require('fs'); function mkdir (dirPath, callback) { fs.mkdir(dirPath, (err) => { callback(err && err.code !== 'EEXIST' ? err : null); }); } mkdir('./existingDir', (err) => { if (err) return console.error(err.code); // Do something with
```

```
`./existingDir` here }));
```

Synchronous version with `fs.mkdirSync()`

```
function mkdirSync (dirPath) { try { fs.mkdirSync(dirPath); } catch(e) { if ( e.code !== 'EEXIST' ) throw e; } }  
mkdirSync('./existing-dir'); // Do something with `./existing-dir` now
```

Section 4.10: Cloning a file using streams

This program illustrates how one can copy a file using readable and writable streams using the `createReadStream()`, and `createWriteStream()` functions provided by the file system module.

```
//Require the file System module  
var fs = require('fs');  
  
/*  
  Create readable stream to file in current directory (__dirname) named 'node.txt'  
  Use utf8 encoding  
  Read the data in 16-kilobyte chunks  
*/  
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 *  
1024 });  
  
// create writable stream  
var writable = fs.createWriteStream(__dirname + '/nodeCopy.txt');  
  
// Write each chunk of data to the writable stream  
readable.on('data', function(chunk) {  
  writable.write(chunk);  
});
```

Section 4.11: Writing to a file using `writeFile` or `writeFileSync`

```
var fs = require('fs');  
  
// Save the string "Hello world!" in a file called "hello.txt" in  
// the directory "/tmp" using the default encoding (utf8).  
// This operation will be completed in background and the callback  
// will be called when it is either done or failed.  
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {  
  // If an error occurred, show it and return  
  if(err) return console.error(err);  
  // Successfully wrote to the file!  
});  
  
// Save binary data to a file called "binary.txt" in the current  
// directory. Again, the operation will be completed in background.  
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);  
fs.writeFile('binary.txt', buffer, function(err) {  
  // If an error occurred, show it and return  
  if(err) return console.error(err);  
  // Successfully wrote binary contents to the file!  
});
```

`fs.writeFileSync` behaves similarly to `fs.writeFile`, but does not take a callback as it completes synchronously and therefore blocks the main thread. Most node.js developers prefer the asynchronous variants which will cause virtually no delay in the program execution.

Note: Blocking the main thread is bad practice in node.js. Synchronous function should only be used when debugging or when no other options are availables.

```
// Write a string to another file and set the file mode to 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Section 4.12: Changing contents of a text file

Example. It will be replacing the word email to a name in a text file `index.txt` with simple RegExp `replace(/email/gim, 'name')`

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

fs.readFile('index.txt', 'utf-8', function(err, data) {
  if (err) throw err;

  var newValue = data.replace(/email/gim, 'name');

  fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
    if (err) throw err;
    console.log('Done!');
  })
})
```

Section 4.13: Deleting a file using unlink or unlinkSync

Delete a file asynchronously:

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

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('file deleted');
});
```

You can also delete it synchronously*:

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

fs.unlinkSync('/path/to/file.txt');
console.log('file deleted');
```

* avoid synchronous methods because they block the entire process until the execution finishes.

Section 4.14: Reading a file into a Buffer using streams

While reading content from a file is already asynchronous using the `fs.readFile()` method, sometimes we want to get the data in a Stream versus in a simple callback. This allows us to pipe this data to other locations or to process it as it comes in versus all at once at the end.

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

```

// Store file data chunks in this array
let chunks = [];
// We can use this variable to store the final data
let fileBuffer;

// Read file into stream.Readable
let fileStream = fs.createReadStream('text.txt');

// An error occurred with the stream
fileStream.once('error', (err) => {
  // Be sure to handle this properly!
  console.error(err);
});

// File is done being read
fileStream.once('end', () => {
  // create the final data Buffer from data chunks;
  fileBuffer = Buffer.concat(chunks);

  // Of course, you can do anything else you need to here, like emit an event!
});

// Data is flushed from fileStream in chunks,
// this callback will be executed for each chunk
fileStream.on('data', (chunk) => {
  chunks.push(chunk); // push data chunk to array

  // We can perform actions on the partial data we have so far!
});

```

Chapter 5: Exporting and Consuming Modules

Section 5.1: Creating a hello-world.js module

Node provides the `module.exports` interface to expose functions and variables to other files. The most simple way to do so is to export only one object (function or variable), as shown in the first example.

hello-world.js

```
module.exports = function(subject) {  
  console.log('Hello ' + subject);  
};
```

If we don't want the entire export to be a single object, we can export functions and variables as properties of the `exports` object. The three following examples all demonstrate this in slightly different ways :

- `hello-venus.js` : the function definition is done separately then added as a property of `module.exports`
- `hello-jupiter.js` : the functions definitions are directly put as the value of properties of `module.exports`
- `hello-mars.js` : the function definition is directly declared as a property of `exports` which is a short version of `module.exports`

hello-venus.js

```
function hello(subject) {  
  console.log('Venus says Hello ' + subject);  
}  
  
module.exports = {  
  hello: hello  
};
```

hello-jupiter.js

```
module.exports = {  
  hello: function(subject) {  
    console.log('Jupiter says hello ' + subject);  
  },  
  
  bye: function(subject) {  
    console.log('Jupiter says goodbye ' + subject);  
  }  
};
```

hello-mars.js

```
exports.hello = function(subject) {  
  console.log('Mars says Hello ' + subject);  
};
```

Loading module with directory name

We have a directory named `hello` which includes the following files:

index.js

```
// hello/index.js
module.exports = function(){
  console.log('Hej');
};
```

main.js

```
// hello/main.js
// We can include the other files we've defined by using the `require()` method
var hw = require('./hello-world.js'),
    hm = require('./hello-mars.js'),
    hv = require('./hello-venus.js'),
    hj = require('./hello-jupiter.js'),
    hu = require('./index.js');

// Because we assigned our function to the entire `module.exports` object, we
// can use it directly
hw('World!'); // outputs "Hello World!"

// In this case, we assigned our function to the `hello` property of exports, so we must
// use that here too
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"

// The result of assigning module.exports at once is the same as in hello-world.js
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"

hj.hello('Universe!'); // outputs "Jupiter says hello Universe!"
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"

hu(); //output 'hej'
```

Section 5.2: Loading and using a module

A module can be "imported", or otherwise "required" by the `require()` function. For example, to load the `http` module that ships with Node.js, the following can be used:

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

Aside from modules that are shipped with the runtime, you can also require modules that you have installed from npm, such as `express`. If you had already installed `express` on your system via npm **install** `express`, you could simply write:

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

You can also include modules that you have written yourself as part of your application. In this case, to include a file named `lib.js` in the same directory as current file:

```
const mylib = require('./lib');
```

Note that you can omit the extension, and `.js` will be assumed. Once you load a module, the variable is populated with an object that contains the methods and properties published from the required file. A full example:

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

// The `http` module has the property `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'
```



```
// Also contains `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>Module Test</body></html>');
  res.end();
}).listen(80);
```

Section 5.3: Folder as a module

Modules can be split across many .js files in the same folder. An example in a *my_module* folder:

function_one.js

```
module.exports = function() {
  return 1;
}
```

function_two.js

```
module.exports = function() {
  return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

A module like this one is used by referring to it by the folder name:

```
var split_module = require('./my_module');
```

Please note that if you required it by omitting *./* or any indication of a path to a folder from the require function argument, Node will try to load a module from the *node_modules* folder.

Alternatively you can create in the same folder a package.json file with these contents:

```
{
  "name": "my_module",
  "main": "./your_main_entry_point.js"
}
```

This way you are not required to name the main module file "index".

Section 5.4: Every module injected only once

NodeJS executes the module only the first time you require it. Any further require functions will re-use the same Object, thus not executing the code in the module another time. Also Node caches the modules first time they are loaded using require. This reduces the number of file reads and helps to speed up the application.

myModule.js

```
console.log(123) ;
exports.var1 = 4 ;
```

```

var a=require('./myModule') ; // Output 123
var b=require('./myModule') ; // No output
console.log(a.var1) ; // Output 4
console.log(b.var1) ; // Output 4
a.var2 = 5 ;
console.log(b.var2) ; // Output 5

```

Section 5.5: Module loading from node_modules

Modules can be required without using relative paths by putting them in a special directory called `node_modules`.

For example, to require a module called `foo` from a file `index.js`, you can use the following directory structure:

```

index.js
├─ node_modules
├─ foo
│  └─ foo.js
└─ package.json

```

Modules should be placed inside a directory, along with a `package.json` file. The `main` field of the `package.json` file should point to the entry point for your module--this is the file that is imported when users do `require('your-module')`. `main` defaults to `index.js` if not provided. Alternatively, you can refer to files relative to your module simply by appending the relative path to the `require` call: `require('your-module/path/to/file')`.

Modules can also be required from `node_modules` directories up the file system hierarchy. If we have the following directory structure:

```

my-project
├─ node_modules
│  └─ foo    // the foo module
├─ ...
├─ baz    // the baz module
│  └─ node_modules
│     └─ bar    // the bar module

```

we will be able to require the module `foo` from any file within `bar` using `require('foo')`.

Note that node will only match the module that is closest to the file in the filesystem hierarchy, starting from (the file's current directory/`node_modules`). Node matches directories this way up to the file system root.

You can either install new modules from the npm registry or other npm registries, or make your own.

Section 5.6: Building your own modules

You can also reference an object to publicly export and continuously append methods to that object:

```

const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {

```

```

    // Authenticate with a Facebook account
  }

  auth.twitter = function (data, callback) {
    // Authenticate with a Twitter account
  }

  auth.slack = function (data, callback) {
    // Authenticate with a Slack account
  }

  auth.stack_overflow = function (data, callback) {
    // Authenticate with a Stack Overflow account
  }

```

To use any of these, just require the module as you normally would:

```

const auth = require('./auth')

module.exports = function (req, res, next) {
  auth.facebook(req.body, function (err, user) {
    if (err) return next(err)

    req.user = user
    next()
  })
}

```

Section 5.7: Invalidating the module cache

In development, you may find that using `require()` on the same module multiple times always returns the same module, even if you have made changes to that file. This is because modules are cached the first time they are loaded, and any subsequent module loads will load from the cache.

To get around this issue, you will have to **delete** the entry in the cache. For example, if you loaded a module:

```
var a = require('./a');
```

You could then delete the cache entry:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

And then require the module again:

```
var a = require('./a');
```

Do note that this is not recommended in production because the **delete** will only delete the reference to the loaded module, not the loaded data itself. The module is not garbage collected, so improper use of this feature could lead to leaking memory.

Chapter 6: Exporting and Importing Module in node.js

Section 6.1: Exporting with ES6 syntax

This is the equivalent of [the other example](#) but using ES6 instead.

```
export function printHelloWorld() {  
  console.log("Hello World!!!");  
}
```

Section 6.2: Using a simple module in node.js

What is a node.js module ([link to article](#)):

A module encapsulates related code into a single unit of code. When creating a module, this can be interpreted as moving all related functions into a file.

Now lets see an example. Imagine all files are in same directory:

File: printer.js

```
"use strict";  
  
exports.printHelloWorld = function () {  
  console.log("Hello World!!!");  
}
```

Another way of using modules:

File animals.js

```
"use strict";  
  
module.exports = {  
  lion: function() {  
    console.log("ROAARR!!!");  
  }  
};
```

File: app.js

Run this file by going to your directory and typing: node app.js

```
"use strict";  
  
//require('./path/to/module.js') node which module to load  
var printer = require('./printer');  
var animals = require('./animals');  
  
printer.printHelloWorld(); //prints "Hello World!!!"  
animals.lion(); //prints "ROAARR!!!"
```

Chapter 7: Installing Node.js

Section 7.1: Using Node Version Manager (nvm)

[Node Version Manager](#), otherwise known as nvm, is a bash script that simplifies the management of multiple Node.js versions.

To install nvm, use the provided install script:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

For windows there is a nvm-windows package with an installer. This [GitHub](#) page has the details for installing and using the nvm-windows package.

After installing nvm, run "nvm on" from command line. This enables nvm to control the node versions.

Note: You may need to restart your terminal for it to recognize the newly installed nvm command.

Then install the latest Node version:

```
$ nvm install node
```

You can also install a specific Node version, by passing the major, minor, and/or patch versions:

```
$ nvm install 6  
$ nvm install 4.2
```

To list the versions available for install:

```
$ nvm ls-remote
```

You can then switch versions by passing the version the same way you do when installing:

```
$ nvm use 5
```

You can set a specific version of Node that you installed to be the **default version** by entering:

```
$ nvm alias default 4.2
```

To display a list of Node versions that are installed on your machine, enter:

```
$ nvm ls
```

To use project-specific node versions, you can save the version in .nvmrc file. This way, starting to work with another project will be less error-prone after fetching it from its repository.

```
$ echo "4.2" > .nvmrc  
$ nvm use  
Found '/path/to/project/.nvmrc' with version <4.2>  
Now using node v4.2 (npm v3.7.3)
```

When Node is installed via nvm we don't have to use sudo to install global packages since they are installed in home folder. Thus `npm i -g http-server` works without any permission errors.

Section 7.2: Installing Node.js on Mac using package manager

Homebrew

You can install Node.js using the [Homebrew](#) package manager.

Start by updating brew:

```
brew update
```

You may need to change permissions or paths. It's best to run this before proceeding:

```
brew doctor
```

Next you can install Node.js by running:

```
brew install node
```

Once Node.js is installed, you can validate the version installed by running:

```
node -v
```

Macports

You can also install node.js through [Macports](#).

First update it to make sure the latest packages are referenced:

```
sudo port selfupdate
```

Then install nodejs and npm

```
sudo port install nodejs npm
```

You can now run node through CLI directly by invoking node. Also, you can check your current node version with

```
node -v
```

Section 7.3: Installing Node.js on Windows

Standard installation

All Node.js binaries, installers, and source files can be downloaded [here](#).

You can download just the `node.exe` runtime or use the Windows installer (`.msi`), which will also install `npm`, the recommended package manager for Node.js, and configure paths.

Installation by package manager

You can also install by package manager [Chocolatey](#) (Software Management Automation).

```
# choco install nodejs.install
```

More information about current version, you can find in the choco repository [here](#).

Section 7.4: Install Node.js on Ubuntu

Using the apt package manager

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

the node & npm versions in apt are outdated. This is how you can update them:

```
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (or lts, or a specific version)
```

Using the latest of specific version (e.g. LTS 6.x) directly from nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
apt-get install -y nodejs
```

Also, for the right way to install global npm modules, set the personal directory for them (eliminates the need for sudo and avoids EACCES errors):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

Section 7.5: Installing Node.js with n

First, there is a really nice wrapper for setting up n on your system. Just run:

```
curl -L https://git.io/n-install | bash
```

to install n. Then install binaries in a variety of ways:

latest

```
n latest
```

stable

```
n stable
```

lts

```
n lts
```

Any other version

```
n <version>
```

e.g. n 4.4.7

If this version is already installed, this command will activate that version.

Switching versions

n by itself will produce a selection list of installed binaries. Use up and down to find the one you want and Enter to activate it.

Section 7.6: Install Node.js From Source with APT package manager

Prerequisites

```
sudo apt-get install build-essential
sudo apt-get install python

[optional]
sudo apt-get install git
```

Get source and build

```
cd ~
git clone https://github.com/nodejs/node.git
```

OR For the latest LTS Node.js version 6.10.2

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzf node-v6.10.2.tar.gz
```

Change to the source directory such as in `cd ~/node-v6.10.2`

```
./configure
make
sudo make install
```

Section 7.7: Install Node.js from source on Centos, RHEL and Fedora

Prerequisites

- git
- clang and clang++ 3.4^ or gcc and g++ 4.8^
- Python 2.6 or 2.7
- GNU Make 3.81^

Get source

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Build

```
cd node
./configure
make -jX
su -c make install
```


X - the number of processor cores, greatly speeds up the build

Cleanup [Optional]

```
cd
rm -rf node
```

Section 7.8: Installing with Node Version Manager under Fish Shell with Oh My Fish!

[Node Version Manager](#) (nvm) greatly simplifies the management of Node.js versions, their installation, and removes the need for sudo when dealing with packages (e.g. npm **install** ...). [Fish Shell](#) (fish) "is a smart and user-friendly command line shell for OS X, Linux, and the rest of the family" that is a popular alternative among programmers to common shells such as bash. Lastly, [Oh My Fish](#) (omf) allows for customizing and installing packages within Fish shell.

This guide assumes you are already using Fish as your shell.

Install nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Install Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Note: You will be prompted to restart your terminal at this point. Go ahead and do so now.)

Install plugin-nvm for Oh My Fish

We will install [plugin-nvm](#) via Oh My Fish to expose nvm capabilities within the Fish shell:

```
omf install nvm
```

Install Node.js with Node Version Manager

You are now ready to use nvm. You may install and use the version of Node.js of your liking. Some examples:

- Install the most recent Node version: `nvm install node`
- Install 6.3.1 specifically: `nvm install 6.3.1`
- List installed versions: `nvm ls`
- Switch to a previously installed 4.3.1: `nvm use 4.3.1`

Final Notes

Remember again, that we no longer need sudo when dealing with Node.js using this method! Node versions, packages, and so on are installed in your home directory.

Section 7.9: Installing Node.js on Raspberry PI

To install v6.x update the packages

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Using the apt package manager

```
sudo apt-get install -y nodejs
```

Chapter 8: MySQL integration

In this topic you will learn how to integrate with Node.js using MySQL database management tool. You will learn various ways to connect and interact with data residing in mysql using a nodejs program and script.

Section 8.1: Connect to MySQL

One of the easiest ways to connect to MySQL is by using `mysql` module. This module handles the connection between Node.js app and MySQL server. You can install it like any other module:

```
npm install --save mysql
```

Now you have to create a mysql connection, which you can later query.

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'me',
  password  : 'secret',
  database  : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

In the next example you will learn how to query the connection object.

Section 8.2: Using a connection pool

a. Running multiple queries at same time

All queries in MySQL connection are done one after another. It means that if you want to do 10 queries and each query takes 2 seconds then it will take 20 seconds to complete whole execution. The solution is to create 10 connection and run each query in a different connection. This can be done automatically using connection pool

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass',
  database        : 'schema'
});

for(var i=0;i<10;i++){
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
    console.log(rows[0].example); //Show 1
  });
}
```

It will run all the 10 queries in parallel.

When you use pool you don't need the connection anymore. You can query directly the pool. MySQL module will search for the next free connection to execute your query.

b. Achieving multi-tenancy on database server with different databases hosted on it.

Multitenancy is a common requirement of enterprise application nowadays and creating connection pool for each database in database server is not recommended. so, what we can do instead is create connection pool with database server and then switch them between databases hosted on database server on demand.

Suppose our application has different databases for each firm hosted on database server. We will connect to respective firm database when user hits the application. Here is the example on how to do that:-

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Let me break down the example:-

When defining pool configuration i did not gave the database name but only gave database server i.e

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

so when we want to use the specific database on database server, we ask the connection to hit database by using:-

```
connection.changeUser({database : "firm1"});
```

you can refer the official documentation [here](#)

Section 8.3: Query a connection object with parameters

When you want to use user generated content in the SQL, it with done with parameters. For example for searching user with the name aminadav you should do:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
```

```
rows.forEach(function(row) {
  console.log(row.name, 'email address is', row.email);
});
} else {
  console.log('There were no results.');
```

Section 8.4: Query a connection object without parameters

You send the query as a string and in response callback with the answer is received. The callback gives you error, array of rows and fields. Each row contains all the column of the returned table. Here is a snippet for the following explanation.

```
connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;

  console.log('There are:', rows.length, ' users');
  console.log('First user name is:', rows[0].name)
});
```

Section 8.5: Run a number of queries with a single connection from a pool

There may be situations where you have setup a pool of MySQL connections, but you have a number of queries you would like to run in sequence:

SELECT 1; SELECT 2;

You *could* just run then using `pool.query` [as seen elsewhere](#), however if you only have one free connection in the pool you must wait until a connection becomes available before you can run the second query.

You can, however, retain an active connection from the pool and run as many queries as you would like using a single connection using `pool.getConnection`:

```
pool.getConnection(function (err, conn) { if (err) return callback(err); conn.query('SELECT 1 AS seq', function (err, rows) { if (err) throw err; conn.query('SELECT 2 AS seq', function (err, rows) { if (err) throw err; conn.release(); callback(); }); }); });
```

Note: You must remember to `release` the connection, otherwise there is one less MySQL connection available to the rest of the pool!

For more information on pooling MySQL connections [check out the MySQL docs](#).

Section 8.6: Export Connection Pool

```
// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
```

```
});

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}

// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});
});
```

Section 8.7: Return the query when an error occurs

You can attach the query executed to your `err` object when an error occurs:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err, result) { if (err) { //
Table 'test.pokedex' doesn't exist err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
callback(err); } else { callback(null, result); } });
```

Chapter 9: Readline

Section 9.1: Line-by-line file reading

```
const fs = require('fs');
const readline = require('readline');

const rl = readline.createInterface({
  input: fs.createReadStream('text.txt')
});

// Each new line emits an event - every time the stream receives \r, \n, or \r\n
rl.on('line', (line) => {
  console.log(line);
});

rl.on('close', () => {
  console.log('Done reading file');
});
```

Section 9.2: Prompting user input via CLI

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('What is your name?', (name) => {
  console.log(`Hello ${name}!`);

  rl.close();
});
```

Chapter 10: package.json

Section 10.1: Exploring package.json

A `package.json` file, usually present in the project root, contains metadata about your app or module as well as the list of dependencies to install from npm when running `npm install`.

To initialize a `package.json` type `npm init` in your command prompt.

To create a `package.json` with default values use:

```
npm init --yes
# or
npm init -y
```

To install a package and save it to `package.json` use:

```
npm install {package name} --save
```

You can also use the shorthand notation:

```
npm i -S {package name}
```

NPM aliases `-S` to `--save` and `-D` to `--save-dev` to save in your production or development dependencies respectively.

The package will appear in your dependencies; if you use `--save-dev` instead of `--save`, the package will appear in your `devDependencies`.

Important properties of `package.json`:

```
{
  "name": "module-name",
  "version": "10.3.1",
  "description": "An example module to illustrate the usage of a package.json",
  "author": "Your Name <your.name@example.org>",
  "contributors": [{
    "name": "Foo Bar",
    "email": "foo.bar@example.com"
  }],
  "bin": {
    "module-name": "./bin/module-name"
  },
  "scripts": {
    "test": "vows --spec --isolate",
    "start": "node index.js",
    "predeploy": "echo About to deploy",
    "postdeploy": "echo Deployed",
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"
  },
  "main": "lib/foo.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/username/repo"
  },
  "bugs": {
    "url": "https://github.com/username/issues"
  }
}
```



```

},
"keywords": [
  "example"
],
"dependencies": {
  "express": "4.2.x"
},
"devDependencies": {
  "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
},
"peerDependencies": {
  "moment": ">2.0.0"
},
"preferGlobal": true,
"private": true,
"publishConfig": {
  "registry": "https://your-private-hosted-npm.registry.domain.com"
},
"subdomain": "foobar",
"analyze": true,
"license": "MIT",
"files": [
  "lib/foo.js"
]
}

```

Information about some important properties:

name

The unique name of your package and should be down in lowercase. This property is required and your package will not install without it.

1. The name must be less than or equal to 214 characters.
2. The name can't start with a dot or an underscore.
3. New packages must not have uppercase letters in the name.

version

The version of the package is specified by [Semantic Versioning](#) (semver). Which assumes that a version number is written as MAJOR.MINOR.PATCH and you increment the:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backwards-compatible manner
3. PATCH version when you make backwards-compatible bug fixes

description

The description of the project. Try to keep it short and concise.

author

The author of this package.

bin

An object which is used to expose binary scripts from your package. The object assumes that the key is the name of

the binary script and the value a relative path to the script.

This property is used by packages that contain a CLI (command line interface).

script

A object which exposes additional npm commands. The object assumes that the key is the npm command and the value is the script path. These scripts can get executed when you run `npm run {command name}` or `npm run-script {command name}`.

Packages that contain a command line interface and are installed locally can be called without a relative path. So instead of calling `./node_modules/.bin/mocha` you can directly call `mocha`.

main

The main entry point to your package. When calling `require('{module name}')` in node, this will be actual file that is required.

It's highly advised that requiring the main file does not generate any side affects. For instance, requiring the main file should not start up a HTTP server or connect to a database. Instead, you should create something like `exports.init = function () {...}` in your main script.

keywords

An array of keywords which describe your package. These will help people find your package.

devDependencies

These are the dependencies that are only intended for development and testing of your module. The dependencies will be installed automatically unless the `NODE_ENV=production` environment variable has been set. If this is the case you can still these packages using `npm install --dev`

peerDependencies

If you are using this module, then `peerDependencies` lists the modules you must install alongside this one. For example, `moment-timezone` must be installed alongside `moment` because it is a plugin for `moment`, even if it doesn't directly `require("moment")`.

preferGlobal

A property that indicates that this page prefers to be installed globally using `npm install -g {module-name}`. This property is used by packages that contain a CLI (command line interface).

In all other situations you should NOT use this property.

publishConfig

The `publishConfig` is an object with configuration values that will be used for publishing modules. The configuration values that are set override your default npm configuration.

The most common use of the `publishConfig` is to publish your package to a private npm registry so you still have the benefits of npm but for private packages. This is done by simply setting URL of your private npm as value for the registry key.

files

This is an array of all the files to include in the published package. Either a file path or folder path can be used. All the contents of a folder path will be included. This reduces the total size of your package by only including the correct files to be distributed. This field works in conjunction with a `.npmignore` rules file.

[Source](#)

Section 10.2: Scripts

You can define scripts that can be executed or are triggered before or after another script.

```
{
  "scripts": {
    "pretest": "scripts/pretest.js",
    "test": "scripts/test.js",
    "posttest": "scripts/posttest.js"
  }
}
```

In this case, you can execute the script by running either of these commands:

```
$ npm run-script test
$ npm run test
$ npm test
$ npm t
```

Pre-defined scripts

Script Name	Description
prepublish	Run before the package is published.
publish, postpublish	Run after the package is published.
preinstall	Run before the package is installed.
install, postinstall	Run after the package is installed.
preuninstall, uninstall	Run before the package is uninstalled.
postuninstall	Run after the package is uninstalled.
preversion, version	Run before bump the package version.
postversion	Run after bump the package version.
pretest, test, posttest	Run by the npm <code>test</code> command
prestop, stop, poststop	Run by the npm <code>stop</code> command
prestart, start, poststart	Run by the npm <code>start</code> command
prerestart, restart, postrestart	Run by the npm <code>restart</code> command

User-defined scripts

You can also define your own scripts the same way you do with the pre-defined scripts:

```
{
  "scripts": {
    "preci": "scripts/preci.js",
    "ci": "scripts/ci.js",
    "postci": "scripts/postci.js"
  }
}
```

In this case, you can execute the script by running either of these commands:

```
$ npm run-script ci
```

```
$ npm run ci
```

User-defined scripts also supports *pre* and *post* scripts, as shown in the example above.

Section 10.3: Basic project definition

```
{
  "name": "my-project",
  "version": "0.0.1",
  "description": "This is a project.",
  "author": "Someone <someone@example.com>",
  "contributors": [{
    "name": "Someone Else",
    "email": "else@example.com"
  }],
  "keywords": ["improves", "searching"]
}
```

Field	Description
name	a required field for a package to install. Needs to be lowercase, single word without spaces. (Dashes and underscores allowed)
version	a required field for the package version using semantic versioning .
description	a short description of the project
author	specifies the author of the package
contributors	an array of objects, one for each contributor
keywords	an array of strings, this will help people finding your package

Section 10.4: Dependencies

"dependencies": { "module-name": "0.1.0" }

- **exact:** 0.1.0 will install that specific version of the module.
- **newest minor version:** ^0.1.0 will install the newest minor version, for example 0.2.0, but won't install a module with a higher major version e.g. 1.0.0
- **newest patch:** 0.1.x or ~0.1.0 will install the newest patch version available, for example 0.1.4, but won't install a module with higher major or minor version, e.g. 0.2.0 or 1.0.0.
- **wildcard:** * will install the latest version of the module.
- **git repository:** the following will install a tarball from the master branch of a git repo. A #sha, #tag or #branch can also be provided:
 - **GitHub:** user/project or user/project#v1.0.0
 - **url:** git://gitlab.com/user/project.git or git://gitlab.com/user/project.git#develop
- **local path:** file:../lib/project

After adding them to your package.json, use the command npm **install** in your project directory in terminal.

devDependencies

```
"devDependencies": {
  "module-name": "0.1.0"
}
```

For dependencies required only for development, like testing styling proxies ext. Those dev-dependencies won't be installed when running "npm install" in production mode.

Section 10.5: Extended project definition

Some of the additional attributes are parsed by the npm website like repository, bugs or homepage and shown in the infobox for this packages

```
{
  "main": "server.js",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"
  },
  "bugs": {
    "url": "https://github.com/<accountname>/<repositoryname>/issues"
  },
  "homepage": "https://github.com/<accountname>/<repositoryname>#readme",
  "files": [
    "server.js", // source files
    "README.md", // additional files
    "lib" // folder with all included files
  ]
}
```

Field	Description
main	Entry script for this package. This script is returned when a user requires the package.
repository	Location and type of the public repository
bugs	Bugtracker for this package (e.g. github)
homepage	Homepage for this package or the general project
files	List of files and folders which should be downloaded when a user does a npm install <packagename>

Chapter 11: Event Emitters

Section 11.1: Basics

Event Emitters are built into Node, and are for pub-sub, a pattern where a *publisher* will emit events, which *subscribers* can listen and react to. In Node jargon, publishers are called *Event Emitters*, and they emit events, while subscribers are called *listeners*, and they react to the events.

```
// Require events to start using them const EventEmitter = require('events').EventEmitter; // Dogs have events to
publish, or emit class Dog extends EventEmitter {}; class Food {}; let myDog = new Dog(); // When myDog is chewing,
run the following function myDog.on('chew', (item) => { if (item instanceof Food) { console.log('Good dog'); } else {
console.log(`Time to buy another ${item}`); } }); myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy
another shoe') const bacon = new Food(); myDog.emit('chew', bacon); // Will result in console.log('Good dog')
```

In the above example, the dog is the publisher/EventEmitter, while the function that checks the item was the subscriber/listener. You can make more listeners too:

```
myDog.on('bark', () => { console.log('WHO'S AT THE DOOR?'); // Panic });
```

There can also be multiple listeners for a single event, and even remove listeners:

```
myDog.on('chew', takeADeepBreathe); myDog.on('chew', calmDown); // Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

If you want to listen to a event only once, you can use:

```
myDog.once('chew', pet);
```

Which will remove the listener automatically without race conditions.

Section 11.2: Get the names of the events that are subscribed to

The function **EventEmitter.eventNames()** will return an array containing the names of the events currently subscribed to.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter {}

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
  console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
  console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
  console.log("a data just occurred!!");
});

console.log(emitter.eventNames()); //=> ["message", "data"]
emitter.removeAllListeners("data");//=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]
```

Section 11.3: HTTP Analytics through an Event Emitter

In the HTTP server code (e.g. `server.js`):

```
const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// Set up an HTTP server
const http = require('http')
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents
```

In supervisor code (e.g. `supervisor.js`):

```
const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})
```

Whenever the server gets a request, it will emit an event called `request` which the supervisor is listening for, and then the supervisor can react to the event.

Section 11.4: Get the number of listeners registered to listen for a specific event

The function `Emitter.listenerCount(eventName)` will return the number of listeners that are currently listening for the event provided as argument

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();

emitter
.on("data", ()=>{ // add listener for data event
  console.log("data event emitter");
});

console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ //add listener for message event
  console.log("message event emitted");
});
console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff)=>{ //add another listener for data event
  console.log(`Tell me my ${stuff}`);
})
```

```
console.log(emitter.listenerCount("data")) // => 2  
console.log(emitter.listenerCount("message"))// => 1
```


Chapter 12: Autoreload on changes

Section 12.1: Autoreload on source code changes using nodemon

The nodemon package makes it possible to automatically reload your program when you modify any file in the source code.

Installing nodemon globally

```
npm install -g nodemon (or npm i -g nodemon)
```

Installing nodemon locally

In case you don't want to install it globally

```
npm install --save-dev nodemon (or npm i -D nodemon)
```

Using nodemon

Run your program with `nodemon entry.js` (or `nodemon entry`)

This replaces the usual use of `node entry.js` (or `node entry`).

You can also add your nodemon startup as an npm script, which might be useful if you want to supply parameters and not type them out every time.

Add **package.json**:

```
"scripts": {  
  "start": "nodemon entry.js -devmode -something 1"  
}
```

This way you can just use `npm start` from your console.

Section 12.2: Browsersync

Overview

[Browsersync](#) is a tool that allows for live file watching and browser reloading. It's available as a [NPM package](#).

Installation

To install Browsersync you'll first need to have [Node.js](#) and NPM installed. For more information see the SO documentation on [Installing and Running Node.js](#).

Once your project is set up you can install Browsersync with the following command:

```
$ npm install browser-sync -D
```

This will install Browsersync in the local `node_modules` directory and save it to your developer dependencies.

If you'd rather install it globally use the `-g` flag in place of the `-D` flag.

Windows Users

If you're having trouble installing Browsersync on Windows you may need to install Visual Studio so you can access the build tools to install Browsersync. You'll then need to specify the version of Visual Studio you're using like so:

```
$ npm install browser-sync --msvs_version=2013 -D
```

This command specifies the 2013 version of Visual Studio.

Basic Usage

To automatically reload your site whenever you change a JavaScript file in your project use the following command:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Replace `myproject.dev` with the web address that you are using to access your project. Browsersync will output an alternate address that can be used to access your site through the proxy.

Advanced Usage

Besides the command line interface that was described above Browsersync can also be used with [Grunt.js](#) and [Gulp.js](#).

Grunt.js

Usage with Grunt.js requires a plugin that can be installed like so:

```
$ npm install grunt-browser-sync -D
```

Then you'll add this line to your `gruntfile.js`:

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

Browsersync works as a [CommonJS](#) module, so there's no need for a Gulp.js plugin. Simply require the module like so:

```
var browserSync = require('browser-sync').create();
```

You can now use the [Browsersync API](#) to configure it to your needs.

API

The Browsersync API can be found here: <https://browsersync.io/docs/api>

Chapter 13: Environment

Section 13.1: Accessing environment variables

The `process.env` property returns an object containing the user environment.

It returns an object like this one :

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

```
process.env.HOME // '/Users/maciej'
```

If you set environment variable F00 to foobar, it will be accessible with:

```
process.env.F00 // 'foobar'
```

Section 13.2: process.argv command line arguments

`process.argv` is an array containing the command line arguments. The first element will be node, the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.

Code Example:

Output sum of all command line arguments

`index.js`

```
var sum = 0;
for (i = 2; i < process.argv.length; i++) {
  sum += Number(process.argv[i]);
}

console.log(sum);
```

Usage Exaple:

```
node index.js 2 5 6 7
```

Output will be **20**

A brief explanation of the code:

Here in for loop `for (i = 2; i < process.argv.length; i++)` loop begins with 2 because first two elements in `process.argv` array **always** is `['path/to/node.exe', 'path/to/js/file', ...]`

Converting to number `Number(process.argv[i])` because elements in `process.argv` array **always** is string

Section 13.3: Loading environment properties from a "property file"

- Install properties reader:

```
npm install properties-reader --save
```

- Create a **directory env** to store your properties files:

```
mkdir env
```

- Create **environments.js**:

```
process.argv.forEach(function (val, index, array) {  
  var arg = val.split("=");  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./env/' + arg[1] + '.properties');  
      module.exports = env;  
    }  
  }  
});
```

- Sample **development.properties** properties file:

```
# Dev properties  
[main]  
# Application port to run the node server  
app.port=8080  
  
[database]  
# Database connection to mysql  
mysql.host=localhost  
mysql.port=2500  
...
```

- Sample usage of the loaded properties:

```
var environment = require('./environments');  
var PropertiesReader = require('properties-reader');  
var properties = new PropertiesReader(environment);  
  
var someVal = properties.get('main.app.port');
```

- Starting the express server

```
npm start env=development
```

or

```
npm start env=production
```

Section 13.4: Using different Properties/Configuration for different environments like dev, qa, staging etc

Large scale applications often need different properties when running on different environments. we can achieve

this by passing arguments to NodeJs application and using same argument in node process to load specific environment property file.

Suppose we have two property files for different environment.

- dev.json

```
{
  PORT : 3000,
  DB : {
    host : "localhost",
    user : "bob",
    password : "12345"
  }
}
```

- qa.json

```
{
  PORT : 3001,
  DB : {
    host : "where_db_is_hosted",
    user : "bob",
    password : "54321"
  }
}
```

Following code in application will export respective property file which we want to use.

