Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

**ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)**

# FILE SYSTEM, STREAMS & EVENT-DRIVEN PROGRAMMING (WEEKS 3-4)

# READING AND WRITING FILES ASYNCHRONOUSLY IN NODE.JS

## CHAPTER 1: INTRODUCTION TO FILE SYSTEM OPERATIONS IN NODE.JS

### 1.1 Understanding Asynchronous File Handling

Node.js provides a built-in **File System (fs) module** that allows developers to work with files on their system. This module supports both **synchronous** and **asynchronous** operations.

- **Synchronous file operations** block the execution of other tasks until they are completed.
- **Asynchronous file operations** allow the program to continue running while the file tasks are processed in the background.

In most cases, **asynchronous file handling** is preferred because it improves the performance and scalability of an application.

**Example Use Cases of Asynchronous File Handling:**

✓ Reading large files without freezing the application.

✓ Writing logs without delaying request processing.
✓ Streaming data efficiently to users.

---

## CHAPTER 2: READING FILES ASYNCHRONOUSLY

### 2.1 Using fs.readFile()

The fs.readFile() method reads the content of a file **asynchronously**. Unlike synchronous methods, it does not block execution while waiting for the file to load.

**Example: Reading a File Asynchronously**

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


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

**Explanation:**

- The fs.readFile() function takes the filename (example.txt), encoding (utf8), and a **callback function** that executes once the file is read.
- If an error occurs (e.g., the file does not exist), it is handled inside err.

- If successful, the file content is displayed.

---

### 2.2 Using Promises with fs.promises.readFile()

The **Promise-based** approach provides a cleaner way to handle asynchronous operations, avoiding callback nesting.

**Example: Reading a File Using Promises**

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


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

- This approach returns a **Promise,** allowing .then() to handle success and .catch() to handle errors.

---

### 2.3 Using Async/Await with fs.promises.readFile()

With async/await, file reading becomes even more readable and structured.

**Example: Reading a File Using Async/Await**

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

```
async function readFileAsync() {
  try {
    const data = await fs.readFile('example.txt', 'utf8');
    console.log('File content:', data);
  } catch (err) {
    console.error('Error reading file:', err);
  }
}

readFileAsync();
```

- The await keyword ensures the file is read before moving to the next instruction.
- Errors are handled using try...catch.

---

## CHAPTER 3: WRITING FILES ASYNCHRONOUSLY

### 3.1 Using fs.writeFile()

Node.js allows writing to files asynchronously using fs.writeFile(). This method **overwrites** existing content in the file or creates a new file if it does not exist.

**Example: Writing to a File Asynchronously**

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

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

- If output.txt exists, its content is replaced.
- The callback function handles errors and confirms the write operation.

---

### 3.2 Using Promises with fs.promises.writeFile()

For a **Promise-based** approach:

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

fs.writeFile('output.txt', 'Hello, Node.js!')
  .then(() => console.log('File written successfully!'))
  .catch((err) => console.error('Error writing to file:', err));
```

- This improves readability, especially when dealing with multiple file operations.

---

### 3.3 Using Async/Await with fs.promises.writeFile()

**Example: Writing a File Using Async/Await**

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


async function writeFileAsync() {
  try {
    await fs.writeFile('output.txt', 'Hello, Node.js!');
    console.log('File written successfully!');
  } catch (err) {
    console.error('Error writing to file:', err);
  }
}


writeFileAsync();
```

- await ensures the write operation completes before executing the next instruction.

---

## Case Study: How Asynchronous File Handling Helps Large-Scale Applications

### Background

A cloud storage company faced frequent slowdowns when handling large file uploads and downloads due to **synchronous file operations**. The server became unresponsive while processing large files, causing delays for users.

### Challenges

- **Long waiting times** for file read/write operations.

- **High memory consumption** due to synchronous execution.
- **Server crashes** when handling multiple file operations concurrently.

**Solution: Implementing Asynchronous File Operations**

The company switched to **asynchronous file handling using fs.readFile() and fs.writeFile()**. The improvements included:

✓ **Faster file operations** due to non-blocking execution.
✓ **Better resource management,** reducing server crashes.
✓ **Improved scalability,** allowing thousands of users to access files simultaneously.

This case study highlights the **importance of asynchronous programming in Node.js** for handling file operations efficiently.

