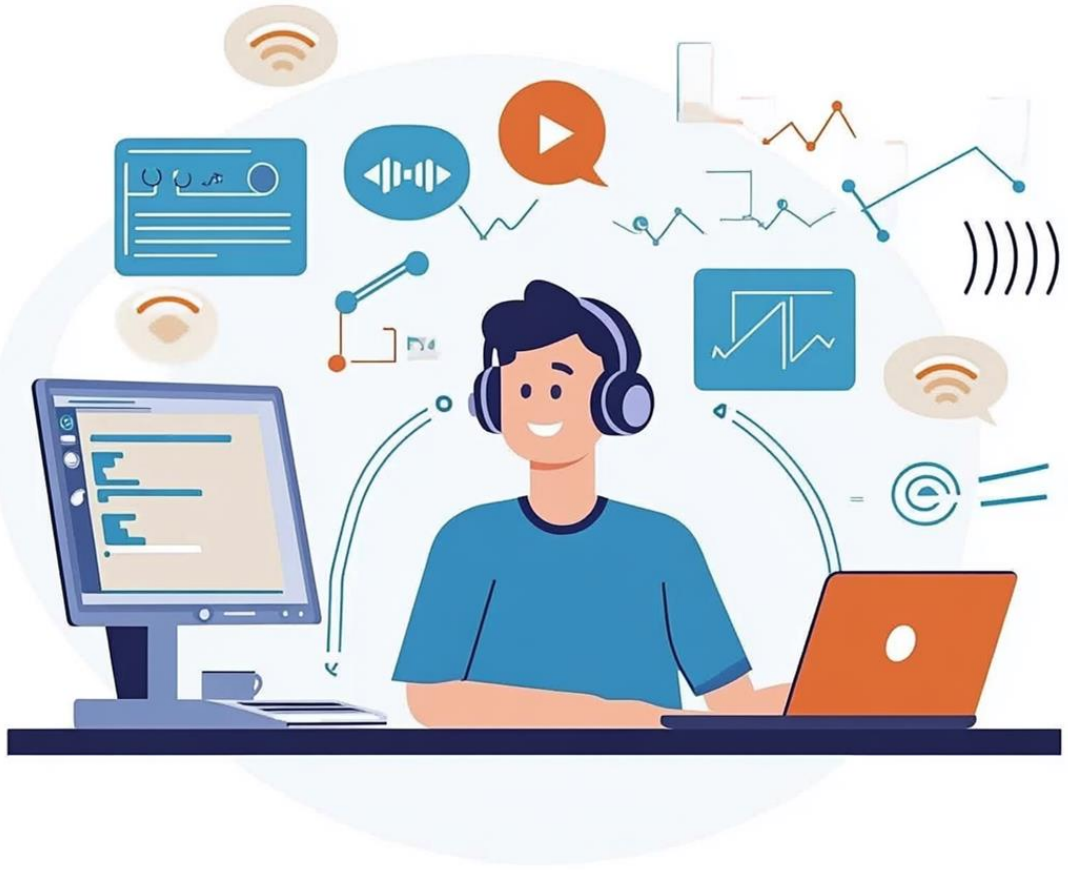


node JS

Gnanavel Durairaj

Content

On Day – 2

1. Understanding `require` and `module.exports`
2. File System Module (`fs`): Reading, writing, and deleting files
3. Path Module: Working with file and directory paths
4. OS Module: Fetching system-related information
5. Events and EventEmitter: Creating & handling custom events
6. Streams and Buffers: Understanding data streaming
7. Exercise and Quiz

Understanding `require` and `module.exports`

In Node.js, `require` and `module.exports` are essential parts of the CommonJS module system, which is the default module system in Node.js. They allow you to organize code into reusable pieces, making your application more modular and maintainable.

`module.exports` is used to export functionality (like functions, objects, or values) from a module so that it can be imported and used in other files.
By default, every JavaScript file in Node.js is treated as a module, and `module.exports` is an empty object. You attach properties or methods to this object to make them available outside the module.

Let's look at real-time examples where `require` and `module.exports` are used in typical Node.js applications. Examples that simulate real-world scenarios, such as a simple web server, a database connection module, and a utility module.

