**NodeJS**

**How to start** with Node.js after I installed it?

Once you have installed Node, let's try building our first web server. Create a file named "app.js", and paste the following code:

const http = require('http');

const hostname = '127.0.0.1';

const port = 3000;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello World\n');

});

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

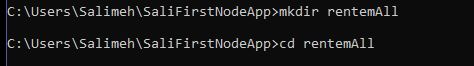
After that, run your web server using node app.js, visit [http://localhost:3000](http://localhost:3000/), and you will see a message 'Hello World'

## How to Initialise

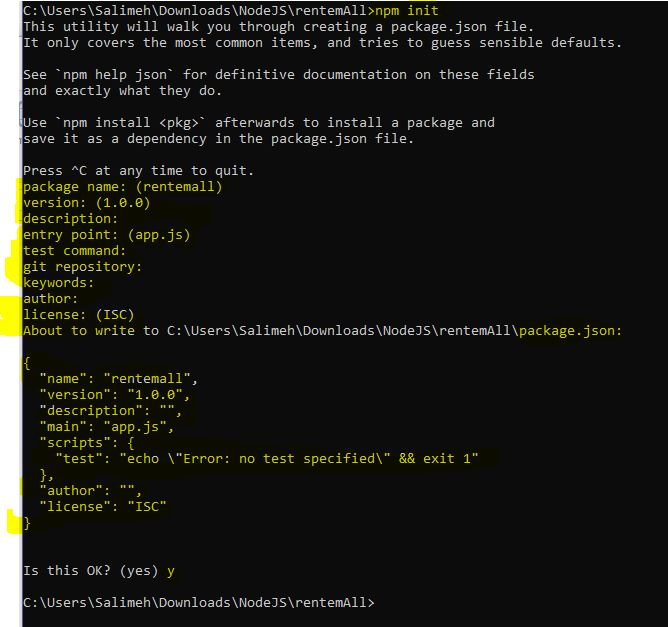
You can follow the code snippets in this post to build up the basis of a simple website or head over to [**this repo**](https://github.com/bengourley/basic-express-site-2016) to check out the entire codebase.

mkdir simple-website-2016 && cd simple-website-2016

npm will prompt you for a bunch of info, so feed it something appropriate.

****

npm init

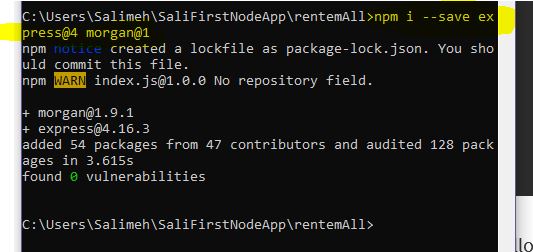
****

**Express** has stood the test of time and is still the module I’d reach for to build a simple website in Node.js. Over the last four years, unlike many long-lived modules, Express has trimmed the fat and pushed lots of functionality out of core and into external modules. This means it’s more lightweight than ever, and doesn’t load you up with a ton of bloat that you’ll never need. The only caveat of this is that you need to explicitly install certain bits which used to be bundled in, for instance the logger (now called morgan):

npm i --save express@4 morgan@1

Like Express, over the years my preference remains uneroded for Jade and Stylus as my template language and css preprocessor:

npm i --save jade@1 stylus@0

****

Like Express, over the years my preference remains uneroded for Jade and Stylus as my template language and css preprocessor:

npm i --save jade@1 stylus@0

The biggest change is the approach to rendering/compilation of the Jade and Stylus. Previously, for rendering Jade templates the Express “view engine” was used. For Stylus, the compilation “middleware” was added to the server. Both of these mechanisms mean that the assets are rendered/compiled “on-demand” – i.e. when a request for the compiled assets reaches the server. There are a couple of problems with this:

1. If you have an error (syntax or runtime) in your Jade/Stylus, you won’t find out until the a request is made to the server requiring those compiled assets. For frequently visited pages, this isn’t a huge problem, but you could have your website running for hours, even days before someone hits a page which renders an erroneous.
2. The first request to hit the server incurs some latency while the assets are compiled. While subsequent requests will serve a cached asset and won’t trigger a recompile, they still have to wait for the filesystem when checking if any files were modified.
3. Architecturally, this is a poor separation of concerns. A web server becomes a lot easier to test, maintain and reason about if its job is to serve content. Adding the responsibility of compiling static assets (tasks which makes more sense in a “build” process) introduces coupling and complexity.

**Build Process**

If your build process becomes unwieldy, a tool like grunt, gulp or pliers can help to organise things. But equally (especially with grunt and gulp) their configuration and setup can be equally unwieldy, and significantly bewildering.

Replace the scripts section of your package.json with the following:

{

"build-css": "stylus source/stylesheets/index.styl -o static/css",

"watch-css": "stylus source/stylesheets/index.styl -o static/css -w",

"clean": "rm -rf static/css && mkdir -p static/css",

"build": "npm run clean && npm run build-css",

"watch": "npm run clean && npm run watch-css & nodemon server -e js,jade",

"start": "node server"

}

*>>>>>>>>>>>>>>>>>>>>> I did save a backup for “ package-Backup-Sali.json “*

*Note:* if you’re on Windows, the clean task needs to use different path separators…

rm -rf static\\css && mkdir -p static\\css

Thanks to Timothy Trowbridge for pointing this out.