Suppose the code is in environment.js

```
process.argv.forEach(function (val, index, array) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./' + arg[1] + '.json');
      module.exports = env;
    }
  }
});
```

We give arguments to the application like following

```
node app.js env=dev
```

if we are using process manager like *forever* than it as simple as

```
forever start app.js env=dev
```

How to use the configuration file

```
var env= require("environment.js");
```

Chapter 14: Callback to Promise

Section 14.1: Promisifying a callback

Callback-based:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
  if (error) {
    console.log(error);
  }

  // normal code here
});
```

This uses bluebird's `promisifyAll` method to promisify what is conventionally callback-based code like above. bluebird will make a promise version of all the methods in the object, those promise-based methods names has `Async` appended to them:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {

  // normal code here
})
.catch(console.error);
```

If only specific methods need to be promisified, just use its `promisify`:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {

  // normal code here
});
.catch(console.error);
```

There are some libraries (e.g., `MassiveJS`) that can't be promisified if the immediate object of the method is not passed on second parameter. In that case, just pass the immediate object of the method that need to be promisified on second parameter and enclosed it in `context` property.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {

  // normal code here
});
.catch(console.error);
```

Section 14.2: Manually promisifying a callback

Sometimes it might be necessary to manually promisify a callback function. This could be for a case where the callback does not follow the standard [error-first format](#) or if additional logic is needed to promisify:

Example with `fs.exists(path, callback)`:

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

```

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists is a boolean
      if (exists) {
        // Resolve successfully
        resolve();
      } else {
        // Reject with error
        reject(new Error('path does not exist'));
      }
    });
  });
}

// Use as a promise now
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // file does not exist
  console.error(err);
});

```

Section 14.3: setTimeout promisified

```

function wait(ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, ms)
  })
}

```

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 which might print output over a period of time rather than printing and exiting immediately), use `child_process.spawn()`.

This method spawns a new process using a given command and an array of arguments. The return value is an instance of `ChildProcess`, which in turn provides the `stdout` and `stderr` properties. Both of those streams 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: ${data}`);
});

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

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

Another example command:

```
zip -0vr "archive" ./image.png
```

Might be written as:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

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 not a stream), use `child_process.exec`. For example, if you wanted to run the command `cat *.js file | wc -l`, with no options, that would look like this:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

The function accepts up to three parameters:


```
child_process.exec(command[, options][, callback]);
```

The command parameter is a string, and is required, while the options object and callback are both optional. If no options object is specified, then exec will use the following as a default:

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

The options object also supports a shell parameter, which is by default `/bin/sh` on UNIX and `cmd.exe` on Windows, a uid option for setting the user identity of the process, and a gid option for the group identity.

The callback, which is called when the command is done executing, is called with the three arguments (`err`, `stdout`, `stderr`). If the command executes successfully, `err` will be `null`, otherwise it will be an instance of `Error`, with `err.code` being the exit code of the process and `err.signal` being the signal that was sent to terminate it.

The `stdout` and `stderr` arguments are the output of the command. It is decoded with the encoding specified in the options object (default: string), but can otherwise be returned as a `Buffer` object.

There also exists a synchronous version of `exec`, which is `execSync`. The synchronous version does not take a callback, and will return `stdout` instead of an instance of `ChildProcess`. If the synchronous version encounters an error, it *will* throw and halt your program. It looks like this:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

Section 15.3: Spawning a process to run an executable

If you are looking to run a file, such as an executable, use `child_process.execFile`. Instead of spawning a shell like `child_process.exec` would, it will directly create a new process, which is slightly more efficient than running a command. The function can be used like so:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr) => {
  if (err) {
    throw err;
  }

  console.log(stdout);
});
```

Unlike `child_process.exec`, this function will accept up to four parameters, where the second parameter is an array of arguments you'd like to supply to the executable:

```
child_process.execFile(file[, args][, options][, callback]);
```

Otherwise, the options and callback format are otherwise identical to `child_process.exec`. The same goes for the synchronous version of the function:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Chapter 16: Cluster Module

Section 16.1: Hello World

This is your `cluster.js`:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

This is your `main server.js`:

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

function startServer() {
  const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello Http');
  });

  server.listen(3000);
}

if(!module.parent) {
  // Start server if file is run directly
  startServer();
} else {
  // Export server, if file is referenced via cluster
  module.exports = startServer;
}
```

In this example, we host a basic web server, however, we spin up workers (child processes) using the built-in **cluster** module. The number of processes forked depend on the number of CPU cores available. This enables a Node.js application to take advantage of multi-core CPUs, since a single instance of Node.js runs in a single thread. The application will now share the port 8000 across all the processes. Loads will automatically be distributed between workers using the Round-Robin method by default.

Section 16.2: Cluster Example

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, application can be launched in a cluster of Node.js processes to handle the load.

The cluster module allows you to easily create child processes that all share server ports.

Following example create the worker child process in main process that handles the load across multiple cores.

Example

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUs

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();    //creating child process
  }

  //on exit of cluster
  cluster.on('exit', (worker, code, signal) => {
    if (signal) {
      console.log(`worker was killed by signal: ${signal}`);
    } else if (code !== 0) {
      console.log(`worker exited with error code: ${code}`);
    } else {
      console.log('worker success!');
    }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

Chapter 17: Exception handling

Section 17.1: Handling Exception In Node.Js

Node.js has 3 basic ways to handle exceptions/errors:

1. **try-catch** block
2. **error** as the first argument to a callback
3. emit an **error** event using eventEmitter

try-catch is used to catch the exceptions thrown from the synchronous code execution. If the caller (or the caller's caller, ...) used try/catch, then they can catch the error. If none of the callers had try-catch than the program crashes.

If using try-catch on an async operation and exception was thrown from callback of async method than it will not get caught by try-catch. To catch an exception from async operation callback, it is preferred to use *promises*. Example to understand it better

```
// ** Example - 1 **
function doSomeSynchronousOperation(req, res) {
    if(req.body.username === ''){
        throw new Error('User Name cannot be empty');
    }
    return true;
}

// calling the method above
try {
    // synchronous code
    doSomeSynchronousOperation(req, res)
} catch(e) {
    //exception handled here
    console.log(e.message);
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
    // imitating async operation
    return setTimeout(function(){
        cb(null, []);
    }, 1000);
}

try {
    // asynchronous code
    doSomeAsynchronousOperation(req, res, function(err, rs){
        throw new Error("async operation exception");
    })
} catch(e) {
    // Exception will not get handled here
    console.log(e.message);
}

// The exception is unhandled and hence will cause application to break
```

callbacks are mostly used in Node.js as callback delivers an event asynchronously. The user passes you a function (the callback), and you invoke it sometime later when the asynchronous operation completes. The usual pattern is that the callback is invoked as a *callback(err, result)*, where only one of err and result is non-null, depending on whether the operation succeeded or failed.

```
function doSomeAsynchronousOperation(req, res, callback) {
  setTimeout(function(){
    return callback(new Error('User Name cannot be empty'));
  }, 1000);
  return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
  if (err) {
    //exception handled here
    console.log(err.message);
  }

  //do some stuff with valid data
});
```

emit For more complicated cases, instead of using a callback, the function itself can return an EventEmitter object, and the caller would be expected to listen for error events on the emitter.

```
const EventEmitter = require('events');

function doSomeAsynchronousOperation(req, res) {
  let myEvent = new EventEmitter();

  // runs asynchronously
  setTimeout(function(){
    myEvent.emit('error', new Error('User Name cannot be empty'));
  }, 1000);

  return myEvent;
}

// Invoke the function
let event = doSomeAsynchronousOperation(req, res);

event.on('error', function(err) {
  console.log(err);
});

event.on('done', function(result) {
  console.log(result); // true
});
```

Section 17.2: Unhandled Exception Management

Because Node.js runs on a single process uncaught exceptions are an issue to be aware of when developing applications.

Silently Handling Exceptions

Most of the people let node.js server(s) silently swallow up the errors.

- Silently handling the exception

```
process.on('uncaughtException', function (err) {
  console.log(err);
});
```

This is bad, it will work but:

- Root cause will remain unknown, as such will not contribute to resolution of what caused the Exception (Error).
- In case of database connection (pool) gets closed for some reason this will result in constant propagation of errors, meaning that server will be running but it will not reconnect to db.

Returning to Initial state

In case of an " `uncaughtException` " it is good to restart the server and return it to its **initial state**, where we know it will work. Exception is logged, application is terminated but since it will be running in a container that will make sure that the server is running we will achieve restarting of the server (returning to the initial working state).

- Installing the forever (or other CLI tool to make sure that node server runs continuously)

```
npm install forever -g
```

- Starting the server in forever

```
forever start app.js
```

Reason why it started and why we use forever is after the server is **terminated** forever process will start the server again.

- Restarting the server

```
process.on('uncaughtException', function (err) {
  console.log(err);

  // some logging mechanism
  // ....

  process.exit(1); // terminates process
});
```

On a side note there was a way also to handle exceptions with **Clusters and Domains**.

Domains are deprecated more information [here](#).

Section 17.3: Errors and Promises

Promises handle errors differently to synchronous or callback-driven code.

```
const p = new Promise(function (resolve, reject) {
  reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `.then` will not be called
p
  .then(() => {
    console.log("won't be called");
  })
  .catch(e => {
    console.log(e.message); // output: Oops
```

```
})  
// once the error is caught, execution flow resumes  
.then(() => {  
  console.log('hello!'); // output: hello!  
});
```

currently, errors thrown in a promise that are not caught results in the error being swallowed, which can make it difficult to track down the error. This can be [solved](#) using linting tools like [eslint](#) or by ensuring you always have a **catch** clause.

This behaviour is deprecated [in node 8](#) in favour of terminating the node process.

Chapter 18: Keep a node application constantly running

Section 18.1: Use PM2 as a process manager

PM2 lets you run your nodejs scripts forever. In the event that your application crashes, PM2 will also restart it for you.

Install PM2 globally to manager your nodejs instances

```
npm install pm2 -g
```

Navigate to the directory in which your nodejs script resides and run the following command each time you want to start a nodejs instance to be monitored by pm2:

```
pm2 start server.js --name "app1"
```

Useful commands for monitoring the process

1. List all nodejs instances managed by pm2

```
pm2 list
```

```
[tknew:~/Unitech/pm2] master(+84/-121)+* ± pm2 list
PM2 - Process listing
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
bashscript.sh	6	fork	8278	online	0	10s	1.379 MB	/home/tknew/.pm2/logs/bashscript.sh-err.log
checker	5	cluster	0	stopped	0	2m	0 B	/home/tknew/.pm2/logs/checker-err.log
interface-api	3	cluster	7526	online	0	3m	15.445 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	2	cluster	7517	online	0	3m	15.453 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	1	cluster	7512	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	0	cluster	7507	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log

2. Stop a particular nodejs instance

```
pm2 stop <instance named>
```

3. Delete a particular nodejs instance

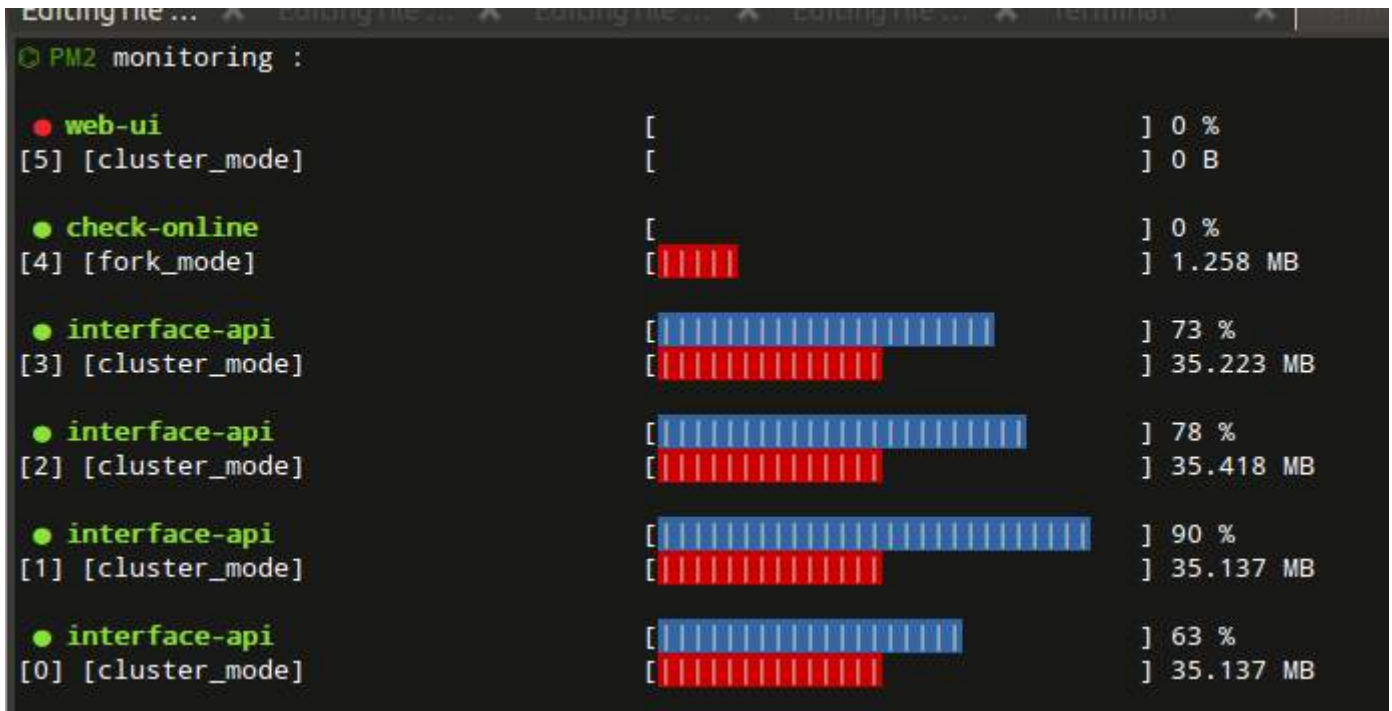
```
pm2 delete <instance name>
```

4. Restart a particular nodejs instance

```
pm2 restart <instance name>
```

5. Monitoring all nodejs instances

```
pm2 monit
```



6. Stop pm2

```
pm2 kill
```

7. As opposed to restart, which kills and restarts the process, reload achieves a 0-second-downtime reload

```
pm2 reload <instance name>
```

8. View logs

```
pm2 logs <instance_name>
```

Section 18.2: Running and stopping a Forever daemon

To start the process:

```
$ forever start index.js
warn:   --minUptime not set. Defaulting to: 1000ms
warn:   --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
info:   Forever processing file: index.js
```

List running Forever instances:

```
$ forever list
info:   Forever processes running
```

data:	index	uid	command	script	forever pid	id	logfile
uptime							
data:	[0]	f4Kt	/usr/bin/nodejs	src/index.js	2131	2146	/root/.forever/f4Kt.log
0:0:0:11.485							

Stop the first process:

```
$ forever stop 0  
$ forever stop 2146  
$ forever stop --uid f4Kt  
$ forever stop --pidFile 2131
```

Section 18.3: Continuous running with nohup

An alternative to forever on Linux is nohup.

To start a nohup instance

1. cd to the location of app.js or wwwfolder
2. run `nohup nodejs app.js &`

To kill the process

1. run `ps -ef | grep nodejs`
2. kill `-9 <the process number>`

Chapter 19: Uninstalling Node.js

Section 19.1: Completely uninstall Node.js on Mac OSX

In Terminal on your Mac operating system, enter the following 2 commands:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do sudo rm /usr/local/${f}; done  
  
sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

Section 19.2: Uninstall Node.js on Windows

To uninstall Node.js on Windows, use Add or Remove Programs like this:

1. Open Add or Remove Programs from the start menu.
2. Search for Node.js.

Windows 10:

3. Click Node.js.
4. Click Uninstall.
5. Click the new Uninstall button.

Windows 7-8.1:

3. Click the Uninstall button under Node.js.

Chapter 20: nvm - Node Version Manager

Section 20.1: Install NVM

You can use curl:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Or you can use wget:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Section 20.2: Check NVM version

To verify that nvm has been installed, do:

```
command -v nvm
```

which should output 'nvm' if the installation was successful.

Section 20.3: Installing an specific Node version

Listing available remote versions for installation

```
nvm ls-remote
```

Installing a remote version

```
nvm install <version>
```

For example

```
nvm install 0.10.13
```

Section 20.4: Using an already installed node version

To list available local versions of node through NVM:

```
nvm ls
```

For example, if nvm ls returns:

```
$ nvm ls
  v4.3.0
  v5.5.0
```

You can switch to v5.5.0 with:

```
nvm use v5.5.0
```

Section 20.5: Install nvm on Mac OSX

INSTALLATION PROCESS

You can install Node Version Manager using git, curl or wget. You run these commands in **Terminal** on **Mac OSX**.

curl example:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

wget example:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

TEST THAT NVM WAS PROPERLY INSTALLED

To test that nvm was properly installed, close and re-open Terminal and enter nvm. If you get a **nvm: command not found** message, your OS may not have the necessary **.bash_profile** file. In Terminal, enter `touch ~/.bash_profile` and run the above install script again.

If you still get **nvm: command not found**, try the following:

- In Terminal, enter `nano .bashrc`. You should see an export script almost identical to the following:

```
export NVM_DIR="/Users/johndoe/.nvm" [ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
```

- Copy the export script and remove it from **.bashrc**
- Save and Close the .bashrc file (CTRL+O – Enter – CTRL+X)
- Next, enter `nano .bash_profile` to open the Bash Profile
- Paste the export script you copied into the Bash Profile on a new line
- Save and Close the Bash Profile (CTRL+O – Enter – CTRL+X)
- Finally enter `nano .bashrc` to re-open the **.bashrc** file
- Paste the following line into the file:

```
source ~/.nvm/nvm.sh
```

- Save and Close (CTRL+O – Enter – CTRL+X)
- Restart Terminal and enter nvm to test if it's working

Section 20.6: Run any arbitrary command in a subshell with the desired version of node

List all the node versions installed

```
nvm ls
v4.5.0
v6.7.0
```

Run command using any node installed version

```
nvm run 4.5.0 --version or nvm exec 4.5.0 node --version
Running node v4.5.0 (npm v2.15.9)
```

```
v4.5.0
```

```
nvm run 6.7.0 --version or nvm exec 6.7.0 node --version  
Running node v6.7.0 (npm v3.10.3)  
v6.7.0
```

using alias

```
nvm run default --version or nvm exec default node --version  
Running node v6.7.0 (npm v3.10.3)  
v6.7.0
```

To install node LTS version

```
nvm install --lts
```

Version Switching

```
nvm use v4.5.0 or nvm use stable ( alias )
```

Section 20.7: Setting alias for node version

If you want to set some alias name to installed node version, do:

```
nvm alias <name> <version>
```

Similar to unalias, do:

```
nvm unalias <name>
```

A proper usecase would be, if you want to set some other version than stable version as default alias. **default** aliased versions are loaded on console by default.

Like:

```
nvm alias default 5.0.1
```

Then every time **console/terminal** starts 5.0.1 would be present by default.

Note:

```
nvm alias # lists all aliases created on nvm
```

Chapter 21: http

Section 21.1: http server

A basic example of HTTP server.

write following code in http_server.js file:

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

var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: ' + connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

then from your http_server.js location run this command:

```
node http_server.js
```

you should see this result:

```
> Start HTTP on port 80
```

now you need to test your server, you need to open your internet browser and navigate to this url:

```
http://127.0.0.1:80
```

if your machine running Linux server you can test it like this:

```
curl 127.0.0.1:80
```

you should see following result:

```
ok
```

in your console, that running the app, you will see this results:

```
> Request received: HTTP GET /
> Client IP: ::ffff:127.0.0.1
```


Section 21.2: http client

a basic example for http client:

write the following code in http_client.js file:

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

var options = {
  hostname: '127.0.0.1',
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function(chunk) {
    console.log('Response: ' + chunk);
  });
  res.on('end', function(chunk) {
    console.log('Response ENDED');
  });
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

req.end();
```

then from your http_client.js location run this command:

```
node http_client.js
```

you should see this result:

```
> STATUS: 200
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016 11:27:17 GMT","connection":"close","transfer-encoding":"chunked"}
> Response: OK
> Response ENDED
```

note: this example depends on http server example.

Chapter 22: 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 writeable
Transform Stream	type of duplex stream that can transform data as it is being read and then written

Section 22.1: Read Data from TextFile with Streams

I/O 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(8000);
```

This code works but it's bulky and buffers up the entire data.txt 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(8000);
```

Here .pipe() takes care of listening for 'data' and 'end' events from the fs.createReadStream(). This code is not only 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 22.2: Piping streams

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

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

var readable = fs.createReadStream('file1.txt')
var writable = fs.createWriteStream('file2.txt')
```

```
readable.pipe(writable) // returns writable
```

When writable streams are also readable streams, i.e. when they're *duplex* streams, you can continue piping it to other writable streams.

```
var zlib = require('zlib')

fs.createReadStream('style.css')
  .pipe(zlib.createGzip()) // The returned object, zlib.Gzip, is a duplex stream.
  .pipe(fs.createWriteStream('style.css.gz'))
```

Readable streams can also be piped into multiple streams.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css'))
```

Note that you must pipe to the output streams synchronously (at the same time) before any data 'flows'. Failure to do so might lead to incomplete data being streamed.

Also note that stream objects can emit error events; be sure to responsibly handle these events on *every* stream, as needed:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

Section 22.3: Creating your own readable/writable stream

We will see stream objects being returned by modules like fs etc but what if we want to create our own streamable object.

To create Stream object we need to use the stream module provided by Nodejs

```
var fs = require("fs");
var stream = require("stream").Writable;

/*
 * Implementing the write function in writable stream class.
 * This is the function which will be used when other stream is piped into this
 * writable stream.
 */
stream.prototype._write = function(chunk, data){
  console.log(data);
}

var customStream = new stream();

fs.createReadStream("am1.js").pipe(customStream);
```

This will give us our own custom writable stream. we can implement anything within the `_write` function. Above method works in Nodejs 4.x.x version but in Nodejs 6.x **ES6** introduced classes therefore syntax have changed. Below is the code for 6.x version of Nodejs

```
const Writable = require('stream').Writable;
```

```
class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }

  _write(chunk, encoding, callback) {
    console.log(chunk);
  }
}
```

Section 22.4: Why Streams?

Lets examine the following two examples for reading a file's contents:

The first one, which uses an async method for reading a file, and providing a callback function which is called once the file is fully read into the memory:

```
fs.readFile(`${__dirname}/utils.js`, (err, data) => {
  if (err) {
    handleError(err);
  } else {
    console.log(data.toString());
  }
})
```

And the second, which uses streams in order to read the file's content, piece by piece:

```
var fileStream = fs.createReadStream(`${__dirname}/file`);
var fileContent = '';
fileStream.on('data', data => {
  fileContent += data.toString();
})

fileStream.on('end', () => {
  console.log(fileContent);
})

fileStream.on('error', err => {
  handleError(err)
})
```

It's worth mentioning that both examples do the **exact same thing**. What's the difference then?

- The first one is shorter and looks more elegant
- The second lets you do some processing on the file **while** it is being read (!)

When the files you deal with are small then there is no real effect when using streams, but what happens when the file is big? (so big that it takes 10 seconds to read it into memory)

Without streams you'll be waiting, doing absolutely nothing (unless your process does other stuff), until the 10 seconds pass and the file is **fully read**, and only then you can start processing the file.

With streams, you get the file's contents piece by piece, **right when they're available** - and that lets you process the file **while** it is being read.

The above example does not illustrate how streams can be utilized for work that cannot be done when going the

callback fashion, so let's look at another example:

I would like to download a gzip file, unzip it and save its content to the disk. Given the file's url this is what's needed to be done:

- Download the file
- Unzip the file
- Save it to disk

Here's a [small file][1], which is stored in my S3 storage. The following code does the above in the callback fashion.

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // here, the whole file was downloaded

  zlib.gunzip(data.Body, (err, data) => {
    // here, the whole file was unzipped

    fs.writeFile(`${__dirname}/tweets.json`, data, err => {
      if (err) console.error(err)

      // here, the whole file was written to disk
      var endTime = Date.now()
      console.log(`${endTime - startTime} milliseconds`) // 1339 milliseconds
    })
  })
})

// 1339 milliseconds
```

This is how it looks using streams:

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`${__dirname}/tweets.json`));

// 1204 milliseconds
```

Yep, it's not faster when dealing with small files - the tested file weighs 80KB. Testing this on a bigger file, 71MB gzipped (382MB unzipped), shows that the streams version is much faster

- It took 20925 milliseconds to download 71MB, unzip it and then write 382MB to disk - **using the callback fashion.**
- In comparison, it took 13434 milliseconds to do the same when using the streams version (35% faster, for a not-so-big file)

Chapter 23: Deploying Node.js applications in production

Section 23.1: Setting NODE_ENV="production"

Production deployments will vary in many ways, but a standard convention when deploying in production is to define an environment variable called `NODE_ENV` and set its value to *"production"*.

Runtime flags

Any code running in your application (including external modules) can check the value of `NODE_ENV`:

```
if(process.env.NODE_ENV === 'production') {  
  // We are running in production mode  
} else {  
  // We are running in development mode  
}
```

Dependencies

When the `NODE_ENV` environment variable is set to *'production'* all `devDependencies` in your `package.json` file will be completely ignored when running `npm install`. You can also enforce this with a `--production` flag:

```
npm install --production
```

For setting `NODE_ENV` you can use any of these methods

method 1: set NODE_ENV for all node apps

Windows :

```
set NODE_ENV=production
```

Linux or other unix based system :

```
export NODE_ENV=production
```

This sets `NODE_ENV` for current bash session thus any apps started after this statement will have `NODE_ENV` set to `production`.

method 2: set NODE_ENV for current app

```
NODE_ENV=production node app.js
```

This will set `NODE_ENV` for the current app only. This helps when we want to test our apps on different environments.

method 3: create .env file and use it

This uses the idea explained [here](#). Refer this post for more detailed explanation.

Basically you create `.env` file and run some bash script to set them on environment.

To avoid writing a bash script, the [env-cmd](#) package can be used to load the environment variables defined in the

`.env` file.

```
env-cmd .env node app.js
```

method 4: Use `cross-env` package

This [package](#) allows environment variables to be set in one way for every platform.

After installing it with npm, you can just add it to your deployment script in `package.json` as follows:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Section 23.2: Manage app with process manager

It's a good practice to run NodeJS apps controlled by process managers. Process manager helps to keep application alive forever, restart on failure, reload without downtime and simplifies administrating. Most powerful of them (like [PM2](#)) have a built-in load balancer. PM2 also enables you to manage application logging, monitoring, and clustering.

PM2 process manager

Installing PM2:

```
npm install pm2 -g
```

Process can be started in cluster mode involving integrated load balancer to spread load between processes:

```
pm2 start app.js -i 0 --name "api" (-i is to specify number of processes to spawn. If it is 0, then process number will be based on CPU cores count)
```

While having multiple users in production, its must to have a single point for PM2. Therefore pm2 command must be prefixed with a location (for PM2 config) else it will spawn a new pm2 process for every user with config in respective home directory. And it will be inconsistent.

Usage: `PM2_HOME=/etc/.pm2 pm2 start app.js`

Section 23.3: Deployment using process manager

Process manager is generally used in production to deploy a nodejs app. The main functions of a process manager are restarting the server if it crashes, checking resource consumption, improving runtime performance, monitoring etc.

Some of the popular process managers made by the node community are forever, pm2, etc.

Forever

[forever](#) is a command-line interface tool for ensuring that a given script runs continuously. forever's simple interface makes it ideal for running smaller deployments of Node .js apps and scripts.

forever monitors your process and restarts it if it crashes.

Install forever globally.

```
$ npm install -g forever
```

Run application :

```
$ forever start server.js
```

This starts the server and gives an id for the process(starts from 0).

Restart application :

```
$ forever restart 0
```

Here 0 is the id of the server.

Stop application :

```
$ forever stop 0
```

Similar to restart, 0 is the id the server. You can also give process id or script name in place of the id given by the forever.

For more commands : <https://www.npmjs.com/package/forever>

Section 23.4: Deployment using PM2

PM2 is a production process manager for Node.js applications, that allows you to keep applications alive forever and reload them without downtime. PM2 also enables you to manage application logging, monitoring, and clustering.

Install pm2 globally.

```
npm install -g pm2
```

Then, run the node.js app using PM2.

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app
[PM2] restartProcessId process id 0
```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	0	fork	64029	online	1	0s	17.816 MB	disabled

```
Use the `pm2 show <id|name>` command to get more details about an app.
```

Following commands are useful while working with PM2.

List all running processes:

```
pm2 list
```

Stop an app:

```
pm2 stop my-app
```


Restart an app:

```
pm2 restart my-app
```

To view detailed information about an app:

```
pm2 show my-app
```

To remove an app from PM2's registry:

```
pm2 delete my-app
```

Section 23.5: Using different Properties/Configuration for different environments like dev, qa, staging etc

Large scale applications often need different properties when running on different environments. we can achieve this by passing arguments to NodeJs application and using same argument in node process to load specific environment property file.

Suppose we have two property files for different environment.

- dev.json

```
{
  "PORT": 3000,
  "DB": {
    "host": "localhost",
    "user": "bob",
    "password": "12345"
  }
}
```

- qa.json

```
{
  "PORT": 3001,
  "DB": {
    "host": "where_db_is_hosted",
    "user": "bob",
    "password": "54321"
  }
}
```

Following code in application will export respective property file which we want to use.

```
process.argv.forEach(function (val) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./' + arg[1] + '.json');
      exports.prop = env;
    }
  }
});
```

```
}  
});
```

We give arguments to the application like following

```
node app.js env=dev
```

if we are using process manager like *forever* than it as simple as

```
forever start app.js env=dev
```

Section 23.6: Taking advantage of clusters

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load.

```
var cluster = require('cluster');  
  
var numCPUs = require('os').cpus().length;  
  
if (cluster.isMaster) {  
    // In real life, you'd probably use more than just 2 workers,  
    // and perhaps not put the master and worker in the same file.  
    //  
    // You can also of course get a bit fancier about logging, and  
    // implement whatever custom logic you need to prevent DoS  
    // attacks and other bad behavior.  
    //  
    // See the options in the cluster documentation.  
    //  
    // The important thing is that the master does very little,  
    // increasing our resilience to unexpected errors.  
    console.log('your server is working on ' + numCPUs + ' cores');  
  
    for (var i = 0; i < numCPUs; i++) {  
        cluster.fork();  
    }  
  
    cluster.on('disconnect', function(worker) {  
        console.error('disconnect!');  
        //clearTimeout(timeout);  
        cluster.fork();  
    });  
  
} else {  
    require('./app.js');  
}
```

Chapter 24: Securing Node.js applications

Section 24.1: SSL/TLS in Node.js

If you choose to handle SSL/TLS in your Node.js application, consider that you are also responsible for maintaining SSL/TLS attack prevention at this point. In many server-client architectures, SSL/TLS terminates on a reverse proxy, both to reduce application complexity and reduce the scope of security configuration.

If your Node.js application should handle SSL/TLS, it can be secured by loading the key and cert files.

If your certificate provider requires a certificate authority (CA) chain, it can be added in the `ca` option as an array. A chain with multiple entries in a single file must be split into multiple files and entered in the same order into the array as Node.js does not currently support multiple `ca` entries in one file. An example is provided in the code below for files `1_ca.crt` and `2_ca.crt`. If the `ca` array is required and not set properly, client browsers may display messages that they could not verify the authenticity of the certificate.

Example

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
  ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Section 24.2: Preventing Cross Site Request Forgery (CSRF)

CSRF is an attack which forces end user to execute unwanted actions on a web application in which he/she is currently authenticated.

It can happen because cookies are sent with every request to a website - even when those requests come from a different site.

We can use `csrf` module for creating csrf token and validating it.

Example

```
var express = require('express')
var cookieParser = require('cookie-parser')    //for cookie parsing
var csrf = require('csrf')    //csrf module
var bodyParser = require('body-parser')    //for body parsing

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()
```

```
// parse cookies
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // generate and pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('data is being processed')
})
```

So, when we access GET /form, it will pass the csrf token csrfToken to the view.

Now, inside the view, set the csrfToken value as the value of a hidden input field named _csrf.

e.g. for handlebar templates

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

e.g. for jade templates

```
form(action="/process" method="post")
  input(type="hidden", name="_csrf", value=csrfToken)

  span Name:
    input(type="text", name="name", required=true)
  br

  input(type="submit")
```

e.g. for ejs templates

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

Section 24.3: Setting up an HTTPS server

Once you have node.js installed on your system, just follow the procedure below to get a basic web server running with support for both HTTP and HTTPS!

Step 1 : Build a Certificate Authority

1. create the folder where you want to store your key & certificate :

```
mkdir conf
```

2. go to that directory :

```
cd conf
```

3. grab this ca.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. create a new certificate authority using this configuration :

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. now that we have our certificate authority in ca-key.pem and ca-cert.pem, let's generate a private key for the server :

```
openssl genrsa -out key.pem 4096
```

6. grab this server.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generate the certificate signing request using this configuration :

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. sign the request :

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Step 2 : Install your certificate as a root certificate

1. copy your certificate to your root certificates' folder :

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. update CA store :

```
sudo update-ca-certificates
```

Section 24.4: Using HTTPS

The minimal setup for an HTTPS server in Node.js would be something like this :

```
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
```

```

    cert: fs.readFileSync('path/to/server-crt.pem')
  });

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

  https.createServer(httpsOptions, app).listen(4433);

```

If you also want to support http requests, you need to make just this small modification:

```

const http = require('http');
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

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

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);

```

Section 24.5: Secure express.js 3 Application

The configuration to make a secure connection using express.js (Since version 3):

```

var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');

// Define your key and cert

var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// your express configuration here

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// Using port 8080 for http and 8443 for https

httpServer.listen(8080);
httpsServer.listen(8443);

```

In that way you provide express middleware to the native http/https server

If you want your app running on ports below 1024, you will need to use sudo command (not recommended) or use a reverse proxy (e.g. nginx, haproxy).

Chapter 25: Mongoose Library

Section 25.1: Connect to MongoDB Using Mongoose

First, install Mongoose with:

```
npm install mongoose
```

Then, add it to `server.js` as dependencies:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Next, create the database schema and the name of the collection:

```
var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});
```

Create a model and connect to the database:

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

Next, start MongoDB and run `server.js` using `node server.js`

To check if we have successfully connected to the database, we can use the events `open`, `error` from the `mongoose.connection` object.

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

Section 25.2: Find Data in MongoDB Using Mongoose, Express.js Routes and \$text Operator

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to `server.js`, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
```

```

var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

```

Now add Express.js routes that we will use to query the data:

```

app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      })))
    }
  })
})

```

Assume that the following documents are in the collection in the model:

```

{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}

```

And that the goal is to find and display all the documents containing only "JavaScript" word under the "request" key.

To do this, first create a *text index* for "request" in the collection. For this, add the following code to `server.js`:


```
schemaName.index({ request: 'text' });
```

And replace:

```
Model.find({
  'request': query
}, function(err, result) {
```

With:

```
Model.find({
  $text: {
    $search: query
  }
}, function(err, result) {
```

Here, we are using \$text and \$search MongoDB operators for find all documents in collection `collectionName` which contains at least one word from the specified find query.

Usage

To use this to find data, go to the following URL in a browser:

```
http://localhost:8080/find/<query>
```

Where **<query>** is the search query.

Example:

```
http://localhost:8080/find/JavaScript
```

Output:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Section 25.3: Save Data to MongoDB using Mongoose and Express.js Routes

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to your `server.js` file, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Now add Express.js routes that we will use to write the data:

```
app.get('/save/:query', cors(), function(req, res) {
  var query = req.params.query;

  var savedata = new Model({
    'request': query,
    'time': Math.floor(Date.now() / 1000) // Time of save the data in unix timestamp format
  }).save(function(err, result) {
    if (err) throw err;

    if(result) {
      res.json(result)
    }
  })
});
```

Here the query variable will be the `<query>` parameter from the incoming HTTP request, which will be saved to MongoDB:

```
var savedata = new Model({
  'request': query,
  //...
```

If an error occurs while trying to write to MongoDB, you will receive an error message on the console. If all is successful, you will see the saved data in JSON format on the page.

```
//...
```

```

}).save(function(err, result) {
    if (err) throw err;

    if(result) {
        res.json(result)
    }
})
//...

```

Now, you need to start MongoDB and run your server .js file using node server.js.

Usage

To use this to save data, go to the following URL in your browser:

```
http://localhost:8080/save/<query>
```

Where <query> is the new request you wish to save.

Example:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Output in JSON format:

```

{
  __v: 0,
  request: "JavaScript is Awesome",
  time: 1469411348,
  _id: "57957014b93bc8640f2c78c4"
}

```

Section 25.4: Find Data in MongoDB Using Mongoose and Express.js Routes

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to server.js, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```

var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
}

```

```
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Now add Express.js routes that we will use to query the data:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      })))
    }
  })
})
```

Assume that the following documents are in the collection in the model:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

And the goal is to find and display all the documents containing "JavaScript is Awesome" under the "request" key.

For this, start MongoDB and run `server.js` with `node server.js`:

Usage

To use this to find data, go to the following URL in a browser:

```
http://localhost:8080/find/<query>
```

Where `<query>` is the search query.

Example:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Output:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Section 25.5: Useful Mongoose functions

Mongoose contains some built in functions that build on the standard `find()`.

