JavaScript vs Nodejs:

| **JavaScript** | **NodeJS** |
| --- | --- |
| **Javascript**is a **programming language**that is **used for writing scripts**on the website. | **NodeJS**is a **Javascript runtime environment**. |
| Javascript can only be run in the browsers. | We can run Javascript outside the browser with the help of NodeJS. |
| It is used on the client side. | It is mostly used on the server side. |
| Javascript is capable enough to add HTML and play with the DOM. | Nodejs does not have capability to add HTML tags. |
| Javascript can run in any browser engine as like JS core in Safari and Spidermonkey in Firefox. | V8 is the Javascript engine inside of node.js that parses and runs Javascript. |
| Javascript is used in front-end development. | Nodejs is used in server-side development. |
| It is the upgraded version of the ECMA script that uses Chrome’s V8 engine written in C++. | Nodejs is written in C, C++ and Javascript. |

### **1. Install Node.js**

If Node.js is not already installed on your system:

* **Download Node.js** from the [official Node.js website](https://nodejs.org/).
  + Choose the **LTS version** for stability or the **Current version** for the latest features.
* Follow the installation instructions for your operating system (Windows, macOS, or Linux).

### **2. Verify Installation**

After installation, verify that Node.js and NPM (Node Package Manager) are installed:

* Open a terminal (Command Prompt,or any terminal in your OS).

node -v

Check NPM

npm -v

#### **Node REPL (Read-Eval-Print Loop):**

The Node REPL is an interactive shell that allows you to run JavaScript code line by line. It's a great tool for testing code snippets and exploring JavaScript features.

To launch the Node REPL, open your terminal and type node. You'll see a prompt (>) where you can enter JavaScript code.

JavaScript

let greeting = 'Hello, Node.js!';

undefined >

console.log(greeting);

Hello, Node.js!

undefined

### **3. Run Node.js**

There are a few ways to start Node.js, depending on your needs:

#### **a. Start Node.js REPL (Interactive Mode)**

* Open a terminal and type:

node

* Press **Enter** to open the Node.js REPL environment.
* Now you can type and execute JavaScript commands interactively:

> console.log('Hello, Node.js!');

Hello, Node.js!

Undefined

#### **b. Run a JavaScript File**

1. Create a new JavaScript file, e.g., app.js, in your project folder.
2. Add some code to the file, like:

console.log('Hello from Node.js!');

1. Open a terminal, navigate to the file's directory, and run:

node app.js

1. You should see the output:

Hello from Node.js!

### **4. Install and Use Node.js Packages**

Node.js works seamlessly with NPM, allowing you to install packages and modules:

* To install a package:

npm install <package-name>

For example, install express:

npm install express

### **Key Fields in** package.json

| **Field** | **Description** |
| --- | --- |
| **name** | The name of your project/package. Must be lowercase and URL-safe. |
| **version** | The version of your project, following [SemVer](https://semver.org/). |
| **description** | A short description of your project. |
| **main** | The entry point of your application (e.g., index.js). |
| **scripts** | Custom commands that can be run using npm run <script-name>. |
| **keywords** | An array of keywords to help others find your project (useful for publishing). |
| **author** | The author of the project. |
| **license** | The license type for your project (e.g., MIT, Apache-2.0). |
| **dependencies** | Lists libraries your app needs to run. |
| **devDependencies** | Lists libraries used for development only (e.g., testing tools) |

### **Common NPM Commands**

Here are some commonly used NPM commands:

| **Command** | **Description** |
| --- | --- |
| npm init | Initializes a new Node.js project and creates a package.json file. |
| npm install <package> | Installs a package locally (e.g., npm install express). |
| npm install <package> -g | Installs a package globally, making it accessible from anywhere (e.g., npm install -g nodemon). |
| npm update | Updates all installed packages to their latest versions. |
| npm uninstall <package> | Removes a package from your project. |
| npm run <script> | Runs a custom script defined in the package.json file. |
| npm list | Displays all installed packages and their dependencies. |
| npm outdated | Shows packages that are outdated and need updating. |
| npm publish | Publishes your own package to the NPM Registry. |

**How to Use Node.jS?,**

1. npm (Node Package Manager):

Where to Use: Package management: npm is used to install, manage, and share JavaScript

packages (libraries and tools).

**How to Use:**

Install a package: npm install <package>

Initialize a new Node.js project: npminit

Start the application defined in the "scripts" section: npm start

**Why to Use:**

Simplifies dependency management and allows easy sharing of code with

other developers.

**2. node**:

Execute JavaScript code outside of a web browser using the Node.js runtime.

● Execute a script: node myScript.js

● Allows running JavaScript code on the server-side, enabling the development of

server applications.

**3. npx:**

● Execute npm package binaries without installing them globally.

● Run a package binary: npx<package>

● Provides a way to use tools without installing them globally, reducing potential

conflicts.

**4. nodemon:**

● Development workflow to automatically restart the server on file changes.

● Run a script with nodemon: nodemon myServer.js

● Saves time by avoiding manual server restarts during development.

**5. npm-run-all:**

● Run multiple npm scripts sequentially or in parallel.

