What is Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link <https://nodejs.org/en/>

Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly used to run real-time server applications.

The definition given by its official documentation is as follows:

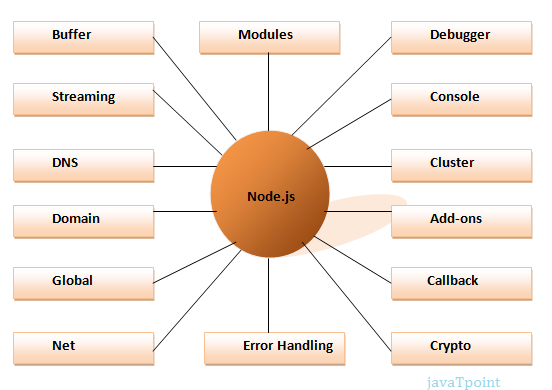
Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

1. Node.js = Runtime Environment + JavaScript Library

**Different parts of Node.js**

The following diagram specifies some important parts of Node.js:



Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:**Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:**All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:**Node.js follows a single threaded model with event looping.
4. **Highly Scalable:**Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:**Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:**Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:**Node.js is released under the MIT license.

# **Node.js First Example**

There can be console-based and web-based node.js applications.

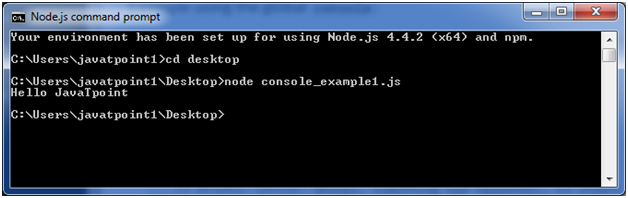
## Node.js console-based Example

File: console\_example1.js

1. console.log('Hello');

Open Node.js command prompt and run the following code:

1. node console\_example1.js



Here, console.log() function displays message on console.

## Node.js web-based Example

A node.js web application contains the following three parts:

1. **Import required modules:** The "require" directive is used to load a Node.js module.
2. **Create server:**You have to establish a server which will listen to client's request similar to Apache HTTP Server.
3. **Read request and return response:** Server created in the second step will read HTTP request made by client which can be a browser or console and return the response.

**How to create node.js web applications**

Follow these steps:

1. **Import required module:**The first step is to use ?require? directive to load http module and store returned HTTP instance into http variable. For example:
   1. var http = require("http");
2. **Create server:**In the second step, you have to use created http instance and call http.createServer() method to create server instance and then bind it at port 8081 using listen method associated with server instance. Pass it a function with request and response parameters and write the sample implementation to return "Hello World". For example:

http.createServer(function (request, response) {

   // Send the HTTP header

   // HTTP Status: 200 : OK

   // Content Type: text/plain

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

   // Send the response body as "Hello World"

   response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

**Combine step1 and step2 together** in a file named "main.js".

File: main.js

var http = require("http");

http.createServer(function (request, response) {

 // Send the HTTP header

   // HTTP Status: 200 : OK

   // Content Type: text/plain

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

   // Send the response body as "Hello World"

   response.end('Hello World\n');

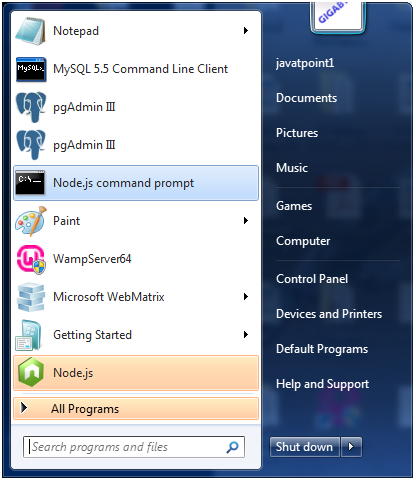
}).listen(8081);

// Console will print the message

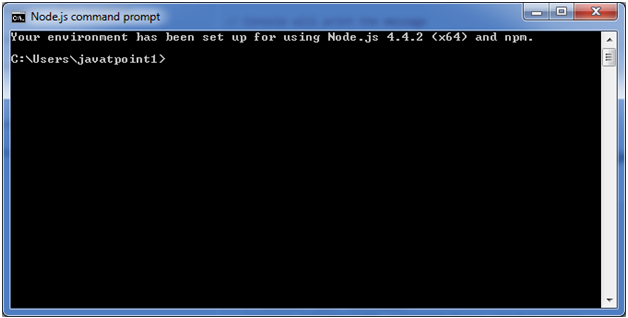
console.log('Server running at http://127.0.0.1:8081/');

**How to start your server:**

Go to start menu and click on the Node.js command prompt.



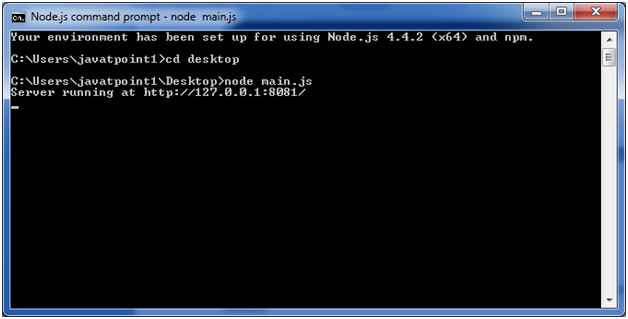
Now command prompt is open:



**Set path:**Here we have save "main.js" file on the desktop.

So type **cd desktop** on the command prompt. After that execute the main.js to start the server as follows:

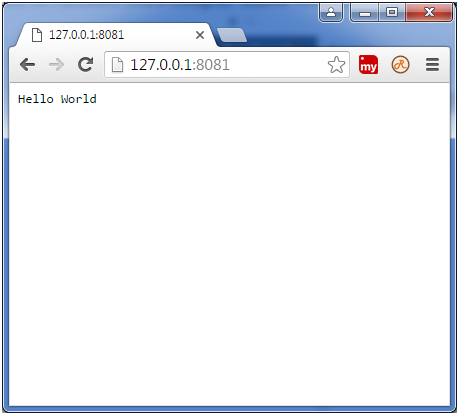
1. node main.js



Now server is started.

**Make a request to Node.js server:**

Open http://127.0.0.1:8081/ in any browser. You will see the following result.



# **Node.js REPL**

The term REPL stands for **Read Eval Print**and**Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

## REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

**Read:** It reads user's input; parse the input into JavaScript data-structure and stores in memory.

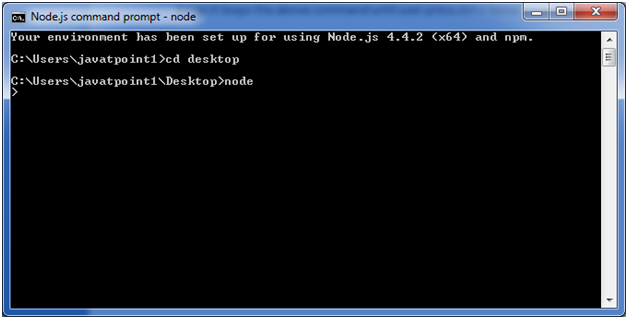
**Eval:**It takes and evaluates the data structure.

