UNIT-3

Node.js

3.1.NODE.JS INTRODUCTION:

Node.js is an open-source server side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.

Node.js can be used to build different types of applications such as command line application, web application, real-time chat application, REST API server etc. However, it is mainly used to build network programs like web servers, similar to PHP, Java, or ASP.NET.

Advantages of Node.js

1. Node.js is an open-source framework under MIT license. (MIT license is a free software license originating at the Massachusetts Institute of Technology (MIT).)
2. Uses JavaScript to build entire server side application.
3. Lightweight framework that includes bare minimum modules. Other modules can be included as per the need of an application.
4. Asynchronous by default. So it performs faster than other frameworks.
5. Cross-platform framework that runs on Windows, MAC or Linux

3.2.Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

3.3.Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

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

var server = http.createServer(function(req, res){

  //write code here

});

server.listen(5000);
```

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

Log.js

```
var log = {
          info: function (info) {
             console.log('Info: ' + info);
          },
          warning:function (warning) {
             console.log('Warning: ' + warning);
          },
          error:function (error) {
             console.log('Error: ' + error);
          }
    };

module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to **module.exports**. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Now, let's see how to use the above logging module in our application.

Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js

```
 var myLogModule = require('./Log.js');


myLogModule.info('Node.js started');
```

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Run the above example using command prompt (in Windows) as shown below.

```
C:\> node app.js
```

```
Info: Node.js started
```

Thus, you can create a local module using module.exports and use it in your application.

Export Module in Node.js

Here, you will learn how to expose different types as a module using module.exports.

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

Let's see how to expose different types as a module using module.exports.

Export Literals

As mentioned above, exports is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

```
module.exports = 'Hello world';
```

Now, import this message module and use it as shown below.

app.js

```
var msg = require('./Messages.js');

console.log(msg);
```

Run the above example and see the result, as shown below.

```
C:\> node app.js
```

```
Hello World
```

3.4.NPM - Node Package Manager

Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application. It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.

NPM is included with Node.js installation. After you install Node.js, verify NPM installation by writing the following command in terminal or command prompt.

```
C:\>npm -v
```

If you have an older version of NPM then you can update it to the latest version using the following command.

```
C:/> npm install npm -g
```

To access NPM help, write **npm help** in the command prompt or terminal window.

```
C:\> npm help
```

NPM performs the operation in two modes: global and local. In the global mode, NPM performs operations which affect all the Node.js applications on the computer whereas in the local mode, NPM performs operations for the particular local directory which affects an application in that directory only.

Install Package Locally

Use the following command to install any third party module in your local Node.js project folder.

```
C:\>npm install <package name>
```

For example, the following command will install ExpressJS into MyNodeProj folder.

```
C:\MyNodeProj> npm install express
```

All the modules installed using NPM are installed under **node_modules** folder. The above command will create ExpressJS folder under node_modules folder in the root folder of your project and install Express.js there.

Add Dependency into package.json

Use --save at the end of the install command to add dependency entry into package.json of your application.

For example, the following command will install ExpressJS in your application and also adds dependency entry into the package.json.

```
C:\MyNodeProj> npm install express –save
```

Install Package Globally

NPM can also install packages globally so that all the node.js application on that computer can import and use the installed packages. NPM installs global packages into */<User>/local/lib/node_modules* folder.

Apply -g in the install command to install package globally. For example, the following command will install ExpressJS globally.

```
C:\MyNodeProj> npm install -g express
```

Update Package

To update the package installed locally in your Node.js project, navigate the command prompt or terminal window path to the project folder and write the following update command.

```
C:\MyNodeProj> npm update <package name>
```

The following command will update the existing ExpressJS module to the latest version.

```
C:\MyNodeProj> npm update express
```

Uninstall Packages

Use the following command to remove a local package from your project.

```
C:\>npm uninstall <package name>
```

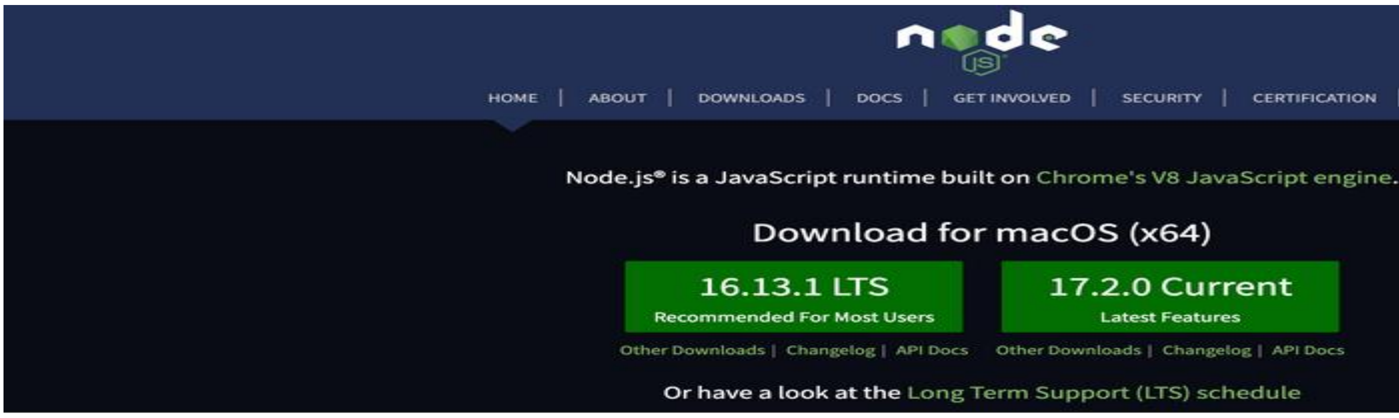
The following command will uninstall ExpressJS from the application.

```
C:\MyNodeProj> npm uninstall express
```

3.5.Build a Secure Server with Node.js

Install Node.js and NPM
If you haven't yet done so, you'll need to install Node and npm on your machine.

1. Go to the Node.js website
2. Click on the recommended download button


HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for macOS (x64)

16.13.1 LTS
Recommended For Most Users

17.2.0 Current
Latest Features

Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs

Or have a look at the Long Term Support (LTS) schedule

NODE HOME PAGE

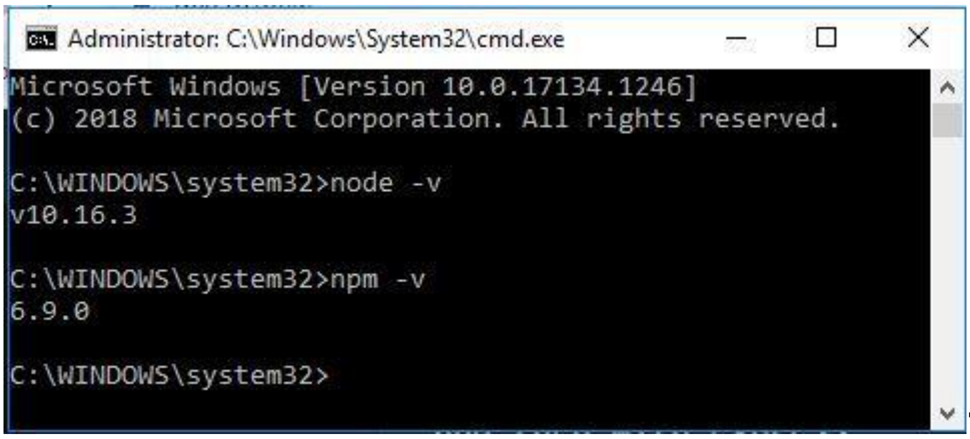
When the download is complete, install Node using the downloaded .exe file (it follows the normal installation process).

