Unit : 3. Application building using ES6

Introduction to ES6

ES6, or **ECMAScript 2015**, is the 6th edition of the ECMAScript language specification, and it introduced major improvements to JavaScript. It brought modern syntax and powerful features that make code more concise, readable, and modular. Some of the key features include:

- let and const for block-scoped variable declarations
- Arrow functions (=>) for shorter function expressions
- Destructuring assignment for extracting values from arrays/objects
- Modules (import/export) for cleaner code organization
- Classes, Promises, async/await, and much more

These features are particularly useful in building scalable Node.js applications, as they improve code clarity and help manage asynchronous operations more effectively.

Using ES6 in Node.js

1. ES6 Support in Node.js

Modern Node.js versions (v12 and above) support most ES6 and later features **natively**. However, **modules (import/export)** require special configuration because Node.js uses CommonJS (require) by default.

To Enable ES Modules:

Option 1: Set type in package.json

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

Then use:

```
// index.js
import express from 'express';
```

Option 2: Use .mjs extension

Rename your files:

```
mv index.js index.mjs
```

Then:

```
Import fs from 'fs';
```

3.1 ES6 Variables in Node.js

Introduction

Before ES6, JavaScript only supported variable declarations using the var keyword. With ES6, two new keywords—let and const—were introduced to solve these problems.

➤ **let – Block Scope and Reassignable**

The let keyword allows you to declare variables that are **limited to the block**, statement, or expression in which they are used. It can be reassigned, which makes it useful for values that may change over time.

```
let counter = 0;

if (true) {

let counter = 5;

console.log(counter); // 5 (block-scoped)

}

console.log(counter); // 0
```

➤**const – Block Scope and Immutable Binding**

const is also block-scoped but **cannot be reassigned** after its initial value is set. However, if the variable holds an object, the object's contents **can still be modified**.

```
const name = "Chirag";

// name = "Prajapati"; // ▯ Error


const user = { age: 30 };
```

```
user.age = 31; // ⍰ Allowed

console.log(user); // { age: 31 }
```

Use in Node.js Modules

When writing clean Node.js code using ES6, it’s recommended to use const by default, and only use let when you’re sure the variable needs to be reassigned.

```
// file: app.js

const express = require('express');

let port = 3000;


const app = express();

app.listen(port, () => console.log(`Server running on port ${port}`));
```

Common Mistakes to Avoid

1. **Re-declaring with let or const in the same scope** is not allowed.

```
let name = "Chirag";

let name = "Prajapati"; // ⍰SyntaxError
```

2. **Using let/const before declaration** leads to a ReferenceError due to the **temporal dead zone**.

```
console.log(x); // ⍰ReferenceError

let x = 10;
```

3.1.1 Functions with ES6 in Node.js

Introduction

Functions are fundamental building blocks in JavaScript. ES6 introduced a new, concise syntax for writing functions called **arrow functions** (=>), along with support for **default parameters**, **rest**

parameters, and **enhanced object literals**. These features help write cleaner and more readable code—especially useful in modern Node.js applications.

Traditional vs. Arrow Functions

Traditional Function

```
function greet(name) {

return `Hello, ${name}`;

}
```

Arrow Function (ES6)

```
const greet = (name) => `Hello, ${name}`;
```

Arrow functions are **anonymous** by nature and are often assigned to a const. They are especially useful for **short callbacks** and **functional programming** patterns.

Key Features of Arrow Functions

1. **Shorter syntax**
2. **Implicit return** if the body has only one expression.
3. **Lexical this binding** (does not bind its own this, which is very useful in callbacks or class methods).

Example in Node.js

Using Arrow Function with Express

```
import express from 'express';

const app = express();


app.get('/hello', (req, res) => {

res.send("Hello from Node.js using Arrow Function!");

});
```

```
app.listen(3000, () => console.log("Server running on port 3000"));
```

Default Parameters

Default values can be assigned to function parameters to avoid undefined.

```
const multiply = (a, b = 1) => a * b;

console.log(multiply(5)); // 5
```

3.1.2 Import and Export with ES6 in Node.js

Introduction

In large applications, code modularity is essential. ES6 introduced a standardized module system using import and export statements. This allows developers to **split JavaScript code across multiple files**, making it easier to maintain, reuse, and debug.

Before ES6, Node.js used the **CommonJS module system**, which relied on require() and module.exports. ES6 modernized this approach, enabling **static analysis**, **tree-shaking**, and better support for **tooling** like Webpack and Babel.

