Node.js for Everyone
Clear Steps and Practical Examples

By Ryan Nastaran

Version 2.1.1

Contents

Introduction to Node.js

What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment. It allows you to run JavaScript code on the server-side, outside of a web browser. Node.js uses the V8 JavaScript engine (the same engine used by Google Chrome) to execute code.
Node.js is designed to be **asynchronous** and **non-blocking**. This means it can handle multiple operations at the same time without waiting for one to finish before starting another. This is achieved through event-driven architecture.

What is npm?

npm stands for **Node Package Manager**. It is a package manager for JavaScript and is the default package manager for Node.js. npm allows you to install, share, and manage code packages written in JavaScript. These packages can be **libraries**, **tools**, or **frameworks** that you can use in your Node.js projects.

Installing Node.js and npm

Open your web browser and go to official Node.js website. On the Node.js homepage, you will see two versions: LTS (Long Term Support) and the Current version. It is recommended to download the LTS version for most users. Click on the LTS version download button. The website should automatically detect your operating system and provide the appropriate installer (e.g., .msi for Windows, .pkg for macOS, or a binary for Linux). Run the installer and follow the instructions.

Note: The Node.js installer includes npm (Node Package Manager). When you install Node.js, npm is automatically installed as well.

Verify the Installation

On **Windows**: You can use Command Prompt or PowerShell. Open it by searching for "cmd" or "PowerShell" in the Start menu and selecting it.
On **macOS** or **Linux**: Open the Terminal application from your "Applications" or "Utilities folder".

run **node -v** and **npm -v** to check the installed versions of **Node.js** and **npm**.

Input	Output
node -v	v20.13.1
npm -v	10.5.2

Node.js REPL

REPL stands for **Read-Eval-Print Loop**. It's an interactive shell that allows you to execute JavaScript code one line at a time.
For start the REPL open your terminal. Type **node** and press Enter. You should see a prompt (>) where you can start typing JavaScript code.

Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a popular open-source code editor developed by Microsoft. It is widely used by developers for various programming languages and comes with many powerful features that make coding easier and more efficient. For download go to the official Visual Studio Code website.

First Project (Hello World)

Let's build our first project in Node.js: Hello World" in the following way:

1. Open your code editor (like Visual Studio Code).
2. Create a new file and name it hello.js.
3. Write following code:

```
console.log("Hello, Node.js!");
```

4. Save the hello.js file.
5. Open your terminal navigate to the directory where your hello.js file is located.
6. Run the following command:

```
node hello.js
```

7. You should see the output: **Hello, Node.js!**

Note: The node command is used to run Node.js scripts. It allows you to execute JavaScript code outside of a web browser, directly in your operating system's terminal or command prompt.

Modules in Node.js

Modules are reusable pieces of code that you can use to organize your Node.js applications. They help you break down your application into smaller, manageable parts, making your code more modular, maintainable, and reusable.
In Node.js, there are three types of modules:

- Core (built-in) modules,
- Local (custom) modules,
- Third-party modules.

Core (built-in) modules

These are modules that come with Node.js by default. You don't need to install them separately. Some common core modules are:

1. **fs (file system):**
   allows you to interact with the file system on your computer. It provides functions to read, write, delete, and manipulate files and directories.
2. **path:**
   The path module in Node.js provides utilities for working with file and directory paths.
3. **http:**
   The http module in Node.js allows you to create an **HTTP server** that can handle **requests** and send **responses** (create and manage HTTP servers and clients).
4. **os:**
   The os module in Node.js provides several operating system-related utility methods. It allows you to interact with the **operating system** to retrieve information like the **system's hostname**, **platform**, **memory usage**, and more.

Local (custom) modules

These are modules that you create yourself. You can create custom modules to organize your code better and make it reusable across different parts of your application.

```
//Create a file named math.js:

// math.js
function add(a, b) {
  return a + b;
}
function subtract(a, b) {
  return a - b;
}
module.exports = {
  add,
  subtract,
};
```

```
//Use the custom module in another file:

// app.js
const math = require("./math");

const sum = math.add(5, 3);
const difference = math.subtract(5, 3);

console.log(`Sum: ${sum}`); // Sum: 8
console.log(`Difference: ${difference}`); // Difference: 2
```

Third-party modules (using npm)

These are modules created by the community that you can install and use in your project. npm (Node Package Manager) is the default package manager for Node.js and allows you to install, update, and manage third-party modules.