```
doc.find({'some.value':5}, function(err, docs){
  //returns array docs
});

doc.findOne({'some.value':5}, function(err, doc){
  //returns document doc
});

doc.findById(obj._id, function(err, doc){
  //returns document doc
});
```

Section 25.6: Indexes in models

MongoDB supports secondary indexes. In Mongoose, we define these indexes within our schema. Defining indexes at schema level is necessary when we need to create compound indexes.

Mongoose Connection

```
var strConnection = 'mongodb://localhost:27017/dbName';
var db = mongoose.createConnection(strConnection)
```

Creating a basic schema

```
var Schema = require('mongoose').Schema;
var usersSchema = new Schema({
  username: {
    type: String,
```

```

        required: true,
        unique: true
    },
    email: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },
    created: {
        type: Date,
        default: Date.now
    }
});

var userModel = db.model('users', usersSchema);
module.exports = userModel;

```

By default, mongoose adds two new fields into our model, even when those are not defined in the model. Those fields are:

`_id`

Mongoose assigns each of your schemas an `_id` field by default if one is not passed into the Schema constructor. The type assigned is an ObjectId to coincide with MongoDB's default behavior. If you don't want an `_id` added to your schema at all, you may disable it using this option.

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, {
    _id: false
  });

```

`__v` or `versionKey`

The `versionKey` is a property set on each document when first created by Mongoose. This key's value contains the internal revision of the document. The name of this document property is configurable.

You can easily disable this field in the model configuration:

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, {
    versionKey: false
  });

```

Compound indexes

We can create another indexes besides those Mongoose creates.

```

usersSchema.index({username: 1 });

```

```
usersSchema.index({email: 1 });
```

In these case our model have two more indexes, one for the field username and another for email field. But we can create compound indexes.

```
usersSchema.index({username: 1, email: 1 });
```

Index performance impact

By default, mongoose always call the ensureIndex for each index sequentially and emit an 'index' event on the model when all the ensureIndex calls succeeded or when there was an error.

In MongoDB ensureIndex is deprecated since 3.0.0 version, now is an alias for createIndex.

Is recommended disable the behavior by setting the autoIndex option of your schema to false, or globally on the connection by setting the option config.autoIndex to false.

```
usersSchema.set('autoIndex', false);
```

Section 25.7: find data in mongodb using promises

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to server.js, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

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

```
app.use(function(req, res, next) {
  res.status(404).send('Sorry cant find that!');
});
```

Now add Express.js routes that we will use to query the data:

```
app.get('/find/:query', cors(), function(req, res, next) {
  var query = req.params.query;

  Model.find({
    'request': query
  })
  .exec() //remember to add exec, queries have a .then attribute but aren't promises
  .then(function(result) {
    if (result) {
      res.json(result)
    } else {
      next() //pass to 404 handler
    }
  })
  .catch(next) //pass to error handler
})
```

Assume that the following documents are in the collection in the model:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

And the goal is to find and display all the documents containing "JavaScript is Awesome" under the "request" key.

For this, start MongoDB and run `server.js` with `node server.js`:

Usage

To use this to find data, go to the following URL in a browser:

```
http://localhost:8080/find/<query>
```

Where `<query>` is the search query.

Example:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```


Output:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Chapter 26: async.js

Section 26.1: Parallel : multi-tasking

`async.parallel(tasks, afterTasksCallback)` will execute a set of tasks in parallel and **wait the end of all tasks** (reported by the call of **callback** function).

When tasks are finished, *async* call the main callback with all errors and all results of tasks.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}

async.parallel([
  shortTimeFunction,
  mediumTimeFunction,
  longTimeFunction
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Result : `["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"]`.

Call `async.parallel()` with an object

You can replace the *tasks* array parameter by an object. In this case, results will be also an object **with the same keys than tasks**.

It's very useful to compute some tasks and find easily each result.

```
async.parallel({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }
}
```

```
console.log(results);
});
```

Result : {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}.

Resolving multiple values

Each parallel function is passed a callback. This callback can either return an error as the first argument or success values after that. If a callback is passed several success values, these results are returned as an array.

```
async.parallel({
  short: function shortTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
    }, 200);
  },
  medium: function mediumTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
    }, 500);
  }
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Result :

```
{
  short: ["resultOfShortTime1", "resultOfShortTime2"],
  medium: ["resultOfMediumTime1", "resultOfMediumTime2"]
}
```

Section 26.2: async.each(To handle array of data efficiently)

When we want to handle array of data, its better to use **async.each**. When we want to perform something with all data & want to get the final callback once everything is done, then this method will be useful. This is handled in parallel way.

```
function createUser(userName, callback)
{
  //create user in db
  callback(null)//or error based on creation
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

  // Perform operation on each user.
  console.log('Creating user '+eachUserName);
  //Returning callback is must. Else it wont get the final callback, even if we miss to return one
  callback
```

```

    createUser(eachUserName, callback);
}, function(err) {
    //If any of the user creation failed may throw error.
    if( err ) {
        // One of the iterations produced an error.
        // All processing will now stop.
        console.log('unable to create user');
    } else {
        console.log('All user created successfully');
    }
});

```

To do one at a time can use **async.eachSeries**

Section 26.3: Series : independent mono-tasking

[*async.series\(tasks, afterTasksCallback\)*](#) will execute a set of tasks. Each task are executed **after another**. If a task fails, **async stops immediately the execution and jump into the main callback**.

When tasks are finished successfully, *async* call the "master" callback with all errors and all results of tasks.

```

function shortTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfShortTime');
    }, 200);
}

function mediumTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfMediumTime');
    }, 500);
}

function longTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfLongTime');
    }, 1000);
}

async.series([
    mediumTimeFunction,
    shortTimeFunction,
    longTimeFunction
],
function(err, results) {
    if (err) {
        return console.error(err);
    }

    console.log(results);
});

```

Result : `["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"]`.

Call `async.series()` with an object

You can replace the *tasks* array parameter by an object. In this case, results will be also an object **with the same keys than tasks**.

It's very useful to compute some tasks and find easily each result.

```
async.series({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Result : {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}.

Section 26.4: Waterfall : dependent mono-tasking

[*async.waterfall\(tasks, afterTasksCallback\)*](#) will execute a set of tasks. Each task are executed **after another, and the result of a task is passed to the next task**. As *async.series()*, if a task fails, *async* stop the execution and call immediately the main callback.

When tasks are finished successfully, *async* call the "master" callback with all errors and all results of tasks.

```
function getUserRequest(callback) {
  // We simulate the request with a timeout
  setTimeout(function() {
    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
  }, 500);
}

function getUserFriendsRequest(user, callback) {
  // Another request simulate with a timeout
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
      friendsResult = [{
        name : 'Alice'
      }, {
        name: 'Bob'
      }];
    }

    callback(null, friendsResult);
  }, 500);
}

async.waterfall([
  getUserRequest,
  getUserFriendsRequest
],
function(err, results) {
  if (err) {
    return console.error(err);
  }
});
```

```

    }

    console.log(JSON.stringify(results));
  });

```

Result: results contains the second callback parameter of the last function of the waterfall, which is friendsResult in that case.

Section 26.5: async.times(To handle for loop in better way)

To execute a function within a loop in node.js, it's fine to use a **for** loop for short loops. But the loop is long, using **for** loop will increase the time of processing which might cause the node process to hang. In such scenarios, you can use: **async.times**

```

function recursiveAction(n, callback)
{
    //do whatever want to do repeatedly
    callback(err, result);
}
async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // we should now have 5 result
});

```

This is called in parallel. When we want to call it one at a time, use: **async.timesSeries**

Section 26.6: async.series(To handle events one by one)

In async.series, all the functions are executed in series and the consolidated outputs of each function is passed to the final callback. e.g/

```

var async = require('async');
async.series([ function (callback) { console.log('First Execute..'); callback(null, 'userPersonalData'); },
function (callback) { console.log('Second Execute.. '); callback(null, 'userDependentData'); } ],
function (err, result) { console.log(result); });

```

//Output:

First Execute.. Second Execute.. ['userPersonalData','userDependentData'] //result

Chapter 27: File upload

Section 27.1: Single File Upload using multer

Remember to

- create folder for upload (uploads in example).
- install multer `npm i -S multer`

server.js:

```
var express = require("express");
var multer  = require('multer');
var app     = express();
var fs = require('fs');

app.get('/', function(req, res){
  res.sendFile(__dirname + "/index.html");
});

var storage = multer.diskStorage({
  destination: function (req, file, callback) {
    fs.mkdir('./uploads', function(err) {
      if(err) {
        console.log(err.stack)
      } else {
        callback(null, './uploads');
      }
    })
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

app.post('/api/file', function(req, res){
  var upload = multer({ storage : storage}).single('userFile');
  upload(req, res, function(err) {
    if(err) {
      return res.end("Error uploading file.");
    }
    res.end("File is uploaded");
  });
});

app.listen(3000, function(){
  console.log("Working on port 3000");
});
```

index.html:

```
<form id      = "uploadForm"
  enctype     = "multipart/form-data"
  action      = "/api/file"
  method      = "post"
>
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
```

```
</form>
```

Note:

To upload file with extension you can use Node.js [path](#) built-in library

For that just require path to server.js file:

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

and change:

```
callback(null, file.fieldname + '-' + Date.now());
```

adding a file extension in the following way:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

How to filter upload by extension:

In this example, view how to upload files to allow only certain extensions.

For example only images extensions. Just add to `var upload = multer({ storage : storage }).single('userFile');` fileFilter condition

```
var upload = multer({
  storage: storage,
  fileFilter: function (req, file, callback) {
    var ext = path.extname(file.originalname);
    if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
      return callback(new Error('Only images are allowed'))
    }
    callback(null, true)
  }
}).single('userFile');
```

Now you can upload only image files with png, jpg, gif or jpeg extensions

Section 27.2: Using formidable module

Install module and read [docs](#)

```
npm i formidable@latest
```

Example of server on 8080 port

```
var formidable = require('formidable'),
    http = require('http'),
    util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
        do-smth; // process error
    });
  }
});
```



```

    // Copy file from temporary place
    // var fs = require('fs');
    // fs.rename(file.path, <targetPath>, function (err) { ... });

    // Send result on client
    res.writeHead(200, {'content-type': 'text/plain'});
    res.write('received upload:\n\n');
    res.end(util.inspect({fields: fields, files: files}));
  });

  return;
}

// show a file upload form
res.writeHead(200, {'content-type': 'text/html'});
res.end(
  '<form action="/upload" enctype="multipart/form-data" method="post">'+
  '<input type="text" name="title"><br>'+
  '<input type="file" name="upload" multiple="multiple"><br>'+
  '<input type="submit" value="Upload">'+
  '</form>'
);
}).listen(8080);

```

Chapter 28: Socket.io communication

Section 28.1: "Hello world!" with socket messages

Install node modules

```
npm install express
npm install socket.io
```

Node.js server

```
const express = require('express');
const app = express();
const server = app.listen(3000, console.log("Socket.io Hello World server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  //console.log("Client connected!");
  socket.on('message-from-client-to-server', (msg) => {
    console.log(msg);
  })
  socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Browser client

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
  </head>
  <body>
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
      var socket = io("http://localhost:3000");
      socket.on("message-from-server-to-client", function(msg) {
        document.getElementById('message').innerHTML = msg;
      });
      socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
  </body>
</html>
```

Chapter 29: Mongodb integration

Parameter	Details
document	A javascript object representing a document
documents	An array of documents
query	An object defining a search query
filter	An object defining a search query
callback	Function to be called when the operation is done
options	(optional) Optional settings (default: null)
w	(optional) The write concern
wtimeout	(optional) The write concern timeout. (default: null)
j	(optional) Specify a journal write concern (default: false)
upsert	(optional) Update operation (default: false)
multi	(optional) Update one/all documents (default: false)
serializeFunctions	(optional) Serialize functions on any object (default: false)
forceServerObjectId	(optional) Force server to assign _id values instead of driver (default: false)
bypassDocumentValidation	(optional) Allow driver to bypass schema validation in MongoDB 3.2 or higher (default: false)

Section 29.1: Simple connect

MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) { if(error) return console.log(error); const collection = database.collection('collectionName'); collection.insert({key: 'value'}, function(error, result) { console.log(error, result); }); });

Section 29.2: Simple connect, using promises

```
const MongoDB = require('mongodb');

MongoDB.connect('mongodb://localhost:27017/databaseName')
  .then(function(database) {
    const collection = database.collection('collectionName');
    return collection.insert({key: 'value'});
  })
  .then(function(result) {
    console.log(result);
  });
// ...
```

Section 29.3: Connect to MongoDB

Connect to MongoDB, print 'Connected!' and close the connection.

```
const MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'
  if (err) throw new Error(err);
  console.log("Connected!");
  db.close(); // Don't forget to close the connection when you are done
});
```

MongoClient method Connect()

```
MongoClient.connect(url, options, callback)
```

Argument	Type	Description
url	string	A string specifying the server ip/hostname, port and database
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the connection attempt is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

Section 29.4: Insert a document

Insert a document called 'myFirstDocument' and set 2 properties, greetings and farewell

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hellu",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Collection method insertOne()

```
db.collection(collection).insertOne(document, options, callback)
```

Argument	Type	Description
collection	string	A string specifying the collection
document	object	The document to be inserted into the collection
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the insert operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- result : object - An object containing details about the insert operation

Section 29.5: Read a collection

Get all documents in the collection 'myCollection' and print them to the console.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';
```

```

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {
      db.close(); // Don't forget to close the connection when you are done
    }
  });
});

```

Collection method find()

```

db.collection(collection).find()

```

Argument Type	Description
collection	string A string specifying the collection

Section 29.6: Update a document

Find a document with the property { greetings: 'Hellu' } and change it to { greetings: 'Whut?' }

```

const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').updateOne({ // Update method 'updateOne'
    greetings: "Hellu" },
    { $set: { greetings: "Whut?" } },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    }
  });
});

```

Collection method updateOne()

```

db.collection(collection).updateOne(filter, update, options, callback)

```

Parameter	Type	Description
filter	object	Specifies the selection criteria
update	object	Specifies the modifications to apply
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

Section 29.7: Delete a document

Delete a document with the property `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteOne(// Delete method 'deleteOne'
    { greetings: "Whut?" },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

Collection method `deleteOne()`

```
db.collection(collection).deleteOne(filter, options, callback)
```

Parameter	Type	Description
filter	object	A document specifying the selection criteria
options	object	<i>(optional)</i> Optional settings <i>(default: null)</i>
callback	Function	Function to be called when the operation is done

The callback function takes two arguments

- `err` : Error - If an error occurs the `err` argument will be defined
- `db` : object - The MongoDB instance

Section 29.8: Delete multiple documents

Delete ALL documents with a 'farewell' property set to 'okay'.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteMany(// MongoDB delete method 'deleteMany'
    { farewell: "okay" }, // Delete ALL documents with the property 'farewell: okay'
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

Collection method `deleteMany()`

```
db.collection(collection).deleteMany(filter, options, callback)
```

Parameter	Type	Description
filter	document	A document specifying the selection criteria

`options` `object` *(optional)* Optional settings *(default: null)*
`callback` `function` Function to be called when the operation is done

The `callback` function takes two arguments

- `err` : Error - If an error occurs the `err` argument will be defined
- `db` : object - The MongoDB instance

Chapter 30: Handling POST request in Node.js

Section 30.1: Sample node.js server that just handles POST requests

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
    const responseString = `Received string ${buffer}`;
    console.log(`Responding with: ${responseString}`);
    response.writeHead(200, "Content-Type: text/plain");
    response.end(responseString);
  });
}).listen(PORT, () => {
  console.log(`Listening on ${PORT}`);
});
```


Chapter 31: Simple REST based CRUD API

Section 31.1: REST API for CRUD in Express 3+

```
var express = require("express"),
    bodyParser = require("body-parser"),
    server = express();

//body parser for parsing request body
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//temporary store for `item` in memory
var itemStore = [];

//GET all items
server.get('/item', function (req, res) {
  res.json(itemStore);
});

//GET the item with specified id
server.get('/item/:id', function (req, res) {
  res.json(itemStore[req.params.id]);
});

//POST new item
server.post('/item', function (req, res) {
  itemStore.push(req.body);
  res.json(req.body);
});

//PUT edited item in-place of item with specified id
server.put('/item/:id', function (req, res) {
  itemStore[req.params.id] = req.body;
  res.json(req.body);
});

//DELETE item with specified id
server.delete('/item/:id', function (req, res) {
  itemStore.splice(req.params.id, 1);
  res.json(req.body);
});

//START SERVER
server.listen(3000, function () {
  console.log("Server running");
});
```

Chapter 32: Template frameworks

Section 32.1: Nunjucks

Server-side engine with block inheritance, autoescaping, macros, asynchronous control, and more. Heavily inspired by jinja2, very similar to Twig (php).

Docs - <http://mozilla.github.io/nunjucks/>

Install - `npm i nunjucks`

Basic usage with [Express](#) below.

app.js

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

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views/'], { // set folders with templates
  autoescape: true,
  express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
  console.log('myFilter', obj, arg1, arg2);
  // Do smth with obj
  return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
  console.log('myFunc', obj, arg1);
  // Do smth with obj
  return obj;
});

app.get('/', function(req, res){
  res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
  res.locals.smthVar = 'This is Sparta!';
  res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
  console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
  <title>Nunjucks example</title>
</head>
<body>
{% block content %}
{{title}}
```

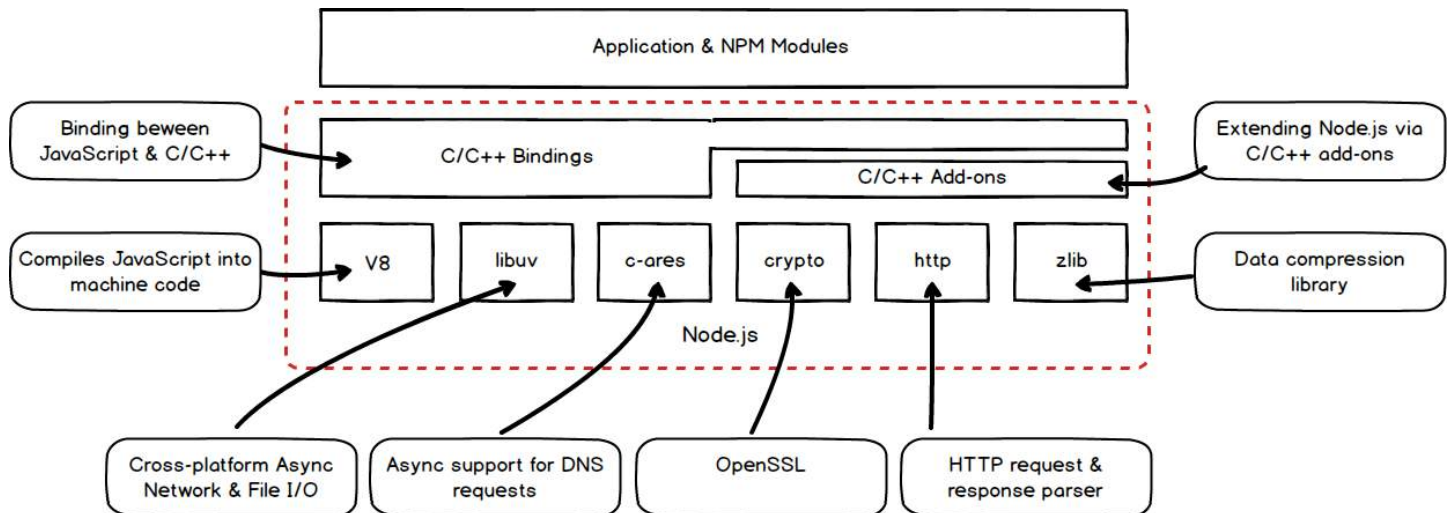
```
{% endblock %}  
</body>  
</html>
```

/views/foo.html

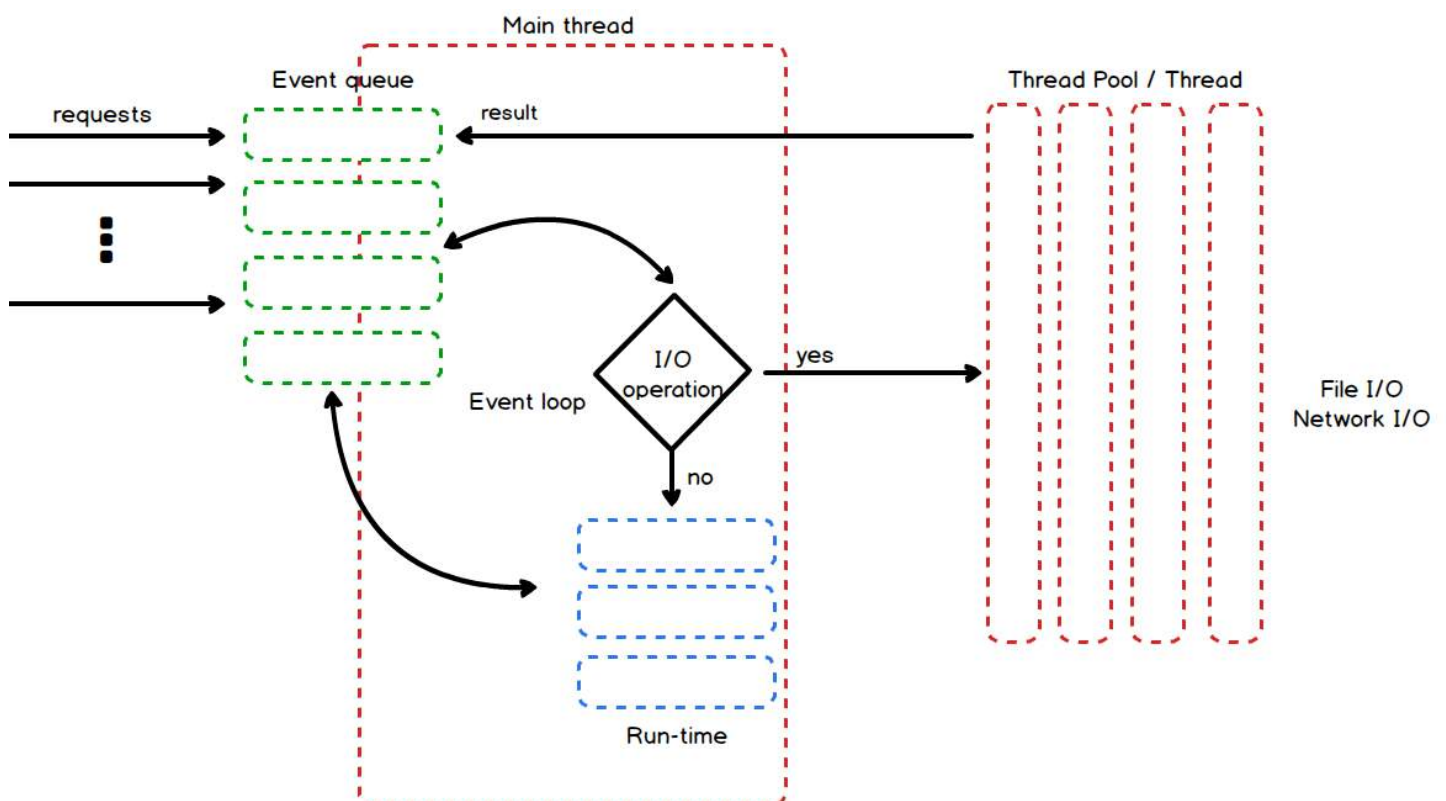
```
{% extends "index.html" %}  
  
{# This is comment #}  
{% block content %}  
  <h1>{{title}}</h1>  
  {# apply custom function and next build-in and custom filters #}  
  {{ myFunc(smithVar) | lower | myFilter(5, 'abc') }}  
{% endblock %}
```

Chapter 33: Node.js Architecture & Inner Workings

Section 33.1: Node.js - under the hood



Section 33.2: Node.js - in motion



Chapter 34: Debugging Node.js application

Section 34.1: Core node.js debugger and node inspector

Using core debugger

Node.js provides a built-in non-graphical debugging utility. To start the built-in debugger, start the application with this command:

```
node debug filename.js
```

Consider the following simple Node.js application contained in the `debugDemo.js`

```
'use strict';

function addTwoNumber(a, b){
  // function returns the sum of the two numbers
  debugger
  return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

The keyword `debugger` will stop the debugger at that point in the code.

Command reference

1. Stepping

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

2. Breakpoints

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

To debug the above code run the following command

```
node debug debugDemo.js
```

Once the above commands run you will see the following output. To exit from the debugger interface, type `process.exit()`

```

ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $

```

Use `watch(expression)` command to add the variable or expression whose value you want to watch and restart to restart the app and debugging.

Use `repl` to enter code interactively. The `repl` mode has the same context as the line you are debugging. This allows you to examine the contents of variables and test out lines of code. Press `Ctrl+C` to leave the debug `repl`.

Using Built-in Node inspector

Version ≥ v6.3.0

You can run node's [built in](#) v8 inspector! The [node-inspector](#) plug-in is not needed anymore.

Simply pass the inspector flag and you'll be provided with a URL to the inspector

```
node --inspect server.js
```

Using Node inspector

Install the node inspector:

```
npm install -g node-inspector
```

Run your app with the node-debug command:

```
node-debug filename.js
```

After that, hit in Chrome:

```
http://localhost:8080/debug?port=5858
```

Sometimes port 8080 might not be available on your computer. You may get the following error:

Cannot start the server at 0.0.0.0:8080. Error: listen EACCES.

In this case, start the node inspector on a different port using the following command.

```
$node-inspector --web-port=6500
```

You will see something like this:

The screenshot shows the Node.js Inspector interface. On the left, a code editor displays a file named `debugDemo.js` with the following content:

```
1 // A Demo Code Showing the basic capabilities of the nodejs debugging module
2
3 'use strict';
4
5 function addTwoNumber(a, b){
6   // function returns the sum of the two numbers
7   return a + b;
8 }
9
10 var result = addTwoNumber(5, 9);
11 console.log(result);
12
```

Line 10 is highlighted in yellow, indicating the current execution point. On the right, the 'Call Stack' panel is expanded, showing the following stack:

Function	File	Ln	Col
anonymous(exports, require, module, ...)	nodejs/nodejsDebuggin...	10	1
Module._compile(content, filename)	module.js	409	26
Module._extensions.js(module, filena...	module.js	416	10
Module.load(filename)	module.js	343	32
Module._load(request, parent, isMain)	module.js	300	12
Module.runMain()	module.js	441	10
listOnTimeout()	timers.js	92	15

Below the call stack, the 'Local Variables' panel is also expanded, showing the following variables:

Variable	Value	Type
__dirname	"/home/ubuntu/workspace/nodejs/nod..."	string
__filename	"/home/ubuntu/workspace/nodejs/nod..."	string
addTwoNumber	function ()	function
exports	[Object]	object
module	[Object]	object
require	function ()	function

The status bar at the bottom indicates '10:1 JavaScript Spaces: 4'.

Chapter 35: Node server without framework

Section 35.1: Framework-less node server

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

http.createServer(function (request, response) {
  console.log('request ', request.url);

  var filePath = '.' + request.url;
  if (filePath == './')
    filePath = './index.html';

  var extname = String(path.extname(filePath)).toLowerCase();
  var contentType = 'text/html';
  var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpeg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'application/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
  };

  contentType = mimeTypes[extname] || 'application/octet-stream';

  fs.readFile(filePath, function(error, content) {
    if (error) {
      if(error.code == 'ENOENT'){
        fs.readFile('./404.html', function(error, content) {
          response.writeHead(200, { 'Content-Type': contentType });
          response.end(content, 'utf-8');
        });
      }
      else {
        response.writeHead(500);
        response.end('Sorry, check with the site admin for error: '+error.code+ ' ..\n');
        response.end();
      }
    }
    else {
      response.writeHead(200, { 'Content-Type': contentType });
      response.end(content, 'utf-8');
    }
  });
}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```


Section 35.2: Overcoming CORS Issues

```
// Website you wish to allow to connect to
response.setHeader('Access-Control-Allow-Origin', '*');

// Request methods you wish to allow
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// Request headers you wish to allow
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent
// to the API (e.g. in case you use sessions)
response.setHeader('Access-Control-Allow-Credentials', true);
```

Chapter 36: Node.JS with ES6

ES6, ECMAScript 6 or ES2015 is the latest [specification](#) for JavaScript which introduces some syntactic sugar to the language. It's a big update to the language and introduces a lot of new [features](#)

More details on Node and ES6 can be found on their site <https://nodejs.org/en/docs/es6/>

Section 36.1: Node ES6 Support and creating a project with Babel

The whole ES6 spec is not yet implemented in its entirety so you will only be able to use some of the new features. You can see a list of the current supported ES6 features at <http://node.green/>

Since NodeJS v6 there has been pretty good support. So if you using NodeJS v6 or above you can enjoy using ES6. However, you may also want to use some of the unreleased features and some from beyond. For this you will need to use a transpiler

It is possible to run a transpiler at run time and build, to use all of the ES6 features and more. The most popular transpiler for JavaScript is called [Babel](#)

Babel allows you to use all of the features from the ES6 specification and some additional not-in-spec features with 'stage-0' such as `import` thing from 'thing' instead of `var` thing = require('thing')

If we wanted to create a project where we use 'stage-0' features such as import we would need to add Babel as a transpiler. You'll see projects using react and Vue and other commonJS based patterns implement stage-0 quite often.

create a new node project

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Install babel the ES6 preset and stage-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Create a new file called `server.js` and add a basic HTTP server.

```
import http from 'http'

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'})
  res.end('Hello World\n')
}).listen(3000, '127.0.0.1')

console.log('Server running at http://127.0.0.1:3000/')
```

Note that we use an `import` http from 'http' this is a stage-0 feature and if it works it means we've got the transpiler working correctly.

If you run `node server.js` it will fail not knowing how to handle the import.

Creating a `.babelrc` file in the root of your directory and add the following settings

```
{
  "presets": ["es2015", "stage-2"],
  "plugins": []
}
```

you can now run the server with `node src/index.js --exec babel-node`

Finishing off it is not a good idea to run a transpiler at runtime on a production app. We can however implement some scripts in our package.json to make it easier to work with.

```
"scripts": {
  "start": "node dist/index.js",
  "dev": "babel-node src/index.js",
  "build": "babel src -d dist",
  "postinstall": "npm run build"
},
```

The above will on `npm install` build the transpiled code to the dist directory allow `npm start` to use the transpiled code for our production app.

`npm run dev` will boot the server and babel runtime which is fine and preferred when working on a project locally.

Going one further you could then install nodemon `npm install nodemon --save-dev` to watch for changes and then reboot the node app.

This really speeds up working with babel and NodeJS. In your package.json just update the "dev" script to use nodemon

```
"dev": "nodemon src/index.js --exec babel-node",
```

Section 36.2: Use JS es6 on your NodeJS app

JS es6 (also known as es2015) is a set of new features to JS language aim to make it more intuitive when using OOP or while facing modern development tasks.

Prerequisites:

1. Check out the new es6 features at <http://es6-features.org> - it may clarify to you if you really intend to use it on your next NodeJS app
2. Check the compatibility level of your node version at <http://node.green>
3. If all is ok - let's code on!

Here is a very short sample of a simple hello world app with JS es6

```
'use strict'

class Program
{
  constructor()
  {
    this.message = 'hello es6 :)';
  }
}
```

```

    print()
    {
        setTimeout(() =>
        {
            console.log(this.message);

            this.print();

        }, Math.random() * 1000);
    }
}

new Program().print();

```

You can run this program and observe how it print the same message over and over again.

Now.. let break it down line by line:

```
'use strict'
```

This line is actually required if you intend to use js es6. `strict` mode, intentionally, has different semantics from normal code (please read more about it on MDN -

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

```
class Program
```

Unbelievable - a `class` keyword! Just for a quick reference - before es6 the only way do define a class in js was with the... `function` keyword!

```

function MyClass() // class definition
{
}

var myClassObject = new MyClass(); // generating a new object with a type of MyClass

```

When using OOP, a class is a very fundamental ability which assist the developer to represent a specific part of a system (breaking down code is crucial when the code is getting larger.. for instance: when writing server-side code)

```

constructor()
{
    this.message = 'hello es6 :)';
}

```

You got to admit - this is pretty intuitive! This is the c'tor of my class - this unique "function" will occur every time an object is created from this particular class (in our program - only once)

```

print()
{
    setTimeout(() => // this is an 'arrow' function
    {
        console.log(this.message);

        this.print(); // here we call the 'print' method from the class template itself (a recursion in this particular case)

    }, Math.random() * 1000);
}

```

```
}
```

Because print is defined in the class scope - it is actually a method - which can be invoked from either the object of the class or from within the class itself!

So.. till now we defined our class.. time to use it:

```
new Program().print();
```

Which is truly equals to:

```
var prog = new Program(); // define a new object of type 'Program'  
prog.print(); // use the program to print itself
```

In conclusion: JS es6 can simplify your code - make it more intuitive and easy to understand (comparing with the previous version of JS).. you may try to re-write an existing code of yours and see the difference for yourself

ENJOY :)

Chapter 37: Interacting with Console

Section 37.1: Logging

Console Module

Similar to the browser environment of JavaScript node.js provides a **console** module which provides simple logging and debugging possibilities.

The most important methods provided by the console module are `console.log`, `console.error` and `console.time`. But there are several others like `console.info`.

`console.log`

The parameters will be printed to the standard output (stdout) with a new line.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

`console.error`

The parameters will be printed to the standard error (stderr) with a new line.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

`console.time`, `console.timeEnd`

`console.time` starts a timer with an unique label that can be used to compute the duration of an operation. When you call `console.timeEnd` with the same label, the timer stops and it prints the elapsed time in milliseconds to stdout.

```
> console.time("label");
undefined
> console.timeEnd("label");
label: 9297.320ms
```

Process Module

It is possible to use the **process** module to write **directly** into the standard output of the console. Therefore it exists the method `process.stdout.write`. Unlike `console.log` this method does not add a new line before your output.

So in the following example the method is called two times, but no new line is added in between their outputs.

```
> process.stdout.write("123");process.stdout.write("456");
123456true
```

Formatting

One can use **terminal (control) codes** to issue specific commands like switching colors or positioning the cursor.

```
> console.log("\033[31mThis will be red");
This will be red
```

General

Effect	Code
Reset	\033[0m
Hicolor	\033[1m
Underline	\033[4m
Inverse	\033[7m

Font Colors

Effect	Code
Black	\033[30m
Red	\033[31m
Green	\033[32m
Yellow	\033[33m
Blue	\033[34m
Magenta	\033[35m
Cyan	\033[36m
White	\033[37m

Background Colors

Effect	Code
Black	\033[40m
Red	\033[41m
Green	\033[42m
Yellow	\033[43m
Blue	\033[44m
Magenta	\033[45m
Cyan	\033[46m
White	\033[47m

Chapter 38: Cassandra Integration

Section 38.1: Hello world

For accessing Cassandra [cassandra-driver](#) module from DataStax can be used. It supports all the features and can be easily configured.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
  contactPoints: ["host1", "host2"],
  keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
  if (err) {
    return console.error(err);
  }

  console.log(results.rows);
});
```


Chapter 39: Creating API's with Node.js

Section 39.1: GET api using Express

Node.js apis can be easily constructed in Express web framework.

Following example creates a simple GET api for listing all users.

Example

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

var users =[{
  id: 1,
  name: "John Doe",
  age : 23,
  email: "john@doe.com"
}];

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);    //return response as JSON
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

Section 39.2: POST api using Express

Following example create POST api using Express. This example is similar to GET example except the use of body-parser that parses the post data and add it to req.body.

Example

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users =[{
  id: 1,
  name: "John Doe",
  age : 23,
  email: "john@doe.com"
}];

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);
});

/* POST /api/users
```

```
{
  "user": {
    "id": 3,
    "name": "Test User",
    "age" : 20,
    "email": "test@test.com"
  }
}

*/
app.post('/api/users', function (req, res) {
  var user = req.body.user;
  users.push(user);

  return res.send('User has been added successfully');
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

Chapter 40: Graceful Shutdown

Section 40.1: Graceful Shutdown - SIGTERM

By using **server.close()** and **process.exit()**, we can catch the server exception and do a graceful shutdown.

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

var server = http.createServer(function (req, res) {
  setTimeout(function () { //simulate a long request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
    process.exit(0);
  });
});
```

Chapter 41: Using IISNode to host Node.js Web Apps in IIS

Section 41.1: Using an IIS Virtual Directory or Nested Application via <appSettings>

Using a Virtual Directory or Nested Application in IIS is a common scenario and most likely one that you'll want to take advantage of when using IISNode.

IISNode doesn't provide direct support for Virtual Directories or Nested Applications via configuration so to achieve this we'll need to take advantage of a feature of IISNode that isn't part of the configuration and is much lesser known. All children of the <appSettings> element with the Web.config are added to the process.env object as properties using the appSetting key.

Lets create a Virtual Directory in our <appSettings>

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

Within our Node.js App we can access the virtualDirPath setting

```
console.log(process.env.virtualDirPath); // prints /foo
```

Now that we can use the <appSettings> element for configuration, lets take advantage of that and use it in our server code.

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
  return res.status(200).send('Hello World');
});
```

We can use the virtualDirPath with our static resources as well

```
// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));
```

Lets put all of that together

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

const port = process.env.PORT || 3000;
```

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Section 41.2: Getting Started

[IISNode](#) allows Node.js Web Apps to be hosted on IIS 7/8 just like a .NET application would. Of course, you can self host your `node.exe` process on Windows but why do that when you can just run your app in IIS.

IISNode will handle scaling over multiple cores, process management of `node.exe`, and auto-recycle your IIS Application whenever your app is updated, just to name a few of its [benefits](#).

Requirements

IISNode does have a few requirements before you can host your Node.js app in IIS.

1. Node.js must be installed on the IIS host, 32-bit or 64-bit, either are supported.
2. IISNode installed [x86](#) or [x64](#), this should match the bitness of your IIS Host.
3. The [Microsoft URL-Rewrite Module for IIS](#) installed on your IIS host.
 - This is key, otherwise requests to your Node.js app won't function as expected.
4. A `Web.config` in the root folder of your Node.js app.
5. IISNode configuration via an `iisnode.yml` file or an `<iisnode>` element within your `Web.config`.

Section 41.3: Basic Hello World Example using Express

To get this example working, you'll need to create an IIS 7/8 app on your IIS host and add the directory containing the Node.js Web App as the Physical Directory. Ensure that your Application/Application Pool Identity can access the Node.js install. This example uses the Node.js 64-bit installation.

Project Structure

This is the basic project structure of a IISNode/Node.js Web app. It looks almost identical to any non-IISNode Web App except for the addition of the `Web.config`.