● Run scripts concurrently: npm-run-all script1 script2

● Streamlines complex build or development workflows.

**6. pm2:**

● Process management for Node.js applications in production.

● Start a Node.js application: pm2 start myApp.js

● Ensures continuous uptime, automatic restarts, and load balancing in

production environments.

**7. eslint:**

● Code quality enforcement and identifying issues in JavaScript code.

● Lint a file: eslint myFile.js

● Helps maintain a consistent and high-quality codebase.

**8. ava:**

● Test runner for Node.js applications.

● Run tests: ava test.js

● Facilitates writing and running test cases, ensuring code reliability.

**9. forever:**

● Ensure that a given script runs continuously as a persistent process.

● Start a script with forever: forever start myScript.js

● Useful for running scripts persistently, especially in server environments.

**Where to Use Command Utilities:**

Here's a brief overview of how to

use some common Node.js command utilities:

• Web servers and APIs

• Real-time apps

• Command-line tools

• Desktop apps (via Electron)

• Microservices

• IoT applications

• Serverless functions

**Node.js Command Line Options**

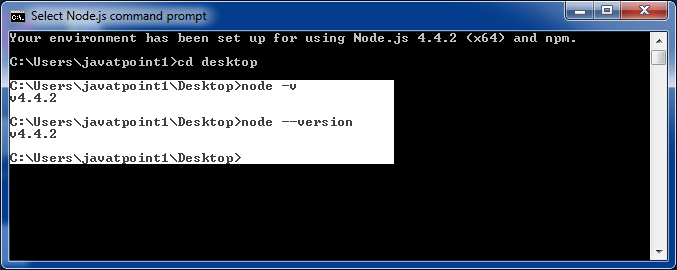
|  |  |  |
| --- | --- | --- |
| **Index** | **Option** | **Description** |
| 1. | v, --version | It is used to print node's version. |
| 2. | -h, --help | It is used to print node command line options. |
| 3. | -e, --eval "script" | It evaluates the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script. |
| 4. | -p, --print "script" | It is identical to -e but prints the result. |
| 5. | -c, --check | Syntax check the script without executing. |
| 6. | -i, --interactive | It opens the REPL even if stdin does not appear to be a terminal. |
| 7. | -r, --require module | It is used to preload the specified module at startup. It follows require()'s module resolution rules. Module may be either a path to a file, or a node module name. |
| 8. | --no-deprecation | Silence deprecation warnings. |
| 9. | --trace- deprecation | It is used to print stack traces for deprecations. |
| 10. | --throw- deprecation | It throws errors for deprecations. |
| 11. | --no-warnings | It silence all process warnings (including deprecations). |
| 12. | --trace-warnings | It prints stack traces for process warnings (including deprecations). |
| 13. | --trace-sync-io | It prints a stack trace whenever synchronous i/o is detected after the first turn of the event loop. |
| 14. | --zero-fill-buffers | Automatically zero-fills all newly allocated buffer and slowbuffer instances. |
| 15. | --track-heap- objects | It tracks heap object allocations for heap snapshots. |
| 16. | --prof-process | It processes V8 profiler output generated using the v8 option --prof. |

|  |  |  |
| --- | --- | --- |
| 17. | --V8-options | It prints V8 command line options. |
| 18. | --tls-cipher- list=list | It specifies an alternative default tls cipher list. (requires node.js to be built with crypto support. (default)) |
| 19. | --enable-fips | It enables fips-compliant crypto at startup. (requires node.js to be built with ./configure –openssl-fips) |
| 20. | --force-fips | It forces fips-compliant crypto on startup. (cannot be disabled from script code.) (same requirements as --enable-fips) |
| 21. | --icu-data- dir=file | It specifies ICU data load path. (Overrides node\_icu\_data) |

**Examples: -**

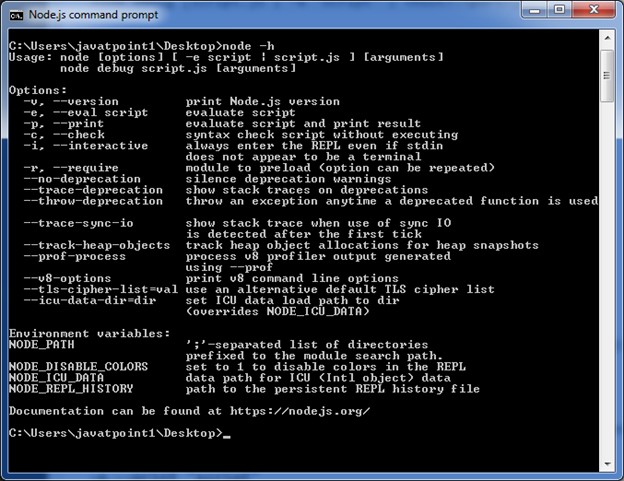
To see the version of the running Node:

Open Node.js command prompt and run command node -v or node --version



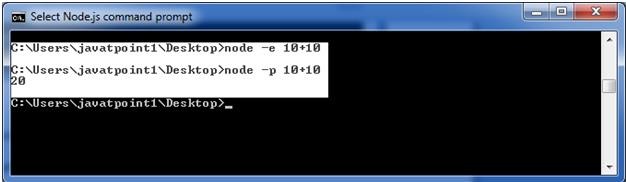
**For Help:**

Use command node? h or node --help



To evaluate an argument (but not print result): Use command node -e, --eval "script"

To evaluate an argument and print result also: Use command node -p "script"



To open REPL even if stdin doesn't appear: Use command node -i, or node --interactive



#### **Node Modules**

* **Node JS Modules**
  + In Node.js, modules are a fundamental concept that allows you to organize code into reusable and maintainable units.
  + Modules can be a single file or a collection of multiple files/folders.
  + Programmers are heavily reliant on modules is because of their reusability as well as the ability to break down a complex piece of code into manageable chunks.
  + The CommonJS pattern, using **exports** and **require**, is used for defining and importing modules.

#### **✔ Basic Structure of a Module**:

A Node.js module typically consists of the following elements:

#### **Exports:**

* + Objects, functions, or variables that are made available to other modules.
  + Defined using **module. Exports** or **exports**.

#### **Require:**

* + Allows a module to include and use functionality from another module.
  + Achieved using the **require** function.

#### ✔ Modules are of three types:

✔ Core Modules

✔ local Modules

✔ Third-party Modules

#### **✔ Core Modules**

fs: Handles file system operations.

http: Enables creating servers and handling HTTP requests.

path: Provides utilities for working with file and directory paths.

os: Retrieves information about the operating system.

events: Implements an event-driven programming model.

* + - Core modules in Node.js are built-in modules provided by the Node.js runtime itself.
    - These modules offer essential functionalities for common tasks, such as working with the file system, handling HTTP requests, managing paths, and more.
    - Core modules are part of the Node.js standard library and are available for use without requiring installation via npm.
    - To use a core module, you simply need to use the **require** function.

**Syntax:** const module =require('module\_name');

Here are some examples of core modules in Node.js:

**1. Core Modules**

These are built into Node.js and provide essential functionalities without requiring installation.Node.js comes with a set of built-in modules that provide various functionalities. Here are explanations for a few of them with code snippets:

### 1. fs **Module (File System):**

* The fs module provides an API for interacting with the file system.

const fs = require('fs');

// Reading a file asynchronously

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) {

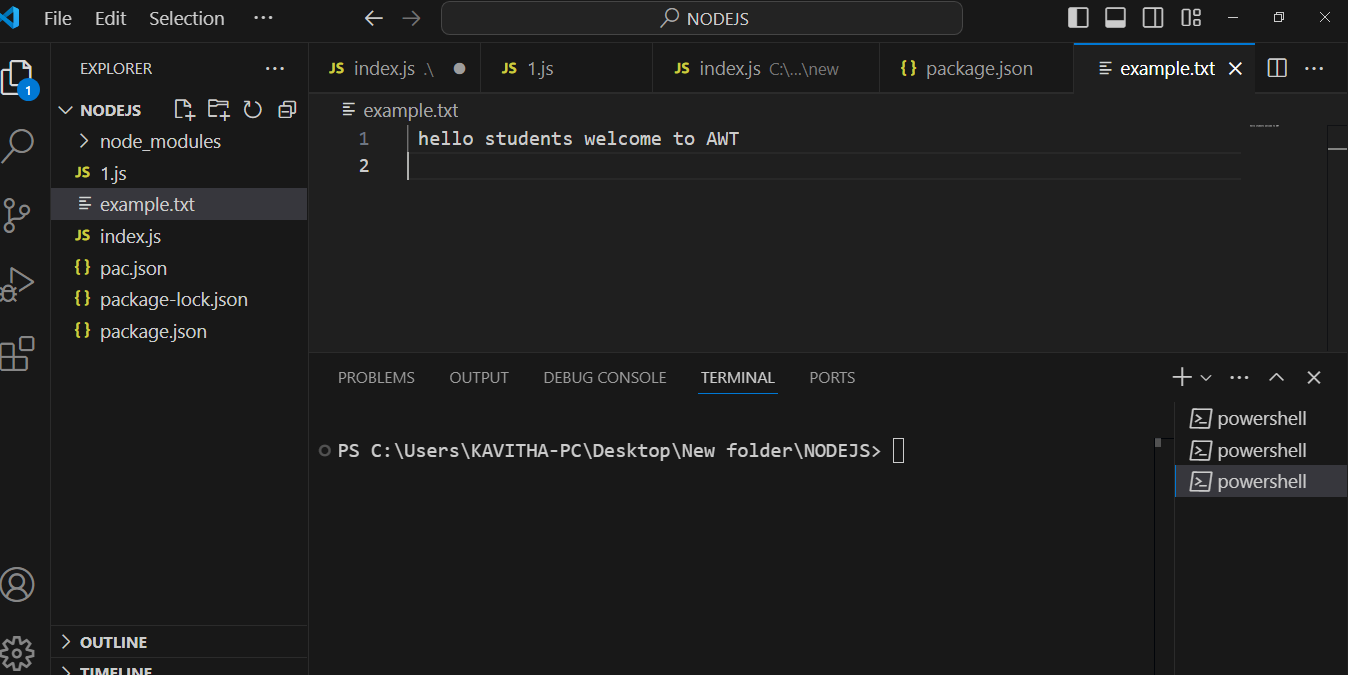
console.error(err);

