The fs module in Node.js is used to interact with the file system, enabling operations like reading, writing, updating, and deleting files and directories. Using the fs/promises version of the module allows you to leverage async/await, making asynchronous file operations more straightforward and readable compared to traditional callbacks.

**Importing fs/promises**

const fs = require("fs/promises");

The fs/promises API provides a promise-based interface to file system operations, making it compatible with async/await.

**Common File Operations Using async/await**

**1. Write to a File: fs.writeFile()**

Creates or overwrites a file with the specified content.

const writeFileExample = async () => {

try {

await fs.writeFile("example.txt", "Hello, World!", "utf-8");

console.log("File written successfully");

} catch (error) {

console.error("Error writing file:", error);

}

};

writeFileExample();

* **Parameters:**
  + path: File name or path.
  + data: Content to write.
  + encoding: Encoding format (optional, default is utf-8).

**2. Read from a File: fs.readFile()**

Reads the content of a file.

const readFileExample = async () => {

try {

const data = await fs.readFile("example.txt", "utf-8");

console.log("File content:", data);

} catch (error) {

console.error("Error reading file:", error);

}

};

readFileExample();

* **Parameters:**
  + path: File name or path.
  + encoding: Encoding format (optional, default is null, returns Buffer).

**3. Append to a File: fs.appendFile()**

Adds content to the end of a file without overwriting existing content.

const appendFileExample = async () => {

try {

await fs.appendFile("example.txt", "\nAdditional content.");

console.log("Content appended successfully");

} catch (error) {

console.error("Error appending file:", error);

}

};

appendFileExample();

* **Parameters:**
  + path: File name or path.
  + data: Content to append.

**4. Delete a File: fs.unlink()**

Removes a file.

const deleteFileExample = async () => {

try {

await fs.unlink("example.txt");

console.log("File deleted successfully");

} catch (error) {

console.error("Error deleting file:", error);

}

};

deleteFileExample();

**5. Check if a File Exists: fs.access()**

Tests permissions for a file or directory. Use it to check if a file exists.

const checkFileExample = async () => {

try {

await fs.access("example.txt");

console.log("File exists");

} catch (error) {

console.log("File does not exist");

}

};

checkFileExample();

* **Returns:** If the file exists, the promise resolves; otherwise, it rejects.

**6. Rename or Move a File: fs.rename()**

Renames or moves a file.

const renameFileExample = async () => {

try {

await fs.rename("example.txt", "new-example.txt");

console.log("File renamed successfully");

} catch (error) {

console.error("Error renaming file:", error);

}

};

renameFileExample();

**7. Create a Directory: fs.mkdir()**

Creates a new directory.

const createDirExample = async () => {

try {

await fs.mkdir("new-folder");

console.log("Directory created successfully");

} catch (error) {

console.error("Error creating directory:", error);

}

};

createDirExample();

**8. Read a Directory: fs.readdir()**

Lists the contents of a directory.

const readDirExample = async () => {

try {

const files = await fs.readdir(".");

console.log("Directory content:", files);

} catch (error) {

console.error("Error reading directory:", error);

}

};

readDirExample();

**9. Delete a Directory: fs.rmdir()**

Removes an empty directory.

const deleteDirExample = async () => {

try {

await fs.rmdir("new-folder");

console.log("Directory deleted successfully");

} catch (error) {

console.error("Error deleting directory:", error);

}

};

deleteDirExample();

**Advantages of async/await with fs/promises**

1. **Cleaner Code**: Removes nested callbacks, making the code more readable.
2. **Error Handling**: Use try-catch blocks to handle errors gracefully.
3. **Promise Chaining**: Works seamlessly with other promise-based APIs.

**Example: Combine Multiple Operations**

const fileOperations = async () => {

try {

await fs.writeFile("example.txt", "Initial content", "utf-8");

console.log("File created");

await fs.appendFile("example.txt", "\nAppended content");

console.log("Content appended");

const data = await fs.readFile("example.txt", "utf-8");

console.log("File content:", data);

await fs.rename("example.txt", "renamed-example.txt");

console.log("File renamed");

} catch (error) {

console.error("Error during file operations:", error);

}

};

fileOperations();

This approach provides a clear and intuitive way to handle file system operations in Node.js.