The FS Module

This module helps you with file handling operations such as:

- Reading a file (sync or async way)
- Writing to a file (sync or async way)
- Deleting a file
- Reading the contents of a director
- Renaming a file
- Watching for changes in a file, and much more

Let's perform some of these tasks to see the `fs` (File System) module in action below:

How to create a directory using `fs.mkdir()`

The `fs.mkdir()` function in Node.js is used to create a new directory. It takes two arguments: the path of the directory to be created and an optional callback function that gets executed when the operation is complete.

- **path**: Here, path refers to the location where you want to create a new folder. This can be an absolute or a relative path. In my case, the path to the present working directory (the folder I am currently in), is: `C:\Desktop\NodeJSTut`. So, Let's create a folder in the `NodeJSTut` directory called `myFolder`.

- **callback function:** The purpose of the callback function is to notify that the directory creation process has completed. This is necessary because the `fs.mkdir()` function is asynchronous, meaning that it does not block the execution of the rest of the code while the operation is in progress. Instead, it immediately returns control to the callback function, allowing it to continue executing other tasks. Once the directory has been created, the callback function is called with an error object (if any) and any other relevant data related to the operation. In the below code, we are just using it to display a success message in the console or any error.

```
// Import fs module
const fs = require('fs');

// Present Working Directory: C:\Desktop\NodeJSTut
// Making a new directory called ./myFolder:

fs.mkdir('./myFolder', (err) => {
    if(err){
        console.log(err);
    } else{
        console.log('Folder Created Successfully');
    }
})
```

After executing the above code, you will see a new folder called `myFolder` created in the `NodeJSTut` directory.

How to create and write to a file asynchronously using `fs.writeFile()`

After the `myFolder` directory is created successfully, it's time to create a file and write something to it by using the `fs` module.

There are basically 2 ways of doing this:

- **Synchronous Approach:** In this approach, we create a file and write the data to it in a blocking manner, which means that NodeJS waits for the creation and write operation to complete before moving on to the next line of code. If an error occurs during this process, it throws an exception that must be caught using `try...catch`.
- **Asynchronous Approach:** In this approach, we create and write data to a file in a non-blocking manner, which means that NodeJS does not wait for the write operation to complete before moving on to the next line of code. Instead, it takes a callback function that gets called once the entire process is completed. If an error occurs during the write operation, the error object is passed to the callback function.

In this tutorial, we will be using the `fs.writeFile()` function which follows the asynchronous approach.

`writeFile()` is a method provided by the `fs` (file system) module in Node.js. It is used to write data to a file asynchronously. The method takes three arguments:

1. The **path** of the file to write to (including the file name and extension)
2. The **data** to write to the file (as a string or buffer)
3. An optional **callback function** that is called once the write operation is complete or an error occurs during the write operation.

When `writeFile()` is called, Node.js creates a new file or overwrites an existing file at the specified **path**. It then writes the provided **data** to the file and closes it. Since the method is asynchronous, the write operation does not block the event loop, allowing other operations to be performed in the meantime.

Below is the code where we create a new file called `myFile.txt` in the `myFolder` directory and write this `data` to it: `Hi,this is newFile.txt`.

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

const data = "Hi,this is newFile.txt";

fs.writeFile('./myFolder/myFile.txt', data, (err)=> {
    if(err){
        console.log(err);
        return;
    } else {
        console.log('Writen to file successfully!');
    }
})
```

Since `newFile.txt` didn't exist previously, Hence the `writeFile()` function created this file for us on the provided path and then wrote the value in the `data` variable to the file. Suppose this file already existed. In that case, `writeFile()` will just open the file, erase all the existing text present in it and then write the data to it.

The problem with this code is: when you run the same code multiple times, it erases the previous data that is already present in `newFile.txt` and writes the data to it.

In case you do not want the original data to get deleted and just want the new data to be added/appended at the end of the file, you need to make a little change in the above code

by adding this "options object": `{flag: 'a'}` as the third parameter to `writeFile()` – like this:

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

const data = 'Hi,this is newFile.txt';

fs.writeFile('./myFolder/myFile.txt', data, {flag: 'a'}, (err) => {
    if(err){
        console.log(err);
        return;
    } else {
        console.log('Writen to file successfully!');
    }
})
```

Once you run the above code again and again, you will see that the `myFile.txt` has the value of the `data` variable written to it multiple times. This is because the object (3rd parameter): `{flag: 'a'}` indicates the `writeFile()` method to append the `data` at the end of the file instead of erasing the previous data present in it.

How to read a file asynchronously using `fs.readFile()`

After creating and writing to the file, it's time we learn how to read the data present in the file using the `fs` module.

Again there are 2 ways of doing this: Synchronous approach and the Asynchronous approach (just like the previous function). Here we are going to use the `readFile()` function provided by `fs` module which performs the reading operation asynchronously.

The `readFile()` function takes 3 parameters:

1. The **path** to the file which is to be read.
2. The **encoding** of the file.
3. The **callback function** that gets executed once the reading operation is completed or if any error occurs during the reading operation. It accepts 2 parameters: first parameter stores the file data (if read operation is successful) and the second parameter stores the error object (if read operation fails due to some error).

The `readFile()` function is very intuitive and once called, it reads the data present in the provided file according to the given encoding. If the read operation is successful, it returns the data to the callback function and if not, it will return the error occurred.

In the below code, we read the contents of the file - `myFile.txt` which we had created while learning the previous function and then log the data stored in it in the console.

