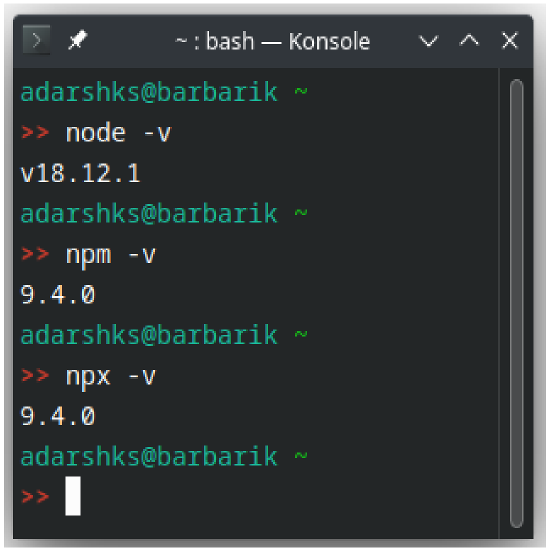
Creating First NodeJS Server

Table of Contents

npm - Node Package Manager

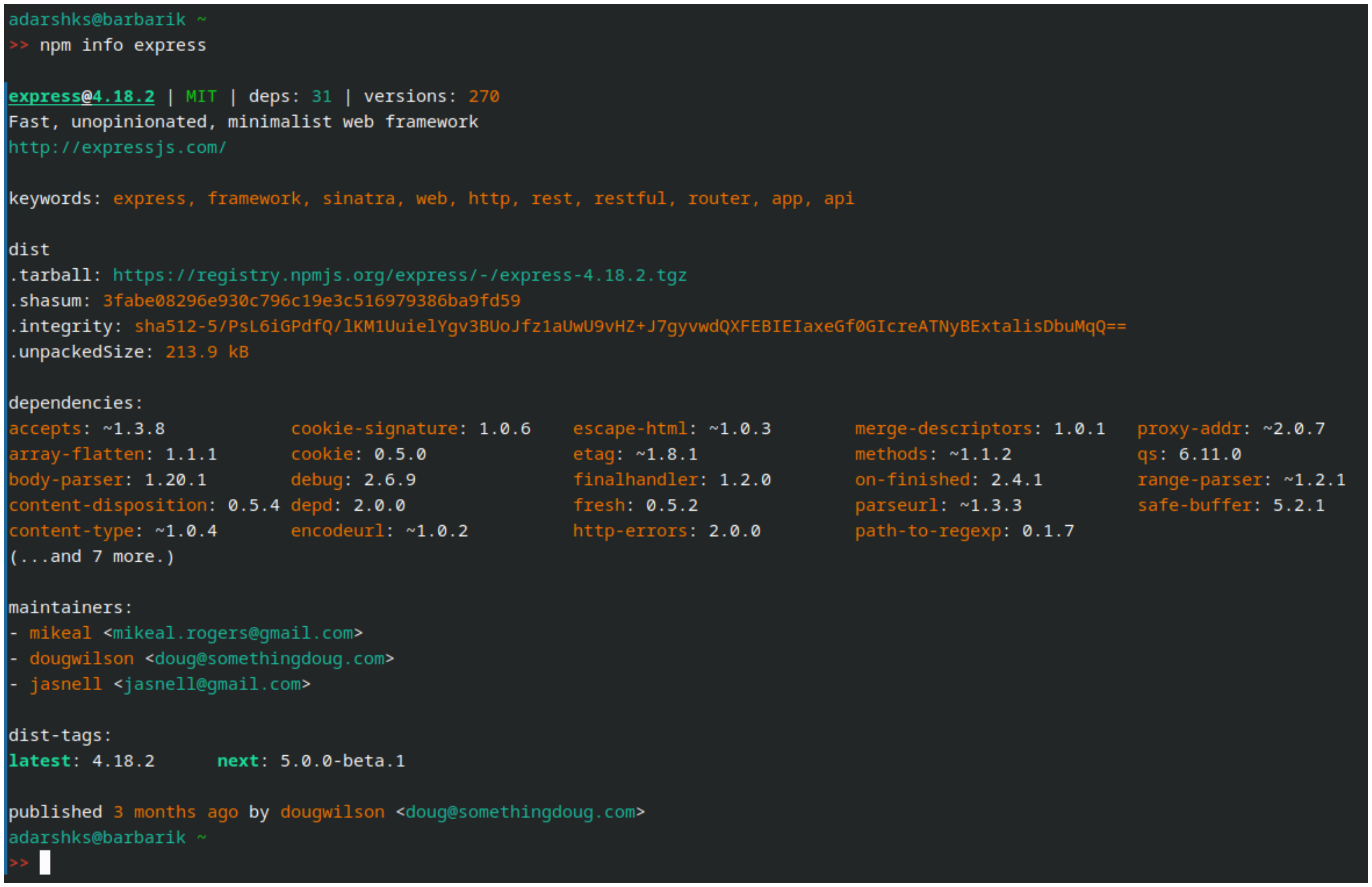
npm (Node Package Manager) is the default package manager for the JavaScript runtime environment Node.js. It's a command line tool that allows developers to easily download, install, and manage packages (libraries and tools) for her Node.js project.
npm is the world's largest software package repository containing over 1 million packages. npm makes it easy for developers to find and reuse existing code instead of creating everything from scratch. npm makes it easy to manage dependencies and keep track of packages used in your project. It also provides a mechanism for sharing packages with other users.

npm is the default package manager for the JavaScript runtime environment Node.js
npm is a must-have tool for developers working with Node.js and is widely used for building web applications, command-line tools, and other types of applications. The npm registry is open source, and developers can publish their own packages to the registry for others to use.
You can test the version of node installed in the system using `` `node -v` `` or
`` `node --version` ``



```
adarshks@barbarik ~
>> node -v
v18.12.1
adarshks@barbarik ~
>> npm -v
9.4.0
adarshks@barbarik ~
>> npx -v
9.4.0
adarshks@barbarik ~
>>
```

You can get information about a particular package using the command `npm info` like `npm info express`

