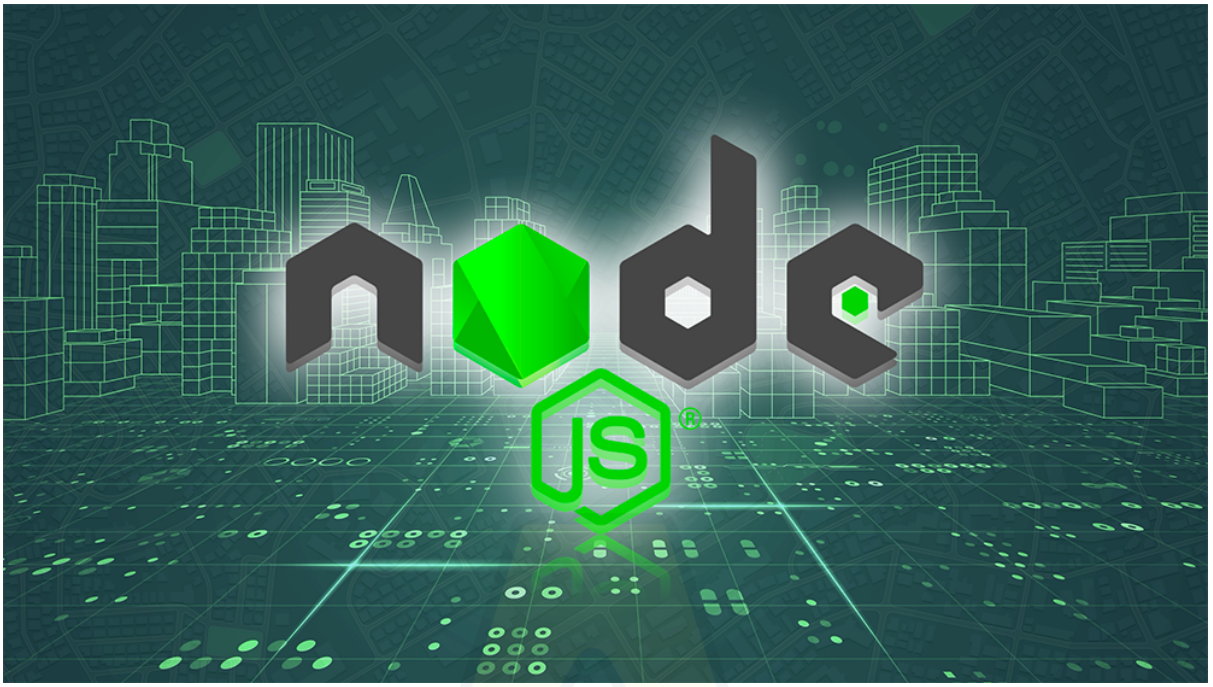
AIMERZ.ai
Aim.Act.Achieve

NOTES

Node.js - I

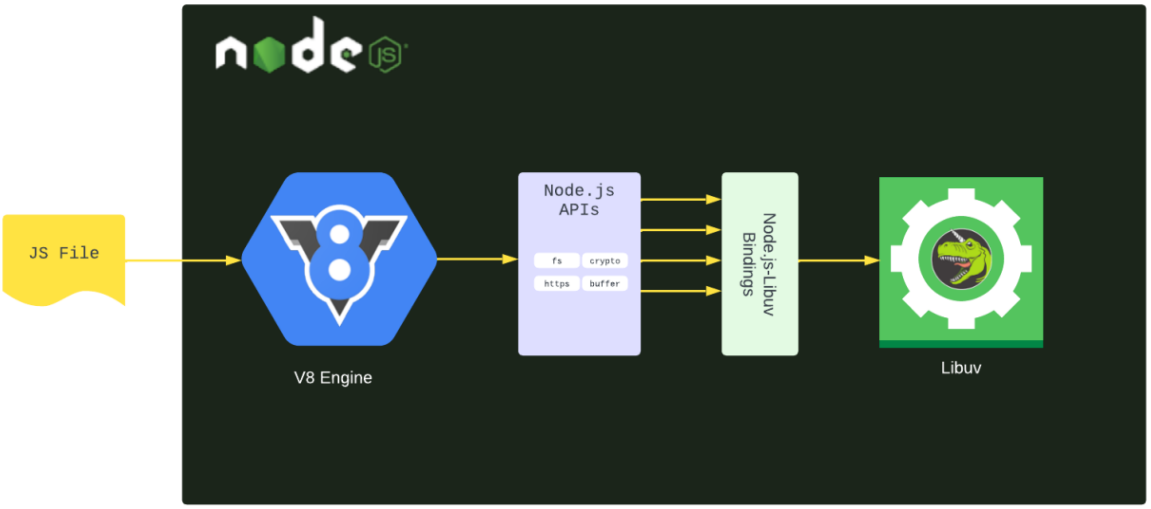
Introduction to NodeJS



Topic Covered:

1.What is NodeJS & V8 engine
2.NodeJS Architecture and working
3.Installing NodeJS
4.Introduction to NodeJS Module System
5.Global Object
6.Modules, creation and loading of modules
7.Path Module
8.OS Module
9.File System Module
10.HTTP module

What is NodeJS & V8 engine



Node.js Runtime

Node.js was made to allow developers to use JavaScript on the server side, making it easier to build fast, scalable, and real-time applications, like chat apps or APIs, with efficient handling of multiple connections at once.

Node.js allows developers to create both front-end and back-end applications using JavaScript. It was released in 2009 by Ryan Dahl.

"Node.js is an open-source and cross-platform JavaScript runtime environment."

Node.js is open-source: This means that the source code for Node.js is publicly available. And it's maintained by contributors from all over the world.

Node.js is cross-platform: Node.js is not dependent on any operating system software. It can work on Linux, macOS, or Windows.

Node.js is a JavaScript runtime environment: When you write JavaScript code in your text editor, that code cannot perform any task unless you execute (or run) it. And to run your code, you need a runtime environment.

Browsers like Chrome and Firefox have runtime environments. That is why they can run JavaScript code. Before Node.js was created, JavaScript could only run in a browser. And it was used to build only front-end applications.

Imagine you're shopping online. There are many actions happening behind the scenes:

1. **User interacts with the website**: You browse products, add them to your cart, and check out.