Check if the installation was successful

1. Go to your terminal/command prompt *(run as administrator if possible)*
2. Type in each of the following commands and hit Enter

```
node -v
npm -v
```

Your output should be similar to the image below:



```
Microsoft Windows [Version 10.0.17134.1246]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>node -v
v10.16.3

C:\WINDOWS\system32>npm -v
6.9.0

C:\WINDOWS\system32>
```

Terminal showing the versions of node and npm

The version may be different but that's OK.

How to Create a Node Server without Express

Let's start by creating a project directory. Open a terminal and type the following to create a directory and open it:

```
mkdir server-tut

cd server-tut
```

In the terminal, type npm init. Hit the Enter button for all prompts. When completed, you should have a package.json file seated in your project directory.
The package.json file is just a file with all the details of your project. You don't have to open it.
Create a file called index.js.

In the file, require the HTTP module like so:

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

Call the createServer() method on it and assign it to a constant like this:

```
const server = http.createServer();
```

Call the listen() method on the server constant like this:

```
server.listen();
```

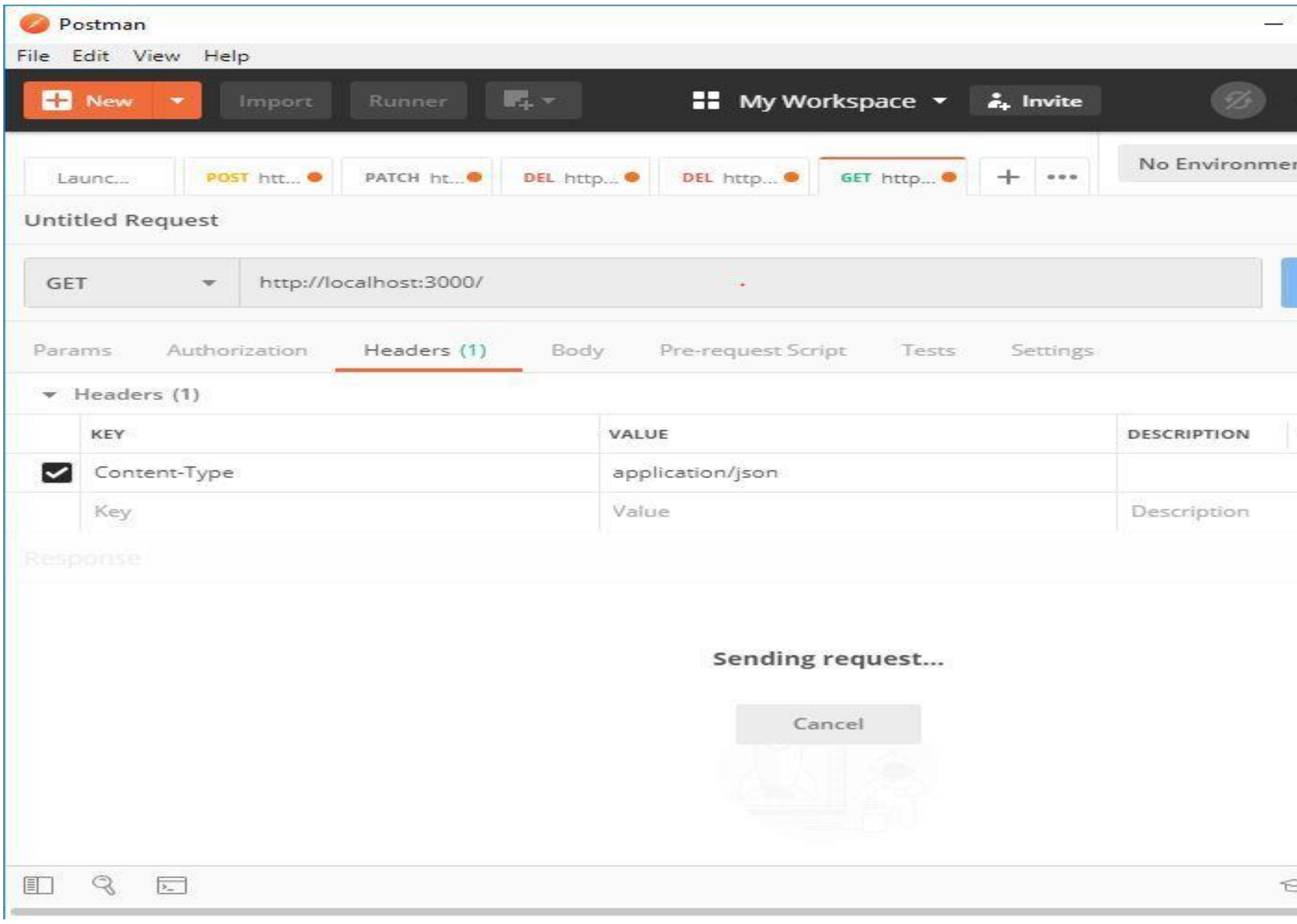
Give it a port to listen to. Now this could be any free port, but we will be using port 3000 which is the conventional port. So we have this:

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


const server = http.createServer();