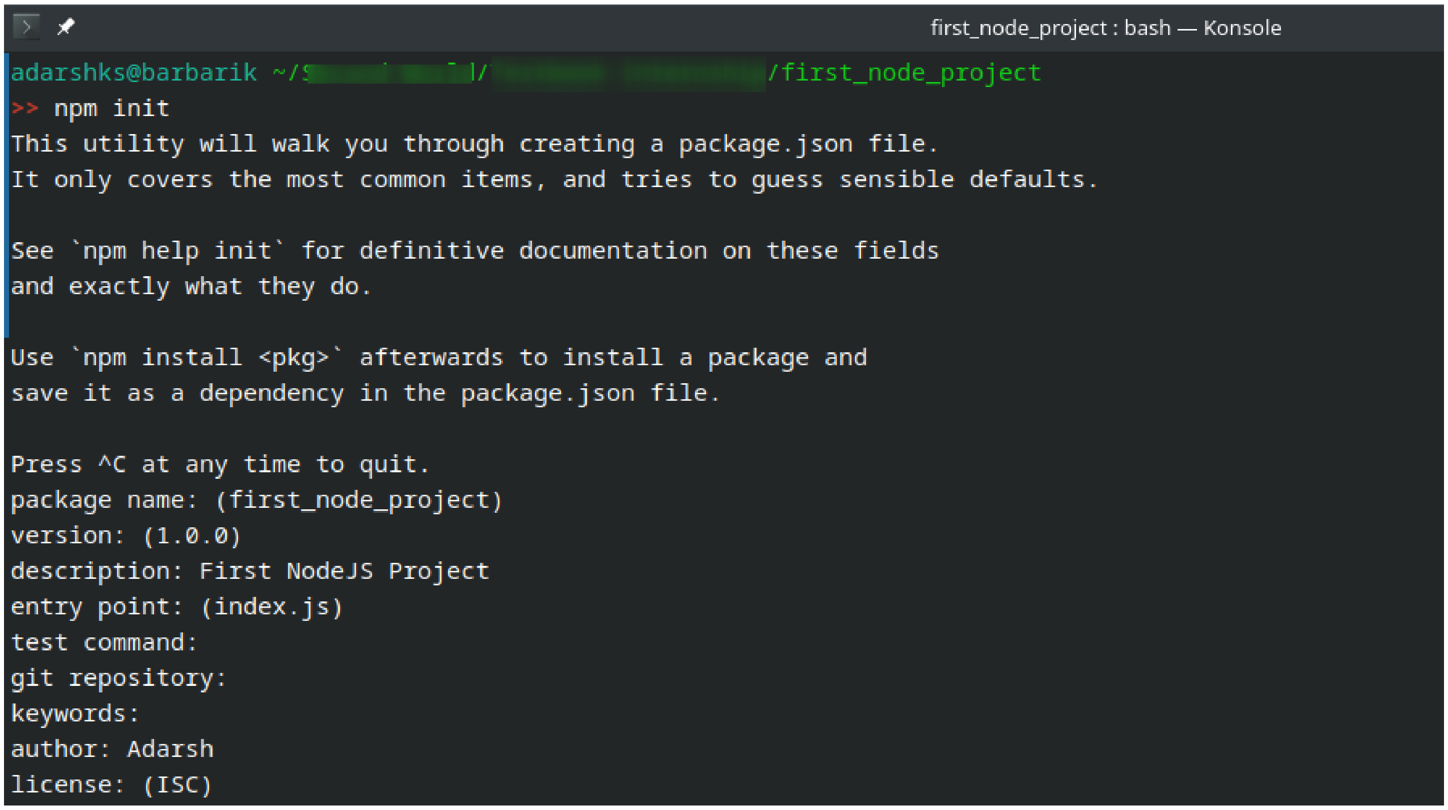


```
adarshks@barbarik ~
>> npm info express

express@4.18.2 | MIT | deps: 31 | versions: 270
Fast, unopinionated, minimalist web framework
http://expressjs.com/

keywords: express, framework, sinatra, web, http, rest, restful, router, app, api

dist
.tarball: https://registry.npmjs.org/express/-/express-4.18.2.tgz
.shasum: 3fabe08296e930c796c19e3c516979386ba9fd59
.integrity: sha512-5/PsL6iGPdfQ/lKM1UuielYgv3BUoJfz1aUwU9vHZ+J7gyvwdQXFEBIEIaxeGf0GIcreATNyBExtalisDbuMqQ==
.unpackedSize: 213.9 kB

dependencies:
accepts: ~1.3.8            cookie-signature: 1.0.6    escape-html: ~1.0.3        merge-descriptors: 1.0.1   proxy-addr: ~2.0.7
array-flatten: 1.1.1       cookie: 0.5.0              etag: ~1.8.1               methods: ~1.1.2            qs: 6.11.0
body-parser: 1.20.1        debug: 2.6.9               finalhandler: 1.2.0        on-finished: 2.4.1         range-parser: ~1.2.1
content-disposition: 0.5.4 depd: 2.0.0                fresh: 0.5.2               parseurl: ~1.3.3           safe-buffer: 5.2.1
content-type: ~1.0.4       encodeurl: ~1.0.2          http-errors: 2.0.0         path-to-regexp: 0.1.7
(...and 7 more.)

maintainers:
- mikeal <mikeal.rogers@gmail.com>
- dougwilson <doug@somethingdoug.com>
- jasnell <jasnell@gmail.com>

dist-tags:
latest: 4.18.2       next: 5.0.0-beta.1

published 3 months ago by dougwilson <doug@somethingdoug.com>
adarshks@barbarik ~
>>
```

