**Introduction to Node JS & Node JS Modules Introduction to Node JS & Node JS Modules**

**Node.js Basics**

1. What is Node.js and what is it used for?

* Node.js is an open-source.
* Cross-platform JavaScript runtime environment that runs JavaScript code outside of a browser.
* It's used for building fast and scalable server-side and networking applications, such as web servers, real-time chat apps, and APIs.

1. Explain the main differences between Node.js and traditional webserver environments like Apache or Nginx.

|  |  |  |
| --- | --- | --- |
|  | Node.js | Apache or Nginx. |
| Language | uses JavaScript | Use configuration files. |
| Concurrency | uses an event-driven, non-blocking I/O model | Use a threaded or process-based model. |
| Scalability | Node.js excels at handling many concurrent connections | May require more resources to handle similar loads. |
| Usage | ideal for real-time applications | Often used for static content and traditional web hosting. |

1. What is the V8 engine and how does Node.js utilize it?

* The V8 engine is Google's open-source JavaScript engine that compiles JavaScript into machine code.
* Node.js utilizes the V8 engine to execute JavaScript code on the server side, enabling fast and efficient performance.

1. Describe the event-driven architecture of Node.js.

* Node.js’s event-driven architecture means it uses an event loop to handle asynchronous operations.
* Instead of waiting for tasks to complete, Node.js processes other tasks and responds to events as they occur, making it efficient for handling multiple concurrent operations.

1. What are some common use cases for Node.js?

* 1. Real-time chat applications
* 2. Web servers and APIs
* 3. Streaming services
* 4. Single-page applications (SPAs)
* 5. IoT applications

1. How does Node.js handle asynchronous operations?

* Node.js handles asynchronous operations using an event loop, call-backs, and promises.
* This allows it to perform non-blocking I/O operations, enabling it to handle multiple tasks concurrently without waiting for each task to complete.

1. What is the purpose of the package.json file in a Node.js project?

* The `package.json` file in a Node.js project is used to manage the project's metadata, dependencies, scripts, and configuration settings.
* It defines the project's name, version, main file, and other important properties.

1. Explain the role of the Node Package Manager (NPM).

* The Node Package Manager (NPM) is a tool for managing JavaScript packages.
* It allows developers to install, update, and manage dependencies for Node.js projects, and also provides a registry for sharing reusable code.

1. What is the node\_modules folder and why is it important?

* The `node\_modules` folder contains all the installed dependencies for a Node.js project.
* It is important because it stores the code and libraries needed for the project to run.

1. How can you check the version of Node.js and NPM installed on your system?

* To check the version of Node.js installed, run `node -v` in your terminal. To check the version of npm installed, run `npm -v`.

1. How does Node.js handle concurrency and what are the benefits of this approach?

* Node.js handles concurrency through its event-driven, non-blocking I/O model.
* This approach allows it to efficiently handle multiple tasks simultaneously without blocking other operations.
* Benefits include high scalability, better performance for I/O-heavy applications, and effective utilization of hardware resources.

1. How does Node.js handle file I/O? Provide an example of reading a file asynchronously.

const fs = require('fs');

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

if (err) {

console.error(err);

return;

}

console.log(data);

});

* Fs.readFile reads the file example.txt asynchronously.
* ‘utf8’ specifies the encoding of the file data.
* The call back function (err, data) is executed once the file is read or an error occurs.

1. What are streams in Node.js and how are they useful?

* Streams in Node.js are objects that allow you to read or write data continuously.
* They are useful for handling large amounts of data efficiently, as they enable you to process data in chunks rather than all at once, reducing memory consumption and improving performance.
* Streams are commonly used for file operations, network communications, and processing data in real-time.

**Node.jsModules**

1. What are modules in Node.js and why are they important?

* Modules in Node.js are reusable pieces of code that can be imported into other files.
* They help organize code by splitting it into manageable sections, enhance code reusability, and simplify maintenance.
* Modules are important because they allow developers to keep their codebase clean and modular, making it easier to debug, update, and scale applications.

2. How do you create a module in Node.js? Provide a simple example.

* To create a module in Node.js, have to define the code in a separate file and export it using module. Exports. Then, you can import it into other files using require ( ).

