NodeJS-MasterClass Andrew Lee, **August, 04, 2018**

**---**

Github: <https://github.com/pirple/The-NodeJS-Master-Class>

Good Terminal Commands: <https://medium.com/the-code-review/top-10-bash-file-system-commands-you-cant-live-without-4cd937bd7df1>

---

**Background Information:**

*The Story of Node.js*

* Ryan Dahl, creator of NodeJS, created around 2009, got big around 2011.
* Isaac Schlueter created NPM in January, 2010
* MongoDB was created February, 2009
* Social Media sites made JSON big
* Node was very unstable at first because of how little support it had
* A company called Joyent purchased Node and backed it money
* In 2012 Dahl stepped aside and passed the flag to Isaac
* Currently NodeJS is under the governance of the NodeJS Foundation.

In 2010 Google invited Ryan to speak to their employees about new technologies emerging. Even the announcer that presented him was talking down on his idea of moving Javascript to the server when everyone else was trying to run Java on the client.

*What is V8, exactly?*

* Computers read programming languages with Interpreters, Compilers and Transpilers
* V8 is a Javascript engine which is an Interpreter. V8 compiles Javascript down and executes it.
* SpiderMonkey(Firefox JS Engine) was the first ever JS engine and was written by Brendon Ike. The same person that invented JS.

Compilers: take source code and turn it into an executable file (a file full of code the computer can understand).

Transpilers: take source code and turn it into a source code of another type.

Interpreters: take source code and directly execute it by taking realtime intermediate steps, But, they don’t leave anything behind.

*What is NodeJS, exactly?*

* Server-side JS run-time environment. Looks and JS and decide when and where it should be executed on the computer
* V8 is the car engine and drive train, Nodejs is everything else.
* Node is a C++ application that embeds V8
* Node is invoked with a single file (app.js) then imported files from all over the place makes an application
* REPL === Read, Execute, Print, Loop

Non-blocking tasks (setTimeout()) get added to the to do list during NodeJS event loop. It is possible for the event loop to never end which is what people do for forever running servers.

The concept of non blocking does not allow node to do multiple tasks at once but instead schedule tasks to do later (async vs sync).

**REPL**: Works like the console in a web browser. It is a way to define and execute JS code against the V8 engine in real time.

*Anatomy of Node Application*

* const express = require(‘express’), the header area where we import are called “Dependencies”

*Common Node Conventions*

* Package.lock locks down the exact version of the node modules installed.
* Common Code Comments
  + @Params
  + @TODO
  + @Author
  + @Date
* Throwing errors will kill the entire app. Use with caution.

*Nodejs vs the Browsers*

* Global variables that are in the Browser will not be in node (e.g window.open)
* Nothing related to window, browser object the screen or the browser location make any sense in Node
* Node is a single static environment. Example is browser has “cross browser compatibility”, on the backend there is no such issue.
* The Node source code can not be seen by the viewer. This means you can use certain sensitive information without prying eyes.

**Building a RESTful API:**

*Section Overview*

* Building a RESTful API for an uptime monitoring application
  + This means a user can enter a url and the app will check if the site it up or down
  + User can signup and signin
* Only using Nodes core modules
* We’ll be using the file system to mimic a DB in our computer

*Basic Scaffolding*

* Just testing index.js

*Starting A Server*

* Http server : We first need to define the http module to define what the server does. Later on we need to tell the server to listen to a specific port
* http.createServer(), the method to create server
* After we start it with `node index.js` we can test it by running the command `curl localhost:3000` in another terminal

*Parsing Request Paths*

* Node has a nice helper library for all things related to url functions, called `url`
* const parsedUrl = url.parse(req.url, true); --- this is where we start explaining what is in the createServers callback
  + When the server is created and we have it listen on a port and then a request comes in the (req, res) parameters get filled and is used in the code block of the createServers function everytime the request comes in.
  + So in the case of a user hitting localhost:3000 the createServer function will be called as many times it is loaded. Every time the function is called the (req, res) are new.
* The request (req) object contains information the user is asking for
* **e.g:** www.example.com/**this/is/the/path**

*Parsing Http Methods*

* Added http method [const method = req.method.toLowerCase()] to source code

*Parsing Query Strings*

* Get the query string as an object, The query string will let us understand what the user is requesting or modifying so we need to parse that as well.
* **e.g:** www.example.com/path**?foo=bar**