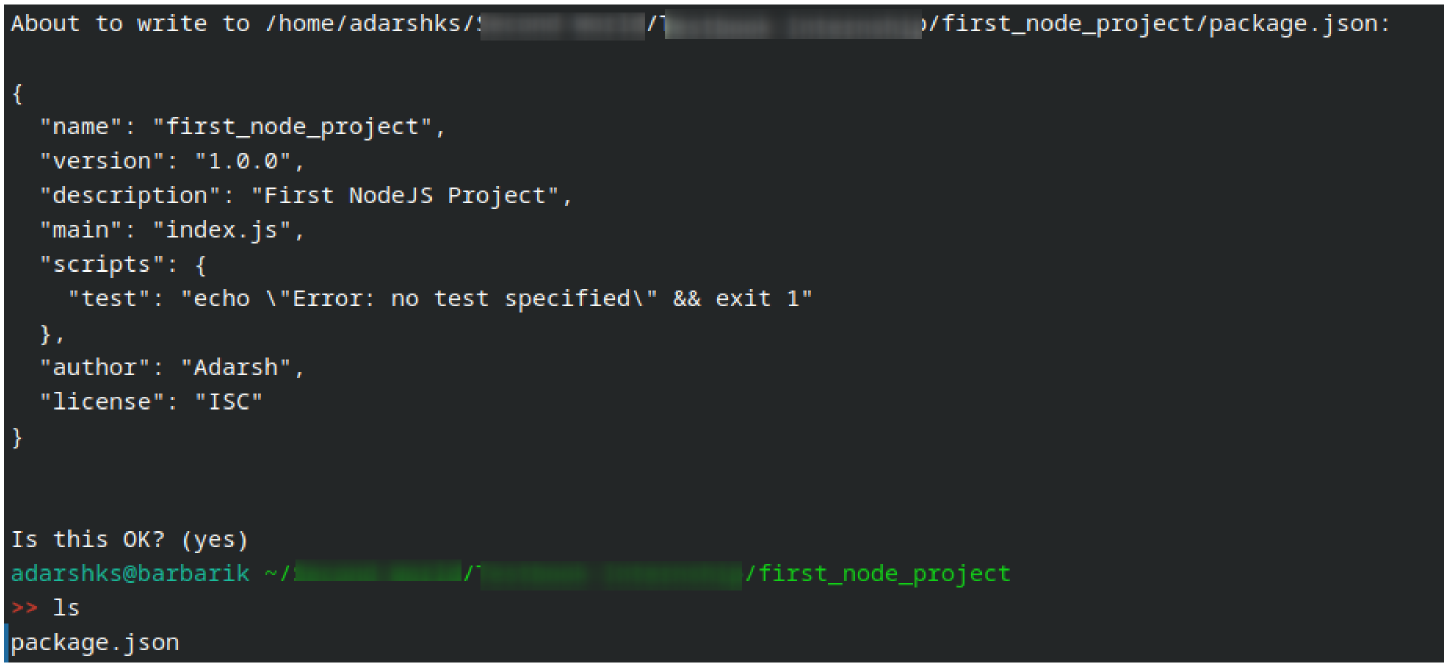
skill academy
by testbook

Creating a new NodeJS Project

1. Create a Project Folder: Create a new folder for your project, and navigate to it in your terminal or command prompt.
2. Initialize npm: In the project folder inside `VSCode` or in `terminal / command Prompt / Powershell`, run the following command to initialize npm: `npm init` and fill in the required information, such as the project name and version. For most of the questions you will be good by pressing the enter key. You will see that a `package.json` file is created at the folder, you can change the inputs that you provided earlier by editing the `package.json` file. `package.json` file is a manifest file which stores important information about the project.



```
adarshks@barbarik ~/[redacted]/[redacted]/first_node_project
>> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (first_node_project)
version: (1.0.0)
description: First NodeJS Project
entry point: (index.js)
test command:
git repository:
keywords:
author: Adarsh
license: (ISC)
```



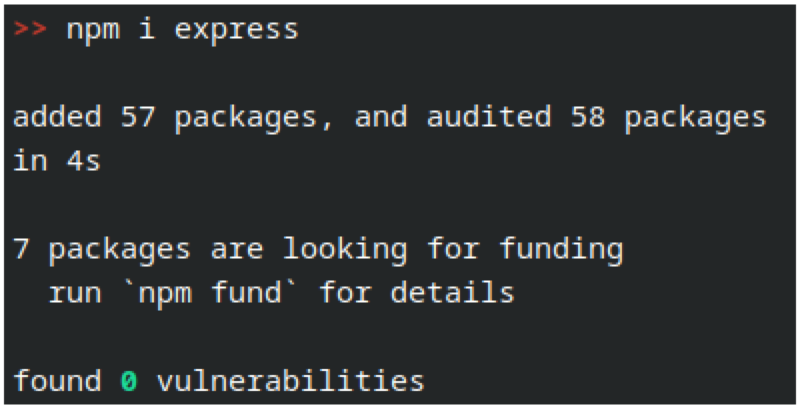
```
About to write to /home/adarshks/[redacted]/[redacted]/first_node_project/package.json:

{
  "name": "first_node_project",
  "version": "1.0.0",
  "description": "First NodeJS Project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Adarsh",
  "license": "ISC"
}


Is this OK? (yes)
adarshks@barbarik ~/[redacted]/[redacted]/first_node_project
>> ls
package.json
```

3. Create a Main File: Create a main file for your project, such as `index.js` (in this case as we have declared it to be the starting point in `package.json` file)file, and write the code for your application.

4. Install Dependencies: If your project requires any dependencies, install them using the npm install command, for example: `npm install express`

There are two types of dependencies :

1. Local Dependency : used in a particular project only
   use command : `npm i <packageName>` when you need to install local dependency, you will see that a node_module folder is created in the folder this folder contains the dependencies installed.

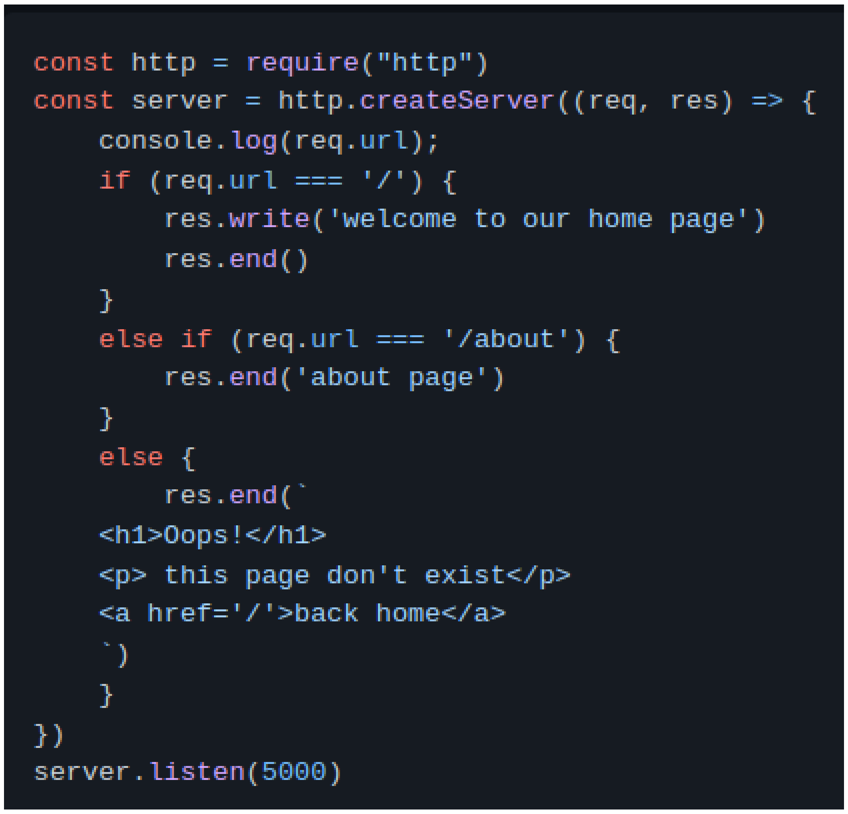


```
>> npm i express

added 57 packages, and audited 58 packages
in 4s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

2. Global Dependency : can be used anywhere
   use command : `npm i -g <packageName>` for installing a package as global dependency
   Note that `i` is short for `install`, you can also type `install` in the command
3. We can also use the `npx` tool for installing some packages only once, the packages installed using `npx` are temporarily available in the system, ex: `npx create-react-app my-app` will create a basic ReactJS Project.

Creating basic NodeJS server using `http` module

1. Create `index.js` file in the npm initiated project folder.
2. Import the http module in the file like : `const http = require("http")`
3. Create a server by writing the following code :



