

Introduction to Node.js

What is Node.js?

Node.js is an **open-source, cross-platform JavaScript runtime environment** that allows developers to execute JavaScript code outside of a browser. It is widely used for building scalable and high-performance applications.

Key Features of Node.js:

- **V8 Engine:** Node.js runs on the V8 JavaScript engine (also used in Google Chrome), which makes it fast and efficient.
 - **Asynchronous and Event-Driven:** Node.js handles I/O operations (network requests, database access, file system operations) asynchronously, meaning it doesn't block the execution of other code.
 - **Single-Threaded, Non-Blocking I/O:** Unlike traditional multi-threaded models, Node.js operates on a single thread but can handle multiple requests concurrently.
 - **Rich Package Ecosystem:** Node.js has a vast collection of open-source modules available via npm (Node Package Manager), making development faster and easier.
-

What is a Module in Node.js?

A **module** in Node.js is a reusable block of code that can be imported into other files. It helps in organizing the code and improving maintainability.

Types of Modules in Node.js:

1. **Core Modules:** Built-in modules provided by Node.js, such as http, fs, path, os.
2. **Local Modules:** Custom modules created by developers within a project.
3. **Third-Party Modules:** Modules installed via npm, like express, mongoose, dotenv.

Example of Importing a Core Module:

```
const fs = require('fs'); // Importing the File System module
```

The HTTP Module in Node.js

The **http module** in Node.js allows the creation of web servers and handling HTTP requests and responses.

Creating a Basic Web Server:

```
const { createServer } = require('node:http'); // Importing the HTTP module

const hostname = '127.0.0.1'; // Localhost IP

const port = 3000; // Port number

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

  res.statusCode = 200; // HTTP status code for success

  res.setHeader('Content-Type', 'text/plain'); // Setting response header

  res.end('Hello World'); // Sending response body

});
```

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

Explanation:

1. ``: Loads the built-in HTTP module.
2. ``: Creates an HTTP server that listens for incoming requests.

3. Request and Response Objects:

- o req: Represents the HTTP request (contains request headers, method, URL, etc.).
- o res: Represents the HTTP response (used to send data back to the client).

4. ``: Binds the server to a specific port and hostname.
5. ``: Ends the response and sends data to the client.

Running the Server:

1. Save the file as server.js.
 2. Run the command:
 3. node server.js
 4. Open a browser and visit <http://127.0.0.1:3000/> to see the output.
-

Conclusion

Node.js is a powerful tool for building fast and scalable web applications. Understanding **modules**, the **http module**, and the **createServer() method** is crucial for backend development. With its non-blocking I/O model, Node.js can efficiently handle multiple connections, making it ideal for real-time applications like chat apps, APIs, and data-intensive services.

1. **Asynchronous programming and callbacks**
2. **Timers**
3. **Promises**
4. **Async and Await**
5. **Closures**
6. **The Event Loop**

Asynchronous Programming and Callbacks JavaScript is single-threaded, meaning it executes one operation at a time. However, asynchronous programming allows tasks like I/O operations, network requests, and timers to run in the background without blocking the main thread. This is crucial in Node.js since it handles multiple requests efficiently.

Callbacks: A callback is a function passed as an argument to another function and executed later. Callbacks are commonly used in asynchronous operations. Callback functions are a way to ensure certain code runs only after another code has already finished execution.

Example:

```
function greet(name, callback) {
  console.log('Hello ' + name);
  callback();
}
```

```

function logEnd() {
    // Passing 'logEnd' as a callback to the 'greet'
    // function
    console.log('Function execution ended.');
}

// Output:
// Hello Alice
// Function execution ended.

Another Example:

function processUserData(userId, callback) {
    // Simulate fetching user data with a 1.5-second delay
    setTimeout(() => {
        if (userId) {
            callback(null, { id: userId, name: "John Doe" }); // Success case
        } else {
            callback("User ID not provided", null); // Error case
        }
    }, 1500);
}

```

- **processUserData:** This function takes two parameters: userId and callback.
 - userId: The ID of the user whose data is to be fetched.
 - callback: A function that will be executed after the data fetching operation completes.
- **setTimeout:** Simulates a delay of 1.5 seconds to mimic an asynchronous operation (like fetching data from a server).
 - After 1.5 seconds, it checks if userId is provided.
 - If userId is provided, it calls the callback function with null as the first argument (indicating no error) and a user object { id: userId, name: "John Doe" } as the second argument (indicating success).
 - If userId is not provided, it calls the callback function with an error message "User ID not provided" as the first argument and null as the second argument.