Basic Syntax

Exporting

You can export variables, functions, or classes from a file.

```
// math.js

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

exportconst subtract = (a, b) => a - b;
```

You can also use export default for a single default export:

```
// greeting.js

const greet = (name) => `Hello, ${name}`;

export default greet;
```

Importing

```
// app.js
import { add, subtract } from './math.js';
import greet from './greeting.js';


console.log(add(5, 3));       // 8
console.log(greet("Chirag")); // Hello, Chirag
```

ES Modules in Node.js

By default, Node.js uses CommonJS. To use ES6 import/export, you must either:

1. Add "type": "module" in package.json:

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

2. Or rename your files with .mjs extension:

```
mv app.js app.mjs
```

Now you can use ES6 imports in your Node.js files.

Named vs. Default Exports

Type	**Import Syntax**	**Example**
Named Export	import { add } from './file'	export const add = () => {}
Default Export	import anyName from './file'	export default function() {}

Real-World Example in Node.js (Express App)

/utils/logger.js

```
exportconst log = (msg) => {
```

```
console.log(`[LOG] ${msg}`);

};
```

/routes/home.js

```
export default (req, res) => {

res.send("Welcome to the Home Page");

};
```

index.js

```
import express from 'express';

import { log } from './utils/logger.js';

importhomeRoute from './routes/home.js';


const app = express();

app.get("/", homeRoute);


app.listen(3000, () => log("Server running on port 3000"));
```

Advantages of ES Modules

- **Cleaner syntax** (compared to CommonJS)
- **Static structure** (analyzable at compile time)
- **Tree-shaking** support (remove unused code)
- **Scoped imports/exports**, reducing global pollution

Migration Tip

When moving from CommonJS (require) to ES Modules:

```
// CommonJS
```

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


// ES6

importfs from 'fs';
```

Note: You cannot mix CommonJS and ES6 modules in the same file.

3.1.3 Async/Await in Node.js

Introduction

One of the biggest challenges in JavaScript and Node.js is handling asynchronous operations, like file access, API calls, and database queries. Initially, callbacks were used, but they led to **callback hell** — deeply nested and unreadable code. Promises improved this by allowing chaining, but they could still be complex when dealing with multiple asynchronous operations.

ES6 introduced Promises, and **ES8 (2017)** introduced async and await as syntactic sugar over Promises, providing a cleaner, more readable way to write asynchronous code.

What is async/await?

- async is a keyword that marks a function as asynchronous, which means it **returns a Promise**.
- await is used **inside an async function** to pause the execution until the Promise is resolved or rejected.

Syntax

```
async function fetchData() {

const result = await someAsyncFunction();

console.log(result);

}
```

Using await allows you to write code that looks synchronous but behaves asynchronously.

Example in Node.js (API Fetch)

Using **Node.js v18+** (which supports native fetch):

```
constfetchData = async () => {
try {
const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
const data = await response.json();
console.log(data);
  } catch (error) {
console.error("Error fetching data:", error);
  }
};


fetchData();
```

If you're using Node.js <18, you’ll need to install node-fetch:

```
npm install node-fetch
```

Then:

```
import fetch from 'node-fetch';
```

Async/Await with File System in Node.js

Node.js has a Promise-based version of the File System module (fs/promises):

```
import { readFile } from 'fs/promises';


constreadText = async () => {
```

```
try {
const content = await readFile('example.txt', 'utf-8');
console.log(content);
  } catch (err) {
console.error('File read error:', err);
  }
};
readText();
```

3.2 Introduction to Babel (in Node.js)

What is Babel?

Babel is a **JavaScript compiler** that allows developers to use **next-generation JavaScript features** (ES6 and beyond) by **transpiling** the code into a version that is compatible with older JavaScript engines. Babel ensures that modern code (with features like arrow functions, classes, async/await, optional chaining, etc.) works in environments that don't fully support them — including some browsers and earlier versions of Node.js.

Even though modern versions of Node.js support most ES6+ features, **Babel is still valuable** in scenarios like:

- Creating cross-platform libraries (Node.js + browser)
- Working in enterprise codebases that need compatibility
- Using JSX (for full-stack apps with React + Node)

Why Do We Need Babel in Node.js?

While Node.js has solid support for ES6+ (especially from v12 onwards), certain features (like top-level await, decorators, optional chaining, or class fields) might still need to be transpiled for compatibility or consistency across team workflows.

Also, some teams use Babel as part of a **build toolchain** (with Webpack, ESLint, etc.) to ensure uniform syntax and browser compatibility in **isomorphic apps** (same code runs on server and client).

How Babel Works

Babel reads your source files and converts ES6+ code to ES5 or compatible syntax using **plugins and presets**. You configure Babel using a .babelrc file or babel.config.js.

Setting Up Babel in Node.js

Step 1: Initialize project

```
npm init -y
```

Step 2: Install Babel dependencies

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

Step 3: Create .babelrc configuration file

