Node.js FS Module: Comprehensive Notes for Revision and Technical Interviews

This Markdown file provides detailed, technical yet simple notes on the Node.js `fs` (File System) module. Perfect for quick revision before interviews and hands-on practice. Every concept includes:

- **What** it is
- **Why** we use it
- **Real-world technical examples**
- **Practice code snippets** (copy-paste and run in Node.js)

---

1. Introduction to the FS Module

What is the FS Module?

The `fs` module allows Node.js programs to interact with the operating system's file system. It supports full CRUD operations (Create, Read, Update, Delete) on files and directories.

Why Use the FS Module?

- Essential for server-side tasks: reading config files, logging errors, handling uploads, serving static files, etc.
- Gives direct, low-level control over the file system.
- Offers both **synchronous** (blocking) and **asynchronous** (non-blocking) methods to suit different use cases.

How to Import

```
// CommonJS
const fs = require('node:fs');

// ES Modules
import fs from 'node:fs';
```

Interview Tip: The `node:` prefix ensures you're loading the built-in module, not a third-party package with the same name.

---

2. Synchronous Operations (Blocking)

What & Why?

Synchronous methods end with `Sync` and **block** the event loop until the operation finishes.
Use them in:

- Simple scripts
- CLI tools
- Startup tasks where order matters and performance isn't critical

Avoid in production servers — blocking kills concurrency.

Key Methods

1. Create/Overwrite File — `writeFileSync`

```
fs.writeFileSync('./data.txt', 'Hello Sync!');
console.log('File created/overwritten');
```

2. Read File — `readFileSync`

Returns Buffer by default. Use `'utf-8'` for string.

```
const content = fs.readFileSync('./data.txt', 'utf-8');
console.log(content); // Hello Sync!
```

Why utf-8? Backward compatible with ASCII + variable-length encoding (efficient for most text).

3. Append to File — `appendFileSync`

Adds data at the end. Creates file if missing.

```
fs.appendFileSync('./data.txt', '\nNew line appended');
```

4. Delete File — `unlinkSync`

```
try {
  fs.unlinkSync('./data.txt');
  console.log('File deleted');
} catch (err) {
  if (err.code === 'ENOENT') console.log('File not found');
}
```

5. Create Directory — `mkdirSync`

```
fs.mkdirSync('./logs/2025/12', { recursive: true }); // Creates nested folders
```

6. Remove Directory — `rmSync` (preferred) or `rmdirSync` (older)

```
fs.rmSync('./logs', { recursive: true, force: true });
```

7. Rename / Move File or Folder — `renameSync`

```
fs.renameSync('./old.txt', './new-name.txt');     // Rename
fs.renameSync('./file.txt', '../archive/file.txt'); // Move
```

8. Copy File — `copyFileSync`

```
fs.copyFileSync('./source.txt', './backup.txt');
```

---

3. Asynchronous Operations — Callbacks

What & Why?

Non-blocking. Use callbacks (error-first pattern).
Ideal for servers — keeps event loop free.

Key Methods

Read File

```
fs.readFile('./data.txt', 'utf-8', (err, data) => {
  if (err) return console.log(err);
  console.log(data);
});
console.log('This runs immediately!');
```

Write File

```
fs.writeFile('./async.txt', 'Async content', (err) => {
  if (err) console.log(err);
  console.log('File written');
});
```

Append File

```
fs.appendFile('./async.txt', '\nMore data', (err) => {
  if (err) console.log(err);
  console.log('Appended');
});
```

Ensuring Order (Avoid Callback Hell)

```
fs.writeFile('./order.txt', 'Step 1', (err) => {
  if (err) throw err;
  fs.appendFile('./order.txt', '\nStep 2', (err) => {
    if (err) throw err;
    fs.appendFile('./order.txt', '\nStep 3', () => {
      console.log('All steps done in order');
    });
  });
});
```

Interview Question:

Q: What is callback hell?

A: Deeply nested callbacks → hard to read/maintain. Solution: Promises or async/await.

---

4. Asynchronous Operations — Promises (`fs/promises`)

What & Why?

Cleaner than callbacks. Chain with `.then()` / `.catch()`.

```
const fsP = require('node:fs/promises');

// Write
fsP.writeFile('./promise.txt', 'Promise data')
  .then(() => console.log('Written'))
  .catch(err => console.log(err));

// Read
fsP.readFile('./promise.txt', 'utf-8')
  .then(data => console.log(data))
  .catch(err => console.log(err));

// Append
fsP.appendFile('./promise.txt', '\nAppended via promise');
```

---

5. Asynchronous Operations — Async/Await (Best for Modern Code)

What & Why?