I’ll explain each command in detail. The reason to put them in the package.json; is simply to give them aliases. The benefits are three-fold: npm allows you to use locally installed modules without typing node\_modules/.bin; you don’t have to remember how to type the entire command each time; and it means that you can freely change the contents of the command (you could, for example, change the build-css command to use Sass instead of Stylus, and nobody else on your team would need to know – they continue building their css blissfully unaware).

Running commands is simple: npm run <name>

* npm run build-css – this uses the Stylus CLI to compile the index.styl stylesheet to css, and place it in the static/css directory
* npm run watch-css – this is exactly the same as the previous command, except that it will continue running until manually stopped, compiling the stylesheet any time the source files are change. This task essentially supersedes the stylus middleware I mentioned earlier.
* npm run clean – this removes any built files (currently only css, but it might later include browserify-ed JS) and creates any required directories.
* npm run build – this does everything required for the server to run correctly, which is just to run the clean and build-css commands
* npm run watch – this is the command that is most useful in development. It will watch the entire project for changes and recompile assets or restart the server accordingly.
* npm run start – this simply starts the server and does no watching at all.

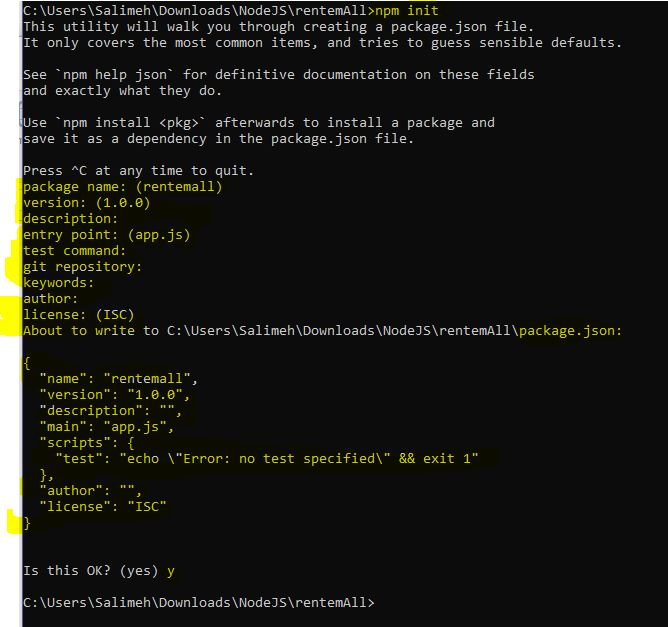
There was one additional dependency introduced here – nodemon. It does a great job at watching files and restarting on change, so install it like so:

npm i --save nodemon@1.9

Here’s **how to use MySQL in Node** in 5 easy steps:

1. Create a new project: mkdir mysql-test && cd mysql-test
2. Create a package.json file: npm init –y
3. Install the mysql module: npm install mysql –save
4. Create an app.js file and copy in the snippet below.
5. Run the file: node app.js. Observe a “Connected!” message.

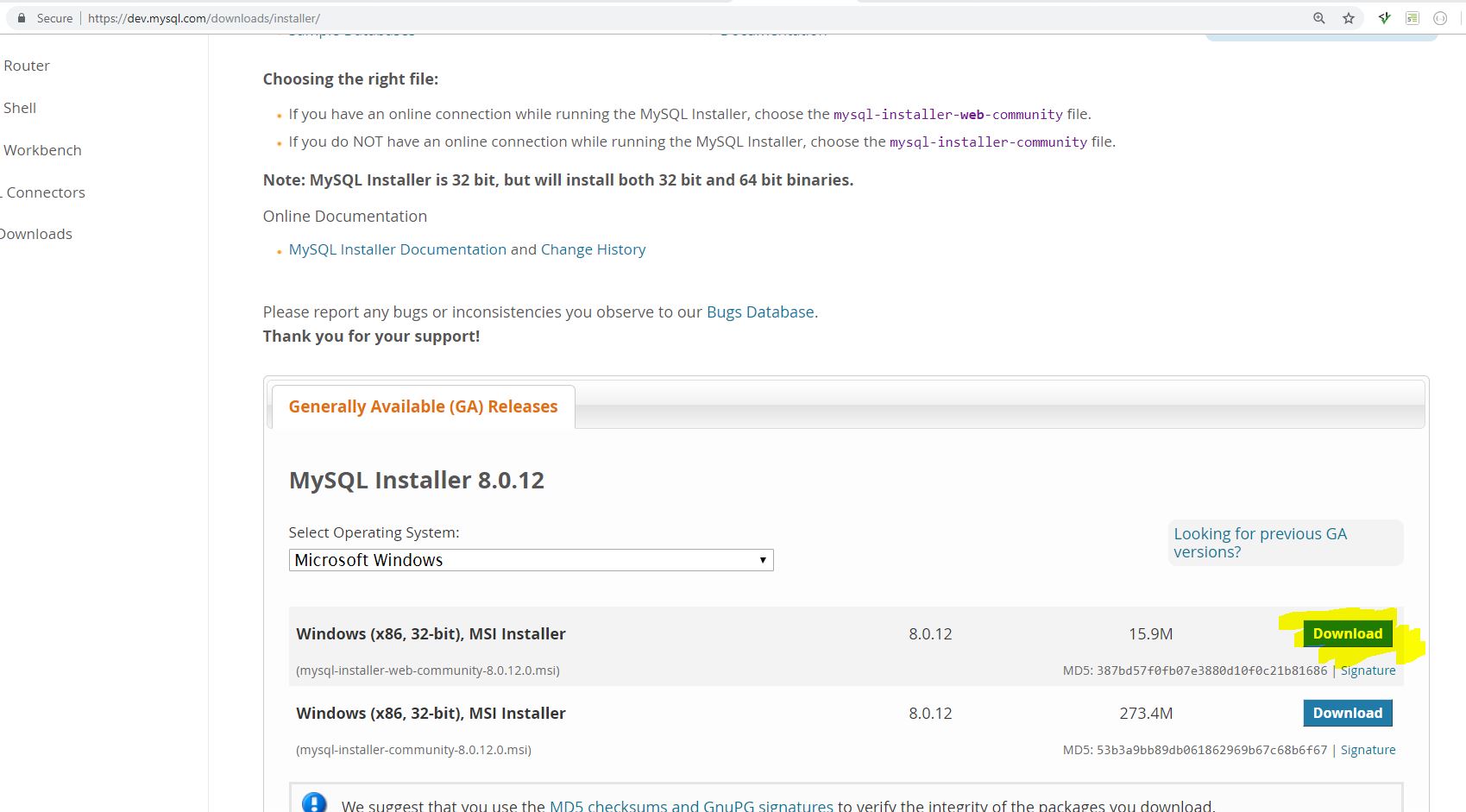
**2.** npm init –y npm will use only defaults and not prompt you for any options