**Print:**It prints the result.

**Loop:** It loops the above command until user press ctrl-c twice.

## How to start REPL

You can start REPL by simply running "node" on the command prompt. See this:

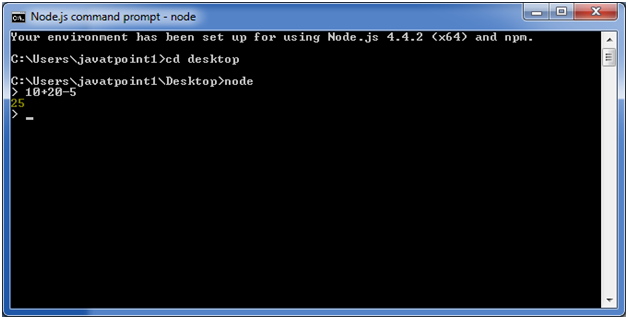


You can execute various mathematical operations on REPL Node.js command prompt:

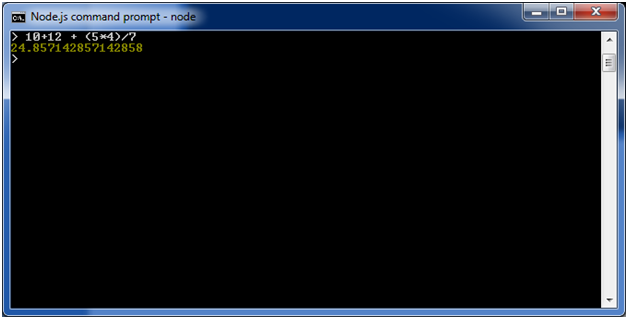
## Node.js Simple expressions

After starting REPL node command prompt put any mathematical expression:

1. Example: **>**10+20-5
2. 25



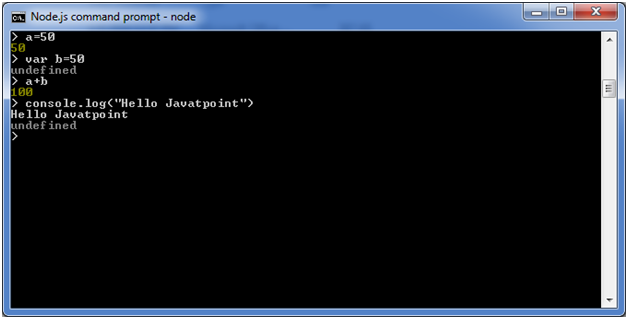
1. Example2: **>**10+12 + (5\*4)/7



## Using variable

Variables are used to store values and print later. If you don't use **var**keyword then value is stored in the variable and printed whereas if **var** keyword is used then value is stored but not printed. You can print variables using console.log().

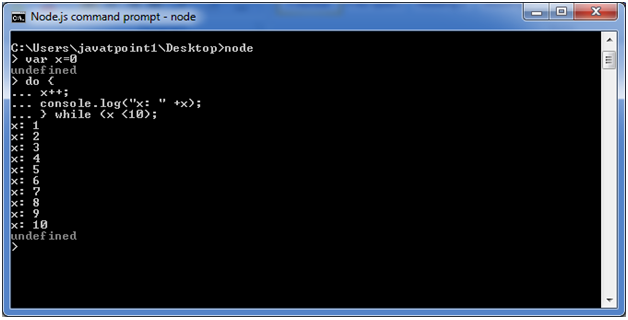
**Example:**



## Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript. See the following do-while loop example:

1. var x = 0
2. undefined
3. **>** do {
4. ... x++;
5. ... console.log("x: " + x);
6. ... } while ( x **<** 10 );



## Node.js Underscore Variable

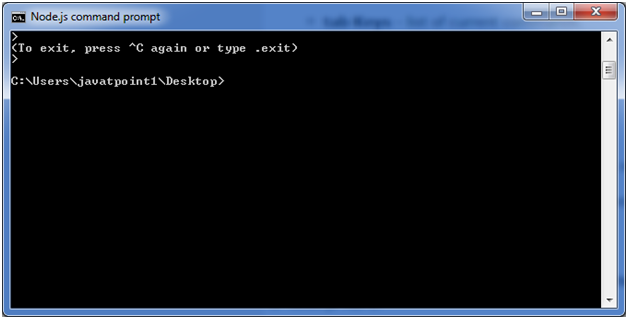
You can also use underscore \_ to get the last result.

**Example:**

## Node.js REPL Commands

## Node.js Exit REPL

Use ctrl + c command twice to come out of Node.js REPL.



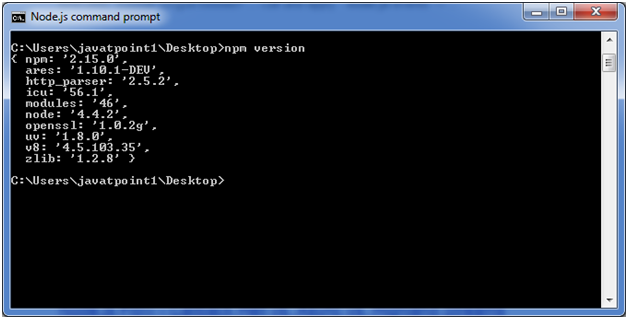
# **Node.js Package Manager**

Node Package Manager provides two main functionalities:

* It provides online repositories for node.js packages/modules which are searchable on search.nodejs.org
* It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

The npm comes bundled with Node.js installables in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

1. npm  version



## Installing Modules using npm

Following is the syntax to install any Node.js module:

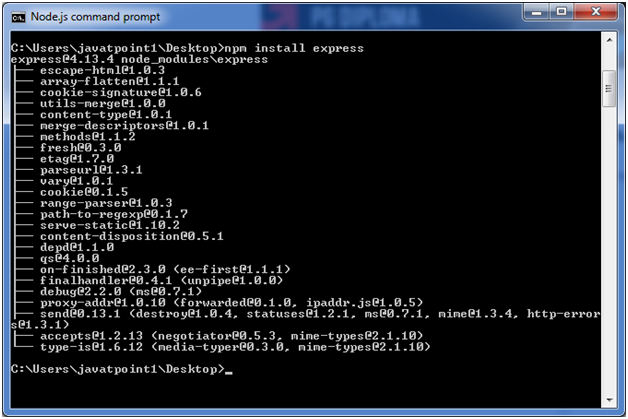
1. npm install **<**Module Name**>**

Let's install a famous Node.js web framework called express:

Open the Node.js command prompt and execute the following command:

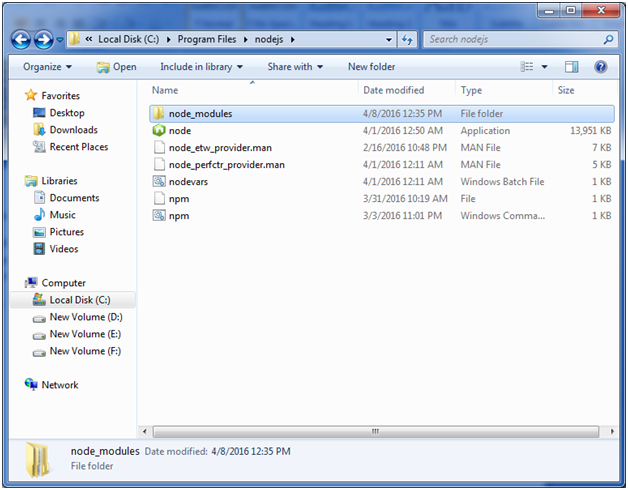
1. npm install express

You can see the result after installing the "express" framework.