Most readable. Looks synchronous but is non-blocking.

```
const fsP = require('node:fs/promises');

async function fileOps() {
  try {
    await fsP.writeFile('./await.txt', 'Start');
    await fsP.appendFile('./await.txt', '\nMiddle');
    await fsP.appendFile('./await.txt', '\nEnd');
    const content = await fsP.readFile('./await.txt', 'utf-8');
    console.log(content);
  } catch (err) {
    console.log('Error:', err);
  }
}

fileOps();
```

Interview Tip: Async functions return a Promise. `await` pauses execution inside the function but doesn't block the event loop.

---

6. Buffer in Node.js

What is Buffer?

Global class for handling raw binary data (fixed-size array of bytes).

Why Needed?

Files are binary. Text encoding (utf-8) converts to string, but sometimes you need raw bytes (images, videos, encryption).

```
const buf = Buffer.from('NodeJS');
console.log(buf);           // <Buffer 4e 6f 64 65 4a 53>
console.log(buf.toString()); // NodeJS

// Allocate empty buffer
const empty = Buffer.alloc(10);
empty.write('Hi');
console.log(empty.toString()); // Hi
```

Interview Question:

Q: `Buffer.alloc()` vs `Buffer.from()`?

A: `alloc(size)` creates zero-filled buffer. `from(data)` creates from string/array/etc.

---

7. Streams — The Most Efficient Way for Large Files

What are Streams?

Process data in chunks instead of loading everything into memory.

Why Use Streams?

- Low memory usage (critical for GB-sized files)
- Faster start (can begin processing first chunk immediately)
- Backpressure handling

Types

- **Readable** — `createReadStream`
- **Writable** — `createWriteStream`
- **Duplex** — both read & write
- **Transform** — modify data

Basic Readable Stream

```
const readStream = fs.createReadStream('./big-file.txt', {
  encoding: 'utf-8',
  highWaterMark: 1024 // 1KB chunks
});

readStream.on('data', (chunk) => {
  console.log('Chunk received:', chunk.length);
});
```

Piping (Best Way to Copy)

```
const read = fs.createReadStream('./source.txt');
const write = fs.createWriteStream('./copy.txt');

read.pipe(write); // Automatically handles chunks & backpressure
write.on('finish', () => console.log('Copy complete'));
```

Transform Example (Uppercase Converter)

```
const read = fs.createReadStream('./input.txt', { encoding: 'utf-8' });
const write = fs.createWriteStream('./upper.txt');

read.on('data', (chunk) => {
  write.write(chunk.toUpperCase());
});
```

Interview Question:
Q: Why use streams instead of `readFileSync` for large files?
A: `readFileSync` loads entire file → high memory usage + slow start. Streams use constant memory and start processing immediately.

1. What is the Node.js `fs` module and why is it important?

Answer:
The `fs` (File System) module provides an API to interact with the file system (read, write, delete files/folders). It's crucial for server-side applications that need to handle logs, configs, uploads, static assets, or any persistent data operations.

---

2. How do you import the `fs` module in Node.js?

Answer:

```
// CommonJS
const fs = require('node:fs');

// ES Modules
import fs from 'node:fs';
```

Using `node:` prefix ensures it's the built-in module.

---

3. What is the difference between synchronous and asynchronous `fs` methods?

Answer:

- Synchronous (`...Sync`) methods block the event loop until completion (e.g., `readFileSync`).
- Asynchronous methods are non-blocking and use callbacks/promises (e.g., `readFile`).
  Use async in servers for concurrency; sync in scripts for simplicity.

---

4. Why should you avoid synchronous `fs` methods in production servers?

Answer:
Node.js is single-threaded. Synchronous operations block the event loop, preventing it from handling other requests, timers, or I/O. This reduces throughput and can make the server unresponsive.

---

5. Explain `readFileSync` vs `readFile`.

Answer:

- `readFileSync`: Blocks execution, returns data directly.

- `readFile`: Non-blocking, takes a callback (or returns Promise in `fs/promises`).
  Example: `readFile` allows the server to serve other clients while reading a file.

---

6. What happens if you don't specify encoding in `readFileSync` or `readFile`?

Answer:
It returns a `Buffer` object (raw binary data) instead of a string. You need to call `.toString()` or specify `'utf-8'` to get readable text.

---

7. Why is UTF-8 the most commonly used encoding?

Answer:

- Backward compatible with ASCII (English characters use 1 byte).
- Variable-length encoding (1–4 bytes per character) — efficient for most languages.
- Default in Node.js for `toString()` on Buffers.

---

8. What is a Buffer in Node.js?

Answer:
A `Buffer` is a global class for handling raw binary data directly. It's like an array of bytes, used when dealing with files, networks, images, or streams where text encoding isn't appropriate.

---

9. Difference between `Buffer.from()` and `Buffer.alloc()`?

Answer:

- `Buffer.from('hello')`: Creates buffer from data (string, array, etc.).
- `Buffer.alloc(10)`: Creates a zero-filled buffer of fixed size (safe, no leftover data).
  Use `alloc` for security when initializing empty buffers.