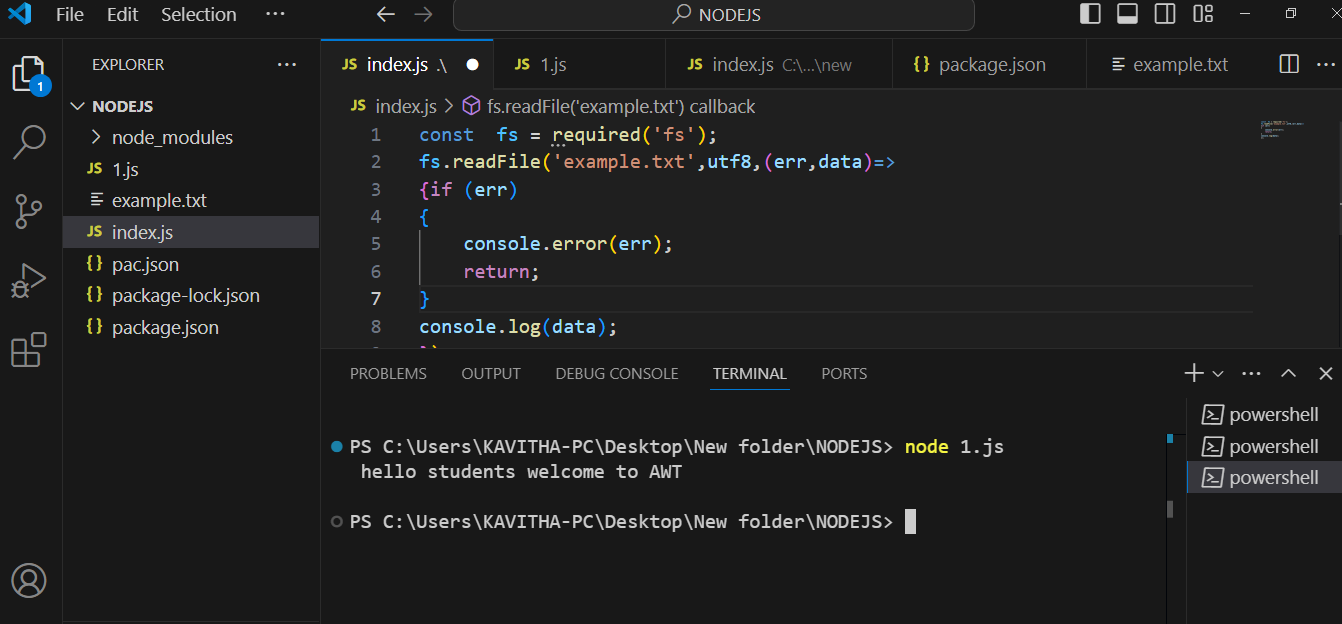
return;

}

console.log(data);

});





**// Writing to a file asynchronously**

const fs = require('fs');

fs.writeFile('newFile.txt', 'Hello, Node.js!', (err) => {

if (err) {

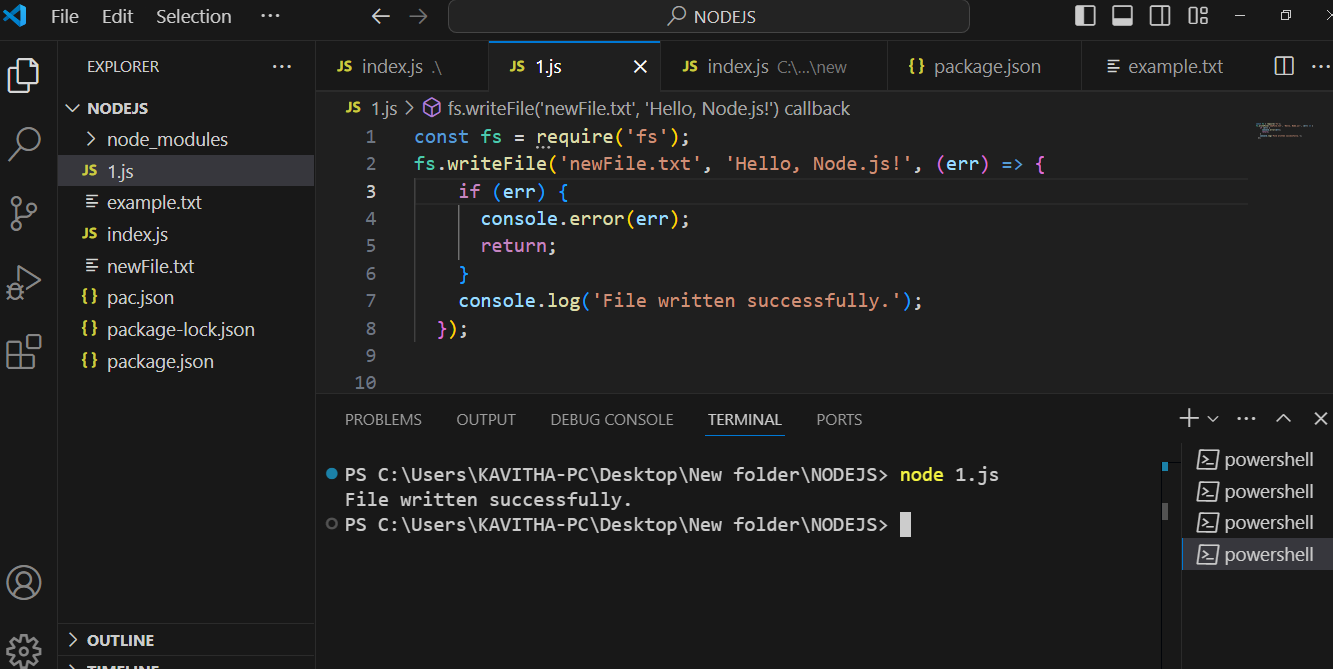
console.error(err);

