fs Module in node.js

Topics

1. Introduction to File System:
Introduction to the fs module:
Explanation of the fs module and its role in the Node.js runtime
Overview of the functions provided by the fs module
2. Reading and Writing Files:
fs.readFile()
fs.writeFile()
fs.appendFile()
Reading and writing binary data
3. Working with Directories:
fs.mkdir()
fs.rmdir()
fs.readdir()
4. File and Directory Information:
fs.stat()
fs.lstat()
fs.fstat()
5. File System Watchers:
fs.watch()
fs.watchFile()
fs.unwatchFile()
6. File System Streams:
fs.createReadStream()
fs.createWriteStream()
7. File System Operations:
fs.rename()
fs.truncate()
fs.unlink()
fs.link()
fs.symlink()

1. Introduction to file system

What is the fs Module?

The fs (File System) module is a built-in Node.js module that allows you to interact with the file system. You can use it to:

- Read and write files.
- Create and remove directories.
- Get information about files and directories.
- Watch for changes in files or directories.

> **Note:** The fs module provides both synchronous and asynchronous methods. Asynchronous methods are non-blocking and preferred for scalable applications.

Importing the fs Module

javascript

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

Synchronous vs. Asynchronous

- **Synchronous (Blocking):** Waits for a task to finish before moving on.
- **Asynchronous (Non-Blocking):** Doesn't wait; it uses callbacks or promises.

Example: Reading a file synchronously vs. asynchronously.

```
// Synchronous (Blocking)
 const data = fs.readFileSync('example.txt', 'utf8');
 console.log('Synchronous Read:', data);

 // Asynchronous (Non-Blocking)
 fs.readFile('example.txt', 'utf8', (err, data) => {
   if (err) throw err;
   console.log('Asynchronous Read:', data);
 });
```

---

2. Reading and Writing Files with the fs Module in Node.js

Let's explore how to read and write files using the fs module.

✅ a. Reading Files

Asynchronous Read (Recommended for non-blocking code)

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

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

```
        return;
    }
    console.log('Asynchronous Read:', data);
});
```

Synchronous Read (Blocks the execution until the file is read)

```
const data = fs.readFileSync('example.txt', 'utf8');
console.log('Synchronous Read:', data);
```

🔄 *Use synchronous methods only for simple scripts or initialization tasks.*

✅ b. Writing Files

a) Asynchronous Write (Creates the file if it doesn’t exist or overwrites it)

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

b) Synchronous Write

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

✅ c. Appending Data to Files

a) Asynchronous append

```
fs.appendFile('example.txt', '\nThis is appended content.', (err) => {
    if (err) {
        console.error('Error appending to file:', err);
        return;
    }
    console.log('Content appended successfully!');
});
```

b) Synchronous append

```
        const filename = 'example.txt';
       const dataToAppend = '\nThis is some appended data (synchronously).';

// Append data to the file synchronously
```

```
try {
    fs.appendFileSync(filename, dataToAppend);
    console.log('Data appended successfully!');
} catch (err) {
    console.error('Error appending data:', err);
}
```

✅ d. Reading and Writing Binary Data

```
const buffer = Buffer.from('Binary data example', 'utf-8');
fs.writeFile('binary.dat', buffer, (err) => {
    if (err) throw err;
    console.log('Binary data written!');
});

fs.readFile('binary.dat', (err, data) => {
    if (err) throw err;
    console.log('Binary Data:', data.toString());
});
```

✅ Quick Tips

- Always handle errors to avoid unexpected crashes.
- Use asynchronous methods for better performance in production environments

---

3. Working with Directories in Node.js using the fs Module

Let's explore how to manage directories—creating, reading, and removing them.

✅ a. Creating a Directory

a) Asynchronous Method

javascript

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

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

b) Synchronous Method

javascript

```
fs.mkdirSync('newDir');
console.log('Directory created successfully!');
```

🔄 *Use the { recursive: true } option to avoid errors if the directory already exists.*

javascript

```
fs.mkdir('newDir', { recursive: true }, (err) => {
  if (err) throw err;
  console.log('Directory created (or already exists).');
});
```

✅ b. Reading the Contents of a Directory

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

✅ c. Removing a Directory

Note: The directory must be empty to remove it.

a) Asynchronous Method

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

b) Synchronous Method

```
fs.rmdirSync('newDir');
console.log('Directory removed successfully!');
```

To remove a directory with its content (Node.js 12+):

```
fs.rm('newDir', { recursive: true, force: true }, (err) => {
  if (err) throw err;
  console.log('Directory and its contents removed!');
});
```

✅ Quick Tips

- Always check for directory existence before performing operations.
- Use fs.existsSync() to check if a directory exists.