2. **Server needs to process your request**: When you click "Checkout," the server processes your information, calculates shipping costs, and checks product availability.

In this case, Node.js acts like the **"server engine"** that efficiently handles requests coming from multiple users simultaneously.

V8 engine

Google wanted to make web pages load faster by improving how JavaScript (the language used to create interactive websites) is executed.V8 was built to be very **fast** and **efficient** by directly converting JavaScript into **machine code** (the language understood by the computer's CPU), rather than interpreting it line-by-line like older engines.

The V8 engine is a high-performance JavaScript engine developed by **Google**. It is written in **C++** and is used in applications like **Google Chrome** and **Node.js** to execute JavaScript code.This makes it possible to build back-end applications using the same JavaScript programming language you may be familiar with.

How V8 Works in Google Chrome

1. **You open a web page** with JavaScript (say, an interactive webpage with animations).
2. **V8 takes the JavaScript code** from the page, compiles it into machine code, and executes it. This makes animations smooth and ensures that interactive elements work as expected.
3. If certain parts of the JavaScript are run repeatedly (like functions triggered by user clicks), V8 will **optimize** that code for better performance, reducing the time it takes to execute.

V8 allows Node.js to be **fast and efficient**, enabling it to handle a large number of concurrent connections in a **non-blocking, asynchronous manner,** which is a key reason why Node.js is popular for building scalable web applications.

