**🌐 How require() Works Behind the Scenes in Node.js (Step-by-Step)**

This is how Node.js handles the require() function when you import or load another file or module.

Let’s break it into **5 stages**:

**✅ Step 1: Resolving the Module Path**

**📦 What does this mean?**

When you write something like:

js

CopyEdit

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

Node.js first **tries to figure out what file you're talking about**.

## ⚙️ Step-by-Step: What Happens When You Write require('something')

Node checks in **this exact order** ⬇️

## 1️⃣ Is it a ****Local File****?

If you write something like:

js

CopyEdit

require('./math')

Node will try to find a matching **file**:

bash

CopyEdit

✔️ ./math.js

✔️ ./math.json

✔️ ./math.node ← (compiled C++ addon)

📌 If any of these exist, Node loads and evaluates it.

## 2️⃣ Is it a ****Directory****?

If ./math is a **folder**, Node checks for an **entry file inside it**:

pgsql

CopyEdit

✔️ ./math/index.js

✔️ ./math/index.json

✔️ ./math/index.node

✅ This allows a folder to act like a single module — as long as it has an index.\* file.

## 3️⃣ Is it a ****Package from**** node\_modules?

If the path **doesn’t start with ./, ../, or /**, Node treats it as a **third-party package**.

Example:

js

CopyEdit

require('express');

Node will now search:

bash

CopyEdit

✔️ ./node\_modules/express

✔️ ../node\_modules/express

✔️ ../../node\_modules/express

✔️ ... (keeps going up until root)

📦 This is how Node loads installed npm packages from the nearest node\_modules directory.

📌 **This process is called "Module Resolution"**.

**Module Resolution** is the process Node.js uses to figure out **where the file/module you wrote inside require() actually exists** on your file system.

✅ Step 2: Loading the Module — What Happens After Node Finds the File?

Once Node **resolves the path** (Step 1), it now needs to **load and understand** what’s inside the file.

But every file is not the same — so Node handles them **differently based on file type**.

**🔧 Here's how it works for each file type:**

**1️⃣ .js file → Treated as JavaScript**

Let’s say:

js

CopyEdit

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

📦 Node will:

* Read the content as **plain JavaScript code**
* Wrap it in a function (IIFE)
* Run it using the V8 JavaScript engine
* Collect the result from module.exports

✅ This is the **most common** case.

**2️⃣ .json file → Parsed using JSON.parse()**

Let’s say:

js

CopyEdit

const data = require('./config.json');

📦 Node will:

* Read the file as a string: '{ "port": 3000 }'
* Use JSON.parse() behind the scenes to convert it into an object:

js

CopyEdit

{ port: 3000 }

✅ You can now use it directly like a JavaScript object!

**3️⃣ .node file → Compiled C++ Addon (Advanced Use Case)**

Let’s say:

js

CopyEdit

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

📦 This is:

* A **compiled binary file**, usually written in C++
* Used to extend Node.js with native performance
* Loaded using Node’s internal C++ bindings

❗You won’t use this unless you’re doing **very low-level or performance-heavy** work (e.g. image processing, system-level drivers)

**✅ Step 3: Wrapping the Module (VERY important)**

This is a key behind-the-scenes magic.

### 📦 What Actually Happens When You Write:

js

CopyEdit

require('./math');

### Behind the scenes, Node does something like:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// 👇 Your actual math.js module code

const add = (a, b) => a + b;

module.exports = add;

})(...); // Node automatically injects the arguments

🧠 This is called an **IIFE** — Immediately Invoked Function Expression.

**✅ Why does Node wrap your code?**

# ✅ Reason 1: To Keep Variables Private (Avoid Global Conflicts)

## 💡 What it means:

If Node didn’t wrap your code in a function, every variable you declare in a file would go into the **global scope** (shared space).

This could cause **errors** if two files use the same variable name.

## 💣 Problem (If Node Did NOT Wrap the Code):

Imagine this:

### math.js

js

CopyEdit

const x = 100;

### index.js

js

CopyEdit

const x = 200;

require('./math');

console.log(x);

❌ Without wrapping, both files would share the same space.

Since both use const x, it will throw:

vbnet

CopyEdit

❌ Error: Identifier 'x' has already been declared

This happens because const and let don’t allow the same name twice in the same scope.

