



CipherSchools

Class 1 NodeJS

Overview of Node.js

Node.js is an *open-source, cross-platform JavaScript runtime environment*. This means its source code is publicly available for sharing and modification, and it runs on various operating systems including Mac, Windows, and Linux. The core concept of a "JavaScript runtime environment" is central to understanding Node.js and will be explored further in the associated learning series.

Prerequisites

The sole prerequisite for this Node.js learning series is **modern JavaScript**.

ECMAScripts

The evolution of ECMAScript and related technologies began with the emergence of web browsers and the need for dynamic web content:

- **1993:** The first web browser with a user interface, called Mosaic, was released.
- **1994:**
 - The lead developers of Mosaic founded a company called Netscape.
 - Netscape released a more polished browser called Netscape Navigator.
 - Netscape Navigator, despite its popularity, lacked dynamic behavior and interactivity for web pages, meaning pages were static after loading.
- **1995:**
 - To address the lack of interactivity, Netscape created a new scripting language called JavaScript. The name was chosen purely for marketing purposes, as Java was a popular language at the time.

- During the same year, Microsoft debuted its browser, Internet Explorer, which initiated a "browser war" with Netscape.
- **1996:**
 - Microsoft recognized that JavaScript significantly improved the user experience of the web and wanted to incorporate similar functionality into Internet Explorer. However, there was no specification for them to follow.
 - Microsoft reverse-engineered Netscape's Navigator interpreter to create its own scripting language, called *jscript*. While *jscript* served the same purpose as JavaScript, its implementation was different.
 - These differences between JavaScript and *jscript* created a problem for developers, making it difficult to ensure their websites worked well in both browsers. This led to common "best viewed in Netscape" or "best viewed in Internet Explorer" badges.
 - To resolve this inconsistency, in November 1996, Netscape submitted JavaScript to ECMA International.
 - *ECMA International*: This is an industry association dedicated to the standardization of information and communication systems. Netscape's goal was to establish a standard specification that all browser vendors could conform to, ensuring consistency across different browser implementations.
 - ECMA provides a standard specification and a committee for each new specification
 - For JavaScript, the standard is named ECMA-262

Summary:

- ECMA 262: "the language specification."
- ECMAScript: "a language that implements ekma 262."
- JavaScript: "basically ecmascript at its core but Builds on top of that."

Relationship between ECMAScript and JavaScript:

- ECMAScript refers to the standard language, while JavaScript is the language used in practice, which builds on top of ECMAScript

Javascript Engines.

The Purpose of a JavaScript Engine

At its core, a JavaScript engine is a program that converts JavaScript code, written by developers, into machine code that a computer can understand and execute. The JavaScript code we write cannot be understood by the computer. This conversion process allows computers to "perform specific tasks" based on JavaScript instructions. In essence, "a JavaScript engine can execute JavaScript code."

Major JavaScript Engines

JavaScript engines are typically developed by web browser vendors. Key examples include:

- **V8:** Developed by Google for Chrome (and central to Node.js).
- **SpiderMonkey:** Used by Firefox.
- **JavaScriptCore:** Developed by Apple for Safari.
- **Chakra:** Used in the original Microsoft Edge.

Google's V8 Engine:

Github: <https://github.com/v8/v8>

Key Characteristics:

Our primary interest lies in the V8 engine, which is foundational to Node.js.

- **Open Source:** V8 is "Google's open source JavaScript engine."
- **ECMAScript Implementation:** V8 "implements ecma script as specified in ecma 262." This means it adheres to the standard language specification for JavaScript.
- **Written in C++:** A surprising but crucial detail is that "V8 is written in C++ and is used in Google Chrome." This highlights that the engine itself is not written in JavaScript.
- **Standalone or Embeddable:** V8 can function independently to execute JavaScript code, or it "can be embedded into any c++ application." This embeddability is a critical feature that facilitates the creation of Node.js.

V8's Role in Node.js:

Extending JavaScript Functionality

The ability to embed V8 into other C++ applications is the core idea behind Node.js.