server.listen(3000);
```

Basically, that is all you need do to create a server.

How to Test the Server

In your terminal (should be in the project directory), type node index.js and hit the Enter button.
Open a new tab in postman or any web browser and in the address bar, type http://localhost:3000/, and hit the Enter button. (I will be using postman because of its extended functionality outside the box.)
You will notice that your browser or postman keeps loading indefinitely like so:



Yay! That is fine. Our Server is up and running.

We need to make the server talk to us.

How to Send Back a Response from the Server

Back in the code, add the following to const server = http.createServer();:

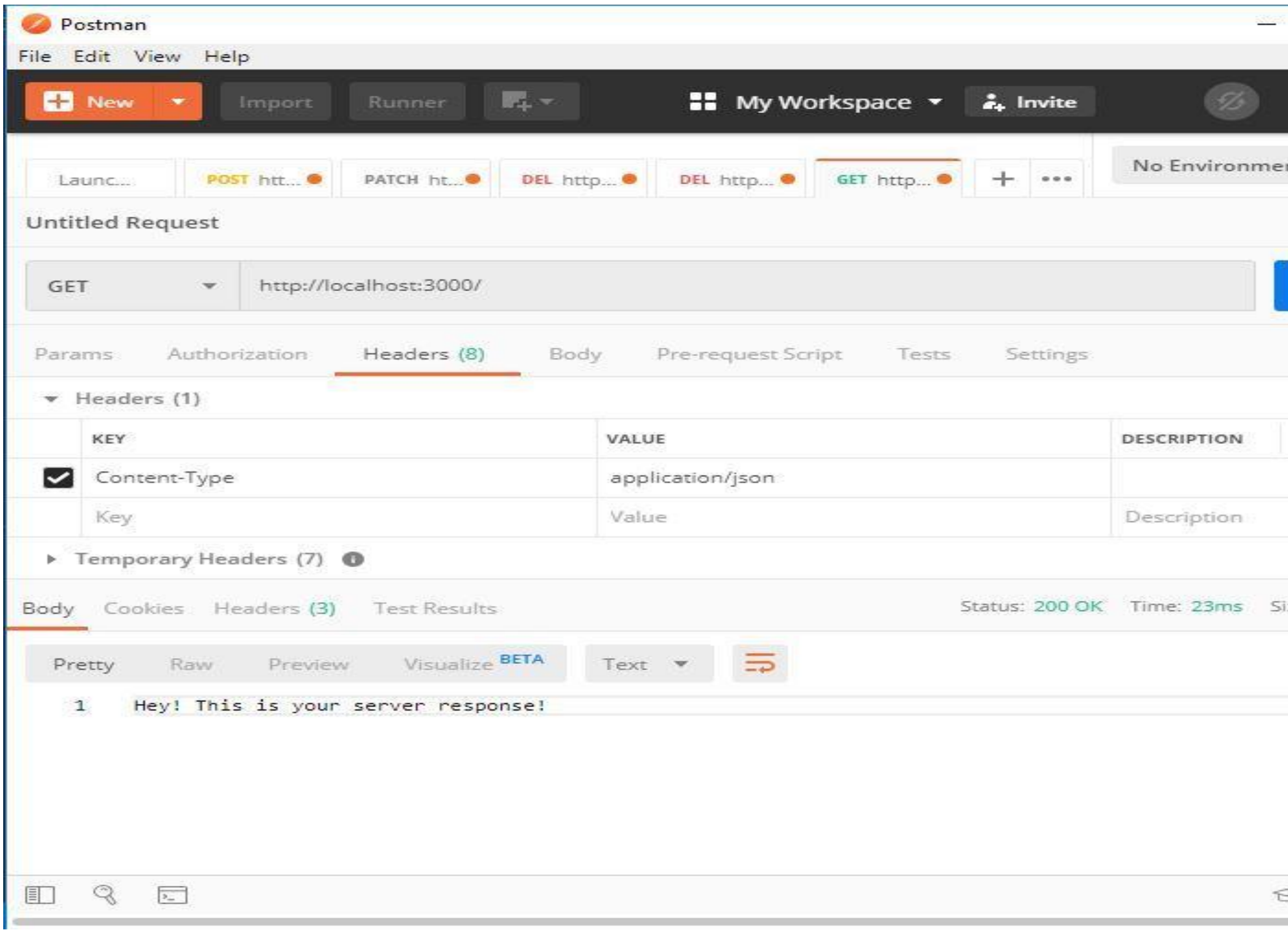
```
 (request, response) => {
    response.end('Hey! This is your server response!');
 }
```

So we now have:

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

  const server = http.createServer((request, response) => {
    response.end('Hey! This is your server response!');
  });

server.listen(3000);
```

In basic terms, the request object tells the server that we want something, the response object tells us what the server has to say about our request, and the end() method terminates the communication with the server response.
Hopefully, that makes sense!

Now, test the server again following the steps we outlined above and your server should be talking to you. This is my output:



GET http://localhost:3000/

Headers (1)

KEY	VALUE	DESCRIPTION
✓ Content-Type	application/json	
Key	Value	Description

Temporary Headers (7)

Status: 200 OK Time: 23ms

```
1   Hey! This is your server response!
```

Feel free to change the string as you wish.

Use Control/Command + C to terminate the server and run node index to start the server again.

How to Create a Node Server With Express

In this section, we want to make our lives easier by using Express and Nodemon (node-mon or no-demon, pronounce it as you wish).
In the terminal, install the following:

```
npm install express --save
npm install nodemon --save-dev
```

Create a new file named app.js or whatever suits you

In the file,

1. Require express like so:

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

2. Assign the express method to a constant like this:

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

3. Export the app constant to make it available for use in other files within the directory like so:

```
module.exports = app;
```

So we have:

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


const app = express();
module.exports = app;
```

In the index.js file, require the app we exported a while ago:

```
const app = require('./app');
```

Next, set the port using the app like so:

```
app.set('port', 3000);
```

And replace the code in the http.createServer() method with just app like this:

```
const server = http.createServer(app);
```

This directs all API management to the app.js file helping with separation of concerns.
So our index.js file now looks like this:

```
const http = require('http');
const app = require('./app');


app.set('port', 3000);
const server = http.createServer(app);


server.listen(3000);
```

Back in our app.js file, since we have directed all API management to it, let's create an endpoint to speak to us like before.
So before the module.exports = app, add the following code:

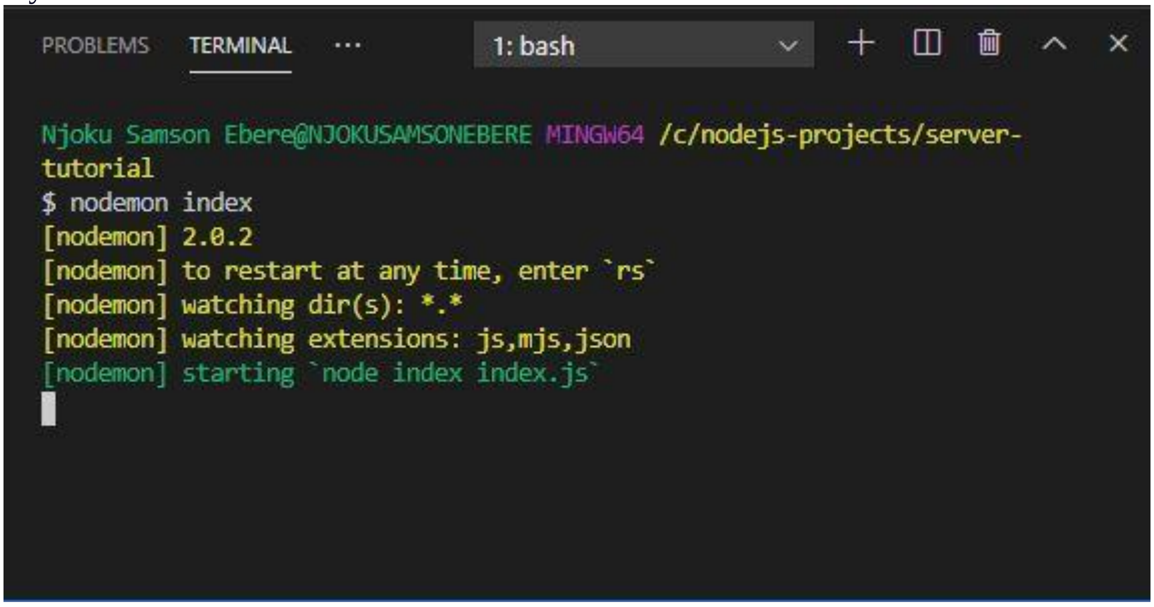
```
app.use((request, response) => {
  response.json({ message: 'Hey! This is your server response!' });
});
```

We now have:

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

app.use((request, response) => {
  response.json({ message: 'Hey! This is your server response!' });
});

module.exports = app;
```

Ahaaa... It's time to test our app.