Common core modules

FS (File System)

allows you to interact with the file system on your computer. It provides functions to read, write, delete, and manipulate files and directories.
To use the fs module, you need to **import it into your Node.js script** with const fs = require("fs");

Read a file

You can **read files** using various methods provided by the fs module. The **callback function** receives two arguments: an error object (if an error occurred) and the file content. **readFileSync** is the synchronous version of **readFile**. It blocks the event loop until the file is read.

```
//Asynchronous File Reading

fs.readFile("example.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err);
    return;
  }
  console.log("File contents:", data);
});
```

```
//Synchronous File Reading

try {
  const data = fs.readFileSync("example.txt", "utf8");
  console.log("File contents:", data);
} catch (err) {
  console.error("Error reading file:", err);
}
```

Write a file

You can **write to files** using the fs module. The callback function receives an error object if an error occurred. **writeFileSync** is the synchronous version of **writeFile**. It blocks the event loop until the file is written.

```
//Asynchronous File Writing

const content = "Hello, Node.js!";
fs.writeFile("example.txt", content, "utf8", (err) => {
  if (err) {
    console.error("Error writing file:", err);
    return;
  }
  console.log("File has been written");
});
```

```
//Synchronous File Writing

const content = "Hello, Node.js!";
try {
  fs.writeFileSync("example.txt", content, "utf8");
  console.log("File has been written");
} catch (err) {
  console.error("Error writing file:", err);
}
```

Append data

You can **append data to an existing file** using **fs.appendFile** or **fs.appendFileSync**.

```
//Asynchronous Appending data to an existing file

const content = " This is appended text.";
fs.appendFile("example.txt", content, "utf8", (err) => {
  if (err) {
    console.error("Error appending file:", err);
    return;
  }
  console.log("Content appended");
});
```

```
//Synchronous Appending data to an existing file

const content = " This is appended text.";

try {
  fs.appendFileSync("example.txt", content, "utf8");
  console.log("Content appended");
} catch (err) {
  console.error("Error appending file:", err);
}
```

Buffers

Buffers in Node.js are a way to handle **binary data** directly. They are particularly useful when working with streams, file systems, or network communications where data is often **received** or **sent** in **raw binary format**.

What is a Buffer?
A buffer is a **temporary storage area** for binary data. In Node.js, the Buffer class is a **global** type for dealing with binary data directly.
Buffers are like arrays of integers but correspond to raw memory allocations outside the V8 heap. Each element in a buffer is an 8-bit integer. Buffers are used to handle raw binary data that might come from a file, network, or other streams.

Create a Buffer

1. **From a String:** each character is converted to its binary equivalent.

```
const buffer = Buffer.from("Hello, World!");
console.log(buffer);
// Output: <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21>
```

2. **Allocating a Buffer:** Create a buffer of length x, initialized with zeros

```
const buffer = Buffer.alloc(10); // Create a buffer of length 10, initialized with zeros
console.log(buffer);
// Output: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

3. **Allocating an Unsafe Buffer:** Create a buffer of length 10, uninitialized

```
const buffer = Buffer.allocUnsafe(10); // Create a buffer of length 10, uninitialized
console.log(buffer);
// Output: <Buffer xx xx xx xx xx xx xx xx xx xx>
```

Read and Write to a Buffer

```
//Writing to a Buffer

const buffer = Buffer.alloc(10);
buffer.write("Hello");
console.log(buffer);
// Output: <Buffer 48 65 6c 6c 6f 00 00 00 00 00>
console.log(buffer.toString());
// Output: 'Hello'
```

```
//Reading from a Buffer

const buffer = Buffer.from('Hello, World!');
console.log(buffer.toString());
// Output: 'Hello, World!'
```

Manipulate a Buffer

```
//Slice a Buffer

const buffer = Buffer.from('Hello, World!');
const slice = buffer.slice(0, 5);
console.log(slice.toString());
// Output: 'Hello'
```

```
//Copying Buffers

const buffer1 = Buffer.from('Hello, World!');
const buffer2 = Buffer.alloc(5);
buffer1.copy(buffer2, 0, 0, 5);
console.log(buffer2.toString());
// Output: 'Hello'
```

Buffer Methods

1. **Buffer.byteLength():** Returns the length of the buffer in bytes.

```
const str = "Hello, World!";
const length = Buffer.byteLength(str);
console.log(length);
// Output: 13
```

2. **Buffer.concat():** Concatenates multiple buffers.