```
{
  "presets": ["@babel/preset-env"]
}
```

This tells Babel to convert your modern JavaScript code into code that works in your target environments (Node.js, older browsers, etc.).

Writing ES6 Code

Create a file named src/index.js:

```
const greet = (name = "Guest") => {
console.log(`Hello, ${name}`);
};


greet("Chirag");
```

You can now use ES6 features like arrow functions, default parameters, etc.

Transpiling the Code

To transpile your code using Babel:

```
npx babel src --out-dirdist
```

This will convert the ES6 code from the src folder to ES5 code in the dist folder.

To run the transpiled code:

```
nodedist/index.js
```

Babel with import/export

If you want to use import and export in Node.js without "type": "module" in package.json, Babel will convert it to require() syntax internally.

Example src/utils.js:

```
exportconst sum = (a, b) => a + b;
```

Example src/index.js:

```
import { sum } from './utils.js';

console.log(sum(5, 3)); // 8
```

After transpilation, this will become:

```
const { sum } = require('./utils');
```

3.3 REST API with ES6 in Node.js

What is a REST API?

A **REST (Representational State Transfer) API** is a web service architecture that uses HTTP methods (GET, POST, PUT, DELETE) to interact with resources. Each resource (e.g., user, product, post) is represented by a unique URI and manipulated using standard operations.

Node.js, combined with **Express.js** and modern **ES6+ features**, is one of the most popular stacks for building fast, scalable RESTful APIs.

Key ES6 Features Used in REST APIs

When building a REST API in Node.js, you commonly use:

- import/export for modular code
- const and let for variable declarations
- Arrow functions for cleaner syntax
- async/await for handling asynchronous operations
- Destructuring, spread/rest operators, and template literals

---

Setting Up an ES6 REST API

Step 1: Initialize the Project

```
npminit -y

npm install express
```

Step 2: Enable ES6 Module Support

In your package.json, add:

```
"type": "module"
```

Folder Structure

```
project-root/
├── index.js
├──routes/
│   └── userRoutes.js
└── controllers/
    └── userController.js
```

---

index.js (Main File)

```
import express from 'express';

importuserRoutes from './routes/userRoutes.js';


const app = express();

const PORT = 3000;


// Middleware to parse JSON

app.use(express.json());


// Routes

app.use('/api/users', userRoutes);


// Start Server

app.listen(PORT, () => {

console.log(`▯ Server running at http://localhost:${PORT}`);

});
```

---

routes/userRoutes.js

```
import express from 'express';

import { getUsers, createUser } from '../controllers/userController.js';


const router = express.Router();
```

```
router.get('/', getUsers);
router.post('/', createUser);

export default router;
```

---

controllers/userController.js

```
const users = [
{ id: 1, name: 'Chirag' },
{ id: 2, name: 'Prajapati' }
];

// GET /api/users
exportconstgetUsers = (req, res) => {
res.status(200).json(users);
};

// POST /api/users
exportconstcreateUser = (req, res) => {
const { name } = req.body;
constnewUser = { id: users.length + 1, name };
users.push(newUser);
res.status(201).json(newUser);
};
```

Testing the API

You can test your API using:

- **Postman**
- **Insomnia**
- **cURL**
- **Browser (for GET)**

Example GET request:

GET http://localhost:3000/api/users

Example POST request (with JSON body):

POST http://localhost:3000/api/users

Body: { "name": "Karan" }

Browsing HTTP Requests with Fetch (ES6)

What is fetch?

The fetch() API is a modern ES6+ method for making HTTP requests. Originally available only in browsers, it allows developers to send requests to servers, retrieve data (like JSON), and handle asynchronous network operations in a simpler way than older techniques like XMLHttpRequest.

ES6 + Fetch in Node.js

Node.js **doesn't have fetch natively** (except from v18+). If you're using an earlier version or want cross-platform support, you need to install a package like:

npm install node-fetch

And then use it with import:

import fetch from 'node-fetch';

Or with require if using CommonJS:

const fetch = require('node-fetch');

Syntax of fetch():

```
fetch(url, {
  method: 'GET' | 'POST' | 'PUT' | 'DELETE',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(data) // only for POST/PUT
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

Practical Example (Node.js + Fetch with ES6)

Let's fetch a list of products from a fake REST API.

File: app.js

```
import fetch from 'node-fetch';


const URL = 'https://fakestoreapi.com/products';


async function getProducts() {
  try {
    const response = await fetch(URL);
    if (!response.ok) throw new Error('Network response was not ok');


    const products = await response.json();
    console.log("Product List:");

    products.forEach(p => console.log(`${p.id}. ${p.title}`));

  } catch (error) {

    console.error('Fetch error:', error.message);

  }

}


getProducts();
```