---

**Exercise**

1. Modify the following code to read a file asynchronously using Promises:

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

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

console.log(data);
```

2. Write an async function that writes "Asynchronous File Handling" to a file named test.txt.

---

**Conclusion**

In this section, we explored:
✓ **How to read and write files asynchronously in Node.js using**

**fs.readFile() and fs.writeFile().**

**✓ The advantages of Promises and Async/Await over callback-based file operations.**

**✓ A real-world case study demonstrating the benefits of asynchronous file handling.**

# WORKING WITH DIRECTORIES AND PATHS IN NODE.JS

## CHAPTER 1: INTRODUCTION TO FILE SYSTEM MANAGEMENT IN NODE.JS

### 1.1 Understanding Directories and Paths in Node.js

In Node.js, managing files and directories is crucial for building applications that interact with the filesystem. The **File System (fs) module** provides built-in methods to create, read, update, and delete directories and files. Additionally, the **path module** helps handle file paths efficiently across different operating systems.

Key aspects of working with directories and paths in Node.js include:

- **Creating, reading, renaming, and deleting directories**.
- **Manipulating file paths to ensure compatibility across operating systems**.
- **Working with absolute and relative paths**.

By leveraging these functionalities, developers can **organize files dynamically**, manage **uploads**, and create **structured file management systems**.



## CHAPTER 2: THE FS MODULE AND DIRECTORY OPERATIONS

### 2.1 Creating and Removing Directories

Node.js provides various methods to create and delete directories using the **fs module**.

**Creating a Directory (fs.mkdir)**

The fs.mkdir method is used to create a new directory.

**Example: Creating a New Directory**

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


fs.mkdir('newFolder', (err) => {
  if (err) {
    return console.error("Error creating directory:", err);
  }
  console.log("Directory created successfully!");
});
```

- If the folder newFolder does not exist, it will be created.
- The callback function handles any errors that may occur.

**Removing a Directory (fs.rmdir)**

To remove a directory, use the fs.rmdir method:

```
fs.rmdir('newFolder', (err) => {
  if (err) {
    return console.error("Error removing directory:", err);
  }
  console.log("Directory removed successfully!");
});
```

- This method deletes only **empty directories**.
- To remove **non-empty directories**, use fs.rm with { recursive: true }.

**Example: Removing a Non-Empty Directory**

```
fs.rm('newFolder', { recursive: true }, (err) => {
  if (err) {
    return console.error("Error removing directory:", err);
  }
  console.log("Directory removed successfully!");
});
```

### 2.2 Reading Directory Contents (fs.readdir)

To list all files and folders inside a directory, use the fs.readdir method:

```
fs.readdir('.', (err, files) => {
  if (err) {
    return console.error("Error reading directory:", err);
  }
  console.log("Directory contents:", files);
});
```

- The '.' parameter represents the **current working directory**.
- The files array contains a list of all files and subdirectories.

---

## CHAPTER 3: THE PATH MODULE AND FILE PATH OPERATIONS

### 3.1 Introduction to the path Module

The path module in Node.js helps manage file paths across different operating systems. It ensures that file and folder paths are correctly

formatted regardless of whether the application runs on **Windows (uses \)** or **Linux/macOS (uses /)**.

### 3.2 Key Methods in the path Module

**Getting the Directory Name (path.dirname)**

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


const filePath = '/Users/john/project/index.js';
console.log("Directory Name:", path.dirname(filePath));
```

Output:

Directory Name: /Users/john/project

**Getting the File Extension (path.extname)**

```
console.log("File Extension:", path.extname(filePath));
```

Output:

File Extension: .js

**Getting the File Name (path.basename)**

```
console.log("File Name:", path.basename(filePath));
```

Output:

File Name: index.js

**Joining Paths (path.join)**