- **Custom C++ Program + V8:** By embedding V8, developers can "write C++ code that gets executed when a user writes JavaScript code." This effectively allows for the addition of new features to JavaScript itself.
- **Bridging JavaScript and Low-Level Operations:** C++ excels at "lower level operations like file handling, database connections and network operations." By integrating V8 with a C++ program (which is what Node.js essentially is), developers gain "the power to add all of that functionality in JavaScript."
- **Node.js as an Extension:** Node.js can be understood as a C++ program that embeds V8. This allows Node.js to "run ecma script and additional features that you choose to incorporate," particularly "features that are available in C++ but not available in JavaScript." This "is the idea behind node.js."

Overview of Node.js

Node.js is an open-source, cross-platform JavaScript runtime environment that allows JavaScript code to be executed outside a web browser. Introduced in 2009, Node.js revolutionized JavaScript development by expanding its capabilities beyond front-end browser-based applications.

Key Characteristics:

Open Source: "Its source code is publicly available for sharing and modification," fostering community collaboration and development.

Cross-Platform: Works seamlessly on various operating systems, including "Mac, Windows, and Linux."

JavaScript Runtime Environment: Provides "all the necessary components in order to use and run a JavaScript program."

Browser Independence: A crucial distinction is that "node.js can run a JavaScript program outside the browser," unlocking a "new world of possibilities."

What Can You Build with Node.js?

The ability to run JavaScript outside the browser significantly broadens the scope of applications that can be built. Node.js enables the creation of:

- Traditional websites
- Back-end services like APIs
- Real-time applications
- Streaming services (e.g., Netflix)
- Command Line Interface (CLI) tools
- Multiplayer games

In essence, "the bottom line is node.js allows you to build complex and powerful applications."

Inside the Node.js Runtime: A Code Representation

The Node.js runtime is not a monolithic "black box" but rather a combination of code written in C++ and JavaScript. Its source code is publicly available on GitHub (github.com/nodejs/node) and is organized into three primary folders:

deps (Dependencies Folder):

- Contains external code essential for Node.js's functioning.
- V8: This is the "JavaScript engine present in the Chrome browser." It is critical because "without V8 there is no way for node.js to understand the JavaScript code we write."

- libuv (UV): An "open source Library" that "provides node.js access to the underlying operating system features such as file system and networking."

src (Source Folder):

- Contains the C++ source code of the Node.js runtime.
- This folder addresses a fundamental limitation of JavaScript: "JavaScript as a language was not designed to deal with low level functionality like file system or networking."
- C++ provides "features such as file system handling and networking," which Node.js integrates.

lib (Library Folder):

- Contains JavaScript code that serves as a bridge, allowing JavaScript developers to easily access the low-level C++ features provided by Node.js.
- This eliminates the need for JavaScript developers to know C++. For example, fs.js contains JavaScript code for accessing the file system, which internally calls the corresponding C++ features that rely on libuv.
- This folder also includes "a few utility functions that you may need when writing code with node.js." This constitutes the "standard library of JavaScript code."

Browser Vs NodeJS

What is a Browser Environment?

Browser = Client-side environment

A browser provides:

- DOM APIs: document, window, localStorage
- UI interaction: alert, prompt, event listeners
- Network: fetch, XMLHttpRequest
- Timer APIs: setTimeout, setInterval

Use Case: User interaction, rendering UI, form validation, etc.

What is Node.js?

Node.js is a runtime environment that allows you to run JavaScript outside the browser.

Built on:

- V8 Engine (same engine as Chrome)
- libuv (provides async I/O via event loop)

Provides:

- File system access (fs)
- OS interaction (os)
- Network creation (http, net)
- Process control (process)
- CLI tools
- Module system (require, module.exports)

Use Case: Server-side apps, backend APIs, file operations, CLI tools, streaming servers, etc.

Key Reasons for Node.js:

- JavaScript on the server: Before Node, backend meant PHP, Java, Python, etc. Node enables full-stack JavaScript.
- Access to System Resources: Browsers sandbox JS for security (can't access disk, memory, OS). Node allows full system access.
- Build servers: You can build an HTTP/HTTPS server in Node without Apache/Nginx.
- npm: Huge ecosystem of packages and tools.
- Asynchronous & Fast: Thanks to the non-blocking event loop, Node is ideal for I/O-heavy tasks (like APIs).

Modules

In Node.js, a module is a file containing code (functions, variables, classes) that is executed in its own scope. It allows you to break your program into reusable pieces.