```
const buffer1 = Buffer.from("Hello, ");
const buffer2 = Buffer.from("World!");
const buffer3 = Buffer.concat([buffer1, buffer2]);
console.log(buffer3.toString());
// Output: 'Hello, World!'
```

3. **buffer.equals():** Compares two buffers.

```
const buffer1 = Buffer.from("Hello");
const buffer2 = Buffer.from("Hello");
const buffer3 = Buffer.from("World");
console.log(buffer1.equals(buffer2)); // true
console.log(buffer1.equals(buffer3)); // false
```

Streams

Streams are a fundamental concept in Node.js for handling I/O operations efficiently. They allow you to read or write data sequentially in small chunks, rather than loading all data into memory at once. This makes streams particularly useful for working with large files or data from network sources.

Types of Streams

1. **Readable Streams:** Used for reading data in chunks.
2. **Writable Streams:** Used for writing data in chunks.
3. **Duplex Streams:** Can read and write data.
4. **Transform Streams:** A type of duplex stream where the output is computed based on the input.
5. **Piping:** Directly transfers data from one stream to another.

Streams allow efficient handling of data, especially for large amounts of data, by processing it in chunks and keeping memory usage low.

Readable Streams

Readable streams are used to read data. Examples include reading data from a file, an HTTP request.

```
const fs = require("fs");

// Create a readable stream
const readableStream = fs.createReadStream("example.txt", { encoding: "utf8" });

// Handle 'data' event to read chunks of data
readableStream.on("data", (chunk) => {
  console.log("Received chunk:", chunk);
});

// Handle 'end' event when the stream is finished
readableStream.on("end", () => {
  console.log("End of file.");
});

// Handle 'error' event for errors
readableStream.on("error", (err) => {
  console.error("Error:", err);
});
```

Writable Streams

Writable streams are used to write data. Examples include writing data to a file or sending data in an HTTP response.

```
const fs = require("fs");

// Create a writable stream
const writableStream = fs.createWriteStream("output.txt");

// Write data to the stream
writableStream.write("Hello, World!\n");
writableStream.write("This is a writable stream example.");
writableStream.end(); // Close the stream

// Handle 'finish' event when writing is finished
writableStream.on("finish", () => {
  console.log("Finished writing to file.");
});

// Handle 'error' event for errors
writableStream.on("error", (err) => {
  console.error("Error:", err);
});
```

Duplex Streams

Duplex streams are both readable and writable. Examples include TCP sockets and zlib streams for compression.

```
const net = require("net");

// Create a duplex stream (TCP server)
const server = net.createServer((socket) => {
  socket.write("Hello from server!\n"); // Write to client

  socket.on("data", (data) => {
    console.log("Received from client:", data.toString()); // Read from client
  });
});

server.listen(8080, () => {
  console.log("Server listening on port 8080");
});
```

Transform Streams

Transform streams are a type of duplex stream where the output is computed based on the input. A common example is compression/decompression or encryption/decryption.

```
const fs = require("fs");
const zlib = require("zlib");

// Create a readable stream
const readableStream = fs.createReadStream("input.txt");

// Create a writable stream
const writableStream = fs.createWriteStream("output.txt.gz");

// Create a transform stream (gzip)
const gzip = zlib.createGzip();

// Pipe the readable stream into the gzip stream and then into the writable stream
readableStream.pipe(gzip).pipe(writableStream);

// Handle 'finish' event when the process is complete
writableStream.on("finish", () => {
  console.log("File successfully compressed.");
});
```

Piping Streams

Piping is a mechanism to connect the output of one stream directly to the input of another stream. This is commonly used for streaming data from a readable stream to a writable stream.

```
const fs = require("fs");

// Create a readable stream
const readableStream = fs.createReadStream("input.txt");

// Create a writable stream
const writableStream = fs.createWriteStream("output.txt");

// Pipe the readable stream into the writable stream
readableStream.pipe(writableStream);

writableStream.on("finish", () => {
  console.log("Piping complete.");
});
```

Delete a file

You can delete files using fs.unlink or fs.unlinkSync.

```
//Asynchronous File Delete

fs.unlink("example.txt", (err) => {
  if (err) {
    console.error("Error deleting file:", err);
    return;
  }
  console.log("File deleted");
});
```

```
//Synchronous File Delete

try {
  fs.unlinkSync("example.txt");
  console.log("File deleted");
} catch (err) {
  console.error("Error deleting file:", err);
}
```

Create and delete a directory

The fs module also allows you to **create** and **delete** directories.