2. Function Call

javascript

```

processUserData(1, (error, user) => {
    if (error) {
        console.log("Error:", error); // Logs error message if userId is not provided
    } else {
        console.log("User Data:", user); // Logs user data if successful
    }
})

```

```
});
```

- **processUserData(1, callback):** Calls the processUserData function with 1 as the userId and an anonymous function as the callback.

- Inside the callback function:

- It checks if there is an error.
 - If there is an error, it logs the error message to the console.
 - If there is no error, it logs the user data to the console.

Timers Timers allow executing code after a specified delay or at intervals. JavaScript provides three main timer functions:

- `setTimeout(fn, delay):` Executes fn after delay milliseconds.
- `setInterval(fn, delay):` Repeats execution of fn every delay milliseconds.
- `clearTimeout` and `clearInterval`: Stop the execution of timers.

Example:

```
setTimeout(() => console.log("Executed after 2 seconds"), 2000);

let interval = setInterval(() => console.log("Repeating every 1 second"), 1000);

setTimeout(() => clearInterval(interval), 5000); // Stops after 5 seconds
```

A Promise in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It provides a cleaner, more robust way to handle asynchronous operations compared to traditional callback functions, which can lead to callback hell.

Key Concepts

- **Pending:** The initial state; neither fulfilled nor rejected.
- **Fulfilled:** The operation completed successfully.
- **Rejected:** The operation failed.

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {

  const success = true; // You can change this to `false` to simulate failure

  if (success) {
    resolve("Operation succeeded!"); // If successful, resolve the promise
  } else {
    reject("Operation failed."); // If failed, reject the promise
  }
})
```

});

Handling Promises

You can handle promises using then and catch methods:

The then and catch methods are used to handle the outcome of a Promise in JavaScript.

then Method

The then method is used to handle the resolved value of a Promise. It takes up to two arguments:

1. A callback function for the onFulfilled case (when the Promise is resolved successfully).
2. An optional callback function for the onRejected case (when the Promise is rejected).

catch Method

The catch method is used to handle the rejected value of a Promise. It takes one argument, a callback function for the onRejected case.

```
myPromise
```

```
.then((message) => {  
  console.log("Success:", message); // Runs if the promise is resolved  
})  
.catch((error) => {  
  console.log("Error:", error); // Runs if the promise is rejected  
});
```

Using Async/Await

Async/await syntax provides an even cleaner way to work with promises:

Asynchronous in JavaScript

In JavaScript, asynchronous operations don't block the execution of other code. This is achieved using constructs like callbacks, Promises, and the async/await syntax.

async Keyword

The async keyword is used to declare an asynchronous function. This function returns a Promise implicitly, and you can use the await keyword within it.

await Keyword

The await keyword can only be used inside an async function. It pauses the execution of the async function until the Promise is resolved or rejected.

Handling Errors

When using async and await, it's important to handle errors using try and catch blocks. If the awaited Promise is rejected, the code in the catch block will execute.

```
async function handleOperations() {  
  try {  
    const result1 = await asyncOperation1;  
    console.log(result1);  
    const result2 = await asyncOperation2;  
    console.log(result2);  
  } catch (error) {  
    console.error("An error occurred: ", error);  
  }  
}
```

```
} catch (error) {  
    console.error(error);  
}  
}  
  
handleOperations();
```

In this example, `async` keyword is used to define an asynchronous function, and `await` pauses the function execution until the promise settles.

Summary

- **Promises** simplify asynchronous code, making it more readable and manageable.
- **Then** and **catch** methods handle success and error cases.
- **Chaining** allows for sequential asynchronous operations.
- **Async/await** provides a syntactically cleaner approach for dealing with promises.

Async and Await `async` and `await` make asynchronous code look synchronous and are used with Promises.

Example:

```
function fetchData() {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            resolve("Data fetched successfully!");  
        }, 2000);  
    });  
}  
  
async function getData() {  
    console.log("Fetching data...");  
    try {  
        const data = await fetchData(); // Pauses here until fetchData() is resolved  
        console.log(data); // Logs "Data fetched successfully!" after 2 seconds  
    } catch (error) {  
        console.error("Error:", error); // Catches and logs any error from fetchData()  
    }  
    console.log("Done fetching data.");  
}  
getData();
```

Breakdown of the Example

1. fetchData Function:

- Returns a Promise that resolves with the message "Data fetched successfully!" after a 2-second delay.

2. getData Function:

- Declared as an async function.
 - Logs "Fetching data..." to the console.
 - Uses await to wait for fetchData to resolve and assigns the resolved value to the data variable.
 - Logs the data to the console.
 - Logs "Done fetching data." to the console after the asynchronous operation is complete.
-

Closures

A closure is a function that remembers the environment in which it was created. It allows a function to access variables from its outer (enclosing) scope even after that outer function has finished executing.

Here's an example to illustrate the concept:

```
function outerFunction() {  
  const outerVariable = 'I am from the outer scope';  
  
  function innerFunction() {  
    console.log(outerVariable); // Accesses outerVariable even after outerFunction is done  
  }  
  
  return innerFunction;  
}  
  
const myClosure = outerFunction();  
  
myClosure(); // Logs: 'I am from the outer scope'
```

Explanation:

- outerFunction creates a local variable outerVariable and defines innerFunction.
 - innerFunction has access to outerVariable even after outerFunction has returned.
 - myClosure holds the innerFunction, which still has access to outerVariable due to closure.
-

The Event Loop The Event Loop handles JavaScript's asynchronous operations, ensuring that non-blocking code is executed efficiently. It processes the **Call Stack**, **Web APIs**, **Callback Queue**, and **Microtask Queue (Promises)** in a cyclic manner.

Example:

```
console.log("Start");  
  
setTimeout(() => console.log("Timeout callback"), 0);
```

```
Promise.resolve().then(() => console.log("Promise resolved"));
console.log("End");
```

Output:

Start

End

Promise resolved

Timeout callback

Promises are executed before setTimeout because they are in the **Microtask Queue**, which has higher priority than the **Callback Queue**.

These concepts are fundamental to mastering JavaScript and Node.js for backend development and job interviews.

Here's a beginner-friendly breakdown of the topics with easy-to-follow points:

V8 JavaScript Engine

V8 is the JavaScript engine that makes Chrome and Node.js run JavaScript fast. Let's break it down:

1. What is V8?

- It is the engine that runs JavaScript code inside Google Chrome.
- Node.js also uses it to run JavaScript outside the browser.

2. What does it do?

- It **parses** (reads) and **executes** JavaScript code.
- The browser provides extra features like the **DOM (Document Object Model)** and other Web APIs.

3. V8 is not tied to Chrome!

- It can run JavaScript outside of browsers too (this is how **Node.js** works).
- Thanks to V8, JavaScript can be used for **server-side** coding, not just in web pages.

4. Other JavaScript Engines:

-  **SpiderMonkey** → Used in Mozilla Firefox
-  **JavaScriptCore (Nitro)** → Used in Safari
-  **V8** → Used in Chrome & Edge (Edge now uses Chromium)

5. Why is V8 important?

- It helps JavaScript run super fast.
 - It continuously improves to make web pages and Node.js applications perform better.
-

How JavaScript Runs in V8 (Compilation & Performance) ↴

1. **JavaScript was originally "interpreted"** (executed line by line).
 2. **Now, V8 compiles JavaScript** before running it (this makes it faster).
 3. **JIT (Just-In-Time) Compilation**
 - Instead of running directly, JavaScript is first **compiled into machine code** (fast format for computers).
 - This speeds up execution a lot, which is crucial for big applications like Google Maps.
 4. **Every year, JavaScript engines compete to be faster** → This benefits developers and users!
-

npm (Node Package Manager) – Your JavaScript Toolbox ⚙

What is npm?

- **npm = Node.js package manager** (used to install and manage code libraries).
- It has **millions of reusable packages** for JavaScript developers.

Why use npm?

- Easily install code libraries (instead of writing everything from scratch).
- Automatically **manage dependencies** (code that your project needs to work).
- Helps in **frontend & backend** development (not just Node.js).

Installing All Dependencies

- If a project has a package.json file, run:
 - npm install
 - This installs all required libraries in the **node_modules** folder.

Installing a Specific Package

- To install a package:
 - npm install <package-name>
- Example:
 - npm install express

Common npm Flags

Flag	Meaning
--save	Adds package to dependencies (default in npm 5+)
--save-dev	Adds package to development dependencies
--no-save	Installs but does not add to package.json
--save-optional	Adds to optional dependencies
-S	Shortcut for --save

Flag	Meaning
-D	Shortcut for --save-dev

Updating Packages

- Update all dependencies:
- npm update
- Update a specific package:
- npm update <package-name>

Versioning with npm

- You can install a specific version of a package:
- npm install <package-name>@<version>

Example:

```
npm install express@4.17.1
```

Running Custom Commands with npm ↴

Scripts in package.json

- You can define commands inside package.json like this:
 - {
 - "scripts": {
 - "start": "node app.js",
 - "dev": "nodemon app.js"
 - }
 - }
 - Then, instead of typing long commands, you can run:
 - npm run start
 - npm run dev
-

Final Thoughts ⚡

- **V8** makes JavaScript fast & powerful.
- **npm** helps manage JavaScript libraries efficiently.
- Modern JavaScript **compiles before execution** for better performance.
- Knowing these basics will help you in **Node.js development and job interviews!**

Let me know if you want more details or examples! 💬

1. What is ECMAScript (ES6)?

- ECMAScript (or ES) is the standard for JavaScript.
- ES6 (ECMAScript 2015) introduced many modern JavaScript features.
- Node.js follows the latest updates of this standard by using the **V8 engine**.

2. Features in Node.js

There are three types of JavaScript features in Node.js:

1. **Shipping features** – Fully stable and enabled by default.
2. **Staged features** – Almost ready but require a special flag (--harmony).
3. **In-progress features** – Still being developed, risky to use.

⌚ To check available in-progress features:

Run this command in your terminal:

```
node --v8-options | grep "in progress"
```

3. Which Features Are Available in My Node.js Version?

- Use [node.green](#) to check which ECMAScript features are supported in different Node.js versions.

4. The --harmony Flag

- This flag was used to enable experimental JavaScript features.
- It now means the same as --es_staging, enabling staged (not fully stable) features.
- If you want stability in production, **avoid using it** because future Node.js updates may break your code.

5. Checking V8 Version in Node.js

- To see which version of V8 your Node.js is using, run:

```
node -p process.versions.v8
```

Development vs Production in Node.js

- Node.js itself has no special settings for "development" or "production."
- However, some libraries check the **NODE_ENV** variable to adjust their settings.

Best Practice: Set NODE_ENV to Production

```
NODE_ENV=production node app.js
```

- This improves performance by disabling extra debugging tools.

Why NODE_ENV Can Be a Bad Practice?

- Developers sometimes use it to change how the code works in different environments:

```
if (process.env.NODE_ENV === 'development') {  
  console.log("Development mode!");  
}
```

```
if (process.env.NODE_ENV === 'production') {  
    console.log("Production mode!");  
}
```

- This can **cause unexpected bugs** because **staging and production behave differently**.
 - Instead, use **feature flags or config files** to handle environment-specific behavior.
-

WebAssembly in Node.js

1. What is WebAssembly?

- A super-fast, low-level language that runs in **browsers and Node.js**.
- You can write WebAssembly (.wasm files) in languages like **C, C++, Rust, or AssemblyScript**.

2. WebAssembly Key Concepts

- **Module** – The compiled WebAssembly file (.wasm).
- **Memory** – A resizable memory buffer.
- **Table** – A list of references (like function pointers).
- **Instance** – A running version of a WebAssembly module.

3. How to Use WebAssembly in Node.js

Example: Running a WebAssembly module in Node.js

```
const fs = require('fs');  
  
// Load the WebAssembly file  
  
const wasmBuffer = fs.readFileSync('/path/to/add.wasm');  
  