Understanding `require` and `module.exports`

Example 1: Building a Simple Web Server

Imagine you're building a web server with Express in Node.js. You might create a modular structure to keep your routes, controllers, and server configuration separate.

Step 1: Create the route module (`routes.js`)

In a real-world app, routes can get complex, and separating them into their own module makes things more organized.

```
// routes.js
module.exports = function(app) {
  // Define a simple route
  app.get('/', (req, res) => {
    res.send('Hello, World!');
  });
  app.get('/about', (req, res) => {
    res.send('About Page');
  });};
```

Here, we're exporting a function that takes an `app` object (an Express instance) and defines routes within that function.

Understanding `require` and `module.exports`

Step 2: Create the server module (`server.js`)

```
// server.js
const express = require('express');
const app = express();
// Import the routes module
const routes = require('./routes');
// Use the imported routes in the app
routes(app);
// Start the server
app.listen(3000, () => {  console.log('Server is running on http://localhost:3000');});
```

In the `server.js` file, we create an Express app, require the `routes.js` file, and pass the `app` instance to the routes function. This keeps the routing logic separate from the server setup, making your code more modular and maintainable.

Output:

- Navigating to `http://localhost:3000` will show "Hello, World!".
- Navigating to `http://localhost:3000/about` will show "About Page".

Understanding `require` and `module.exports`

Example 2: Database Connection and Querying (Database Module)

Let's simulate connecting to a database and querying it, keeping the database connection logic in a separate module.

Step 1: Create the database connection module (`db.js`)

```
// db.js
const mysql = require('mysql');
// Set up the connection
const connection = mysql.createConnection({ host: 'localhost',  user: 'root',
  password: 'password',  database: 'my_database'});
// Connect to the database
connection.connect((err) => {
  if (err) {    console.error('Error connecting to the database:', err.stack);
    return;  }
  console.log('Connected to the database');});
// Export the connection for use in other modules
module.exports = connection;
```

Here, we're creating a database connection using `mysql` (which is a popular library for MySQL in Node.js) and exporting the `connection` object so that other parts of the application can use it.

Understanding `require` and `module.exports`

Step 2: Create a module to query the database (`users.js`)

```
// users.js
const db = require('./db');
// Function to get users from the database
module.exports.getUsers = function(callback) {
  db.query('SELECT * FROM users', (err, results) => {
    if (err) {
      console.error('Error fetching users:', err);
      return;
    }
    callback(results);
  });};
```

Here, we import the `db.js` module and use the database connection to run a SQL query to fetch users.

Understanding `require` and `module.exports`

Step 3: Create the main application file (`app.js`)

```
// app.js
const express = require('express');
const app = express();
// Import the users module
const users = require('./users');
app.get('/users', (req, res) => {
  users.getUsers((results) => {
    res.json(results); // Send users as JSON response
  });
});
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

In the `app.js` file, we import the `users` module and use it to fetch users from the database when the `/users` route is accessed.

Output:

- If there are users in your MySQL database, navigating to `http://localhost:3000/users` will return a JSON array of users.

Understanding `require` and `module.exports`

ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY ‘password';

FLUSH PRIVILEGES;

Understanding `require` and `module.exports`

Example 3: Utility Functions

Utility modules are common in larger apps. For example, imagine you need a set of utility functions like logging or working with dates. Instead of repeating these functions across your app, you can create a utility module.

Step 1: Create the utility functions module (`utils.js`)

```
// utils.js

// Utility function to log messages
module.exports.logInfo = function(message) {
  console.log(`[INFO] ${new Date().toISOString()} - ${message}`);};

// Utility function to log errors
module.exports.logError = function(message) {
  console.error(`[ERROR] ${new Date().toISOString()} - ${message}`);
};
```

Here, we define two utility functions (`logInfo` and `logError`) to handle logging. Both are exported using `module.exports`.

Understanding `require` and `module.exports`

Step 2: Create the main application file (`app.js`)

```
// app.js
const express = require('express');
const app = express();
// Import the utility module
const utils = require('./utils');
app.get('/', (req, res) => {
  utils.logInfo('Handling request to /');
  res.send('Hello, World!');});
app.get('/error', (req, res) => {
  utils.logError('An error occurred');
  res.status(500).send('Internal Server Error');});
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');});
```

In this case, we're using the utility module to log informational and error messages when handling requests.

Output:

- When you access `http://localhost:3000`, it logs an info message.
- When you access `http://localhost:3000/error`, it logs an error message.

Understanding `require` and `module.exports`

Example 4: Exporting Classes or Objects

In larger applications, you might need to export classes or complex objects. Here's an example of how you might do that.

Step 1: Create a class module (`Car.js`)