**Module File**

function add(a, b) {

return a + b;

}

module.exports = add;

**Main File**

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

console.log(add(2, 3));

3. Explain the difference between require and import statements in Node.js.

* In Node.js, require is used for loading CommonJS modules, while import is used for loading ES6 (ECMAScript 2015) modules.
* **require:** Synchronous, used in CommonJS modules.

Example:

const module = require('module-name');

* **import:** Asynchronous, used in ES6 modules

**Example:**

import module from 'module-name';

* require is widely used in Node.js, but import is becoming more common as ES6 modules gain support.

4. What is the module.exports object and how is it used?

* The module. Exports object in Node.js is used to export functions, objects, or values from a module so they can be used in other files.
* Example:

**math.js**

module.exports = {

add: function(a, b) {

return a + b;

}

};

**app.js**

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

console.log(math.add(2, 3));

5. Describe how you can use the exports shorthand to export module contents.

* can use the exports shorthand to export module contents by directly attaching properties to the exports object.

Example:

**math.js**

exports.add = function(a, b) {

return a + b;

};

**app.js**

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

console.log(math.add(2, 3));

6. What is the Common JS module system?

* The CommonJS module system is a standard used in Node.js for defining and importing modules.
* It uses `require()` to load modules and `module.exports` or `exports` to export them, enabling modular code organization and reuse.

7. How can you import a module installed via NPM in your Node.js application?

* You can import a module installed via npm in your Node.js application using the require () function.

Example:

const moduleName = require('module-name');

* Replace ‘module-name’ with the name of the installed NPM package.

8. Explain how the path module works in Node.js. Provide an example of using it.

* The path module in Node.js provides utilities for working with file and directory paths.

Example:

**app.js**

const path = require('path');

const filePath = path.join(\_\_dirname, 'folder', 'file.txt');

console.log(filePath); // Outputs the absolute path to file.txt

9. How do you handle circular dependencies in Node.js modules?

* To handle circular dependencies in Node.js modules, structure your code to avoid them if possible, or use conditional require statements and split interdependent code into separate modules.
* Node.js will resolve the circular dependency by returning a partially loaded module if necessary.

Example:

// a.js

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

module.exports = () => console.log('a');

// b.js

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

module.exports = () => console.log('b');

10. What is a built-in module in Node.js? Name a few and explain their purposes.

* A built-in module in Node.js is a module provided by Node.js itself, without needing to install it separately.

Examples:

* Fs: Handles file system operations (e.g., reading/writing files).
* Http: Creates HTTP servers and handles requests/responses.
* Path: Manages and manipulates file and directory paths.
* Os: Provides operating system-related utility methods and properties.

11. Give me the short and direct and correct answer

* Relative module path: Begins with `./` or `../` and resolves relative to the current file's directory.

Example: `const module = require('./module-name');`

* Absolute module path: Begins with the module name or `/`, resolving based on the project's root directory.

Example: `const module = require('module-name');`

12. What is a module wrapper function in Node.js?

* A module wrapper function is a wrapper around every module's code that Node.js automatically applies before executing the module.
* It provides context to the module by exposing specific variables like `exports`, `module`, `require`, `\_\_filename`, and `\_\_dirname`.
* This wrapper function ensures that module-level variables and functions are scoped properly and that the module can interact correctly with other modules.

1. Describe the buffer module and its use in Node.js.

* The `buffer` module in Node.js provides a way to work with binary data directly.
* It allows for handling raw data in a fixed-size memory allocation outside of the JavaScript engine's garbage-collected heap.
* Buffers are used for reading from or writing to streams, handling binary data in protocols like TCP or HTTP, and manipulating data efficiently when dealing with files or network operations.

**Starting an HTTP Server in Node.js**

1. How do you create a simple HTTP server in Node.js? Provide a code example.

* Can create a simple HTTP server in Node.js using the built-in http module.

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}/`);

});

2. Explain the purpose of the http module inNode.js.

* The `http` module in Node.js is used to create and manage HTTP servers and clients. It allows you to handle HTTP requests and responses, enabling you to build web applications and APIs.