```
const newPath = path.join(__dirname, 'files', 'data.txt');
console.log("Joined Path:", newPath);
```

- __dirname represents the **current directory**.

- The method **joins directory paths correctly** regardless of the operating system.

---

### Case Study: How Large Applications Handle File Management

#### Background

A cloud storage company faced issues when managing **millions of user-uploaded files**. Users uploaded documents, images, and videos that needed to be **organized, retrieved efficiently**, and **secured properly**.

#### Challenges

- Managing **directory structures** for thousands of users.
- Handling **large file paths dynamically** across different operating systems.
- Ensuring **scalability** while preventing filesystem overloads.

#### Solution: Using Node.js fs and path Modules

The company implemented **Node.js file management strategies**:

- Used fs.mkdir to **dynamically create folders** for each user.
- Applied fs.readdir to **list files efficiently** when users requested them.
- Integrated path.join to ensure **cross-platform compatibility**.
- Implemented an **automated file cleanup system** using fs.rm({ recursive: true }).

#### Results

- **50% faster file retrieval** compared to the previous system.

- **Efficient storage management** by organizing user files systematically.
- **Better cross-platform compatibility,** allowing users to access files from Windows, macOS, and Linux.

This case study demonstrates how **Node.js directories and paths** can be managed efficiently in **large-scale applications**.

---

## Exercise

1. Write a Node.js script to create a directory named "testFolder".
2. Modify the script to check if the directory already exists before creating it.
3. Use path.join to create a file path pointing to "testFolder/data.txt".

---

## Conclusion

In this section, we explored:

✓ **How to create, delete, and read directories using the fs module.**

✓ **How to manage file paths across different operating systems using the path module.**

✓ **Real-world use cases for efficient file organization in applications.**

# File System Operations: Rename, Delete, Move Files

## Chapter 1: Introduction to File System Operations in Node.js

### 1.1 Understanding the Node.js File System Module

The **File System (fs) module** in Node.js provides built-in methods to interact with files and directories. Since Node.js is commonly used for backend development, working with the file system is essential for tasks such as:

- Reading and writing files
- Creating, renaming, deleting, and moving files
- Managing directories and their contents
- Handling asynchronous file operations efficiently

Node.js supports both **synchronous** and **asynchronous** file system operations:

- **Synchronous methods** block execution until the operation is completed.
- **Asynchronous methods** allow the program to continue executing other tasks while waiting for file operations to complete.

To use the file system module in Node.js, import it with:

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

# CHAPTER 2: RENAMING FILES IN NODE.JS

## 2.1 Using fs.rename() to Rename a File

The fs.rename() method is used to rename files or move them to a different location. This method is **asynchronous,** meaning it does not block the execution of other operations.

### Example: Renaming a File

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


fs.rename('oldfile.txt', 'newfile.txt', (err) => {
  if (err) {
    console.error('Error renaming file:', err);
  } else {
    console.log('File renamed successfully!');
  }
});
```

- The first parameter is the current filename (oldfile.txt).
- The second parameter is the new filename (newfile.txt).
- The callback function handles success or error cases.

## 2.2 Synchronous Method for Renaming a File

For scenarios where blocking execution is acceptable, use fs.renameSync():

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


try {
```

```
  fs.renameSync('oldfile.txt', 'newfile.txt');
  console.log('File renamed successfully!');
} catch (err) {
  console.error('Error renaming file:', err);
}
```

- The **synchronous method** does not require a callback but should be used cautiously in performance-sensitive applications.

---

## CHAPTER 3: DELETING FILES IN NODE.JS

### 3.1 Using fs.unlink() to Delete a File

The fs.unlink() method removes a file from the file system. This action is **irreversible**, so it should be used carefully.

**Example: Deleting a File**

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

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

- If the file does not exist, an error will be thrown.
- Always handle errors to prevent crashes.

### 3.2 Checking If a File Exists Before Deleting

To avoid errors when deleting non-existent files, check if the file exists using fs.existsSync():

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


if (fs.existsSync('sample.txt')) {
  fs.unlink('sample.txt', (err) => {
    if (err) {
      console.error('Error deleting file:', err);
    } else {
      console.log('File deleted successfully!');
    }
  });
} else {
  console.log('File does not exist.');
}
```

This ensures that the file deletion is only attempted if the file is present.

---

## CHAPTER 4: MOVING FILES IN NODE.JS

### 4.1 Moving Files Using fs.rename()

In Node.js, moving a file is achieved by **renaming** it with a new path.

**Example: Moving a File to Another Directory**

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


fs.rename('file.txt', 'new_directory/file.txt', (err) => {
  if (err) {
    console.error('Error moving file:', err);
  } else {
    console.log('File moved successfully!');
  }
});
```

- The new path specifies the target directory (new_directory/file.txt).
- If the directory does not exist, an error will occur.

**4.2 Creating the Directory Before Moving a File**

If the target directory is missing, create it first using fs.mkdir():

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


const newDir = 'new_directory';
const filePath = 'file.txt';
const newFilePath = `${newDir}/file.txt`;


// Ensure the directory exists
```

```
if (!fs.existsSync(newDir)) {
  fs.mkdirSync(newDir);
}

// Move the file
fs.rename(filePath, newFilePath, (err) => {
  if (err) {
    console.error('Error moving file:', err);
  } else {
    console.log('File moved successfully!');
  }
});
```

- fs.mkdirSync() ensures the target directory exists before moving the file.
- This prevents errors related to missing directories.

---

## Case Study: How a Media Storage Company Used Node.js to Automate File Management

### Background

A **media storage company** handling large volumes of images and videos needed an automated file management system to:

- **Organize files into categorized folders** (e.g., images, videos, documents).
- **Rename files based on user inputs** to maintain consistency.