****

**From MySQL**

Download MySQL application from Oracle site after signing in.

[**https://dev.mysql.com/downloads/installer/**](https://dev.mysql.com/downloads/installer/)

****

**then login and download**

****

**After installation on the machine:**

**create app.js and put**

****

**node app.js**

**gives error**

**Go to MySQlWOrkbench**

To see if root has all the privilege

“ SHOW GRANTS FOR 'admin'@'localhost'; “

“ GRANT ALL PRIVILEGES ON \*.\* TO ‘root’@'localhost' WITH GRANT OPTION; “

run the above command to see if it worked

“ SHOW GRANTS FOR 'root'@'localhost'; “

To create more users:

“ CREATE USER ‘sali’@'localhost' IDENTIFIED BY 'Group8'; “

“ GRANT ALL PRIVILEGES ON \*.\* TO 'sali'@'localhost' WITH GRANT OPTION; “

“ SHOW GRANTS FOR 'root'@'localhost'; “

“ CREATE USER ‘root’@'localhost' IDENTIFIED WITH mysql\_native\_password BY 'Group8'; “

If needed:

“ DROP USER ‘root’ “

“ DELETE FROM rentemall.user

WHERE user = 'jack'; “

“ INSERT INTO mysql.USER (Host, User, Password) VALUES ('localhost', 'root', password('Group8')); “

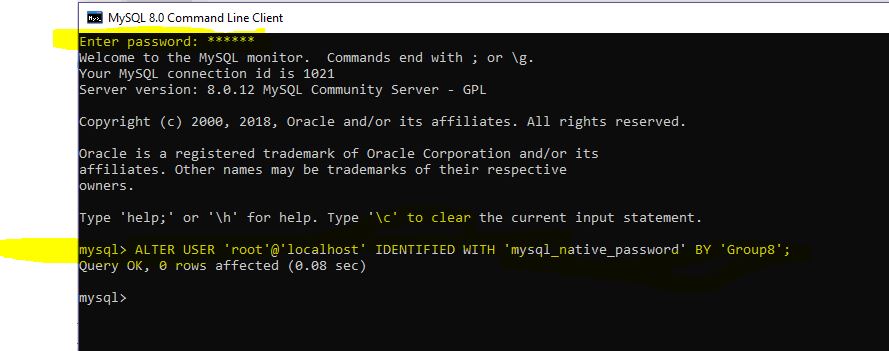
“ INSERT INTO mysql.user (Host, User, Password) VALUES ('%', 'root', password('YOURPASSWORD'));

GRANT ALL ON \*.\* TO 'root'@'%' WITH GRANT OPTION; “

Go to MySQL 8.0 Command Line Client applicatrion

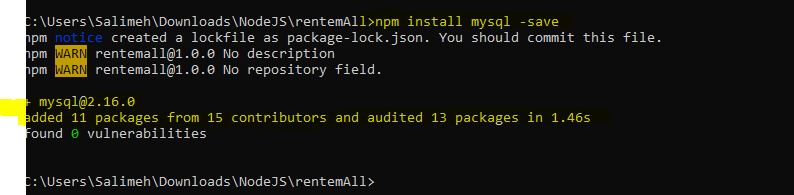
Type password: Group8

“ ALTER USER 'root'@'localhost' IDENTIFIED WITH 'mysql\_native\_password' BY 'Group8'; “