return;

}

console.log('File written successfully.');

}**);**

****

**2. `http` Module (HTTP Server):**

The `http` module allows you to create HTTP servers and make HTTP requests.

const http = require('http');

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

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

res.end('Hello, Node.js!');

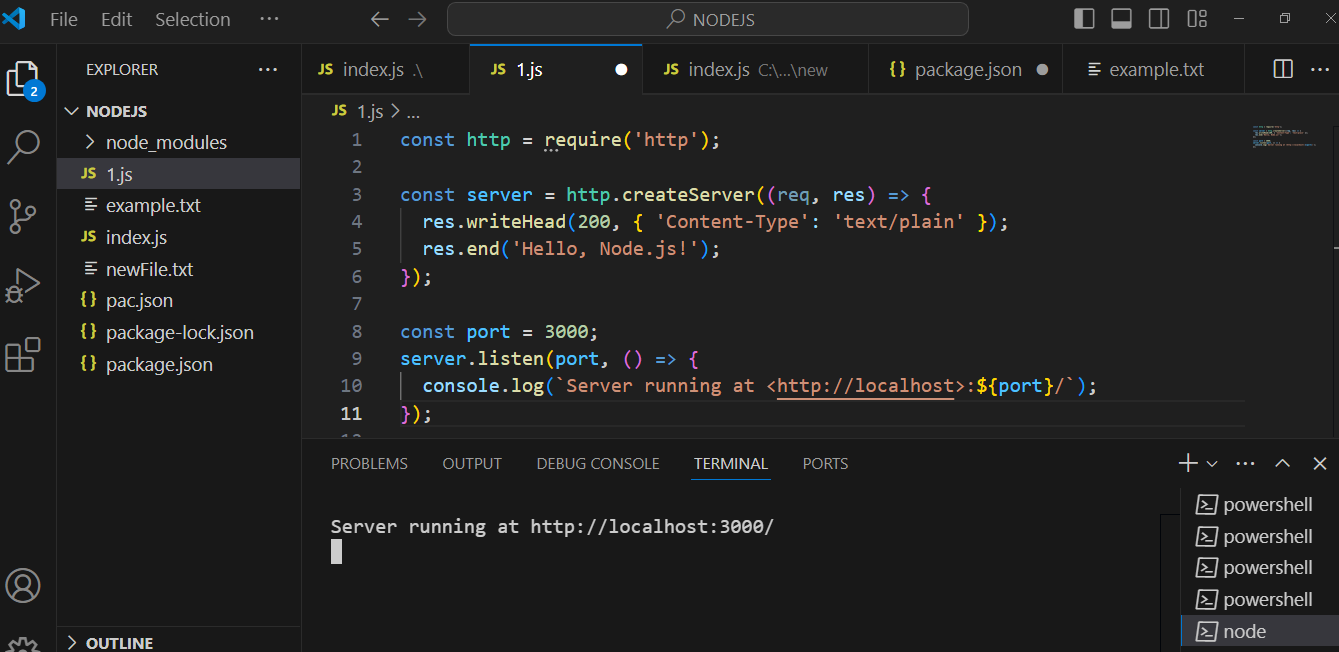
});