- **Delete unnecessary temporary files** to free up storage space.

**Challenges**

- Manually organizing and renaming files took significant time.
- Large volumes of unused temporary files consumed excessive disk space.

**Solution: Implementing a Node.js Script for File Automation**

The company developed a **Node.js-based file automation script** that:

✓ Scanned a directory and **categorized files** into subfolders.
✓ Used fs.rename() to **rename files** based on pre-defined naming conventions.
✓ Used fs.unlink() to **delete old or unused files** to free up storage.

**Results**

- **80% reduction** in manual file handling efforts.
- Improved **disk space management,** freeing up **20GB+ per month**.
- Faster **file retrieval and organization,** enhancing user experience.

This case study highlights how **Node.js simplifies file system automation,** improving efficiency for businesses handling large amounts of data.

---

**Exercise**

1. Write a Node.js script to rename data.txt to backup.txt.
2. Modify the script to check if data.txt exists before renaming.

3. Write a function that deletes a file called logfile.log and logs a success message.
4. Move a file named report.pdf to a directory called archives.

---

## Conclusion

In this section, we explored:

✓ **How to rename files using fs.rename() and handle errors effectively**.

✓ **How to delete files using fs.unlink() while ensuring file existence**.

✓ **How to move files to new directories and create missing folders dynamically**.

# UNDERSTANDING STREAMS & BUFFERS

## CHAPTER 1: INTRODUCTION TO STREAMS AND BUFFERS

### 1.1 What Are Streams in Node.js?

Streams are an essential feature in **Node.js** that allow handling data efficiently, especially when dealing with large files, network connections, or real-time processing. Instead of reading or writing a file in one go, streams process data **chunk by chunk**, improving performance and memory usage.

**Key Features of Streams:**

- **Efficient Memory Usage** – Streams process data in chunks rather than loading the entire content into memory.
- **Improved Performance** – Since data is handled in smaller portions, applications run faster, especially when dealing with large files.
- **Asynchronous and Non-blocking** – Streams do not halt the execution of other tasks, making them suitable for high-performance applications.
- **Event-Driven Model** – Streams use event listeners (data, end, error, finish) to manage data flow.

**Example: Traditional File Reading vs. Streams**

**Without Streams (Loading Entire File into Memory)**

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


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

```
console.log(data);
```

- **Problem**: This method loads the entire file into memory at once. If the file is too large, it can crash the system.

**With Streams (Reading File in Chunks)**

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


const stream = fs.createReadStream('largeFile.txt', 'utf8');


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


stream.on('end', () => {
  console.log("File read complete.");
});
```

- **Solution**: This approach reads the file **chunk by chunk**, making it **memory-efficient**.

---

## CHAPTER 2: TYPES OF STREAMS IN NODE.JS

### 2.1 Four Types of Streams

Streams in Node.js come in four types, each serving a specific purpose:

1. **Readable Streams** – Used to read data from a source. Example: fs.createReadStream().

2. **Writable Streams** – Used to write data to a destination. Example: fs.createWriteStream().
3. **Duplex Streams** – Used for both reading and writing. Example: net.Socket (used in networking).
4. **Transform Streams** – A type of Duplex stream that modifies or transforms data. Example: zlib.createGzip() (used for compressing files).

**Example: Writable Stream**

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


const stream = fs.createWriteStream('output.txt');

stream.write('Hello, this is a writable stream!\n');

stream.end();

console.log("Data written successfully.");
```

- This writes data to output.txt using a **writable stream** instead of loading everything in memory first.

---

## CHAPTER 3: UNDERSTANDING BUFFERS

### 3.1 What Are Buffers in Node.js?

A **Buffer** is a temporary storage area for binary data. Since JavaScript (by default) does not support binary data, **Node.js introduced Buffers** to handle raw binary streams efficiently.

**Why Buffers Are Important:**

- **Used for handling binary data** – Buffers store raw data before processing.

- **Required for streams** – Streams use Buffers to process data in chunks.
- **No need for encoding** – Buffers allow manipulation of raw binary data without encoding conversions.

**Example: Creating a Buffer in Node.js**

```
const buffer = Buffer.from('Hello, Node.js');

console.log(buffer);
```

- This will output a **binary representation** of "Hello, Node.js".

---

### 3.2 Manipulating Buffers

Buffers can be manipulated in various ways, such as writing, reading, slicing, and converting to strings.

**Example: Buffer to String Conversion**

```
const buffer = Buffer.from('Hello, Node.js');

console.log(buffer.toString());
```

- Converts the **binary buffer** back to a **string**.