```
// Car.js
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
Car.prototype.getDetails = function() {
  return `${this.year} ${this.make} ${this.model}`;};
// Exporting the class
module.exports = Car;
```

In this file, we're creating a `Car` class with a constructor and a method `getDetails`. We export the `Car` constructor function so that other parts of the application can create instances of `Car`.

Understanding `require` and `module.exports`

Step 2: Use the `Car` class in the main application (`app.js`)

```
// app.js
const Car = require('./Car'); // Import the Car class
const myCar = new Car('Toyota', 'Corolla', 2021);
console.log(myCar.getDetails()); // Output: 2021 Toyota Corolla
```

Here, we require the `Car` module, create an instance of the `Car` class, and then call the `getDetails` method to log the car's details.

These examples show how `module.exports` and `require` are used in real-time scenarios to help build maintainable, modular, and scalable applications in Node.js. Each module is self-contained, making the code easier to test, reuse, and manage.

exports vs module.exports

FEATURES	exports	module.exports
Shortcut for module.exports	✓	✓
Can be Reassigned Completely	✗	✓
Best for exporting multiple properties	✓	✓
Best for exporting a single function/class	✗	✓

Example: Incorrect And Correct Usage

```
exports = function() { return 'This won't work!' }; // ✗
```

```
module.exports = function() { return 'This works!'; }; // ✓
```

File System Module (`fs`): Read, Write & Delete Files

- The fs (File System) module allows interaction with the file system.
- Supports both **synchronous** and **asynchronous** operations.
- Common operations:
  - Reading files
  - Writing files
  - Deleting files

```
const fs = require('fs'); // Importing fs module
```

Reading a File (Asynchronous)

- fs.readFile() is used to read a file asynchronously.
- Callback function handles errors and returns file data.

```
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }
    console.log('File content:', data);
});
```

Advantages:

- Non-blocking (doesn't pause execution).
- Ideal for large file processing.

Reading a File (Synchronous)

fs.readFileSync() reads a file synchronously.

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

console.log('File content:', data);
```

Key Differences:

- Blocks execution until file is read.
- Useful for small files.

Writing to a File (Asynchronous)

fs.writeFile() creates or overwrites a file asynchronously.

```
fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {
    if (err) {
        console.error('Error writing file:', err);
        return;
    }
    console.log('File written successfully!');
});
```

Notes:

- If file does not exist, it is created.
- If file exists, content is overwritten.

Writing to a File (Synchronous)

fs.writeFileSync() writes to a file synchronously.

```
fs.writeFileSync('example.txt', 'Hello, World!');
console.log('File written successfully!');
```

Considerations:

•Blocks execution.

•Not recommended for high-performance applications.

Appending to a File

fs.appendFile() adds data to an existing file.

```
fs.appendFile('example.txt', '\nAppended text.', (err) => {
    if (err) {
        console.error('Error appending file:', err);
        return;
    }
    console.log('Text appended successfully!');
});
```

Use case:
Logging systems, updating configuration files.

Deleting a File

fs.unlink() removes a file asynchronously.

```
fs.unlink('example.txt', (err) => {
    if (err) {
        console.error('Error deleting file:', err);
        return;
    }
    console.log('File deleted successfully!');
});
```

Precautions:

- Ensure file exists before deletion.
- Handle errors properly to avoid crashes.

Checking if a File Exists (Synchronous)

fs.existsSync() checks file existence before performing operations.

```
if (fs.existsSync('example.txt')) {
    console.log('File exists');
} else {
    console.log('File does not exist');
}
```

Why use this?