// Instantiate WebAssembly in Node.js  
  
WebAssembly.instantiate(wasmBuffer).then(wasmModule => {  
    const { add } = wasmModule.instance.exports;  
    console.log(add(5, 6)); // Output: 11  
});
```

4. Generating WebAssembly Files

You can create .wasm files using:

- **Emscripten** – Convert C/C++ code to WebAssembly.
- **wasm-pack** – Convert Rust code to WebAssembly.
- **AssemblyScript** – Write WebAssembly with a TypeScript-like syntax.

5. WebAssembly and the OS

- WebAssembly **cannot access the OS** directly.
 - Use **Wasmtime + WASI API** to allow WebAssembly to interact with files, network, etc.
-

Final Takeaways

- ✓ Node.js follows ECMAScript standards using the V8 engine.
- ✓ New JavaScript features are released in three stages: shipping, staged, and in progress.
- ✓ Setting NODE_ENV=production improves performance, but don't rely on it for logic changes.
- ✓ WebAssembly lets you run super-fast code in Node.js using .wasm files.
- ✓ Use tools like wasm-pack, emscripten, and AssemblyScript to create WebAssembly modules.

Let me know if you want more details on any section! ↗

Here's a beginner-friendly breakdown of the concepts in your text, presented as a list for easier understanding. ↗

What is Undici?

- ◊ Undici is an HTTP client library for Node.js that powers the fetch API.
- ◊ It is built from scratch and does not use Node.js's built-in HTTP module.
- ◊ It is high-performance and good for handling many requests efficiently.

Basic Usage of Fetch API with Undici

❖ GET Request (Fetching Data)

- Use fetch(url) to make a request to a web server.
- Example: Get a list of posts from an API.

```
async function main() {  
  
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');  
  
  const data = await response.json();  
  
  console.log(data);  
  
}  
  
main().catch(console.error);
```

The .json() method converts the response into a JavaScript object.

❖ POST Request (Sending Data)

- Use fetch(url, options) to send data to a server.
- Example: Sending a new post to an API.

```
const body = { title: 'foo', body: 'bar', userId: 1 };  
  
async function main() {  
  
  const response = await fetch('https://jsonplaceholder.typicode.com/posts', {  
  
    method: 'POST',  
  
    headers: { 'Content-Type': 'application/json' },  
  
  })  
  
  const data = await response.json();  
  
  console.log(data);  
  
}  
  
main().catch(console.error);
```

```

    body: JSON.stringify(body),
  });

const data = await response.json();
console.log(data);
}

main().catch(console.error);

```

- The headers define the request type (JSON).
 - The body is converted into a JSON string before sending.
-

Customizing Fetch API with Undici

- ◊ You can **customize requests** using headers, methods, and body options.
- ◊ Example: Sending a **POST request** to an **LLM API (Ollama)**.

```

import { Pool } from 'undici';

const ollamaPool = new Pool('http://localhost:11434', { connections: 10 });

async function streamOllamaCompletion(prompt) {
  const { statusCode, body } = await ollamaPool.request({
    path: '/api/generate',
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ prompt, model: 'mistral' }),
  });

  if (statusCode !== 200) {
    throw new Error(`Ollama request failed with status ${statusCode}`);
  }

  let partial = "";

  const decoder = new TextDecoder();
  for await (const chunk of body) {
    partial += decoder.decode(chunk, { stream: true });
    console.log(partial);
  }
}

await streamOllamaCompletion('What is recursion?');

```

- Uses **pools** to manage multiple connections efficiently.
- Streaming** allows real-time processing of data chunks.

Streaming Responses with Undici

- ◊ **Streaming** allows you to process data in chunks instead of waiting for the full response.
- ◊ Example: Fetching GitHub repositories.

```
import { stream } from 'undici';

import { Writable } from 'stream';

async function fetchGitHubRepos() {

  const url = 'https://api.github.com/users/nodejs/repos';

  const { statusCode } = await stream(

    url,

    { method: 'GET', headers: { 'User-Agent': 'undici-stream-example', Accept: 'application/json' } },

    () => {

      let buffer = "";

      return new Writable({

        write(chunk, encoding, callback) {

          buffer += chunk.toString();

          try {

            const json = JSON.parse(buffer);

            console.log('Repository Names:', json.map(repo => repo.name));

            buffer = "";

          } catch (error) {

            console.error('Error parsing JSON:', error);

          }

          callback();