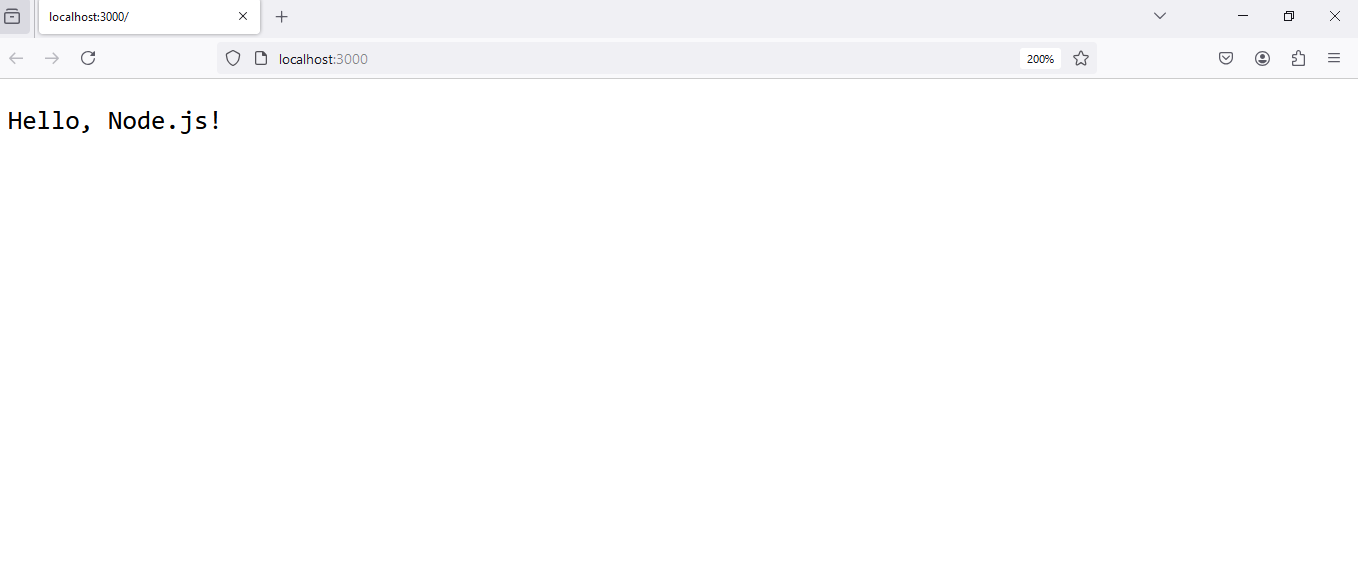
const port = 3000;

server.listen(port, () => {

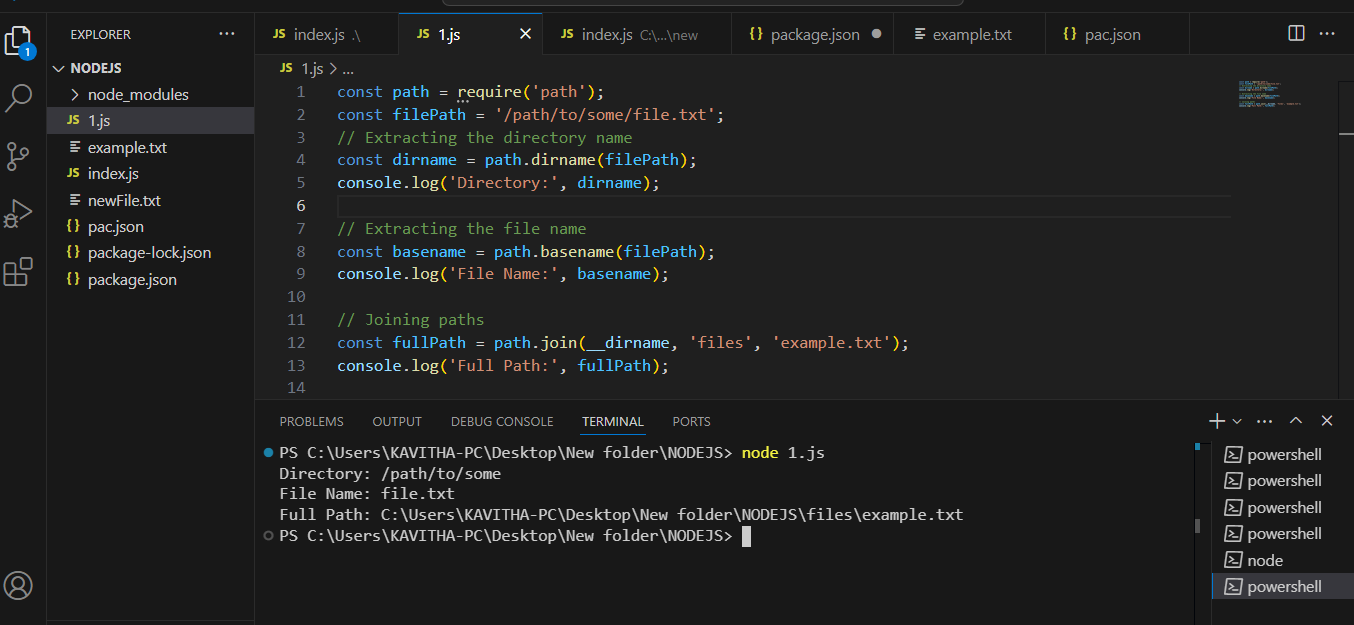
console.log(`Server running at <http://localhost>:${port}/`);

});

****

****

**3. `path` Module (Path Operations):**

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

4. os:

● Offers operating system-related utility methods.

● Example:

const os = require('os'); console.log('Platform:', os.platform()); console.log('Architecture:', os.arch()); console.log('Total Memory:', os.totalmem());

5. events:

● Provides an event-driven programming interface.