```
- /app_root
- package.json
- server.js
```

- Web.config

server.js - Express Application

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

// We need to get the port that IISNode passes into us
// using the PORT environment variable, if it isn't set use a default value
const port = process.env.PORT || 3000;

// Setup a route at the index of our app
server.get('/', (req, res) => {
  return res.status(200).send('Hello World');
});

server.listen(port, () => {
  console.log(`Listening on ${port}`);
});
```

Configuration & Web.config

The Web.config is just like any other IIS Web.config except the following two things must be present, URL `<rewrite><rules>` and an IISNode `<handler>`. Both of these elements are children of the `<system.webServer>` element.

Configuration

You can configure IISNode by using a `iisnode.yml` file or by adding the `<iisnode>` element as a child of `<system.webServer>` in your Web.config. Both of these configuration can be used in conjunction with one another however, in this case, Web.config will need to specify the `iisnode.yml` file **AND** any configuration conflicts will be take from the iisnode.yml file instead. This configuration overriding cannot happen the other way around.

IISNode Handler

In order for IIS to know that `server.js` contains our Node.js Web App we need to explicitly tell it that. We can do this by adding the IISNode `<handler>` to the `<handlers>` element.

```
<handlers>
  <add name="iisnode" path="server.js" verb="*" modules="iisnode" />
</handlers>
```

URL-Rewrite Rules

The final part of the configuration is ensuring that traffic intended for our Node.js app coming into our IIS instance is being directed to IISNode. Without URL rewrite rules, we would need to visit our app by going to `http://<host>/server.js` and even worse, when trying to request a resource supplied by `server.js` you'll get a **404**. This is why URL rewriting is necessary for IISNode web apps.

```
<rewrite>
  <rules>
    <!-- First we consider whether the incoming URL matches a physical file in the /public folder -->

    <rule name="StaticContent" patternSyntax="Wildcard">
      <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true" />
      <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
      </conditions>
      <match url="*.*/>
    </rule>
```

```

    <!-- All other URLs are mapped to the Node.js application entry point -->
    <rule name="DynamicContent">
      <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
      </conditions>
      <action type="Rewrite" url="server.js"/>
    </rule>
  </rules>
</rewrite>

```

This is a [working Web.config file for this example](#), setup for a 64-bit Node.js install.

That's it, now visit your IIS Site and see your Node.js application working.

Section 41.4: Using Socket.io with IISNode

To get Socket.io working with IISNode, the only changes necessary when not using a Virtual Directory/Nested Application are within the Web.config.

Since Socket.io sends requests starting with /socket.io, IISNode needs to communicate to IIS that these should also be handled IISNode and aren't just static file requests or other traffic. This requires a different **<handler>** than standard IISNode apps.

```

<handlers>
  <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>

```

In addition to the changes to the **<handlers>** we also need to add an additional URL rewrite rule. The rewrite rule sends all /socket.io traffic to our server file where the Socket.io server is running.

```

<rule name="SocketIO" patternSyntax="ECMAScript">
  <match url="socket.io.+"/>
  <action type="Rewrite" url="server.js"/>
</rule>

```

If you are using IIS 8, you'll need to disable your webSockets setting in your Web.config in addition to adding the above handler and rewrite rules. This is unnecessary in IIS 7 since there is no webSocket support.

```

<webSocket enabled="false" />

```

Chapter 42: CLI

Section 42.1: Command Line Options

```
-v, --version
```

Added in: v0.1.3 Print node's version.

```
-h, --help
```

Added in: v0.1.3 Print node command line options. The output of this option is less detailed than this document.

```
-e, --eval "script"
```

Added in: v0.5.2 Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.

```
-p, --print "script"
```

Added in: v0.6.4 Identical to -e but prints the result.

```
-c, --check
```

Added in: v5.0.0 Syntax check the script without executing.

```
-i, --interactive
```

Added in: v0.7.7 Opens the REPL even if stdin does not appear to be a terminal.

```
-r, --require module
```

Added in: v1.6.0 Preload the specified module at startup.

Follows require()'s module resolution rules. module may be either a path to a file, or a node module name.

```
--no-deprecation
```

Added in: v0.8.0 Silence deprecation warnings.

```
--trace-deprecation
```

Added in: v0.8.0 Print stack traces for deprecations.

```
--throw-deprecation
```

Added in: v0.11.14 Throw errors for deprecations.

```
--no-warnings
```

Added in: v6.0.0 Silence all process warnings (including deprecations).

```
--trace-warnings
```


Added in: v6.0.0 Print stack traces for process warnings (including deprecations).

```
--trace-sync-io
```

Added in: v2.1.0 Prints a stack trace whenever synchronous I/O is detected after the first turn of the event loop.

```
--zero-fill-buffers
```

Added in: v6.0.0 Automatically zero-fills all newly allocated Buffer and SlowBuffer instances.

```
--preserve-symlinks
```

Added in: v6.3.0 Instructs the module loader to preserve symbolic links when resolving and caching modules.

By default, when Node.js loads a module from a path that is symbolically linked to a different on-disk location, Node.js will dereference the link and use the actual on-disk "real path" of the module as both an identifier and as a root path to locate other dependency modules. In most cases, this default behavior is acceptable. However, when using symbolically linked peer dependencies, as illustrated in the example below, the default behavior causes an exception to be thrown if moduleA attempts to require moduleB as a peer dependency:

```
{appDir}
  ??? app
    ?   ??? index.js
    ?   ??? node_modules
    ?     ??? moduleA -> {appDir}/moduleA
    ?     ??? moduleB
    ?       ??? index.js
    ?       ??? package.json
  ??? moduleA
    ??? index.js
    ??? package.json
```

The `--preserve-symlinks` command line flag instructs Node.js to use the symlink path for modules as opposed to the real path, allowing symbolically linked peer dependencies to be found.

Note, however, that using `--preserve-symlinks` can have other side effects. Specifically, symbolically linked native modules can fail to load if those are linked from more than one location in the dependency tree (Node.js would see those as two separate modules and would attempt to load the module multiple times, causing an exception to be thrown).

```
--track-heap-objects
```

Added in: v2.4.0 Track heap object allocations for heap snapshots.

```
--prof-process
```

Added in: v6.0.0 Process v8 profiler output generated using the v8 option `--prof`.

```
--v8-options
```

Added in: v0.1.3 Print v8 command line options.

Note: v8 options allow words to be separated by both dashes (-) or underscores (_).

For example, `--stack-trace-limit` is equivalent to `--stack_trace_limit`.

```
--tls-cipher-list=list
```

Added in: v4.0.0 Specify an alternative default TLS cipher list. (Requires Node.js to be built with crypto support. (Default))

```
--enable-fips
```

Added in: v6.0.0 Enable FIPS-compliant crypto at startup. (Requires Node.js to be built with `./configure --openssl-fips`)

```
--force-fips
```

Added in: v6.0.0 Force FIPS-compliant crypto on startup. (Cannot be disabled from script code.) (Same requirements as `--enable-fips`)

```
--icu-data-dir=file
```

Added in: v0.11.15 Specify ICU data load path. (overrides `NODE_ICU_DATA`)

Environment Variables

```
NODE_DEBUG=module[, ...]
```

Added in: v0.1.32 `'`-separated list of core modules that should print debug information.

```
NODE_PATH=path[:...]
```

Added in: v0.1.32 `'`-separated list of directories prefixed to the module search path.

Note: on Windows, this is a `'`-separated list instead.

```
NODE_DISABLE_COLORS=1
```

Added in: v0.3.0 When set to 1 colors will not be used in the REPL.

```
NODE_ICU_DATA=file
```

Added in: v0.11.15 Data path for ICU (Intl object) data. Will extend linked-in data when compiled with small-icu support.

```
NODE_REPL_HISTORY=file
```

Added in: v5.0.0 Path to the file used to store the persistent REPL history. The default path is `~/.node_repl_history`, which is overridden by this variable. Setting the value to an empty string (`""` or `" "`) disables persistent REPL history.

Chapter 43: NodeJS Frameworks

Section 43.1: Web Server Frameworks

Express

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Koa

```
var koa = require('koa');
var app = koa();

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

Section 43.2: Command Line Interface Frameworks

Commander.js

```
var program = require('commander');

program
  .version('0.0.1')

program
  .command('hi')
  .description('initialize project configuration')
  .action(function(){
    console.log('Hi my Friend!!!');
  });

program
  .command('bye [name]')
  .description('initialize project configuration')
  .action(function(name){
    console.log('Bye ' + name + '. It was good to see you!');
  });

program
  .command('*')
  .action(function(env){
    console.log('Enter a Valid command');
```

```
    terminate(true);  
  });  
  
program.parse(process.argv);
```

Vorpal.js

```
const vorpal = require('vorpal')();  
  
vorpal  
  .command('foo', 'Outputs "bar".')  
  .action(function(args, callback) {  
    this.log('bar');  
    callback();  
  });  
  
vorpal  
  .delimiter('myapp$')  
  .show();
```

Chapter 44: grunt

Section 44.1: Introduction To GruntJs

Grunt is a JavaScript Task Runner, used for automation of repetitive tasks like minification, compilation, unit testing, linting, etc.

In order to get started, you'll want to install Grunt's command line interface (CLI) globally.

```
npm install -g grunt-cli
```

Preparing a new Grunt project: A typical setup will involve adding two files to your project: package.json and the Gruntfile.

package.json: This file is used by npm to store metadata for projects published as npm modules. You will list grunt and the Grunt plugins your project needs as devDependencies in this file.

Gruntfile: This file is named Gruntfile.js and is used to configure or define tasks and load Grunt plugins.

Example package.json:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
  }
}
```

Example gruntfile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%= pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};
```

Section 44.2: Installing gruntplugins

Adding dependency

To use a gruntplugin, you first need to add it as a dependency to your project. Let's use the jshint plugin as an example.

```
npm install grunt-contrib-jshint --save-dev
```

The `--save-dev` option is used to add the plugin in the `package.json`, this way the plugin is always installed after a `npm install`.

Loading the plugin

You can load your plugin in the gruntfile file using `loadNpmTasks`.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Configuring the task

You configure the task in the gruntfile adding a property called `jshint` to the object passed to `grunt.initConfig`.

```
grunt.initConfig({  
  jshint: {  
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']  
  }  
});
```

Don't forget you can have other properties for other plugins you are using.

Running the task

To just run the task with the plugin you can use the command line.

```
grunt jshint
```

Or you can add `jshint` to another task.

```
grunt.registerTask('default', ['jshint']);
```

The default task runs with the `grunt` command in the terminal without any options.

Chapter 45: Using WebSocket's with Node.JS

Section 45.1: Installing WebSocket's

There are a few way's to install WebSocket's to your project. Here are some example's:

```
npm install --save ws
```

or inside your package.json using:

```
"dependencies": {  
  "ws": "*"   
},
```

Section 45.2: Adding WebSocket's to your file's

To add ws to your file's simply use:

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

Section 45.3: Using WebSocket's and WebSocket Server's

To open a new WebSocket, simply add something like:

```
var WebSocket = require("ws");  
var ws = new WebSocket("ws://host:8080/OptionalPathName");  
// Continue on with your code...
```

Or to open a server, use:

```
var WebSocketServer = require("ws").Server;  
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

Section 45.4: A Simple WebSocket Server Example

```
var WebSocketServer = require('ws').Server  
, wss = new WebSocketServer({ port: 8080 }); // If you want to add a path as well, use path:  
"PathName"  
  
wss.on('connection', function connection(ws) {  
  ws.on('message', function incoming(message) {  
    console.log('received: %s', message);  
  });  
  
  ws.send('something');  
});
```

Chapter 46: metalsmith

Section 46.1: Build a simple blog

Assuming that you have node and npm installed and available, create a project folder with a valid package.json. Install the necessary dependencies:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Create a file called build.js at the root of your project folder, containing the following:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
  .use(inPlace('handlebars'))
  .build(function(err) {
    if (err) throw err;
    console.log('Build finished!');
  });
```

Create a folder called src at the root of your project folder. Create index.html in src, containing the following:

```
---
title: My awesome blog
---
```

```
{{ title }}
```

Running node build.js will now build all files in src. After running this command, you'll have index.html in your build folder, with the following contents:

```
<h1>My awesome blog</h1>
```


Chapter 47: Parsing command line arguments

Section 47.1: Passing action (verb) and values

```
const options = require("commander");

options
  .option("-v, --verbose", "Be verbose");

options
  .command("convert")
  .alias("c")
  .description("Converts input file to output file")
  .option("-i, --in-file <file_name>", "Input file")
  .option("-o, --out-file <file_name>", "Output file")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options){
  //do something with options.inFile and options.outFile
};
```

Section 47.2: Passing boolean switches

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose){
  console.log("Let's make some noise!");
}
```

Chapter 48: Client-server communication

Section 48.1: /w Express, jQuery and Jade

```
///'client.jade'

//a button is placed down; similar in HTML
button(type='button', id='send_by_button') Modify data

    #modify Lorem ipsum Sender

    //loading jQuery; it can be done from an online source as well
    script(src='./js/jquery-2.2.0.min.js')

    //AJAX request using jQuery
    script
        $(function () {
            $('#send_by_button').click(function (e) {
                e.preventDefault();

                //test: the text within brackets should appear when clicking on said button
                //window.alert('You clicked on me. - jQuery');

                //a variable and a JSON initialized in the code
                var predeclared = "Katamori";
                var data = {
                    Title: "Name_SenderTest",
                    Nick: predeclared,
                    FirstName: "Zoltan",
                    Surname: "Schmidt"
                };

                //an AJAX request with given parameters
                $.ajax({
                    type: 'POST',
                    data: JSON.stringify(data),
                    contentType: 'application/json',
                    url: 'http://localhost:7776/domaintest',

                    //on success, received data is used as 'data' function input
                    success: function (data) {
                        window.alert('Request sent; data received.');

                        var jsonstr = JSON.stringify(data);
                        var jsonobj = JSON.parse(jsonstr);

                        //if the 'nick' member of the JSON does not equal to the predeclared string
                        (as it was initialized), then the backend script was executed, meaning that communication has been
                        established

                        if(data.Nick != predeclared){
                            document.getElementById("modify").innerHTML = "JSON changed!\n" +
jsonstr;
                        }
                    }
                });
            });
        });

    //domaintest_route.js'
```

```

var express = require('express');
var router = express.Router();

//an Express router listening to GET requests - in this case, it's empty, meaning that nothing is
//displayed when you reach 'localhost/domainatest'
router.get('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
  res.setHeader('Content-Type', 'application/json');

  //content generated here
  var some_json = {
    Title: "Test",
    Item: "Crate"
  };

  var result = JSON.stringify(some_json);

  //content got 'client.jade'
  var sent_data = req.body;
  sent_data.Nick = "ttony33";

  res.send(sent_data);
});

module.exports = router;

```

//based on a personally used gist: <https://gist.github.com/Katamori/5c9850f02e4baf6e9896>

Chapter 49: Node.js Design Fundamental

Section 49.1: The Node.js philosophy

Small Core, Small Module :-

Build small and single purpose modules not in term of code size only, but also in term of scope that serves a single purpose

```
a - "Small is beautiful"
b - "Make each program do one thing well."
```

The Reactor Pattern

The Reactor Pattern is the heart of the node.js asynchronous nature. Allowed the system to be implemented as a single-threaded process with a series of event generators and event handlers, with the help of event loop that runs continuously.

The non-blocking I/O engine of Node.js – libuv -

The Observer Pattern(EventEmitter) maintains a list of dependents/observers and notifies them

```
var events = require('events');
var EventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
  console.log('tring tring tring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Chapter 50: Connect to Mongodb

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas.

For more details go to <https://www.mongodb.com/>

Section 50.1: Simple example to Connect mongoDB from Node.JS

```
MongoClient.connect('mongodb://localhost:27017/myNewDB', function (err, db) {  
  if(err)  
    console.log("Unable to connect DB. Error: " + err)  
  else  
    console.log('Connected to DB');  
  
  db.close();  
});
```

myNewDB is DB name, if it does not exists in database then it will create automatically with this call.

Section 50.2: Simple way to Connect mongoDB with core Node.JS

```
var MongoClient = require('mongodb').MongoClient;  
  
//connection with mongoDB  
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {  
  //check the connection  
  if(err){  
    console.log("connection failed.");  
  }else{  
    console.log("successfully connected to mongoDB.");  
  }  
});
```

Chapter 51: Performance challenges

Section 51.1: Processing long running queries with Node

Since Node is single-threaded, there is a need of workaround if it comes to a long-running calculations.

Note: this is "ready to run" example. Just, don't forget to get jQuery and install the required modules.

Main logic of this example:

1. Client sends request to the server.
2. Server starts the routine in separate node instance and sends immediate response back with related task ID.
3. Client continuously sends checks to a server for status updates of the given task ID.

Project structure:

```
project
?  package.json
?  index.html
?
????js
?    main.js
?    jquery-1.12.0.min.js
?
????srv
?    app.js
???? models
?    task.js
???? tasks
    data-processor.js
```

app.js:

```
var express      = require('express');
var app          = express();
var http         = require('http').Server(app);
var mongoose     = require('mongoose');
var bodyParser   = require('body-parser');

var childProcess= require('child_process');

var Task         = require('./models/task');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
    //create new task item for status tracking
    var t = new Task({ status: 'Starting ...' });

    t.save(function(err, task){
```

```

//create new instance of node for running separate task in another thread
taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

//process the messages coming from the task processor
taskProcessor.on('message', function(msg){
    task.status = msg.status;
    task.save();
}).bind(this));

//remove previously opened node instance when we finished
taskProcessor.on('close', function(msg){
    this.kill();
});

//send some params to our separate task
var params = {
    message: 'Hello from main thread'
};

taskProcessor.send(params);
response.status(200).json(task);
});
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');

```

task.js:

```

var mongoose = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);

module.exports = mongoose.model('Task');

```

data-processor.js:

```

process.on('message', function(msg){
    init = function(){
        processData(msg.message);
    }.bind(this)();

    function processData(message){
        //send status update to the main app
        process.send({ status: 'We have started processing your data.' });
    }
});

```

```

    //long calculations ..
    setTimeout(function(){
        process.send({ status: 'Done!' });

        //notify node, that we are done with this task
        process.disconnect();
    }, 5000);
}
});

process.on('uncaughtException',function(err){
    console.log("Error happened: " + err.message + "\n" + err.stack + "\n");
    console.log("Gracefully finish the routine.");
});

```

index.html:

```

<!DOCTYPE html>
<html>
  <head>
    <script src="./js/jquery-1.12.0.min.js"></script>
    <script src="./js/main.js"></script>
  </head>
  <body>
    <p>Example of processing long-running node requests.</p>
    <button id="go" type="button">Run</button>

    <br />

    <p>Log:</p>
    <textarea id="log" rows="20" cols="50"></textarea>
  </body>
</html>

```

main.js:

```

$(document).on('ready', function(){

    $('#go').on('click', function(e){
        //clear log
        $('#log').val('');

        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
                $('#log').val( $('#log').val() + '\n' + task.status);

                //function for tracking the status of the task
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
                        .done(function(response){
                            $('#log').val( $('#log').val() + '\n' + response.status);

                            if(response.status != 'Done!'){
                                checkTaskTimeout = setTimeout(updateStatus, 500);
                            }
                        });
                }

                //start checking the task
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });

```



```
});  
});
```

package.json:

```
{  
  "name": "nodeProcessor",  
  "dependencies": {  
    "body-parser": "^1.15.2",  
    "express": "^4.14.0",  
    "html": "0.0.10",  
    "mongoose": "^4.5.5"  
  }  
}
```

Disclaimer: this example is intended to give you basic idea. To use it in production environment, it needs improvements.

Chapter 52: Send Web Notification

Section 52.1: Send Web notification using GCM (Google Cloud Messaging System)

Such Example is knowing wide spreading among **PWAs** (Progressive Web Applications) and in this example we're going to send a simple Backend like notification using **NodeJS** and **ES6**

1. Install Node-GCM Module : `npm install node-gcm`
2. Install Socket.io : `npm install socket.io`
3. Create a GCM Enabled application using [Google Console](#).
4. Grabe your GCM Application Id (we will need it later on)
5. Grabe your GCM Application Secret code.
6. Open Your favorite code editor and add the following code :

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
  data: {
    key1: 'msg1'
  }
});

// [*] Configuring our static files.
app.use(express.static('public/'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
  console.log('we have a new connection ...');
  socket.on('new_user', (reg_id) => {
    // [*] Adding our user notification registration token to our list typically hided in
    a secret place.
    if (regTokens.indexOf(reg_id) === -1) {
      regTokens.push(reg_id);
    }
  });
});
```

```

        // [*] Sending our push messages
        sender.send(message, {
            registrationTokens: regTokens
        }, (err, response) => {
            if (err) console.error('err', err);
            else console.log(response);
        });
    }
});

module.exports = app

```

PS : I'm using here a special hack in order to make Socket.io works with Express because simply it doesn't work outside of the box.

Now Create a **.json** file and name it : **Manifest.json**, open it and past the following :

```

{
  "name": "Application Name",
  "gcm_sender_id": "GCM Project ID"
}

```

Close it and save in your application **ROOT** directory.

PS : the Manifest.json file needs to be in root directory or it won't work.

In the code above I'm doing the following :

1. I seted up and sent a normal index.html page that will use socket.io also.
2. I'm listening on a **connection** event fired from the **front-end** aka my **index.html page** (it will be fired once a new client successfully connected to our pre-defined link)
3. I'm sending a special token know's as the **registration token** from my index.html via socket.io **new_user** event, such token will be our user unique passcode and each code is generated usually from a supporting browser for the **Web notification API** (read more [here](#)).
4. I'm simply using the **node-gcm** module to send my notification which will be handled and shown later on using **Service Workers`**.

This is from **NodeJS** point of view. in other examples I will show how we can send custom data, icons ..etc in our push message.

PS : you can find the full working demo over [here](#).

Chapter 53: Remote Debugging in Node.JS

Section 53.1: Use the proxy for debugging via port on Linux

If you start your application on Linux, use the proxy for debugging via port, for example:

```
socat TCP-LISTEN:9958, fork TCP:127.0.0.1:5858 &
```

Use port 9958 for remote debugging then.

Section 53.2: NodeJS run configuration

To set up Node remote debugging, simply run the node process with the `--debug` flag. You can add a port on which the debugger should run using `--debug=<port>`.

When your node process starts up you should see the message

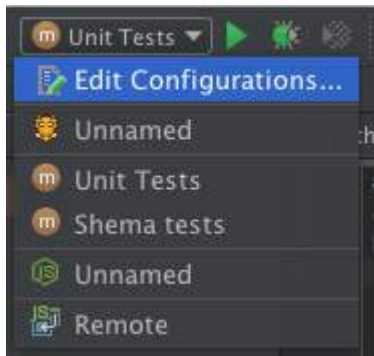
```
Debugger listening on port <port>
```

Which will tell you that everything is good to go.

Then you set up the remote debugging target in your specific IDE.

Section 53.3: IntelliJ/Webstorm Configuration

1. Make sure that the NodeJS plugin is enabled
2. Select your run configurations (screen)



3. Select + > **Node.js Remote Debug**



4. Make sure you enter the port selected above as well as the correct host

A screenshot of the configuration form for 'Node.js Remote Debug'. The form has a dark background. It contains three input fields: 'Name' with the value 'Remote', 'Host' with the value '127.0.0.1', and 'Port' with the value '5859'. To the right of the 'Name' field, there are two checkboxes: 'Share' (unchecked) and 'Single instance only' (checked).

Once those are configured simply run the debug target as you normally would and it will stop on your breakpoints.

Chapter 54: Mysql Connection Pool

Section 54.1: Using a connection pool without database

Achieving multitenancy on database server with multiple databases hosted on it.

Multitenancy is common requirement of enterprise application nowadays and creating connection pool for each database in database server is not recommended. so, what we can do instead is create connection pool with database server and than switch between databases hosted on database server on demand.

Suppose our application has different databases for each firm hosted on database server. We will connect to respective firm database when user hits the application. here is the example on how to do that:-

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Let me break down the example:-

When defining pool configuration i did not gave the database name but only gave database server i.e

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

so when we want to use the specific database on database server, we ask the connection to hit database by using:-

```
connection.changeUser({database : "firm1"});
```

you can refer the official documentation [here](#)

Chapter 55: Database (MongoDB with Mongoose)

Section 55.1: Mongoose connection

Make sure to have mongod running first! `mongod --dbpath data/`

package.json

```
"dependencies": {  
  "mongoose": "^4.5.5",  
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
const db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
var db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

Section 55.2: Model

Define your model(s):

app/models/user.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
const userSchema = new mongoose.Schema({  
  name: String,  
  password: String  
});  
  
const User = mongoose.model('User', userSchema);  
  
export default User;
```

app/model/user.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
var userSchema = new mongoose.Schema({  
  name: String,  
  password: String  
});  
  
var User = mongoose.model('User', userSchema);
```

```
module.exports = User
```

Section 55.3: Insert data

ECMA 6:

```
const user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save((err) => {
  if (err) throw err;

  console.log('User saved!');
});
```

ECMA5.1:

```
var user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save(function (err) {
  if (err) throw err;

  console.log('User saved!');
});
```

Section 55.4: Read data

ECMA6:

```
User.findOne({
  name: 'stack'
}, (err, user) => {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

ECMA5.1:

```
User.findOne({
  name: 'stack'
}, function (err, user) {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```



```
});
```

Chapter 56: Good coding style

Section 56.1: Basic program for signup

Through this example, it will be explained to divide the **node.js** code into different **modules/folders** for better understandability. Following this technique makes it easier for other developers to understand the code as he can directly refer to concerned file instead of going through whole code. The major use is when you are working in a team and a new developer joins at a later stage, it will get easier for him to gel up with the code itself.

index.js :- This file will manage server connection.

```
//Import Libraries
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

//Import custom modules
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//Connect to Mongo DB
mongoose.connect(config.getDBString());

//Create a new Express application and Configure it
var app = express();

//Configure Routes
app.use(config.API_PATH, userRoutes());

//Start the server
app.listen(config.PORT);
console.log('Server started at - ' + config.URL + ":" + config.PORT);
```

config.js:-This file will manage all the configuration related params which will remain same throughout.

```
var config = {
  VERSION: 1,
  BUILD: 1,
  URL: 'http://127.0.0.1',
  API_PATH : '/api',
  PORT : process.env.PORT || 8080,
  DB : {
    //MongoDB configuration
    HOST : 'localhost',
    PORT : '27017',
    DATABASE : 'db'
  },
  /*
   * Get DB Connection String for connecting to MongoDB database
   */
  getDBString : function(){
    return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/' + this.DB.DATABASE;
  },
  /*
   * Get the http URL
   */
}
```

```

*/
getHTTPOr1 : function(){
    return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;

```

user.js:- Model file where schema is defined

```

var mongoose = require('mongoose');
var Schema = mongoose.Schema;

//Schema for User
var UserSchema = new Schema({
  name: {
    type: String,
    // required: true
  },
  email: {
    type: String
  },
  password: {
    type: String,
    //required: true
  },
  dob: {
    type: Date,
    //required: true
  },
  gender: {
    type: String, // Male/Female
    // required: true
  }
});

//Define the model for User
var User;
if(mongoose.models.User)
  User = mongoose.model('User');
else
  User = mongoose.model('User', UserSchema);

//Export the User Model
module.exports = User;

```

UserController:- This file contains the function for user signUp

```

var User = require('../models/user');
var crypto = require('crypto');

//Controller for User
var UserController = {

  //Create a User
  create: function(req, res){
    var repassword = req.body.repassword;
    var password = req.body.password;
    var userEmail = req.body.email;

    //Check if the email address already exists
    User.find({"email": userEmail}, function(err, usr){

```



```
var UserController = require('../controllers/userController');

//Routes for User
var UserRoutes = function(app)
{
    var router = express.Router();

    router.route('/users')
        .post(UserController.create);

    return router;
}

module.exports = UserRoutes;
```

The above example may appear too big but if a beginner at node.js with a little blend of express knowledge tries to go through this will find it easy and really helpful.

Chapter 57: Restful API Design: Best Practices

Section 57.1: Error Handling: GET all resources

How do you handle errors, rather than log them to the console?

Bad way:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
```

Better way:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        return next(err)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        return next(err)
      } else {

```

```
    res.json(r);  
  }  
});  
});
```

Chapter 58: Deliver HTML or any other sort of file

Section 58.1: Deliver HTML at specified path

Here's how to create an Express server and serve `index.html` by default (empty path `/`), and `page1.html` for `/page1` path.

Folder structure

```
project root
|   server.js
|___views
|   index.html
|   page1.html
```

server.js

```
var express = require('express');
var path = require('path');
var app = express();

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function(request, response) {
  response.sendFile(path.join(__dirname, 'views', 'page1.html', function(error) {
    if (error) {
      // do something in case of error
      console.log(err);
      response.end(JSON.stringify({error:"page not found"}));
    }
  }));
});

app.listen(8080);
```

Note that `sendFile()` just streams a static file as response, offering no opportunity to modify it. If you are serving an HTML file and want to include dynamic data with it, then you will need to use a *template engine* such as Pug, Mustache, or EJS.

Chapter 59: TCP Sockets

Section 59.1: A simple TCP server

```
// Include Nodejs' 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 = new Net.Server();
// The server listens to a socket for a client to make a connection request.
// Think of a socket as an end point.
server.listen(port, function() {
  console.log(`Server listening for connection requests on socket localhost:${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: ${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: ${err}`);
  });
});
```

Section 59.2: A simple TCP client

```
// Include Nodejs' 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 = new 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.');
```

```
// The client can now send data to the server by writing to its socket.
client.write('Hello, server.');
```



```
});
```



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

  // Request an end to the connection after the data has been received.
  client.end();
});
```



```
client.on('end', function() {
  console.log('Requested an end to the TCP connection');
});
```

Chapter 60: Hack

Section 60.1: Add new extensions to require()

You can add new extensions to `require()` by extending `require.extensions`.

For a **XML** example:

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
  const fs = require('fs')
  const xml2js = require('xml2js')

  module.exports = (callback) => {
    // Read required file.
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        callback(err)
        return
      }
      // Parse it.
      xml2js.parseString(data, (err, result) => {
        callback(null, result)
      })
    })
  }
}
```

If the content of `hello.xml` is following:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>baz</bar>
  <qux />
</foo>
```

You can read and parse it through `require()`:

```
require('./hello')((err, xml) {
  if (err)
    throw err;
  console.log(err);
})
```

It prints `{ foo: { bar: ['baz'], qux: [''] } }`.

Chapter 61: Bluebird Promises

Section 61.1: Converting nodeback library to Promises

```
const Promise = require('bluebird'),
      fs = require('fs')

Promise.promisifyAll(fs)

// now you can use promise based methods on 'fs' with the Async suffix
fs.readFileAsync('file.txt').then(contents => {
  console.log(contents)
}).catch(err => {
  console.error('error reading', err)
})
```

Section 61.2: Functional Promises

Example of map:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {
  return Promise.resolve(el * el) // return some async operation in real world
})
```

Example of filter:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {
  return Promise.resolve(el % 2 === 0) // return some async operation in real world
}).then(console.log)
```

Example of reduce:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {
  return Promise.resolve(prev + curr) // return some async operation in real world
}).then(console.log)
```

Section 61.3: Coroutines (Generators)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {
  const data = yield fs.readFileAsync(file) // this returns a Promise and resolves to the file contents

  return data.toString().toUpperCase()
})

promiseReturningFunction('file.txt').then(console.log)
```

Section 61.4: Automatic Resource Disposal (Promise.using)

```
function somethingThatReturnsADisposableResource() {
  return getSomeResourceAsync(...).disposer(resource => {
    resource.dispose()
  })
}
```

```
Promise.using(somethingThatReturnsADisposableResource(), resource => {  
  // use the resource here, the disposer will automatically close it when Promise.using exits  
})
```

Section 61.5: Executing in series

```
Promise.resolve([1, 2, 3])  
  .mapSeries(el => Promise.resolve(el * el)) // in real world, use Promise returning async function  
  .then(console.log)
```

Chapter 62: Async/Await

Async/await is a set of keywords that allows writing of asynchronous code in a procedural manner without having to rely on callbacks (*callback hell*) or promise-chaining (`.then().then().then()`).

This works by using the `await` keyword to suspend the state of an async function, until the resolution of a promise, and using the `async` keyword to declare such async functions, which return a promise.

Async/await is available from node.js 8 by default or 7 using the flag `--harmony-async-await`.

Section 62.1: Comparison between Promises and Async/Await

Function using promises:

```
function myAsyncFunction() {
  return aFunctionThatReturnsAPromise()
    // doSomething is a sync function
    .then(result => doSomething(result))
    .catch(handleError);
}
```

So here is when Async/Await enter in action in order to get cleaner our function:

```
async function myAsyncFunction() {
  let result;

  try {
    result = await aFunctionThatReturnsAPromise();
  } catch (error) {
    handleError(error);
  }

  // doSomething is a sync function
  return doSomething(result);
}
```

So the keyword `async` would be similar to write `return new Promise((resolve, reject) => {...})`.

And `await` similar to get your result in then callback.

Here I leave a pretty brief gif that will not left any doubt in mind after seeing it:

[GIF](#)

Section 62.2: Async Functions with Try-Catch Error Handling

One of the best features of async/await syntax is that standard try-catch coding style is possible, just like you were writing synchronous code.

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // handle errors here
  }
};
```

Here's an example with Express and promise-mysql:

```
router.get('/flags/:id', async (req, res) => {

  try {

    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
                    FROM flags f
                    WHERE f.id = ?
                    LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });

    } finally {
      pool.releaseConnection(connection);
    }

  } catch (err) {
    // handle errors here
  }
});
```

Section 62.3: Stops execution at await

If the promise doesn't return anything, the async task can be completed using await.

```
try{
  await User.findByIdAndUpdate(user._id, {
    $push: {
      tokens: token
    }
  }).exec()
}catch(e){
  handleError(e)
}
```

Section 62.4: Progression from Callbacks

In the beginning there were callbacks, and callbacks were ok:

```
const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {
  http.get('www.pollution.com/current', (res) => {
    callback(res.data.pollution)
  });
}

getTemperature(function(temp) {
```

```

getAirPollution(function(pollution) {
  console.log(`the temp is ${temp} and the pollution is ${pollution}.`)
  // The temp is 27 and the pollution is 0.5.
})
})

```

But there were a few really frustrating issues with callbacks so we all started using promises.

```

const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
  .then(temp => console.log(`the temp is ${temp}`))
  .then(() => getAirPollution())
  .then(pollution => console.log(`and the pollution is ${pollution}`))
// the temp is 32
// and the pollution is 0.5

```

This was a bit better. Finally, we found async/await. Which still uses promises under the hood.

```

const temp = await getTemperature()
const pollution = await getAirPollution()

```


Chapter 63: Koa Framework v2

Section 63.1: Hello World example

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
  ctx.body = 'Hello World'
})

app.listen(8080)
```

Section 63.2: Handling errors using middleware

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

Chapter 64: Unit testing frameworks

Section 64.1: Mocha Asynchronous (async/await)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
})
```

Section 64.2: Mocha synchronous

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

Section 64.3: Mocha asynchronous (callback)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

Chapter 65: ECMAScript 2015 (ES6) with Node.js

Section 65.1: const/let declarations

Unlike **var**, **const**/let are bound to lexical scope rather than function scope.

```
{
  var x = 1 // will escape the scope
  let y = 2 // bound to lexical scope
  const z = 3 // bound to lexical scope, constant
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

[Run in RunKit](#)

Section 65.2: Arrow functions

Arrow functions automatically bind to the 'this' lexical scope of the surrounding code.

```
performSomething(result => {
  this.someVariable = result
})
```

vs

```
performSomething(function(result) {
  this.someVariable = result
}).bind(this))
```

Section 65.3: Arrow Function Example

Let's consider this example, that outputs the squares of the numbers 3, 5, and 7:

```
let nums = [3, 5, 7]
let squares = nums.map(function (n) {
  return n * n
})
console.log(squares)
```

[Run in RunKit](#)

The function passed to `.map` can also be written as arrow function by removing the **function** keyword and instead adding the arrow `=>`:

```
let nums = [3, 5, 7]
let squares = nums.map((n) => {
  return n * n
})
console.log(squares)
```

[Run in RunKit](#)

However, this can be written even more concise. If the function body consists of only one statement and that statement computes the return value, the curly braces of wrapping the function body can be removed, as well as the **return** keyword.