```
const http = require("http")
const server = http.createServer((req, res) => {
    console.log(req.url);
    if (req.url === '/') {
        res.write('welcome to our home page')
        res.end()
    }
    else if (req.url === '/about') {
        res.end('about page')
    }
    else {
        res.end(`
    <h1>Oops!</h1>
    <p> this page don't exist</p>
    <a href='/'>back home</a>
    `)
    }
})
server.listen(5000)
```

In the above code the `http.createServer()` function creates a server which will listen for the requests from the client (ex: browser).

The `request` from the browser will be present in the `req` object which is an argument in the function.

req object contains an `url` key which has the `endpoint` which is requested by the client.

Based on what the endpoint is we are providing the response to the client.

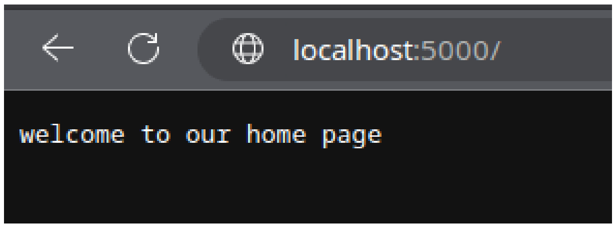
Run the server by using the command `node index.js` in the terminal.

Now open the browser and hit the url

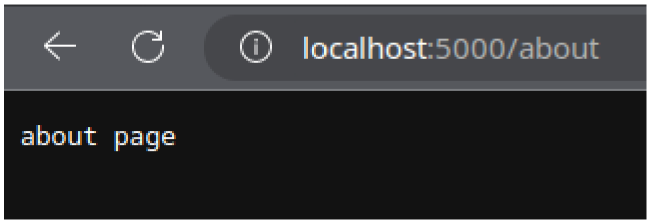
http://localhost:5000/

http://localhost:5000/about

You will see the following responses from the server that we have just created.


localhost:5000/

```
welcome to our home page
```


localhost:5000/about

```
about page
```

`res.write("something here")` will return the response to the client

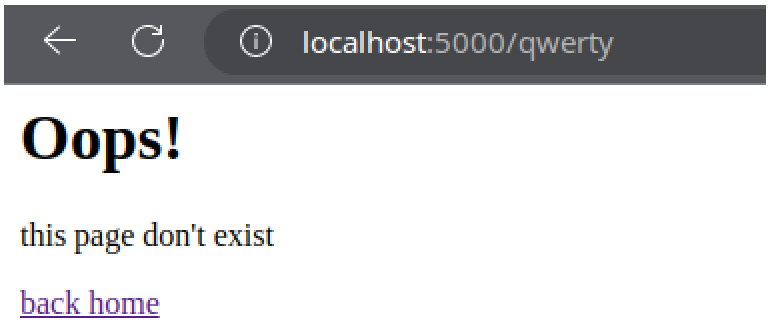
`res.end()` will make the client aware that the response from the server has now ended.

Note that instead of calling the function `res.write()` for sending the response we can send the response in the `res.end()` function as well.

`server.listen(5000)` this function runs the server on the port number provided(here 5000).

The response present in the last else block will run if some random endpoint is a client which we have not defined in the server. Note that here we are sending an `html` element as response so it will be rendered as a `html` page it is also containing a link to the home page.

You can try different variations of making the request and responses.


localhost:5000/qwerty

Oops!

this page don't exist

back home

Basics of fs module - Reading a file

The fs (“file system”) module is a built-in module in Node.js that provides functionality for interacting with the file system on your computer. It allows you to read, write, and manipulate files and directories.
Here we will learn how to read the data from a file using the module, and will learn more on how to create, write and delete in the later part of the course.

The fs module in Node.js provides two methods to read files: fs.readFileSync() and fs.readFile(). Both methods are used to read the contents of a file, but they differ in the way they are executed.

1. fs.readFileSync() is a synchronous method, which means that it blocks the execution of any other code until it has finished reading the file. It takes two arguments: the path to the file, and an optional encoding type.
   Example :
   ```
   const fs = require('fs');
   try {
     const data = fs.readFileSync('example.txt', 'utf8');
     console.log(data);
   } catch (err) {
     console.error(err);
   }
   ```
   In this example, we're using fs.readFileSync() to read the contents of the file example.txt and storing it in a variable called data. We're also specifying the encoding type as utf8, which tells the method to return a string instead of a buffer. Finally, we're logging the contents of the file to the console.

2. fs.readFile() is an asynchronous method, which means that it does not block the execution of any other code while it is reading the file. It takes two arguments: the path to the file, and a callback function that will be called once the file has been read.
   Example :
   ```
   const fs = require('fs');
   fs.readFile('example.txt', 'utf8', (err, data) => {
     if (err) {
       console.error(err);
       return;
     }
     console.log(data);
   });
   ```
   In this example, we're using fs.readFile() to read the contents of the file example.txt and passing a callback function that will be called once the file has been read. The callback function takes two arguments:
   a. **err**, which will contain any errors that occurred while reading the file, &
   b. **data**, which will contain the contents of the file.

   If an error occurs, we're logging it to the console. If not, we're logging the contents of the file to the console.

Both fs.readFileSync() and fs.readFile() are useful methods for reading files in Node.js, but it's important to choose the right one for your situation. If you need to read a file synchronously and want to block the execution of any other code until the file has been read, use fs.readFileSync(). If you need to read a file asynchronously and want to continue executing other code while the file is being read, use fs.readFile().

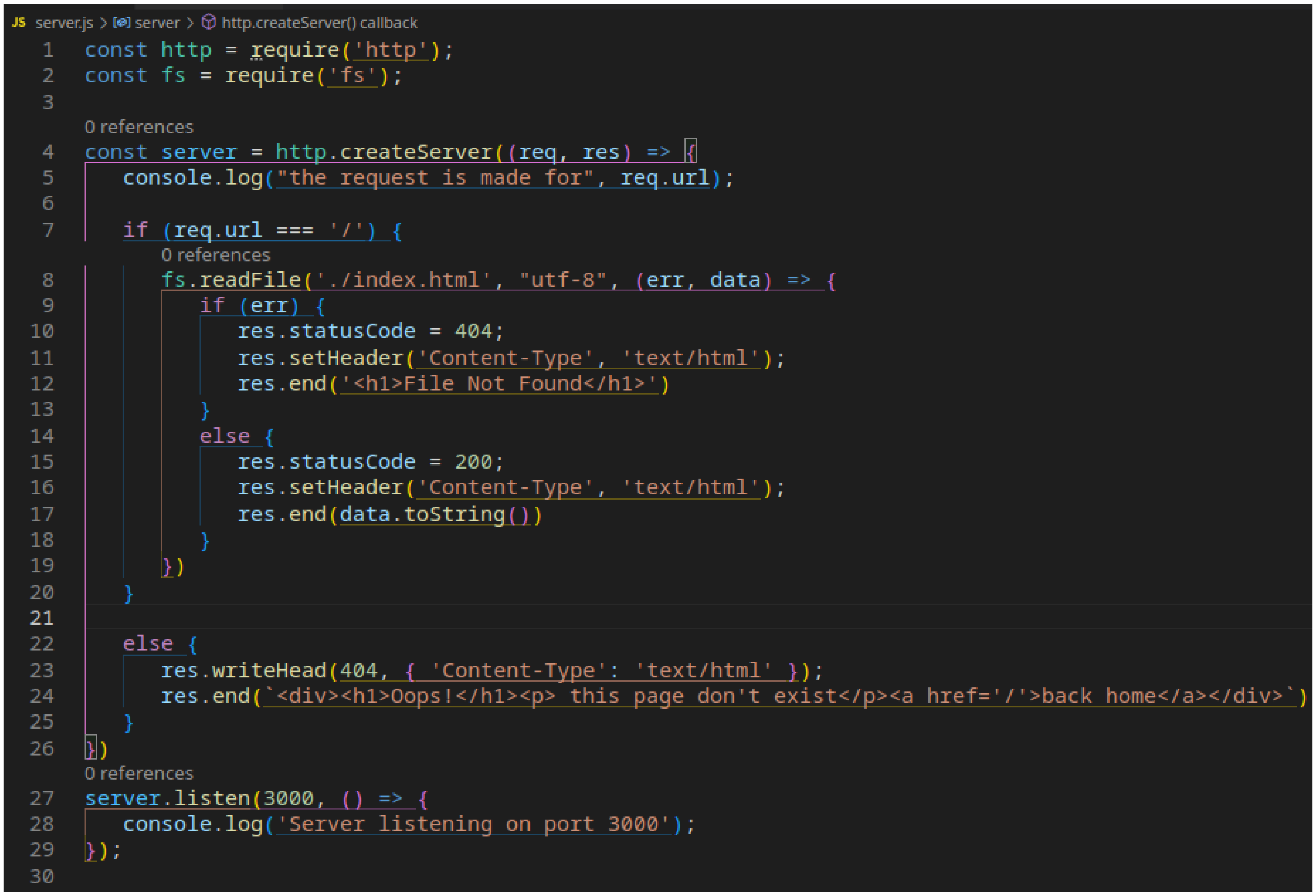
Serving a HTML page using HTTP Module

Serving HTML files using the http module of Node.js is a common task when building web applications

1. First, create a simple HTML file. For this example, let's call it `index.html` .

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>HTTP Module - NodeJS</title>
</head>
<body>
  Welcome to the Home page
</body>
</html>
```