**Example: Allocating Buffers**

```
const buf = Buffer.alloc(10); // Creates a buffer of 10 bytes

console.log(buf); // <Buffer 00 00 00 00 00 00 00 00 00 00>
```

- Allocates a **10-byte** buffer filled with zeroes.

**Example: Writing Data to a Buffer**

```
const buf = Buffer.alloc(10);

buf.write('Hello');
```

```
console.log(buf.toString());
```

- Writes "Hello" into the buffer and retrieves it.

---

## Case Study: How Netflix Uses Streams for Video Streaming

### Background

Netflix is one of the world's largest streaming platforms, delivering **high-quality video content** to millions of users worldwide.

### Challenges

- Streaming large videos requires **huge memory resources** if loaded all at once.
- Users expect **seamless playback** without buffering delays.
- The system must **handle multiple concurrent requests efficiently**.

### Solution: Implementing Streams in Netflix

Netflix adopted **Node.js streams** to improve video delivery:

- **Chunk-based video streaming** – Videos are streamed **in small chunks** instead of loading the entire file at once.
- **Lower memory usage** – Using streams prevents server crashes due to memory overload.
- **Faster content delivery** – Instead of waiting for the full download, **users start watching immediately**.

### Results

- **40% reduction in server load** due to efficient memory management.

- **Seamless user experience,** even with fluctuating internet speeds.
- **Scalable performance,** handling millions of simultaneous video streams.

Netflix's use of **Node.js Streams** showcases their ability to **handle large-scale applications efficiently**.

---

## Exercise

1. What is the difference between **Readable Streams** and **Writable Streams**?
2. How do **Buffers** help in handling binary data in Node.js?
3. Write a simple Node.js script that reads a file using a **Readable Stream**.

---

## Conclusion

In this section, we explored:

**✓ Streams and their role in handling large data efficiently.**
**✓ The four types of streams: Readable, Writable, Duplex, and Transform.**
**✓ Buffers and how they enable handling of raw binary data in Node.js.**

# WORKING WITH READABLE & WRITABLE STREAMS IN NODE.JS

## CHAPTER 1: INTRODUCTION TO STREAMS IN NODE.JS

### 1.1 Understanding Streams and Their Importance

Streams in Node.js are a powerful way to handle **large data efficiently** without consuming excessive memory. Unlike traditional file reading/writing methods that load entire data into memory before processing, **streams process data in chunks**, making them ideal for **handling large files, network communications, and real-time applications**.

**Key Benefits of Streams:**

✓ **Efficient Memory Usage** – Streams process data in small chunks instead of loading the entire file into memory.

✓ **Faster Processing** – Streams start processing data immediately rather than waiting for the complete file to load.

✓ **Pipelining** – Allows chaining multiple stream operations, improving performance.

### 1.2 Types of Streams in Node.js

Node.js provides **four types of streams**:

1. **Readable Streams** – Used for reading data from a source (e.g., files, HTTP requests).
2. **Writable Streams** – Used for writing data to a destination (e.g., files, HTTP responses).
3. **Duplex Streams** – Both readable and writable (e.g., sockets, network connections).

4. **Transform Streams** – Used to modify data as it passes through (e.g., compressing files).

This section focuses on **Readable and Writable Streams**, which are essential for efficient file and network operations in Node.js.

---

## CHAPTER 2: WORKING WITH READABLE STREAMS

### 2.1 Creating a Readable Stream

Readable streams are used to **read** data from a source in chunks. Instead of loading the entire file, data is processed incrementally.

**Example: Reading a File Using Readable Streams**

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


const readableStream = fs.createReadStream('largeFile.txt', 'utf8');


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


readableStream.on('end', () => {
  console.log('Finished reading the file.');
});


readableStream.on('error', (err) => {
```

```
  console.error('Error reading file:', err);
});
```

### 2.2 Explanation of the Readable Stream Code

- fs.createReadStream('largeFile.txt', 'utf8') creates a **stream** to read largeFile.txt in chunks.
- .on('data', callback) is triggered **each time a chunk of data** is received.
- .on('end', callback) is triggered when the entire file has been read.
- .on('error', callback) handles errors (e.g., file not found).

### 2.3 Advantages of Readable Streams

✓ **Handles large files without high memory usage**.
✓ **Starts processing data immediately** instead of waiting for the entire file to load.
✓ **Improves performance in real-time applications** such as video streaming.

---

## CHAPTER 3: WORKING WITH WRITABLE STREAMS

### 3.1 Creating a Writable Stream

Writable streams allow data to be **written to a file or another output** incrementally, instead of writing everything at once.

### Example: Writing Data Using Writable Streams

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


const writableStream = fs.createWriteStream('output.txt');
```

```
writableStream.write('Hello, this is a writable stream!\n');

writableStream.write('Writing more data in chunks.\n');


writableStream.end(() => {
  console.log('Finished writing to the file.');
});


writableStream.on('error', (err) => {
  console.error('Error writing file:', err);
});
```