Each file in Node.js is a module.

Why Use Modules?

- Reusability: Write once, use anywhere.
- Separation of concerns: Keeps codebase modular.
- Maintainability: Easier to manage small modules.
- Encapsulation: Avoids polluting the global scope.

How Node.js Modules Work

Every file is wrapped in a function by Node.js internally:

```
(function(exports, require, module, __filename, __dirname) {  
  // Your module code  
});
```

This gives access to:

- exports: Object to export module content
- require(): To import other modules
- module: Contains metadata about the module
- __filename: Full path of current file
- __dirname: Directory path of current module

Types of Modules

Local Modules

A local module is a user-defined module created in your own project. It resides in your project's directory structure (unlike core or external modules).

Local modules are used to organize your code into separate reusable files.

How to Create a Local Module

Suppose you want a utility file for math operations:

Step 1: Create math.js

```
function add(a, b) {
```



```
    return a + b;
}

function sub(a, b) {
    return a - b;
}

// Export functions
module.exports = {
    add,
    sub
};
```

Step 2: Use it in another file

```
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data); 15
```

Built-in Modules

These are modules provided by Node.js itself. You don't need to install them via npm.

Examples:

- fs – File system operations
- http – Create HTTP servers
- path – File path manipulations
- crypto – Cryptographic operations
- os – Operating system info

Example: Using fs module

```
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);
```

 Core modules are loaded faster and cached automatically.

External Modules


These are modules created by others and shared via npm (Node Package Manager). You install them into your project and then import.

Steps:

```
npm install lodash
```

Example:

```
const _ = require('lodash');  
console.log(_.isEmpty({})); // true
```

 Use third-party modules for tasks like routing, validation, databases, logging, etc.

Global Modules


These are npm modules installed globally on your system (not specific to a project). They're usually CLI tools.

Install globally:

```
npm install -g nodemon
```

Use in terminal:

```
nodemon app.js
```

 Global modules aren't used with `require()` in code; they're used via the command line.

JSON Modules

Node.js can import .json files as modules directly.


Example:

config.json

```
{  
  "port": 3000,  
  "env": "development"  
}
```

app.js

```
const config = require('./config.json');  
console.log(config.port); // 3000
```

 Node parses the JSON and gives you a JS object.

Module Caching

What is Module Caching?

When you import a module using `require()`:

- Node.js loads and executes the module only once.
- After that, it caches the exported result.
- Any future `require()` of the same module returns the cached version, not a re-executed one.

How Caching Works: Under the Hood

When you call:

```
const lib = require('./lib');
```

Node.js:

Resolves the path of `'./lib'`

Checks if it's in `require.cache`

- If not cached:
- Loads and runs the file
- Stores it in `require.cache`

If cached:

- Returns the already exported object

Caching Example

```
//File: logger.js
console.log("Logger module loaded");

module.exports = function log(msg) {
  console.log("Log:", msg);
};
```

```
//File: app.js
const log1 = require('./logger');
log1('Hello');

const log2 = require('./logger'); // Will NOT re-run logger.js
log2('World');
```

```
//Output:
Logger module loaded
Log: Hello
Log: World
```

"Logger module loaded" only runs once – due to caching

Can You Clear the Cache?

- Yes – by manually deleting from require.cache:

```
delete require.cache[require.resolve('./logger')];
```

- Then re-require:

```
const logger = require('./logger'); // Reloaded
```

Why Is Caching Useful?

- Boosts performance (avoids reloading every time)
- Keeps singleton state (e.g., DB connection, config)
- Good for shared state (e.g., counters, cache)

Circular Dependency

What is a Circular Dependency?