“ server.js “ which has connection information must be in the same root folder and not inside “ Node\_module “ folder

npm install mysql –save

****

### // **app.js**

### const mysql = require('mysql');

### // console.log('Sali MUST make it!!!!');

### 

### console.log('Get connection ...');

### 

### var conn = mysql.createConnection({

### database: 'prj566\_182a08',

### host: "zenit.senecac.on.ca",

### path: '/phpMyAdmin/',

### user: "prj566\_182a08",

### password: "jaMW2249"

### });

### 

### conn.connect(function(err) {

### if (err) throw err;

### console.log("Connected!");

### var sql = "CREATE TABLE PRJ\_666 (course\_name VARCHAR(15), course\_code VARCHAR(10))";

### conn.query(sql, function (err, result) {

### if (err) throw err;

### console.log("Table created");

### });

### });

### 

### Using Grunt to Watch the Files for Changes

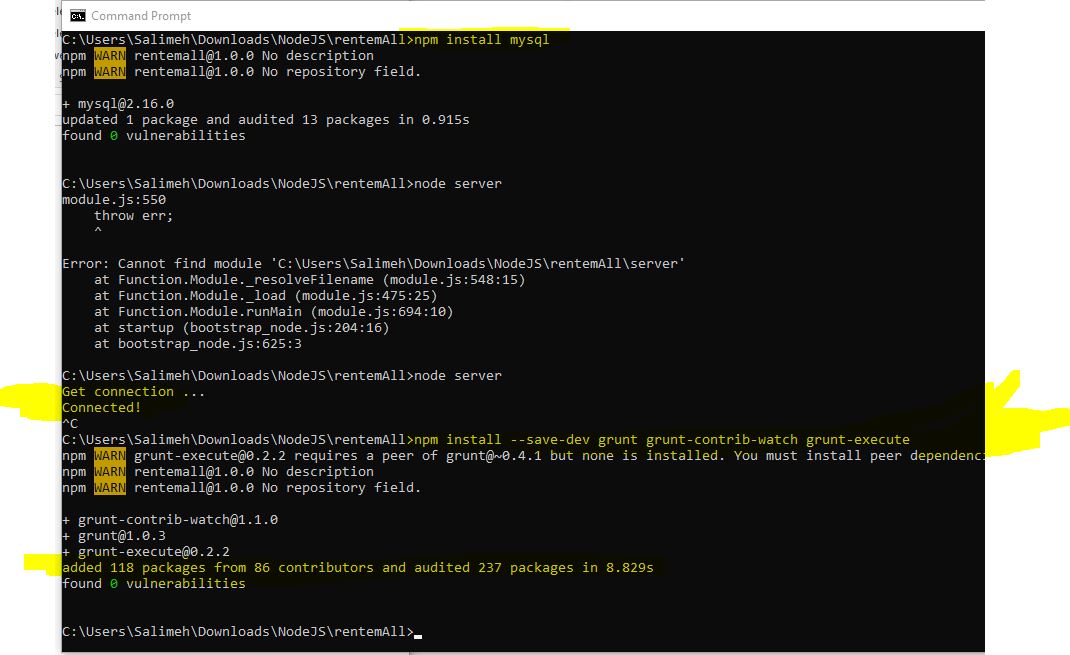
Running node app.js by hand every time we make a change to our code is going to get a bit tedious, so let’s automate that. This part isn’t necessary to follow along with the rest of the tutorial, but will certainly save you some keystrokes.

Let’s start off by installing a couple of packages:

**npm install --save-dev grunt grunt-contrib-watch grunt-execute**

[Grunt](http://gruntjs.com/) is the well-know JavaScript task runner, [grunt-contrib-watch](https://www.npmjs.com/package/grunt-contrib-watch) runs a pre-defined task whenever a watched file changes, and [grunt-execute](https://www.npmjs.com/package/grunt-execute) can be used to run the node app.js command.

Once these are installed, create a file called Gruntfile.js in the project root and add the following code.



// Gruntfile.js - create this file in the root directory and not inside “ node\_module “

module.exports = (grunt) => {

grunt.initConfig({

execute: {

target: {

src: ['app.js']

}

},

watch: {

scripts: {

files: ['app.js'],

tasks: ['execute'],

},

}

});

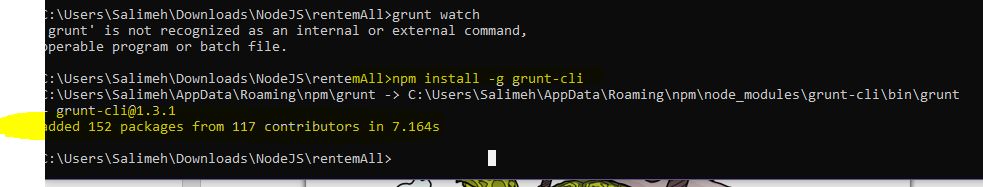
grunt.loadNpmTasks('grunt-contrib-watch');

grunt.loadNpmTasks('grunt-execute');

};

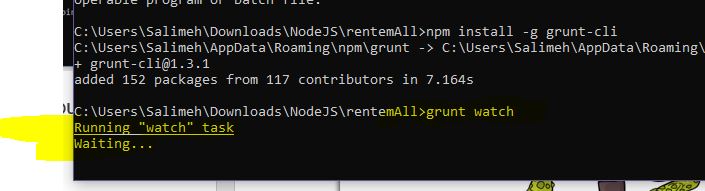
or use

**npm install -g grunt-cli**

****

Now run **grunt watch** and make a change to **app.js**. Grunt should detect the change and re-run the node app.js command.