```
//Create Directory

fs.mkdir("newDir", (err) => {
  if (err) {
    console.error("Error creating directory:", err);
    return;
  }
  console.log("Directory created");
});
```

```
//Delete Directory

fs.rmdir("newDir", (err) => {
  if (err) {
    console.error("Error deleting directory:", err);
    return;
  }
  console.log("Directory deleted");
});
```

Read the files of the directory

The fs module also allows you to **read the contents of the directory**.

```
fs.readdir(".", (err, files) => {
  if (err) {
    console.error("Error reading directory:", err);
    return;
  }
  console.log("Directory contents:", files);
});
```

Check Status (file or directory)

You can check the status of a **file** or **directory** using **fs.stat** or **fs.statSync**.
stats.isDirectory(): Method to check if the given path is a directory.
readdirSync: Method to read the contents of the directory synchronously after confirming it is a directory.

```
//Asynchronous File Status

fs.stat("example.txt", (err, stats) => {
  if (err) {
    console.error("Error getting file stats:", err);
    return;
  }
```

```
//Synchronous File Status

try {
  const stats = fs.statSync("example.txt");
  console.log("File stats:", stats);
} catch (err) {
  console.error("Error getting file stats:", err);
```

```
  console.log("File stats:", stats);
});

// Checking Directory Status and Contents Asynchronously

fs.stat('myDirectory', (err, stats) => {
  if (err) {
    console.error('Error getting directory stats:', err);
    return;
  }

  if (stats.isDirectory()) {
    console.log('This is a directory');
    fs.readdir('myDirectory', (err, files) => {
      if (err) {
        console.error('Error reading directory:', err);
        return;
      }
      console.log('Directory contents:', files);
    });
  } else {
    console.log('This is not a directory');
  }
}


// Checking Directory Status and Contents Synchronously

try {
  const stats = fs.statSync("myDirectory");

  if (stats.isDirectory()) {
    console.log("This is a directory");
    const files = fs.readdirSync("myDirectory");
    console.log("Directory contents:", files);
  } else {
    console.log("This is not a directory");
  }
} catch (err) {
  console.error("Error getting directory stats:", err);
}
```

Path

The path module in Node.js provides utilities for working with **file** and **directory paths**. First, you need to import the path module:

```
const path = require("path");
```

Get the last part of a path (the filename)

```
//Code

const filePath = "/users/admin/docs/file.txt";
const baseName = path.basename(filePath);
console.log(baseName);

// To get the filename without the extension
const baseNameWithoutExt = path.basename(filePath, ".txt");
console.log(baseNameWithoutExt);
```

```
// Outputs

file.txt
file
```

Get the directory name of a path

```
//Code
const filePath = "/users/admin/docs/file.txt";
const dirName = path.dirname(filePath);
console.log(dirName);
```

```
// Outputs

/users/admin/docs
```

Get the file extension of a path

```
//Code

const filePath = "/users/admin/docs/file.txt";
const extName = path.extname(filePath);
console.log(extName);
```

```
// Outputs

.txt
```

Join multiple path segments into one path

```
//Code

const joinedPath = path.join("/users", "admin", "docs",
"file.txt");
console.log(joinedPath);
```

```
// Outputs

/users/admin/docs/file.txt
```

Resolve a sequence of paths into an absolute path

```
//Code

const absolutePath = path.resolve("users", "admin", "docs",
"file.txt");
console.log(absolutePath);
```

```
// Outputs

an absolute path, e.g., /home/user/users/admin/docs/file.txt on
Unix-like systems
```

Normalize a path by resolving ".." and "." segments

```
//Code

const messyPath = "/users/admin/../docs/./file.txt";
const normalizedPath = path.normalize(messyPath);
console.log(normalizedPath);
```

```
// Outputs

/users/docs/file.txt
```

The path.normalize() method is useful for cleaning up paths by resolving ".." and "." segments and reducing multiple slashes to a single slash. This ensures that paths are consistent and easier to work with. For example,

- /home/user/.. moves up one directory to /home.
- /projects/./ becomes /projects/ (current directory . is removed).
- project1/../ moves up one directory to projects.
- project2// is reduced to project2/.

Return an object with properties representing different parts of a path

```
//Code

const filePath = "/users/admin/docs/file.txt";
const parsedPath = path.parse(filePath);
console.log(parsedPath);
```

```
// Outputs

{
  root: '/',
  dir: '/users/admin/docs',
  base: 'file.txt',
  ext: '.txt',
  name: 'file'
}
```

Return a path string from an object