A circular dependency happens when:

- Module A requires Module B
- AND Module B requires Module A

This creates a loop that can lead to:

- Incomplete module exports
- Unexpected behavior

Circular Dependency Example

File: a.js

```
const b = require('./b');
console.log('a.js loaded');
module.exports = 'I am A';
```

```
File: b.js
const a = require('./a');
console.log('b.js loaded');
console.log('a from b:', a); // a will be {} (incomplete)
module.exports = 'I am B';
```

File: main.js

```
require('./a');
```

Output:

```
b.js loaded
a from b: {}
a.js loaded
```

a is {} in b.js because it wasn't fully exported yet — it was still being initialized


Why This Happens

Node caches partial exports during the first pass:

If Module A is not fully executed, but Module B tries to require('./a'), it gets an incomplete object ({})

How to Solve: Lazy require()

File: a.js

```
function getB() {  
  const b = require('./b'); //  Lazy require inside a function  
  return b;  
}  
  
console.log("In a.js, b =", getB().valueFromB);  
  
module.exports = {  
  valueFromA: "This is A",  
  getB  
};
```

File: b.js

```
function getA() {  
  const a = require('./a'); // Lazy require inside a function  
  return a;  
}  
  
console.log("In b.js, a =", getA().valueFromA);  
  
module.exports = {  
  valueFromB: "This is B",  
  getA  
};
```

Output:

In b.js, a = This is A

In a.js, b = This is B

Both modules work fine because the require() happens after exports are ready.

Another Example:

Imagine this:

- database.js exports a function that connects to the DB.
- service.js uses that DB connection.
- But then, database.js also wants to call a function from service.js.

So you end up with this:

database.js → requires service.js

service.js → requires database.js

This is a circular dependency! If you do it the normal way, it breaks.

Problematic Version (Direct require())

database.js

```
const service = require('./service'); // ❌ Circular dependency (top-level)
function connect() {
  console.log('Connecting to DB...');
  service.sayHello(); // ← uses service.js
}
module.exports = { connect };
```

service.js

```
const db = require('./database'); // ❌ Circular dependency (top-level)
function sayHello() {
  console.log('Hello from service');
}
db.connect(); // ← uses database.js

module.exports = { sayHello };
```

Output:

```
TypeError: service.sayHello is not a function
```

Why? Because:

service.js is not fully exported when database.js tries to use it.

That's the circular dependency problem.

Solution: Use Lazy require() Inside the Function

Instead of requiring service.js at the top of database.js, do it inside the function, after all exports are ready.

Fixed Version Using Lazy require()

database.js

```
function connect() {  
  const service = require('./service'); // Lazy require: happens at  
  runtime  
  console.log('Connecting to DB...');  
  service.sayHello(); // works fine now  
}  
  
module.exports = { connect };
```

service.js

```
const db = require('./database'); // still required at top  
function sayHello() {  
  console.log('Hello from service');  
}  
  
db.connect(); // calls the function  
  
module.exports = { sayHello };
```

Output:

```
Connecting to DB...  
Hello from service
```

Now it works because:

- The first time require('./database') is called, it exports connect.
- When connect() is called later, it then requires service.js.
- At that time, service.js is fully initialized, and its exports (sayHello) are ready.

CommonJS Vs EcmaScript

| Feature | CommonJS | ES6 Modules (ESM) |
|-----------------------|------------------------------------|-----------------------------------|
| Syntax | require, module.exports | import, export |
| File Extension | .js | .mjs or .js with "type": "module" |
| Load Type | Synchronous | Asynchronous (non-blocking) |
| EnvironmentNode.js | (default) | Browser & Node.js (modern) |
| Top-level await | Not supported | Supported |
| Circular Dependency | Loads partially initialized module | Handles it better |
| Default Export Syntax | module.exports = ... | export default ... |
| Named Export | exports.name = ... | export const name = ... |
| Mutable exports | Yes | Exports are read-only |
| Tree-shaking | No | Yes (important for bundlers) |

Tree shaking is the process of removing unused code (a.k.a. "dead code elimination") during the build process, especially from ES6 modules.

- It's called tree shaking because your code is treated like a tree:
- The parts you use are like branches that stay.
- The unused parts are "shaken off" and removed from the final bundle.

Types of Imports and Exports(CommonJS)

Node.js uses the CommonJS module system by default.

1. module.exports — Default Export

Export a single value, object, class, or function.

Export:

```
// logger.js
module.exports = function (msg) {
  console.log(`Log: ${msg}`);
};
```

Import:

```
const logger = require('./logger');
logger('This is a log');
```

You can also export an object directly:

```
module.exports = {
  log: function (msg) {
    console.log('Log:', msg);
  },
  warn: function (msg) {
    console.warn('Warning:', msg);
  }
};
```

2. exports — Named Export (Shorthand)

Export multiple functions or variables.