2. Next, create a Node.js file. Let's call it `server.js` . It will contain the basic http module server setup. The code of server.js looks like this and is explained below



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

0 references
const server = http.createServer((req, res) => {
    console.log("the request is made for", req.url);

    if (req.url === '/') {
        0 references
        fs.readFile('./index.html', "utf-8", (err, data) => {
            if (err) {
                res.statusCode = 404;
                res.setHeader('Content-Type', 'text/html');
                res.end('<h1>File Not Found</h1>')
            }
            else {
                res.statusCode = 200;
                res.setHeader('Content-Type', 'text/html');
                res.end(data.toString())
            }
        })
    }

    else {
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end(`<div><h1>Oops!</h1><p> this page don't exist</p><a href='/'>back home</a></div>`)
    }
})
0 references
server.listen(3000, () => {
    console.log('Server listening on port 3000');
});
```

Now we will understand what is being done in server.js file

We have setup the createServer method which takes two arguments req, res and based upon the request url (req.url) of the client we are going to send appropriate response

In the function, we first check if the request is for the root path ("/"). If it is, we use the fs module to read the index.html file asynchronously (so that we do not slow down the server. If there is an error reading the file asynchronously , we return an error, with appropriate message.
We also have to set the status code of the response message so that the client can know if it is a successful response or an error response.
If data is successfully read we also set the content type to text/html and return the HTML data.