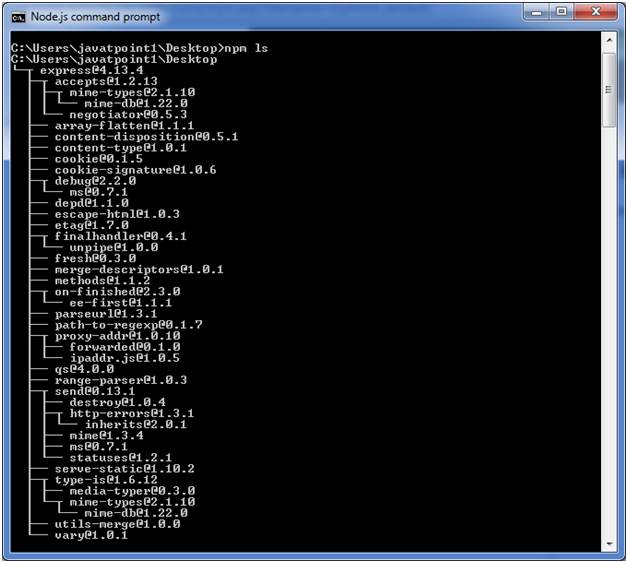
## Global vs Local Installation

By default, npm installs dependency in local mode. Here local mode specifies the folder where Node application is present. For example if you installed express module, it created node\_modules directory in the current directory where it installed express module.



You can use npm ls command to list down all the locally installed modules.

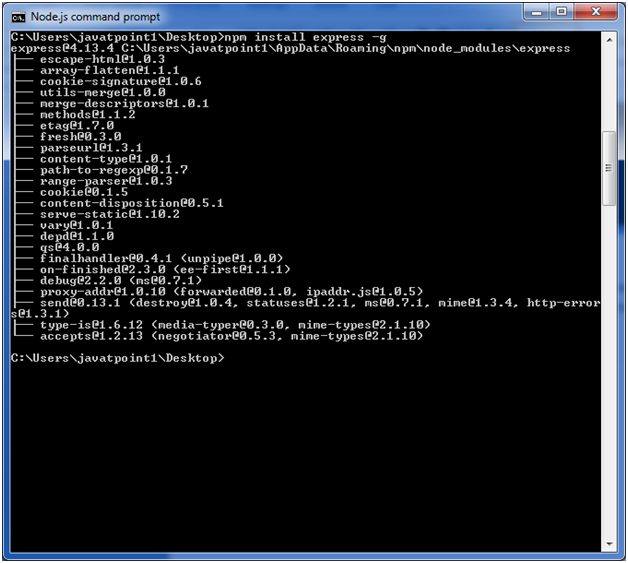
Open the Node.js command prompt and execute "npm ls":



Globally installed packages/dependencies are stored in system directory. Let's install express module using global installation. Although it will also produce the same result but modules will be installed globally.

Open Node.js command prompt and execute the following code:

1. npm install express -g

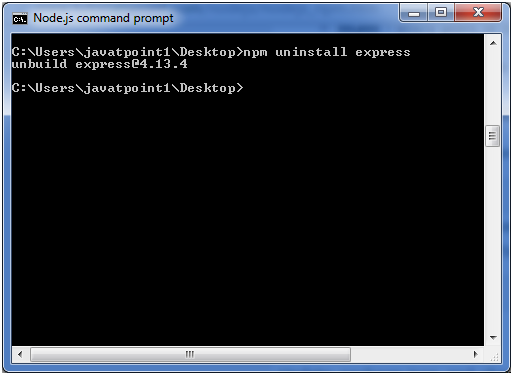


Here first line tells about the module version and its location where it is getting installed.

## Uninstalling a Module

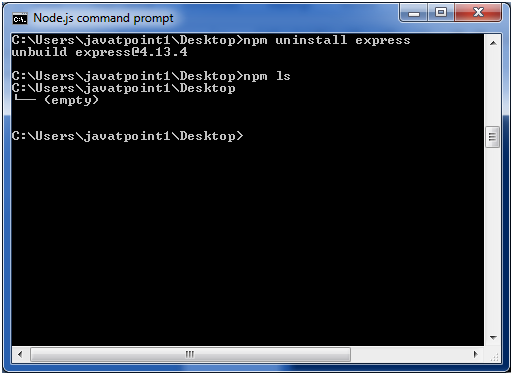
To uninstall a Node.js module, use the following command:

1. npm uninstall express



The Node.js module is uninstalled. You can verify by using the following command:

1. npm ls

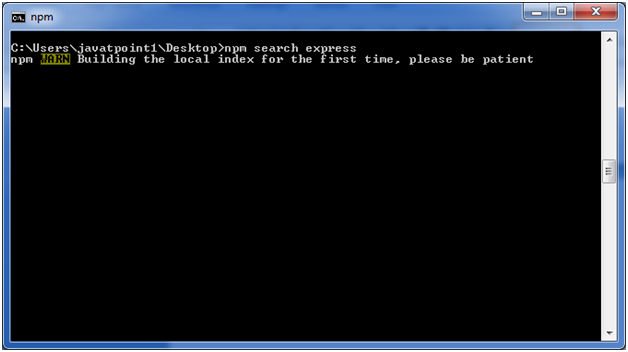
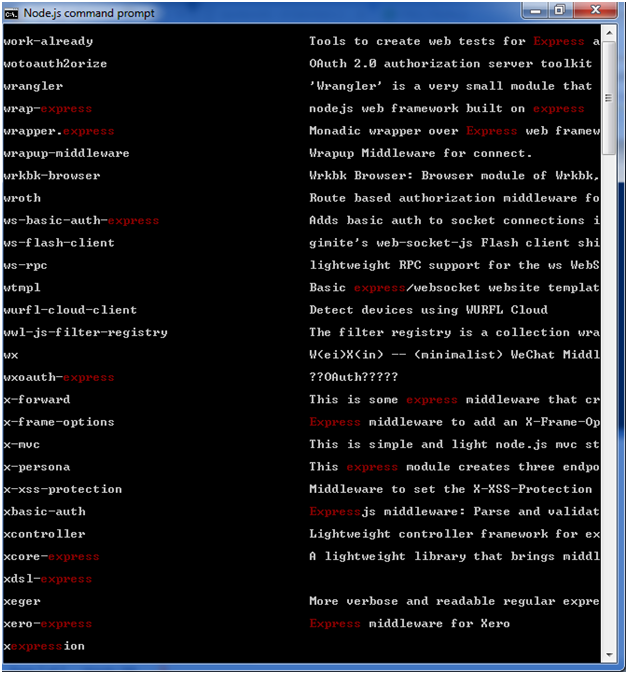


You can see that the module is empty now.

## Searching a Module

"npm search express" command is used to search express or module.

1. npm search express

In **Node.js**, global objects are objects that are **available in all modules**. You can access them without requiring any modules or importing anything. These objects are part of the global scope, but unlike in the browser (where window is the global object), Node.js uses a different global object called global.

### 🔹 Common Global Objects in Node.js

| **Global Object** | **Description** |
| --- | --- |
| global | The global namespace object (like window in browsers). Variables attached to it are accessible globally. |
| \_\_dirname | The absolute path of the directory containing the current module. |
| \_\_filename | The absolute path of the current module file. |
| module | Represents the current module and has information about it (like exports). |
| exports | Used to export functions/objects from a module. |
| require() | Function to import modules. |
| setTimeout() | Executes a function after a delay. |
| setInterval() | Executes a function repeatedly at intervals. |
| clearTimeout() | Cancels a setTimeout. |
| clearInterval() | Cancels a setInterval. |
| Console | Used to print output to the terminal (log, error, etc.). |
| Process | Provides information and control over the current Node.js process. Useful for environment variables, exiting, etc. |

### 🔹 Example

console.log(\_\_dirname); // Prints directory of the current file

console.log(\_\_filename); // Prints file path

setTimeout(() => {

console.log('Hello after 2 seconds');

}, 2000);

console.log(global.setTimeout === setTimeout); // true

### 🔸 Note

Unlike in browsers, **variables declared with let, const, or var inside a file are not added to the global object** in Node.js due to its **module system**.