When an operation is **non-blocking**, it means that the **execution of other code is not halted** or "blocked" while waiting for that operation to complete.In Node.js, however, because it is **non-blocking**, it can initiate an I/O operation and then **continue executing other code** while the operation completes in the background.When an operation is **asynchronous**, it means that the operation **does not stop or wait for completion** before moving on to the next task. Instead, it allows the code to continue running, and when the operation is finished, it will notify the program (usually by calling a **callback function** or using **Promises**).

NodeJS Architecture and working

Node.js Server Architecture: To manage several concurrent clients, Node.js employs a “Single Threaded Event Loop” design. The JavaScript event-based model and the JavaScript callback mechanism are employed in the Node.js Processing Model.

It employs two fundamental concepts:

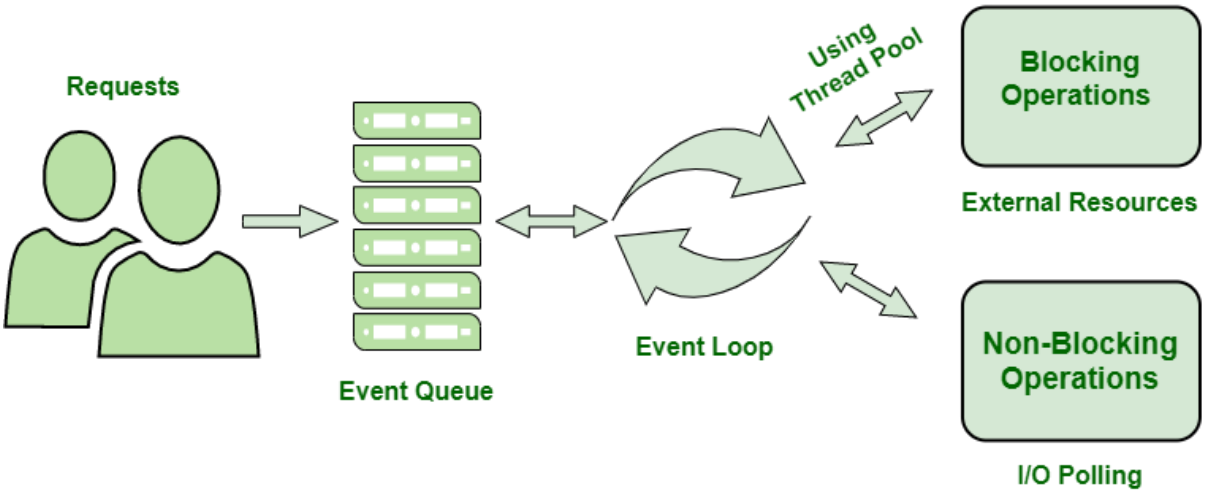
1. **Asynchronous model**

2. **Non-blocking of I/O operations**

These features enhance the scalability, performance, and throughput of Node.js web applications.

Components of the Node.js Architecture

- **Requests:** Depending on the actions that a user needs to perform, the requests to the server can be either blocking (complex) or non-blocking (simple).
- **Node.js Server:** The Node.js server accepts user requests, processes them, and returns results to the users.
- **Event Queue:** The main use of Event Queue is to store the incoming client requests and pass them sequentially to the Event Loop.
- **Thread Pool:** The Thread pool in a Node.js server contains the threads that are available for performing operations required to process requests.
- **Event Loop:** Event Loop receives requests from the Event Queue and sends out the responses to the clients.
- **External Resources:** In order to handle blocking client requests, external resources are used. They can be of any type (computation, storage, et



- Users send requests (blocking or non-blocking) to the server for performing operations.
- The requests enter the Event Queue first at the server-side.
- The Event queue passes the requests sequentially to the event loop. The event loop checks the nature of the request (blocking or non-blocking).
- Event Loop processes the non-blocking requests which do not require external resources and returns the responses to the corresponding clients
- For blocking requests, a single thread is assigned to the process for completing the task by using external resources.
- After the completion of the operation, the request is redirected to the Event Loop which delivers the response back to the client.