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

[Run in RunKit](#)

Section 65.4: destructuring

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

Section 65.5: flow

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
}));
```

Section 65.6: ES6 Class

```
class Mammel {
  constructor(legs){
    this.legs = legs;
  }
  eat(){
    console.log('eating...');
  }
  static count(){
    console.log('static count...');
  }
}
```

```
class Dog extends Mammel{
  constructor(name, legs){
    super(legs);
    this.name = name;
  }
  sleep(){
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Chapter 66: Routing ajax requests with Express.JS

Section 66.1: A simple implementation of AJAX

You should have the basic express-generator template

In app.js, add (you can add it anywhere after `var app = express.app()`):

```
app.post(function(req, res, next){
  next();
});
```

Now in your index.js file (or its respective match), add:

```
router.get('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});
});
router.post('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});
});
```

Create an ajax.jade / ajax.pug or ajax.ejs file in /views directory, add:

For Jade/PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: !{quote}
form(method="post" id="changeQuote")
  input(type='text', placeholder='Set quote of the day', name='quote')
  input(type="submit", value="Save")
```

For EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
  <input type="text" placeholder="Set quote of the day" name="quote"/>
  <input type="submit" value="Save">
</form>
```

Now, create a file in /public called magic.js

```
$(document).ready(function(){
  $("form#changeQuote").on('submit', function(e){
    e.preventDefault();
    var data = $('input[name=quote]').val();
    $.ajax({
      type: 'post',
      url: '/ajax',
      data: data,
      dataType: 'text'
    })
  })
});
```

```
    .done(function(data){  
        $('h1').html(data.quote);  
    });  
});
```

And there you have it! When you click Save the quote will change!

Chapter 67: Sending a file stream to client

Section 67.1: Using fs And pipe To Stream Static Files From The Server

A good VOD (Video On Demand) service should start with the basics. Lets say you have a directory on your server that is not publicly accessible, yet through some sort of portal or paywall you want to allow users to access your media.

```
var movie = path.resolve('./public/' + req.params.filename);

fs.stat(movie, function (err, stats) {

  var range = req.headers.range;

  if (!range) {

    return res.sendStatus(416);

  }

  //Chunk logic here
  var positions = range.replace(/bytes=/, "").split("-");
  var start = parseInt(positions[0], 10);
  var total = stats.size;
  var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
  var chunksize = (end - start) + 1;

  res.writeHead(206, {

    'Transfer-Encoding': 'chunked',

    "Content-Range": "bytes " + start + "-" + end + "/" + total,

    "Accept-Ranges": "bytes",

    "Content-Length": chunksize,

    "Content-Type": mime.lookup(req.params.filename)

  });

  var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true })

    .on('end', function () {

      console.log('Stream Done');

    })

    .on("error", function (err) {

      res.end(err);

    })

    .pipe(res, { end: true });

});
```


The above snippet is a basic outline for how you would like to stream your video to a client. The chunk logic depends on a variety of factors, including network traffic and latency. It is important to balance chunk size vs. quantity.

Finally, the `.pipe` call lets node.js know to keep a connection open with the server and to send additional chunks as needed.

Section 67.2: Streaming Using fluent-ffmpeg

You can also use `fluent-ffmpeg` to convert `.mp4` files to `.flv` files, or other types:

```
res.contentType('flv');
```

```
var pathToMovie = './public/' + req.params.filename;

var proc = ffmpeg(pathToMovie)

  .preset('flashvideo')

  .on('end', function () {

    console.log('Stream Done');

  })

  .on('error', function (err) {

    console.log('an error happened: ' + err.message);

    res.send(err.message);

  })

  .pipe(res, { end: true });
```

Chapter 68: NodeJS with Redis

Section 68.1: Getting Started

node_redis, as you may have guessed, is the [Redis client for Node.js](#). You can install it via npm using the following command.

```
npm install redis
```

Once you have installed node_redis module you are good to go. Let's create a simple file, app.js, and see how to connect with Redis from Node.js.

```
app.js
```

```
var redis = require('redis');  
client = redis.createClient(); //creates a new client
```

By default, redis.createClient() will use 127.0.0.1 and 6379 as the hostname and port respectively. If you have a different host/port you can supply them as following:

```
var client = redis.createClient(port, host);
```

Now, you can perform some action once a connection has been established. Basically, you just need to listen for connect events as shown below.

```
client.on('connect', function() {  
    console.log('connected');  
});
```

So, the following snippet goes into app.js:

```
var redis = require('redis');  
var client = redis.createClient();  
  
client.on('connect', function() {  
    console.log('connected');  
});
```

Now, type node app in the terminal to run the app. Make sure your Redis server is up and running before running this snippet.

Section 68.2: Storing Key-Value Pairs

Now that you know how to connect with Redis from Node.js, let's see how to store key-value pairs in Redis storage.

Storing Strings

All the Redis commands are exposed as different functions on the client object. To store a simple string use the following syntax:

```
client.set('framework', 'AngularJS');
```

Or

```
client.set(['framework', 'AngularJS']);
```

The above snippets store a simple string AngularJS against the key framework. You should note that both the snippets do the same thing. The only difference is that the first one passes a variable number of arguments while the later passes an args array to `client.set()` function. You can also pass an optional callback to get a notification when the operation is complete:

```
client.set('framework', 'AngularJS', function(err, reply) {  
  console.log(reply);  
});
```

If the operation failed for some reason, the `err` argument to the callback represents the error. To retrieve the value of the key do the following:

```
client.get('framework', function(err, reply) {  
  console.log(reply);  
});
```

`client.get()` lets you retrieve a key stored in Redis. The value of the key can be accessed via the callback argument `reply`. If the key doesn't exist, the value of `reply` will be empty.

Storing Hash

Many times storing simple values won't solve your problem. You will need to store hashes (objects) in Redis. For that you can use `hmset()` function as following:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');  
  
client.hgetall('frameworks', function(err, object) {  
  console.log(object);  
});
```

The above snippet stores a hash in Redis that maps each technology to its framework. The first argument to `hmset()` is the name of the key. Subsequent arguments represent key-value pairs. Similarly, `hgetall()` is used to retrieve the value of the key. If the key is found, the second argument to the callback will contain the value which is an object.

Note that Redis doesn't support nested objects. All the property values in the object will be coerced into strings before getting stored. You can also use the following syntax to store objects in Redis:

```
client.hmset('frameworks', {  
  'javascript': 'AngularJS',  
  'css': 'Bootstrap',  
  'node': 'Express'  
});
```

An optional callback can also be passed to know when the operation is completed.

All the functions (commands) can be called with uppercase/lowercase equivalents. For example, `client.hmset()`

and `client.HMSET()` are the same. Storing Lists

If you want to store a list of items, you can use Redis lists. To store a list use the following syntax:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
  console.log(reply); //prints 2
});
```

The above snippet creates a list called `frameworks` and pushes two elements to it. So, the length of the list is now two. As you can see I have passed an args array to `rpush`. The first item of the array represents the name of the key while the rest represent the elements of the list. You can also use `lpush()` instead of `rpush()` to push the elements to the left.

To retrieve the elements of the list you can use the `lrange()` function as following:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
  console.log(reply); // ['angularjs', 'backbone']
});
```

Just note that you get all the elements of the list by passing `-1` as the third argument to `lrange()`. If you want a subset of the list, you should pass the end index here.

Storing Sets

Sets are similar to lists, but the difference is that they don't allow duplicates. So, if you don't want any duplicate elements in your list you can use a set. Here is how we can modify our previous snippet to use a set instead of list.

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
  console.log(reply); // 3
});
```

As you can see, the `sadd()` function creates a new set with the specified elements. Here, the length of the set is three. To retrieve the members of the set, use the `smembers()` function as following:

```
client.smembers('tags', function(err, reply) {
  console.log(reply);
});
```

This snippet will retrieve all the members of the set. Just note that the order is not preserved while retrieving the members.

This was a list of the most important data structures found in every Redis powered app. Apart from strings, lists, sets, and hashes, you can store sorted sets, hyperLogLogs, and more in Redis. If you want a complete list of commands and data structures, visit the official Redis documentation. Remember that almost every Redis command is exposed on the client object offered by the `node_redis` module.

Section 68.3: Some more important operations supported by node_redis

Checking the Existence of Keys

Sometimes you may need to check if a key already exists and proceed accordingly. To do so you can use `exists()`

function as shown below:

```
client.exists('key', function(err, reply) {
  if (reply === 1) {
    console.log('exists');
  } else {
    console.log('doesn\'t exist');
  }
});
```

Deleting and Expiring Keys

At times you will need to clear some keys and reinitialize them. To clear the keys, you can use `del` command as shown below:

```
client.del('frameworks', function(err, reply) {
  console.log(reply);
});
```

You can also give an expiration time to an existing key as following:

```
client.set('key1', 'val1');
client.expire('key1', 30);
```

The above snippet assigns an expiration time of 30 seconds to the key `key1`.

Incrementing and Decrementing

Redis also supports incrementing and decrementing keys. To increment a key use `incr()` function as shown below:

```
client.set('key1', 10, function() {
  client.incr('key1', function(err, reply) {
    console.log(reply); // 11
  });
});
```

The `incr()` function increments a key value by 1. If you need to increment by a different amount, you can use `incrby()` function. Similarly, to decrement a key you can use the functions like `decr()` and `decrby()`.

Chapter 69: Using Browserify to resolve 'required' error with browsers

Section 69.1: Example - file.js

In this example we have a file called **file.js**.

Let's assume that you have to parse a URL using JavaScript and NodeJS `querystring` module.

To accomplish this all you have to do is to insert the following statement in your file:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

What is this snippet doing?

Well, first, we create a `querystring` module which provides utilities for parsing and formatting URL query strings. It can be accessed using:

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

Then, we parse a URL using the `.parse()` method. It parses a URL query string (`str`) into a collection of key and value pairs.

For example, the query string `'foo=bar&abc=xyz&abc=123'` is parsed into:

```
{ foo: 'bar', abc: ['xyz', '123'] }
```

Unfortunately, Browsers don't have the `require` method defined, but Node.js does.

Install Browserify

With Browserify you can write code that uses `require` in the same way that you would use it in Node. So, how do you solve this? It's simple.

1. First install node, which ships with npm. Then do:

```
npm install -g browserify
```

2. Change into the directory in which your `file.js` is and Install our `querystring` module with npm:

```
npm install querystring
```

Note: If you don't change in the specific directory the command will fail because it can't find the file which contains the module.

3. Now recursively bundle up all the required modules starting at `file.js` into a single file called `bundle.js` (or whatever you like to name it) with the **browserify command**:

```
browserify file.js -o bundle.js
```

Browserify parses the Abstract Syntax Tree for *require()* calls to traverse the entire dependency graph of your

4. Finally Drop a single tag into your html and you're done!

```
<script src="bundle.js"></script>
```

What happens is that you get a combination of your old .js file (**file.js** that is) and your newly created **bundle.js** file. Those two files are merged into one single file.

Important

Please keep in mind that if you want to make any changes to your file.js and will not affect the behaviour of your program. **Your changes will only take effect if you edit the newly created bundle.js**

What does that mean?

This means that if you want to edit **file.js** for any reasons, the changes will not have any effects. You really have to edit **bundle.js** since it is a merge of **bundle.js** and **file.js**.

Chapter 70: Node.JS and MongoDB.

Section 70.1: Connecting To a Database

To connect to a mongo database from node application we require mongoose.

Installing Mongoose Go to the toot of your application and install mongoose by

```
npm install mongoose
```

Next we connect to the database.

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

//connect to the test database running on default mongod port of localhost
mongoose.connect('mongodb://localhost/test');

//Connecting with custom credentials
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');

//Using Pool Size to define the number of connections opening
//Also you can use a call back function for error handling
mongoose.connect('mongodb://localhost:27017/consumers',
  {server: { poolSize: 50 }},
  function(err) {
    if(err) {
      console.log('error in this')
      console.log(err);
      // Do whatever to handle the error
    } else {
      console.log('Connected to the database');
    }
  }
);
```

Section 70.2: Creating New Collection

With Mongoose, everything is derived from a Schema. Lets create a schema.

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

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
  name : String,
  countOf: Number,
});
// defining the document structure

// by default the collection created in the db would be the first parameter we use (or the plural of it)
module.exports = mongoose.model('Auto', AutoSchema);

// we can over write it and define the collection name by specifying that in the third parameters.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');
```



```
// We can also define methods in the models.
AutoSchema.methods.speak = function () {
  var greeting = this.name
    ? "Hello this is " + this.name+ " and I have counts of "+ this.countOf
    : "I don't have a name";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Remember methods must be added to the schema before compiling it with `mongoose.model()` like done above ..

Section 70.3: Inserting Documents

For inserting a new document in the collection, we create a object of the schema.

```
var Auto = require('models/auto')
var autoObj = new Auto({
  name: "NewName",
  countOf: 10
});
```

We save it like the following

```
autoObj.save(function(err, insertedAuto) {
  if (err) return console.error(err);
  insertedAuto.speak();
  // output: Hello this is NewName and I have counts of 10
});
```

This will insert a new document in the collection

Section 70.4: Reading

Reading Data from the collection is very easy. Getting all data of the collection.

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection
  console.log(autos);
})
```

Reading data with a condition

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection whose count is greater than
  5
  console.log(autos);
})
```

You can also specify the second parameter as object of what all fields you need

```
Auto.find({}, {name:1}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of name field of all the documents in the collection
  console.log(autos);
})
```

```
})
```

Finding one document in a collection.

```
Auto.findOne({name:"newName"}, function (err, auto) {  
    if (err) return console.error(err);  
    //will return the first object of the document whose name is "newName"  
    console.log(auto);  
})
```

Finding one document in a collection by id .

```
Auto.findById(123, function (err, auto) {  
    if (err) return console.error(err);  
    //will return the first json object of the document whose id is 123  
    console.log(auto);  
})
```

Section 70.5: Updating

For updating collections and documents we can use any of these methods:

Methods

- update()
- updateOne()
- updateMany()
- replaceOne()

Update()

The update() method modifies one or many documents (update parameters)

```
db.lights.update(  
  { room: "Bedroom" },  
  { status: "On" }  
)
```

This operation searches the 'lights' collection for a document where room is **Bedroom** (1st parameter). It then updates the matching documents status property to **On** (2nd parameter) and returns a WriteResult object that looks like this:

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

The UpdateOne() method modifies ONE document (update parameters)

```
db.countries.update(  
  { country: "Sweden" },  
  { capital: "Stockholm" }  
)
```

This operation searches the 'countries' collection for a document where country is **Sweden** (1st parameter). It then updates the matching documents property capital to **Stockholm** (2nd parameter) and returns a WriteResult object that looks like this:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

The UpdateMany() method modifies multiple documents (update parameters)

```
db.food.updateMany(  
  { sold: { $lt: 10 } },  
  { $set: { sold: 55 } }  
)
```

This operation updates all documents (in a 'food' collection) where sold is **less than 10** *(1st parameter) by setting sold to **55**. It then returns a WriteResult object that looks like this:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

a = Number of matched documents

b = Number of modified documents

ReplaceOne

Replaces the first matching document (replacement document)

This example collection called **countries** contains 3 documents:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Spain" }
```

The following operation replaces the document { country: "Spain" } with document { country: "Finland" }

```
db.countries.replaceOne(  
  { country: "Spain" },  
  { country: "Finland" }  
)
```

And returns:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

The example collection **countries** now contains:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Finland" }
```

Section 70.6: Deleting

Deleting documents from a collection in mongoose is done in the following manner.

```
Auto.remove({_id:123}, function(err, result){  
  if (err) return console.error(err);  
  console.log(result); // this will specify the mongo default delete result.  
});
```

Chapter 71: Passport integration

Section 71.1: Local authentication

The **passport-local** module is used to implement a local authentication.

This module lets you authenticate using a username and password in your Node.js applications.

Registering the user :

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
  // Overriding defaults expected parameters,
  // which are 'username' and 'password'
  usernameField: 'email',
  passwordField: 'password',
  passReqToCallback: true // allows us to pass back the entire request to the callback
},
function(req, email, password, next) {
  // Check in database if user is already registered
  findUserByEmail(email, function(user) {
    // If email already exists, abort registration process and
    // pass 'false' to the callback
    if (user) return next(null, false);
    // Else, we create the user
    else {
      // Password must be hashed !
      let newUser = createUser(email, password);

      newUser.save(function() {
        // Pass the user to the callback
        return next(null, newUser);
      });
    }
  });
});
```

Logging in the user :

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
  usernameField : 'email',
  passwordField : 'password',
},
function(email, password, next) {
  // Find the user
  findUserByEmail(email, function(user) {
    // If user is not found, abort signing in process
    // Custom messages can be provided in the verify callback
    // to give the user more details concerning the failed authentication
    if (!user)
```

```

        return next(null, false, {message: 'This e-mail address is not associated with any
account.'});
    // Else, we check if password is valid
    else {
        // If password is not correct, abort signing in process
        if (!isPasswordValid(password)) return next(null, false);
        // Else, pass the user to callback
        else return next(null, user);
    }
    });
});

```

Creating routes :

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
}));

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
}));

// Call req.logout() to log out
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);

```

Section 71.2: Getting started

Passport must be initialized using `passport.initialize()` middleware. To use login sessions, `passport.session()` middleware is required.

Note that `passport.serialize()` and `passport.deserializeUser()` methods must be defined. **Passport** will serialize and deserialize user instances to and from the session

```

const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // Serialize the user in the session
    next(null, user);
});

```

```

passport.deserializeUser(function(user, next) {
  // Use the previously serialized user
  next(null, user);
});

// Configuring express-session middleware
app.use(session({
  secret: 'The cake is a lie',
  resave: true,
  saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);

```

Section 71.3: Facebook authentication

The **passport-facebook** module is used to implement a **Facebook** authentication. In this example, if the user does not exist on sign-in, he is created.

Implementing strategy :

```

const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
  clientID: 'yourclientid',
  clientSecret: 'yourclientsecret',
  callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {
  // Check in database if user is already registered
  findUserByFacebookId(profile.id, function(user) {
    // If user exists, returns his data to callback
    if (user) return next(null, user);
    // Else, we create the user
    else {
      let newUser = createUserFromFacebook(profile, token);

      newUser.save(function() {
        // Pass the user to the callback
        return next(null, newUser);
      });
    }
  });
});

```

Creating routes :

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route

```

```

app.get('/auth/facebook', passport.authenticate('facebook', {
  // Ask Facebook for more permissions
  scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
  passport.authenticate('facebook', {
    successRedirect : '/me',
    failureRedirect : '/'
  }));

//...

app.listen(3000);

```

Section 71.4: Simple Username-Password Authentication

In your routes/index.js

Here user is the model for the userSchema

```

router.post('/login', function(req, res, next) {
  if (!req.body.username || !req.body.password) {
    return res.status(400).json({
      message: 'Please fill out all fields'
    });
  }

  passport.authenticate('local', function(err, user, info) {
    if (err) {
      console.log("ERROR : " + err);
      return next(err);
    }

    if(user) {
      console.log("User Exists!")
      //All the data of the user can be accessed by user.x
      res.json({"success" : true});
      return;
    } else {
      res.json({"success" : false});
      console.log("Error" + errorResponse());
      return;
    }
  })(req, res, next);
});

```

Section 71.5: Google Passport authentication

We have simple module available in npm for google authentication name **passport-google-oauth20**

Consider the following example In this example have created a folder namely config having the passport.js and google.js file in the root directory. In your app.js include the following

```

var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed

```

```
var app = express();
passport(app);
```

// other code to initialize the server , error handle

In the passport.js file in the config folder include the following code

```
var passport = require ('passport'),
google = require('./google'),
User = require('../model/user'); // User is the mongoose model

module.exports = function(app){
  app.use(passport.initialize());
  app.use(passport.session());
  passport.serializeUser(function(user, done){
    done(null, user);
  });
  passport.deserializeUser(function (user, done) {
    done(null, user);
  });
  google();
};
```

In the google.js file in the same config folder include following

```
var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('../model/user');
module.exports = function () {
  passport.use(new GoogleStrategy({
    clientID: 'CLIENT ID',
    clientSecret: 'CLIENT SECRET',
    callbackURL: "http://localhost:3000/auth/google/callback"
  },
  function(accessToken, refreshToken, profile, cb) {
    User.findOne({ googleId : profile.id }, function (err, user) {
      if(err){
        return cb(err, false, {message : err});
      }else {
        if (user != '' && user != null) {
          return cb(null, user, {message : "User "});
        } else {
          var username = profile.displayName.split(' ');
          var userData = new User({
            name : profile.displayName,
            username : username[0],
            password : username[0],
            facebookId : '',
            googleId : profile.id,
          });
          // send email to user just in case required to send the newly created
          // credentails to user for future login without using google login
          userData.save(function (err, newuser) {
            if (err) {
              return cb(null, false, {message : err + " !!! Please try again"});
            }else{
              return cb(null, newuser);
            }
          });
        }
      }
    });
  })
}
```



```
    });  
  }  
  ));  
};
```

Here in this example, if user is not in DB then creating a new user in DB for local reference using the field name `googleId` in user model.

Chapter 72: Dependency Injection

Section 72.1: Why Use Dependency Injection

1. **Fast Development process**
2. **Decoupling**
3. **Unit test writing**

Fast Development process

When using dependency injection node developer can faster their development proceess because after DI there is less code conflict and easy to manage all module.

Decoupling

Modules becomes less couple then it is easy to maintain.

Unit test writing

Hardcoded dependencies can pass them into the module then easy to write unit test for each module.

Chapter 73: NodeJS Beginner Guide

Section 73.1: Hello World !

Place the following code into a file name `helloworld.js`

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

Save the file, and execute it through Node.js:

```
node helloworld.js
```

Chapter 74: Use Cases of Node.js

Section 74.1: HTTP server

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

console.log('Starting server...');
var config = {
  port: 80,
  contentType: 'application/json; charset=utf-8'
};
// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
  if (err.code === 'EADDRINUSE') console.error('Port ' + config.port + ' is already in use');
  else console.error(err.message);
});
server.on('request', (request, res) => {
  var remoteAddress = request.headers['x-forwarded-for'] || request.connection.remoteAddress; //
  Client address
  console.log(remoteAddress + ' ' + request.method + ' ' + request.url);

  var out = {};
  // Here you can change output according to `request.url`
  out.test = request.url;
  res.writeHead(200, {
    'Content-Type': config.contentType
  });
  res.end(JSON.stringify(out));
});
server.on('listening', () => {
  c.info('Server is available: http://localhost:' + config.port);
});
```

Section 74.2: Console with command prompt

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
  promptPrefix: ' > '
}

/*
  Commands recognition
  BEGIN
*/
var commands = {
  eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
    arg = arg.join(' ');
    try { console.log(eval(arg)); }
    catch (e) { console.log(e); }
  },
  exit: function(arg) {
```

```

        process.exit();
    }
};
rl.on('line', (str) => {
    rl.pause();
    var arg = str.trim().match(/([^\"]+)/("?:[^\\"\\]|\\.)+"/g); // Applying regular expression for
removing all spaces except for what between double quotes:
http://stackoverflow.com/a/14540319/2396907
    if (arg) {
        for (let n in arg) {
            arg[n] = arg[n].replace(/^\"/\"$/g, ' ');
        }
        var commandName = arg[0];
        var command = commands[commandName];
        if (command) {
            arg.shift();
            command(arg);
        }
        else console.log('Command "' + commandName + '" doesn\'t exist');
    }
    rl.prompt();
});
/*
    END OF
    Commands recognition
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();

```

Chapter 75: Sequelize.js

Section 75.1: Defining Models

There are two ways to define models in sequelize; with `sequelize.define(...)`, or `sequelize.import(...)`. Both functions return a sequelize model object.

1. `sequelize.define(modelName, attributes, [options])`

This is the way to go if you'd like to define all your models in one file, or if you want to have extra control of your model definition.

```
/* Initialize Sequelize */ const config = { username: "database username", password: "database password",
database: "database name", host: "database's host URL", dialect: "mysql" // Other options are postgres, sqlite,
mariadb and mssql. } var Sequelize = require("sequelize"); var sequelize = new Sequelize(config); /* Define Models
*/ sequelize.define("MyModel", { name: Sequelize.STRING, comment: Sequelize.TEXT, date: { type: Sequelize.DATE,
allowNull: false } });
```

For the documentation and more examples, check out the [doclets documentation](#), or [sequelize.com's documentation](#).

2. `sequelize.import(path)`

If your model definitions are broken into a file for each, then `import` is your friend. In the file where you initialize Sequelize, you need to call import like so:

```
/* Initialize Sequelize */ // Check previous code snippet for initialization /* Define Models */
sequelize.import("../models/my_model.js"); // The path could be relative or absolute
```

Then in your model definition files, your code will look something like this:

```
module.exports = function(sequelize, DataTypes) { return sequelize.define("MyModel", { name: DataTypes.STRING,
comment: DataTypes.TEXT, date: { type: DataTypes.DATE, allowNull: false } }); };
```

For more information on how to use `import`, check out sequelize's [express example on GitHub](#).

Section 75.2: Installation

Make sure that you first have Node.js and npm installed. Then install sequelize.js with npm

```
npm install --save sequelize
```

You will also need to install supported database Node.js modules. You only need to install the one you are using

For MySQL and Mariadb

```
npm install --save mysql
```

For PostgreSQL

```
npm install --save pg pg-hstore
```

For SQLite

```
npm install --save sqlite
```

For MSSQL

```
npm install --save tedious
```

Once you have you set up installed you can include and create a new Sequelize instance like so.

ES5 syntax

```
var Sequelize = require('sequelize');  
var sequelize = new Sequelize('database', 'username', 'password');
```

ES6 stage-0 Babel syntax

```
import Sequelize from 'sequelize';  
const sequelize = new Sequelize('database', 'username', 'password');
```

You now have an instance of sequelize available. You could if you so feel inclined call it a different name such as

```
var db = new Sequelize('database', 'username', 'password');
```

or

```
var database = new Sequelize('database', 'username', 'password');
```

that part is your prerogative. Once you have this installed you can use it inside of your application as per the API documentation <http://docs.sequelizejs.com/en/v3/api/sequelize/>

Your next step after install would be to [set up your own model](#)

Chapter 76: PostgreSQL integration

Section 76.1: Connect To PostgreSQL

Using PostgreSQL npm module.

install dependency from npm

```
npm install pg --save
```

Now you have to create a PostgreSQL connection, which you can later query.

Assume you Database_Name = students, Host = localhost and DB_User= postgres

```
var pg = require("pg")
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

Section 76.2: Query with Connection Object

If you want to use connection object for query database you can use this sample code.

```
var queryString = "SELECT name, age FROM students " ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
  result.addRow(row);
});

query.on("end", function (result) {
  //LOGIC
});
```


Chapter 77: How modules are loaded

Section 77.1: Global Mode

If you installed Node using the default directory, while in the global mode, NPM installs packages into `/usr/local/lib/node_modules`. If you type the following in the shell, NPM will search for, download, and install the latest version of the package named `sax` inside the directory `/usr/local/lib/node_modules/express`:

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

Make sure that you have sufficient access rights to the folder. These modules will be available for all node process which will be running in that machine

In local mode installation. Npm will download and install modules in the current working folders by creating a new folder called `node_modules` for example if you are in `/home/user/apps/my_app` a new folder will be created called `node_modules` `/home/user/apps/my_app/node_modules` if its not already exist

Section 77.2: Loading modules

When we refer the module in the code, node first looks up the `node_module` folder inside the referenced folder in required statement. If the module name is not relative and is not a core module, Node will try to find it inside the `node_modules` folder in the current directory. For instance, if you do the following, Node will try to look for the file `./node_modules/myModule.js`:

```
var myModule = require('myModule.js');
```

If Node fails to find the file, it will look inside the parent folder called `../node_modules/myModule.js`. If it fails again, it will try the parent folder and keep descending until it reaches the root or finds the required module.

You can also omit the `.js` extension if you like to, in which case node will append the `.js` extension and will search for the file.

Loading a Folder Module

You can use the path for a folder to load a module like this:

```
var myModule = require('./myModuleDir');
```

If you do so, Node will search inside that folder. Node will presume this folder is a package and will try to look for a package definition. That package definition should be a file named `package.json`. If that folder does not contain a package definition file named `package.json`, the package entry point will assume the default value of `index.js`, and Node will look, in this case, for a file under the path `./myModuleDir/index.js`.

The last resort if module is not found in any of the folders is the global module installation folder.

Chapter 78: Node.js with Oracle

Section 78.1: Connect to Oracle DB

A very easy way to connect to an ORACLE database is by using [oracledb](#) module. This module handles the connection between your Node.js app and Oracle server. You can install it like any other module:

```
npm install oracledb
```

Now you have to create an ORACLE connection, which you can later query.

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

oracledb.getConnection(
  {
    user          : "oli",
    password      : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
  },
  connExecute
);
```

The connectString "ORACLE_DEV_DB_TNA_NAME" may live in a tnsnames.org file in the same directory or where your oracle instant client is installed.

If you don't have any oracle instant client installed on you development machine you may follow the [instant client installation guide](#) for your operating system.

Section 78.2: Using a local module for easier querying

To simplify your querying from ORACLE-DB, you may want to call your query like this:

```
const oracle = require('./oracle.js');

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
  .then(function(result) {
    console.log(result.rows[0]['C2']);
  })
  .catch(function(err) {
    next(err);
  });
```

Building up the connection and executing is included in this oracle.js file with content as follows:

```
'use strict';
const oracledb = require('oracledb');

const oracleDbRelease = function(conn) {
  conn.release(function (err) {
    if (err)
      console.log(err.message);
  });
};

function queryArray(sql, bindParams, options) {
```

```

options.isAutoCommit = false; // we only do SELECTs

return new Promise(function(resolve, reject) {
  oracledb.getConnection(
    {
      user      : "oli",
      password  : "password",
      connectString : "ORACLE_DEV_DB_TNA_NAME"
    }
  )
  .then(function(connection){
    //console.log("sql log: " + sql + " params " + bindParams);
    connection.execute(sql, bindParams, options)
    .then(function(results) {
      resolve(results);
      process.nextTick(function() {
        oracleDbRelease(connection);
      });
    })
    .catch(function(err) {
      reject(err);

      process.nextTick(function() {
        oracleDbRelease(connection);
      });
    })
  })
  .catch(function(err) {
    reject(err);
  });
});
}

function queryObject(sql, bindParams, options) {
  options['outFormat'] = oracledb.OBJECT; // default is oracledb.ARRAY
  return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;

```

Note that you have both methods `queryArray` and `queryObject` to call on your oracle object.

Section 78.3: Query a connection object without parameters

Use may now use the `connExecute`-Function for executing a query. You have the option to get the query result as an object or array. The result ist printed to `console.log`.

```

function connExecute(err, connection)
{
  if (err) {
    console.error(err.message);
    return;
  }
  sql = "select 'test' as c1, 'oracle' as c2 from dual";
  connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
    function(err, result)
    {
      if (err) {
        console.error(err.message);
        connRelease(connection);
      }
    }
  );
}

```

```

        return;
    }
    console.log(result.metaData);
    console.log(result.rows);
    connRelease(connection);
});
}

```

Since we used a non-pooling connection, we have to release our connection again.

```

function connRelease(connection)
{
    connection.close(
        function(err) {
            if (err) {
                console.error(err.message);
            }
        });
}

```

The output for an object will be

```

[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]

```

and the output for an array will be

```

[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]

```

Chapter 79: Synchronous vs Asynchronous programming in nodejs

Section 79.1: Using async

The [async package](#) provides functions for asynchronous code.

Using the [auto](#) function you can define asynchronous relations between two or more functions:

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

async.auto({
  get_data: function(callback) {
    console.log('in get_data');
    // async code to get some data
    callback(null, 'data', 'converted to array');
  },
  make_folder: function(callback) {
    console.log('in make_folder');
    // async code to create a directory to store a file in
    // this is run at the same time as getting the data
    callback(null, 'folder');
  },
  write_file: ['get_data', 'make_folder', function(results, callback) {
    console.log('in write_file', JSON.stringify(results));
    // once there is some data and the directory exists,
    // write the data to a file in the directory
    callback(null, 'filename');
  }],
  email_link: ['write_file', function(results, callback) {
    console.log('in email_link', JSON.stringify(results));
    // once the file is written let's email a link to it...
    // results.write_file contains the filename returned by write_file.
    callback(null, {'file':results.write_file, 'email':'user@example.com'});
  }]
}, function(err, results) {
  console.log('err = ', err);
  console.log('results = ', results);
});
```

This code could have been made synchronously, by just calling the `get_data`, `make_folder`, `write_file` and `email_link` in the correct order. Async keeps track of the results for you, and if an error occurred (first parameter of callback unequal to `null`) it stops the execution of the other functions.

Chapter 80: Node.js Error Management

We will learn how to create Error objects and how to throw & handle errors in Node.js

Future edits related to best practices in error handling.

Section 80.1: try...catch block

try...catch block is for handling exceptions, remember exception means the thrown error not the error.

```
try {
  var a = 1;
  b++; //this will cause an error because b is undefined
  console.log(b); //this line will not be executed
} catch (error) {
  console.log(error); //here we handle the error caused in the try block
}
```

In the **try** block **b++** cause an error and that error passed to **catch** block which can be handled there or even can be thrown the same error in catch block or make little bit modification then throw. Let's see next example.

```
try {
  var a = 1;
  b++;
  console.log(b);
} catch (error) {
  error.message = "b variable is undefined, so the undefined can't be incremented"
  throw error;
}
```

In the above example we modified the message property of error object and then throw the modified error.

You can throw any error in your try block and handle it in the catch block:

```
try {
  var a = 1;
  throw new Error("Some error message");
  console.log(a); //this line will not be executed
} catch (error) {
  console.log(error); //will be the above thrown error
}
```

Section 80.2: Creating Error object

new Error(message)

Creates new error object, where the value message is being set to message property of the created object. Usually the message arguments are being passed to Error constructor as a string. However if the message argument is object not a string then Error constructor calls **.toString()** method of the passed object and sets that value to message property of the created error object.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
```

```
// at ...
```

Each error object has stack trace. Stack trace contains the information of error message and shows where the error happened (the above output shows the error stack). Once error object is created the system captures the stack trace of the error on current line. To get the stack trace use stack property of any created error object. Below two lines are identical:

```
console.log(err);  
console.log(err.stack);
```

Section 80.3: Throwing Error

Throwing error means exception if any exception is not handled then the node server will crash.

The following line throws error:

```
throw new Error("Some error occurred");
```

or

```
var err = new Error("Some error occurred");  
throw err;
```

or

```
throw "Some error occurred";
```

The last example (throwing strings) is not good practice and is not recommended (always throw errors which are instances of Error object).

Note that if you **throw** an error in your, then the system will crash on that line (if there is no exception handlers), no any code will be executed after that line.

```
var a = 5;  
var err = new Error("Some error message");  
throw err; //this will print the error stack and node server will stop  
a++; //this line will never be executed  
console.log(a); //and this one also
```

But in this example:

```
var a = 5;  
var err = new Error("Some error message");  
console.log(err); //this will print the error stack  
a++;  
console.log(a); //this line will be executed and will print 6
```

Chapter 81: Node.js v6 New Features and Improvement

With node 6 becoming the new LTS version of node. We can see an number of improvements to the language through the new ES6 standards introduces. We'll be walking through some of the new features introduced and examples of how to implement them.

Section 81.1: Default Function Parameters

```
function addTwo(a, b = 2) {  
    return a + b;  
}  
  
addTwo(3) // Returns the result 5
```

With the addition of default function parameters you can now make arguments optional and have them default to a value of your choice.

Section 81.2: Rest Parameters

```
function argumentLength(...args) {  
    return args.length;  
}  
  
argumentLength(5) // returns 1  
argumentLength(5, 3) //returns 2  
argumentLength(5, 3, 6) //returns 3
```

By prefacing the last argument of your function with `...` all arguments passed to the function are read as an array. In this example we get pass in multiple arguments and get the length of the array created from those arguments.

Section 81.3: Arrow Functions

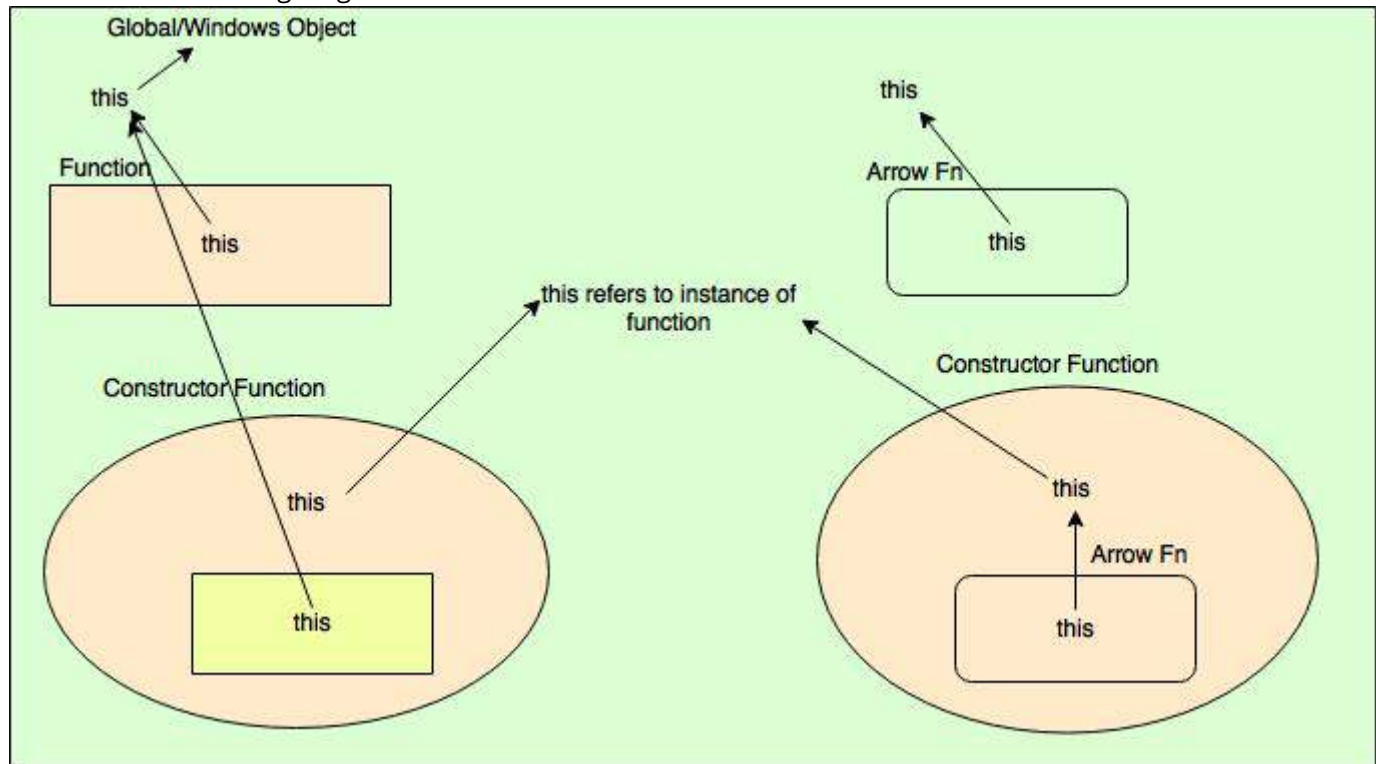
Arrow function is the new way of defining a function in ECMAScript 6.