## Node.js vs Browser JavaScript

| **Feature** | **Node.js** | **Browser JavaScript** |
| --- | --- | --- |
| **Environment** | Server-side | Client-side (browser) |
| **Global Object** | Global | window |
| **Module System** | CommonJS (require, module.exports) | ES Modules (import, export) |
| **File System Access** | ✅ Yes (fs module) | ❌ No (for security reasons) |
| **Networking** | ✅ Yes (http, net, https) | ✅ Limited (via fetch, WebSocket) |
| **Core Modules** | ✅ Yes (fs, path, http, etc.) | ❌ No |
| **DOM Access** | ❌ No | ✅ Yes (document, window, etc.) |
| **Event Loop** | ✅ Yes (uses libuv) | ✅ Yes (browser-based loop) |
| **Tooling** | Backend scripting, servers, CLIs | Frontend UI/UX interactions |
| **Package Manager** | Npm | Not required (uses CDN or bundlers like Webpack) |
| **Security Context** | Full access to OS (dangerous if misused) | Sandboxed (restricted access for security) |
| **Use Cases** | APIs, backend logic, database access, scripting | UI/UX, DOM manipulation, client interactions |

## 🔍 Example Differences

### ✅ Node.js (server-side)

const fs = require('fs');

fs.writeFileSync('file.txt', 'Hello from Node!');

### ✅ Browser (client-side)

document.querySelector('button').addEventListener('click', () => {

alert('Hello from Browser!');

});

## 🧠 TL;DR

| **Node.js is great for:** | **Browser JS is great for:** |
| --- | --- |
| Building APIs and servers | Handling user interaction |
| File & database operations | Manipulating the DOM |
| CLI tools & automation | Animations, rendering, UI logic |

Module 2

**fs module** in **Node.js** for basic file operations: **reading**, **writing**, **appending**, and **deleting** files.

## 🔹 Importing fs Module

const fs = require('fs');

You can use both **asynchronous (non-blocking)** and **synchronous (blocking)** methods.

## 📘 1. Reading Files

### ✅ Asynchronous (Recommended)

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

if (err) throw err;

console.log(data);

});

### 🛑 Synchronous

const data = fs.readFileSync('example.txt', 'utf8');

console.log(data);

## 📘 2. Writing Files (Creates new file or overwrites)

### ✅ Asynchronous

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

if (err) throw err;

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

});

### 🛑 Synchronous

fs.writeFileSync('example.txt', 'Hello, Node.js!');

## 📘 3. Appending to Files (Adds content to the end)

### ✅ Asynchronous

fs.appendFile('example.txt', '\nAppended content.', (err) => {

if (err) throw err;

console.log('Content appended!');

});

### 🛑 Synchronous

fs.appendFileSync('example.txt', '\nAppended content.');

## 📘 4. Deleting Files

### ✅ Asynchronous

fs.unlink('example.txt', (err) => {

if (err) throw err;

console.log('File deleted!');

});

### 🛑 Synchronous

fs.unlinkSync('example.txt');

## 💡 Notes:

* Always prefer **asynchronous** methods in production to avoid blocking the event loop.
* Use 'utf8' encoding to work with text files properly.
* Make sure to handle errors properly to avoid crashes.

The **path module** in **Node.js** provides utilities for **working with file and directory paths**. It's a built-in core module, so no installation needed.

**📦 Importing the Module**

const path = require('path');

**🔹 Commonly Used Methods**

| **Method** | **Description** |
| --- | --- |
| path.basename(p) | Returns the file name (with extension). |
| path.dirname(p) | Returns the directory name of the path. |
| path.extname(p) | Returns the file extension (e.g. .txt). |
| path.join([...paths]) | Joins paths together using the correct platform separator (/ or \). |
| path.resolve([...paths]) | Resolves absolute path from a sequence of paths. |
| path.parse(p) | Returns an object with root, dir, base, ext, and name. |
| path.format(obj) | Opposite of path.parse() — builds a path string from an object. |
| path.isAbsolute(p) | Checks if a path is absolute. |
| path.normalize(p) | Fixes and cleans up a path (removes redundant .., ., etc.). |
| path.sep | Platform-specific separator (\ on Windows, / on POSIX). |

**🧪 Examples**

const filePath = '/user/local/index.html';

console.log(path.basename(filePath)); // index.html

console.log(path.dirname(filePath)); // /user/local

console.log(path.extname(filePath)); // .html

console.log(path.join('/user', 'local', 'test.txt')); // /user/local/test.txt

console.log(path.resolve('folder', 'file.txt')); // Absolute path from current dir

console.log(path.parse(filePath));

// {

// root: '/',

// dir: '/user/local',

// base: 'index.html',

// ext: '.html',

// name: 'index'

// }

const parsed = path.parse(filePath);

console.log(path.format(parsed)); // /user/local/index.html

console.log(path.isAbsolute('folder/file.txt')); // false

console.log(path.normalize('/user//local/../test')); // /user/test

console.log(path.sep); // / (POSIX) or \ (Windows)

**📌 When to Use path**

* Creating OS-independent file paths
* Resolving relative and absolute paths
* Safely parsing filenames and extensions
* Avoiding hardcoded / or \ in file paths

The **os module** in **Node.js** provides built-in utilities for interacting with the **operating system** — such as memory, CPU info, user data, and system uptime. It’s useful for system diagnostics, logging, monitoring, and cross-platform support.

**📦 Import the Module**

const os = require('os');

**🔹 Common os Module Methods**

| **Method** | **Description** |
| --- | --- |
| os.arch() | Returns the CPU architecture (x64, arm, etc.). |
| os.platform() | Returns the OS platform (win32, linux, darwin, etc.). |
| os.type() | Returns the OS name (Linux, Windows\_NT, etc.). |
| os.hostname() | Returns the hostname of the system. |
| os.release() | Returns the OS release version. |
| os.uptime() | Returns the system uptime in seconds. |
| os.totalmem() | Returns total system memory in bytes. |
| os.freemem() | Returns free system memory in bytes. |
| os.cpus() | Returns info about each logical CPU core. |
| os.networkInterfaces() | Returns network interface details. |
| os.userInfo() | Returns information about the current user. |
| os.homedir() | Returns the path to the current user’s home directory. |
| os.tmpdir() | Returns the OS default temp directory. |
| os.endianness() | Returns the CPU endianness (BE or LE). |

**🧪 Example**

const os = require('os');

console.log('Platform:', os.platform());

console.log('Architecture:', os.arch());

console.log('OS Type:', os.type());

console.log('OS Release:', os.release());

console.log('Hostname:', os.hostname());

console.log('Total Memory:', os.totalmem());

console.log('Free Memory:', os.freemem());

console.log('Uptime (seconds):', os.uptime());

console.log('Home Directory:', os.homedir());

console.log('Temp Directory:', os.tmpdir());

console.log('User Info:', os.userInfo());

console.log('CPU Info:', os.cpus());

console.log('Network Interfaces:', os.networkInterfaces());

**💡 Use Cases**

* Building **system info dashboards**
* Monitoring **server health**
* Detecting **platform-specific behavior**
* Creating **temp file paths** or accessing **user home directories**

The **events module** in Node.js provides the **EventEmitter** class, which allows you to create and handle **custom events** — kind of like pub/sub or observer patterns.

## 📦 Importing and Creating an EventEmitter

const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

## 📘 Basic Usage Example

### 🔹 1. Register (Listen) for an Event

eventEmitter.on('greet', () => {

console.log('Hello! An event occurred.');

});

### 🔹 2. Emit (Trigger) the Event

eventEmitter.emit('greet');

🧠 Output:

Hello! An event occurred.