Installing NodeJS

Here are the steps to install Node.js.The latest version of Node.js is 23.5.0, which was released on December 19, 2024.

Step 1: Download and Install Node.js

Option 1: Using the Official Website (Recommended for Beginners)

1. **Go to the Official Node.js Website**:
    - Open your web browser and go to the Node.js official website.
2. **Choose the Version to Install**:
    - You'll see two versions of Node.js:
        - **LTS (Long Term Support)**: Recommended for most users, as it's stable and reliable.
        - **Current**: Contains the latest features, but may have some unstable changes.

The major difference is **LTS** is Stable and reliable, great for production use.**Current** has the newest features but might be less stable

    - For beginners, it's best to choose the **LTS version**.
3. **Download Node.js**:
    - Click on the download link for your operating system (Windows, macOS, or Linux). This will download a file to your computer.
        - For **Windows** and **macOS**, it will be an installer .msi or .pkg file.
        - For **Linux**, it might provide a .tar.xz file or instructions to use a package manager.
4. **Install Node.js**:
    - Once the file is downloaded, **open the installer** and follow the steps to install Node.js on your system.

For **Windows/macOS**, you just need to click through the prompts and agree to the terms.

On **Linux**, follow the instructions given on the download page (or use your system's package manager like apt or yum).

Step 2: Verify the Installation and Check the Version

After Node.js is installed, you can confirm that the installation was successful and check the version of Node.js and npm (Node Package Manager) by following these steps:

1.Open the Terminal/Command Prompt:

- On **Windows**, open **Command Prompt** (press Windows + R, type cmd, and press Enter).
- On **macOS/Linux**, open **Terminal**.

2.Check the Version of Node.js:

Type the following command and press Enter:

```
node -v
```

This will show the installed version of Node.js.

Example output:
v16.13.0

3.Check the Version of npm (Node Package Manager):

NPM (Node Package Manager) is a tool that comes with Node.js. It's used to:

1. **Install packages** (libraries or tools) for your project.
2. **Manage dependencies** easily.
3. **Share your own code** as reusable packages.

Type the following command and press Enter:

npm -v

This will show the installed version of npm.

Example output:
8.1.0

Step 3: Test Node.js (Optional)

To ensure everything is working correctly, you can create a simple "Hello World" script in Node.js:

1. **Create a new file**:

Open a text editor (like Notepad, VSCode, or any text editor you like) and create a new file named app.js.

2. **Write the code**:

Add the following code to the app.js file:

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

3. **Run the script**:

Save the file and go back to your **Terminal/Command Prompt**.

Navigate to the folder where app.js is saved using the cd command.

```javascript
cd path/to/your/folder
```

Then run the script with the command:
node app.js
You should see the output:

Hello, Node.js!

Introduction to NodeJS Module System

In Node.js, the Module System is a way of organizing and structuring your code by splitting it into separate files, each responsible for a specific functionality. This modular approach helps keep your code clean, reusable, and maintainable.

Module:

A **module** in Node.js is simply a file (or a piece of code) that contains related functionality. These modules can be imported (or required) into other files to reuse the code.

In JavaScript, a module can export its functionality using the **`module.exports`** object, and other files can import it using the **`require()`** function.

Advantage of Modules in NodeJS:

Code Reusability: By dividing the functionality into modules, you can reuse the same code in multiple places without repeating it.

Separation of Concerns: Different parts of your application can be kept in separate files, making it easier to manage.

Maintainability: As your project grows, managing everything in a single file would become difficult. Splitting the code into modules makes it easier to update, debug, and enhance individual parts.

How Node.js Module System Works

Node.js provides a built-in module system. Here's how it works:

1. **Modules in Node.js:**
    - **Core Modules:** These are pre-built modules that come with Node.js, such as `fs` (file system), `http`, `path`, etc.
    - **External Modules:** These are third-party modules that you can install from the Node.js package manager, npm.
    - **Custom Modules:** These are your own modules that you create to break down your application into smaller, manageable pieces.