```
//Code

const pathObject = {
  root: "/",
  dir: "/users/admin/docs",
  base: "file.txt",
};
const formattedPath = path.format(pathObject);
console.log(formattedPath);
```

```
// Outputs

/users/admin/docs/file.txt
```

Http

The http module in Node.js allows you to create an **HTTP server** that can handle **requests** and send **responses** (create and manage HTTP servers and clients).

Create an HTTP Server

1. You need to import the **http** module using the **Require** function.
2. Create an HTTP Server: Use the "**http.createServer**" method to create an HTTP server. **This method takes a callback function that gets executed every time a request is made to the server.**
   - req: This is the **request object**. It contains information about the HTTP request.
   - res: This is the **response object**. You use it to send a response back to the client.

3. Make the Server Listen on a Specific Port: You need to specify the port and hostname for your server to listen to incoming requests.

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

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
 res.statusCode = 200;
 res.setHeader("Content-Type", "text/plain");
 res.end("Hello, World!\n");
});

server.listen(port, hostname, () => {
 console.log(`Server running at http://${hostname}:${port}/`);
});
```

Different Routes

Routes define the endpoints (URLs) that your application responds to. Each route can be associated with a different function, handling specific requests. In a basic Node.js HTTP server, **routes determine what content or functionality is provided when a user accesses a specific path**.

In Node.js, routes are managed using the http module. Here's a deeper dive into how you can handle different routes.

1. Create the HTTP Server: As before, you start by creating an HTTP server.
2. Check the Request URL: Use req.url to determine the path that the client has requested.
3. Respond Based on the Path: Send different responses based on the requested path.

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

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
 // Set the response header
 res.setHeader("Content-Type", "text/plain");

 // Check the URL of the request
 if (req.url === "/") {
  // Home route
  res.statusCode = 200;
  res.end("Welcome to the Home Page!\n");
 } else if (req.url === "/about") {
  // About route
  res.statusCode = 200;
  res.end("Welcome to the About Page!\n");
 } else {
  // Any other route
  res.statusCode = 404;
  res.end("Page Not Found\n");
 }
});

server.listen(port, hostname, () => {
 console.log(`Server running at http://${hostname}:${port}/`);
});
```

Send HTML Response

When a web server sends an HTML response to a client (typically a web browser), it means the server is sending HTML code that the browser will render as a webpage. It consists of elements like headings, paragraphs, links, images, etc.
Key Components of an HTML Response:

- **HTTP Status Code:** Indicates the result of the HTTP request. Common status codes include:
  - 200 OK: The request was **successful**.
  - 404 Not Found: The requested resource could **not be found**.
  - 500 Internal Server Error: The server encountered an **error**.
- **HTTP Headers:** Provide metadata about the response. Important headers include:
  - Content-Type: Specifies the media type of the resource. For HTML, it's **text/html**.
  - Content-Length: Indicates the **size of the response body** in **bytes**.
- **Response Body:** The actual content that is returned to the client, which in this case is HTML code.

Here's a detailed explanation of how to send HTML Responses in Node.js:

1. Import the http Module
2. Create an HTTP Server
3. Set the Content-Type Header (set the Content-Type header to **text/html** to indicate that the response will be **HTML**)
4. Write HTML Content to the Response Body
5. Handle Different Routes
6. Make the Server Listen on a Specific Port

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

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
 res.setHeader("Content-Type", "text/html");

 if (req.url === "/") {
  res.statusCode = 200;
  res.end("<h1>Welcome to the Home Page!</h1>\n");
 } else if (req.url === "/about") {
  res.statusCode = 200;
  res.end("<h1>Welcome to the About Page!</h1>\n");
 } else {
  res.statusCode = 404;
  res.end("<h1>Page Not Found</h1>\n");
 }
});

server.listen(port, hostname, () => {
 console.log(`Server running at http://${hostname}:${port}/`);
});
```

Send JSON Response

Handling JSON responses is a common requirement in web development, especially when working with APIs. JSON (JavaScript Object Notation) is a lightweight data interchange format that's easy for humans to read and write and easy for machines to parse and generate. Here's a detailed breakdown of how to Send JSON responses using the http module in Node.js:

1. **Set Content-Type Header to application/json**
   This tells the client that the response body contains JSON data.
2. **Create a JavaScript Object**
   This object will be converted into a JSON string.
3. **Convert the Object to a JSON String**
   Use JSON.stringify to convert the JavaScript object to a JSON string.
4. **Send the JSON String as a Response**
   Use the res.end method to send the JSON string as the response.