## ✨ Emitting with Arguments

eventEmitter.on('greetUser', (name) => {

console.log(`Hello, ${name}!`);

});

eventEmitter.emit('greetUser', 'Alice');

🧠 Output:

Hello, Alice!

## 🧯 Using .once() for One-Time Listeners

eventEmitter.once('firstTime', () => {

console.log('This runs only once.');

});

eventEmitter.emit('firstTime'); // ✅ runs

eventEmitter.emit('firstTime'); // ❌ ignored

## 🔥 Removing Listeners

function handler() {

console.log('Event triggered');

}

eventEmitter.on('eventX', handler);

eventEmitter.removeListener('eventX', handler);

// or: eventEmitter.off('eventX', handler); // modern way

## 🧠 Real-World Use Cases

* Handling server events (like HTTP requests)
* Custom pub/sub systems
* Chat apps (message events)
* File watchers
* Logging systems

const EventEmitter = require('events');

const emitter = new EventEmitter();

// Listener

emitter.on('dataReceived', (data) => {

console.log('Received:', data);

});

// Trigger

emitter.emit('dataReceived', { id: 1, name: 'Node.js' });

🧠 Output:

Received: { id: 1, name: 'Node.js' }

## Full Mini Demo

**File operations** (fs module) and **event-based programming** (events module) in a simple hands-on demo.

We'll create a program that:

* Writes to a file
* Appends content
* Reads the file
* Deletes the file
* Uses EventEmitter to trigger each operation in sequence

## 🛠️ Demo: Event-Based File Operations

### ✅ Step-by-step File Operations with Events

const fs = require('fs');

const EventEmitter = require('events');

// Create a custom event emitter

const fileEvents = new EventEmitter();

const fileName = 'demo.txt';

// 1. Write to file

fileEvents.on('write', () => {

fs.writeFile(fileName, 'Initial content.\n', (err) => {

if (err) throw err;

console.log('File written.');

fileEvents.emit('append');

});

});

// 2. Append content

fileEvents.on('append', () => {

fs.appendFile(fileName, 'Appended line.\n', (err) => {

if (err) throw err;

console.log('Content appended.');

fileEvents.emit('read');

});

});

// 3. Read file content

fileEvents.on('read', () => {

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

if (err) throw err;

console.log('Reading file:\n', data);

fileEvents.emit('delete');

});

});

// 4. Delete the file

fileEvents.on('delete', () => {

fs.unlink(fileName, (err) => {

if (err) throw err;

console.log('File deleted.');

});

});

// 🔥 Start the event chain

fileEvents.emit('write');

## 🧠 Output Will Be:

File written.

Content appended.

Reading file:

Initial content.

Appended line.

File deleted.

## 💡 What You Learn From This:

* Chaining operations using custom events
* Real-world async control using EventEmitter
* Clean, event-driven architecture for file handling

**Module 3**

**Synchronous vs Asynchronous** in **Node.js**, especially since it’s a key concept in how Node handles operations like file I/O, HTTP requests, etc.

## 🔁 Synchronous vs Asynchronous in Node.js

| **Concept** | **Synchronous** | **Asynchronous** |
| --- | --- | --- |
| **Definition** | Executes tasks **one after another**, blocking the thread until each is done. | Executes tasks **non-blocking**, allowing other operations to continue in parallel. |
| **Blocking?** | ✅ Yes | ❌ No |
| **Performance** | Slower for I/O tasks | Much faster for I/O-heavy applications |
| **Callback/Promise needed?** | ❌ No | ✅ Yes (Callback, Promise, or async/await) |
| **Example Use** | Simple scripts, or when the order of execution is critical | Server tasks, file I/O, API calls, DB queries |

## 🧪 Example with fs (File System)

### 📄 Synchronous (Blocking)

const fs = require('fs');

const data = fs.readFileSync('file.txt', 'utf8');

console.log('File content:', data);

console.log('This prints after reading is complete.');

🧠 Here, the second console.log waits until file is read.

### ⚡ Asynchronous (Non-blocking)

const fs = require('fs');

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

if (err) throw err;

console.log('File content:', data);

});

console.log('This prints immediately (before reading finishes).');

🧠 The read happens in the background. Other code runs while it’s waiting.

## ✅ Real-world Analogy

Imagine you're ordering food:

* **Synchronous**: You order food and **wait in line** until it's served before doing anything else.
* **Asynchronous**: You order food, they give you a buzzer, and **you go sit down** or do other stuff until it's ready.

## 🔥 Bonus: Asynchronous with Promises & async/await

const fs = require('fs').promises;

async function readFileAsync() {

try {

const data = await fs.readFile('file.txt', 'utf8');

console.log('File:', data);

} catch (err) {

console.error(err);

}

}

readFileAsync();

## 🧠 Summary

| **Task Type** | **Use Synchronous When** | **Use Asynchronous When** |
| --- | --- | --- |
| CPU-intensive or script-like | Quick scripts, small utilities | ✔ Not ideal for I/O |
| I/O (File, DB, APIs) | ❌ Avoid blocking the main thread | ✔ Use async for better performance |
| Real-time systems (e.g. servers) | ❌ No | ✔ Absolutely necessary |

**directory creation and removal** in Node.js using the fs module:

## 📁 1. Creating a Directory

### ✅ Asynchronous

const fs = require('fs');

fs.mkdir('myFolder', (err) => {

if (err) throw err;

console.log('Directory created!');

});

### ✅ With recursive: true (to create nested directories)

fs.mkdir('parentFolder/childFolder', { recursive: true }, (err) => {

if (err) throw err;

console.log('Nested directories created!');

});

### 🛑 Synchronous

fs.mkdirSync('myFolder');

## 🗑️ 2. Removing a Directory

Use fs.rmdir() (deprecated in latest versions) or fs.rm() instead.

### ✅ Asynchronous (Non-empty support in Node.js 14+)

fs.rm('myFolder', { recursive: true, force: true }, (err) => {

if (err) throw err;

console.log('Directory removed!');

});

### 🛑 Synchronous

fs.rmSync('myFolder', { recursive: true, force: true });

## ⚠️ Notes

* If you're using Node.js 14 or above, prefer fs.rm() over fs.rmdir() because rmdir() only works on **empty directories**.
* Use recursive: true if you want to remove folders with contents inside.
* force: true suppresses errors if the folder doesn't exist.

**Streams** and **Buffers** are powerful for handling **large files**, **data chunks**, or anything where you don’t want to load everything into memory at once.

**🚿 What Are Streams?**

Streams are **interfaces for working with streaming data** — chunks of data that arrive over time (e.g., file system, HTTP requests).

There are 4 types of streams:

1. **Readable** – e.g., reading from a file
2. **Writable** – e.g., writing to a file
3. **Duplex** – both readable and writable (e.g., TCP sockets)
4. **Transform** – duplex stream that transforms the data (e.g., zipping)

**📦 What Are Buffers?**

**Buffers** are temporary storage for binary data (used internally in streams). They're especially useful for working with binary data like images, videos, etc.

const buf = Buffer.from('Hello');

console.log(buf); // <Buffer 48 65 6c 6c 6f>

console.log(buf.toString()); // Hello

**📘 Read Stream Example (Reading a large file)**

const fs = require('fs');

const readStream = fs.createReadStream('bigfile.txt', 'utf8');

readStream.on('data', (chunk) => {

console.log('Received chunk:', chunk);

});

readStream.on('end', () => {

console.log('Finished reading');

});

readStream.on('error', (err) => {

console.error('Error:', err);

});

**📘 Write Stream Example (Writing to a file)**

const fs = require('fs');

const writeStream = fs.createWriteStream('output.txt');

writeStream.write('Hello, ');

writeStream.write('this is written in chunks.\n');

writeStream.end('Finished writing!\n');

writeStream.on('finish', () => {

console.log('All data written.');

});

writeStream.on('error', (err) => {

console.error('Error:', err);

});

**🔁 Piping Streams (Read + Write)**

const fs = require('fs');

const readStream = fs.createReadStream('input.txt');

const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);

console.log('Piping completed');

pipe() reads data chunk by chunk and sends it directly to the writable stream — very memory-efficient!

