File-System

Introduction:

- *fs (File System) is a built-in Node.js module that allows interaction with files and directories.*
- *It supports both **synchronous** and **asynchronous** operations.*

Common Methods in fs Module:

1. ***File Read/Write:***
    - *fs.readFile(): Reads a file asynchronously.*
    - *fs.writeFile(): Writes data to a file asynchronously.*
    - *fs.readFileSync(): Reads a file synchronously.*
    - *fs.writeFileSync(): Writes data to a file synchronously.*
2. ***File/Directory Management:***
    - *fs.unlink(): Deletes a file.*
    - *fs.mkdir(): Creates a directory.*
    - *fs.rmdir(): Deletes a directory.*
    - *fs.rename(): Renames a file or directory.*
3. ***File/Directory Information:***
    - *fs.stat(): Provides details of a file/directory (size, creation time, etc.).*
    - *fs.existsSync(): Checks if a file or directory exists.*
4. ***Streams:***
    - *fs.createReadStream(): Reads large files efficiently using streams.*
    - *fs.createWriteStream(): Writes large files efficiently using streams.*

Examples:

1. Asynchronous File Read/Write:

```
const fs = require('fs');
// Reading a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('File read error:', err);
    return;
  }
  console.log('File data:', data);
});

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

2. Synchronous File Read/Write:

```
const fs = require('fs');
try {
  // Reading a file synchronously
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log('File data:', data);
} catch (err) {
  console.error('File read error:', err);
}
```

```
try {
   // Writing to a file synchronously
   fs.writeFileSync('example.txt', 'Hello, Node.js!');
   console.log('File written successfully!');
} catch (err) {
   console.error('File write error:', err);
}
```

Additional Operations:

- ***Append data to a file:***

```
fs.appendFile('example.txt', '\nAppending new content.', (err) => {
   if (err) throw err;
   console.log('Data appended successfully!');
});
```

- ***Rename a file:***

```
fs.rename('example.txt', 'renamed.txt', (err) => {
   if (err) throw err;
   console.log('File renamed successfully!');
});
```

- ***Delete a file:***

```
fs.unlink('renamed.txt', (err) => {
   if (err) throw err;
   console.log('File deleted successfully!');
});
```

- ***Create a directory:***

```
fs.mkdir('myFolder', (err) => {
   if (err) throw err;
   console.log('Directory created successfully!');
```

```
});
```

---

fs.stat() and fs.existsSync() in Node.js

---

1. fs.stat()

- *Purpose:*
  *➔ Retrieves **details** about a file or directory (like size, creation date, modified date, etc.).*
- *It works **asynchronously**.*
- ***Syntax:***

```
fs.stat(path, callback)
```

- ***Example:***

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

fs.stat("file1.txt", (err, stats) => {
 if (err) {
  console.error("Error fetching file stats:", err);
  return;
 }
 console.log("File Stats:", stats);
 console.log("Is file?", stats.isFile());
 console.log("Is directory?", stats.isDirectory());
 console.log("File Size:", stats.size, "bytes");
 console.log("Created on:", stats.birthtime);
});
```

- ***Output:***
- *File Stats: [object with all details]*

- *Is file? true*
- *Is directory? false*
- *File Size: 30 bytes*
- *Created on: 2025-04-28T12:00:00.000Z*

---

2. fs.existsSync()

- *Purpose:*
  *➔ **Synchronously checks** if a file or directory exists.*
- ***Syntax:***

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


if (fs.existsSync("file1.txt")) {
 console.log("File exists!");
} else {
 console.log("File does not exist!");
}
```

- ***Note:***
  - *It returns true if the file/directory exists.*
  - *It returns false if it doesn't.*
  - ***Synchronous**, so it **blocks** the code execution temporarily.*

Quick Difference:

Feature	***fs.stat()***	***fs.existsSync()***
Checks details	✅	❌
Checks existence	✅ *(indirectly, via error)*	✅
Type	*Asynchronous*	*Synchronous*
Return Value	*File information (stats object)*	*Boolean (true/false)*

4. Streams in Node.js

What are Streams?

- *Streams are used to **read** or **write** large amounts of data **efficiently**.*
- *Instead of reading/writing the entire file at once (which can crash memory for big files), streams **process small chunks** of data at a time.*
- *Very useful for handling **large files** like videos, big text files, etc., especially in real-world projects like your BrajYatraa website where server efficiency matters!*

Important Methods:

---

1. fs.createReadStream()

- ***Purpose:***
  *➔ Reads large files **chunk by chunk**.*
- ***Syntax:***

  const readStream = fs.createReadStream(path, options);
- ***Example:***

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

const readStream = fs.createReadStream('largefile.txt', 'utf8');

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

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

```
readStream.on('error', (err) => {
  console.error('Error while reading:', err);
});
```

---

2. fs.createWriteStream()

- ***Purpose:***
  ➔ *Writes large data into a file* ***chunk by chunk****.*
- ***Syntax:***

  ```
  const writeStream = fs.createWriteStream(path, options);
  ```

- ***Example:***

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

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

writeStream.write('Hello World!\n');
writeStream.write('Writing more data...\n');
writeStream.end();

writeStream.on('finish', () => {
console.log('Finished writing to file.');
});

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

Why use Streams?

Without Streams	***With Streams***
Loads entire file into memory	*Loads file part-by-part*
Memory crash for very big files	*Memory efficient*
Slower for huge files	*Faster and scalable*

Real-world use cases:

- *Uploading/downloading files on a web server.*
- *Sending video/audio files over the network.*
- *Writing logs continuously in real-time (very useful for server-side applications).*