*Parsing Headers*

* Use req.headers to get the headers from a response. Use a tool like postman to check if it is working

*Parsing Payloads*

* **Payload:** the part of transmitted data that is the actual intended message
* Bring in `string\_decoder` dependency. Declare a new StringDecoder(‘utf8’) [set with encoding of utf8.]
* Node deals heavily with `Streams`. *Streams are bits of information that are coming in a little bit at of time.* Payloads that come as part of an http request, come into the http server as a `Stream`
* We need to collect the stream as it comes in and when the stream tells us its in the end we need to coalesce that into one coherent thing. We are interested in finding out what the entire payload is once it is finished streaming in.
* Before we start catching the payload as it comes in we want to create a new string to hold it. We will call this `buffer`.
* The buffer will start as an empty string and as new streams begin to come in we will append it to the empty string.
* We do this by binding to an event that the request object emits and the event is called data. So when the request object emits the event we want to grab it within a callback
  + req.on(‘data’, (data) => {})
* So when data is being streamed in we append the decoded data onto the buffer. Remember we set the decoder to be utf8. So in all the data is being decoded into utf8 and being appended to the buffer.
* The event that lets us know the stream is done is req.on(‘end’);
* Once the stream(request) is completed we can now do what we were doing before. In this case sending the response and logging the request. We do this by moving the response and logging code into the handler of the `end` event.
* The end event is always called but the data event may not always be called.
* This is how streams are handled in Node. We get bits of streams at a time and let Node know when the stream is finished.

*Routing Requests*

* Now that we have all the data of this request we want to package it up into a nice object and send it on (route it) to some request handlers.
* We are going to route requests based on the path that the user is asking for.
* So if the path that the user is requesting exists as a key on the router, for example if their requesting /sample, sample exists on the router. So that request should be routed to handlers.sample.
* But if the path does not exist, for example /foo, then it will be routed to the notFound handler.
* When constructing the data object to send to the handler, we add a bunch of keys into the object that comprises all the data that we parsed out of the request as we’ve been working with it.

We calling the chosen handler, were sending it a bunch of data which is the same data that we spent time gathering. Then were calling a handler which we know since it is specified in the router. When the handler calls back we expect it to send us a statusCode and a payload. If it doesn’t send either of those or if the status code isn’t a number or if the payload isn’t an object we default them to 200 and an empty object respectively. Then we convert the payload to a string using JSON.stringify. We go ahead and return the status code to the user and the payload to the user as a string and then we log out what we did.

*Returning JSON*

* We are going to set the headers to let the user know that it’s content-type is `application/json`,

we can use res.setHeader();

* By specifically setting it to `application.json` the payload response the user gets back will be in JSON

*Adding Configuration*

* Were going to add a config file to store different configuration variables, so we can start the app in different ways for different environment
* We make a new file in the same dir name config.js. The config.js is required inside index.js. Within the config file we will define different environments/varialbes and export only the environment/variables that is called in the terminal
* There is a global variable, which we call NODE\_ENV, that we setup in the config file. it’s achieved by using process.env.NODE\_ENV.
* In the terminal is we pass in the command `NODE\_ENV=staging node index.js` it will start in staging mode based on how we setup the config.js file.

*Adding HTTPS Support*

* We get a SSL certificate and put it into the https folder. We change the config file to have it both `env` and `prod` mode to have a port for http and https.
* We create a function to handle both http and https server `unifiedServer()`.
* When creating the https server we need to create an object with options to the SSL keys that were generated with OpenSSL. To read the file we need to bring in nodes `filesystem`
* After this we have 2 different servers (one secure, one not) that are listening on 2 different ports but passing all the meat of the logic into the `unifiedServer()`

*Service 1: /ping*

* At this point we have enough of the scaffolding built and we can begin building on the production routes
* We will build a route /ping to check the status (200) of the app

*Storing Data*

* We make a .data (hidden folder). We do this because we want to make it clear that the files inside are not logic but files being written for the app to run
* require(‘path’) is used to normalize the path to different directories
* We made a library with the FS that performs CRUD operations with json files

*Service 2: /users*

* \_users is a style of writing code that indicates it is going to be a private method for `this`, it’s not something that is calling this library that will use directly
* It’s a lot of stuff, look at the source code

*Service 3: /tokens*

* The concept is very similar to users but creating the process to create and validate tokens. We wrap the validation inside some of the user’s methods to verify the user is logged in.