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

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
 // Set the Content-Type to application/json
 res.setHeader("Content-Type", "application/json");

 // Define responses based on the request URL
 if (req.url === "/") {
  // Create a JavaScript object for the home page response
  const homePageResponse = {
   message: "Welcome to the Home Page!",
   status: 200,
  };

  // Convert the JavaScript object to a JSON string
  const jsonResponse = JSON.stringify(homePageResponse);

  // Send the JSON string as a response
  res.statusCode = 200;
  res.end(jsonResponse);
 } else if (req.url === "/about") {
  // Create a JavaScript object for the about page response
  const aboutPageResponse = {
   message: "Welcome to the About Page!",
   status: 200,
  };

  // Convert the JavaScript object to a JSON string
  const jsonResponse = JSON.stringify(aboutPageResponse);
```

```
    // Send the JSON string as a response
    res.statusCode = 200;
    res.end(jsonResponse);
  } else {
    // Create a JavaScript object for the 404 response
    const notFoundResponse = {
      error: "Page Not Found",
      status: 404,
    };

    // Convert the JavaScript object to a JSON string
    const jsonResponse = JSON.stringify(notFoundResponse);

    // Send the JSON string as a response
    res.statusCode = 404;
    res.end(jsonResponse);
  }
});

// Start the server and listen on the specified port and hostname
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

In brief Set the **Content-Type** to **application/json**, convert JavaScript objects to JSON strings using **JSON.stringify**, and send the JSON string as the response.

Receive JSON Response

To receive JSON data in a POST request, you need to:

1. Read the incoming data chunks.
2. Concatenate the data chunks to form the complete data string.
3. Parse the data string into a JavaScript object using JSON.parse.

"Collect data chunks" and **"Process the complete data"** are two sequential steps in handling incoming data in a Node.js server. First, during the "Collect data chunks" step, the server listens for data events, accumulating the small pieces of incoming data into a single string. This step ensures all parts of the data are gathered. Once all data chunks are received, the "Process the complete data" step begins, triggered by the end event. Here, the complete data string is processed, typically by parsing it from JSON format into a JavaScript object. The order is crucial: **you must collect all chunks before you can accurately process the complete data**. The purpose of collecting chunks is to ensure you have the full data set before converting or using it, ensuring data integrity and accurate processing.

Collect Data Chunks:

When a client sends data to a server (e.g., in a POST request), the data is often sent in small pieces called chunks. These chunks need to be collected and combined into one complete string before processing. Listen for data events on the request object:

1. Each chunk of data received triggers a data event.
2. Append each chunk to a variable (typically a string).

```
let body = "";

req.on("data", (chunk) => {
  body += chunk.toString();
});
```

Process the Complete Data:

Once all data chunks have been received, a final end event is emitted. At this point, you can process the complete data string. Listen for the end event on the request object:

1. This event signifies that all data chunks have been received.
2. Parse the complete data string (e.g., convert it from JSON format to a JavaScript object).

```
req.on("end", () => {
  try {
    const parsedData = JSON.parse(body);
    // Now you can use parsedData
  } catch (error) {
    // Handle JSON parsing error
  }
});
```

Here's a full example demonstrating both steps together in a Node.js server handling JSON data from a POST request:

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

const hostname = "127.0.0.1";
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.method === "POST" && req.url === "/data") {
```

```
    let body = "";

    // Step 1: Collect data chunks
    req.on("data", (chunk) => {
      body += chunk.toString(); // Convert Buffer to string and append to body
    });

    // Step 2: Process the complete data
    req.on("end", () => {
      try {
        const parsedData = JSON.parse(body); // Parse JSON string to JavaScript object
        res.statusCode = 200;
        res.setHeader("Content-Type", "application/json");
        res.end(JSON.stringify({ receivedData: parsedData })); // Send response back
      } catch (error) {
        res.statusCode = 400;
        res.setHeader("Content-Type", "application/json");
        res.end(JSON.stringify({ error: "Invalid JSON" })); // Handle JSON parsing error
      }
    });
  } else {
    res.statusCode = 404;
    res.end("Page Not Found\n");
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

By separating these two steps, you ensure that you handle incoming data correctly and efficiently in your Node.js server.

OS

The os module in Node.js provides several operating system-related utility methods. It allows you to interact with the **operating system** to retrieve information like the **system's hostname**, **platform**, **memory usage**, and more. You need to import the **os** module using the Require function.

