

CommonJS (CJS) and ECMAScript Modules (ESM) have coexisted for years, causing confusion and technical headaches. Their split has led to compatibility issues, inconsistent tooling, and frustrating workarounds.

You've likely dealt with libraries that support only one format, leading to mismatched imports, runtime errors, and complex build setups. Even within Node.js, interoperability remains a challenge.

This article breaks down the differences between CJS and ESM, giving you a clear understanding of when and how to use each module system—without the headaches.

## The evolution of JavaScript modules

JavaScript was originally a simple scripting language without built-in modularity. As the language matured and developers began building large applications, it became clear that a standardized way to manage code organization and dependencies was necessary.

In the early days, developers used techniques like the module pattern and namespaces to organize code, but these solutions were limited and difficult to scale. Eventually, two major module systems emerged:

- CommonJS, designed primarily for server-side JavaScript, particularly in Node.js.
- ECMAScript Modules, introduced as the official JavaScript standard for browsers and servers.

While both serve the purpose of modularity, they are fundamentally different in their design, syntax, and execution. Let's explore each module system before diving into their differences.

## CommonJS

CommonJS was introduced in 2009, Kevin Dangoor as an effort to create a standardized module system for JavaScript outside the browser. Shortly

after, **Node.js adopted CommonJS**, making it the default module system for server-side JavaScript.

**CommonJS modules are synchronous, meaning they execute and load files in a blocking manner.** This approach works well in server environments where files are loaded from the local filesystem but **can become problematic in browsers, where asynchronous execution is preferred.**

The following are some of the key features of CommonJS: - **Uses** `require()` **to import modules.** - **Uses** `module.exports` **or** `exports` **to export functions, objects, or variables.** - Supports caching of required modules.

Here's how you can export a function using CommonJS with `module.exports` :

addTwo.js

```
function addTwo(num) {  
  return num + 2;  
}  
module.exports = { addTwo };
```

Importing the function in another file can be done like this:

main.js

```
const { addTwo } = require('./addTwo.js');  
console.log(addTwo(4)); // Prints: 6
```

Despite its popularity in Node.js, **CommonJS has no static analysis, meaning tools cannot optimize code by removing unused functions (no tree shaking).** It also **relies on synchronous execution, which is inefficient for browser-based applications.**

**Tree shaking is a process of dead code elimination, where unused code is removed from the final bundle. It relies on the static structure of ES6 modules (import and export statements) to determine which code is actually used and which can be safely discarded.**

These drawbacks led to the development of ECMAScript modules, which offers better performance and different syntax.

# ECMAScript modules

ECMAScript modules were introduced in ES6 (2015) as the official JavaScript module system. Unlike CommonJS, ESM is **designed for both browsers and servers**, providing a unified standard for JavaScript modularity.

The following are some of the key features of ESM:

- Uses `import` and `export` instead of `require()` and `module.exports`.
- Enables static analysis, allowing tree shaking (removal of unused code).
- Supports top-level `await` - Requires explicit file extensions in relative imports (`.mjs`, `.js`, or `.cjs`).

To use ESM in Node.js, update `package.json` by adding `"type": "module"`.

This ensures that `.js` files are treated as ES modules. If you prefer not to modify the project-wide setting, rename files to `.mjs` instead.

package.json

```
{  
  "type": "module"  
  ...  
}
```

Here's how you can export a function using ESM with the `export` keyword:

addTwo.mjs

```
export function addTwo(num) {  
  return num + 2;  
}
```

Importing the function in another file can be done like this:

```
// main.mjs (ESM)  
import { addTwo } from './addTwo.mjs';  
console.log(addTwo(4)); // Prints: 6
```

Despite its advantages, ESM adoption in Node.js has been slow because many existing projects were built using CommonJS. However, **Node.js now supports both module systems**, allowing you to transition from CommonJS to ESM gradually.

## CommonJS vs ES Modules

With the overviews of both module systems covered, let's now examine the major differences between CommonJS (CJS) and ECMAScript Modules (ESM).