*Service 4: /checks*

* A check is a task that tells our system go check this url every X number of seconds and then tell the user if the url is up or down
* Here we implemented checks by checking a url. Everytime a check is executed we store it in a directory along with the user who performed the checks information.

*Connecting to an API*

* We are going to be integrating with Twilio. This API allows you to do stuff with phone numbers. We will be able to send SMS to users with this API
* We create a library to integrate with Twilio in the helpers.js

*Background Workers*

* We need to refactor the code of index.js to be much smaller. We are making it so that when the server is started it will call a server file and also call a worker file to start up the new workers we will be writing
* We move all the logic that was currently in the index file to the new lib/server.js file. Were pretty much splitting the tasks to a server file and a worker file. In the index we will initiate and execute both files.
* The workers are going to perform all the checking that has been configured by all the users. One of the things that workers will do is gather up all the checks. We are storing all the checks as separate files in a directory. So one of the thing workers will do is list all of the available checks in that directory.
* We add a list function onto our data library
* Workers are scripts that run in the background consistently. In our app our worker checks once a minute to see if the sites to check are down or up. If down the worker updates it to up and vice versa. The worker also texts the user based on logic and condition of the status change.

*Logging to Files*

* We will be adding features for logging with access control. We will be adding an admin to see and use logs easier.
* We will be logging physical files through the fs file. The same log we get in the console from the worker.js, we will be sending it to actual log files. We will eventually write a worker to rotate the logs and compress them through the fs
* Workers.log is more of a private function that is only used within itself so we do not need to do sanity checking
* We use zlib a built in library to compress files. It is easily compressible because there is a lot of repeat data.

*Logging to Console*

* We use the util.debug module to change the console.logs to debug. With this if we start up node with NODE\_DEBUG=nameOfDebug node index.js. The console will only show for the module with that command

*Section Review*

* Congrats we built a raw API with NodeJS!!! All other sections should be easier
* Check out `Command Line Options` in the node docs for useful terminal options
* We used the `Crypto` user to hash passwords, you can see the docs
* We used `filesystem` it allows you to work with the computers filesystem
* We used `Globals`, which do not need to be required but are available to you globally, e.g. \_\_dirname && module.exports
* We used `http && https`, these allow us to create new servers, listen on ports and make http requests that can be sent to endpoints
* A section called `Modules` in the doc is useful to research what modules can generally do in node
* We used `path` which helps resolve and normalize our path to make it clean enough to read a certificate or write into the file system
* We used `process`, process was used to determine the env when starting
* We used `querystring` which took a url that was full of path and query string parameters and split it into pieces
* We used `String Decoder` to get the payload parsed that the user has been sending in.
* We used `Timers`, does not need to be required and it exposes some globals to us like setTimeout, setInterval and so on.
* We used `url` which we used to parse out the hostname or pathname from the url
* We used `utilities`, we used it to use the debug tool from it to help structure our debug to be consoled based on how we start node
* We used `Buffer` which we do not need to require but we compressed and decompressed a file by moving a string into a buffer and then back into a string. Also when we were decoding that same file we went from a base64 string up into a new buffer and then back into a plain text string. This is useful for streaming data, payload data and this is used a lot
* We used `zlib` to compress files by using gzip
* We used `Stream` module when it pertained to the incoming payload that we were parsing when users make a request or a post with a JSON payload, we stream it in a little bit at a time.

***Extra Notes:***

**Events & Handlers:** In programming, an event is an action that occurs as a result of the user or another source, such as a mouse being clicked, or a key being pressed. An event handler is a routine that is used to deal with the event, allowing a programmer to write code that will be executed when the event occurs.

**Building a Web App GUI:**

*Overview*

* Presents a GUI to the user
* User can sign up, login/logout
* Modify account / delete
* CRUD
* View Dashboard

*Refactoring for a GUI*

* We changed the router and put it in its own path. e.g. api/users
* We refactor the chosen response to change based on the type of response. So if JSON response application/json and html response is text.html