3. What method do you use to start the HTTP server and make it listen on a specific port?

* Can use the `listen` method to start the HTTP server and make it listen on a specific port.

4. How can you send a response to the client in an HTTP server created with Node.js?

* Can can send a response to the client using the `res.end()` method, optionally setting headers with `res.setHeader()` and the status code with `res.statusCode`.

5. Explain the request and response objects in the context of an HTTP server.

* `Request` object (`req`): Represents the client's request, containing details such as the URL, HTTP method, headers, and any data sent by the client.
* `Response` object (`res`): Represents the server's response, allowing you to set the status code, headers, and send data back to the client.

6. How do you handle different HTTP methods (GET,POST,etc.) in a Node.js HTTP server?

* By checking req.method in the request handler.

const http = require('http');

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

if (req.method === 'GET') {

// Handle GET request

} else if (req.method === 'POST') {

// Handle POST request

}

// Handle other methods

res.end();

});

server.listen(3000, () => {

console.log('Server running at http://127.0.0.1:3000/');

});

7. What is middle ware in the context of a Node.js HTTP server?

* Middleware in a Node.js HTTP server is a function that processes requests before they reach the final request handler. It can modify the request and response objects, and can either end the request-response cycle or pass control to the next middleware function.

8. How can you serve static files using an HTTP server in Node.js?

* Can serve static files using the fs module to read the files and the http module to serve them.

const http = require('http');

const fs = require('fs');

const path = require('path');

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

const filePath = path.join(\_\_dirname, 'public', req.url === '/' ? 'index.html' : req.url);

const extname = String(path.extname(filePath)).toLowerCase();

const mimeTypes = {

'.html': 'text/html',

'.js': 'text/javascript',

'.css': 'text/css',

'.json': 'application/json',

'.png': 'image/png',

'.jpg': 'image/jpg',

'.gif': 'image/gif',

'.wav': 'audio/wav',

'.mp4': 'video/mp4',

'.woff': 'application/font-woff',

'.ttf': 'application/font-ttf',

'.eot': 'application/vnd.ms-fontobject',

'.otf': 'application/font-otf',

'.svg': 'application/image/svg+xml'

};

const contentType = mimeTypes[extname] || 'application/octet-stream';

fs.readFile(filePath, (error, content) => {

if (error) {

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

fs.readFile(path.join(\_\_dirname, 'public', '404.html'), (error, content) => {

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

res.end(content, 'utf-8');

});

} else {

res.writeHead(500);

res.end(`Server Error: ${error.code}`);

}

} else {

res.writeHead(200, { 'Content-Type': contentType });

res.end(content, 'utf-8');

}

});

});

server.listen(3000, () => {

console.log('Server running at http://127.0.0.1:3000/');

});

9. Explain how to handle errors in an HTTP server created with Node.js.

* To handle errors in an HTTP server created with Node.js, you can use error-checking logic and send appropriate responses.

**1. Check for Errors**: Use try-catch blocks or error callbacks to catch errors.

**2. Send Error Response**: Set the response status code and message.

const http = require('http');

const fs = require('fs');

const path = require('path');

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

const filePath = path.join(\_\_dirname, 'public', req.url === '/' ? 'index.html' : req.url);

fs.readFile(filePath, (error, content) => {

if (error) {

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

// File not found

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

res.end('404 Not Found', 'utf-8');

} else {

// Other server error

res.writeHead(500, { 'Content-Type': 'text/plain' });

res.end(`Server Error: ${error.code}`, 'utf-8');

}

} else {

// Success

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

res.end(content, 'utf-8');

}

});

});

server.listen(3000, () => {

console.log('Server running at http://127.0.0.1:3000/');

});

* This handles both file not found (404) and other server errors (500).

10. How can you implement routing in a Node.js HTTP server without using external libraries?

* Can implement routing in a Node.js HTTP server by checking the req.url property within the request handler and directing requests based on the URL path.
* Example:

const http = require('http');

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

// Routing based on req.url

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

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

res.end('Welcome to the homepage!\n');

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

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

res.end('About us page\n');

} else {

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

res.end('404 Not Found\n');

}

});

server.listen(3000, () => {

console.log('Server running at http://127.0.0.1:3000/');

});