Feature	CommonJS (CJS)	ECMAScript Modules (ESM)
Loading	Synchronous	Asynchronous (better for browsers)
Syntax	<code>require()</code> / <code>module.exports</code>	<code>import</code> / <code>export</code>
Tree shaking	✗ No	✓ Yes (Static Analysis)
Browser support	✗ Requires bundlers	✓ Native browser support
Performance	Slower (blocking)	Faster (async + tree-shaking)
Use case	Server-side (Node.js)	Both server & browser
Top-level <code>await</code>	✗ No	✓ Yes (since Node.js v14.8.0)
<code>__dirname</code> & <code>__filename</code>	✓ Available	✗ Not available (Use <code>import.meta.url</code> )
File extensions	Optional ( <code>.js</code> assumed)	Required ( <code>.mjs</code> , <code>.js</code> , <code>.cjs</code> )
JSON imports	<code>require('./file.json')</code>	<code>import jsonFile from './file.json' with { type: 'json' }</code> (since Node.js v17.1.0)
Dynamic imports	<code>require()</code> only	<code>import()</code> supported in both CJS and ESM
Built-in module imports	<code>require('fs')</code>	<code>import fs from 'node:fs'</code> (since Node.js v12.20.0)
Module caching	✓ <code>require.cache</code>	Separate cache (no <code>require.cache</code> )
<code>require.resolve()</code>	✓ Yes	✗ No ( <code>import.meta.resolve()</code> in Node.js v20.11.0)

## Top-level await

ESM supports top-level `await`, allowing you to use `await` outside an `async` function, which is impossible in CommonJS.

```
const data = await fetch('https://api.example.com/data');
console.log(await data.json());
```

## Synchronous vs. asynchronous execution

One major difference is that **CommonJS is synchronous**, while **ESM is asynchronous**. This means `require()` can be used anywhere in the code, but `import` must be declared at the top level.

`__dirname` and `__filename`

CommonJS provides built-in globals:

```
console.log(__dirname);
console.log(__filename);
```

However, in ESM, these are not available, and you must use `import.meta.url`:

```
import { fileURLToPath } from 'node:url';
import { dirname } from 'node:path';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
console.log(__dirname);
console.log(__filename);
```

## Module resolution

CommonJS allows features like `require.cache` and `require.extensions`, which do not exist in ESM. Instead of `require.resolve()`, ESM provides `import.meta.resolve()`, introduced in Node.js v20.11.0:

```
const resolvedPath = import.meta.resolve('./someFile.js');
console.log(resolvedPath);
```

## JSON imports

CommonJS allows direct `require()` for JSON files:

```
const packageData = require('./package.json');
console.log(packageData);
```

In ESM, JSON files must be explicitly imported with a type attribute, a feature introduced in Node.js v17.1.0:

```
import packageData from './package.json' with { type: 'json' };
console.log(packageData);
```

## Transitioning from CommonJS to ESM in Node.js

With Node.js supporting ECMAScript modules alongside CommonJS (CJS), **you should always use ESM in new projects as it is the modern standard.**

For existing projects using CommonJS, you can slowly transition to ESM. However, migrating a project from CommonJS to ESM involves challenges of changing imports and exports, handling CommonJS-specific features, and ensuring compatibility with dependencies.

So it is important to do it gradually and safely. This section will help you do that. For a smooth experience, you must ensure you have the latest version of Node.js, which is 22 (LTS) at the time of writing.

## Using ESM modules in CommonJS projects

To easily use ESM, you can slowly create ESM modules and import them in the CommonJS module files.

CommonJS does not support static `import`, but you can use dynamic `import()` inside an async function:

```
async function loadModule() {
  const { addTwo } = await import('./addTwo.mjs'); // dynamic import
  console.log(addTwo(3));
}
loadModule();
```

Starting from **Node.js 23**, ES modules can be loaded using `require()` without throwing `ERR_REQUIRE_ESM`, but only if the module meets certain conditions:

- The file has a `.mjs` extension or is marked as `"type": "module"`
- The module **does not use** top-level `await`

Here is an example:

```
const esmModule = require("./esm-file.mjs"); // Works in Node.js 23+
console.log(esmModule);
```

This feature is experimental and can be disabled with `--no-experimental-require-module`.

With this, you can gradually transition to ESM while ensuring compatibility with existing CommonJS code.


## Using CommonJS modules in ESM projects

If you are using ESM already, but need to use a CommonJS module or dependencies that only support CommonJS, you can use `createRequire()` from `node:module` [↗](#):

```
Here is how you can do that: javascript import { createRequire } from
"node:module"; const require = createRequire(import.meta.url); const
somePackage = require("some-package");
```

This allows you to use CommonJS modules within an ESM environment.

Alternatively, you can dynamically import a CommonJS module:



```
const cjsModule = await import('./commonjs-file.cjs');
console.log(cjsModule.default);
```

Dynamic `import()` works inside ESM but must be used inside an `async` function or top-level `await`.

With that, you should be able to safely transition to ESM without disrupting existing functionality or compatibility with CommonJS dependencies.

## Final thoughts

This article explored the key differences between CommonJS and ECMAScript Modules. Hopefully, you now understand how each module system works, what they offer, and how to use them effectively.

If you're working on a new project, ESM is the better choice—it's more flexible, aligns with modern JavaScript standards, and ensures better compatibility for the future.

Thanks for reading—happy coding!