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

fs.readFile('./myFolder/myFile.txt', {encoding: 'utf-8'}, (err, data) => {
    if(err){
        console.log(err);
        return;
    } else {
        console.log('File read successfully! Here is the data');
        console.log(data);
    }
})
```

It is to be noted here that the `encoding` property is set to `'utf-8'`. At this point, some of you may not know about the encoding property, So Let's understand it in a bit more detail:

The `encoding` parameter in the `fs.readFile()` method of Node.js is used to specify the character encoding used to interpret the file data. By default, if no `encoding` parameter is provided, the method returns a raw buffer.

If the `readFile()` method is called without providing an `encoding` parameter, you will see a result similar to this printed in the console:

This raw buffer is difficult to read and interpret as it represents the contents of the file in binary form. To convert the buffer to a readable string, you can specify an `encoding` parameter when calling `readFile()`.

In our case, we specified the `'utf8'` encoding as the second parameter of the `readFile()` method. This tells Node.js to interpret the file contents as a string using the UTF-8 character encoding, thus you see the original data printed in the console. Other common encodings that can be used with `readFile()` include:

- `'ascii'`: Interpret the file contents as ASCII-encoded text.
- `'utf16le'`: Interpret the file contents as 16-bit Unicode text in little-endian byte order.
- `'latin1'`: Interpret the file contents as ISO-8859-1 (also known as Latin-1) encoded text.

Reading and Writing to a File Synchronously

Up until now, you have learned how to write to and read the data from a file asynchronously. But there are synchronous alternatives to the 2 functions we learnt above, namely: `readFileSync()` and `writeFileSync()`.

Note that since these are synchronous operations, they need to be wrapped in a `try...catch` block. In case the operations fail for some reason, the errors thrown will be caught by the `catch` block.

In the below code, we first create a new file: `./myFolder/myFileSync.txt` and write to it using the `writeFileSync()` method. Then we read the contents of the file using the `readFileSync()` method and print the data in the console:

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

try{
    // Write to file synchronously
    fs.writeFileSync('./myFolder/myFileSync.txt', 'myFileSync says Hi');
    console.log('Write operation successful');

    // Read file synchronously
    const fileData = fs.readFileSync('./myFolder/myFileSync.txt', 'utf-8');
    console.log('Read operation successful. Here is the data:');
    console.log(fileData);

} catch(err){
    console.log('Error occurred!');
    console.log(err);
}
```

When you run the above code, a new file called `myFileSync.txt` is created in the `myFolder` directory and it contains the following text in it: `myFileSync says Hi`. This is the output printed in the console:

```
Write operation successful
Read operation successful. Here is the data:
myFileSync says Hi
```

How to read the contents of a directory using `fs.readdir()`

If you have been following along until now, you will see that we currently have 2 files in the `myFolder` directory, i.e, `myFile.txt` and `myFileSync.txt`. The `fs` module provides you

with `readdir()` function using which you can read the contents of a directory (the files and folders present in the directory).

The `readdir()` function accepts 2 parameters:

- The **path** of the folder whose contents are to be read.
- **Callback function** which gets executed once the operation is completed or if any error occurs during the operation. This function accepts 2 parameters: The first one which accepts the error object (if any error occurs) and the second parameter which accepts an array of the various files and folders present in the directory whose path has been provided.

In the code below, we are reading the contents of the `myFolder` directory and printing the result in the console.

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

fs.readdir('./myFolder', (err, files) => {
    if(err){
        console.log(err);
        return;
    }
    console.log('Directory read successfully! Here are the files:');
    console.log(files);
})
```

This is what we get as output when we run the above code:

```
[ 'myFile.txt', 'myFileSync.txt' ]
```

How to rename a file using `fs.rename()`

The `fs.rename()` method in Node.js is used to rename a file or directory. The method takes two arguments, the current file path and the new file path, and a callback function that is executed when the renaming is complete.

Here's the syntax for the `fs.rename()` method:

```
fs.rename(oldPath, newPath, callback);
```

where:

- `oldPath` (string) - The current file path
- `newPath` (string) - The new file path
- `callback` (function) - A callback function to be executed when the renaming is complete. This function takes an error object as its only parameter.

Let's rename the `newFile.txt` file to `newFileAsync.txt`:

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

fs.rename('./newFolder/newFile.txt', './newFolder/newFileAsync.txt', (err)=>{
    if(err){
        console.log(err);
        return;
    }
    console.log('File renamed successfully!')
})
```

Once you run the above code, you will see that the `newFile.txt` gets renamed to `newFileAsync.txt`.

Note that you should only provide valid paths (absolute or relative) to the `rename()` function and not just the names of the files. Remember it's `oldPath` and `newPath` and NOT `oldName` and `newName`.

For example, consider this code: `fs.rename('./newFolder/newFile.txt', 'newFileAsync.txt', ...rest of the code)`. In this case, since we did not provide a proper path in the 2nd parameter, `rename()` assumes that the path to the newly named file should be: `./newFileAsync.txt`. Thus, it basically removes the `newFile.txt` from the `newFolder` directory, renames the file to `newFileAsync.txt` and moves it to the current working directory.

How to delete a file using `fs.unlink()`

Last but not the least, we have the `fs.unlink()` function which is used to delete a file. It takes in 2 parameters:

- The path of the file which you want to delete, and
- The callback function which gets executed once the delete operation is over or if any error occurs during the operation.

Running the following code deletes the `newFileSync.txt` file present in the `myFolder` directory:

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

fs.unlink('./myFolder/myFileSync.txt', (err) => {
    if(err){
        console.log(err);
        return;
    }
    console.log('File Deleted Successfully!')
})
```