•Prevents errors when accessing non-existent files

Checking if a File Exists (Asynchronous)

fs.access() checks file existence asynchronously.

```
// FileExistsAsync.js
const fs = require('fs');
// Checking if a file exists asynchronously
fs.access('example.txt', fs.constants.F_OK, (err) => {
  if (err) {
    console.log('File does not exist!');
  } else {    console.log('File exists!');  }
});
```

Reading a Directory (Asynchronous)

This will list the files and directories in the current directory (./)..

```
// ReadDirectoryAsync.js
const fs = require('fs');
// Reading a directory asynchronously
fs.readdir('./', (err, files) => {
  if (err) {
    console.error('Error reading directory:', err);
    return;
  }
  console.log('Files in the directory:', files);});
```

Creating a Directory

Creating a directory asynchronously

```
// CreateDirectoryAsync.js
const fs = require('fs');
// Creating a directory asynchronously
fs.mkdir('new_folder', (err) => {
  if (err) {
    console.error('Error creating directory:', err);
    return;
  }
  console.log('Directory created!');});
```

Deleting a Directory

Deleting a directory asynchronously

```
const fs = require('fs');
// Deleting a directory asynchronously
fs.rmdir('new_folder', (err) => {
  if (err) {
    console.error('Error deleting directory:', err);
    return;
  }
  console.log('Directory deleted!');
});
```

fs.createReadStream()

The fs.crateReadStream() method is used to **read large files efficiently** in Node.js. Instead of loading the entire file into memory, it reads the file in chunks (streaming data), making it memory-efficient.

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

const readStream = fs.createReadStream('example.txt', {
encoding: 'utf8' });

readStream.on('data', (chunk) => {
    console.log('New chunk received:', chunk);
});
readStream.on('end', () => {
    console.log('Finished reading the file.');
});
readStream.on('error', (err) => {
    console.error('Error reading the file:', err);
});
```

Summary of File System Operations

Operations	Methods	Asynchronous	Synchronous
Read a File	fs.readFile()	✓	✗
Write a File	fs.writeFile()	✓	✗
Append Data to File	fs.appendFile()	✓	✗
Delete a File	fs.unlink()	✓	✗
Check if File Exist	fs.existsSync() or fs.access()	✗	✓
Read a Directory	fs.readdir()	✓	✗
Create a Directory	fs.mkdir()	✓	✗
Delete a Directory	fs.rmdir()	✓	✗

Path Module: Working with file and directory paths

- The path module helps in working with **file and directory paths**.
- Works across **different operating systems**.
- Provides methods for handling file paths easily.

```
const path = require('path'); // Importing the path module
```

Getting the Directory Name

path.dirname(filePath) returns the directory part of a file path.

```
const filePath = '/home/user/docs/file.txt';
console.log(path.dirname(filePath)); // Output: /home/user/docs
```

Use case: Useful when extracting file locations.

Getting the Base File Name

path.basename(filePath) returns the file name.

```
console.log(path.basename(filePath)); // Output: file.txt
```

Variation: Get the file name without extension

```
console.log(path.basename(filePath, '.txt')); // Output: file
```

Getting the File Extension

path.extname(filePath) returns the file extension.

```
console.log(path.extname(filePath)); // Output: .txt
```

Use case: Useful when filtering files by type.

Joining Paths

path.join() creates a proper file path, ensuring correct formatting.

```
const fullPath = path.join('/home', 'user', 'docs', 'file.txt');
console.log(fullPath); // Output: /home/user/docs/file.txt
```

Use case: Avoids errors due to manual path concatenation.

Absolute vs Relative Path

path.resolve() returns an absolute path.

```
console.log(path.resolve('file.txt')); // Output:
/absolute/path/to/file.txt
```

Difference between join and resolve:

- join just combines paths.
- resolve converts it into an absolute path.

Checking if a Path is Absolute

path.isAbsolute(path)

```
console.log(path.isAbsolute('/home/user/file.txt')); // true
console.log(path.isAbsolute('file.txt')); // false
```

Use case: Useful in file system operations.

Parsing a File Path

path.parse(filePath) returns an object with details.

```
console.log(path.parse('/home/user/docs/file.txt'));
```

Output:

```
{
  root: "/",
  dir: "/home/user/docs",
  base: "file.txt",
  ext: ".txt",
  name: "file"
}
```

Formatting a Path

path.format() converts an object to a file path.