Export:

```
// math.js
exports.add = (a, b) => a + b;
exports.sub = (a, b) => a - b;
```

This is a shorthand for `module.exports.add = ...`

Import:

```
const math = require('./math');
console.log(math.add(2, 3));
```

Important Rule:

You should NOT assign a new value directly to `exports`. This will break the link to `module.exports`.

```
exports = function () { }; // won't work
```

Correct:

```
module.exports = function () { }; // works
```

Behind the Scenes: What Happens Internally?

Node.js wraps your module like this:

```
(function (exports, require, module, __filename, __dirname) {  
  // your code here  
});
```

That's why exports is a reference to module.exports.

If you reassign exports = ..., it no longer points to module.exports.

ES6 Modules (Optional in Node.js)

If you're using "type": "module" in package.json or .mjs extension, you can use ES6 imports/exports.

1. Export (Named + Default)

```
// math.mjs  
export const add = (a, b) => a + b;  
export const sub = (a, b) => a - b;  
  
export default function greet() {  
  console.log("Hello");  
}
```

2. Import

```
import greet, { add, sub } from './math.mjs';  
greet();  
console.log(add(3, 2));
```

fs Module (File System Modules)

The fs (File System) module is a core Node.js module that lets you interact with the file system — like reading, writing, updating, deleting files and directories.

Importing fs Module

```
const fs = require('fs'); // CommonJS
// OR (if using ES Module)
// import fs from 'fs';
```

Categories of fs Methods

| Method Style | Description |
|---------------|---|
| Sync | Blocking (stops further execution until done) |
| Async | Non-blocking (uses callbacks or Promises) |
| Promise-based | Cleaner async via fs.promises |

File Operations

Read File

1. Asynchronous (non-blocking)

```
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

2. Synchronous (blocking)

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

Write File

1. Asynchronous

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

2. Synchronous

```
fs.writeFileSync('example.txt', 'Hello again!');
```

Append to File

```
fs.appendFile('example.txt', '\nNew line added!', (err) => {  
  if (err) throw err;  
});
```

Delete File

```
fs.unlink('example.txt', (err) => {  
  if (err) throw err;  
  console.log('File deleted!');  
});
```

Directory Operations

Create Directory

```
fs.mkdir('newFolder', (err) => {  
  if (err) throw err;  
});
```

Read Directory

```
fs.readdir('./', (err, files) => {  
  if (err) throw err;  
  console.log(files); // Array of filenames  
});
```

Delete Directory

```
fs.rmdir('newFolder', (err) => {  
  if (err) throw err;  
});
```

Other Useful Methods

Check if File Exists

```
fs.access('file.txt', fs.constants.F_OK, (err) => {  
  console.log(err ? 'File does NOT exist' : 'File exists');  
});
```

Get File Info (stats)

```
fs.stat('example.txt', (err, stats) => {  
  if (err) throw err;  
  console.log(stats.isFile());      // true  
  console.log(stats.size);          // size in bytes  
  console.log(stats.birthtime);     // creation time  
});
```

Promises-based fs.promises

```
const fsPromises = require('fs').promises;  
  
async function readFileAsync() {  
  try {  
    const data = await fsPromises.readFile('example.txt', 'utf8');  
    console.log(data);  
  } catch (err) {  
    console.error(err);  
  }  
}  
readFileAsync();
```

Watch File for Changes

```
fs.watch('example.txt', (eventType, filename) => {  
  console.log(`File changed: ${filename}`);  
});
```

File Operations

| Operations | Method |
|----------------|----------------------------------|
| Read file | fs.readFile, fs.readFileSync |
| Append to file | fs.appendFile, fs.appendFileSync |
| Write to file | fs.writeFile, fs.writeFileSync |
| Delete file | fs.unlink, fs.unlinkSync |

Directory Operations

| Operations | Method |
|---------------|------------|
| Create Folder | fs.mkdir |
| Read Folder | fs.readdir |
| Delete Folder | fs.rmdir |
| Open Folder | fs.opendir |

Others

| Task | Method |
|---------------------|----------------------------|
| File exists check | fs.access |
| File metadata | fs.stat |
| Watch file changes | fs.watch |
| Promise-based usage | fs.promises.readFile, etc. |