**💡 When to Use Streams and Buffers?**

* Handling **large files** (e.g., video, logs)
* Reading/writing **real-time data** (e.g., chats, live feeds)
* Efficiently managing **memory and performance**

## Project: File Logger App

### 🛠️ Features:

* Log messages to a file with timestamp
* Read logs on demand
* Clear the log file
* Event-driven operations

## 📁 fileLogger.js

const fs = require('fs');

const path = require('path');

const EventEmitter = require('events');

// Logger class extending EventEmitter

class Logger extends EventEmitter {

constructor(logFilePath) {

super();

this.logFile = logFilePath;

// Event listeners

this.on('log', this.writeLog);

this.on('read', this.readLog);

this.on('clear', this.clearLog);

}

getTimestamp() {

return new Date().toISOString();

}

writeLog = (message) => {

const entry = `[${this.getTimestamp()}] ${message}\n`;

fs.appendFile(this.logFile, entry, (err) => {

if (err) throw err;

console.log('✔ Log written.');

});

};

readLog = () => {

fs.readFile(this.logFile, 'utf8', (err, data) => {

if (err) {

if (err.code === 'ENOENT') {

console.log('⚠️ Log file does not exist yet.');

} else {

throw err;

}

} else {

console.log('📜 Log Contents:\n', data || '[Empty]');

}

});

};

clearLog = () => {

fs.writeFile(this.logFile, '', (err) => {

if (err) throw err;

console.log('🧹 Log file cleared.');

});

};

}

module.exports = Logger;

## 📄 app.js – Test the Logger

const path = require('path');

const Logger = require('./fileLogger');

// Create logger instance

const logPath = path.join(\_\_dirname, 'logs.txt');

const logger = new Logger(logPath);

// Fire events

logger.emit('log', 'App started');

logger.emit('log', 'User logged in');

logger.emit('read');

logger.emit('log', 'User clicked "Save"');

logger.emit('read');

logger.emit('clear');

logger.emit('read');

## 🧪 Output Example

✔ Log written.

✔ Log written.

📜 Log Contents:

[timestamp] App started

[timestamp] User logged in

✔ Log written.

📜 Log Contents:

[timestamp] App started

[timestamp] User logged in

[timestamp] User clicked "Save"

🧹 Log file cleared.

📜 Log Contents:

[Empty]

## ✅ What You Learned:

* File creation, appending, reading, clearing
* EventEmitter-based architecture
* Timestamps in logs
* Realistic Node.js backend utility

Module4

**callbacks** and **callback hell** in Node.js

## 🔁 What is a Callback in Node.js?

A **callback** is a function passed **as an argument** to another function to be **called later**, usually after some task finishes (especially async ones like reading a file or querying a DB).

### 🧠 Simple Example:

function greet(name, callback) {

console.log(`Hello, ${name}`);

callback();

}

function sayBye() {

console.log('Goodbye!');

}

greet('Alice', sayBye);

## ⚙️ Real-world Example with fs

const fs = require('fs');

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

if (err) return console.error(err);

console.log('File content:', data);

});

* readFile doesn’t block execution.
* When the file is ready, it **calls the callback** with (err, data).

## 😵 What is Callback Hell?

“**Callback hell**” is what happens when **callbacks are nested within callbacks**, forming a **pyramid of doom**.

### 🔥 Example of Callback Hell:

fs.readFile('file1.txt', 'utf8', (err, data1) => {

if (err) return;

fs.readFile('file2.txt', 'utf8', (err, data2) => {

if (err) return;

fs.writeFile('combined.txt', data1 + data2, err => {

if (err) return;

console.log('Files combined!');

});

});

});

* Hard to **read**, **debug**, and **maintain** 😩

## ✅ How to Fix Callback Hell?

### 1. **Modularize Functions**

Split each async task into a named function instead of nesting.

### 2. **Use Promises**

Convert callbacks into promises for cleaner syntax.

const fs = require('fs').promises;

async function combineFiles() {

try {

const data1 = await fs.readFile('file1.txt', 'utf8');

const data2 = await fs.readFile('file2.txt', 'utf8');

await fs.writeFile('combined.txt', data1 + data2);

console.log('Done!');

} catch (err) {

console.error(err);

}

}

combineFiles()

### 3. **Use Async/Await**

As seen above — clean, readable, and sequential.

## 🧠 Summary

| **Concept** | **Meaning** |
| --- | --- |
| Callback | A function executed after an async operation completes |
| Callback Hell | Nested callbacks leading to messy, unreadable code |
| Solutions | Use Promises, Async/Await, or named functions |

Error handling in Node.js is **super important**—especially when you're working with asynchronous code, file systems, or network requests. Let’s break it down and cover **how, where, and why** to handle errors properly.

**💥 Types of Errors in Node.js**

1. **Synchronous Errors** – thrown immediately
2. **Asynchronous Errors** – occur later (in callbacks, Promises, etc.)

**✅ 1. Handling Synchronous Errors**

Use try...catch:

function divide(a, b) {

try {

if (b === 0) throw new Error('Division by zero');

console.log(a / b);

} catch (err) {

console.error('Error:', err.message);

}

}

divide(10, 0);

**⏳ 2. Handling Errors in Callbacks (Async)**

Use the **error-first callback pattern**:

const fs = require('fs');

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

if (err) {

console.error('Failed to read file:', err.message);

return;

}

console.log('File content:', data);

});

* err is always the **first argument**
* Always **check err before continuing**

**🔄 3. Handling Errors with Promises**

const fs = require('fs').promises;

fs.readFile('file.txt', 'utf8')

.then(data => console.log('File:', data))

.catch(err => console.error('Error:', err.message));

**🧙 4. Error Handling with Async/Await**

Use try...catch:

const fs = require('fs').promises;

async function readFile() {

try {

const data = await fs.readFile('file.txt', 'utf8');

console.log('Data:', data);

} catch (err) {

console.error('Caught error:', err.message);

}

}

readFile()

**🚨 5. Global Error Handling (as a backup)**

process.on('uncaughtException', err => {

console.error('Uncaught Exception:', err);

});

process.on('unhandledRejection', reason => {

console.error('Unhandled Promise Rejection:', reason);

});

⚠️ **Note:** These are safety nets. You should still catch errors where they happen.

**💡 Best Practices**

✅ Always handle errors in callbacks  
✅ Wrap async logic in try...catch when using async/await  
✅ Log detailed error messages  
✅ Never ignore errors silently  
✅ Validate input before performing operations

H**ands-on Async File System project** in Node.js using the fs/promises module. We'll create a small file manager that can:

✅ Create files  
✅ Read files  
✅ Update files  
✅ Delete files  
✅ List all files in a folder

📁 **Project Name**: async-file-manager  
💡 Tech Used: Node.js + fs/promises  
🧩 Concept Focus: Async/Await, Error Handling, File System Operations

**🗂️ Folder Structure**

async-file-manager/

├── manager.js

└── files/

Make sure the files/ folder exists — this is where we’ll work.