```
const fileObject = {
  dir: '/home/user/docs',
  base: 'file.txt'
};
console.log(path.format(fileObject)); // Output: /home/user/docs/file.txt
```

Summary of Path Module

✅ path.dirname() → Get directory path
✅ path.basename() → Get file name
✅ path.extname() → Get file extension
✅ path.join() → Merge paths properly
✅ path.resolve() → Get absolute path
✅ path.parse() → Convert path to object

OS Module: Fetching system-related information

- The os module provides system-related information.
- Useful for system diagnostics, process management.

```
const os = require('os'); // Importing OS module
```

Getting OS Type & Platform

- os.type() → Returns OS name.
- os.platform() → Returns platform name.

```
console.log(os.type()); // Output: Linux / Windows_NT / Darwin
console.log(os.platform()); // Output: win32 / linux / darwin
```

Use case: Helps detect the running OS.

Getting Hostname & User Info

- os.hostname() → Returns computer name.
- os.userInfo() → Returns user details.

```
console.log(os.hostname()); // Output: DESKTOP-ABC123
console.log(os.userInfo());
```

Sample User Info Output:

```
{
  username: "user123",
  homedir: "/home/user",
  shell: "/bin/bash"
}
```

Getting CPU Information

os.cpus() → Returns details about CPU cores.

```
console.log(os.cpus());
```

Use case: Useful for performance monitoring.

Checking Free & Total Memory

- os.freemem() → Free memory in bytes.
- os.totalmem() → Total memory in bytes.

Use case: Helps in memory management.

Getting System Uptime

•os.uptime() → Returns system uptime in seconds.

```
console.log('Uptime in seconds:', os.uptime());
```

Use case: Tracks system stability.

Getting Network Information

- os.networkInterfaces() → Returns network details.

```
console.log(os.networkInterfaces());
```

Use case: Useful for network diagnostics.

Summary of OS Module

- os.type() → Get OS name
- os.platform() → Get platform type
- os.hostname() → Get device hostname
- os.userInfo() → Get logged-in user details
- os.cpus() → Get CPU details
- os.freemem() & os.totalmem() → Check memory status
- os.uptime() → Get system uptime

Events and EventEmitter:

Creating and Handling Custom Events

Introduction to Events in Node.js

•Node.js is **event-driven** and follows an **asynchronous** model.

•Uses the **EventEmitter** class from the events module.

•Custom events can be created and handled dynamically.

```
const EventEmitter = require('events');
const event = new EventEmitter();
```

Creating and Emitting an Event

- .emit() triggers an event.
- .on() listens for an event.

```
event.on('sayHello', () => {
    console.log('Hello, Node.js!');
});
event.emit('sayHello'); // Output: Hello, Node.js!
```

Use Case: Custom event-based handling in applications.

Passing Arguments to Events

Events can pass data when emitted.

```
event.on('greet', (name) => {
    console.log(`Hello, ${name}!`);
});
event.emit('greet', 'Alice'); // Output: Hello, Alice!
```

Use Case: Handling dynamic event responses.

Handling Events Only Once

.once() listens for an event **only once**.

```
event.once('welcome', () => {
    console.log('Welcome, this will show only once!');
});
event.emit('welcome');
event.emit('welcome'); // Will not execute again
```

Use Case: One-time initialization processes.

Removing Event Listeners

.removeListener() removes a specific listener.

```
const greet = () => console.log('Hello there!');
event.on('greet', greet);
event.removeListener('greet', greet);
event.emit('greet'); // No output
```

Use Case: Stopping event listeners dynamically.

Summary of Events and EventEmitter

Key Functions:

- on(): Listen for an event.
- emit(): Trigger an event.
- once(): Execute a listener only once.
- removeListener(): Remove a specific event listener.

Streams and Buffers: Understanding data streaming in Node.js

- Streams handle **large data efficiently** without loading all data into memory.
- Four types of streams:
  - **Readable** – Read data
  - **Writable** – Write data
  - **Duplex** – Read and write
  - **Transform** – Modify data

```
const fs = require('fs');
const stream = fs.createReadStream('file.txt');
```

Reading Data Using Streams

.createReadStream() reads data in chunks.

```
const fs = require('fs');
const stream = fs.createReadStream('data.txt', 'utf8');

stream.on('data', (chunk) => {
    console.log('Received chunk:', chunk);
});
```

Use Case: Reading large files without memory overload.

Writing Data Using Streams

.createWriteStream() writes data to a file.