Types of Modules in Node.js

1. **Core Modules:**
    1. These are built into Node.js and don't need to be installed.
    2. Examples: `http`, `fs`, `path`, `url`, etc.

Example:

```
JavaScript
const http = require('http');  // Importing the 'http'
core module
```

External Modules:

3. These modules are not built into Node.js but are available for installation via npm.
4. Example: Express, Lodash, and many others.

To install an external module (e.g., `express`):

```
npm install express
```

Example:

```
JavaScript
const express = require('express');  // Importing the external
'express' module
```

Custom Modules:

These are modules you create yourself.

Creating a custom module:

Create a module:

Let's say you want to create a `math.js` module that has a function to add two numbers.

```javascript
// math.js

function add(a, b) {

  return a + b;

}

module.exports = add;  // Exporting the function
```

Import the module:

In another file (e.g., `app.js`), you can import and use the `add` function.

```javascript
// app.js

const add = require('./math');  // Importing the custom module

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

module.exports = app;  // Exporting the function
```

In Node.js, there are two main types of export mechanisms used to share functionality between files/modules:

1. `module.exports`:
    - This is the most common way to export a single item (like a function, object, or class) from a module.
    - It directly defines what will be exposed to other files when they require the module.

Example:

```javascript
// myModule.js
function greet() {
  console.log("Hello, World!");
}
module.exports = greet;
```

In another file, you can require this module:

```javascript
// app.js
const greet = require('./myModule');
greet();  // Output: Hello, World!
```

2. `exports` (alias for `module.exports`):
    - `exports` is a shorthand for `module.exports`. It allows you to attach multiple properties to the module.
    - It's commonly used when you want to export more than one function or value.

Example:

```javascript
// myModule.js
exports.greet = function() {
  console.log("Hello!");
};
exports.farewell = function() {
  console.log("Goodbye!");
};
```

In another file, you can require and use both functions:

```javascript
// app.js
const myModule = require('./myModule');
myModule.greet();      // Output: Hello!
myModule.farewell();   // Output: Goodbye!
```

Global Object

In Node.js, the global object is an object that is accessible from anywhere in your application. It provides a set of global variables and functions that can be used throughout your code without the need to explicitly import or require them.

Think of the global object as a container that holds built-in properties and methods that are available everywhere in your application, no matter where you are in your code.

Key Features of the Global Object in Node.js

1. **Accessible Everywhere**:
   - The global object is available in every module and file in your application, which means you don't have to import it explicitly like other modules.
2. **Doesn't Need `require()`**:
   - Unlike other modules where you need to use `require()` to include them in your file, you can access global properties and methods directly without any import or require statements.

Examples of Global Properties in Node.js

Here are some **built-in properties** and **methods** that are available as part of the global object in Node.js:

1. `__dirname`

- **What it is**: It provides the **absolute path** of the **directory** in which the currently executing script resides.
- **Use case**: It's useful for working with file paths relative to the current script, especially when dealing with file system operations.

Example:

```javascript
console.log(__dirname);
// Output: /Users/username/projects/nodeapp
```

2. `__filename`

- It gives the **absolute path** of the **current file** being executed.
- **Use case**: Helpful when you want to know the exact file path that is running the code.

```javascript
console.log(__filename); // Output: /Users/username/projects/nodeapp/app.js
```

3. `global`

- The `global` object itself is the top-level object in Node.js, and it's the equivalent of the **window object** in browsers.
- **Use case**: You can assign properties to `global`, and they will be available globally across all files.

Example:

```javascript
global.myGlobalVar = "Hello, world!";

// In another file, you can access the global variable:
console.log(myGlobalVar);
// Output: Hello, world!
```

4. `process`

- The `process` object provides information and control over the current Node.js process. It gives you access to environment variables, command-line arguments, and more.
- **Use case**: It's used for handling things like environment settings, exiting the process, or getting the current working directory.