```
// traditional way of declaring and defining function  
var sum = function(a,b)  
{  
    return a+b;  
}  
  
// Arrow Function  
let sum = (a, b)=> a+b;  
  
//Function defination using multiple lines  
let checkIfEven = (a) => {  
    if( a % 2 == 0 )  
        return true;  
    else  
        return false;  
}
```


Section 81.4: "this" in Arrow Function

this in function refers to instance object used to call that function but **this** in arrow function is equal to *this* of function in which arrow function is defined.

Let's understand using diagram



Understanding using examples.

```
var normalFn = function(){
  console.log(this) // refers to global/window object.
}

var arrowFn = () => console.log(this); // refers to window or global object as function is defined
in scope of global/window object

var service = {
  constructorFn : function(){
    console.log(this); // refers to service as service object used to call method.

    var nestedFn = function(){
      console.log(this); // refers window or global object because no instance object was used
to call this method.
    }
    nestedFn();
  },
  arrowFn : function(){
    console.log(this); // refers to service as service object was used to call method.
    let fn = () => console.log(this); // refers to service object as arrow function defined in
function which is called using instance object.
    fn();
  }
}
```

```
// calling defined functions
constructorFn();
arrowFn();
service.constructorFn();
service.arrowFn();
```

In arrow function, *this* is lexical scope which is the scope of function where arrow function is defined. The first example is the traditional way of defining functions and hence, *this* refers to *global/window* object. In the second example *this* is used inside arrow function hence *this* refers to the scope where it is defined (which is windows or global object). In the third example *this* is service object as service object is used to call the function. In fourth example, arrow function is defined and called from the function whose scope is *service*, hence it prints *service* object.

Note: - global object is printed in Node.js and windows object in browser.

Section 81.5: Spread Operator

```
function myFunction(x, y, z) { }
var args = [0, 1, 2];
myFunction(...args);
```

The spread syntax allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables are expected. Just like the rest parameters simply preface your array with `...`

Chapter 82: Eventloop

In this post we are going to discuss how the concept of Eventloop emerged and how it can be used for high performance servers and event driven applications like GUIs.

Section 82.1: How the concept of event loop evolved

Eventloop in pseudo code

An event loop is a loop that waits for events and then reacts to those events

```
while true:
    wait for something to happen
    react to whatever happened
```

Example of a single-threaded HTTP server with no event loop

```
while true:
    socket = wait for the next TCP connection
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
```

Here's a simple form of a HTTP server which is a single threaded but no event loop. The problem here is that it waits until each request is finished before starting to process the next one. If it takes a while to read the HTTP request headers or to fetch the file from disk, we should be able to start processing the next request while we wait for that to finish.

The most common solution is to make the program multi-threaded.

Example of a multi-threaded HTTP server with no event loop

```
function handle_connection(socket):
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
while true:
    socket = wait for the next TCP connection
    spawn a new thread doing handle_connection(socket)
```

Now we have made our little HTTP server multi threaded. This way, we can immediately move on to the next request because the current request is running in a background thread. Many servers, including Apache, use this approach.

But it's not perfect. One limitation is that you can only spawn so many threads. For workloads where you have a huge number of connections, but each connection only requires attention every once in a while, the multi-threaded model won't perform very well. The solution for those cases is to use an event loop:

Example of a HTTP server with event loop

```
while true:
    event = wait for the next event to happen
    if (event.type == NEW_TCP_CONNECTION):
        conn = new Connection
        conn.socket = event.socket
```

```

        start reading HTTP request headers from (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_SOCKET):
        conn = event.userdata
        start fetching the requested file from disk with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_DISK):
        conn = event.userdata
        conn.file_contents = the data we fetched from disk
        conn.current_state = "writing headers"
        start writing the HTTP response headers to (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_WRITING_TO_SOCKET):
        conn = event.userdata
        if (conn.current_state == "writing headers"):
            conn.current_state = "writing file contents"
            start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
        else if (conn.current_state == "writing file contents"):
            close(conn.socket)

```

Hopefully this pseudocode is intelligible. Here's what's going on: We wait for things to happen. Whenever a new connection is created or an existing connection needs our attention, we go deal with it, then go back to waiting. That way, we perform well when there are many connections and each one only rarely requires attention.

In a real application (not pseudocode) running on Linux, the "wait for the next event to happen" part would be implemented by calling the `poll()` or `epoll()` system call. The "start reading/writing something to a socket" parts would be implemented by calling the `recv()` or `send()` system calls in non-blocking mode.

Reference:

[1]. "How does an event loop work?" [Online]. Available : <https://www.quora.com/How-does-an-event-loop-work>

Chapter 83: Nodejs History

Here we are going to discuss about the history of Node.js, version information and it's current status.

Section 83.1: Key events in each year

2009

- 3rd March : [The project was named as "node"](#)
- 1st October : [First very early preview of npm, the Node package manager](#)
- 8th November : [Ryan Dahl's \(Creator of Node.js\) Original Node.js Talk at JSConf 2009](#)

2010

- Express: A Node.js web development framework
- Socket.io initial release
- 28th April : [Experimental Node.js Support on Heroku](#)
- 28th July : [Ryan Dahl's Google Tech Talk on Node.js](#)
- 20th August : [Node.js 0.2.0 released](#)

2011

- 31st March : Node.js Guide
- 1st May : [npm 1.0: Released](#)
- 1st May : [Ryan Dahl's AMA on Reddit](#)
- 10th July : [The Node Beginner Book, an introduction to Node.js, is completed.](#)
 - A comprehensive Node.js tutorial for beginners.
- 16th August : [LinkedIn uses Node.js](#)
 - LinkedIn launched its completely overhauled mobile app with new features and new parts under the hood.
- 5th October : [Ryan Dahl talks about the history of Node.js and why he created it](#)
- 5th December : [Node.js in production at Uber](#)
 - Uber Engineering Manager Curtis Chambers explains why his company completely re-engineered their application using Node.js to increase efficiency and improve the partner and customer experience.

2012

- 30th January : [Node.js creator Ryan Dahl steps away from Node's day-to-day](#)
- 25th June : [Node.js v0.8.0 \[stable\] is out](#)
- 20th December : [Hapi, a Node.js framework](#) is released

2013

- 30th April : [The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js](#)
- 17th May : [How We Built eBay's First Node.js Application](#)
- 15th November : [PayPal releases Kraken, a Node.js framework](#)
- 22nd November : [Node.js Memory Leak at Walmart](#)
 - Eran Hammer of Wal-Mart labs came to the Node.js core team complaining of a memory leak he had been tracking down for months.
- 19th December : Koa - Web framework for Node.js

2014

- 15th January : [TJ Fontaine takes over Node project](#)

- 23rd October : [Node.js Advisory Board](#)
 - Joyent and several members of the Node.js community announced a proposal for a Node.js Advisory Board as a next step towards a fully open governance model for the Node.js open source project.
- 19th November : [Node.js in Flame Graphs - Netflix](#)
- 28th November : [IO.js](#) – Evented I/O for V8 Javascript

2015

Q1

- 14th January : [IO.js 1.0.0](#)
- 10th February : [Joyent Moves to Establish Node.js Foundation](#)
 - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP and The Linux Foundation Join Forces to Support Node.js Community With Neutral and Open Governance
- 27th February : [IO.js and Node.js reconciliation proposal](#)

Q2

- 14th April : [npm Private Modules](#)
- 28th May : [Node lead TJ Fontaine is stepping down and leaving Joyent](#)
- 13th May : [Node.js and io.js are merging under the Node Foundation](#)

Q3

- 2nd August : [Trace - Node.js performance monitoring and debugging](#)
 - Trace is a visualized microservice monitoring tool that gets you all the metrics you need when operating microservices.
- 13th August : [4.0 is the new 1.0](#)

Q4

- 12th October : [Node v4.2.0, first Long Term Support release](#)
- 8th December : [Apigee, RisingStack and Yahoo join the Node.js Foundation](#)
- 8th & 9th December : [Node Interactive](#)
 - The first annual Node.js conference by the Node.js Foundation

2016

Q1

- 10th February : [Express becomes an incubated project](#)
- 23rd March : [The leftpad incident](#)
- 29th March : [Google Cloud Platform joins the Node.js Foundation](#)

Q2

- 26th April : [npm has 210.000 users](#)

Q3

- 18th July : [CJ Silverio becomes the CTO of npm](#)
- 1st August : [Trace, the Node.js debugging solution becomes generally available](#)
- 15th September : [The first Node Interactive in Europe](#)

Q4

- 11th October : [The yarn package manager got released](#)
- 18th October : [Node.js 6 becomes the LTS version](#)

Reference

1. "History of Node.js on a Timeline" [Online]. Available : [<https://blog.risingstack.com/history-of-node-js>]

Chapter 84: passport.js

Passport is a popular authorisation module for node. In simple words it handles all the authorisation requests on your app by users. Passport supports over 300 strategies so that you can easily integrate login with Facebook / Google or any other social network using it. The strategy that we will discuss here is the Local where you authenticate an user using your own database of registered users(using username and password).

Section 84.1: Example of LocalStrategy in passport.js

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

passport.serializeUser(function(user, done) { //In serialize user you decide what to store in the session. Here I'm storing the user id only.
  done(null, user.id);
});

passport.deserializeUser(function(id, done) { //Here you retrieve all the info of the user from the session storage using the user id stored in the session earlier using serialize user.
  db.findById(id, function(err, user) {
    done(err, user);
  });
});

passport.use(new LocalStrategy(function(username, password, done) {
  db.findOne({'username':username}, function(err, student){
    if(err) return done(err, {message:message}); //wrong roll_number or password;
    var pass_retrieved = student.pass_word;
    bcrypt.compare(password, pass_retrieved, function(err3, correct) {
      if(err3){
        message = [{"msg": "Incorrect Password!"}];
        return done(null, false, {message:message}); // wrong password
      }
      if(correct){
        return done(null, student);
      }
    });
  });
}));

app.use(session({ secret: 'super secret' })); //to make passport remember the user on other pages too.(Read about session store. I used express-sessions.)
app.use(passport.initialize());
app.use(passport.session());

app.post('/', passport.authenticate('local', {successRedirect:'/users' failureRedirect: '/'}),
  function(req, res, next){
  });
```


Chapter 85: Asynchronous programming

Node is a programming language where everything could run on an asynchronous way. Below you could find some examples and the typical things of asynchronous working.

Section 85.1: Callback functions

Callback functions in JavaScript

Callback functions are common in JavaScript. Callback functions are possible in JavaScript because [functions are first-class citizens](#).

Synchronous callbacks.

Callback functions can be synchronous or asynchronous. Since Asynchronous callback functions may be more complex here is a simple example of a synchronous callback function.

```
// a function that uses a callback named `cb` as a parameter
function getSyncMessage(cb) {
    cb("Hello World!");
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getSyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

The output for the above code is:

```
> Before getSyncMessage call
> Hello World!
> After getSyncMessage call
```

First we will step through how the above code is executed. This is more for those who do not already understand the concept of callbacks if you do already understand it feel free to skip this paragraph. First the code is parsed and then the first interesting thing to happen is line 6 is executed which outputs `Before getSyncMessage call` to the console. Then line 8 is executed which calls the function `getSyncMessage` sending in an anonymous function as an argument for the parameter named `cb` in the `getSyncMessage` function. Execution is now done inside the `getSyncMessage` function on line 3 which executes the function `cb` which was just passed in, this call sends an argument string `"Hello World"` for the param named `message` in the passed in anonymous function. Execution then goes to line 9 which logs `Hello World!` to the console. Then the execution goes through the process of exiting the [callstack](#) (see also) hitting line 10 then line 4 then finally back to line 11.

Some information to know about callbacks in general:

- The function you send in to a function as a callback may be called zero times, once, or multiple times. It all depends on implementation.
- The callback function may be called synchronously or asynchronously and possibly both synchronously and asynchronously.
- Just like normal functions the names you give parameters to your function are not important but the order is. So for example on line 8 the parameter `message` could have been named `statement`, `msg`, or if you're being nonsensical something like `jellybean`. So you should know what parameters are sent into your callback so

you can get them in the right order with proper names.

Asynchronous callbacks.

One thing to note about JavaScript is it is synchronous by default, but there are APIs given in the environment (browser, Node.js, etc.) that could make it asynchronous (there's more about that [here](#)).

Some common things that are asynchronous in JavaScript environments that accept callbacks:

- Events
- setTimeout
- setInterval
- the fetch API
- Promises

Also any function that uses one of the above functions may be wrapped with a function that takes a callback and the callback would then be an asynchronous callback (although wrapping a promises with a function that takes a callback would likely be considered an anti-pattern as there are more preferred ways to handle promises).

So given that information we can construct an asynchronous function similar to the above synchronous one.

```
// a function that uses a callback named `cb` as a parameter
function getAsyncMessage(cb) {
  setTimeout(function () { cb("Hello World!") }, 1000);
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getAsyncMessage(function(message) {
  console.log(message);
});
console.log("After getSyncMessage call");
```

Which prints the following to the console:

```
> Before getSyncMessage call
> After getSyncMessage call
// pauses for 1000 ms with no output
> Hello World!
```

Line execution goes to line 6 logs "Before getSyncMessage call". Then execution goes to line 8 calling getAsyncMessage with a callback for the param cb. Line 3 is then executed which calls setTimeout with a callback as the first argument and the number 300 as the second argument. setTimeout does whatever it does and holds on to that callback so that it can call it later in 1000 milliseconds, but following setting up the timeout and before it pauses the 1000 milliseconds it hands execution back to where it left off so it goes to line 4, then line 11, and then pauses for 1 second and setTimeout then calls its callback function which takes execution back to line 3 where getAsyncMessages callback is called with value "Hello World" for its parameter message which is then logged to the console on line 9.

Callback functions in Node.js

NodeJS has asynchronous callbacks and commonly supplies two parameters to your functions sometimes conventionally called err and data. An example with reading a file text.

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

```
fs.readFile("./test.txt", "utf8", function(err, data) {
  if(err) {
    // handle the error
  } else {
    // process the file text given with data
  }
});
```

This is an example of a callback that is called a single time.

It's good practice to handle the error somehow even if your just logging it or throwing it. The else is not necessary if you throw or return and can be removed to decrease indentation so long as you stop execution of the current function in the if by doing something like throwing or returning.

Though it may be common to see err, data it may not always be the case that your callbacks will use that pattern it's best to look at documentation.

Another example callback comes from the express library (express 4.x):

```
// this code snippet was on http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// this app.get method takes a url route to watch for and a callback
// to call whenever that route is requested by a user.
app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);
```

This example shows a callback that is called multiple times. The callback is provided with two objects as params named here as req and res these names correspond to request and response respectively, and they provide ways to view the request coming in and set up the response that will be sent to the user.

As you can see there are various ways a callback can be used to execute sync and async code in JavaScript and callbacks are very ubiquitous throughout JavaScript.

Section 85.2: Callback hell

Callback hell (also a pyramid of doom or boomerang effect) arises when you nest too many callback functions inside a callback function. Here is an example to read a file (in ES6).

```
const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
  if (exists) {
    fs.stat(filename, (err, stats) => {
      if (err) {
        throw err;
      }
      if (stats.isFile()) {
        fs.readFile(filename, null, (err, data) => {
          if (err) {
            throw err;
          }
          console.log(data);
        });
      }
    });
  }
});
```

```

        });
    }
    else {
        throw new Error("This location contains not a file");
    }
});
}
else {
    throw new Error("404: file not found");
}
});

```

How to avoid "Callback Hell"

It is recommended to nest no more than 2 callback functions. This will help you maintain code readability and will be much easier to maintain in the future. If you have a need to nest more than 2 callbacks, try to make use of [distributed events](#) instead.

There also exists a library called [async](#) that helps manage callbacks and their execution available on npm. It increases the readability of callback code and gives you more control over your callback code flow, including allowing you to run them in parallel or in series.

Section 85.3: Native Promises

Version ≥ v6.0.0

Promises are a tool for async programming. In JavaScript promises are known for their then methods. Promises have two main states 'pending' and 'settled'. Once a promise is 'settled' it cannot go back to 'pending'. This means that promises are mostly good for events that only occur once. The 'settled' state has two states as well 'resolved' and 'rejected'. You can create a new promise using the **new** keyword and passing a function into the constructor **new Promise(function (resolve, reject) {})**.

The function passed into the Promise constructor always receives a first and second parameter usually named resolve and reject respectively. The naming of these two parameters is convention, but they will put the promise into either the 'resolved' state or the 'rejected' state. When either one of these is called the promise goes from being 'pending' to 'settled'. resolve is called when the desired action, which is often asynchronous, has been performed and reject is used if the action has errored.

In the below timeout is a function that returns a Promise.

```

function timeout (ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve("It was resolved!");
        }, ms)
    });
}

timeout(1000).then(function (dataFromPromise) {
    // logs "It was resolved!"
    console.log(dataFromPromise);
})

console.log("waiting...");

```

console output

```
waiting...  
// << pauses for one second>>  
It was resolved!
```

When `timeout` is called the function passed to the `Promise` constructor is executed without delay. Then the `setTimeout` method is executed and its callback is set to fire in the next `ms` milliseconds, in this case `ms=1000`. Since the callback to the `setTimeout` isn't fired yet the `timeout` function returns control to the calling scope. The chain of then methods are then stored to be called later when/if the `Promise` has resolved. If there were `catch` methods here they would be stored as well, but would be fired when/if the promise 'rejects'.

The script then prints 'waiting...'. One second later the `setTimeout` calls its callback which calls the `resolve` function with the string "It was resolved!". That string is then passed into the `then` method's callback and is then logged to the user.

In the same sense you can wrap the asynchronous `setTimeout` function which requires a callback you can wrap any singular asynchronous action with a promise.

Read more about promises in the JavaScript documentation [Promises](#).

Section 85.4: Code example

Question: What is the output of code below and why?

```
setTimeout(function() {  
    console.log("A");  
}, 1000);  
  
setTimeout(function() {  
    console.log("B");  
}, 0);  
  
getDataFromDatabase(function(err, data) {  
    console.log("C");  
    setTimeout(function() {  
        console.log("D");  
    }, 1000);  
});  
  
console.log("E");
```

Output: This is known for sure: EBAD. C is unknown when it will be logged.

Explanation: The compiler will not stop on the `setTimeout` and the `getDataFromDatabase` methodes. So the first line he will log is E. The callback functions (*first argument of `setTimeout`*) will run after the set timeout on a asynchronous way!

More details:

1. E has no `setTimeout`
2. B has a set timeout of 0 milliseconds
3. A has a set timeout of 1000 milliseconds
4. D must request a database, after it must D wait 1000 milliseconds so it comes after A.
5. C is unknown because it is unknown when the data of the database is requested. It could be before or after A.

Section 85.5: Async error handling

Try catch

Errors must always be handled. If you are using synchronous programming you could use a **try catch**. But this does not work if you work asynchronous! Example:

```
try {
  setTimeout(function() {
    throw new Error("I'm an uncaught error and will stop the server!");
  }, 100);
}
catch (ex) {
  console.error("This error will not be work in an asynchronous situation: " + ex);
}
```

Async errors will only be handled inside the callback function!

Working possibilities

Version ≤ v0.8

Event handlers

The first versions of Node.js got an event handler.

```
process.on("UncaughtException", function(err, data) {
  if (err) {
    // error handling
  }
});
```

Version ≥ v0.8

Domains

Inside a domain, the errors are release via the event emitters. By using this are all errors, timers, callback methodes implicitly only registrated inside the domain. By an error, be an error event send and didn't crash the application.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
  d2.add(setTimeout(function() {
    throw new Error("error on the timer of domain 2");
  }, 0));
});

d1.on("error", function(err) {
  console.log("error at domain 1: " + err);
});

d2.on("error", function(err) {
  console.log("error at domain 2: " + err);
});
```

Chapter 86: Node.js code for STDIN and STDOUT without using any library

This is a simple program in node.js to which takes input from the user and prints it to the console.

The **process** object is a global that provides information about, and control over, the current Node.js process. As a global, it is always available to Node.js applications without using `require()`.

Section 86.1: Program

The **process.stdin** property returns a Readable stream equivalent to or associated with stdin.

The **process.stdout** property returns a Writable stream equivalent to or associated with stdout.

```
process.stdin.resume()
console.log('Enter the data to be displayed ');
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Chapter 87: MongoDB Integration for Node.js/Express.js

MongoDB is one of the most popular NoSQL databases, thanks to the help of the MEAN stack. Interfacing with a Mongo database from an Express app is quick and easy, once you understand the kinda-wonky query syntax. We'll use Mongoose to help us out.

Section 87.1: Installing MongoDB

```
npm install --save mongodb
npm install --save mongoose //A simple wrapper for ease of development
```

In your server file (normally named index.js or server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectionString = 'http://localhost/database name';

mongoose.connect(mongoConnectionString, (err) => {
  if (err) {
    console.log('Could not connect to the database');
  }
});
```

Section 87.2: Creating a Mongoose Model

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;

const Article = new Schema({
  title: {
    type: String,
    unique: true,
    required: [true, 'Article must have title']
  },
  author: {
    type: ObjectId,
    ref: 'User'
  }
});

module.exports = mongoose.model('Article', Article);
```

Let's dissect this. MongoDB and Mongoose use JSON(actually BSON, but that's irrelevant here) as the data format. At the top, I've set a few variables to reduce typing.

I create a **new** Schema and assign it to a constant. It's simple JSON, and each attribute is another Object with properties that help enforce a more consistent schema. Unique forces new instances being inserted in the database to, obviously, be unique. This is great for preventing a user creating multiple accounts on a service.

Required is another, declared as an array. The first element is the boolean value, and the second the error message should the value being inserted or updated fail to exist.

ObjectIds are used for relationships between Models. Examples might be 'Users have many Comments'. Other

attributes can be used instead of ObjectId. Strings like a username is one example.

Lastly, exporting the model for use with your API routes provides access to your schema.

Section 87.3: Querying your Mongo Database

A simple GET request. Let's assume the Model from the example above is in the file `./db/models/Article.js`.

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

  app.use('/api', routes);
};
```

We can now get the data from our database by sending an HTTP request to this endpoint. A few key things, though:

1. Limit does exactly what it looks like. I'm only getting 5 documents back.
2. Lean strips away some stuff from the raw BSON, reducing complexity and overhead. Not required. But useful.
3. When using `find` instead of `findOne`, confirm that the `doc.length` is greater than 0. This is because `find` always returns an array, so an empty array will not handle your error unless it is checked for length
4. I personally like to send the error message in that format. Change it to suit your needs. Same thing for the returned document.
5. The code in this example is written under the assumption that you have placed it in another file and not directly on the express server. To call this in the server, include these lines in your server code:

```
const app = express();
require('./path/to/this/file')(app) //
```

Chapter 88: Lodash

Lodash is a handy JavaScript utility library.

Section 88.1: Filter a collection

The code snippet below shows the various ways you can filter on an array of objects using lodash.

```
let lodash = require('lodash');

var countries = [
  {"key": "DE", "name": "Deutschland", "active": false},
  {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
  return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Chapter 89: csv parser in node js

Reading data in from a csv can be handled in many ways. One solution is to read the csv file into an array. From there you can do work on the array.

Section 89.1: Using FS to read in a CSV

fs is the [File System API](#) in node. We can use the method `readFile` on our fs variable, pass it a `data.csv` file, format and function that reads and splits the csv for further processing.

This assumes you have a file named `data.csv` in the same folder.

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

You can now use the array like any other to do work on it.

Chapter 90: Loopback - REST Based connector

Rest based connectors and how to deal with them. We all know Loopback does not provide elegance to REST based connections

Section 90.1: Adding a web based connector

```
//This example gets the response from iTunes { "rest": { "name": "rest", "connector": "rest", "debug": true, "options":  
{ "useQuerystring": true, "timeout": 10000, "headers": { "accepts": "application/json", "content-type":  
"application/json" } }, "operations": [ { "template": { "method": "GET", "url": "https://itunes.apple.com/search",  
"query": { "term": "{keyword}", "country": "{country=IN}", "media": "{itemType=music}", "limit": "{limit=10}",  
"explicit": "false" } }, "functions": { "search": [ "keyword", "country", "itemType", "limit" ] } }, { "template": { "method":  
"GET", "url": "https://itunes.apple.com/lookup", "query": { "id": "{id}" } }, "functions": { "findById": [ "id" ] } } ] }
```

Chapter 91: Running node.js as a service

Unlike many web servers, Node isn't installed as a service out of the box. But in production, it's better to have it run as a *dæmon*, managed by an init system.

Section 91.1: Node.js as a systemd *dæmon*

systemd is the *de facto* init system in most Linux distributions. After Node has been configured to run with systemd, it's possible to use the `service` command to manage it.

First of all, it needs a config file, let's create it. For Debian based distros, it will be in `/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log tot syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environement (dev, prod...)
Environment=NODE_ENV=production

[Install]
# start node at multi user system level (= sysVinit runlevel 3)
WantedBy=multi-user.target
```

It's now possible to respectively start, stop and restart the app with:

```
service node start
service node stop
service node restart
```

To tell systemd to automatically start node on boot, just type: `systemctl enable node`.

That's all, node now runs as a *dæmon*.

Chapter 92: Node.js with CORS

Section 92.1: Enable CORS in express.js

As node.js is often used to build API, proper CORS setting can be a life saver if you want to be able to request the API from different domains.

In the exemple, we'll set it up for the wider configuration (authorize all request types from any domain.

In your server.js after initializing express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');

  // authorized headers for preflight requests
  // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
  next();

  app.options('*', (req, res) => {
    // allowed XHR methods
    res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
    res.send();
  });
});
```

Usually, node is ran behind a proxy on production servers. Therefore the reverse proxy server (such as Apache or Nginx) will be responsible for the CORS config.

To conveniently adapt this scenario, it's possible to only enable node.js CORS when it's in development.

This is easily done by checking NODE_ENV:

```
const app = express();

if (process.env.NODE_ENV === 'development') {
  // CORS settings
}
```

Chapter 93: Getting started with Nodes profiling

The aim of this post is to get started with profiling nodejs application and how to make sense of this results to capture a bug or a memory leak. A nodejs running application is nothing but a v8 engine processes which is in many terms similar to a website running on a browser and we can basically capture all the metrics which are related to a website process for a node application.

The tool of my preference is chrome devtools or chrome inspector coupled with the node-inspector.

Section 93.1: Profiling a simple node application

Step 1 : Install the node-inspector package using npm globally on you machine

```
$ npm install -g node-inspector
```

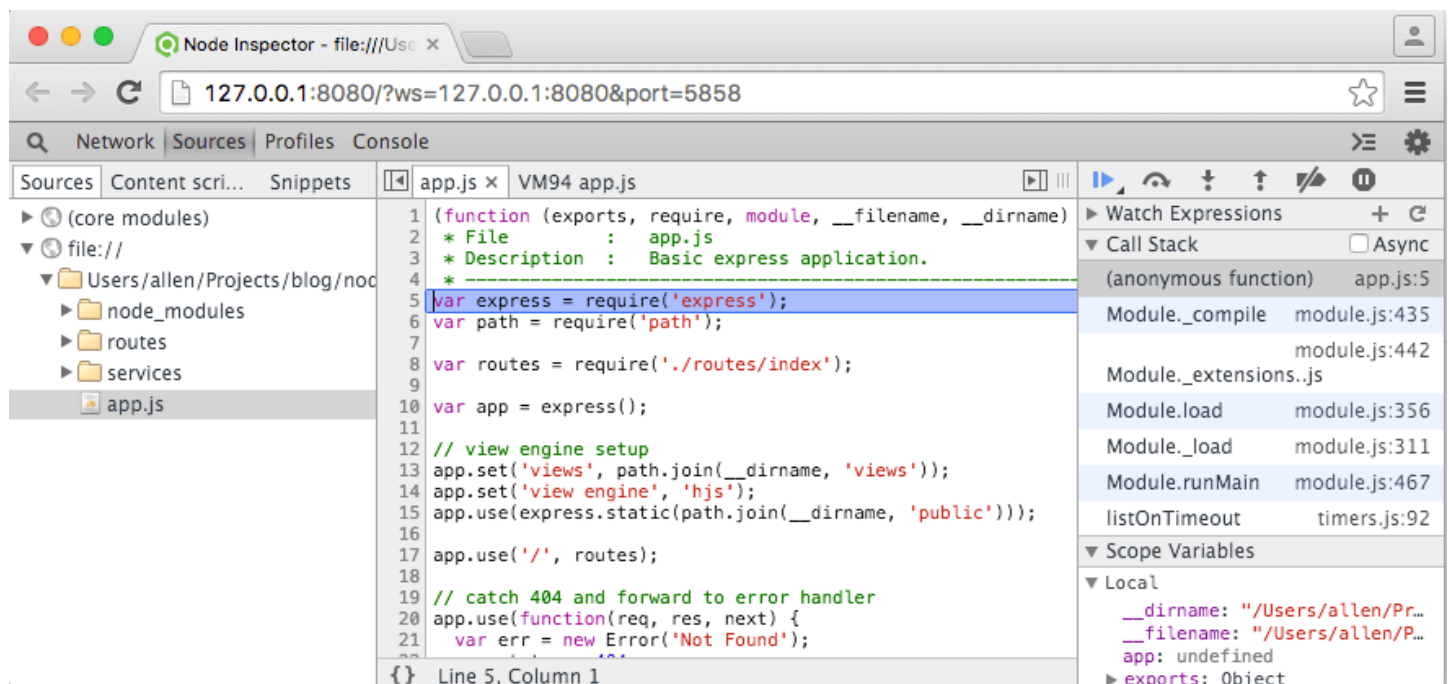
Step 2 : Start the node-inspector server

```
$ node-inspector
```

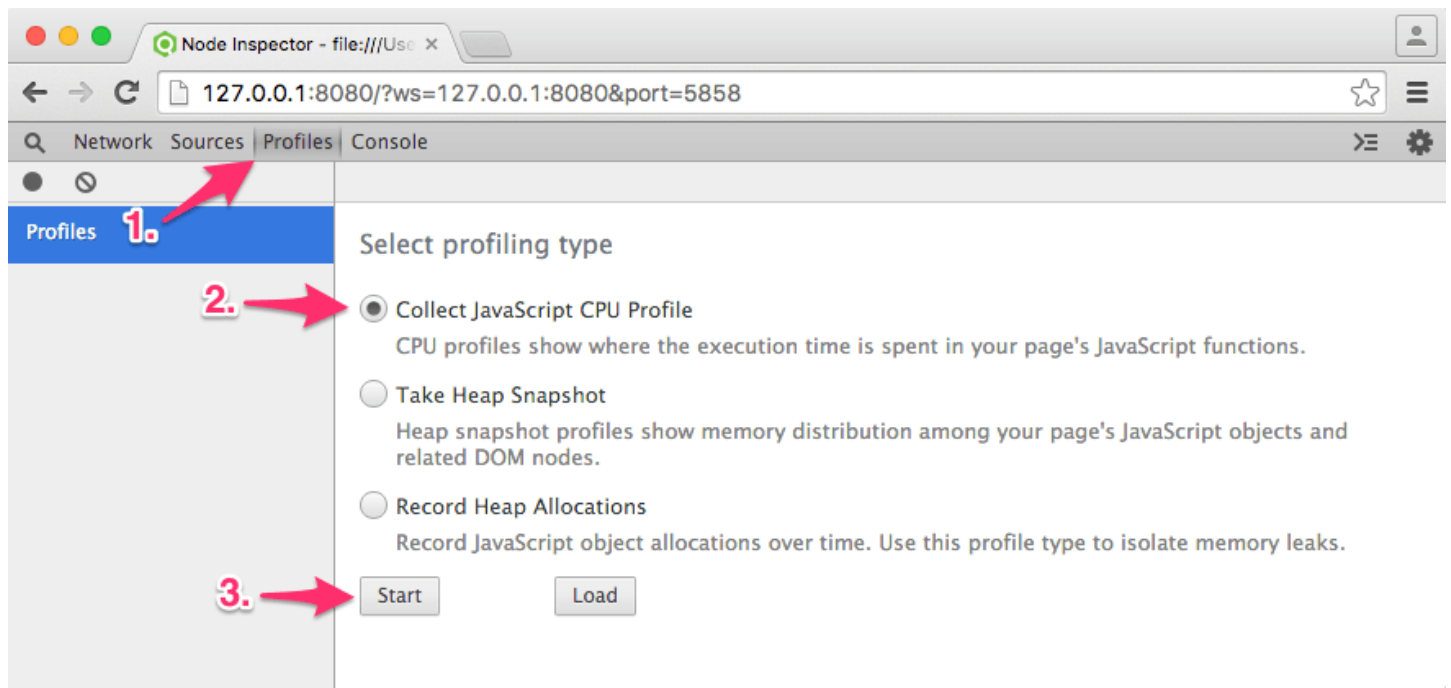
Step 3 : Start debugging your node application

```
$ node --debug-brk your/short/node/script.js
```

Step 4 : Open <http://127.0.0.1:8080/?port=5858> in the Chrome browser. And you will see a chrom-dev tools interface with your nodejs application source code in left panel . And since we have used debug break option while debugging the application the code execution will stop at the first line of code.



Step 5 : This is the easy part where you switch to the profiling tab and start profiling the application . In case you want get the profile for a particular method or flow make sure the code execution is break-pointed just before that piece of code is executed.



Step 6 : Once you have recorded your CPU profile or heap dump/snapshot or heap allocation you can then view the results in the same window or save them to local drive for later analysis or comparison with other profiles.

You can use this articles to know how to read the profiles :

- [Reading CPU Profiles](#)
- [Chrome CPU profiler and Heap profiler](#)

Chapter 94: Node.js Performance

Section 94.1: Enable gzip

```
const http = require('http')
const fs   = require('fs')
const zlib = require('zlib')

http.createServer((request, response) => {
  const stream      = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder      : false,
    contentEncoding: {},
    createEncoder   : () => throw 'There is no encoder'
  }

  if (!acceptsEncoding) {
    acceptsEncoding = ''
  }

  if (acceptsEncoding.match(/\bdeflate\b/)) {
    encoder = {
      hasEncoder      : true,
      contentEncoding: { 'content-encoding': 'deflate' },
      createEncoder   : zlib.createDeflate
    }
  } else if (acceptsEncoding.match(/\bgzip\b/)) {
    encoder = {
      hasEncoder      : true,
      contentEncoding: { 'content-encoding': 'gzip' },
      createEncoder   : zlib.createGzip
    }
  }

  response.writeHead(200, encoder.contentEncoding)

  if (encoder.hasEncoder) {
    stream = stream.pipe(encoder.createEncoder())
  }

  stream.pipe(response)
}).listen(1337)
```

Section 94.2: Event Loop

Blocking Operation Example

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// This operation will block Node.js
// Because, it's CPU-bound
// You should be careful about this kind of code
loop(0, 1e+12)
```

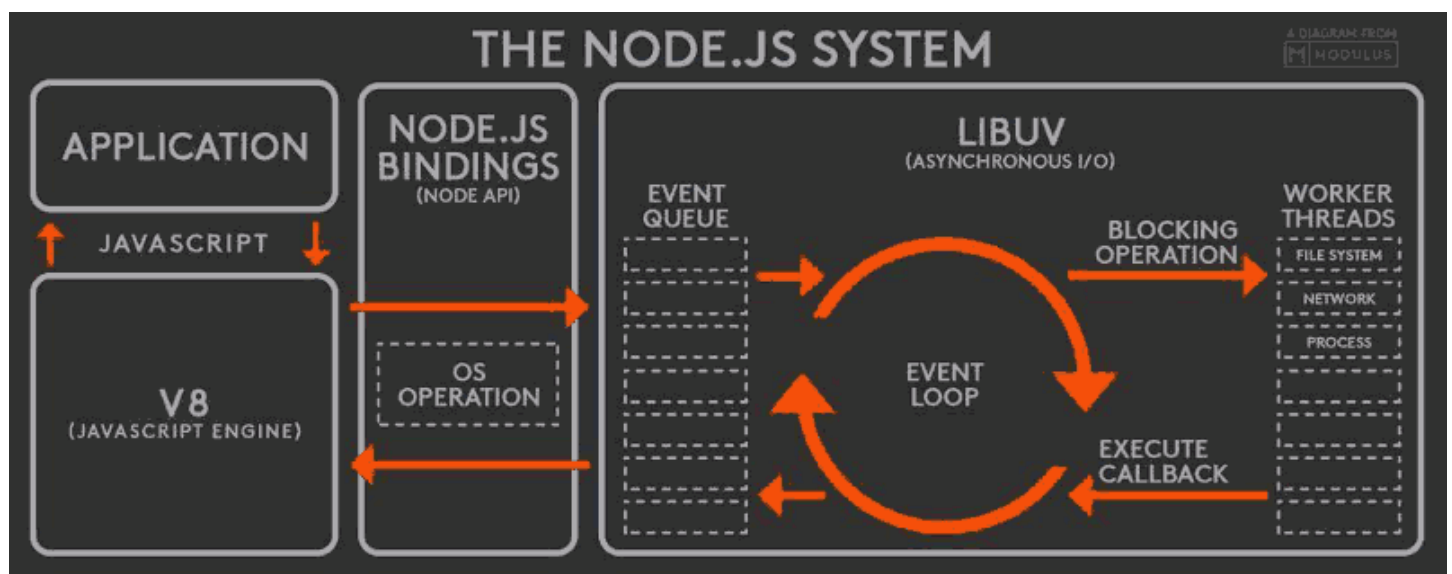
Non-Blocking IO Operation Example

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// this will postpone tick run step's while-loop to event loop cycles
// any other IO-bound operation (like filesystem reading) can take place
// in parallel
tick(1e+6)
tick(1e+7)
console.log('this will output before all of tick operations. i = %d', i)
console.log('because tick operations will be postponed')
tick(1e+8)
```



In simpler terms, Event Loop is a single-threaded queue mechanism which executes your CPU-bound code until end of its execution and IO-bound code in a non-blocking fashion.

However, Node.js under the carpet uses multi-threading for some of its operations through [libuv](#) Library.

Performance Considerations

- Non-blocking operations will not block the queue and will not effect the performance of the loop.
- However, CPU-bound operations will block the queue, so you should be careful not to do CPU-bound operations in your Node.js code.

Node.js non-blocks IO because it offloads the work to the operating system kernel, and when the IO operation supplies data (*as an event*), it will notify your code with your supplied callbacks.

Section 94.3: Increase maxSockets

Basics

```
require('http').globalAgent.maxSockets = 25

// You can change 25 to Infinity or to a different value by experimenting
```

Node.js by default is using `maxSockets = Infinity` at the same time (since [v0.12.0](#)). Until Node v0.12.0, the default was `maxSockets = 5` (see [v0.11.0](#)). So, after more than 5 requests they will be queued. If you want concurrency, increase this number.

Setting your own agent

http API is using a "[Global Agent](#)". You can supply your own agent. Like this:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

Turning off Socket Pooling entirely

```
const http = require('http')
const options = {.....}

options.agent = false

const request = http.request(options)
```

Pitfalls

- You should do the same thing for https API if you want the same effects
- Beware that, for example, [AWS](#) will use 50 instead of `Infinity`.

Chapter 95: Yarn Package Manager

[Yarn](#) is a package manager for Node.js, similar to npm. While sharing a lot of common ground, there are some key differences between Yarn and npm.

Section 95.1: Creating a basic package

The `yarn init` command will walk you through the creation of a package `.json` file to configure some information about your package. This is similar to the `npm init` command in npm.

Create and navigate to a new directory to hold your package, and then run `yarn init`

```
mkdir my-package && cd my-package
yarn init
```

Answer the questions that follow in the CLI

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
? Done in 27.31s.
```

This will generate a package `.json` file similar to the following

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A test package",
  "main": "index.js",
  "author": "StackOverflow Documentation",
  "license": "MIT"
}
```

Now lets try adding a dependency. The basic syntax for this is `yarn add [package-name]`

Run the following to install ExpressJS

```
yarn add express
```

This will add a dependencies section to your package `.json`, and add ExpressJS

```
"dependencies": {
  "express": "^4.15.2"
}
```

Section 95.2: Yarn Installation

This example explains the different methods to install Yarn for your OS.

macOS

Homebrew

```
brew update  
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Adding Yarn to your PATH

Add the following to your preferred shell profile (`.profile`, `.bashrc`, `.zshrc` etc)

```
export PATH="$PATH:`yarn global bin`"
```

Windows

Installer

First, install Node.js if it is not already installed.