### 3.2 Explanation of the Writable Stream Code

- fs.createWriteStream('output.txt') creates a writable stream to write data to output.txt.
- .write(data) writes data **in chunks** instead of loading everything into memory.
- .end(callback) closes the stream and triggers the callback when writing is complete.
- .on('error', callback) handles errors during writing.

### 3.3 Advantages of Writable Streams

**✓ Efficient for logging, data storage, and streaming.**
**✓ Allows writing large amounts of data without blocking execution.**
**✓ Prevents memory overflows by writing in chunks.**

# CHAPTER 4: PIPING READABLE AND WRITABLE STREAMS

## 4.1 What is Stream Piping?

Piping allows **connecting a readable stream directly to a writable stream,** making it easier to handle large data transfers efficiently. Instead of manually handling chunks, **piping automatically transfers data from one stream to another**.

**Example: Copying a File Using Streams**

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


const readableStream = fs.createReadStream('input.txt');

const writableStream = fs.createWriteStream('output.txt');


readableStream.pipe(writableStream);


readableStream.on('end', () => {

  console.log('File copied successfully!');

});
```

## 4.2 Explanation of Piping

- readableStream.pipe(writableStream) **automatically** transfers data from input.txt to output.txt.
- It eliminates the need to manually handle chunks, improving readability.

- Once the reading is complete, on('end') confirms the operation's success.

### 4.3 Benefits of Stream Piping

✓ **Reduces code complexity by automating data transfer**.
✓ **Efficiently processes large files with minimal memory usage**.
✓ **Essential for real-time applications such as video or audio streaming**.

---

### Case Study: How YouTube Uses Streams for Video Processing

#### Background

YouTube processes **millions of video uploads daily,** requiring an efficient system for handling large video files.

#### Challenges

- **High memory consumption** due to loading entire videos before processing.
- **Slow file transfers** affecting the upload experience.
- **Scalability issues** when handling thousands of simultaneous uploads.

#### Solution: Implementing Readable & Writable Streams

YouTube optimized its backend by:

✓ Using **Readable Streams** to process video files in chunks instead of loading them fully.
✓ Using **Writable Streams** to save video data incrementally, reducing server load.
✓ Using **Piping** to efficiently transfer video data from the client to cloud storage.

This approach significantly **reduced memory usage, improved upload speeds, and enhanced scalability**.

---

## Exercise

1. Modify the following code to read a file using a readable stream:

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

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

console.log(data);
```

2. Write a writable stream that appends the text "Node.js Streams are powerful!" to a file named logs.txt.

---

## Conclusion

In this section, we explored:

✓ **How Readable Streams process large files efficiently**.

✓ **How Writable Streams allow writing data in chunks instead of all at once**.

✓ **How Piping simplifies transferring data between streams**.

✓ **How large-scale applications like YouTube use streams to handle big data efficiently**.

# IMPLEMENTING THE EVENT EMITTER PATTERN IN NODE.JS

## CHAPTER 1: INTRODUCTION TO EVENT-DRIVEN PROGRAMMING IN NODE.JS

### 1.1 Understanding the Event Emitter Pattern

The **Event Emitter Pattern** is a fundamental concept in **Node.js** that enables communication between different parts of an application using events. Unlike traditional **synchronous programming**, where functions execute sequentially, event-driven programming allows functions to respond to events **asynchronously** without blocking execution.

Key advantages of the **Event Emitter Pattern** include:

- **Non-blocking execution** – Events allow multiple tasks to run concurrently.
- **Better scalability** – Useful in large applications where components need to communicate efficiently.
- **Improved modularity** – Allows decoupling of logic, making code easier to maintain.

Node.js provides the **events module**, which includes the EventEmitter class for handling events.

**Common Use Cases of Event Emitters in Node.js:**

✓ **Handling user actions in real-time applications** (e.g., chat apps).
✓ **Listening for system events** (e.g., file changes, network requests).
✓ **Creating custom events** for modular programming.