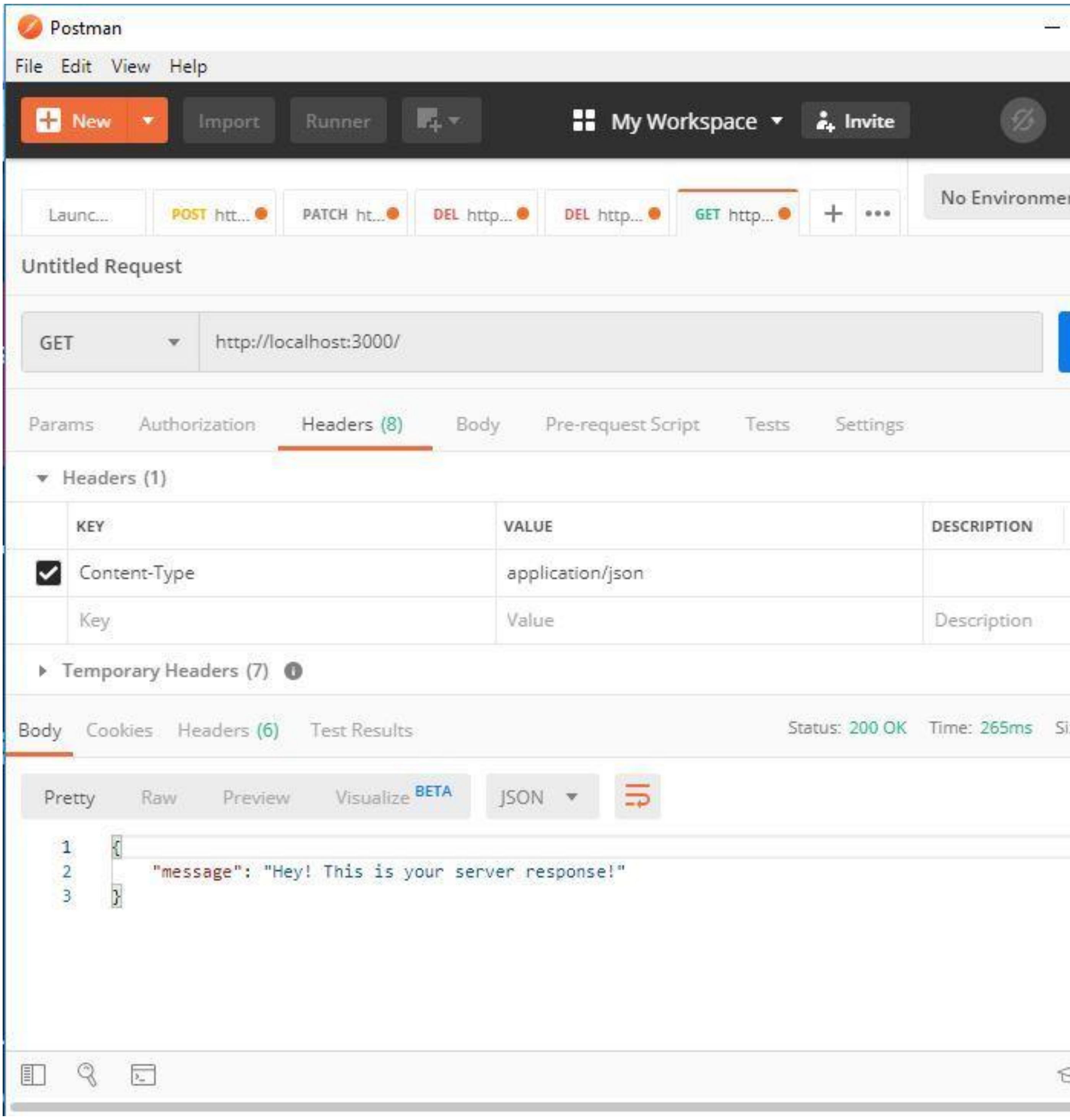
To test our app, we now type nodemon index in our terminal and hit the Enter button. This is my terminal:



```
Njoku Samson Ebere@NJOKUSAMSONEBERE MINGW64 /c/nodejs-projects/server-
tutorial
$ nodemon index
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index index.js`
```

Do you notice that nodemon gives us details of execution in the terminal unlike Node? That's the beauty of nodemon.

You can now go to postman or any browser and in the address bar, type http://localhost:3000/ and hit Enter. See my output:



Untitled Request

GET http://localhost:3000/

Headers (1)

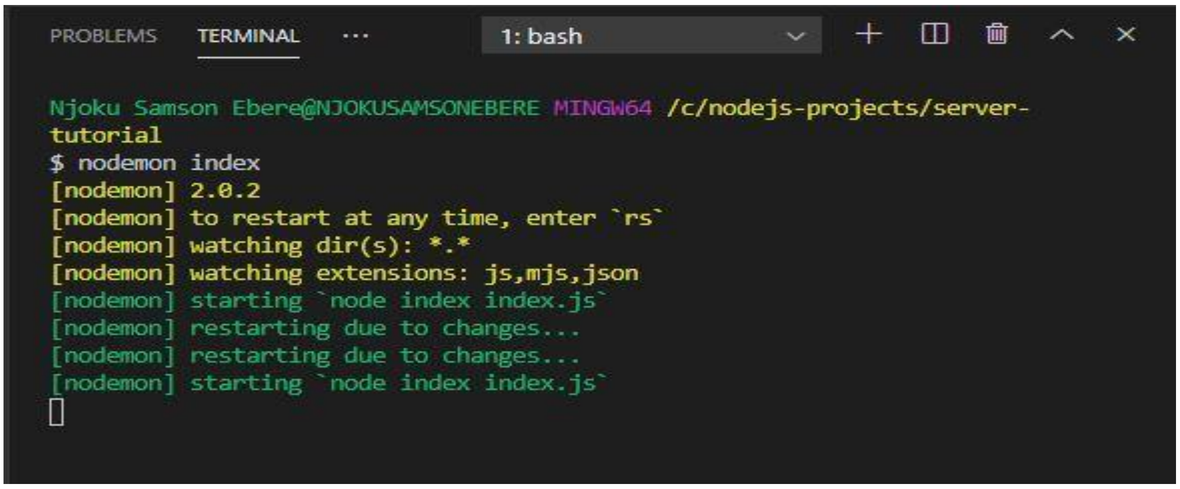
	KEY	VALUE	DESCRIPTION
✓	Content-Type	application/json	
	Key	Value	Description

Temporary Headers (7)

Status: 200 OK Time: 265ms

```
{
    "message": "Hey! This is your server response!"
}
```

Now more reasons to use nodemon. Go to the app.js file and change the message string to any string on your choice, save, and watch the terminal.



```
Njoku Samson Ebere@NJOKUSAMSONEBERE MINGW64 /c/nodejs-projects/server-
tutorial
$ nodemon index
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index index.js`
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node index index.js`
```

Wow... It automatically restarts the server. This was impossible with Node. We had to restart the server ourselves.

3.6.Node.js Debugging:

Debugging is a concept to identify and remove errors from software applications. In this article, we will learn about the technique to debug a Node.js application.

Why not to use console.log()?

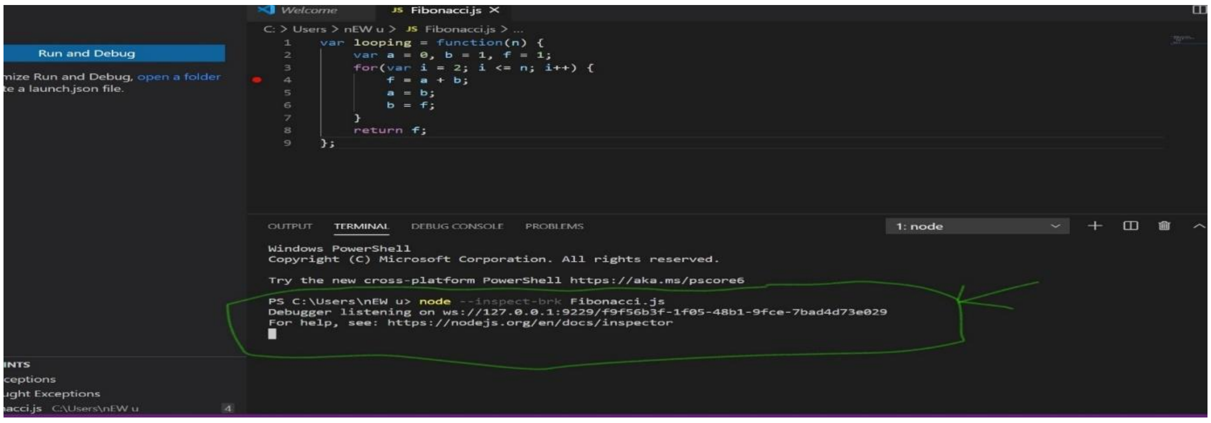
Using **console.log** to debug the code generally dives into an infinite loop of "stopping the app and adding a console.log, and start the app again" operations. Besides slowing down the development of the app, it also makes the writing dirty and creates unnecessary code. Finally, trying to log out variables alongside with the noise of other potential logging operations, may make the process of debugging difficult when attempting to find the values you are debugging.