Download the Yarn installer as an `.msi` from the [Yarn website](#).

Chocolatey

```
choco install yarn
```

Linux

Debian / Ubuntu

Ensure Node.js is installed for your distro, or run the following

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

Configure the YarnPkg repository

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -  
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
```

Install Yarn

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Install Node.js if not already installed

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Install Yarn

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo  
sudo yum install yarn
```

Arch

Install Yarn via AUR.

Example using yaourt:

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

All Distributions

Add the following to your preferred shell profile (`.profile`, `.bashrc`, `.zshrc` etc)

```
export PATH="$PATH:`yarn global bin`"
```

Alternative Method of Installation

Shell script

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

or specify a version to install

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

Tarball

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zxvf latest.tar.gz
```

Npm

If you already have npm installed, simply run

```
npm install -g yarn
```

Post Install

Check the installed version of Yarn by running

```
yarn --version
```

Section 95.3: Install package with Yarn

Yarn uses the same registry that npm does. That means that every package that is available on npm is the same on Yarn.

To install a package, run `yarn add package`.

If you need a specific version of the package, you can use `yarn add package@version`.

If the version you need to install has been tagged, you can use `yarn add package@tag`.

Chapter 96: OAuth 2.0

Section 96.1: OAuth 2 with Redis Implementation - grant_type: password

In this example I will be using oauth2 in rest api with redis database

Important: You will need to install redis database on your machine, Download it from [here](#) for linux users and from [here](#) to install windows version, and we will be using redis manager desktop app, install it from [here](#).

Now we have to set our node.js server to use redis database.

- **Creating Server file: app.js**

```
var express = require('express'),
    bodyParser = require('body-parser'),
    oauthserver = require('oauth2-server'); // Would be: 'oauth2-server'

var app = express();

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// Handle token grant requests
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // Will require a valid access_token
  res.send('Secret area');
});

app.get('/public', function (req, res) {
  // Does not require an access_token
  res.send('Public area');
});

// Error handling
app.use(app.oauth.errorHandler());

app.listen(3000);
```

- **Create Oauth2 model in routes/Oauth2/model.js**

```
var model = module.exports,
    util = require('util'),
    redis = require('redis');

var db = redis.createClient();
```

```

var keys = {
  token: 'tokens:%s',
  client: 'clients:%s',
  refreshToken: 'refresh_tokens:%s',
  grantTypes: 'clients:%s:grant_types',
  user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.token, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      accessToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.getClient = function (clientId, clientSecret, callback) {
  db.hgetall(util.format(keys.client, clientId), function (err, client) {
    if (err) return callback(err);

    if (!client || client.clientSecret !== clientSecret) return callback();

    callback(null, {
      clientId: client.clientId,
      clientSecret: client.clientSecret
    });
  });
};

model.getRefreshToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.refreshToken, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      refreshToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
  db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
};

model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.token, accessToken), {
    accessToken: accessToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

```



```

};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.refreshToken, refreshToken), {
    refreshToken: refreshToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.getUser = function (username, password, callback) {
  db.hgetall(util.format(keys.user, username), function (err, user) {
    if (err) return callback(err);

    if (!user || password !== user.password) return callback();

    callback(null, {
      id: username
    });
  });
};

```

You only need to install redis on your machine and run the following node file

```

#!/usr/bin/env node

var db = require('redis').createClient();

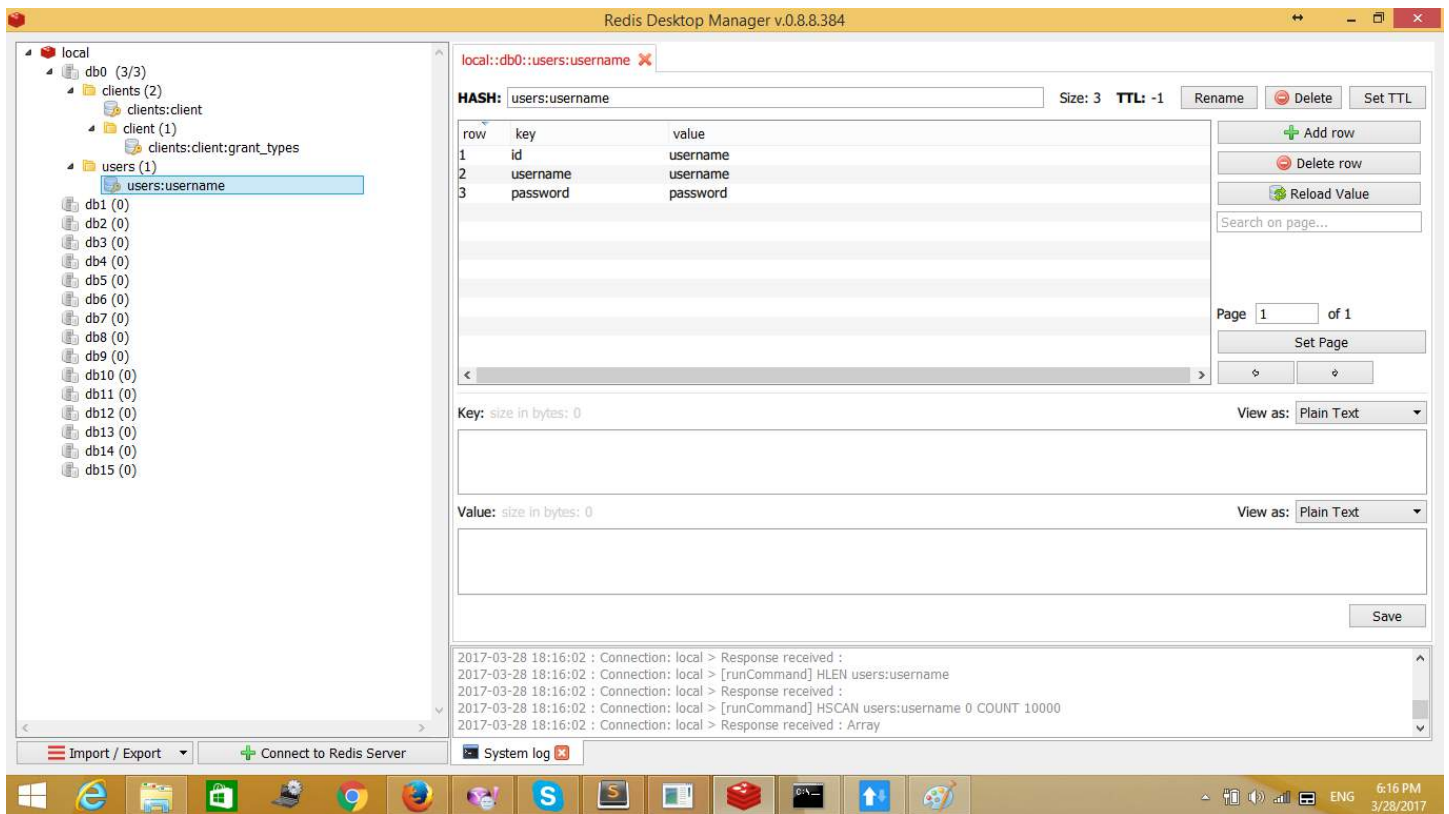
db.multi()
  .hmset('users:username', {
    id: 'username',
    username: 'username',
    password: 'password'
  })
  .hmset('clients:client', {
    clientId: 'client',
    clientSecret: 'secret'
  })//clientId + clientSecret to base 64 will generate Y2xpZW50bnNlY3JldA==
  .sadd('clients:client:grant_types', [
    'password',
    'refresh_token'
  ])
  .exec(function (errs) {
    if (errs) {
      console.error(errs[0].message);
      return process.exit(1);
    }

    console.log('Client and user added successfully');
    process.exit();
  });

```

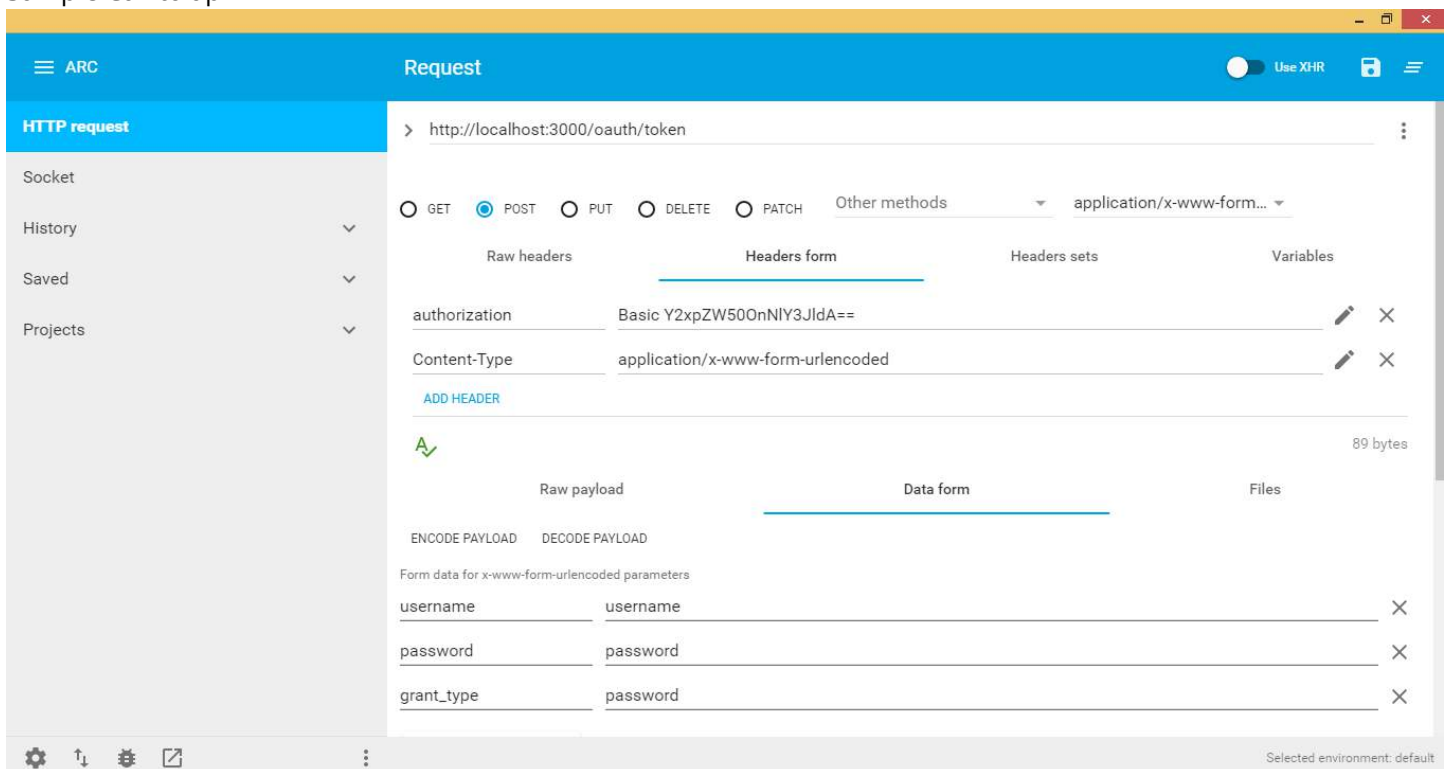
Note: This file will set credentials for your frontend to request token So your request from

Sample redis database after calling the above file:



Request will be as follows:

Sample Call to api



Header:

1. authorization: Basic followed by the password set when you first setup redis:

a. clientId + secretId to base64

2. Data form:

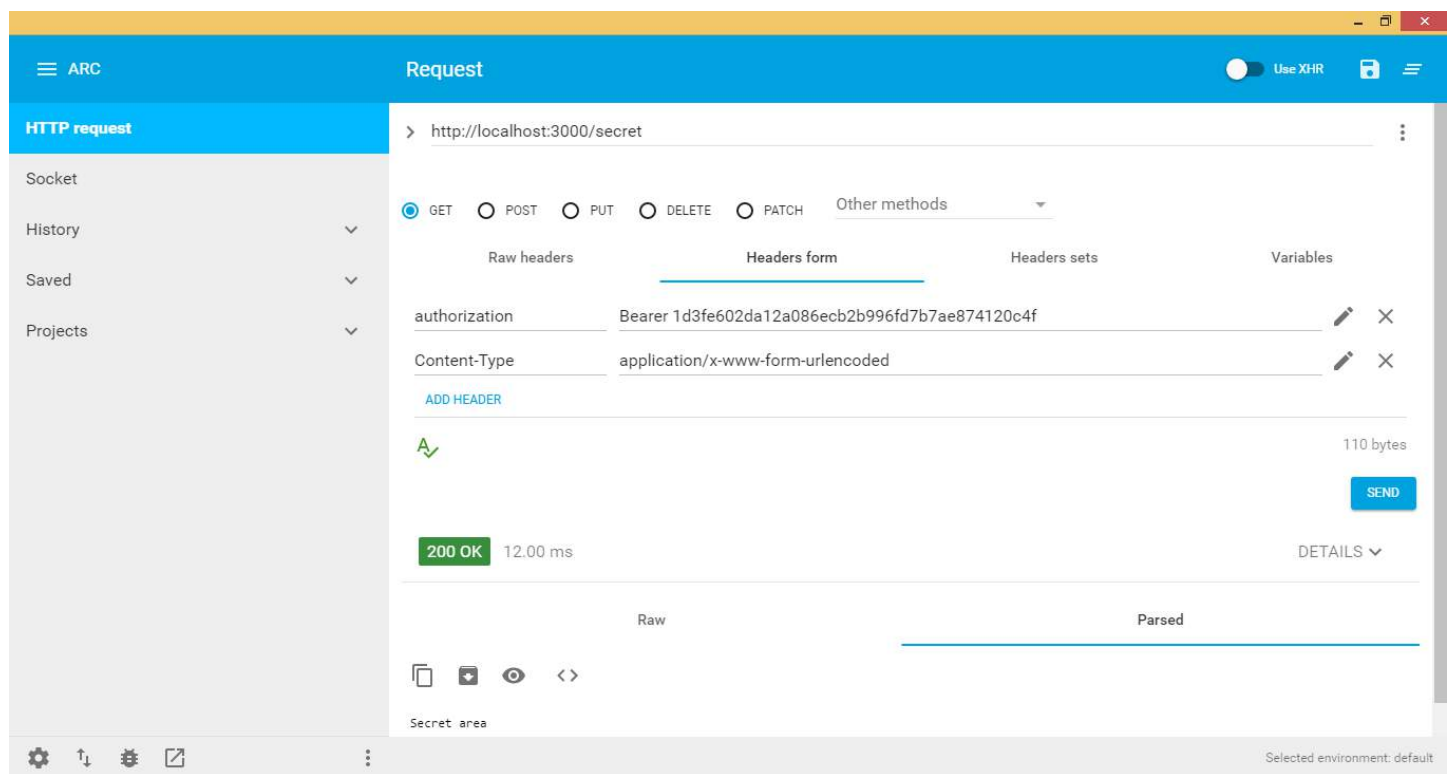
username: user that request token

password: user password

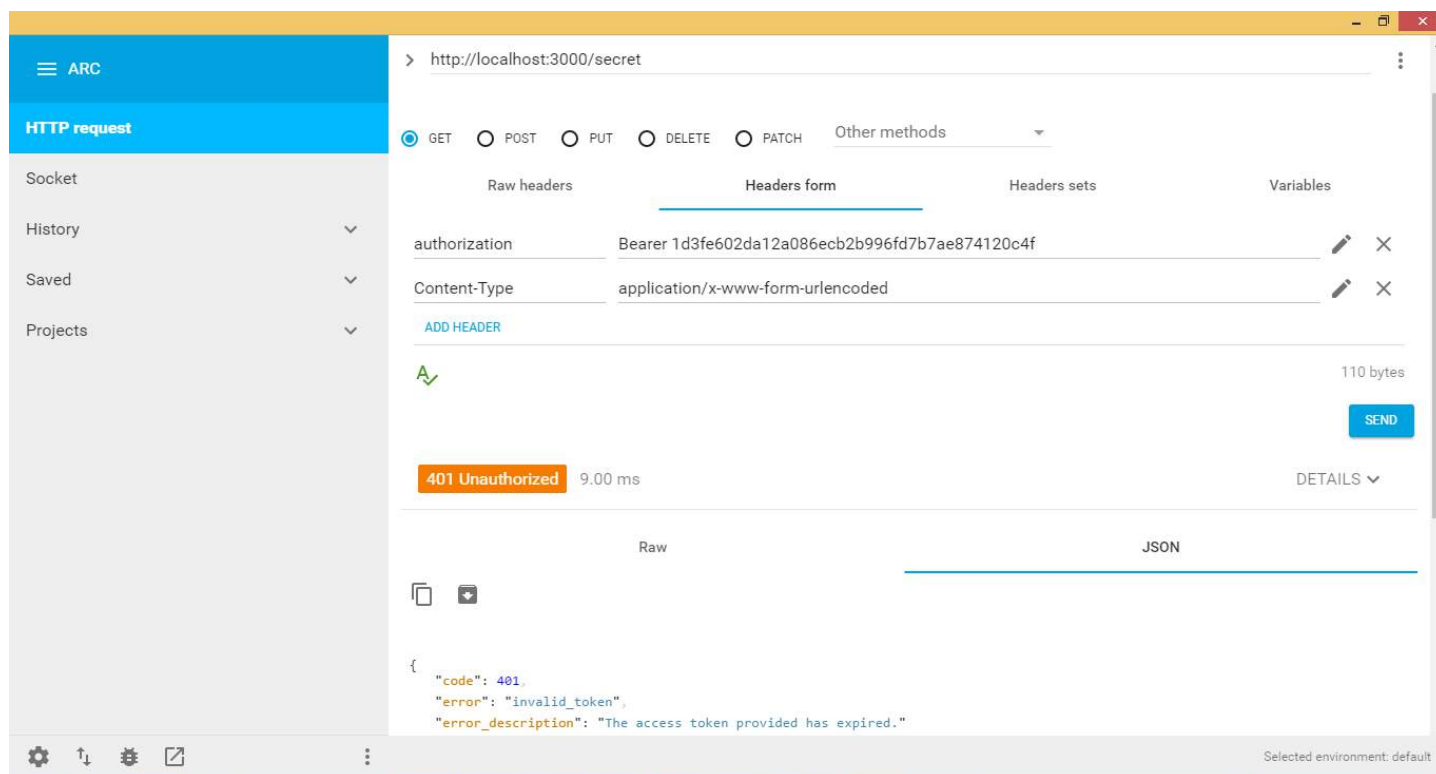
grant_type: depends on what options do you want, I choose password which takes only username and password to be created in redis, Data on redis will be as below:

```
{
  "access_token": "1d3fe602da12a086ecb2b996fd7b7ae874120c4f",
  "token_type": "bearer", // Will be used to access api + access+token e.g. bearer
1d3fe602da12a086ecb2b996fd7b7ae874120c4f
  "expires_in": 3600,
  "refresh_token": "b6ad56e5c9aba63c85d7e21b1514680bbf711450"
}
```

So We need to call our api and grab some secured data with our access token we have just created, see below:

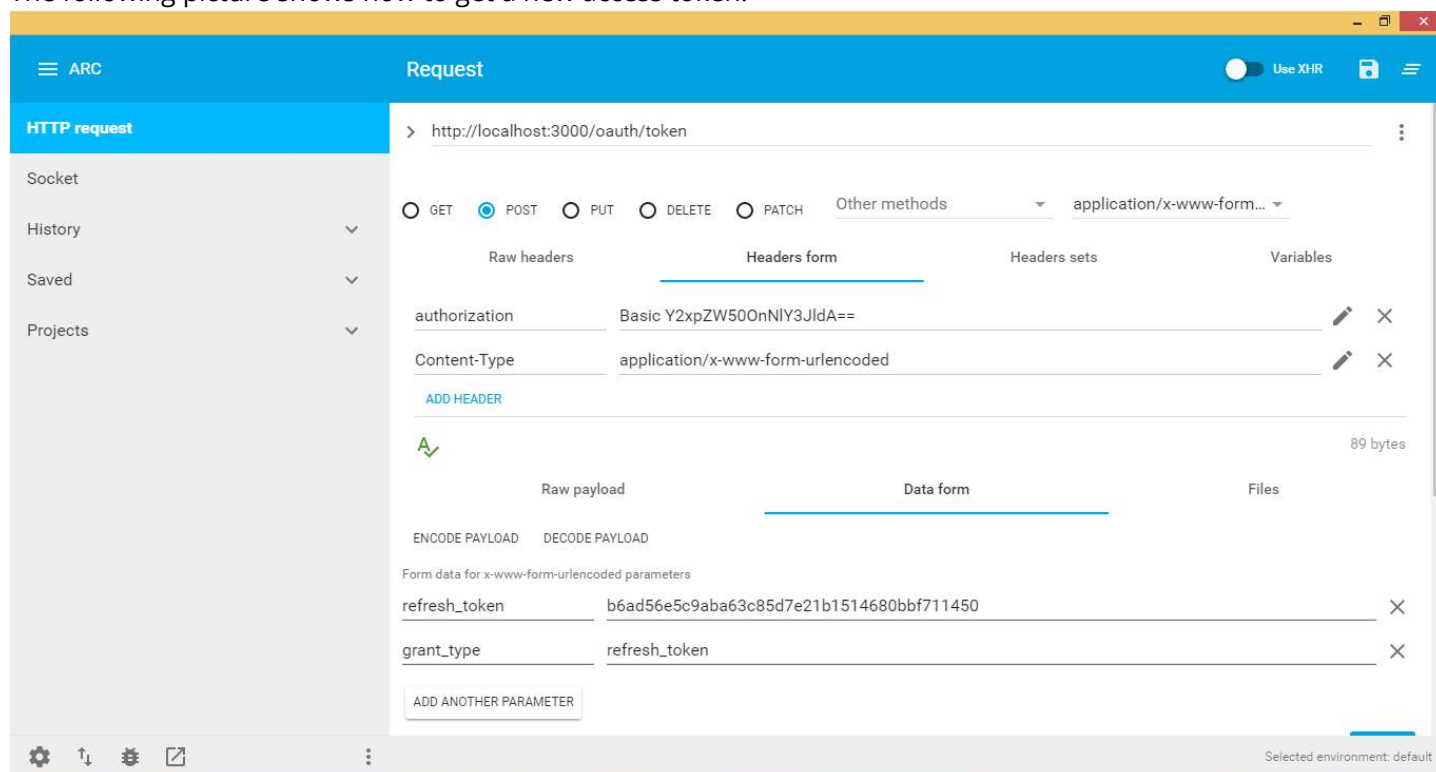


when token expires api will throw an error that the token expires and you cannot have access to any of the api calls, see image below :



Lets see what to do if the token expires, Let me first explain it to you, if access token expires a refresh_token exists in redis that reference the expired access_token So what we need is to call `oauth/token` again with the refresh_token grant_type and set the authorization to the Basic `clientId:clientsecret` (to base 64 !) and finally send the refresh_token, this will generate a new access_token with a new expiry data.

The following picture shows how to get a new access token:



Hope to Help!

Chapter 97: Node JS Localization

Its very easy to maintain localization nodejs express

Section 97.1: using i18n module to maintains localization in node js app

Lightweight simple translation module with dynamic json storage. Supports plain vanilla node.js apps and should work with any framework (like express, restify and probably more) that exposes an `app.use()` method passing in `res` and `req` objects. Uses common `__('...')` syntax in app and templates. Stores language files in json files compatible to webtranslateit json format. Adds new strings on-the-fly when first used in your app. No extra parsing needed.

express + i18n-node + cookieParser and avoid concurrency issues

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
  // setup some locales - other locales default to en silently
  locales: ['en', 'ru', 'de'],

  // sets a custom cookie name to parse locale settings from
  cookie: 'yourcookieName',

  // where to store json files - defaults to './locales'
  directory: __dirname + '/locales'
});

app.configure(function () {
  // you will need to use cookieParser to expose cookies to req.cookies
  app.use(express.cookieParser());

  // i18n init parses req for language headers, cookies, etc.
  app.use(i18n.init);
});

// serving homepage
app.get('/', function (req, res) {
  res.send(res.__('Hello World'));
});

// starting server
if (!module.parent) {
  app.listen(3000);
}
```

Chapter 98: Deploying Node.js application without downtime.

Section 98.1: Deployment using PM2 without downtime

ecosystem.json

```
{
  "name": "app-name",
  "script": "server",
  "exec_mode": "cluster",
  "instances": 0,
  "wait_ready": true
  "listen_timeout": 10000,
  "kill_timeout": 5000,
}
```

wait_ready

Instead of reload waiting for listen event, wait for process.send('ready');

listen_timeout

Time in ms before forcing a reload if app not listening.

kill_timeout

Time in ms before sending a final SIGKILL.

server.js

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

const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
  process.send('ready');
});

process.on('SIGINT', function() {
  server.close(function() {
    process.exit(0);
  });
});
```

You might need to wait for your application to have established connections with your DBs/caches/workers/whatever. PM2 needs to wait before considering your application as online. To do this, you need to provide `wait_ready: true` in a process file. This will make PM2 listen for that event. In your application you will need to add `process.send('ready');` when you want your application to be considered as ready.

When a process is stopped/restarted by PM2, some system signals are sent to your process in a given order.

First a SIGINT a signal is sent to your processes, signal you can catch to know that your process is going to be

stopped. If your application does not exit by itself before 1.6s (customizable) it will receive a SIGKILL signal to force the process exit. So if your application need to clean-up something states or jobs you can catch the SIGINT signal to prepare your application to exit.

Chapter 99: Node.js (express.js) with angular.js Sample code

This example shows how to create a basic express app and then serve AngularJS.

Section 99.1: Creating our project

We're good to go so, we run, again from console:

```
mkdir our_project
cd our_project
```

Now we're in the place where our code will live. To create the main archive of our project you can run

Ok, but how we create the express skeleton project?

It's simple:

```
npm install -g express express-generator
```

Linux distros and Mac should use **sudo** to install this because they're installed in the nodejs directory which is only accessible by the **root** user. If everything went fine we can, finally, create the express-app skeleton, just run

```
express
```

This command will create inside our folder an express example app. The structure is as follow:

```
bin/
public/
routes/
views/
app.js
package.json
```

Now if we run **npm start** and go to <http://localhost:3000> we'll see the express app up and running, fair enough we've generated an express app without too much trouble, but how can we mix this with AngularJS?.

How express works, briefly?

Express is a framework built on top of **Nodejs**, you can see the official documentation at the [Express Site](#). But for our purpose we need to know that **Express** is the responsible when we type, for example, <http://localhost:3000/home> of rendering the home page of our application. From the recently created app created we can check:

```
FILE: routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```


What this code is telling us is that when the user goes to <http://localhost:3000> it must render the **index** view and pass a **JSON** with a title property and value Express. But when we check the views directory and open index.jade we can see this:

```
extends layout
block content
  h1= title
  p Welcome to #{title}
```

This is another powerful Express feature, **template engines**, they allow you to render content in the page by passing variables to it or inherit another template so your pages are more compact and better understandable by others. The file extension is **.jade** as far as I know **Jade** changed the name for **Pug**, basically is the same template engine but with some updates and core modifications.

Installing Pug and updating Express template engine.

Ok, to start using Pug as the template engine of our project we need to run:

```
npm install --save pug
```

This will install Pug as a dependency of our project and save it to **package.json**. To use it we need to modify the file **app.js**:

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

And replace the line of view engine with pug and that's all. We can run again our project with **npm start** and we'll see that everything is working fine.

How AngularJS fits in all of this?

AngularJS is an Javascript **MVW**(Model-View-Whatever) Framework mainly used to create **SPA**(Simple Page Application) installing is fairly simple, you can go to [AngularJS website](#) and download the latest version which is **v1.6.4**.

After we downloaded AngularJS when should copy the file to our **public/javascripts** folder inside our project, a little explanation, this is the folder that serves the static assets of our site, images, css, javascript files and so on. Of course this is configurable through the **app.js** file, but we'll keep it simple. Now we create a file named **ng-app.js**, the file where our application will live, inside our javascripts public folder, just where AngularJS lives. To bring AngularJS up we need to modify the content of **views/layout.pug** as follow:

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body(ng-controller='indexController')
    block content

    script(type='text-javascript', src='javascripts/angular.min.js')
    script(type='text-javascript', src='javascripts/ng-app.js')
```

What are we doing here?, well, we're including AngularJS core and our recently created file **ng-app.js** so when the

template is rendered it will bring AngularJS up, notice the use of the **ng-app** directive, this is telling AngularJS that this is our application name and it should stick to it.

So, the content of our **ng-app.js** will be:

```
angular.module('first-app', [])
  .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
  $scope.name = 'sigfried';
}
```

We're using the most basic AngularJS feature here, **two-way data binding**, this allows us to refresh the content of our view and controller instantly, this is a very simple explanation, but you can make a research in Google or StackOverflow to see how it really works.

So, we have the basic blocks of our AngularJS application, but there is something we got to do, we need to update our index.pug page to see the changes of our angular app, let's do it:

```
extends layout
block content
  div(ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input(type='text' ng-model='name')
```

Here we're just binding the input to our defined property name in the AngularJS scope inside our controller:

```
$scope.name = 'sigfried';
```

The purpose of this is that whenever we change the text in the input the paragraph above will update its content inside the {{name}}, this is called **interpolation**, again another AngularJS feature to render our content in the template.

So, all is setup, we can now run **npm start** go to <http://localhost:3000> and see our express application serving the page and AngularJS managing the application frontend.

Chapter 100: NodeJs Routing

How to set up basic Express web server under the node js and Exploring the Express router.

Section 100.1: Express Web Server Routing

Creating Express Web Server

Express server came handy and it deeps through many user and community. It is getting popular.

Lets create a Express Server. For Package Management and Flexibility for Dependency We will use NPM(Node Package Manager).

1. Go to the Project directory and create package.json file. **package.json** { "name": "expressRouter", "version": "0.0.1", "scripts": { "start": "node Server.js" }, "dependencies": { "express": "^4.12.3" } }
2. Save the file and install the express dependency using following command *npm install*. This will create node_modules in you project directory along with required dependency.

3. Let's create Express Web Server. Go to the Project directory and create server.js file. **server.js**

```
var express = require("express"); var app = express();
```

```
//Creating Router() object
```

```
var router = express.Router();
```

```
// Provide all routes here, this is for Home page.
```

```
router.get("/", function(req, res){  
  res.json({ "message" : "Hello World" });
```

```
});
```

```
app.use("/api",router);
```

```
// Listen to this Port
```

```
app.listen(3000,function(){ console.log("Live at Port 3000"); });
```

For more detail on setting node server you can see [\[here\]](#)^[1].

4. Run the server by typing following command.

```
node server.js
```

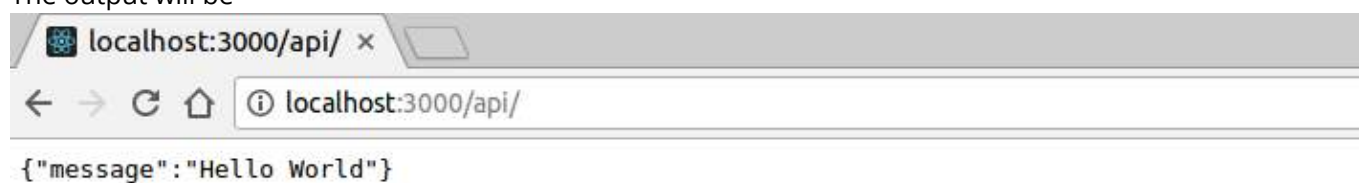
If Server runs successfully, you will se something like this.