**grunt watch**



How to **run html file**:

Create html file and called it “ demoIO.html “

<html>

<body>

<h1>Sali Header is here :D</h1>

<p>My paragraph.</p>

</body>

</html>

Create and js file and called it “ demo\_readfile.js “

var http = require('http');

var fs = require('fs');

http.createServer(function (req, res) {

//Open a file on the server and return it's content:

fs.readFile('demoIO.html', function(err, data) {

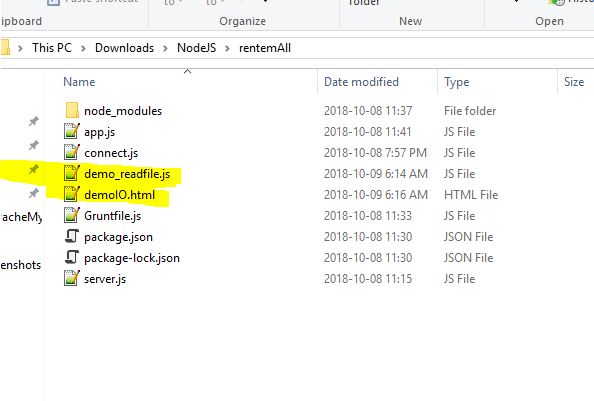
res.writeHead(200, {'Content-Type': 'text/html'});

res.write(data);

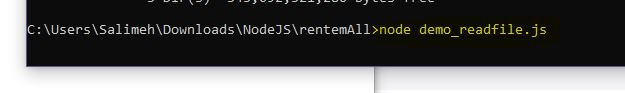
return res.end();

});

}).listen(8080);

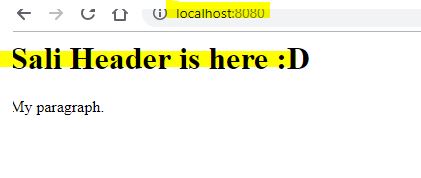


Then “ node demo\_readfile.js “



then at

[**http://localhost:8080/**](http://localhost:8080/)

****

Create Files

The File System module has methods for creating new files:

* fs.appendFile()
* fs.open()
* fs.writeFile()

The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be created:

Create a new file using the appendFile() method:

put this code in “ demo\_fs\_append.js “

var fs = require('fs');

//create a file named mynewfile1.txt:

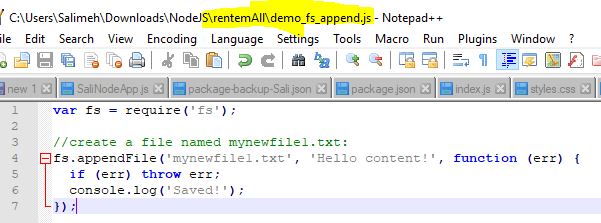
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {

if (err) throw err;

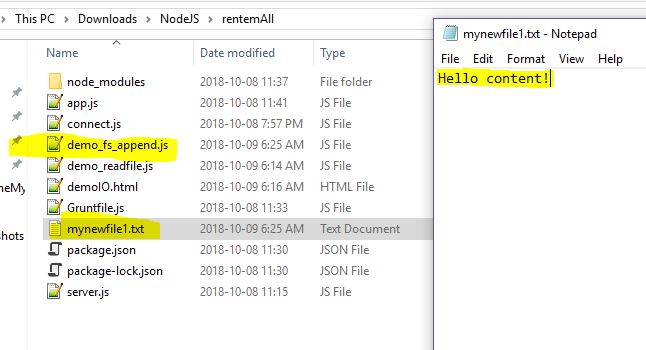
console.log('Saved!');

});

and then run it with “ node demo\_fs\_append.js “

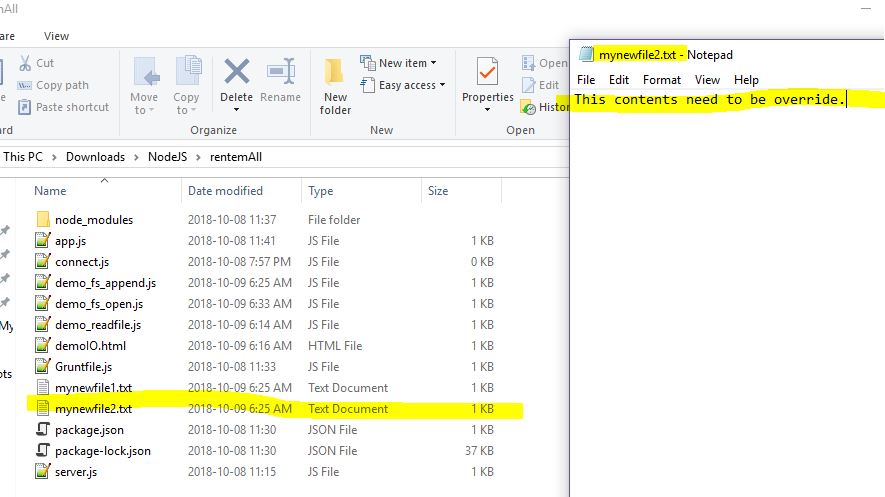


It automatically creates “ mynewfile1.txt “ and put the text “ Hello content “ inside it.