How to debug?

Mostly we used **console.log()** but as mentioned above, it is not always a good practice. We can use a V8 inspector for it.

Steps for debugging:

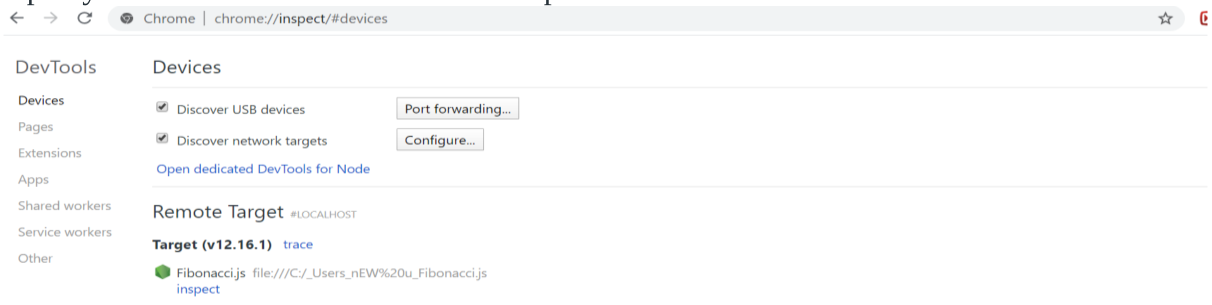
1. Write the following code in the terminal window as shown below:
   node --inspect-brk-filename.js



```
var looping = function(n) {
    var a = 0, b = 1, f = 1;
    for(var i = 2; i <= n; i++) {
        f = a + b;
        a = b;
        b = f;
    }
    return f;
};
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\nEW u> node --inspect-brk Fibonacci.js
Debugger listening on ws://127.0.0.1:9229/f9f56b3f-1f05-48b1-9fce-7bad4d73e029
For help, see: https://nodejs.org/en/docs/inspector
```

2. Open your Chrome browser and write inspect as shown below:



Chrome | chrome://inspect/#devices

DevTools

Devices

Pages

Extensions

Apps

Shared workers

Service workers

Other

Devices

☑ Discover USB devices [Port forwarding...]

☑ Discover network targets [Configure...]

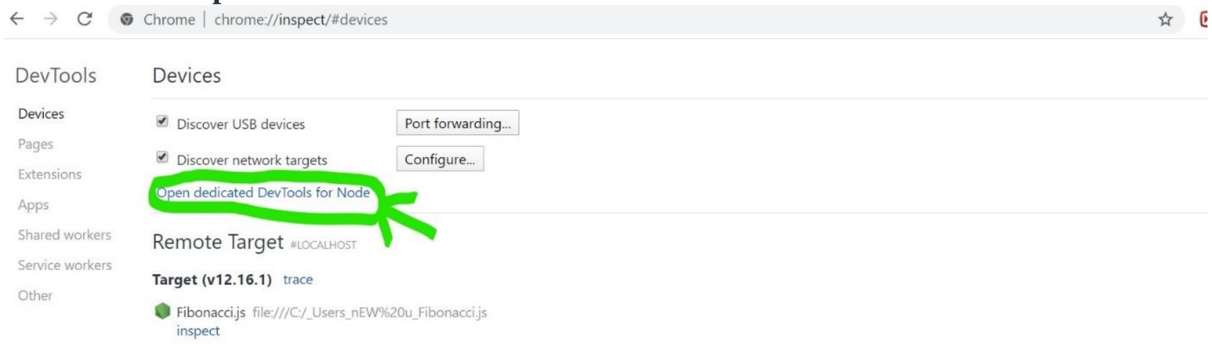
Open dedicated DevTools for Node

Remote Target #LOCALHOST

Target (v12.16.1) trace

Fibonacci.js file:///C:/_Users_nEW%20u_Fibonacci.js
inspect

3. Now click on **Open Dedicated DevTools for Node**.



Chrome | chrome://inspect/#devices

DevTools

Devices

Pages

Extensions

Apps

Shared workers

Service workers

Other

Devices

☑ Discover USB devices [Port forwarding...]

☑ Discover network targets [Configure...]

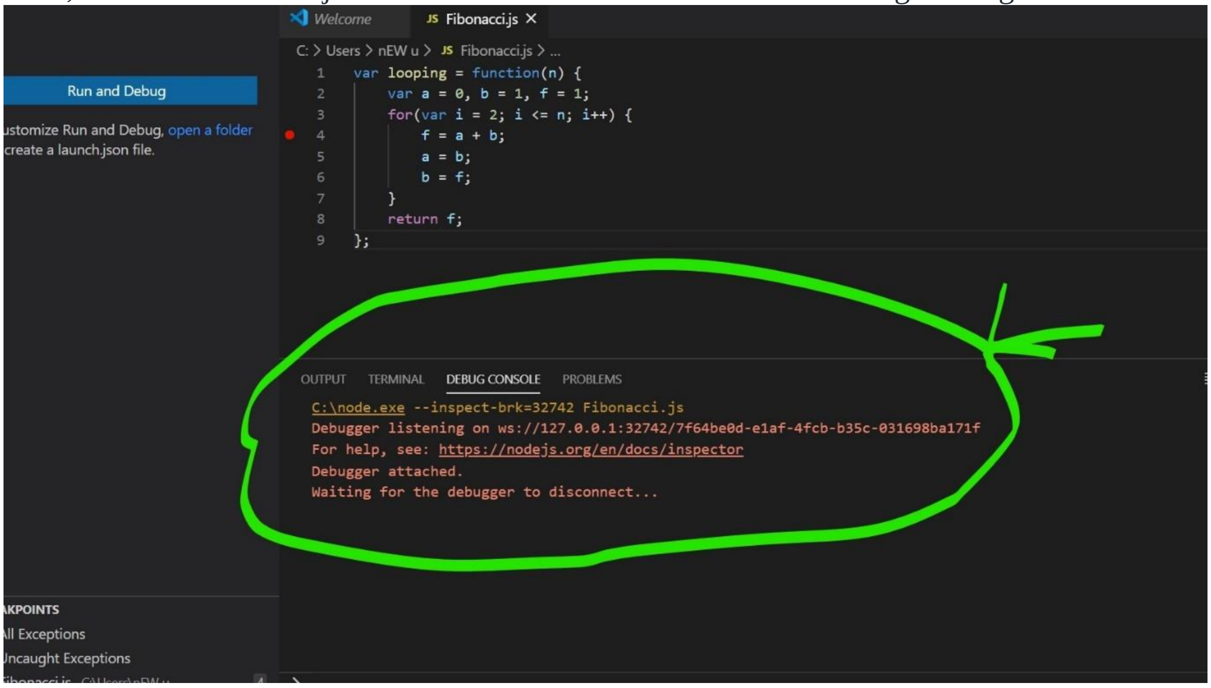
Open dedicated DevTools for Node

Remote Target #LOCALHOST

Target (v12.16.1) trace

Fibonacci.js file:///C:/_Users_nEW%20u_Fibonacci.js
inspect

4. Now, click on the Node.js icon. The terminal will show the following message:



```
var looping = function(n) {
    var a = 0, b = 1, f = 1;
    for(var i = 2; i <= n; i++) {
        f = a + b;
        a = b;
        b = f;
    }
    return f;
};
```

```
C:\node.exe --inspect-brk=32742 Fibonacci.js
Debugger listening on ws://127.0.0.1:32742/7f64be0d-e1af-4fcb-b35c-031698ba171f
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Waiting for the debugger to disconnect...
```