We can send the data using the function res.end() which will send the data and will also inform client that the response has ended here.

res.setHeader()

Here note that the `res.setHeader()` function is used to set the value of an HTTP header for the response. HTTP headers are used to provide additional information about the response, such as the content type. The `res.setHeader()` function takes two arguments: the name of the header, and the value to set. For example, to set the Content-Type header to text/html, you would call `res.setHeader('Content-Type', 'text/html');`.

Other values of content type are as follows :

File	Content-Type
HTML	text/html
CSS	text/css
Javascript	application/javascript
PNG Image	image/png
JPEG	image/jpeg

res.writeHead()

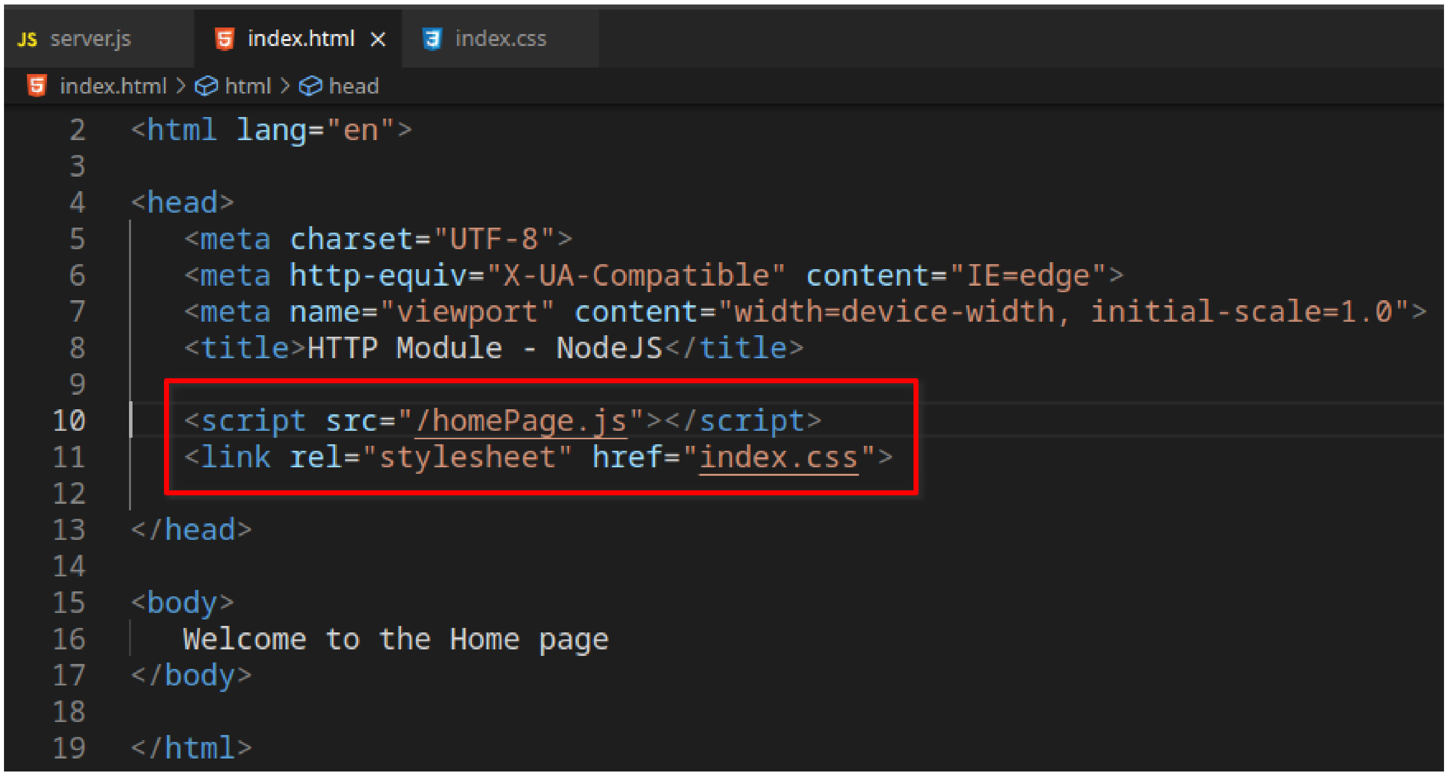
The `res.writeHead()` function takes two arguments: the status code and an object containing the headers. For example, to set the status code to 200 and the Content-Type header to text/html, you would call `res.writeHead(200, {'Content-Type': 'text/html'});`.
So basically writeHead is used to send a response header and status code in a single call.

At this point if you spin the server by command node server.js it will look like this :

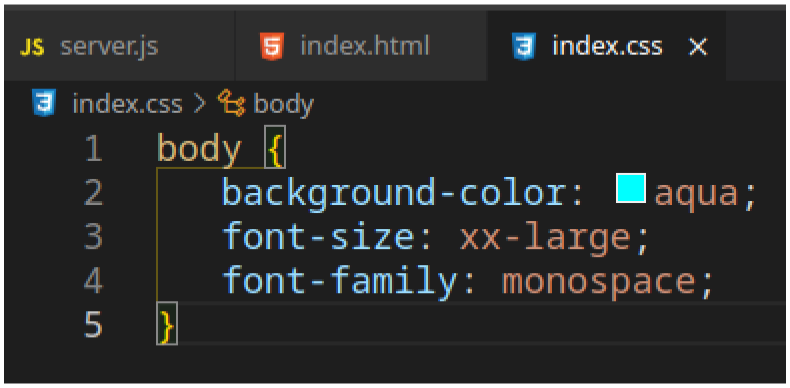
Serving a HTML page with CSS & JS

Now we will look how to serve html files with separate css and js files

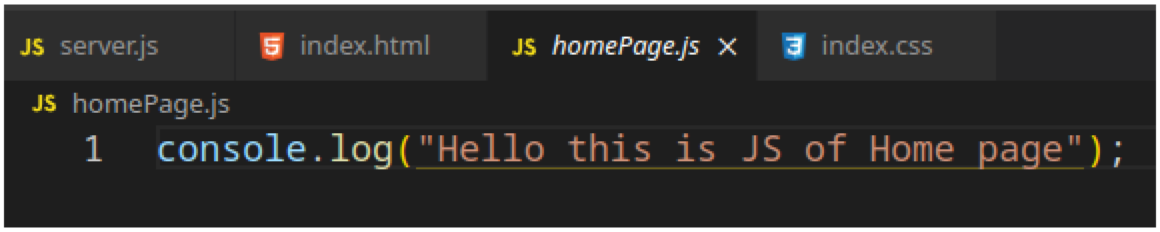
1. Create index.js, index.css and homePage.js files as follows :



```html
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>HTTP Module - NodeJS</title>

    <script src="/homePage.js"></script>
    <link rel="stylesheet" href="index.css">

</head>

<body>
    Welcome to the Home page
</body>

</html>
```

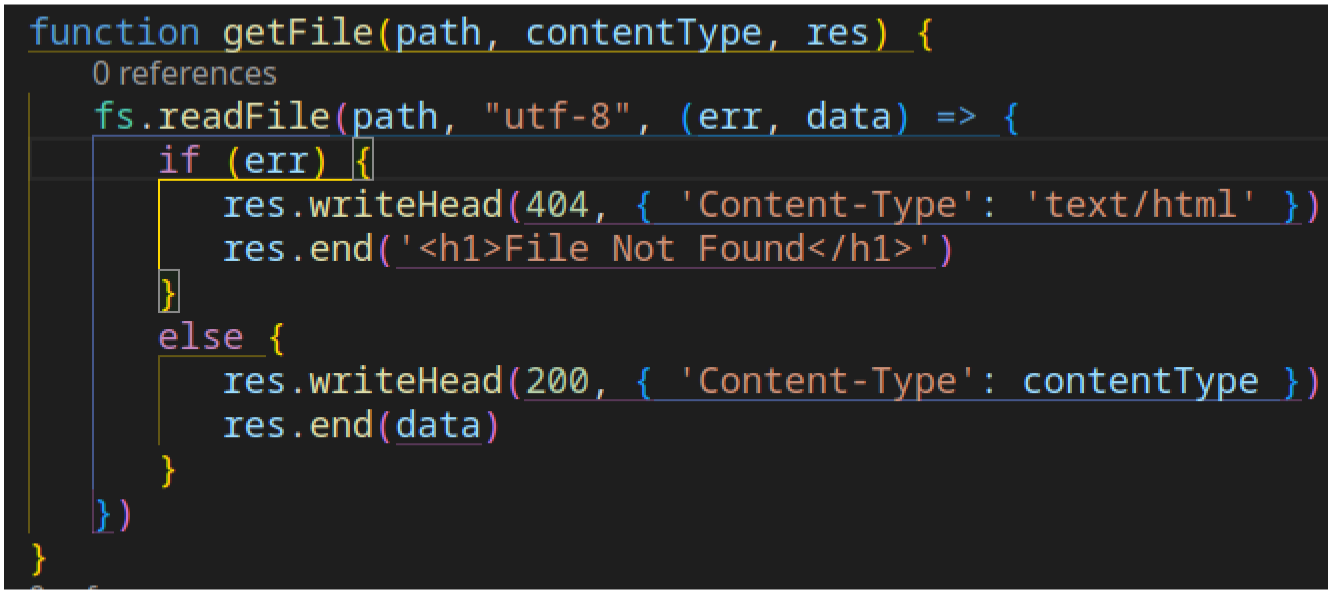


```css
body {
    background-color: aqua;
    font-size: xx-large;
    font-family: monospace;
}
```