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

---

4. ***File and Directory Information using the fs Module***

Node.js provides methods to retrieve metadata about files and directories using the fs module. Let's explore them with examples!

✅ a. fs.stat() – Get File or Directory Info

This method retrieves the status of a file or directory.

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

fs.stat('example.txt', (err, stats) => {
    if (err) {
        console.error('Error fetching stats:', err);
        return;
    }
    console.log('Is File:', stats.isFile());
    console.log('Is Directory:', stats.isDirectory());
    console.log('Size:', stats.size, 'bytes');
    console.log('Created At:', stats.birthtime);
    console.log('Last Modified At:', stats.mtime);
});
```

✅ b. fs.lstat() – Info about Symbolic Links

Similar to fs.stat() but used to get information about symbolic links.

```
fs.lstat('example.txt', (err, stats) => {
    if (err) {
        console.error('Error fetching lstat:', err);
        return;
    }
    console.log('Is Symbolic Link:', stats.isSymbolicLink());
    console.log('Is Directory:', stats.isDirectory());
});
```

✅ c. fs.fstat() – Info Using File Descriptors

This method fetches stats using a file descriptor.

```
fs.open('example.txt', 'r', (err, fd) => {
  if (err) {
    console.error('Error opening file:', err);
    return;
  }

  fs.fstat(fd, (err, stats) => {
    if (err) {
      console.error('Error fetching stats:', err);
      return;
    }
    console.log('File Size:', stats.size);
    fs.close(fd, (err) => {
      if (err) console.error('Error closing file:', err);
    });
  });
});
```

✅ Understanding the Stats Object

The `stats` object provides detailed information, such as:

- `.isFile()` → Checks if it's a file.
- `.isDirectory()` → Checks if it's a directory.
- `.isSymbolicLink()` → Checks if it's a symbolic link.
- `.size` → Size of the file in bytes.
- `.birthtime` → Creation time.
- `.mtime` → Last modified time.

✅ Quick Tip:

Use `fs.promises` for modern async/await syntax:

```
(async () => {
  try {
    const stats = await fs.promises.stat('example.txt');
    console.log('File Size:', stats.size);
  } catch (err) {
    console.error('Error:', err);
  }
})();
```

---

5. File System Watchers in Node.js

File system watchers allow you to monitor files or directories for changes like additions, deletions, or modifications. The `fs` module provides methods for this purpose.

✅ a. fs.watch() – Watch for Real-time Changes

This method monitors a file or directory and triggers an event when changes occur.

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

// Watching a file
fs.watch('example.txt', (eventType, filename) => {
  console.log(`Event Type: ${eventType}`);
  if (filename) {
    console.log(`File changed: ${filename}`);
  } else {
    console.log('Filename not provided');
  }
});

console.log('Watching for changes in example.txt...');
```

Notes:

- eventType can be 'rename' (file moved or deleted) or 'change' (file content changed).
- It works for both files and directories.

✅ b. fs.watchFile() – Polling-Based Watching

This method checks for changes at regular intervals, making it more consistent across platforms.

```
fs.watchFile('example.txt', (curr, prev) => {
  console.log('Previous Modification Time:', prev.mtime);
  console.log('Current Modification Time:', curr.mtime);
});

console.log('Watching example.txt with watchFile...');
```

Notes:

- It uses polling under the hood and is more reliable for frequent file updates.
- The callback receives two fs.Stats objects for comparison.

✅ c. fs.unwatchFile() – Stop Watching a File

Use this to stop watching a file set by fs.watchFile().

```
fs.unwatchFile('example.txt');
console.log('Stopped watching example.txt.');
```

✅ When to Use Which?