Get the OS Platform

Returns a string identifying the operating system platform.

```
const os = require("os");
console.log(`Platform: ${os.platform()}`); // Output: 'win32', 'linux', 'darwin', etc.
```

Get the OS CPU Architecture

Returns a string identifying the operating system CPU architecture.

```
const os = require("os");
console.log(`Architecture: ${os.arch()}`); // Output: 'x64', 'arm', 'ia32', etc.
```

Get the Hostname

Returns the hostname of the operating system.

```
const os = require("os");
console.log(`Hostname: ${os.hostname()}`); // Output: Hostname of the machine
```

Get Network Interfaces

Returns an object containing network interfaces that have been assigned a network address.

```
const os = require("os");
console.log(`Network Interfaces: ${JSON.stringify(os.networkInterfaces(), null, 2)}`);
```

Get Total Memory

Returns the total amount of system memory in bytes.

```
const os = require("os");
console.log(`Total Memory: ${os.totalmem() / (1024 * 1024)} MB`); // Output in Megabytes
```

Get Free Memory

Returns the amount of free system memory in bytes.

```
const os = require("os");
console.log(`Free Memory: ${os.freemem() / (1024 * 1024)} MB`); // Output in Megabytes
```

Get System Uptime

Returns the system uptime in seconds.

```
const os = require("os");
console.log(`System Uptime: ${os.uptime()} seconds`);
```

Get User Information

Returns information about the current user.

```
const os = require("os");
console.log(`User Info: ${JSON.stringify(os.userInfo(), null, 2)}`);
```

Get the Home Directory

Returns the home directory of the current user.

```
const os = require("os");
console.log(`Home Directory: ${os.homedir()}`);
```

Get the Temporary Directory

Returns the default directory for temporary files.

```
const os = require("os");
console.log(`Temporary Directory: ${os.tmpdir()}`);
```

Event-Driven Programming

In Node.js, much of the programming involves **handling events**. Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as **user actions** (mouse clicks, key presses), **sensor outputs**, or **messages** from other **programs/threads**. In Node.js, this paradigm is essential due to its **asynchronous** nature.

There are 3 Key Components of Event-Driven Programming in Node.js:

1. Event Loop
2. EventEmitter Class
3. Callbacks

Event Loop

The **event loop** is the **core** of Node.js's **asynchronous behavior**. It allows Node.js to perform non-blocking I/O operations by offloading operations to the system kernel whenever possible. The event loop is a key concept in Node.js that allows it to **handle multiple operations concurrently without getting stuck**. It works like a manager who **keeps track of tasks** and makes sure they are **done in the right order**.

How the Event Loop Works

1. Start: Node.js begins by running your code from top to bottom.
2. Asynchronous Tasks: When it encounters an **asynchronous task** (like reading a file, making a network request, or using setTimeout), it **starts the task and moves on without waiting for it to complete**.
3. Task Queue: When an asynchronous task is done, its **callback function** is placed in a **task queue**.
4. Event Loop: The **event loop checks the task queue** and **executes the callbacks** when the main code is done running.

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout 1");
}, 1000);

setTimeout(() => {
  console.log("Timeout 2");
}, 500);

console.log("End");
```

```
//Output

Start
End
Timeout 2
Timeout 1
```

The event loop allows Node.js to handle multiple tasks efficiently without waiting for each task to complete before starting the next one. This makes Node.js great for building fast and responsive applications.

EventEmitter Class

The EventEmitter class is at the core of Node.js's event-driven architecture. It provides a way to **emit** and **listen to events**. For using it first, you need to include the **events** module.

```
const EventEmitter = require("events");
```

Emit an Event

Emitting an event means **triggering** or **signaling** that an **event has occurred**. When you emit an event, you tell the EventEmitter instance to **execute all the callback functions that are registered to listen for that specific event**. Trigger an event using **emit()**. This signals that something has happened and calls all the listeners for that event.

Listen to an Event

Listening to an event means **registering a callback function (listener) that will be executed when a specific event is emitted**. Register a callback function for an event using **on()**. This function will be executed whenever the specified event is emitted.

Listen to an Event Once

Register a callback function for an event using **once()**. This function will be executed only the first time the specified event is emitted, after which it is automatically removed.