        },

        final(callback) {

          console.log('Stream processing completed.');

          callback();

        },

      });

    }

  );

  console.log(`Response status: ${statusCode}`);

}
```

```
fetchGitHubRepos().catch(console.error);
```

- Processes GitHub API responses **chunk by chunk** using a writable stream.
 - Prevents **memory overload** by handling data efficiently.
-

5 Security Best Practices in Node.js

1 Common Security Threats & Fixes

1. Denial of Service (DoS) Attack (CWE-400)

- Happens when too many HTTP requests **overload** the server.
- **Fix:** Use a **reverse proxy** and **limit connections** per host.

2. DNS Rebinding Attack (CWE-346)

- Malicious websites trick Node.js into thinking they are local.
- **Fix:** Disable the --inspect flag in production.

3. Information Exposure (CWE-552)

- Sensitive files might be **accidentally published** with an npm package.
- **Fix:** Use .npmignore and npm publish --dry-run before publishing.

4. HTTP Request Smuggling (CWE-444)

- Attacker tricks Node.js into **misinterpreting HTTP requests**.
- **Fix:** **Normalize** requests and use HTTP/2.

5. Timing Attacks (CWE-208)

- Hackers measure **response time** to guess passwords.
- **Fix:** Use **constant-time comparison** like crypto.timingSafeEqual().

6. Malicious Third-Party Modules (CWE-1357)

- Some **npm packages** can steal your data.
- **Fix:**
 - **Pin package versions** (e.g., lodash@4.17.21).
 - **Verify code before installing** (GitHub vs npm).

Conclusion ⚡

- ◊ **Undici** is a high-performance HTTP client for Node.js.
- ◊ The **Fetch API** in Undici works like the browser's fetch().
- ◊ It allows **custom requests, streaming responses, and efficient connection pooling**.
- ◊ Security is **crucial**, and best practices help protect against attacks.

Would you like me to expand on any part? 😊🚀

Sure! Let's break it down step by step in a beginner-friendly way. I'll explain each concept with a simple list and provide examples where needed. 

Node.js Module System

Node.js follows a **module system** to organize code into separate files. This helps make the code reusable, maintainable, and easy to manage.

Global Object

- In Node.js, the **global object** is an object that is accessible from anywhere in your application.
- It provides built-in functions and variables.
- Unlike in browsers (where the global object is window), in Node.js, the global object is **global**.

Example:

```
console.log(global); // Prints all global properties
```

```
console.log(global.setTimeout); // Shows that setTimeout is a global function
```

Modules

- A **module** is just a JavaScript file that contains some code.
- Node.js organizes code into different files (modules) to keep things clean.
- There are **three types** of modules:
 1. **Built-in modules** (provided by Node.js, like fs, http, path)
 2. **User-defined modules** (files you create)
 3. **Third-party modules** (installed via npm, like express)

Creating a Module

- To create a module, you write some code inside a separate file and **export** it using module.exports.

Example: (Creating a module)

math.js

```
function add(a, b) {  
    return a + b;  
}  
  
module.exports = add; // Exporting function
```

Loading a Module

- To use a module in another file, we **import** it using require().

Example: (Loading a module)

app.js

```
const add = require('./math'); // Importing math.js  
console.log(add(5, 3)); // Output: 8
```

Module Wrapper Function

- Every module in Node.js is **wrapped** inside a function before execution.
- This function provides the following arguments:
 - exports → Shortcut for exporting
 - require → Function to import modules
 - module → Object representing the current module
 - __filename → Path of the current file
 - __dirname → Directory of the current file

Example:

```
console.log(__filename); // Prints the full path of this file  
console.log(__dirname); // Prints the directory where this file is located
```

Path Module

- The path module is a **built-in module** in Node.js that helps work with file paths.
- You can use it to join, resolve, and manipulate file paths.

Example:

```
const path = require('path');  
  
console.log(path.basename(__filename)); // Get file name  
console.log(path.dirname(__filename)); // Get directory name  
console.log(path.extname(__filename)); // Get file extension
```

OS Module

- The os module is a **built-in module** that provides system-related information.

Example:

```
const os = require('os');  
  
console.log(os.type()); // OS type (Windows/Linux/Mac)  
console.log(os.freemem()); // Free memory in bytes  
console.log(os.totalmem()); // Total memory in bytes
```