# CHAPTER 2: WORKING WITH THE EVENTEMITTER CLASS

## 2.1 Creating and Emitting Events

To use the Event Emitter Pattern, we need to:

1. Import the events module.
2. Create an instance of EventEmitter.
3. Define and emit custom events.

**Example: Creating an Event and Emitting It**

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


// Create an instance of EventEmitter
const myEmitter = new EventEmitter();


// Define an event listener
myEmitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});


// Emit the event
myEmitter.emit('greet', 'Alice');
```

✓ The .on() method listens for the "greet" event.
✓ The .emit() method triggers the event and passes "Alice" as an argument.

### 2.2 Emitting Events Multiple Times

Events can be emitted multiple times, and each emission will trigger the listener:

myEmitter.emit('greet', 'Bob');

myEmitter.emit('greet', 'Charlie');

✓ This prints greetings for "Bob" and "Charlie", demonstrating event **reusability**.

---

## CHAPTER 3: HANDLING EVENTS MORE EFFICIENTLY

### 3.1 Using once() for One-Time Event Listeners

Sometimes, an event should only be handled **once**, such as **logging in** or **initializing a service**. The once() method ensures the listener executes only the first time an event is emitted.

**Example: Handling an Event Only Once**

```
const oneTimeEmitter = new EventEmitter();


oneTimeEmitter.once('init', () => {
  console.log('This will only run once.');
});


oneTimeEmitter.emit('init'); // Will execute
oneTimeEmitter.emit('init'); // Will NOT execute
```

✓ The second emission of "init" is ignored since .once() was used.

### 3.2 Removing Event Listeners (removeListener)

To **stop listening** for an event, use .removeListener() or .off().

**Example: Removing an Event Listener**

```
const callback = () => console.log("Event triggered!");

myEmitter.on('test', callback);


// Remove the event listener

myEmitter.removeListener('test', callback);


myEmitter.emit('test'); // Nothing happens
```

✓ This prevents memory leaks by removing unnecessary listeners.

---

## CHAPTER 4: EXTENDING EVENTEMITTER FOR CUSTOM CLASSES

### 4.1 Creating a Custom Class with EventEmitter

Many Node.js applications use **classes that extend EventEmitter** for structured event handling.

**Example: Creating a Custom EventEmitter Class**

```
class Person extends EventEmitter {
  constructor(name) {
    super();
    this.name = name;
  }

  introduce() {
```

```
    console.log(`Hi, my name is ${this.name}.`);
    this.emit('introduced', this.name);
  }
}

// Create a new person instance
const alice = new Person('Alice');

// Listen for the 'introduced' event
alice.on('introduced', (name) => {
  console.log(`Nice to meet you, ${name}!`);
});

// Call the method that emits an event
alice.introduce();
```

✓ The Person class extends EventEmitter, allowing it to **emit events within an object-oriented structure**.

---

## Case Study: How Event Emitters Power Real-Time Chat Applications

### Background

A startup developed a **real-time chat application** but struggled with performance issues caused by **synchronous message handling**.

Messages were delayed, and the server struggled to handle multiple users at once.

### Challenges

- **Blocking execution** caused delays in message processing.
- **Hard-to-maintain synchronous code** led to poor scalability.
- **Increased server load** resulted in slow response times.

### Solution: Implementing the Event Emitter Pattern

The development team redesigned the chat system using **Event Emitters**:

- Used .on('message') to **listen for incoming messages**.
- Emitted events asynchronously to **prevent blocking execution**.
- Optimized performance by **removing inactive event listeners**.

### Results

✓ **Instant message delivery** with no delays.
✓ **Improved server efficiency** by handling multiple users asynchronously.
✓ **Scalability increased,** allowing thousands of concurrent users.

By leveraging the **Event Emitter Pattern,** the team **transformed a laggy chat application into a real-time messaging system**.

---

### Exercise

1. Create an EventEmitter instance and define an event called "welcome" that logs "Welcome to Node.js!" when emitted.

2. Modify the script to accept a username as an event parameter and display "Welcome, [username]!".
3. Convert the event to execute only **once** using .once().

---

## Conclusion

In this section, we explored:

✓ **The Event Emitter Pattern and its role in event-driven programming**.

✓ **How to create, emit, and handle events using Node.js**.

✓ **How to extend EventEmitter to build scalable, modular applications**.

# ASSIGNMENT:

## CREATE A FILE-BASED LOGGING SYSTEM USING NODE.JS STREAMS

# SOLUTION GUIDE: CREATE A FILE-BASED LOGGING SYSTEM USING NODE.JS STREAMS

## Step 1: Set Up Your Node.js Environment

### 1.1 Create a Project Directory

Open a terminal and run the following commands to create a new project folder and navigate into it:

mkdir node-logger

cd node-logger

### 1.2 Initialize a Node.js Project

To create a package.json file, run:

npm init -y

This will generate a default package.json file.

### 1.3 Create a JavaScript File for Logging

Run the following command to create the main script file:

touch logger.js

## Step 2: Implement the Logging System

### 2.1 Import the File System Module

Open logger.js and start by importing the **fs module**:

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

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

### 2.2 Define the Log File Path