```
pralad@pralad: ~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
```

5. Now go to the browser or postman and made a request

<http://localhost:3000/api/>

The output will be



That is all, the basic of Express routing.

Now let's handle the GET,POST etc.

Change your server.js file like

```
var express = require("express");
var app = express();

//Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req,res,next) {
  console.log("/" + req.method);
  next();
});

// Provide all routes here, this is for Home page.

router.get("/",function(req,res){
  res.json({"message" : "Hello World"});
});

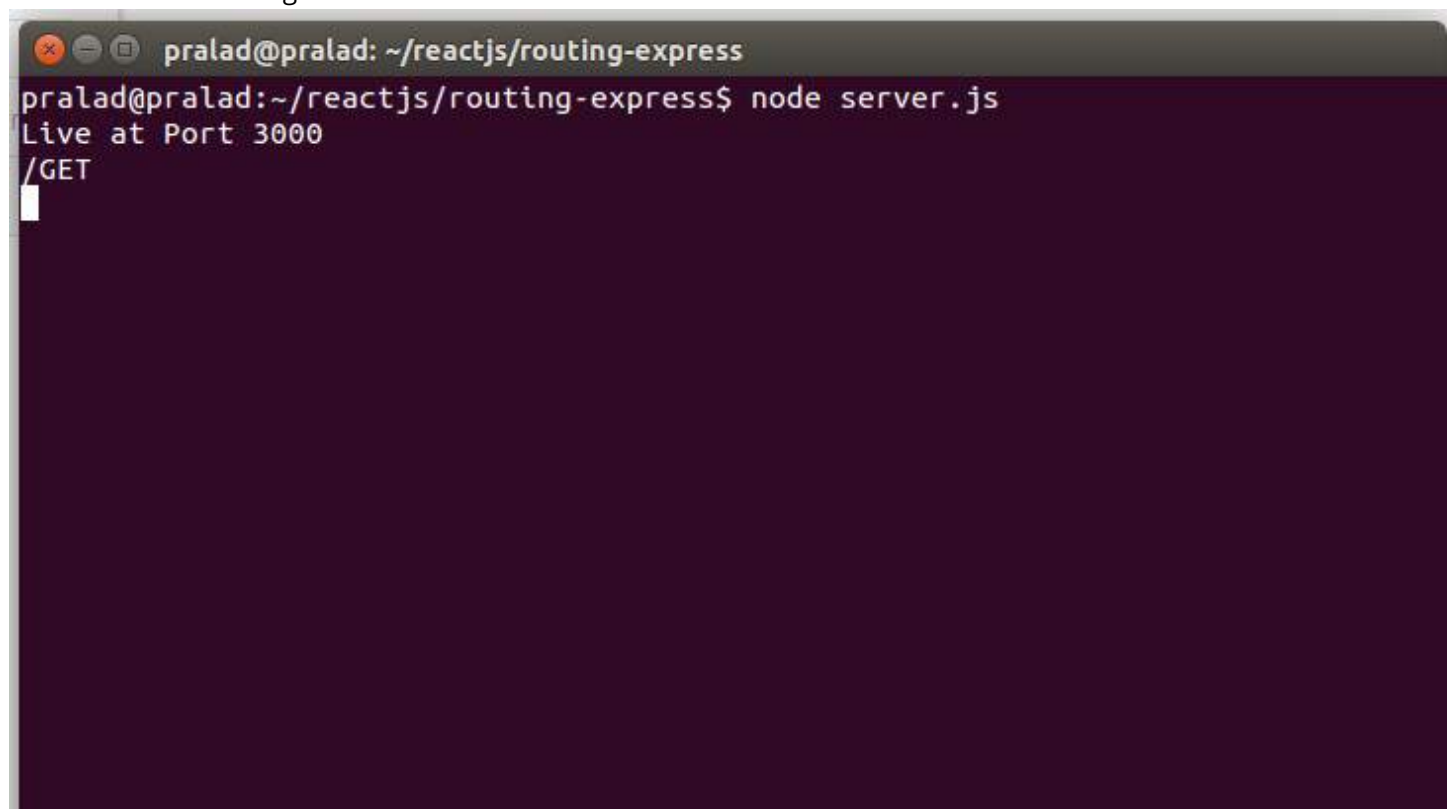
app.use("/api", router);

app.listen(3000,function(){
  console.log("Live at Port 3000");
});
```

Now if you restart the server and made the request to

```
http://localhost:3000/api/
```

You Will see something like

A terminal window with a dark background and light-colored text. The window title is 'pralad@pralad: ~/reactjs/routing-express'. The prompt is 'pralad@pralad:~/reactjs/routing-express\$'. The user has entered 'node server.js'. The output shows 'Live at Port 3000' on the first line and '/GET' on the second line, with a cursor on the third line.

```
pralad@pralad: ~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
/GET

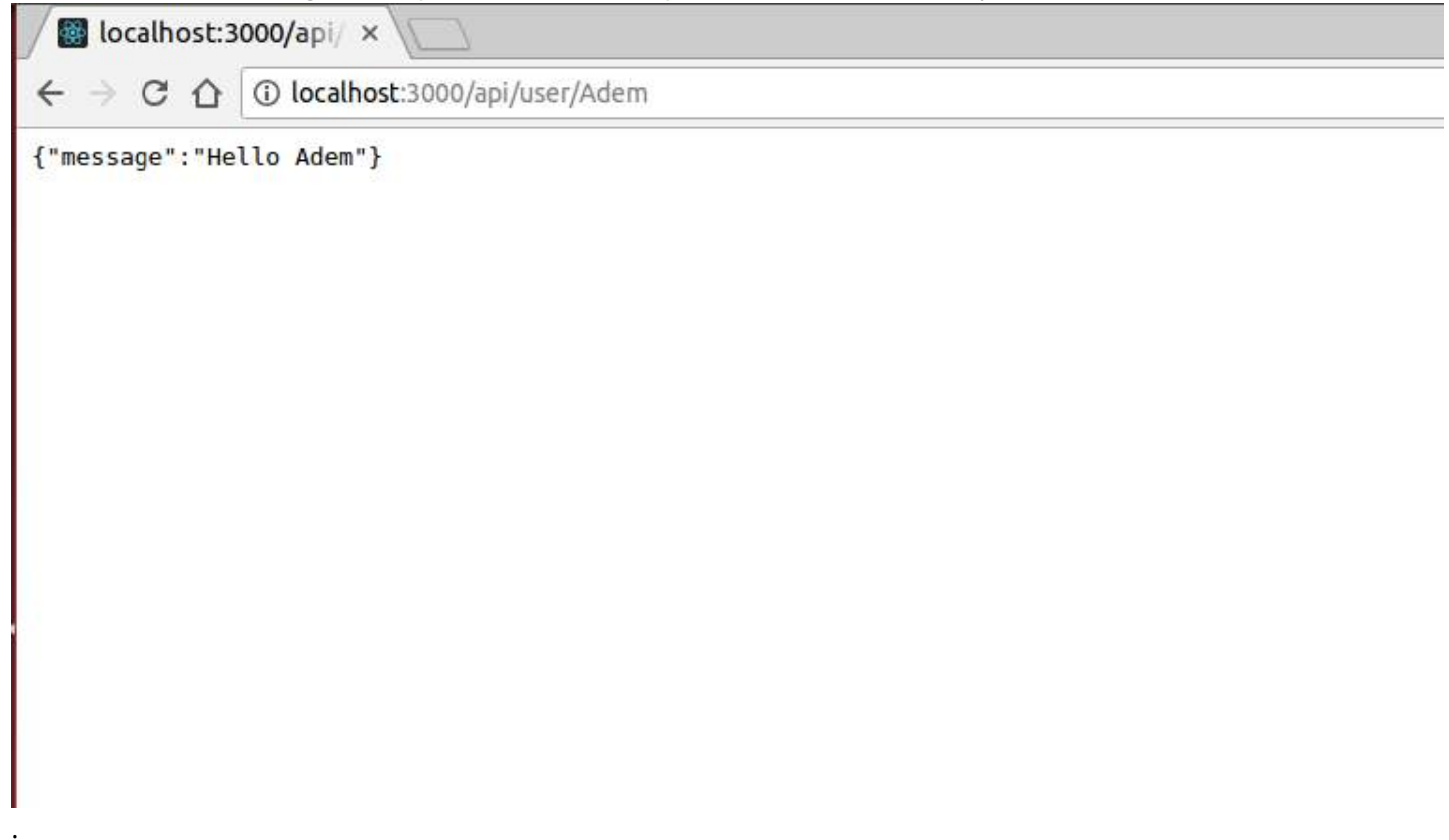
```

Accessing Parameter in Routing

You can access the parameter from url also, Like <http://example.com/api/:name/>. So name parameter can be access. Add the following code into your server.js

```
router.get("/user/:id", function(req, res){  
  res.json({"message" : "Hello " + req.params.id});  
});
```

Now restart server and go to [<http://localhost:3000/api/user/Adem>][4], the output will be like



Chapter 101: Creating a Node.js Library that Supports Both Promises and Error-First Callbacks

Many people like working with promises and/or `async/await` syntax, but when writing a module it would be useful to some programmers to support classic callback style methods as well. Rather than creating two modules, or two sets of functions, or having the programmer promisify your module, your module can support both programming methods at one using bluebird's `asCallback()` or Q's `nodeify()`.

Section 101.1: Example Module and Corresponding Program using Bluebird

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error('"a" must be a number'));
    if (typeof b !== 'number')
      return callback(new Error('"b" must be a number'));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    }).asCallback(callback);
  },
};
```

index.js

```

'use strict';

const math = require('./math');

// classic callbacks

math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('Test 1: ' + err);
  else
    console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('Test 2: ' + err);
  else
    console.log('Test 2: the answer is ' + result);
});

// promises

math.promiseSum(2, 5)
  .then(function(result) {
    console.log('Test 3: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 3: ' + err);
  });

math.promiseSum(1)
  .then(function(result) {
    console.log('Test 4: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 4: ' + err);
  });

// promise/callback method used like a promise

math.sum(8, 2)
  .then(function(result) {
    console.log('Test 5: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 5: ' + err);
  });

// promise/callback method used with callbacks

math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});

```


// promise/callback method used like a promise with async/await syntax

```
(async () => {  
  
  try {  
    let x = await math.sum(6, 3);  
    console.log('Test 7a: ' + x);  
  
    let y = await math.sum(4, 's');  
    console.log('Test 7b: ' + y);  
  
  } catch(err) {  
    console.log(err.message);  
  }  
  
})();
```

Chapter 102: MSSQL Intergration

To integrate any database with nodejs you need a driver package or you can call it a npm module which will provide you with basic API to connect with the database and perform interactions . Same is true with mssql database , here we will integrate mssql with nodejs and perform some basic queries on SQL tabels.

Section 102.1: Connecting with SQL via. mssql npm module

We will start with creating a simple node application with a basic structure and then connecting with local sql server database and performing some queries on that database.

Step 1: Create a directory/folder by the name of project which you intent to create. Initialize a node application using *npm init* command which will create a package.json in current directory .

```
mkdir mySqlApp
//folder created
cd mwSqlApp
//change to newly created directory
npm init
//answer all the question ..
npm install
//This will complete quickly since we have not added any packages to our app.
```

Step 2: Now we will create a App.js file in this directory and install some packages which we are going to need to connect to sql db.

```
sudo gedit App.js
//This will create App.js file , you can use your fav. text editor :)
npm install --save mssql
//This will install the mssql package to you app
```

Step 3: Now we will add a basic configuration variable to our application which will be used by mssql module to establish a connection .

```
console.log("Hello world, This is an app to connect to sql server.");
var config = {
  "user": "myusername", //default is sa
  "password": "yourStrong(!)Password",
  "server": "localhost", // for local machine
  "database": "staging", // name of database
  "options": {
    "encrypt": true
  }
}

sql.connect(config, err => {
  if(err){
    throw err ;
  }
  console.log("Connection Successful !");

  new sql.Request().query('select 1 as number', (err, result) => {
    //handle err
    console.dir(result)
    // This example uses callbacks strategy for getting results.
  })
})
```

```
});  
  
sql.on('error', err => {  
  // ... error handler  
  console.log("Sql database connection error " ,err);  
})
```

Step 4: This is the easiest step ,where we start the application and the application will connect to the sql server and print out some simple results .

```
node App.js  
// Output :  
// Hello world, This is an app to connect to sql server.  
// Connection Successful !  
// 1
```

To use promises or async for query execution refer the official documents of the mssql package :

- [Promises](#)
- [Async/Await](#)

Chapter 103: Project Structure

The structure of nodejs project is influenced by the personal preferences, project's architecture and module injection strategy being used. Also on event based arc' which uses dynamic module instantiation mechanism. To have a MVC structure it is imperative to separate out the server side and client side source code as the client side code will probably be minimized and sent to browser and is public in its basic nature. And the server side or back-end will provide API to perform CRUD operations

Section 103.1: A simple nodejs application with MVC and API

- The first major distinction is between the dynamically generated directories which will be used for hosting and source directories.
- The source directories will have a config file or folder depending on the amount of configuration you may have . This includes the environment configuration and business logic configuration which you may choose to put inside config directory.

```
|-- Config
    |-- config.json
    |-- appConfig
    |-- pets.config
    |-- payment.config
```

- Now the most vital directories where we distinguish between the server side/backend and the frontend modules . The 2 directories *server* and *webapp* represent the backend and frontend respectively which we can choose to put inside a source directory viz. *src*.

You can go with different names as per personal choice for server or webapp depending on what makes sense for you. Make sure you dont want to make it too long or too complex as it is in the end internal project structure.

- Inside the *server* directory you can have the controller ,the App.js/index.js which will be your main nodejs file and start point .The server dir. can also have the *dto* dir which holds all the data transfer objects which will be used by API controllers.

```
|-- server
    |-- dto
    |   |-- pet.js
    |   |-- payment.js
    |-- controller
    |   |-- PetsController.js
    |   |-- PaymentController.js
    |-- App.js
```

- The webapp directory can be divided into two major parts *public* and *mvc* , this is again influenced by what build strategy you want to use. We are using [browserfiy](#) to build the MVC part of webapp and minimize the contents from *mvc* directory simply put.

```
|-- webapp |-- public |-- mvc
```

- Now the public directory can contain all the static resources, images, css (you can have sass files as well) and

most importantly the HTML files .

```
|-- public
|-- build // will contain minified scripts(mvc)
|-- images
    |-- mouse.jpg
    |-- cat.jpg
|-- styles
    |-- style.css
|-- views
    |-- petStore.html
    |-- paymentGateway.html
    |-- header.html
    |-- footer.html
|-- index.html
```

- The *mvc* directory will contain the front-end logic including the *models*, the *view controllers* and any other *utils* modules you may need as part of UI. Also the *index.js* or *shell.js* whichever may suite you is part of this directory as well.

```
|-- mvc
    |-- controllers
        |-- Dashborad.js
        |-- Help.js
        |-- Login.js
    |-- utils
    |-- index.js
```

So in conclusion the entire project structure will look like below. And a simple build task like *gulp browserify* will minify the *mvc* scripts and publish in *public* directory. We can then provide this public directory as static resource via ***express.use(satic('public'))*** api.

```
|-- node_modules
|-- src
    |-- server
        |-- controller
        |-- App.js // node app
    |-- webapp
        |-- public
            |-- styles
            |-- images
            |-- index.html
        |-- mvc
            |-- controller
            |-- shell.js // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

Chapter 104: Avoid callback hell

Section 104.1: Async module

The source is available for download from GitHub. Alternatively, you can install using npm:

```
$ npm install --save async
```

As well as using Bower:

```
$ bower install async
```

Example:

```
var async = require("async");
async.parallel([
  function(callback) { ... },
  function(callback) { ... }
], function(err, results) {
  // optional callback
});
```

Section 104.2: Async Module

Thankfully, libraries like Async.js exist to try and curb the problem. Async adds a thin layer of functions on top of your code, but can greatly reduce the complexity by avoiding callback nesting.

Many helper methods exist in Async that can be used in different situations, like series, parallel, waterfall, etc. Each function has a specific use-case, so take some time to learn which one will help in which situations.

As good as Async is, like anything, its not perfect. Its very easy to get carried away by combining series, parallel, forever, etc, at which point you're right back to where you started with messy code. Be careful not to prematurely optimize. Just because a few async tasks can be run in parallel doesn't always mean they should. In reality, since Node is only single-threaded, running tasks in parallel on using Async has little to no performance gain.

The source is available for download from <https://github.com/caolan/async>. Alternatively, you can install using npm:

```
$ npm install --save async
```

As well as using Bower:

```
$ bower install async
```

Async's waterfall Example:

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

var myFile = '/tmp/test';

async.waterfall([
  function(callback) {
    fs.readFile(myFile, 'utf8', callback);
  },
  function(txt, callback) {
```

```
    txt = txt + '\nAppended something!';  
    fs.writeFile(myFile, txt, callback);  
  }  
], function (err, result) {  
  if(err) return console.log(err);  
  console.log('Appended text!');  
});
```

Chapter 105: Arduino communication with nodeJs

Way to show how Node.js can communicate with Arduino Uno.

Section 105.1: Node Js communication with Arduino via serialport

Node js code

Sample to start this topic is Node.js server communicating with Arduino via serialport.

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

Sample app.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
  baudrate: 9600
});

arduinoSerialPort.on('open', function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
```

```
});

app.get('/', function (req, res) {

  return res.send('Working');

});

app.get('/:action', function (req, res) {

  var action = req.params.action || req.param('action');

  if(action == 'led'){
    arduinoSerialPort.write("w");
    return res.send('Led light is on!');
  }
  if(action == 'off') {
    arduinoSerialPort.write("t");
    return res.send("Led light is off!");
  }

  return res.send('Action: ' + action);

});

app.listen(port, function () {
  console.log('Example app listening on port http://0.0.0.0:' + port + '!');
```



```
});
```

Starting sample express server:

```
node app.js
```

Arduino code

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.

  Serial.begin(9600); // Begin listening on port 9600 for serial

  pinMode(LED_BUILTIN, OUTPUT);

  digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

  if(Serial.available() > 0) // Read from serial port
  {
    char ReaderFromNode; // Store current character
    ReaderFromNode = (char) Serial.read();
    convertToState(ReaderFromNode); // Convert character to state
  }
  delay(1000);
}

void convertToState(char chr) {
  if(chr=='o'){
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
  }
  if(chr=='f'){
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
  }
}
```

Starting Up

1. Connect the arduino to your machine.
2. Start the server

Control the build in led via node js express server.

To turn on the led:

```
http://0.0.0.0:3000/led
```

To turn off the led:

```
http://0.0.0.0:3000/off
```

Chapter 106: N-API

The N-API is a new and better way for creating native module for NodeJS. N-API is in early stage so it may have inconsistent documentation.

Section 106.1: Hello to N-API

This module register hello function on hello module. hello function prints Hello world on console with printf and return 1373 from native function into javascript caller.

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
         * String describing the key for the property, encoded as UTF8.
         */
        .utf8name = "hello",
        /*
         * Set this to make the property descriptor object's value property
         * to be a JavaScript function represented by method.
         * If this is passed in, set value, getter and setter to NULL (since these members won't be
used).
         */
        .method = say_hello,
        /*
         * A function to call when a get access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be used).
         * The given function is called implicitly by the runtime when the property is accessed
         * from JavaScript code (or if a get on the property is performed using a N-API call).
         */
        .getter = NULL,
        /*
         * A function to call when a set access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be used).
         * The given function is called implicitly by the runtime when the property is set
         * from JavaScript code (or if a set on the property is performed using a N-API call).
         */
        .setter = NULL,
        /*
         * The value that's retrieved by a get access of the property if the property is a data
property.
         * If this is passed in, set getter, setter, method and data to NULL (since these members
won't be used).
         */
    }
```

```

    */
    .value = NULL,
    /*
     * The attributes associated with the particular property. See napi_property_attributes.
     */
    .attributes = napi_default,
    /*
     * The callback data passed into method, getter and setter if this function is invoked.
     */
    .data = NULL
};
/*
 * This method allows the efficient definition of multiple properties on a given object.
 */
status = napi_define_properties(env, exports, 1, &desc);

if (status != napi_ok)
    return;
}

NAPI_MODULE(hello, init)

```

Chapter 107: Multithreading

Node.js has been designed to be single threaded. So for all practical purposes, applications that launch with Node will run on a single thread.

However, Node.js itself runs multi-threaded. I/O operations and the like will run from a thread pool. Further will any instance of a node application run on a different thread, therefore to run multi-threaded applications one launches multiple instances.

Section 107.1: Cluster

The `cluster` module allows one to start the same application multiple times.

Clustering is desirable when the different instances have the same flow of execution and don't depend on one another. In this scenario, you have one master that can start forks and the forks (or children). The children work independently and have their own space of Ram and Event Loop.

Setting up clusters can be beneficial for websites / APIs. Any thread can serve any customer, as it doesn't depend on other threads. A Database (like Redis) would be used to share Cookies, as **variables can't be shared!** between the threads.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
  // runs only once (within the master);
  console.log('I am the master, launching workers!');
  for(var i = 0; i < numCPUs; i++) cluster.fork();
} else {
  // runs in each fork
  console.log('I am a fork!');

  // here one could start, as an example, a web server
}

console.log('I am always called as well');
```

Section 107.2: Child Process

Child Processes are the way to go when one wants to run processes independently with different initialization and concerns. Like forks in clusters, a `child_process` runs in its thread, but unlike forks, it has a way to communicate with its parent.

The communication goes both ways, so parent and child can listen for messages and send messages.

Parent (../parent.js)

```
var child_process = require('child_process');
console.log('[Parent]', 'inititalize');

var child1 = child_process.fork(__dirname + '/child');
```

```

child1.on('message', function(msg) {
  console.log('[Parent]', 'Answer from child: ', msg);
});

// one can send as many messages as one want
child1.send('Hello'); // Hello to you too :)
child1.send('Hello'); // Hello to you too :)

// one can also have multiple children
var child2 = child_process.fork(__dirname + '/child');

```

Child (../child.js)

```

// here would one initialize this child
// this will be executed only once
console.log('[Child]', 'inititalize');

// here one listens for new tasks from the parent
process.on('message', function(messageFromParent) {

  //do some intense work here
  console.log('[Child]', 'Child doing some intense work');

  if(messageFromParent == 'Hello') process.send('Hello to you too :');
  else process.send('what?');

})

```

Next to message one can listen to [many events](#) like 'error', 'connected' or 'disconnect'.

Starting a child process has a certain cost associated with it. One would want to spawn as few of them as possible.

Chapter 108: Windows authentication under node.js

Section 108.1: Using activedirectory

The example below is taken from the full docs, available [here \(GitHub\)](#) or [here \(NPM\)](#).

Installation

```
npm install --save activedirectory
```

Usage

```
// Initialize
var ActiveDirectory = require('activedirectory');
var config = {
  url: 'ldap://dc.domain.com',
  baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// Authenticate
ad.authenticate(username, password, function(err, auth) {
  if (err) {
    console.log('ERROR: '+JSON.stringify(err));
    return;
  }
  if (auth) {
    console.log('Authenticated!');
  }
  else {
    console.log('Authentication failed!');
  }
});
```

Chapter 109: Require()

This documentation focuses on explaining the uses and of the `require()` statement that [NodeJS](#) includes in their language.

Require is an import of certain files or packages used with NodeJS's modules. It is used to improve code structure and uses. `require()` is used on files that are installed locally, with a direct route from the file that is require'ing.

Section 109.1: Beginning `require()` use with a function and file

Require is a statement that Node interprets as, in some sense, a getter function. For example, say you have a file named `analysis.js`, and the inside of your file looks like this,

```
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

This file contains only the method, `analyzeWeather(weather_data)`. If we want to use this function, it must be either used inside of this file, or copied to the file it wants to be used by. However, Node has included a very useful tool to help with code and file organization, which is [modules](#).

In order to utilize our function, we must first **export** the function through a statement at the beginning. Our new file looks like this,

```
module.exports = {
  analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

With this small `module.exports` statement, our function is now ready for use outside of the file. All that is left to do is to use `require()`.

When require'ing a function or file, the syntax is very similar. It is usually done at the beginning of the file and set to **var**'s or **const**'s for use throughout the file. For example, we have another file (on the same level as `analyze.js` named `handleWeather.js` that looks like this,

```
const analysis = require('./analysis.js');

weather_data = {
  time: '01/01/2001',
  precip: 0.75,
  temp: 78,
  //More weather data...
};
analysis.analyzeWeather(weather_data);
```

In this file, we are using `require()` to grab our `analysis.js` file. When used, we just call the variable or constant

assigned to this `require` and use whatever function inside that is exported.

Section 109.2: Beginning `require()` use with an NPM package

Node's `require` is also very helpful when used in tandem with an [NPM package](#). Say, for example, you would like to use the NPM package [require](#) in a file named `getWeather.js`. After [NPM installing](#) your package through your command line (`git install request`), you are ready to use it. Your `getWeather.js` file might look like this,

```
var https = require('request');

//Construct your url variable...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('Response => ' + response);
    console.log('Body => ' + body);
  }
});
```

When this file is run, it first `require`'s (imports) the package you just installed called `request`. Inside of the `request` file, there are many functions you now have access to, one of which is called `get`. In the next couple lines, the function is used in order to make an [HTTP GET request](#).

Chapter 110: Route-Controller-Service structure for ExpressJS

Section 110.1: Model-Routes-Controllers-Services Directory Structure

```
????models
?   ???user.model.js
????routes
?   ???user.route.js
????services
?   ???user.service.js
????controllers
?   ???user.controller.js
```

For modular code structure the logic should be divided into these directories and files.

Models - The schema definition of the Model

Routes - The API routes maps to the Controllers

Controllers - The controllers handles all the logic behind validating request parameters, query, Sending Responses with correct codes.

Services - The services contains the database queries and returning objects or throwing errors

This coder will end up writing more codes. But at the end the codes will be much more maintainable and seperated.

Section 110.2: Model-Routes-Controllers-Services Code Structure

user.model.js

```
var mongoose = require('mongoose')

const UserSchema = new mongoose.Schema({
  name: String
})

const User = mongoose.model('User', UserSchema)

module.exports = User;
```

user.routes.js

```
var express = require('express');
var router = express.Router();

var UserController = require('../controllers/user.controller')

router.get('/', UserController.getUsers)
```

```
module.exports = router;
```

user.controllers.js

```
var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
  // Validate request parameters, queries using express-validator

  var page = req.params.page ? req.params.page : 1;
  var limit = req.params.limit ? req.params.limit : 10;
  try {
    var users = await UserService.getUsers({}, page, limit)
    return res.status(200).json({ status: 200, data: users, message: "Succesfully Users Retrieved" });
  } catch (e) {
    return res.status(400).json({ status: 400, message: e.message });
  }
}
```

user.services.js

```
var User = require('../models/user.model')

exports.getUsers = async function (query, page, limit) {

  try {
    var users = await User.find(query)
    return users;
  } catch (e) {
    // Log Errors
    throw Error('Error while Paginating Users')
  }
}
```

Chapter 11: Push notifications

module/framework	description
node.js/express	Simple back-end framework for Node.js application, very easy to use and extremely powerful
Socket.io	Socket.IO enables real-time bidirectional event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed.
Push.js	The world's most versatile desktop notifications framework
OneSignal	Just another form off push notifications for Apple devices
Firebase	Firebase is Google's mobile platform that helps you quickly develop high-quality apps and grow your business.

So if you wanna make web app notification I suggest you to use Push.js or SoneSignal framework for Web/mobile app.

Push is the fastest way to get up and running with Javascript notifications. A fairly new addition to the official specification, the Notification API allows modern browsers such as Chrome, Safari, Firefox, and IE 9+ to push notifications to a user's desktop.

You will have to use Socket.io and some backend framework, I will user Express for this example.

Section 111.1: Web notification

First, you will need to install [Push.js](#) module.

```
$ npm install push.js --save
```

Or import it to your front-end app through [CDN](#)

```
<script src="./push.min.js"></script> <!-- CDN link -->
```

After you are done with that, you should be good to go. This is how it should look like if u wanna make simple notification:

```
Push.create('Hello World!')
```

I will assume that you know how to setup [Socket.io](#) with your app. Here is some code example of my backend app with express:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

After your server is all set up, you should be able to move on to front-end stuff. Now all we have to do is import Socket.io [CDN](#) and add this code to my *index.html* file:

```
<script src="../socket.io.js"></script> <!-- CDN link -->
<script>
  var socket = io.connect('http://localhost');
  socket.on('pushNotification', function (data) {
    console.log(data);
    Push.create("Hello world!", {
      body: data.msg, //this should print "hello"
      icon: '/icon.png',
      timeout: 4000,
      onClick: function () {
        window.focus();
        this.close();
      }
    });
  });
</script>
```

There you go, now you should be able to display your notification, this also works on any Android device, and if u wanna use [Firebase](#) cloud messaging, you can use it with this module, [Here](#) is link for that example written by Nick (creator of Push.js)

Section 11.2: Apple

Keep in mind that this will not work on Apple devices (I didnt test them all), but if you want to make push notifications check [OneSignal](#) plugin.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Abdelaziz Mokhnache	Chapter 1
Aeolingamenfel	Chapters 1 and 4
Ahmed Metwally	Chapter 21
Aikon Mogwai	Chapters 3, 27 and 32
aisflat439	Chapter 89
Ajitej Kaushik	Chapter 56
akinjide	Chapter 24
Alex Logan	Chapters 18 and 25
alexi2	Chapter 3
Alistair Norris	Chapter 34
Aljoscha Meyer	Chapter 1
Amila Sampath	Chapter 1
Aminadav	Chapters 5, 7 and 8
Amreesh Tyagi	Chapter 2
Andrés Encarnación	Chapter 8
Andres C. Viesca	Chapter 3
Andrew Brooke	Chapter 95
AndrewLeonardi	Chapter 6
Anh Cao	Chapter 7
Ankit Gomkale	Chapter 1
Ankur Anand	Chapters 1, 2, 10, 34 and 49
Antenka	Chapter 51
Aph	Chapter 3
Apidcloud	Chapter 23
arcs	Chapter 107
Asaf Manassen	Chapters 3 and 10
asherbar	Chapter 7
Ates Goral	Chapter 2
B Thuy	Chapter 59
baranskistad	Chapter 1
Batsu	Chapters 3 and 7
Bearington	Chapter 18
Benjamin	Chapter 16
Beshoy Hanna	Chapter 67
Bharat	Chapter 6
Big Dude	Chapter 69
Brett Jackson	Chapter 23
brianmearns	Chapter 3
Buzut	Chapters 7, 91 and 92
ccnokes	Chapter 2
CD..	Chapter 2
Chance Snow	Chapter 10
Chezzwizz	Chapters 1 and 7
Christophe Marois	Chapter 4
Christopher Ronning	Chapter 1
CJ Harries	Chapter 108
Clement JACOB	Chapter 14
commonSenseCode	Chapter 6
cyanbeam	Chapters 5 and 22

damitj07	Chapters 93, 102 and 103
Daniel Verem	Chapter 3
Dave	Chapters 62 and 101
David G.	Chapter 2
David Gatti	Chapter 1
David Xu	Chapters 61, 62, 63, 64 and 65
Dean Rather	Chapter 12
devnull69	Chapters 3, 5 and 24
Djizeus	Chapter 1
Dmitriy Borisov	Chapter 7
Dom Vinyard	Chapter 62
DominicValenciana	Chapter 81
DrakaSAN	Chapters 2, 4, 5, 11 and 26
dthree	Chapter 43
Duly Kinsky	Chapters 4, 11 and 22
duncanhall	Chapter 23
efeder	Chapters 10 and 22
Eric Fortin	Chapter 2
Eric Smekens	Chapter 10
evalsocket	Chapter 68
Everettss	Chapter 2
Explosion Pills	Chapter 2
F. Kauder	Chapter 26
FabianCook	Chapters 29 and 50
Fikra	Chapter 75
Florian Hämmerle	Chapters 1, 2, 9, 16, 64, 65 and 94
Forivin	Chapter 28
Freddie Coleman	Chapter 13
fresh5447	Chapter 57
gentlejo	Chapter 98
GilZ	Chapter 7
gnerkus	Chapter 4
guleria	Chapters 7, 20 and 23
H. Pauwelyn	Chapter 85
Harangue	Chapter 1
Hasan A Yousef	Chapter 35
hexacyanide	Chapters 1, 2, 4, 5, 7 and 15
Himani Agrawal	Chapters 18 and 58
Housseem Yahiaoui	Chapter 52
HungryCoder	Chapters 2 and 7
Iceman	Chapters 27 and 31
Inanc Gumus	Chapters 1, 3, 7 and 94
Ionică Bizău	Chapter 2
Ivan Hristov	Chapter 33
Jacek Labuda	Chapter 7
jakerella	Chapter 4
James Billingham	Chapter 6
James Taylor	Chapter 2
jamescostian	Chapter 11
Jason	Chapter 5
Jasper	Chapter 1
jdrydn	Chapters 5 and 8
jemiloi	Chapter 3
Jeremy Banks	Chapter 1
jerry	Chapter 26

João Andrade	Chapter 2
John	Chapter 85
John Slegers	Chapters 1 and 24
John Vincent Jardin	Chapters 7, 19 and 20
johni	Chapter 22
JohnnyCoder	Chapter 8
Jonas S	Chapter 12
Josh	Chapter 2
Joshua Kleveter	Chapter 12
Kapil Vats	Chapter 8
Karlen	Chapter 80
Kelum Senanayake	Chapters 82 and 83
Kid Binary	Chapters 3 and 16
KlwntSingh	Chapters 8, 13, 17, 22, 23, 54 and 81
Kornel	Chapter 10
Léo Martin	Chapter 71
lauriys	Chapter 10
Loufylouf	Chapter 5
Louis Barranqueiro	Chapters 4 and 13
Luis García	Chapter 24
Luis González	Chapter 20
M. A. Cordeiro	Chapter 71
m02ph3u5	Chapters 2 and 5
M1kstur	Chapter 88
Maciej Rostański	Chapter 7
Manas Jayanth	Chapter 30
Manuel	Chapter 62
manuerumx	Chapter 25
Mario Rozic	Chapter 111
Matthew Harwood	Chapter 2
Matthew Shanley	Chapter 34
MatthieuLemoine	Chapter 23
MayorMonty	Chapter 16
Meinkraft	Chapter 1
mezzode	Chapter 10
Michael Buen	Chapter 14
midnightsyntax	Chapters 4, 29 and 70
Mikhail	Chapters 2, 3, 4, 13, 25 and 27
MindlessRanger	Chapter 11
Mindsers	Chapter 2
Mohit Gangrade	Chapter 3
MSB	Chapter 5
Mukesh Sharma	Chapters 3, 16, 23, 24 and 39
Naeem Shaikh	Chapters 3, 25 and 44
Nainesh Raval	Chapter 50
Native Coder	Chapter 85
ndugger	Chapter 1
Nick	Chapters 1 and 2
Niklas	Chapter 3
Niroshan Ranapathi	Chapters 72, 73, 75 and 76
Nivesh	Chapters 3, 12 and 17
nomanbinhussein	Chapters 62 and 110
noob	Chapter 3
num8er	Chapter 2
NuSkooler	Chapter 7

oliolioli	Chapter 78
Oliver	Chapter 6
omgimanerd	Chapter 85
optimus	Chapter 18
Osama Bari	Chapters 65 and 97
Parellin	Chapter 15
Parham Alvani	Chapter 106
parlad neupane	Chapter 100
Pasha Rumkin	Chapter 3
Pedro Otero	Chapter 5
peteb	Chapters 4, 9 and 41
Peter G	Chapter 24
Philip Cornelius Glover	Chapter 109
Philipp Flenker	Chapter 2
Pieter Herroelen	Chapter 2
pranspach	Chapters 20 and 24
Pyloid	Chapter 2
Qiong Wu	Chapter 25
QoP	Chapter 2
Quill	Chapter 2
Rafal Wiliński	Chapters 2 and 3
raghu	Chapter 23
rdegges	Chapter 2
Red	Chapter 84
refaelos	Chapter 2
Rick	Chapter 53
Rishikesh Chandra	Chapter 1
riyadhالنور	Chapters 16 and 17
Rizowski	Chapter 2
Robbie	Chapter 10
Roopesh	Chapter 90
Rowan Harley	Chapter 45
Rupali Pemare	Chapter 71
Sameer Srivastava	Chapter 18
Sanketh Katta	Chapter 14
Sathish	Chapter 40
Satyam S	Chapter 70
sBanda	Chapters 13, 17 and 105
ScientiaEtVeritas	Chapter 37
Shabin Hashim	Chapters 1, 3, 5, 7 and 10
shikhar bansal	Chapter 71
Shiven	Chapters 1, 2 and 4
Shriganesh Kolhe	Chapters 26 and 50
Sibeesh Venu	Chapter 1
Siddharth Srivastva	Chapter 7
sigfried	Chapter 99
signal	Chapter 60
Simplans	Chapter 10
sjmarshy	Chapters 17 and 23
Skanda	Chapter 2
skiilaa	Chapters 64 and 95
Sky	Chapter 26
snuggles08	Chapter 19
Sorangwala Abbasali	Chapter 8
SteveLacy	Chapter 10

still_learning	Chapters 2 and 4
subbu	Chapter 2
Sven 31415	Chapter 10
Sveratum	Chapter 7
Syam Pradeep	Chapter 86
SynapseTech	Chapter 66
tandrewnichols	Chapters 3 and 7
the12	Chapter 2
thefourtheye	Chapter 1
theunexpected1	Chapter 1
Tim Jones	Chapter 4
tlo	Chapter 2
Tomás Cañibano	Chapter 10
Toni Villena	Chapter 62
topheman	Chapter 17
Tushar Gupta	Chapter 20
tverdohleb	Chapters 3 and 23
tyehia	Chapters 23, 96 and 104
umesh	Chapter 77
user2314737	Chapters 1 and 58
user6939352	Chapter 10
uzaif	Chapters 2 and 5
V1P3R	Chapters 7 and 10
Veger	Chapter 79
victorkohl	Chapters 7 and 10
vintproykt	Chapter 74
VladNeacsu	Chapter 2
VooVoo	Chapter 53
Vsevolod Goloviznin	Chapter 38
vsjn3290ckjnaoij2jikndckjb	Chapter 46
Waterscroll	Chapter 44
Will	Chapter 25
William Carron	Chapter 87
xam	Chapters 1, 36 and 75
xims	Chapter 3
Yerko Palma	Chapters 2 and 3
ymz	Chapter 36
yrtimiD	Chapter 47
Zanon	Chapter 4
Ze Rubeus	Chapters 22 and 42
Zoltán Schmidt	Chapter 48
zurfyx	Chapters 12, 36 and 55

You may also like