```
console.log(os.platform()); // OS platform (win32, linux, darwin)
```

💡 Summary:

Concept	Description
Global Object	Built-in global variables and functions in Node.js.
Modules	Code is divided into separate files for better management.
Creating a Module	Use module.exports to export functions/variables.
Loading a Module	Use require() to import modules in other files.
Module Wrapper	Every module is wrapped in a function.
Path Module	Helps work with file paths.
OS Module	Provides system information.

Would you like me to explain any of these in more detail? 😊🚀

Alright! Let's break it down step by step with simple explanations, lists, and examples. 🚀

📦 Node.js Module System

Node.js provides **built-in modules** that help developers perform different tasks easily, such as working with files, handling events, and creating servers.

📁 File System (FS) Module

- The **fs (File System) module** is used to interact with files and directories.
- It allows you to:
 - Read files
 - Write files
 - Delete files
 - Rename files
 - Create directories, etc.

✅ Example: Reading a File

📄 app.js

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

// Read file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
```

```
if (err) {  
    console.error(err);  
    return;  
}  
  
console.log(data); // Prints the content of example.txt  
});
```

Example: Writing to a File

```
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {  
    if (err) throw err;  
    console.log('File written successfully!');  
});
```

Events Module

- The **events module** allows Node.js to handle and trigger events.
- It follows the **Observer pattern** (one part of the code listens for events while another part triggers them).
- We use the `EventEmitter` class to create and manage events.

Example: Creating and Emitting an Event

```
const EventEmitter = require('events');  
  
const emitter = new EventEmitter();  
  
// Define an event listener  
  
emitter.on('greet', () => {  
    console.log('Hello! Event triggered.');//  
});  
  
// Emit (trigger) the event  
  
emitter.emit('greet');
```

Event Arguments

- We can **pass data (arguments)** while emitting an event.
- This helps send information along with an event.

Example: Passing Arguments in Events

```
emitter.on('userLoggedIn', (username) => {  
    console.log(`User ${username} has logged in.');//
```

```
});  
emitter.emit('userLoggedIn', 'Luffy');  
// Output: User Luffy has logged in.
```

4 Extending EventEmitter

- Instead of using EventEmitter directly, we can create our own **custom class** that extends it.
- This is useful when we need custom event-handling logic inside a class.

Example: Creating a Custom EventEmitter Class

```
const EventEmitter = require('events');  
  
class Logger extends EventEmitter {  
  
  log(message) {  
    console.log(message);  
    this.emit('messageLogged', { id: 1, text: message });  
  }  
}  
  
const logger = new Logger();  
  
logger.on('messageLogged', (eventData) => {  
  console.log('Listener received:', eventData);  
});  
  
logger.log('Hello, World!');
```

Explanation:

1. We create a class Logger that **extends** EventEmitter.
 2. The log() method prints a message and emits an event.
 3. When the event is emitted, a listener handles it.
-

5 HTTP Module

- The http module helps create web servers and handle requests.
- It is used to build backend applications in Node.js.

Example: Creating a Simple Web Server

```
const http = require('http');  
  
const server = http.createServer((req, res) => {  
  
  res.writeHead(200, { 'Content-Type': 'text/plain' });  
  res.end('Hello, this is a Node.js server!');
```

```
});  
  
server.listen(3000, () => {  
  console.log('Server is running on port 3000...');  
});
```

Explanation:

1. We import the http module.
 2. We create a server that sends "Hello, this is a Node.js server!" as a response.
 3. The server listens on **port 3000**.
-

Node Core Recap

💧 Summary of Key Node.js Concepts

Concept	Description
FS Module	Works with files (read, write, delete, rename).
Events Module	Handles and triggers custom events.
Event Arguments	Passes data when emitting events.
Extending EventEmitter	Creates custom event-handling classes.
HTTP Module	Builds web servers in Node.js.

💡 Final Thoughts

- The **FS module** helps with files.
- The **Events module** lets us handle and trigger custom events.
- We can **pass data in events** and **extend EventEmitter** for custom event logic.
- The **HTTP module** helps create servers.

Would you like a more detailed explanation of any part? 😊 💧