● Example:

const EventEmitter = require('events'); class MyEmitter extends EventEmitter {} const myEmitter = new MyEmitter(); myEmitter.on('event', () => { console.log('Event occurred!'); }); myEmitter.emit('event');

6. util:

● Provides various utility functions.

● Example:

const util = require('util'); const myFunction = util.promisify(someAsyncFunction);

7. querystring:

● Provides methods for working with URL query strings.

● Example:

const querystring = require('querystring'); const params = { name: 'John', age: 30

}; const queryString = querystring.stringify(params); console.log(queryString);

8. crypto:

● Implements cryptographic functionality.

● Example:

const crypto = require('crypto'); const hash = crypto.createHash('sha256'); hash.update('Hello, Node.js!'); const hashedData = hash.digest('hex'); console.log(hashedData);

9. zlib:

● Provides compression and decompression functionalities.

● Example:

const zlib = require('zlib'); const fs = require('fs'); const gzip = zlib.createGzip(); const readStream = fs.createReadStream('file.txt'); const writeStream = fs.createWriteStream('file.txt.gz'); readStream.pipe(gzip).pipe(writeStream);

**2. Local Modules**

These are custom modules created by developers. They are specific to the application and often written in separate .js files.

● local modules are created locally in your Node.js application

● Local modules in Node.js refer to custom modules created by developers within their project or application. These modules encapsulate related functionalities, making the code more modular, organized, and maintainable.

● Local modules are particularly useful for separating concerns and promoting code reuse.

// math.js (Local Module)

module.exports.add = (a, b) => a + b;

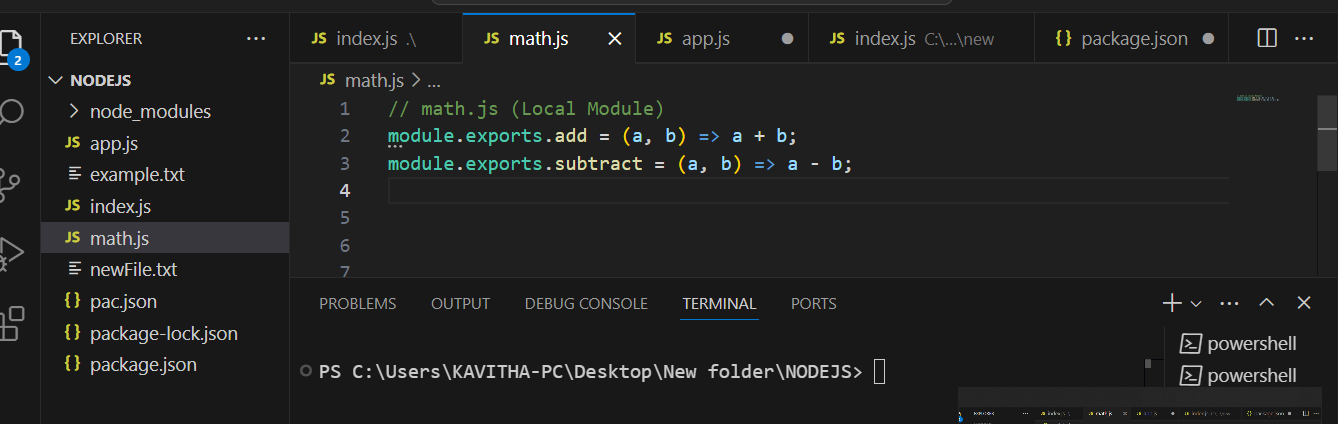
module.exports.subtract = (a, b) => a - b;

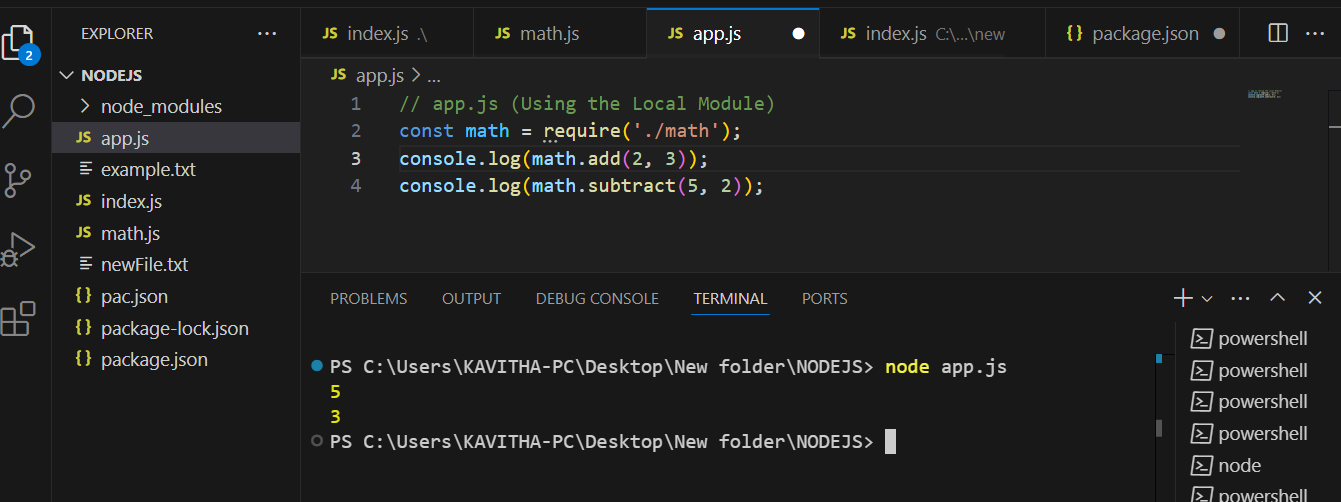
// app.js (Using the Local Module)