## ✅ What Node Actually Does:

Node wraps every file like this behind the scenes:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// Your file's code

})();

So now:

* The x in math.js is **private to that file**
* The x in index.js is **private to this file**

✅ No conflict, no overwrite, no error

**Note – var is overrrideeren (no error) without IIFE**

# ✅ Reason 2: To Inject Useful Tools into Every File

(Explained with real custom files and code)

## 🧠 Why Node Injects Things:

Node wraps your file like this behind the scenes:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// Your file code goes here

})();

This is **how your file magically gets access to**:

* require
* module.exports
* \_\_dirname
* \_\_filename

## 📁 Folder Setup

lua

CopyEdit

project/

├── index.js

├── utils/

│ └── math.js

## 🧪 Let’s Write Real Code in These Files:

### ✅ utils/math.js

js

CopyEdit

console.log('--- math.js running ---');

console.log('Inside math.js');

console.log('require:', typeof require); // function

console.log('module.exports:', module.exports); // empty object

console.log('\_\_dirname:', \_\_dirname); // /full/path/project/utils

console.log('\_\_filename:', \_\_filename); // /full/path/project/utils/math.js

module.exports = function add(a, b) {

return a + b;

};

### ✅ index.js

js

CopyEdit

console.log('--- index.js running ---');

const add = require('./utils/math'); // Triggers math.js

console.log('Result:', add(5, 7)); // 12

## 🧠 What You’ll See When You Run:

bash

CopyEdit

node index.js

### ✅ Output:

lua

CopyEdit

--- index.js running ---

--- math.js running ---

Inside math.js

require: function

module.exports: {}

\_\_dirname: /Users/likan/project/utils

\_\_filename: /Users/likan/project/utils/math.js

Result: 12

## 🔍 What This Proves:

| **Variable** | **Who injects it?** | **What it does** |
| --- | --- | --- |
| Require | Node | Used to load other files/modules |
| module.exports | Node | Used to send values out of a file |
| \_\_dirname | Node | Gives folder path of current file |
| \_\_filename | Node | Gives full path of current file |

✅ These values were **not declared by us** — they were **injected by Node** when wrapping the file!

## 💬 Final Interview-Friendly Explanation:

Node wraps each file in a function so it can pass in helpful tools like require, module.exports, \_\_dirname, and \_\_filename. These are not global — they’re scoped to the module only — and they help us build modular, file-based applications easily.

**✅ Step 4: Code Evaluation & module.exports**

### 🔧 Recap of where we are so far:

✔️ Node resolved the file (e.g. ./math.js)  
✔️ It loaded the content  
✔️ It wrapped it in a function like:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// Your math.js code

})();

## ✅ Now comes: ****Code Evaluation****

That means Node **runs the function** (the wrapped code) using the **V8 JavaScript Engine** — same engine as in Chrome.

So all your code inside math.js starts executing normally.

## 🧠 What Happens During Execution?

Let’s say your math.js file looks like:

js

CopyEdit

function add(a, b) {

return a + b;

}

function subtract(a, b) {

return a - b;

}

module.exports = {

add,

subtract

};

📌 While executing this:

* Node reaches the line module.exports = {...}
* It assigns that object (containing add and subtract) to module.exports

✅ This is what **gets sent back** to the file that used require().

## 💡 So when you write in another file:

### index.js

js

CopyEdit

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

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

console.log(math.subtract(5, 2)); // 3

### What’s happening here:

* require('./math') triggers math.js
* Node runs math.js
* Whatever you set as module.exports in math.js becomes the **return value**
* That value is stored inside const math

✅ So now math.add and math.subtract are functions you can call.

## 💬 Final Interview One-Liner:

When Node runs your wrapped file, it executes all code and returns whatever you assigned to module.exports. That’s the value you receive when using require() in another file.

# ✅ Step 5: Caching the Module in Node.js

Short Theory + Code Examples for both **Performance** and **Consistency**

## 🔁 What Is Caching?

Once a module is loaded using require(), Node stores it in memory — and reuses it for future require() calls.

## ✅ Reason 1: ****Performance****

### 📌 Pin-Point Theory:

* Module is executed only **once**
* Future require() calls return the **cached result**
* ✅ Speeds up execution (no repeated file reads)

### 🧪 Code Proof:

#### log.js

js

CopyEdit

console.log('🔁 log.js executed');

#### app.js

js

CopyEdit

require('./log');

require('./log');

require('./log');

### ✅ Output:

lua

CopyEdit

🔁 log.js executed

Only logs once → shows that module is **not re-executed**.

## ✅ Reason 2: ****Consistency****

### 📌 Pin-Point Theory:

* All files get the **same module instance**
* Good for **shared state** (e.g., counters, DB, config)
* ✅ Prevents duplication

### 🧪 Code Proof:

#### counter.js

js

CopyEdit

let count = 0;

module.exports = {

add: () => ++count,

get: () => count

};

#### a.js

js

CopyEdit

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

counter.add();

console.log('From A:', counter.get()); // 👉 1

#### b.js

js

CopyEdit

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

console.log('From B:', counter.get()); // 👉 1 ✅ Same instance

#### index.js

js

CopyEdit

require('./a');

require('./b');

### ✅ Output:

css

CopyEdit

From A: 1

From B: 1

Both use **same cached object** — count is shared.