Examples:

Get the current process ID:

```javascript
console.log(process.pid);
// Output: The process ID of the current Node.js application
```

Get environment variables:

```javascript
console.log(process.env.NODE_ENV);
// Output: the value of the NODE_ENV environment variable (e.g.,
'development', 'production')
```

5. `require()`

- `require()` is a function that loads and includes external modules into your code. It's part of the global object, so you don’t need to import it to use it.
- **Use case**: It's the primary way to bring external modules (e.g., Node.js core modules, npm packages, or custom modules) into your application.

Example:

```javascript
const fs = require('fs');  // Importing the 'fs' (file system) module
console.log(fs);
```

Modules creation and loading of modules

Steps to Create Modules in Node.js

To create modules in Node.js, write functions, objects, or classes in a separate file and use module.exports to export them. Import these modules in other files using the require() function for reuse.

Step 1: Creating a Module

To create a module, you simply need to write code in a separate file. You can then export specific variables, functions, or objects from that file to be used in other parts of your application.

Let’s start by creating a simple module that performs some basic mathematical operations.

```javascript
// File name: calc.js
```

```
exports.add = function (x, y) {
    return x + y;
};

exports.sub = function (x, y) {
    return x - y;
};

exports.mult = function (x, y) {
    return x * y;
};

exports.div = function (x, y) {
    return x / y;
};
```

Step 2: Using a Module

Now that we've created a module, we can import it into another file and use the exported functions.

JavaScript

```
// File name: App.js
const calculator = require('./calc');

let x = 50, y = 20;

console.log("Addition of 50 and 20 is "
    + calculator.add(x, y));

console.log("Subtraction of 50 and 20 is "
    + calculator.sub(x, y));

console.log("Multiplication of 50 and 20 is "
    + calculator.mult(x, y));

console.log("Division of 50 and 20 is "
    + calculator.div(x, y));
```

Output:

```
Unset
Addition of 50 and 20 is 70
Subtraction of 50 and 20 is 30
Multiplication of 50 and 20 is 1000
Division of 50 and 20 is 2.5
```

Path Module

The path module in Node.js is a built-in module that provides utilities for working with and manipulating file and directory paths. It helps you interact with file systems in a platform-independent way. This means that it abstracts the differences between file paths on Windows, macOS, and Linux, allowing you to write code that works across different operating systems without worrying about path separators.

Why Use the Path Module?

- **Cross-platform compatibility:** Different operating systems use different path formats (e.g., Windows uses backslashes \ while Linux and macOS use forward slashes /). The `path` module handles these differences for you.
- **Simplifies working with file and directory paths:** It provides easy-to-use functions for tasks like joining paths, resolving absolute paths, and getting file extensions.

How to Use the Path Module

To use the `path` module, you need to require it at the top of your Node.js script:

```javascript
const path = require('path');
```

Here are the most commonly used real-world use cases of the path module for web developers:

Serving Static Files:

Resolve absolute paths for static assets (e.g., CSS, JS).

```javascript
const staticPath = path.join(__dirname, 'public');
app.use(express.static(staticPath));
```

Cross-Platform Compatibility:

Handle file paths that work on both Windows and Unix-based systems.

```javascript
const filePath = path.join('user', 'docs', 'file.txt');
```

Path Normalization:

Ensure clean and consistent file paths.

```javascript
const safePath = path.normalize('../../folder/../config.json');
```

Extracting File Details:

Get file name or extension for validation or display.

```
JavaScript
const ext = path.extname('photo.jpg'); // '.jpg'
```

Dynamic File Paths:

Construct paths for uploads, downloads, or saving files.

```
JavaScript
const filePath = path.join(__dirname, 'uploads', userId, 'file.png');
```

Resolving Paths:

Safely locate configuration or template files.

```
JavaScript
const configPath = path.resolve(__dirname, './config/settings.json');
```

setting the views directory

Set up directories for dynamic rendering.

```
JavaScript
app.set('views', path.join(__dirname, 'views'));
```

OS Module

In Node.js, the os module provides a set of operating system-related utility methods and constants. This module allows you to gather information about the system's hardware, memory, CPU, and more. It is built into Node.js, so you don't need to install it separately.

