SECTION 3 — JavaScript Module System

CommonJS vs ES Modules (import / export)

1. Why Do We Need Modules?

As applications grow, writing all code in a single file becomes unmanageable.

Modules help us:

- Split code into multiple files
- Reuse functionality
- Maintain separation of concerns
- Improve readability
- Enable scalable architecture

In backend development, every major concept is separated into modules:

- Models
- Controllers
- Services
- Middleware
- Utilities

Without modules, enterprise-level architecture is impossible.

2. What Is a Module?

A module is simply a file that exports some functionality and can be imported into another file.

Example:

- jwt.util.js
- auth.service.js
- user.model.js

Each file exports something, and another file imports it.

3. Two Module Systems in Node.js

Node.js supports two module systems:

1. CommonJS (Older system)
2. ES Modules (Modern system)

Understanding both is important.

4. CommonJS (require / module.exports)

CommonJS is the traditional module system used in Node.js.

Export Example

```
// math.js
function add(a, b) {
  return a + b;
}

module.exports = add;
```

Import Example

```
// app.js
const add = require("./math");

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

Here:

- module.exports is used to export.
- require() is used to import.

Characteristics of CommonJS

- Synchronous loading
- Used by default in older Node versions
- Uses require
- Uses module.exports

5. ES Modules (import / export)

ES Modules are the modern JavaScript module system.

They are standard in frontend JavaScript and now supported in Node.js.

To enable ES Modules in Node.js, add this to package.json:

"type": "module"

Named Export Example

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Import

```
import { add } from "./math.js";
```

Default Export Example

```
// math.js
export default function add(a, b) {
  return a + b;
}
```

Import

```
import add from "./math.js";
```

6. Named vs Default Export

Named Export

- Can export multiple items
- Must use curly braces when importing

Example:

```
export const name = "Monsta";
export const version = "1.0";
```

Import:

```
import { name, version } from "./file.js";
```

Default Export

- Only one default per file
- Imported without curly braces

Example:

```
export default function connectDB() {}
```

Import:

```
import connectDB from "./db.js";
```

7. Why We Use ES Modules in This Project

We use ES Modules because:

- Modern standard
- Consistent with frontend JavaScript
- Cleaner syntax
- Future-proof
- Recommended for new Node projects

This is enabled using:

"type": "module"

in package.json.

8. How Modules Are Used in This Project

In this backend, modules are used everywhere.

Examples:

Example 1 — Importing Mongoose

```
import mongoose from "mongoose";
```

Example 2 — Importing Custom Utility

```
import generateAccessToken from "../../utils/jwt.util.js";
```

Example 3 — Exporting Controller Function

```
export const login = async (req, res) => {
  ...
};
```

9. Important Rule in ES Modules

When importing local files, always include .js extension:

Correct:

```
import connectDB from "./config/db.js";
```

Incorrect:

```
import connectDB from "./config/db";
```

Node requires the file extension in ES Modules.

10. Common Mistakes Students Make

1. Forgetting "type": "module" in package.json.
2. Forgetting .js in import paths.
3. Mixing require and import in the same file.
4. Using module.exports in ES module projects.

Avoid mixing CommonJS and ES Modules in the same project.

11. Summary

Concept	CommonJS	ES Modules
Export	module.exports	export / export default
Import	require()	import
Modern Standard	No	Yes
Used in This Project	No	Yes

Key Takeaways

- Modules allow scalable architecture.
- ES Modules are modern and recommended.
- Use export and import.
- Always include .js in local imports.
- Avoid mixing module systems.