Other tools to help launch a DevTools window:

june07.com/nim

github.com/jaridmargolin/inspect-process

github.com/darcyclarke/rawkit

Additional Debugging APIs:

1. **Debugging an existing Node process:**
2. process._debugProcess(pid);</pre>

3. **GDB-like CLI Debugger:**
4. node inspect filename.js

5. **Drive with DevTools Protocol via WS port:**

```
const dp = require('chrome-remote-interface');

async function test() {
const client = await dp();
const {Profiler, Runtime} = client;

await Profiler.enable();
await Profiler.setSamplingInterval({interval: 500});

await Profiler.start();
await Runtime.evaluate({expression: 'startTest();'});
await sleep(800);

const data = await Profiler.stop();
require('fs').writeFileSync('data.cpuprofile',
            JSON.stringify(data.profile));
};
```

```
test().then((result)=>{
   console.log(result);
})
.catch((error)=>{
   console.log(error);
});
```

6. **DevTools Protocol via require('inspector'):**

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

const fs = require('fs');

const session = new inspector.Session();

 session.connect();

session.post('Profiler.enable');

session.post('Profiler.start');

  setTimeout( function() {

    session.post('Profiler.stop',

             function(err, data) {

        fs.writeFileSync('data.cpuprofile',

             JSON.stringify(data.profile));

    });

}, 8000);
```

Another awesome thing in using Chrome as a debugging tool is that you can debug both your front-end and back-end JavaScript code with the same interface.

3.7.What is inspector in Node.js?

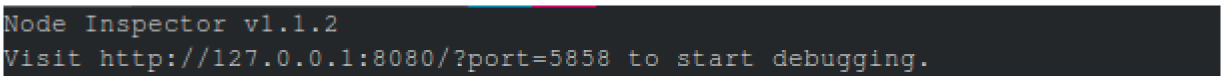
Inspector in node.js is a debugging interface for node.js application that is contained in the app.js file and used blink developer tools. It works almost similar to chrome developer tools. It can support almost all the feature that a debugger generally have such as navigating to source files, setting breakpoints, CPU and heap profiling, network client requests inspection, console output inspections, and many other features.

How to install?

It can be installed by running the following command in the command line after installing npm (node package manager).

$ npm install -g node-inspector

Here in the command, -g flag corresponds to global installation of the inspector. After installing if you run command node-inspector, we get an output like this:



```
Node Inspector v1.1.2
Visit http://127.0.0.1:8080/?port=5858 to start debugging.
```

After successful installation

In the above figure, it displays an URL for debugging purpose. So, when we point our browser to *http://127.0.0.1:8080/?port=5858*, we get a GUI for debugging. Sometimes, port 8080 may not be available on the computer then we will get an error. We can change the port (in this case port 5555) on which node-inspector is running by using the following command:

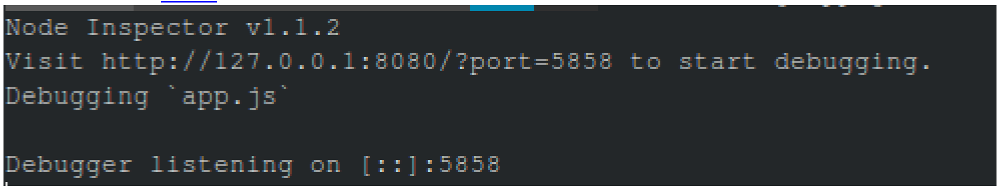
$ node-inspector --web-port=5555

How to start using it?

It can be started using following command in command line:

$ node-debug app.js

where app.js is the name of the main JavaScript application file. Available configuration options can be seen here.



```
Node Inspector v1.1.2
Visit http://127.0.0.1:8080/?port=5858 to start debugging.
Debugging `app.js`

Debugger listening on [::]:5858
```

While debugging app.js file

The node-debug command will load Node Inspector in the default browser.

Note: Node Inspector works in Chrome and Opera only.

Advanced Use: While running node-debug is an easy way to start your debugging session, sometimes we need to tweak the default setup. Then we need to follow three steps given below

1. **Start the node-inspector server:** This can be done by running command:

   $ node-inspector

   The server can be left running in the background, it is possible to debug multiple processes using the same server instance.

2. **Enable debug mode in the node process:** You can either start Node with a debug flag like:

   $ node --debug your/node/program.js

   or, to pause your script on the first line:

   $ node --debug-brk your/short/node/script.js

Or one can enable debugging on a node that is already running by sending it a signal:

1. Get the PID of the node process using your favorite method. pgrep or ps -ef are good.

   ```
   $ pgrep -l node

   2345 node your/node/server.js
   ```

2. Send it the USR1 signal

   ```
   $ kill -s USR1 2345
   ```

3. **Load the debugger UI:** Open *http://127.0.0.1:8080/?port=5858* or the produced URL in the Chrome browser.

Node.js is a widely used javascript library based on Chrome's V8 JavaScript engine for developing server-side applications in web development.

3.8.TESTING IN NODE.JS

Unit Testing is a software testing method where individual units/components are tested in isolation. A unit can be described as the smallest testable part of code in an application. Unit testing is generally carried out by developers during the development phase of an application.
In Node.js there are many frameworks available for running unit tests. Some of them are:

- Mocha
- Jest
- Jasmine
- AVA

Unit testing for a node application using these frameworks:

1. **Mocha:** Mocha is an old and widely used testing framework for node applications. It supports asynchronous operations like callbacks, promises, and async/await. It is a highly extensible and customizable framework that supports different assertions and mocking libraries.
   To install it, open command prompt and type the following command:

   ```
   # Installs globally