Here's an overview of the key features of the os module:

Basic Information about the Operating System

os.platform() – Returns a string identifying the operating system platform (e.g., 'darwin', 'win32', 'linux').

```
JavaScript
const os = require('os');
console.log(os.platform());  // e.g., 'darwin', 'win32', 'linux'
```

os.type() – Returns the operating system name (e.g., 'Linux', 'Darwin', 'Windows_NT').

```
JavaScript
console.log(os.type());  // e.g., 'Linux'
```

os.release() – Returns the operating system's release version.

```
JavaScript
console.log(os.release());  // e.g., '5.4.0'
```

os.hostname() – Returns the hostname of the operating system.

```
JavaScript
console.log(os.hostname());  // e.g., 'my-computer'
```

os.uptime() – Returns the system uptime in seconds (how long the system has been running).

```javascript
console.log(os.uptime());  // e.g., 345678.2345
```

os.arch() – Returns the CPU architecture (e.g., 'x64', 'arm').

```javascript
console.log(os.arch());  // e.g., 'x64'
```

os.cpus() – Returns an array of objects containing information about each CPU/core on the system.

```javascript
console.log(os.cpus());
// Example output:
// [
//   { model: 'Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz', speed: 2800,
times: { user: 1300420, nice: 0, sys: 440389, idle: 19493820, irq: 0 } },
//   { model: 'Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz', speed: 2800,
times: { user: 1201000, nice: 0, sys: 400000, idle: 19000000, irq: 0 } },
//   ...
// ]
```

Here are the real world use case of OS module:

Server Configuration and Monitoring

Use `os.cpus()` to check CPU details and optimize server resource usage.

File Path and Directory Management

Use `os.tmpdir()` to access the temporary directory for uploading files (e.g., handling user-uploaded images before processing them).

Cross-Platform Path Handling

Use `os.platform()` and `os.homedir()` to create platform-independent paths for file storage.

User and Session Information

Use `os.userInfo()` to fetch details about the current user and personalize application settings.

Server Uptime Tracking

Use `os.uptime()` to log or display server uptime on a monitoring dashboard.

Dynamic Environment Handling

Use `os.hostname()` and `os.networkInterfaces()` to configure dynamic environments like assigning IPs or hostnames for microservices.

File System Module

The `fs` module in Node.js provides an API for interacting with the file system. It allows you to perform various file and directory operations such as reading, writing, deleting files, and working with directories. The `fs` module is built into Node.js, so you don't need to install it separately.

Key Features of the `fs` Module

1. File Reading Operations

`fs.readFile(path, encoding, callback)`

Reads the contents of a file asynchronously.

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

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data); // File content
```

```
});
```

- **`path`**: The path to the file.
- **`encoding`**: The encoding to use (e.g., 'utf8').
- **`callback`**: A function to call after the operation. It takes two arguments: `err` (error, if any) and `data` (file contents).

`fs.readFileSync(path, encoding)`

Reads the contents of a file synchronously.

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

try {
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log(data); // File content
} catch (err) {
  console.error(err);
}
```

2. File Writing Operations

`fs.writeFile(path, data, encoding, callback)`

Asynchronously writes data to a file. If the file doesn't exist, it will be created.