The fs.open() method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

create “ demo\_fs\_open.js” \*\*\* “ mynewfile2.txt “ must exist before being write to so create “ mynewfile2.txt “ and then put the code in “ demo\_fs\_open.js “



var fs = require('fs');

//create an empty file named mynewfile2.txt:

fs.open('mynewfile2.txt', 'w', function (err, file) {

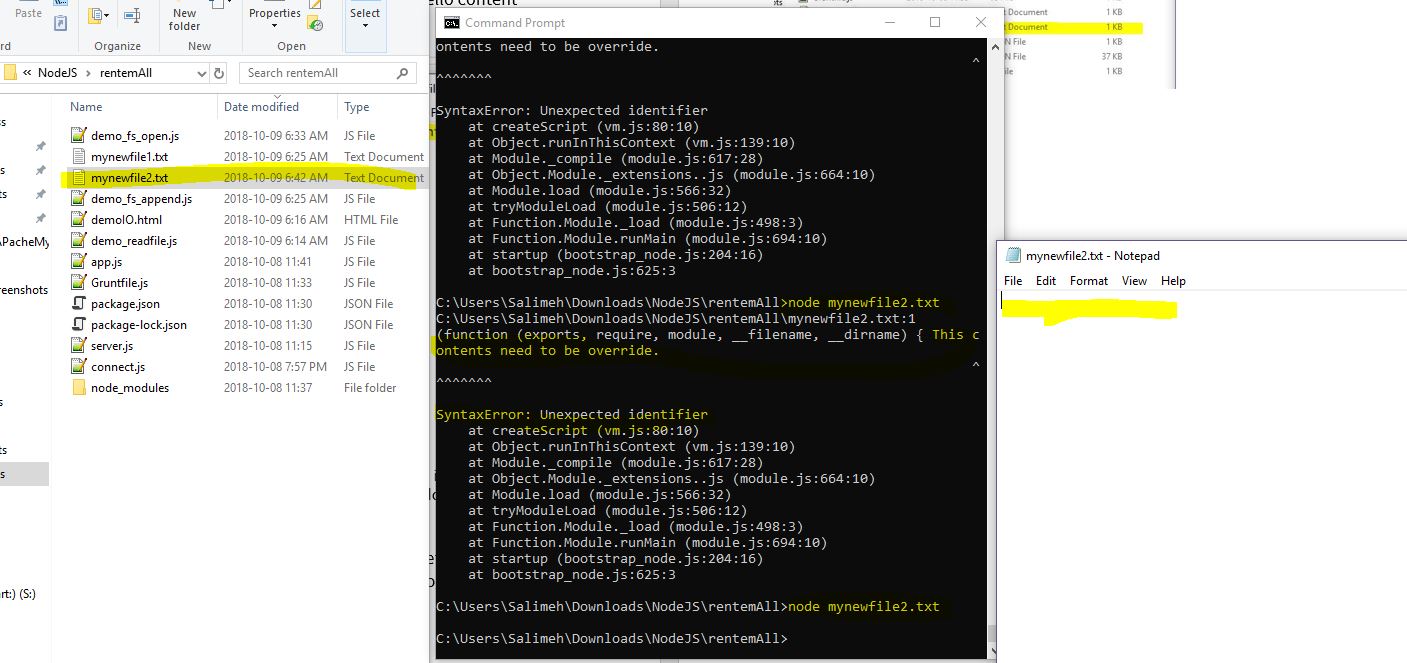
if (err) throw err;

console.log('Saved!');

});

Then run it

Alright, we got error because the txt file must be empty, no text inside it



The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created.

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {…

## Update Files

The File System module has methods for updating files:

* fs.appendFile()
* fs.writeFile()

The fs.appendFile() method appends the specified content at the end of the specified file:

### **Example**

Append "This is my text." to the end of the file "mynewfile1.txt":

var fs = require('fs');  
  
fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {  
  if (err) throw err;  
  console.log('Updated!');  
});

The fs.writeFile() method replaces the specified file and content:

### **Example**

Replace the content of the file "mynewfile3.txt":

var fs = require('fs');  
  
fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {  
  if (err) throw err;  
  console.log('Replaced!');  
});

## Delete Files

To delete a file with the File System module,  use the fs.unlink() method.

The fs.unlink() method deletes the specified file:

### **Example**

Delete "mynewfile2.txt":

var fs = require('fs');  
  
fs.unlink('mynewfile2.txt', function (err) {  
  if (err) throw err;  
  console.log('File deleted!');  
});

## Rename Files

To rename a file with the File System module,  use the fs.rename() method.

The fs.rename() method renames the specified file:

### **Example**

Rename "mynewfile1.txt" to "myrenamedfile.txt":

var fs = require('fs');  
  
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {  
  if (err) throw err;  
  console.log('File Renamed!');  
});

# Node.js URL Module

## The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the require() method:

var url = require('url');

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

Create a file called “ demo\_url.js “ then save the code below

var url = require('url');

var adr = 'http://localhost:8080/default.htm?year=2017&month=february';

//Parse the address:

var q = url.parse(adr, true);

/\*The parse method returns an object containing url properties\*/

console.log(q.host);