   npm install mocha -g

   # installs in the current directory

   npm install mocha --save-dev
   ```

   How to use Mocha?
   In order to use this framework in your application:
   1. Open the root folder of your project and create a new folder called **test** in it.
   2. Inside the test folder, create a new file called test.js which will contain all the code related to testing.
   3. open package.json and add the following line in the scripts block.
   4. "scripts": {
   5. "test": "mocha --recursive --exit"

      }

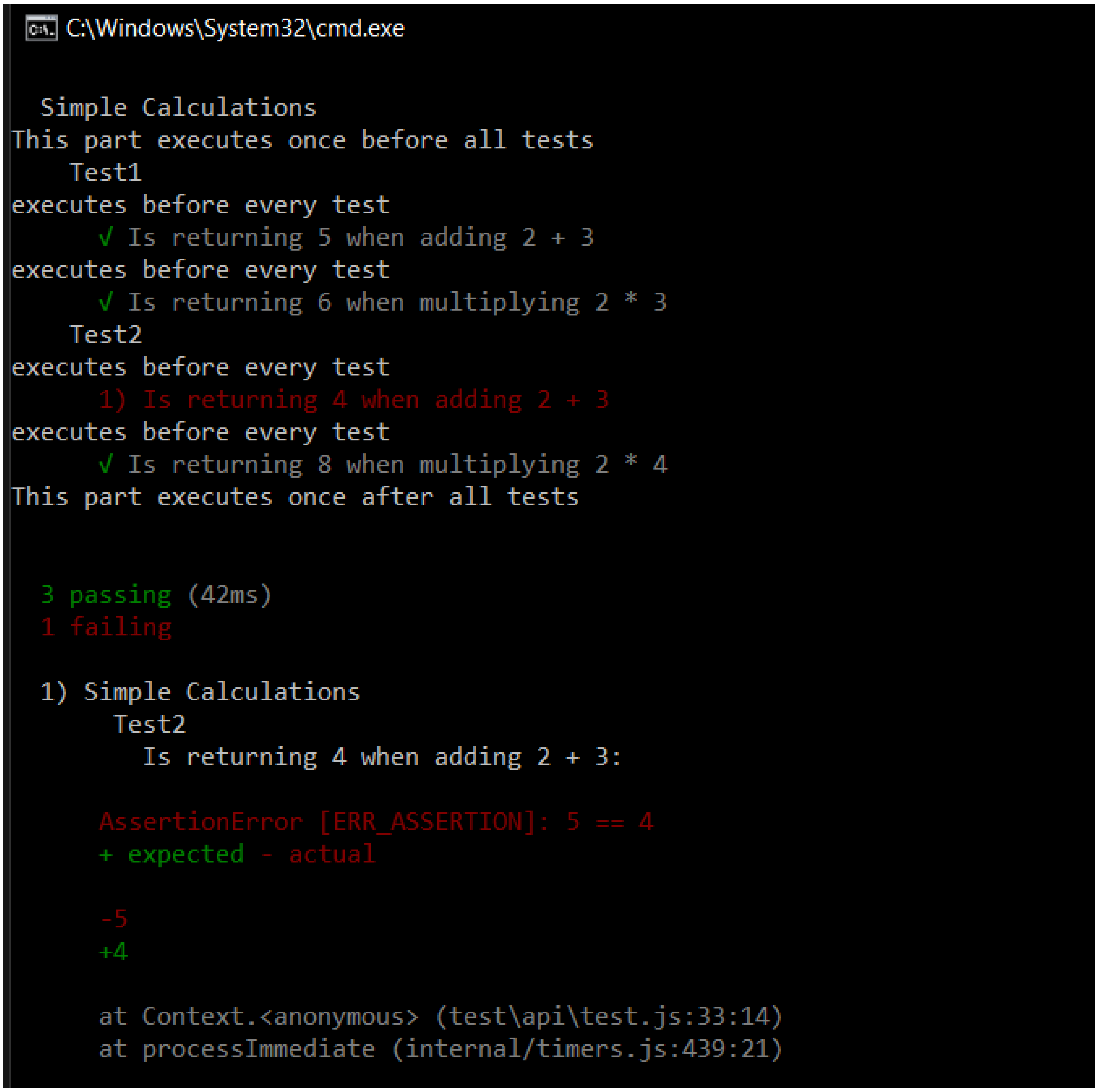
Example:

```
// Requiring module
const assert = require('assert');

// We can group similar tests inside a describe block
describe("Simple Calculations", () => {
 before(() => {
  console.log( "This part executes once before all tests" );
 });

 after(() => {
  console.log( "This part executes once after all tests" );
 });

 // We can add nested blocks for different tests
 describe( "Test1", () => {
  beforeEach(() => {
   console.log( "executes before every test" );
  });

  it("Is returning 5 when adding 2 + 3", () => {
   assert.equal(2 + 3, 5);
  });

  it("Is returning 6 when multiplying 2 * 3", () => {
   assert.equal(2*3, 6);
  });
 });

 describe("Test2", () => {
  beforeEach(() => {
   console.log( "executes before every test" );
  });

  it("Is returning 4 when adding 2 + 3", () => {
   assert.equal(2 + 3, 4);
  });

  it("Is returning 8 when multiplying 2 * 4", () => {
   assert.equal(2*4, 8);
  });
 });
});
```

Copy the above code and paste it in the test.js file that we have created before. To run these tests, open the command prompt in the root directory of the project and type the following command:

```
npm run test
```

Output:


C:\Windows\System32\cmd.exe

```
  Simple Calculations
This part executes once before all tests
    Test1
executes before every test
      √ Is returning 5 when adding 2 + 3
executes before every test
      √ Is returning 6 when multiplying 2 * 3
    Test2
executes before every test
      1) Is returning 4 when adding 2 + 3
executes before every test
      √ Is returning 8 when multiplying 2 * 4
This part executes once after all tests


  3 passing (42ms)
  1 failing

  1) Simple Calculations
       Test2
         Is returning 4 when adding 2 + 3:

      AssertionError [ERR_ASSERTION]: 5 == 4
      + expected - actual

      -5
      +4

      at Context.<anonymous> (test\api\test.js:33:14)
      at processImmediate (internal/timers.js:439:21)
```

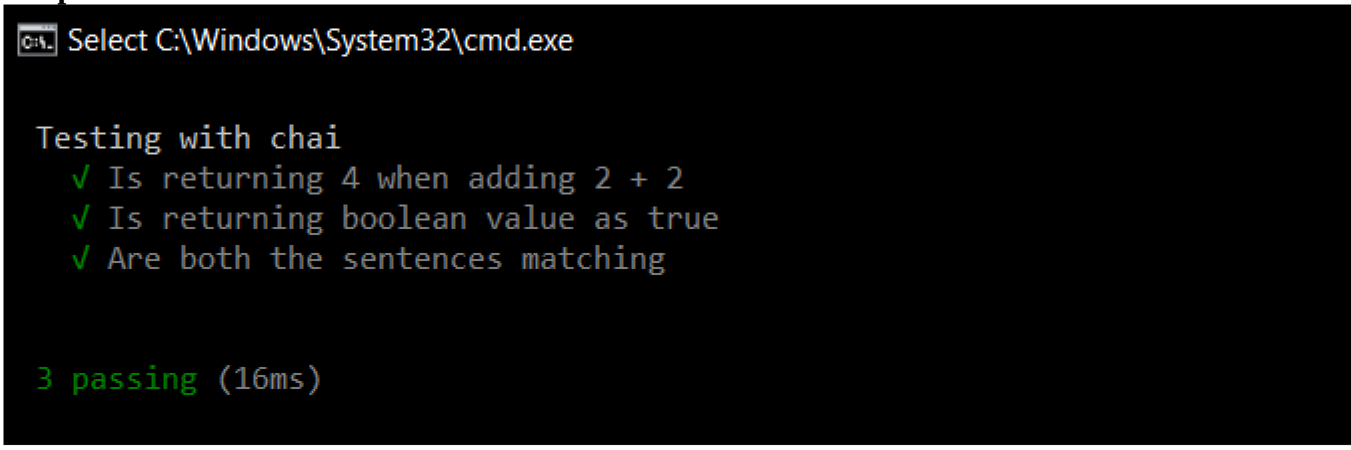
What is Chai?
Chai is an assertion library that is often used alongside Mocha. It can be used as a TTD (Test Driven Development) / BDD (Behavior Driven Development) assertion library for Node.js that can be paired up with any testing framework based on JavaScript. Similar to assert.equal() statement in the above code, we can use Chai to write tests like English sentences.
To install it, open the command prompt in the root directory of the project and type the following command:

```
npm install chai
```

Example:

```
const expect = require('chai').expect;

describe("Testing with chai", () => {
  it("Is returning 4 when adding 2 + 2", () => {
    expect(2 + 2).to.equal(4);
  });

  it("Is returning boolean value as true", () => {
    expect(5 == 5).to.be.true;
  });

  it("Are both the sentences matching", () => {
    expect("This is working").to.equal('This is working');
  });
 });
```

Output:



```
Select C:\Windows\System32\cmd.exe

  Testing with chai
    √ Is returning 4 when adding 2 + 2
    √ Is returning boolean value as true
    √ Are both the sentences matching


  3 passing (16ms)
```

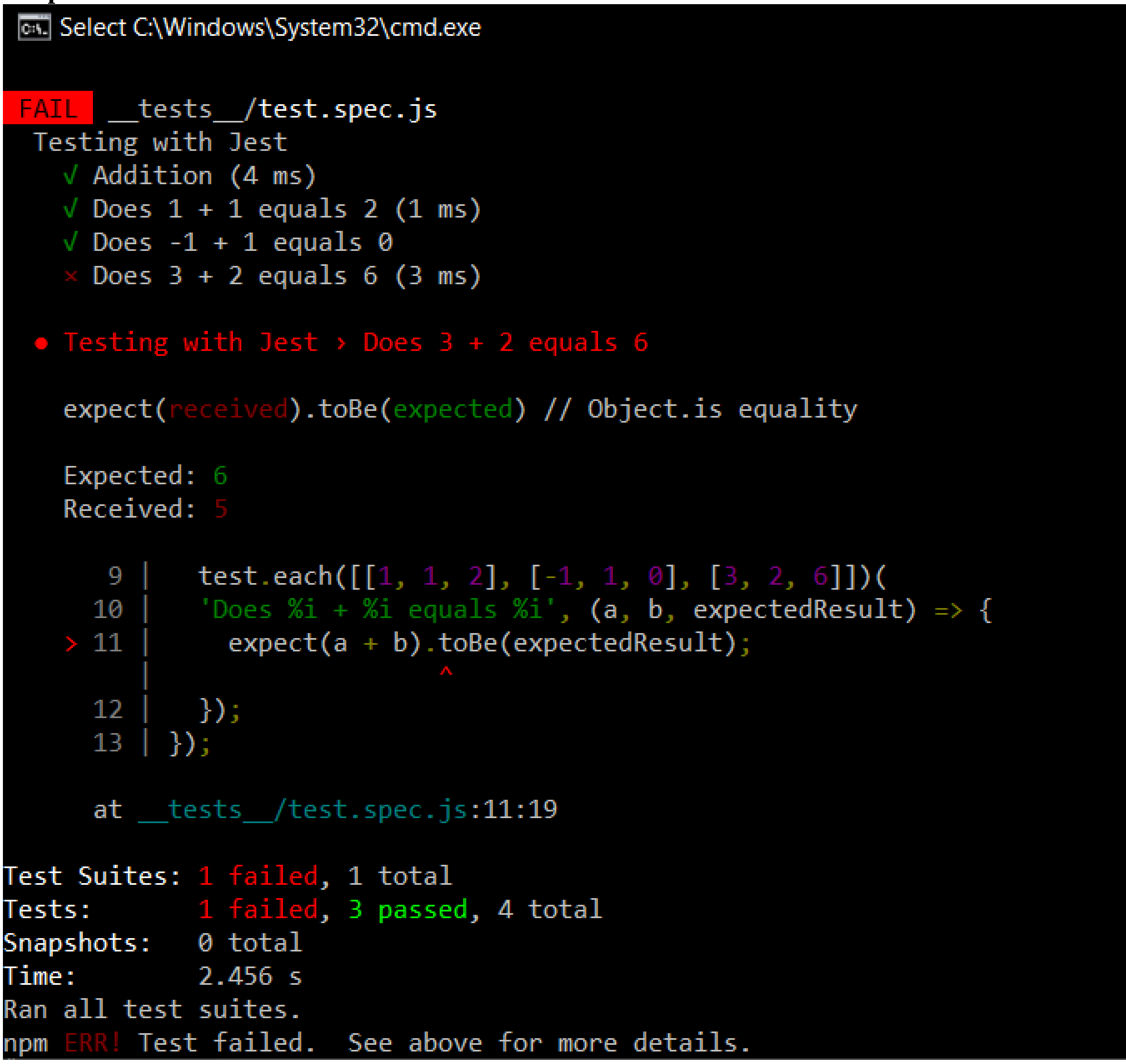
2. **Jest:** Jest is also a popular testing framework that is known for its simplicity. It is developed and maintained regularly by Facebook. One of the key features of jest is it is well documented, and it supports parallel test running i.e. each test will run in their own processes to maximize performance. It also includes several features like test watching, coverage, and snapshots.
You can install it using the following command:

npm install --save-dev jest

Note: By default, Jest expects to find all the test files in a folder called "**__tests__**" in your root folder.
Example:

```
describe("Testing with Jest", () => {
 test("Addition", () => {
   const sum = 2 + 3;
   const expectedResult = 5;
   expect(sum).toEqual(expectedResult);
 });
 // Jest also allows a test to run multiple
 // times using different values
 test.each([[1, 1, 2], [-1, 1, 0], [3, 2, 6]])(
 'Does %i + %i equals %i', (a, b, expectedResult) => {
   expect(a + b).toBe(expectedResult);
 });
});
```

Output:



```
Select C:\Windows\System32\cmd.exe

FAIL  __tests__/test.spec.js
  Testing with Jest
    √ Addition (4 ms)
    √ Does 1 + 1 equals 2 (1 ms)
    √ Does -1 + 1 equals 0
    × Does 3 + 2 equals 6 (3 ms)

  ● Testing with Jest › Does 3 + 2 equals 6

    expect(received).toBe(expected) // Object.is equality

    Expected: 6
    Received: 5

       9 |   test.each([[1, 1, 2], [-1, 1, 0], [3, 2, 6]])(
      10 |   'Does %i + %i equals %i', (a, b, expectedResult) => {
    > 11 |     expect(a + b).toBe(expectedResult);
         |                   ^
      12 |   });
      13 | });

      at __tests__/test.spec.js:11:19

Test Suites: 1 failed, 1 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        2.456 s
Ran all test suites.
npm ERR! Test failed.  See above for more details.
```

3. **Jasmine:** Jasmine is also a powerful testing framework and has been around since 2010. It is a Behaviour Driven Development(BDD) framework for testing JavaScript code. It is known for its compatibility and flexibility with other testing frameworks like Sinon and Chai. Here test files must have a specific suffix (*spec.js).
You can install it using the following command:

npm install jasmine-node

Example:

```
describe("Test", function() {
  it("Addition", function() {
    var sum = 2 + 3;
    expect(sum).toEqual(5);
  });
});
```

4. **AVA:** AVA is a relatively new minimalistic framework that allows you to run your JavaScript tests concurrently. Like the Jest framework, it also supports snapshots and parallel processing which makes it relatively fast compared to other frameworks. The key features include having no implicit globals and built-in support for asynchronous functions.
You can install it using the following command:

npm init ava

Example:

```
import test from 'ava';

test('Addition', t => {
  t.is(2 + 3, 5);
});
```