- Use **fs.watch()** for real-time and efficient watching.
- Use **fs.watchFile()** for environments where fs.watch() is unreliable (like network-mounted files).

✅ **Quick Tip:**
Always handle errors when working with file watchers to avoid unexpected crashes.

---

6. File System Streams in Node.js

Streams are a powerful way to handle reading and writing files, especially large ones. They process data chunk by chunk, which is memory efficient.

✅ a. fs.createReadStream() – Reading Files as Streams

This method reads large files without loading them entirely into memory.

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

// Create a readable stream
const readStream = fs.createReadStream('example.txt', 'utf8');

// Listen to the 'data' event to receive chunks
readStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk);
});

// Listen for the 'end' event when reading is complete
readStream.on('end', () => {
  console.log('No more data to read.');
});

// Handle errors
readStream.on('error', (err) => {
  console.error('Error reading file:', err);
});
```

✅ b. fs.createWriteStream() – Writing Files as Streams

This method writes data to a file in chunks.

```
 // Create a writable stream
const writeStream = fs.createWriteStream('output.txt');

// Write data to the file
writeStream.write('First chunk of data.\n');
writeStream.write('Second chunk of data.\n');

// Close the stream
```

```
writeStream.end(() => {
   console.log('Data written and stream closed.');
});

// Handle errors
writeStream.on('error', (err) => {
   console.log('Error writing file:', err);
});
```

✅ c. Piping Streams – Reading and Writing Together

Efficiently transfer data from one stream to another.

```
const read = fs.createReadStream('example.txt');
const write = fs.createWriteStream('output.txt');

// Pipe the read stream to the write stream
read.pipe(write);

write.on('finish', () => {
   console.log('File copied successfully using streams!');
});
```

🚀 Why Use Streams?

- **Memory Efficient:** Processes data in chunks.
- **Faster:** Doesn't wait for the entire data to be available.
- **Scalable:** Ideal for large files and real-time data processing.

✅ **Quick Tip:** Always handle errors when working with streams to avoid unexpected failures.

---

7. File System Operations in Node.js

Let's explore some essential file system operations using the fs module, like renaming, deleting, linking, and more.

✅ a. fs.rename() – Rename or Move Files

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

// Rename or move a file
fs.rename('oldName.txt', 'newName.txt', (err) => {
   if (err) {
      console.error('Error renaming file:', err);
      return;
   }
   console.log('File renamed successfully!');
});
```

✅ b. fs.truncate() – Truncate (Shorten) a File

This method is used to reduce the file size.

```
fs.truncate('example.txt', 10, (err) => {
  if (err) {
    console.error('Error truncating file:', err);
    return;
  }
  console.log('File truncated to 10 bytes!');
});
```

✅ c. fs.unlink() – Delete a File

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

✅ d. fs.link() – Create a Hard Link

A hard link is an additional name for an existing file.

```
fs.link('source.txt', 'hardlink.txt', (err) => {
  if (err) {
    console.error('Error creating hard link:', err);
    return;
  }
  console.log('Hard link created successfully!');
});
```

✅ e. fs.symlink() – Create a Symbolic Link

A symbolic (or soft) link is a reference to another file or directory.

```
fs.symlink('source.txt', 'symlink.txt', (err) => {
  if (err) {
    console.error('Error creating symbolic link:', err);
    return;
  }
  console.log('Symbolic link created successfully!');
});
```

Types of Links:

- **Hard Link:** Points directly to the data on the disk.
- **Symbolic Link:** Points to another file name (like a shortcut).

🚨 Handling Errors Gracefully

Always check for errors to prevent crashes.

```
fs.unlink('nonexistent.txt', (err) => {
  if (err && err.code === 'ENOENT') {
    console.log('File does not exist.');
  } else if (err) {
    throw err;
  } else {
    console.log('File deleted successfully.');
  }
});
```

✅ Quick Tip:

- Use `fs.promises` for `async/await` style coding.
- Always check if a file exists using `fs.existsSync()` before performing operations.

🎯 Project: Simple File Manager CLI

We'll build a basic Command Line Interface (CLI) that can:

1. **Create, read, and delete files.**
2. **Create and delete directories.**
3. **Rename or move files.**
4. **Watch for file changes.**