## 💬 Final Interview Line:

Node caches modules to boost performance (load once, reuse) and ensure consistency (same shared instance across files).

**🎯 Real-World Analogy (Super Simple)**

Imagine you're in a kitchen (Node.js), and you want to use a "spice mix" (module).

**Step-by-step:**

1. 🔍 You look for the spice mix in your drawer or shelf (./spices, node\_modules/spices)
2. 📖 You open the container (read the file)
3. 📦 You put the spice in a small bowl for use (wrap it in a function)
4. 🧂 You only take the mix that’s needed for this dish (use exports)
5. 🧊 You store the bowl in the fridge for reuse later (caching)

# ✅ What is require.cache and How Caching Works (Super Easy)

## 🎬 Imagine this:

You wrote this code:

js

CopyEdit

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

Node has to go and get the math.js file and run it.

But Node is smart. It thinks:

“Hmm, if this same file is needed again later… I should remember it!”

So it does this:

## 🔁 Step-by-Step Story of Caching

### ✅ Step 1: Node looks for the file

It finds: math.js

### ✅ Step 2: Node runs the file

It reads it and executes it (top to bottom).

js

CopyEdit

console.log('math.js ran');

module.exports = { value: 42 };

### ✅ Step 3: Node remembers it in require.cache

Now Node saves this file in a memory box called require.cache

js

CopyEdit

require.cache['/full/path/math.js'] = {

exports: { value: 42 },

loaded: true

}

So later, if someone asks for this same file again...

### ✅ Step 4: Next time you write require('./math')

Node doesn’t load it again.  
Instead, it says:

“Already have it saved in my cache. I’ll just return that!”

✅ So it’s fast.  
✅ It doesn't run the file again.  
✅ You get the same result.

## 🧹 What if you delete the cache?

You can tell Node:

js

CopyEdit

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

Now Node forgets the file.

### 🤔 What happens next?

If you write require('./math') again...

Node says: “Oh! It’s gone! I’ll load and run it from scratch again.”

✅ So yes — **it will start from the beginning**:

1. Module resolution
2. Load the module
3. Wrap the module
4. Code execution
5. Cahce again

## 🧠 Real Example (Now with Cache Log):

### 📄 math.js

js

CopyEdit

console.log('📦 math.js is running');

module.exports = { value: 10 };

### 📄 app.js

js

CopyEdit

// First load

require('./math');

// Log what's inside the cache

console.log('\n🧠 Current cache:');

console.log(require.cache[require.resolve('./math')]);

// Clear cache

console.log('\n🧹 Deleting cache...');

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

// Load again after deletion

console.log('\n🔁 Loading again after deleting cache:');

require('./math');

// Show cache again

console.log('\n📦 Cache after reload:');

console.log(require.cache[require.resolve('./math')]);

### ✅ Output (example):

yaml

CopyEdit

📦 math.js is running

🧠 Current cache:

{

id: '/your/path/math.js',

exports: { value: 10 },

loaded: true,

...

}

🧹 Deleting cache...

🔁 Loading again after deleting cache:

📦 math.js is running

📦 Cache after reload:

{

id: '/your/path/math.js',

exports: { value: 10 },

loaded: true,

...

}

✅ As you can see:

* File runs only once ✅
* Cache holds the result ✅
* Deleting cache makes Node reload ✅

## 💬 Final One-Liner:

require.cache is how Node remembers your file after the first require(). If you delete it, Node forgets and loads the file again.

# ✅ How require() Works in Node.js (Super Simple)

### 🧠 Think of require() like ordering a pizza 🍕

You tell Node:

js

CopyEdit

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

And Node thinks:

“Alright! Let me go find this file and bring you what it exports.”

## ✅ Step-by-Step (Plain English)

### 1️⃣ ****Find the file****

Node looks for:

* ./math.js
* or ./math/index.js (if it's a folder)

### 2️⃣ ****Read the file****

Node reads the entire file like plain text.

### 3️⃣ ****Wrap the code****

Node secretly wraps your file inside a function like this:

js

CopyEdit

(function(exports, require, module, \_\_filename, \_\_dirname) {

// your code here

})();

This gives you access to things like:

* require
* module.exports
* \_\_dirname

### 4️⃣ ****Run the file****

Node runs the code inside your file.