**📜 manager.js**

const fs = require('fs').promises;

const path = require('path');

const folderPath = path.join(\_\_dirname, 'files');

// Create a new file

async function createFile(fileName, content) {

try {

const filePath = path.join(folderPath, fileName);

await fs.writeFile(filePath, content);

console.log(`✅ File '${fileName}' created.`);

} catch (err) {

console.error('❌ Error creating file:', err.message);

}

}

// Read a file

async function readFile(fileName) {

try {

const filePath = path.join(folderPath, fileName);

const data = await fs.readFile(filePath, 'utf8');

console.log(`📄 Contents of '${fileName}':\n${data}`);

} catch (err) {

console.error('❌ Error reading file:', err.message);

}

}

// Update (append to) a file

async function updateFile(fileName, extraContent) {

try {

const filePath = path.join(folderPath, fileName);

await fs.appendFile(filePath, extraContent);

console.log(`📝 '${fileName}' updated.`);

} catch (err) {

console.error('❌ Error updating file:', err.message);

}

}

// Delete a file

async function deleteFile(fileName) {

try {

const filePath = path.join(folderPath, fileName);

await fs.unlink(filePath);

console.log(`🗑️ '${fileName}' deleted.`);

} catch (err) {

console.error('❌ Error deleting file:', err.message);

}

}

// List all files

async function listFiles() {

try {

const files = await fs.readdir(folderPath);

console.log('📁 Files in folder:', files);

} catch (err) {

console.error('❌ Error listing files:', err.message);

}

}

// Example usage

(async () => {

await createFile('note.txt', 'This is a test note.\n');

await readFile('note.txt');

await updateFile('note.txt', 'Adding more content.\n');

await readFile('note.txt');

await listFiles();

await deleteFile('note.txt');

})();

**🧪 Run It**

node manager.js

You’ll see:

✅ File 'note.txt' created.

📄 Contents of 'note.txt': ...

📝 'note.txt' updated.

📁 Files in folder: [...]

🗑️ 'note.txt' deleted.

Module5

## 🌐 Introduction to the http Module in Node.js

The http module in Node.js is a **core module**, meaning you don't need to install anything — it's built-in. It's used to **create servers**, **handle requests**, and **send responses** over HTTP.

### ✅ How to Import It

const http = require('http');

### 🛠️ Creating a Basic HTTP Server

const http = require('http');

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

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

res.end('Hello, world!'); // Send response and end connection

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

#### 🧠 What’s happening here?

* http.createServer() creates a new server.
* It takes a **callback** with req (request) and res (response) objects.
* res.writeHead() sets the status code and headers.
* res.end() sends the response and ends the connection.
* server.listen() starts the server on a port (3000 in this case).

### 🔍 Accessing Request Info

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

console.log(req.method); // GET, POST, etc.

console.log(req.url); // Path like /home, /about

res.end('Check the console!');

});

### 🧪 Routing Example (Basic)

const http = require('http');

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

if (req.url === '/') {

res.end('Home Page');

} else if (req.url === '/about') {

res.end('About Page');

} else {

res.writeHead(404);

res.end('404 Not Found');

}

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

### 🧰 Common Use Cases

* Build **custom servers** from scratch.
* Understand the basics behind frameworks like **Express.js**.
* Create APIs, serve files, handle forms, etc.

### ⚠️ Note

While http is powerful, for **real-world apps** most developers use **Express.js**, which simplifies routing, middleware, and request parsing.

## Manual Routing with url Module in Node.js

The url module helps you parse the request URL (like path and query parameters), which is super useful when building routes manually without any framework.

### ✅ Step-by-Step Example

const http = require('http');

const url = require('url');

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

const parsedUrl = url.parse(req.url, true); // true = parse query as object

const path = parsedUrl.pathname;

const query = parsedUrl.query;

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

if (path === '/') {

res.end('Welcome to the Home Page');

} else if (path === '/about') {

res.end('About Page');

} else if (path === '/greet' && query.name) {

res.end(`Hello, ${query.name}!`);

} else {

res.end('404 - Page Not Found');

}

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

### 🧠 Breakdown

| **Code** | **What it Does** |
| --- | --- |
| url.parse(req.url) | Parses the URL of the request |
| .pathname | Gives the path (e.g., /about) |
| .query | Gives query params as object {} |
| Manual if/else | Used to simulate routing |

### 🔍 Try This in Browser:

* http://localhost:3000/ → Home Page
* http://localhost:3000/about → About Page
* http://localhost:3000/greet?name=Alice → Hello, Alice!
* http://localhost:3000/unknown → 404

### ⚠️ Why Use url?

Because req.url is just a raw string like /greet?name=Alice. The url module helps break that into:

{

pathname: '/greet',

query: { name: 'Alice' }

}

## Sending ****HTML**** and ****JSON**** Responses in Node.js

### 🔁 Setup First:

const http = require('http');

### 🧱 1. **Sending HTML as a Response**

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

if (req.url === '/html') {

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

res.end(`

<html>

<head><title>HTML Page</title></head>

<body>

<h1>Hello from Node.js</h1>

<p>This is a simple HTML response.</p>

</body>

</html>

`);

}

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000/html');

});

Content-Type: text/html tells the browser it's HTML.

### 🧩 2. **Sending JSON as a Response (for APIs)**

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

if (req.url === '/json') {

const data = {

name: 'Alice',

age: 25,

job: 'Developer'

};

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

res.end(JSON.stringify(data)); // Convert object to JSON string

}

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000/json');

});

Content-Type: application/json tells the client this is JSON.

### 🛠 Combine Both Routes:

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

if (req.url === '/html') {

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

res.end('<h1>Welcome to the HTML Page</h1>');

} else if (req.url === '/json') {

const data = { message: 'This is JSON' };

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

res.end(JSON.stringify(data));

} else {

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.end('404 - Not Found');

}

});

### ✅ Summary

| **Type** | **Content-Type** | **Use For** |
| --- | --- | --- |
| HTML | text/html | Rendering pages |
| JSON | application/json | APIs and frontend-backend comm |

**create a basic website server in Node.js** using the built-in http and fs modules. We'll serve actual **HTML files** like a real website.

## 🛠️ Project Structure

basic-node-server/

├── index.html

├── about.html

└── server.js

### 📄 1. index.html

<!DOCTYPE html>

<html>

<head>

<title>Home</title>

</head>

<body>

<h1>Welcome to My Node.js Website</h1>

<p>This is the home page.</p>

</body>

</html>

### 📄 2. about.html

<!DOCTYPE html>

<html>

<head>

<title>About</title>

</head>

<body>

<h1>About Us</h1>

<p>This is the about page.</p>

</body>

</html>

### 📜 3. server.js

const http = require('http');

const fs = require('fs');

const path = require('path');

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

let filePath = './index.html';

if (req.url === '/about') {

filePath = './about.html';

} else if (req.url !== '/') {

res.writeHead(404, { 'Content-Type': 'text/html' });

return res.end('<h1>404 - Page Not Found</h1>');

}

fs.readFile(filePath, (err, data) => {

if (err) {

res.writeHead(500);

return res.end('Server error');

}

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

res.end(data);

});

});

server.listen(3000, () => {

console.log('🌐 Server is running at http://localhost:3000');

});

### ▶️ Run the Server

1. Open terminal
2. Navigate to the folder: cd basic-node-server
3. Run:

node server.js

1. Open browser:
   * http://localhost:3000 → Home Page
   * http://localhost:3000/about → About Page
   * http://localhost:3000/something → 404

### ✅ What You Learned

* Creating a basic server with http
* Reading HTML files with fs
* Serving pages based on routes
* Sending 404 and 500 responses

**read and write JSON files** in Node.js using the built-in fs module — a must-know skill for working with data in local files (like for configs, small databases, logs, etc.).