```
const logFilePath = path.join(__dirname, 'logs.txt');
```

- __dirname ensures the log file is created in the current directory.
- logs.txt will store all log messages.

### 2.3 Create a Write Stream for Logging

Instead of writing to a file synchronously, we use **streams** for better performance:

```
const logStream = fs.createWriteStream(logFilePath, { flags: 'a' });
```

- { flags: 'a' } ensures that new logs are **appended** instead of overwriting existing content.

### 2.4 Create a Function to Log Messages

```
function logMessage(message) {
  const timestamp = new Date().toISOString();
  logStream.write(`[${timestamp}] ${message}\n`);
}
```

- The **timestamp** ensures each log entry is time-stamped.
- writeStream.write() efficiently adds log entries to the file.

### 2.5 Test the Logging Function

```
logMessage("Application started.");
logMessage("User logged in.");
logMessage("An error occurred: Unable to fetch data.");
```

Run the script with:

node logger.js

Check logs.txt to see the log entries:

cat logs.txt

Example output:

[2025-03-03T14:00:00.123Z] Application started.

[2025-03-03T14:05:32.456Z] User logged in.

[2025-03-03T14:10:15.789Z] An error occurred: Unable to fetch data.

---

**Step 3: Enhance Logging with Different Log Levels**

**3.1 Add Log Levels (INFO, WARN, ERROR)**

Modify the logMessage() function to support log levels:

```
function logMessage(level, message) {
   const timestamp = new Date().toISOString();
   logStream.write(`[${timestamp}] [${level.toUpperCase()}] ${message}\n`);
}
```

**3.2 Log Different Types of Messages**

logMessage("INFO", "Server started successfully.");

logMessage("WARN", "High memory usage detected.");

logMessage("ERROR", "Database connection failed.");

Now, the log file will contain:

[2025-03-03T14:20:00.123Z] [INFO] Server started successfully.

[2025-03-03T14:22:15.456Z] [WARN] High memory usage detected.

[2025-03-03T14:25:45.789Z] [ERROR] Database connection failed.

---

### Step 4: Automatically Rotate Logs Based on File Size

To prevent the log file from growing too large, we can **rotate** it once it exceeds a certain size (e.g., 5MB).

#### 4.1 Check File Size Before Writing Logs

Modify the logMessage() function to rotate the log file:

```
function rotateLogs() {
  const stats = fs.statSync(logFilePath);

  if (stats.size > 5 * 1024 * 1024) { // 5MB limit
    const newLogFilePath = `logs-${Date.now()}.txt`;
    fs.renameSync(logFilePath, newLogFilePath);
    logStream.end(); // Close the existing stream
    logStream = fs.createWriteStream(logFilePath, { flags: 'a' }); // Create a new log file
  }
}

function logMessage(level, message) {
  rotateLogs(); // Check if rotation is needed before writing logs
  const timestamp = new Date().toISOString();
```

```
  logStream.write(`[${timestamp}] [${level.toUpperCase()}] ${message}\n`);
}
```

Now, once logs.txt exceeds 5MB, it will be renamed to logs-<timestamp>.txt, and a new logs.txt will be created automatically.

---

**Step 5: Export the Logger for Reuse in Other Files**

To use the logger in multiple files, create logger.js as a module:

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

const logFilePath = path.join(__dirname, 'logs.txt');
let logStream = fs.createWriteStream(logFilePath, { flags: 'a' });

function logMessage(level, message) {
  const timestamp = new Date().toISOString();
  logStream.write(`[${timestamp}] [${level.toUpperCase()}] ${message}\n`);
}

module.exports = logMessage;
```

Now, you can use it in any Node.js project:

```
const logMessage = require('./logger');
```

```
logMessage("INFO", "Module imported successfully.");
```

---

## Conclusion

✓ We built a **file-based logging system** using Node.js streams.
✓ We implemented **log levels** for better categorization.
✓ We added **log rotation** to prevent file size issues.
✓ We made the logger **reusable across different files**.

This logging system can be further improved by:

- Integrating with **cloud storage (AWS, Google Cloud, etc.)**.
- Sending **real-time log alerts via email or Slack**.
- Storing logs in a **database for analytics and debugging**.