```
const writeStream = fs.createWriteStream('output.txt');
writeStream.write('Hello, this is a stream example.');
writeStream.end();
```

Use Case: Writing logs or saving real-time data.

Piping Streams

Piping passes readable stream data directly to a writable stream.

```
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);
```

Use Case: Efficiently copying large files.

Introduction to Buffers

- Buffers handle **binary data** in Node.js.
- Used in networking, file handling, and streams.

```
const buf = Buffer.from('Hello');
console.log(buf); // Output: <Buffer 48 65 6c 6c 6f>
```

Use Case: Handling raw binary data.

Modifying Buffers

- Buffers can be manipulated like arrays.

```
buf[0] = 65;
console.log(buf.toString()); // Output: Aello
```

Use Case: Encoding/decoding data efficiently.

Summary of Streams and Buffers

✅ **Key Functions:**

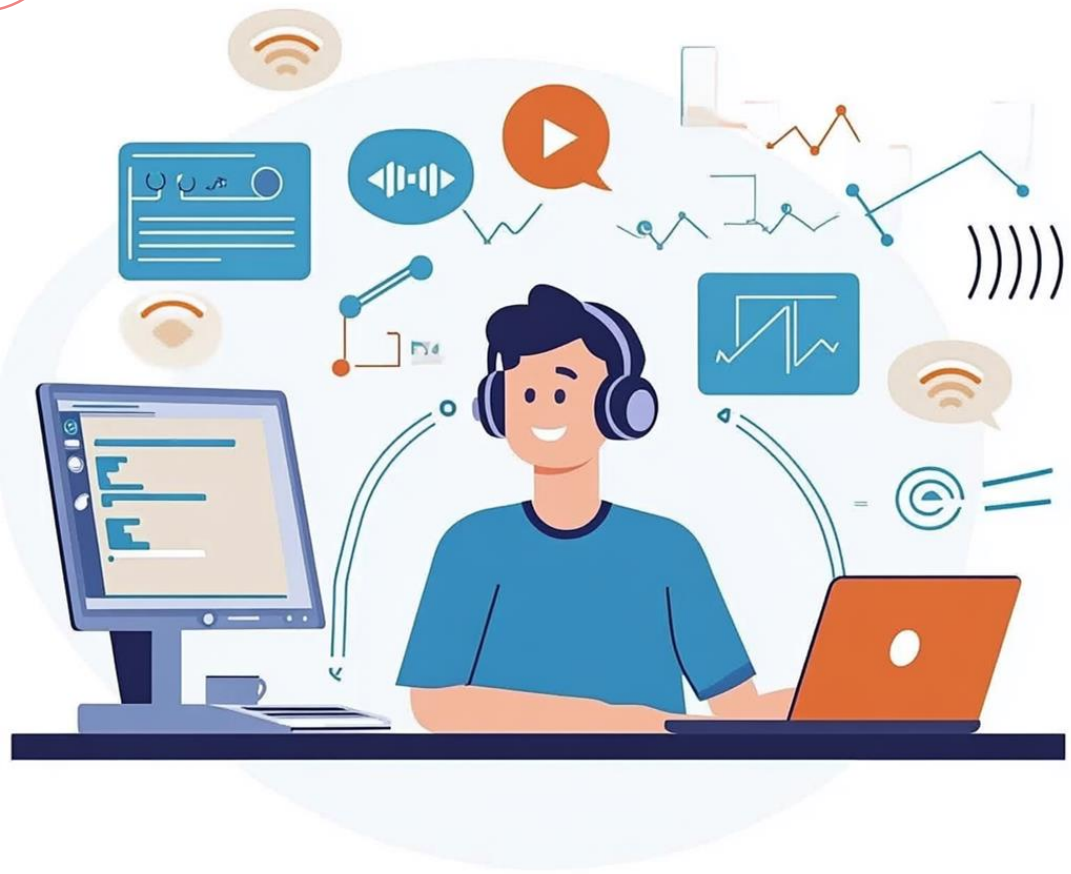
•createReadStream(): Read large files.

•createWriteStream(): Write large files.

•pipe(): Pass data between streams.

•Buffer.from(): Create a buffer.

Exercise and Quiz



Thanks – End of Day 02

Edited by Gnanavel Durairaj

17-Mar-2025