**📁 Example File: data.json**

Create a file named data.json with this content:

[

{ "id": 1, "name": "Alice" },

{ "id": 2, "name": "Bob" }

]

**🧠 1. Reading JSON File in Node.js**

const fs = require('fs');

fs.readFile('data.json', 'utf8', (err, jsonData) => {

if (err) {

console.error('Error reading file:', err);

return;

}

const data = JSON.parse(jsonData); // Convert JSON string to JS object

console.log('Data read from file:', data);

});

✅ JSON.parse() turns the JSON string into an object/array.

**🖊️ 2. Writing JSON File in Node.js**

const fs = require('fs');

const newData = [

{ id: 1, name: "Alice" },

{ id: 2, name: "Bob" },

{ id: 3, name: "Charlie" }

];

fs.writeFile('data.json', JSON.stringify(newData, null, 2), 'utf8', err => {

if (err) {

console.error('Error writing file:', err);

return;

}

console.log('Data written successfully!');

});

✅ JSON.stringify() converts JavaScript object to a JSON string.

* The null, 2 makes it pretty-formatted.

**🔄 3. Append Data (Read → Update → Write)**

const fs = require('fs');

fs.readFile('data.json', 'utf8', (err, jsonData) => {

if (err) throw err;

const data = JSON.parse(jsonData);

data.push({ id: 4, name: "David" });

fs.writeFile('data.json', JSON.stringify(data, null, 2), 'utf8', err => {

if (err) throw err;

console.log('New user added!');

});

});

**🔐 Pro Tips**

* Always use 'utf8' to avoid buffer issues.
* Use try/catch if working with synchronous methods (fs.readFileSync).
* Validate the JSON before writing.

**creating and modifying JSON data dynamically** in Node.js — meaning we'll interact with data.json at runtime: **add, update, and delete records**, just like in a real CRUD app.

## 📁 Setup: data.json

Start with this file:

[

{ "id": 1, "name": "Alice" },

{ "id": 2, "name": "Bob" }

]

## ✅ Create (Add New Data Dynamically)

const fs = require('fs');

function addUser(newUser) {

fs.readFile('data.json', 'utf8', (err, jsonData) => {

if (err) throw err;

const data = JSON.parse(jsonData);

data.push(newUser);

fs.writeFile('data.json', JSON.stringify(data, null, 2), err => {

if (err) throw err;

console.log('User added:', newUser);

});

});

}

// Example

addUser({ id: 3, name: 'Charlie' });

## ✏️ Update (Modify Existing Data)

function updateUser(id, updatedData) {

fs.readFile('data.json', 'utf8', (err, jsonData) => {

if (err) throw err;

const data = JSON.parse(jsonData);

const index = data.findIndex(user => user.id === id);

if (index === -1) {

console.log('User not found');

return;

}

data[index] = { ...data[index], ...updatedData };

fs.writeFile('data.json', JSON.stringify(data, null, 2), err => {

if (err) throw err;

console.log(`User with id ${id} updated.`);

});

});

}

// Example

updateUser(2, { name: 'Robert' });

## ❌ Delete (Remove Data Dynamically)

function deleteUser(id) {

fs.readFile('data.json', 'utf8', (err, jsonData) => {

if (err) throw err;

let data = JSON.parse(jsonData);

data = data.filter(user => user.id !== id);

fs.writeFile('data.json', JSON.stringify(data, null, 2), err => {

if (err) throw err;

console.log(`User with id ${id} deleted.`);

});

});

}

// Example

deleteUser(1);

## 🔁 Combine All (Mini CLI-style Control)

You can structure it like this for modular use:

// addUser({ id: 4, name: 'Diana' });

// updateUser(3, { name: 'Charles' });

// deleteUser(2);

## 🧠 Summary

| **Operation** | **Function** |
| --- | --- |
| Create | push() + write |
| Update | findIndex() + merge |
| Delete | filter() + write |

🔥 **mini RESTful API in Node.js** using only the **http module** — no Express, no external frameworks. This will handle the **basic CRUD operations**: GET, POST, PUT, DELETE.

## 🧱 Project Setup

mini-rest-api/

├── data.json

└── server.js

### 📁 data.json – Sample Data

[

{ "id": 1, "name": "Alice" },

{ "id": 2, "name": "Bob" }

]

### 📜 server.js

const http = require('http');

const fs = require('fs');

const url = require('url');

const PORT = 3000;

const FILE = 'data.json';

const readData = () => {

return JSON.parse(fs.readFileSync(FILE, 'utf8'));

};

const writeData = (data) => {

fs.writeFileSync(FILE, JSON.stringify(data, null, 2));

};

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

const parsedUrl = url.parse(req.url, true);

const id = parseInt(parsedUrl.query.id);

const method = req.method;

const pathname = parsedUrl.pathname;

res.setHeader('Content-Type', 'application/json');

// GET: Fetch all or one

if (method === 'GET' && pathname === '/users') {

const data = readData();

if (id) {

const user = data.find(u => u.id === id);

res.end(JSON.stringify(user || { message: 'User not found' }));

} else {

res.end(JSON.stringify(data));

}

}

// POST: Add user

else if (method === 'POST' && pathname === '/users') {

let body = '';

req.on('data', chunk => (body += chunk));

req.on('end', () => {

const newUser = JSON.parse(body);

const data = readData();

newUser.id = data.length ? data[data.length - 1].id + 1 : 1;

data.push(newUser);

writeData(data);

res.end(JSON.stringify({ message: 'User added', user: newUser }));

});

}

// PUT: Update user

else if (method === 'PUT' && pathname === '/users') {

if (!id) return res.end(JSON.stringify({ message: 'User ID required' }));

let body = '';

req.on('data', chunk => (body += chunk));

req.on('end', () => {

const updates = JSON.parse(body);

const data = readData();

const index = data.findIndex(u => u.id === id);

if (index === -1) return res.end(JSON.stringify({ message: 'User not found' }));

data[index] = { ...data[index], ...updates };

writeData(data);

res.end(JSON.stringify({ message: 'User updated', user: data[index] }));

});

}

// DELETE: Remove user

else if (method === 'DELETE' && pathname === '/users') {

if (!id) return res.end(JSON.stringify({ message: 'User ID required' }));

const data = readData();

const newData = data.filter(u => u.id !== id);

if (data.length === newData.length)

return res.end(JSON.stringify({ message: 'User not found' }));

writeData(newData);

res.end(JSON.stringify({ message: 'User deleted' }));

}

// Fallback

else {

res.writeHead(404);

res.end(JSON.stringify({ message: 'Route not found' }));

}

});

server.listen(PORT, () => {

console.log(`🚀 Server running at http://localhost:${PORT}`);

});

## 🧪 How to Test

You can test using tools like **Postman**, **Insomnia**, or curl.

### 🟢 GET all users

GET http://localhost:3000/users

### 🔍 GET user by ID

GET http://localhost:3000/users?id=2

### ➕ POST new user

POST http://localhost:3000/users

Body (JSON): { "name": "Charlie" }

### ✏️ PUT update user

PUT http://localhost:3000/users?id=1

Body (JSON): { "name": "Alice Updated" }

### ❌ DELETE user

DELETE http://localhost:3000/users?id=2

## 🧠 Key Concepts Covered

* http.createServer() for REST APIs
* Handling req.method and req.url manually
* Reading and writing JSON files
* Parsing body data with req.on('data')
* Dynamic ID handling