```js
console.log("Hello this is JS of Home page");
```

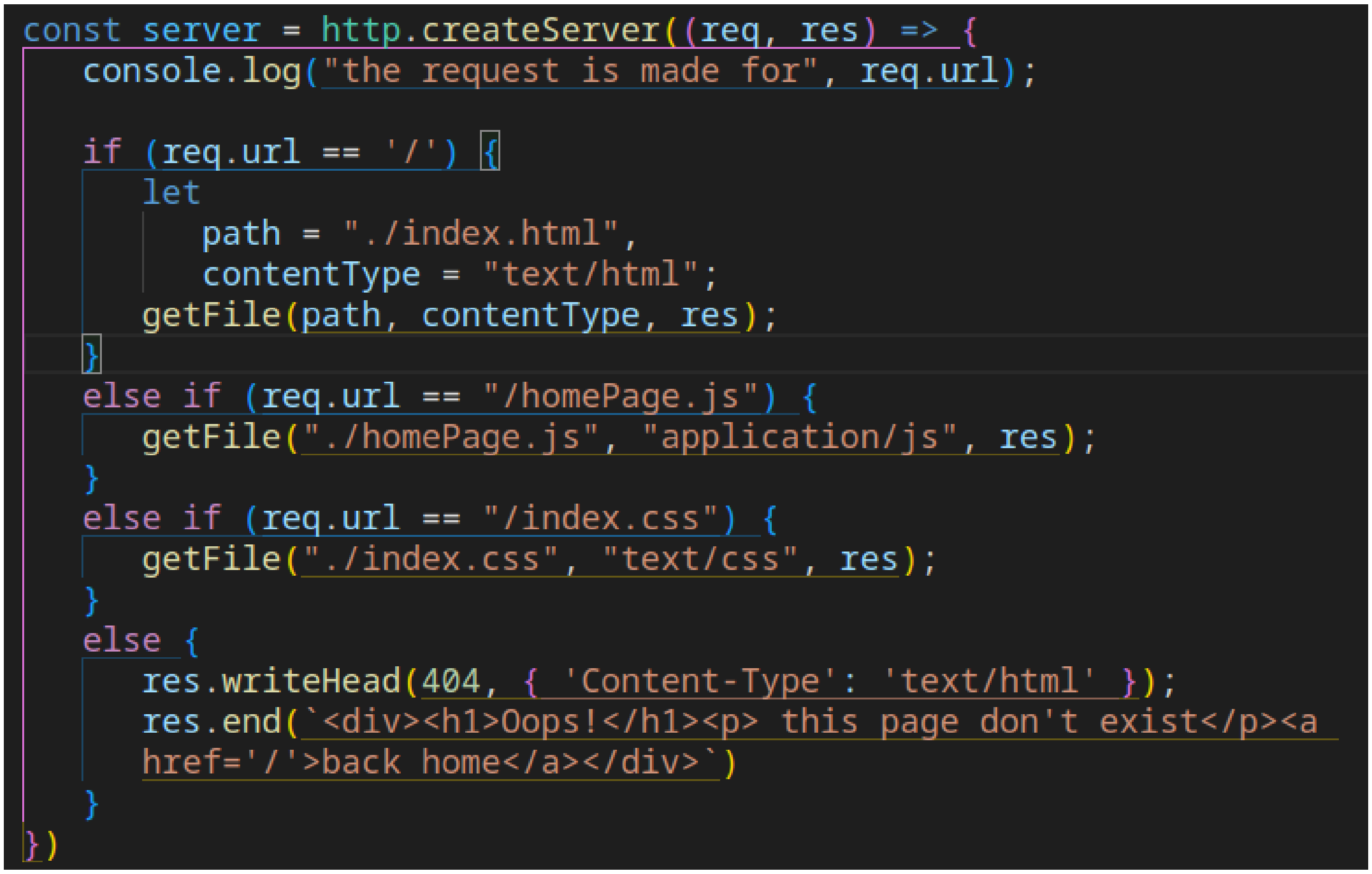
2. In the server.js file create a function getFile like :



```
function getFile(path, contentType, res) {
    0 references
    fs.readFile(path, "utf-8", (err, data) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/html' })
            res.end('<h1>File Not Found</h1>')
        }
        else {
            res.writeHead(200, { 'Content-Type': contentType })
            res.end(data)
        }
    })
}
```

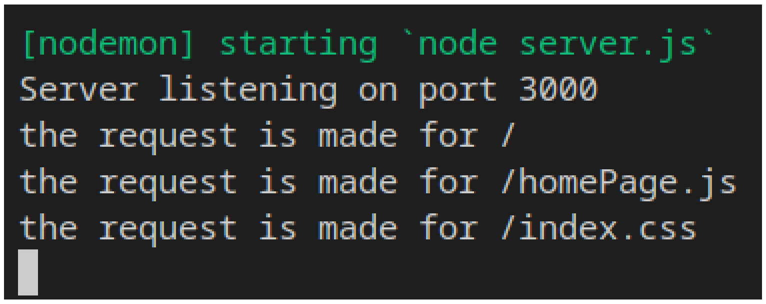
We will use this function everytime we want to read a file and send the contents of the file as a response, we will send the path of the file (like “./index.html”), provide a content type (like “text/html”) and also provide the response object (res) so that the response can be sent to the server from here only.

3. Setup the createServer function like this :



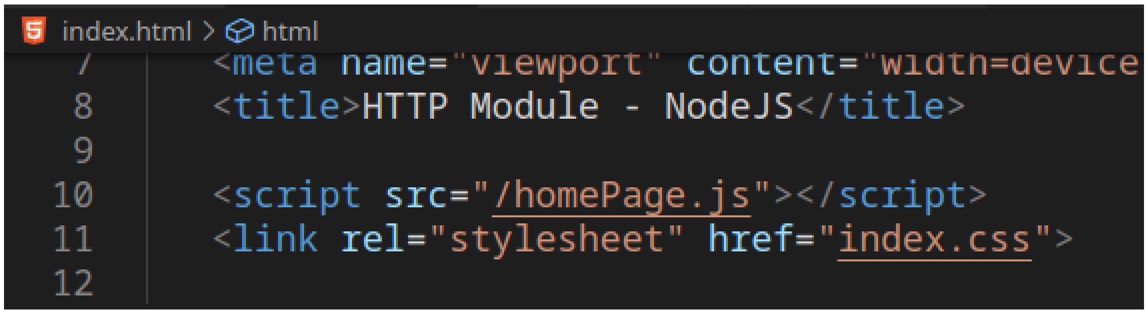
```
const server = http.createServer((req, res) => {
   console.log("the request is made for", req.url);