```
const EventEmitter = require("events");
const myEmitter = new EventEmitter();

// Listen to an event
myEmitter.on("alwaysGreet", () => {
  console.log("Hello, this event will always be handled.");
});

// Listen to an event once
myEmitter.once("greetOnce", () => {
  console.log("Hello, this event will be handled only once.");
});

// Emit the events
console.log("Emitting events the first time:");
myEmitter.emit("alwaysGreet");
myEmitter.emit("greetOnce");

console.log("Emitting events the second time:");
myEmitter.emit("alwaysGreet");
myEmitter.emit("greetOnce"); // This will not trigger the listener
```

Callbacks

A callback is a **function** that is **passed to another function as an argument** and is **executed after some operation has been completed**. In Node.js, callbacks are used extensively to handle **asynchronous** operations, allowing your program to continue running while waiting for the operation to be completed. By using callbacks, Node.js can perform **non-blocking operations**, allowing your applications to be efficient and responsive.

Example: Basic Callback

```
// Define a function that takes a callback
function doSomething(callback) {
  console.log("Doing something...");
  callback();
}

// Define the callback function
function afterDone() {
  console.log("Done!");
}

// Call the function and pass the callback
doSomething(afterDone);
```

Example: Asynchronous Callback with setTimeout

```
// Define a function that takes a callback
function doAsync(callback) {
  console.log("Starting asynchronous operation...");
  setTimeout(() => {
    console.log("Asynchronous operation complete");
    callback();
  }, 2000);
}

// Define the callback function
function afterAsync() {
  console.log("After async operation");
}

// Call the function and pass the callback
doAsync(afterAsync);
```

Example: Read a File Asynchronously

```
const fs = require("fs");

// Define the callback function
function fileReadCallback(err, data) {
  if (err) {
    console.error("Error reading file:", err);
    return;
  }
  console.log("File content:", data);
}

// Read the file and pass the callback
fs.readFile("example.txt", "utf8", fileReadCallback);

console.log("File read initiated");
```

Error Handling in Node.js

Error handling is a programming practice that involves anticipating, detecting, and responding to errors or exceptional conditions that can occur during the execution of a program. Effective error handling ensures that a program can gracefully handle unexpected situations and continue to operate or terminate cleanly, providing useful information to the user or developer about what went wrong. Node.js supports both synchronous and asynchronous error handling mechanisms.

Synchronous Error Handling

Synchronous code is executed line by line, so handling errors in such code is straightforward using **try-catch**.

```
const fs = require("fs");

// Synchronous error handling
function readFileSync() {
  try {
    // This will throw an error if the file doesn't exist
    const data = fs.readFileSync("nonexistentfile.txt", "utf8");
    console.log(data);
  } catch (err) {
    console.error("Error reading file:", err.message);
  }
}

readFileSync(); //output :Error reading file: Error reading file: ENOENT: no such file or directory...
```

In the above example, the readFileSync function tries to read a non-existent file. If an error occurs, it is caught by the catch block, and an error message is printed to the console.

Asynchronous Error Handling

Asynchronous operations, like reading a file or making an HTTP request, require a different approach for error handling. You can use **callbacks**, **promises**, or **async/await**.

Asynchronous error handling using callbacks

In this example, **readFile** is an asynchronous function that uses a callback. If an error occurs, it is passed as the first argument (err) to the callback.

```
const fs = require("fs");

// Asynchronous error handling using callbacks
function readFileAsync() {
  fs.readFile("nonexistentfile.txt", "utf8", (err, data) => {
    if (err) {
      console.error("Error reading file:", err.message);
      return;
    }
    console.log(data);
  });
}
readFileAsync();
```

Asynchronous error handling using async/await

The **async/await** syntax allows you to write asynchronous code that looks like synchronous code. You can use **try-catch blocks** to handle errors in **async functions**.

```
const fs = require("fs").promises;

// Asynchronous error handling using async/await
async function readFileAsync() {
  try {
    const data = await fs.readFile("nonexistentfile.txt", "utf8");
    console.log(data);
  } catch (err) {
    console.error("Error reading file:", err.message);
  }
}

readFileAsync();
```

Asynchronous error handling using promises

A promise represents an asynchronous operation that can succeed (resolve) or fail (reject). Use the **then** method to handle the **success case** and the **catch** method to handle **errors**. Promises provide a clean and readable way to handle asynchronous operations and errors in Node.js.

```
const fs = require("fs").promises; // Use the promises version of the fs module

// Function to read a file using promises
function readFileAsync() {
  fs.readFile("nonexistentfile.txt", "utf8")
    .then((data) => {
      console.log(data); // This runs if the file is read successfully
    })
    .catch((err) => {
      console.error("Error reading file:", err.message); // This runs if there's an error
    });
}

readFileAsync();
```