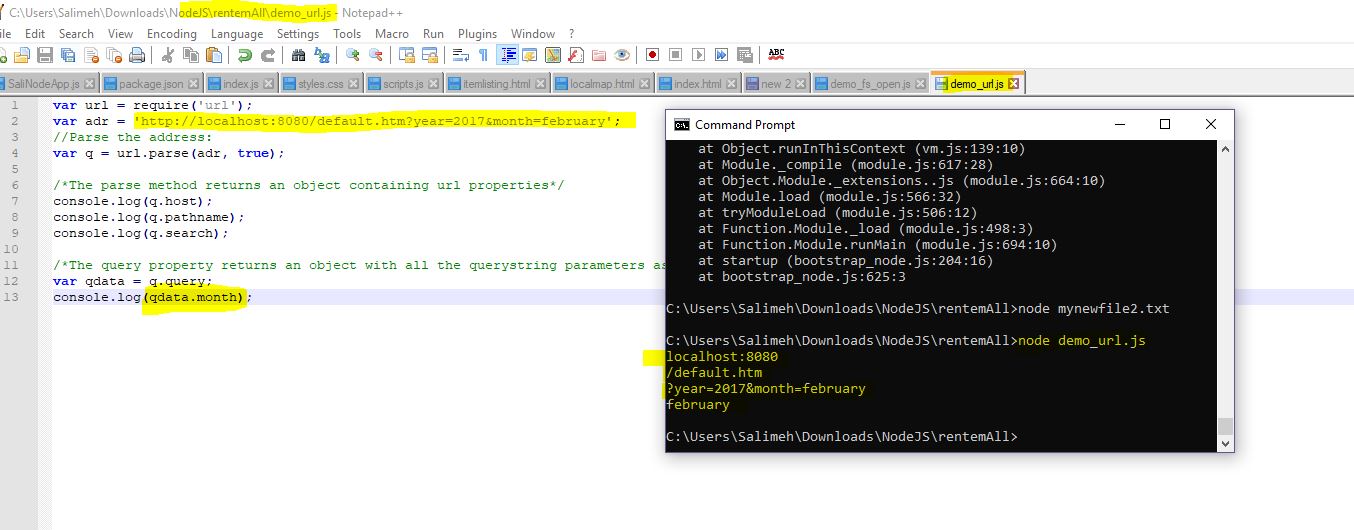
console.log(q.pathname);

console.log(q.search);

/\*The query property returns an object with all the querystring parameters as properties:\*/

var qdata = q.query;

console.log(qdata.month);

****

## Node.js File Server

Now we know how to parse the query string, and in the previous chapter we learned how to make Node.js behave as a file server. Let us combine the two, and serve the file requested by the client.

Create two html files and save them in the same folder as your node.js files.

summer.html

<!DOCTYPE html>  
<html>  
<body>  
<h1>Summer</h1>  
<p>I love the sun!</p>  
</body>  
</html>

winter.html

<!DOCTYPE html>  
<html>  
<body>  
<h1>Winter</h1>  
<p>I love the snow!</p>  
</body>  
</html>

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

**demo\_fileserver.js:**

var http = require('http');  
var url = require('url');  
var fs = require('fs');  
  
http.createServer(function (req, res) {  
  var q = url.parse(req.url, true);  
  var filename = "." + q.pathname;  
  fs.readFile(filename, function(err, data) {  
    if (err) {  
      res.writeHead(404, {'Content-Type': 'text/html'});  
      return res.end("404 Not Found");  
    }    
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(data);  
    return res.end();  
  });  
}).listen(8080);

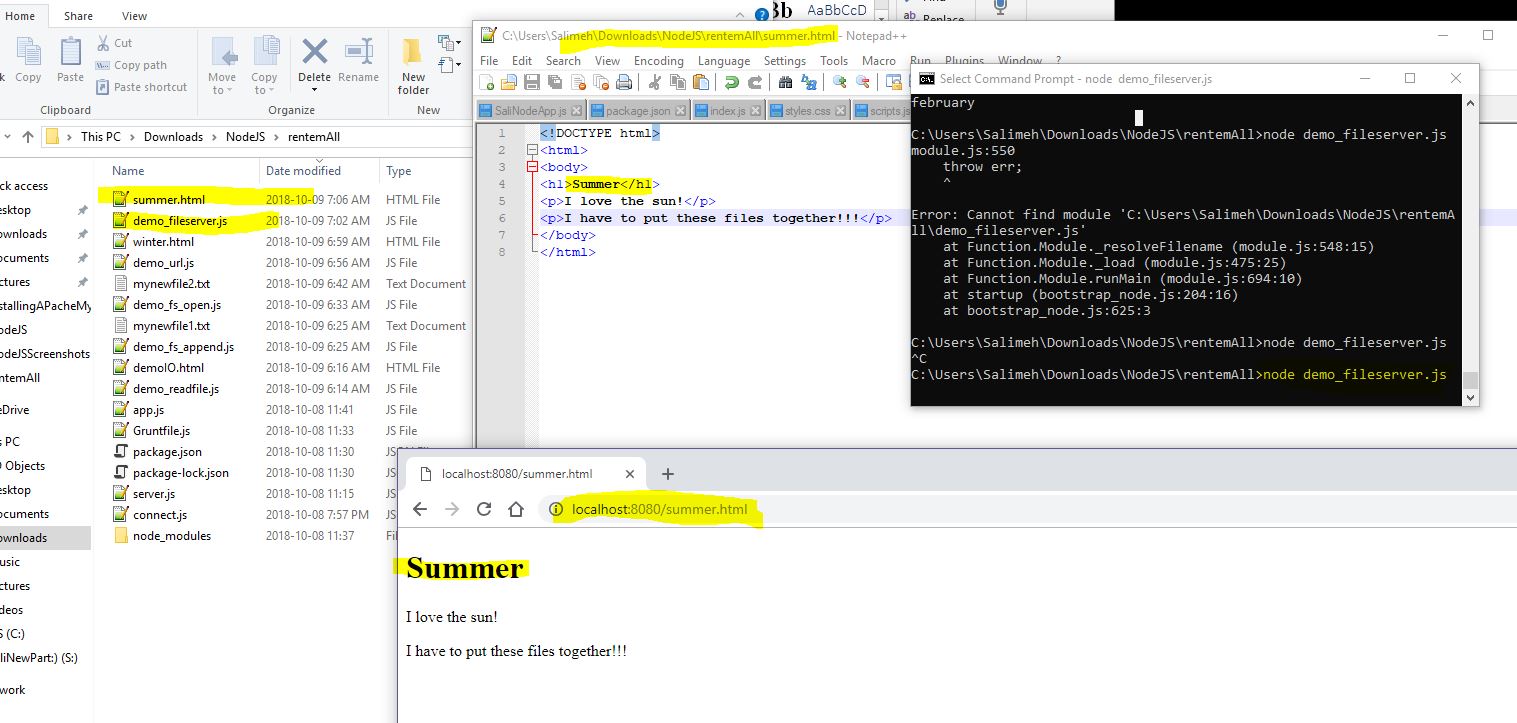
Remember to initiate the file:

Initiate demo\_fileserver.js:

C:\Users\Your Name>node demo\_fileserver.js

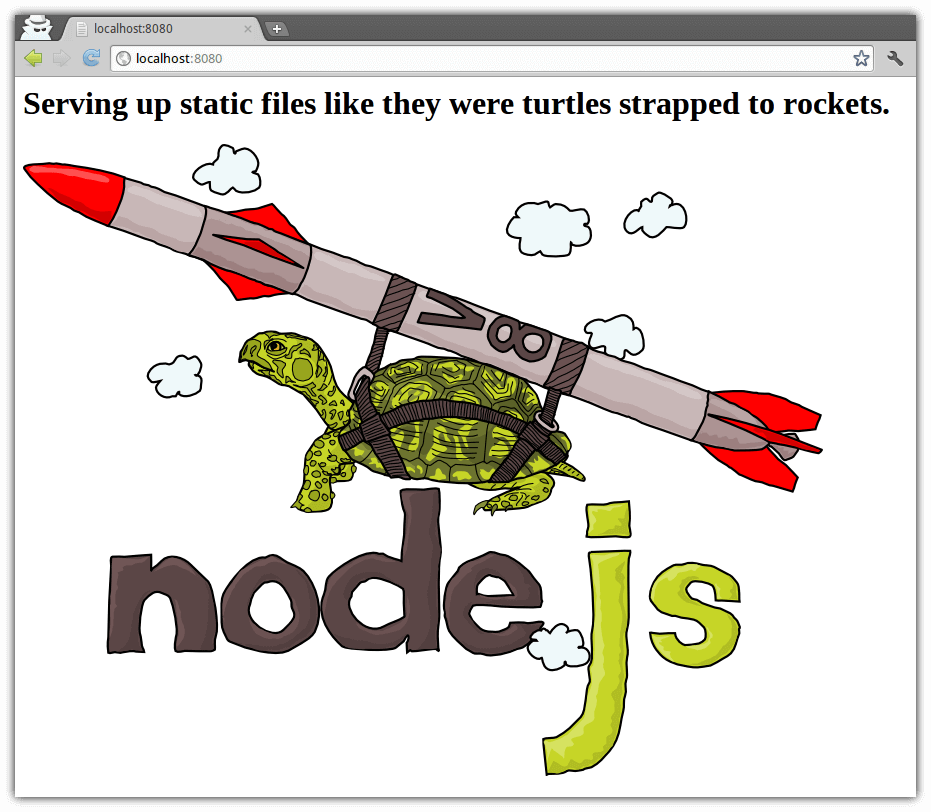
If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

<http://localhost:8080/summer.html>

****

**http-server: a command-line http server**

http-server is a simple, zero-configuration command-line http server. It is powerful enough for production usage, but it's simple and hackable enough to be used for testing, local development, and learning.



**Installing globally:**

Installation via npm:

npm install http-server -g

This will install http-server globally so that it may be run from the command line.

**Usage:**

http-server [path] [options]

[path] defaults to ./public if the folder exists, and ./ otherwise.

*Now you can visit*[*http://localhost:8080*](http://localhost:8080/)*to view your server*

**Available Options:**

-p Port to use (defaults to 8080)

-a Address to use (defaults to 0.0.0.0)

-d Show directory listings (defaults to 'True')

-i Display autoIndex (defaults to 'True')

-g or --gzip When enabled (defaults to 'False') it will serve ./public/some-file.js.gzin place of ./public/some-file.js when a gzipped version of the file exists and the request accepts gzip encoding.

-e or --ext Default file extension if none supplied (defaults to 'html')

-s or --silent Suppress log messages from output

--cors Enable CORS via the Access-Control-Allow-Origin header

-o Open browser window after starting the server

-c Set cache time (in seconds) for cache-control max-age header, e.g. -c10 for 10 seconds (defaults to '3600'). To disable caching, use -c-1.

-U or --utc Use UTC time format in log messages.

-P or --proxy Proxies all requests which can't be resolved locally to the given url. e.g.: -P [http://someurl.com](http://someurl.com/)

-S or --ssl Enable https.

-C or --cert Path to ssl cert file (default: cert.pem).

-K or --key Path to ssl key file (default: key.pem).

-r or --robots Provide a /robots.txt (whose content defaults to 'User-agent: \*\nDisallow: /')

-h or --help Print this list and exit.

**Development**

Checkout this repository locally, then:

$ npm i

$ node bin/http-server

*Now you can visit*[*http://localhost:8080*](http://localhost:8080/)*to view your server*

You should see the turtle image in the screenshot above hosted at that URL. See the ./publicfolder for demo content.