   if (req.url == '/') {
      let
         path = "./index.html",
         contentType = "text/html";
      getFile(path, contentType, res);
   }
   else if (req.url == "/homePage.js") {
      getFile("./homePage.js", "application/js", res);
   }
   else if (req.url == "/index.css") {
      getFile("./index.css", "text/css", res);
   }
   else {
      res.writeHead(404, { 'Content-Type': 'text/html' });
      res.end(`<div><h1>Oops!</h1><p> this page don't exist</p><a
      href='/'>back home</a></div>`)
   }
})
```

Now if you spin the server and open it on browser the terminal looks like this :



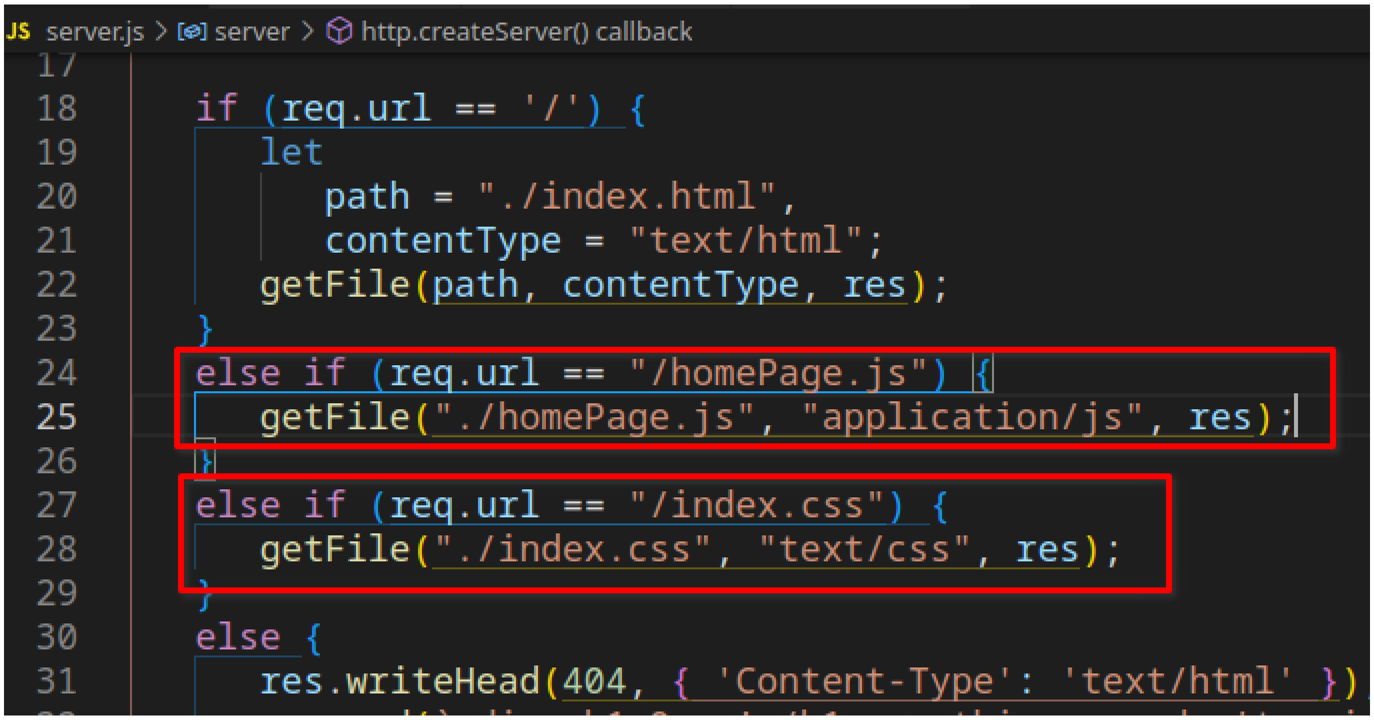
```
[nodemon] starting `node server.js`
Server listening on port 3000
the request is made for /
the request is made for /homePage.js
the request is made for /index.css
```

Here Note we are also getting /homePage.js and /index.css as request urls as we have explicitly mentioned them in the index.html file as :



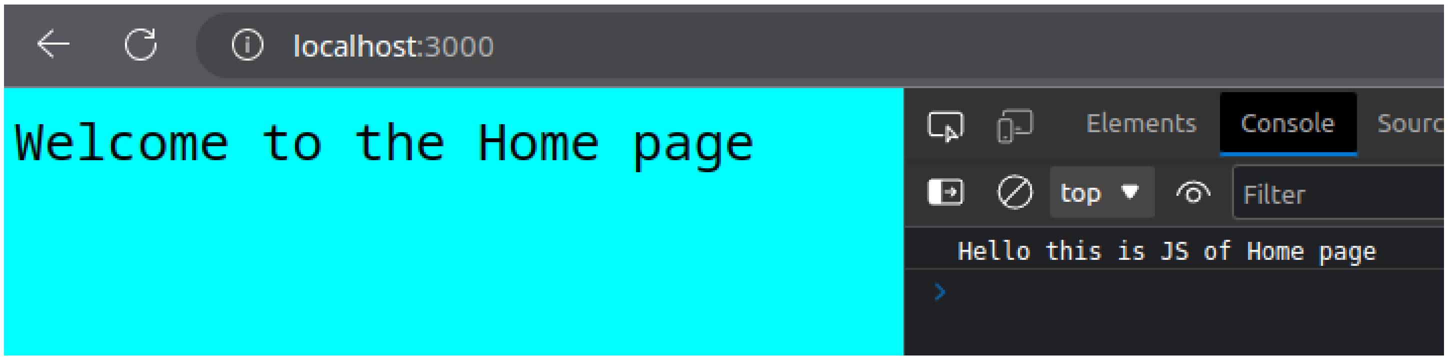
```
index.html > html
    <meta name="viewport" content="width=device
    <title>HTTP Module - NodeJS</title>

    <script src="/homePage.js"></script>
    <link rel="stylesheet" href="index.css">

```

So in order provide appropriate response to these requests we have to setup the required routes in the createServer function :



```
server.js > server > http.createServer() callback

    if (req.url == '/') {
        let
            path = "./index.html",
            contentType = "text/html";
        getFile(path, contentType, res);
    }
    else if (req.url == "/homePage.js") {
        getFile("./homePage.js", "application/js", res);
    }
    else if (req.url == "/index.css") {
        getFile("./index.css", "text/css", res);
    }
    else {
        res.writeHead(404, { 'Content-Type': 'text/html' })
```

Now if you look at the browser it looks like :



```
localhost:3000

Welcome to the Home page

Hello this is JS of Home page
```

As we can see that the css is applied and js is executed along with the html being rendered.