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

fs.writeFile('example.txt', 'Hello, World!', 'utf8', (err) => {
  if (err) {
    console.error(err);
    return;
  }
```

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

`fs.writeFileSync(path, data, encoding)`

Synchronously writes data to a file. If the file doesn't exist, it will be created.

JavaScript

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

try {
  fs.writeFileSync('example.txt', 'Hello, World!', 'utf8');
  console.log('File written successfully!');
} catch (err) {
  console.error(err);
}
```

3. File Appending Operations

`fs.appendFile(path, data, encoding, callback)`

Asynchronously appends data to a file. If the file does not exist, it will be created.

JavaScript

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


fs.appendFile('example.txt', '\nAppended content', 'utf8', (err) => {
  if (err) {
    console.error(err);
    return;
```

```
  }

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

});


fs.appendFileSync(path, data, encoding)
```

Synchronously appends data to a file.

JavaScript

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

try {
  fs.appendFileSync('example.txt', '\nAppended content', 'utf8');
  console.log('Data appended successfully!');
} catch (err) {
  console.error(err);
}
```

4. File Deletion

`fs.unlink(path, callback)`

Asynchronously deletes a file.

JavaScript

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


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

  if (err) {

    console.error(err);
```

```
    return;
  }
  console.log('File deleted successfully!');
});


fs.unlinkSync(path)
```

Synchronously deletes a file.

JavaScript

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

try {
  fs.unlinkSync('example.txt');
  console.log('File deleted successfully!');
} catch (err) {
  console.error(err);
}
```

HTTP module

The `http` module in Node.js is used to create HTTP servers and clients. It provides functionality to handle HTTP requests and responses, making it a core part of building web servers in Node.js.

Key Features of the `http` Module

1. Creating an HTTP Server
2. Creating HTTP Requests (Client)
3. Handling HTTP Methods
4. Working with HTTP Headers
5. Handling Request Data
6. Setting Up HTTP Status Codes

1. Creating an HTTP Server

JavaScript

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, Node.js HTTP Server!');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

2. Making HTTP Requests (GET)

JavaScript

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

http.get('http://www.example.com', (res) => {
  let data = '';
  res.on('data', chunk => data += chunk);
  res.on('end', () => console.log(data));
}).on('error', err => console.log('Error:', err.message));
```

3. Handling Different HTTP Methods

JavaScript

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

const server = http.createServer((req, res) => {
  if (req.method === 'GET') {
    res.statusCode = 200;
    res.end('GET Request');
  } else if (req.method === 'POST') {
```

```
    res.statusCode = 200;
    res.end('POST Request');
  } else {
    res.statusCode = 405;
    res.end('Method Not Allowed');
  }
});

server.listen(3000);
```

4. Setting Response Headers

JavaScript

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


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

  res.setHeader('Content-Type', 'text/html');

  res.setHeader('X-Custom-Header', 'MyHeaderValue');

  res.statusCode = 200;

  res.end('<h1>Hello, World!</h1>');

});


server.listen(3000);
```

5. Handling POST Data

JavaScript

```
const http = require('http');
const querystring = require('querystring');

const server = http.createServer((req, res) => {
  if (req.method === 'POST') {
    let body = '';
    req.on('data', chunk => body += chunk);
    req.on('end', () => {
      const parsedData = querystring.parse(body);
      res.statusCode = 200;
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify({ receivedData: parsedData }));
    });
  } else {
    res.statusCode = 405;
    res.end('Method Not Allowed');
  }
});

server.listen(3000);
```

6. Setting HTTP Status Codes

JavaScript

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

const server = http.createServer((req, res) => {
  if (req.url === '/success') {
    res.statusCode = 200;
    res.end('Success');
  } else if (req.url === '/not-found') {
    res.statusCode = 404;
    res.end('Page Not Found');
  } else {
    res.statusCode = 500;
    res.end('Internal Server Error');
  }
});

server.listen(3000);
```

THANK YOU

AIMERZ.ai
Aim.Act.Achieve

Stay updated with us!

Vishwa Mohan

Vishwa Mohan

vishwa.mohan.singh

Vishwa Mohan