### 5️⃣ ****Save the result****

Whatever you exported using module.exports, Node saves it in memory (require.cache) 💾

### 6️⃣ ****Return the export****

Now your variable gets that result.

js

CopyEdit

const math = require('./math'); // math = whatever math.js exported

### ✅ Example

#### math.js

js

CopyEdit

console.log('math.js is running');

module.exports = { add: (a, b) => a + b };

#### app.js

js

CopyEdit

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

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

🟢 First time: file runs  
🟢 Next time: file doesn’t run — Node returns from memory

## 💬 One-Line Summary:

require() finds your file, runs it once, saves the result, and gives it to you every time you ask for it.

# 📦 Node.js Module Wrapper Function – How the Second () Works

## 🧠 What's the Big Idea?

When you require('./math'), Node doesn't just run math.js directly.

Instead, Node:

1. **Wraps** the entire file code in a special function
2. **Immediately invokes** that function and passes 5 useful values

This pattern is called an **Immediately Invoked Function Expression (IIFE)**.

## 🔄 The Internal Structure (Behind-the-Scenes)

Here’s what **Node really does** when loading any file:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// 👇 Your actual file content

const add = (a, b) => a + b;

module.exports = add;

})(

module.exports, // 🔹 The actual exports object for the file passed by node

require, // 🔹 Node's require function passed by node

module, // 🔹 The module object Node createsand is being passed by node

\_\_filename, // 🔹 Full file path (e.g., /Users/likan/math.js)passed by node

\_\_dirname // 🔹 Directory path (e.g., /Users/likan) passed as an arg by node

);

## 📦 What is module?

It is a special **object** created by Node for **every file**.

🧠 This object helps Node:

* Track what the file is exporting
* Know the full path of the file

So every .js file you load using require() gets its own **module object**.

## ✅ Example: When you run node math.js

Node internally does this:

js

CopyEdit

const module = {

exports: {}, // ✅ empty object to hold exports

filename: '/full/path/math.js', // 📍 path to the file

id: '.', // ID for the main file

loaded: false // not loaded yet

};

## ✅ Then Node wraps your code like:

js

CopyEdit

(function (exports, require, module, \_\_filename, \_\_dirname) {

// 👉 your real code here

})(

module.exports, // ✅ first arg (empty object)

customRequireFunction, // ✅ second arg (require)

module, // ✅ third arg (the whole object)

module.filename, // ✅ fourth arg

path.dirname(filename) // ✅ fifth arg

);

So inside your file, when you write:

js

CopyEdit

console.log(module);

You’ll see the full object Node created!

## 🧠 Why is module important?

* It tells Node what to return when someone does:

const result = require('./math'); // ← this returns module.exports

# 🧠 Question: When does module object (especially module.exports) change?

### 🤔 First understand:

When Node runs a file, it secretly does this 👇:

js

CopyEdit

const module = {

exports: {} // ✅ starting point

};

const exports = module.exports; // ✅ both point to same object

## ✅ Case 1: You add to exports

js

CopyEdit

exports.name = "Likan";

👉 Behind the scenes:

js

CopyEdit

module.exports = { name: "Likan" };

✅ So require() will return: { name: "Likan" }. Reqires always returns module.exports not exports .. esports is useful while adding proeprties

No problem! This is good.

## ❌ Case 2: You overwrite exports

js

CopyEdit

exports = "hello";

👉 Behind the scenes:

* exports now points to a new string "hello"
* But module.exports is **still the old object**

⛔ So require() will **not** return "hello" — it still returns {} or previous data.

❌ Bad: link is broken.

## ✅ Case 3: You overwrite module.exports

js

CopyEdit

module.exports = "✅ this is final";

👉 Now require() will return "✅ this is final"

✅ This is 100% clear and works perfectly.

In **Node.js (CommonJS)**:

exports is just a **shorthand (alias)** for module.exports.  
But they both **reference the same object only until you reassign exports**.

**✅ What you CAN do:**

js

CopyEdit

// Adding properties to exports (works fine)

exports.greet = () => console.log("Hello");

exports.sum = (a, b) => a + b;

✅ This works because you're **mutating** the same object that module.exports points to.

**❌ What you CANNOT do:**

js

CopyEdit

// Reassigning exports directly (does NOT work)

exports = {

greet: () => console.log("Hello")

};

🚫 This breaks the link between exports and module.exports, so nothing gets exported!

**✅ What you should do instead:**

js

CopyEdit

// Correct full assignment

module.exports = {

greet: () => console.log("Hello")

};

**🧠 Key One-Liner (Interview-Friendly):**

exports is a **reference** to module.exports, so you can add properties to it,  
but if you **reassign exports directly**, it **breaks the reference**, and nothing will be exported.