---

10. How do you append data to a file synchronously and asynchronously?

Answer:
Sync: `fs.appendFileSync(path, data)`
Async: `fs.appendFile(path, data, callback)` or `fsP.appendFile()` with promises.

---

11. How do you create nested directories in one command?

Answer:
Use `{ recursive: true }`:

```
fs.mkdirSync('a/b/c', { recursive: true });
```

---

12. How do you delete a non-empty directory?

Answer:
Use `fs.rmSync(path, { recursive: true, force: true })` (modern Node.js) or older `fs.rmdirSync` with recursive.

---

13. What is callback hell? How does it relate to `fs` operations?

Answer:
Callback hell occurs when multiple async `fs` operations are nested for sequential execution:

```
fs.readFile(... , () => {
  fs.writeFile(... , () => {
    // deeply nested
  });
});
```

Solution: Use Promises or async/await.

---

14. How do you use `fs/promises` for cleaner async code?

Answer:

```
const fs = require('node:fs/promises');
await fs.writeFile('file.txt', 'data');
await fs.appendFile('file.txt', 'more');
const data = await fs.readFile('file.txt', 'utf-8');
```

---

15. What are streams in Node.js? Why are they important?

Answer:
Streams allow processing data in chunks instead of loading everything into memory. Critical for large files to avoid memory exhaustion and enable faster processing.

---

16. Name the four types of streams in Node.js.

Answer:

- Readable (e.g., file input)
- Writable (e.g., file output)
- Duplex (read + write, e.g., sockets)

- Transform (modify data while streaming)

---

17. How do you create a readable and writable stream?

Answer:

```
const readStream = fs.createReadStream('big.txt');
const writeStream = fs.createWriteStream('copy.txt');
```

---

18. What is piping? Give an example.

Answer:
Piping connects a readable stream to a writable stream automatically:

```
fs.createReadStream('source.txt').pipe(fs.createWriteStream('dest.txt'));
```

Best way to copy large files efficiently.

---

19. What is `highWaterMark` in streams?

Answer:
It controls the maximum chunk size (in bytes) a readable stream buffers. Default is 64KB. Lower it for slower consumers to manage backpressure.

---

20. Why are streams better than `readFile` for large files?

Answer:
`readFile` loads entire file into memory → high RAM usage + delay.
Streams process in chunks → constant memory, immediate start, better performance.

---

21. How do you handle errors in asynchronous `fs` operations?

Answer:
Always check the first `err` parameter in callbacks:

```
fs.readFile('file.txt', (err, data) => {
  if (err) return console.error(err);
  // use data
});
```

With promises: use try/catch or `.catch()`.

---

22. What does `ENOENT` error mean?

Answer:
"Error NO ENTry" — file or directory does not exist. Common when reading/deleting non-existent paths.

---

23. How do you copy a file using `fs`?

Answer:
Best way: `fs.copyFileSync(src, dest)` (sync) or use streams with `.pipe()` (async).

---

24. How do you rename and move a file in one operation?

Answer:
`fs.renameSync(oldPath, newPath)` works for both renaming and moving (even across directories).

---

25. What is backpressure in streams?

Answer:
When a readable stream produces data faster than the writable stream can consume it. Node.js streams handle it automatically with buffering and pausing.

---

26. Can you transform data while streaming? Example?

Answer:
Yes, using `.on('data')` and modifying chunks:

```
readStream.on('data', chunk => {
  writeStream.write(chunk.toString().toUpperCase());
});
```

Or use built-in `Transform` streams.

---

27. When would you use `fs.watch()`?

Answer:
To monitor file/directory changes (e.g., auto-reload config in dev, file watchers). Note: Not 100% reliable on all OS.

---

28. What is the best practice for file operations in modern Node.js?

Answer:
Use `fs/promises` with `async/await` for clean, readable, non-blocking code. Use streams for large files.

---

29. Explain how to read a large JSON file efficiently.

Answer:
Use streams + JSON parser (like `JSONStream` or manual chunk parsing). Avoid `readFileSync` + `JSON.parse()` — it loads everything into memory.

---

30. If a file operation fails halfway, how do you ensure data consistency?

Answer:

- Write to a temporary file, then `fs.rename()` (atomic on same filesystem).
- Use transactions (if using databases).
- Or implement rollback logic manually.

Tip for Interview: Always mention **non-blocking I/O**, **event loop impact**, and **memory efficiency** when discussing `fs` operations. These show deep understanding of Node.js architecture.

Key Interview Tips Summary

Topic	Key Point to Remember
Sync vs Async	Sync blocks event loop → bad for servers
Callback Hell	Nested callbacks → use Promises or async/await
UTF-8	Most popular encoding: backward compatible + efficient
Buffer	For raw binary data (images, crypto, etc.)
Streams	Best for large files: low memory, fast start, backpressure handling
Best Practice	Use `fs/promises` + async/await in modern Node.js