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

console.log(math.add(2, 3)); // Output: 5

console.log(math.subtract(5, 2)); // Output: 3





**3. Third-party Modules**

These are modules available through the Node Package Manager (NPM). Developers can install, use, and share these modules.

1. Install the Module:
   * Use the npm install command to install a third-party module. For example, to install the axios module for making HTTP requests:

npm install axios

1. Require the Module:
   * In your Node.js script, use the require function to include the module. For example:

const axios = require('axios');

1. Use the Module:
   * Once the module is installed and required, you can use its functions, objects, or methods in your code. For example:

axios.get('https://jsonplaceholder.typicode.com/users')

.then(response=> { console.log(response.data); }) .catch(error => { console.error('Error:', error.message);

});

#### **Where to Use Third-Party Modules:**

1. Web Development:
   * Use third-party modules for tasks such as handling HTTP requests (e.g., axios, request-promise), creating web servers (e.g., express, koa), and managing authentication (e.g., passport).
2. Database Integration:
   * Third-party modules like mongoose are commonly used for connecting to and interacting with databases (e.g., MongoDB).
3. Utility Functions:
   * Use modules like lodash for utility functions that simplify common programming tasks.
4. Testing:
   * Third-party testing frameworks (e.g., mocha, jest) and assertion libraries (e.g., chai, assert) can be employed for writing and running tests.
5. Authentication and Authorization:
   * Modules like passport are widely used for implementing authentication strategies in web applications.
6. Logging and Debugging:
   * Utilize logging modules (e.g., winston, morgan) to enhance debugging and monitor application behavior.
7. File Uploads:
   * Modules like multer simplify handling file uploads in web applications.

#### **Applications of using Third-Party Modules:**

1. Time Efficiency:
   * Third-party modules can save development time by providing pre-built solutions for common tasks, allowing developers to focus on application-specific logic.
2. Code Quality:
   * Reputable third-party modules are often well-tested and maintained, contributing to overall code quality and reliability.
3. Community Collaboration:
   * Leveraging third-party modules allows you to benefit from the expertise and collaboration of the broader Node.js community.
4. Scalability:
   * Many third-party modules are designed to scale, enabling developers to build scalable applications without reinventing the wheel.
5. Feature Enhancement:
   * Third-party modules often provide advanced features or functionalities that might be complex to implement from scratch.
6. Security:
   * Popular and well-maintained modules are regularly updated to address security vulnerabilities, contributing to a more secure development environment.
7. Ecosystem Compatibility:
   * Third-party modules are designed to work seamlessly within the Node.js ecosystem, providing compatibility with other modules and tools.

#### **List of some commonly used and popular third-party modules in the Node.js ecosystem:**

1. ***Express:*** *A minimal and flexible web application framework for building web and mobile applications.*
   * [*Express on npm*](https://www.npmjs.com/package/express)
2. ***Lodash:*** *A utility library that provides helper functions for common programming tasks.*
   * [*Lodash on npm*](https://www.npmjs.com/package/lodash)
3. ***Axios:*** *A promise-based HTTP client for making HTTP requests.*
   * [*Axios on npm*](https://www.npmjs.com/package/axios)
4. ***Mongoose:*** *An elegant MongoDB object modeling for Node.js.*
   * [*Mongoose on npm*](https://www.npmjs.com/package/mongoose)
5. ***Request:*** *Simplified HTTP request client.*
   * [*Request on npm*](https://www.npmjs.com/package/request)
6. ***Body-parser:*** *Middleware for parsing incoming request bodies in a middleware-friendly way.*
   * [*Body-parser on npm*](https://www.npmjs.com/package/body-parser)
7. ***Multer:*** *Middleware for handling* ***multipart/form-data****, used for file uploads.*
   * [*Multer on npm*](https://www.npmjs.com/package/multer)
8. ***Passport:*** *Authentication middleware for Node.js.*
   * [*Passport on npm*](https://www.npmjs.com/package/passport)
9. ***Socket.io:*** *Enables real-time, bidirectional, and event-based communication.*
   * [*Socket.io on npm*](https://www.npmjs.com/package/socket.io)
10. ***Jest:*** *A delightful JavaScript testing framework.*
    * [*Jest on npm*](https://www.npmjs.com/package/jest)
11. ***Mocha:*** *A feature-rich JavaScript test framework.*
    * [*Mocha on npm*](https://www.npmjs.com/package/mocha)
12. ***Chai:*** *Assertion library for node and browsers.*
    * [*Chai on npm*](https://www.npmjs.com/package/chai)