Node.js file system

The fs (File System) module in Node.js provides an **API for interacting with the file system**. It allows you to perform operations such as reading, writing, updating, and deleting files and directories, which are essential for server-side applications and scripts.

To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). All file system operations can have synchronous and asynchronous forms depending upon user requirements. To use this File System module, use the require() method:

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

Uses:

- Read Files
- Write Files
- Append Files
- Close Files
- Delete Files

Asynchronous approach:

They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself. The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command. While the previous command runs in the background and loads the result once it is finished processing the data.

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

// Asynchronous read

fs.readFile("input.txt", function (err, data) {

  if (err) {

    return console.error(err);

  }

  console.log("Asynchronous read: " + data.toString());

});
```

Synchronous approach:

They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

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

// Synchronous read

const data = fs.readFileSync('input.txt');

console.log("Synchronous read: " + data.toString());
```

Open a File

The fs.open() method is used to create, read, or write a file. The fs.readFile() method is only for reading the file and fs.writeFile() method is only for writing to the file, whereas fs.open() method does several operations on a file. First, we need to load the fs class which is a module to access the physical file system.

Syntax:

```
fs.open(path, flags, mode, callback)
```

Parameters:

- **path:** It holds the name of the file to read or the entire path if stored at other locations.
- **flags:** Flags indicate the behavior of the file to be opened. All possible values are (r, r+, rs, rs+, w, wx, w+, wx+, a, ax, a+, ax+).
- **mode:** Sets the mode of file i.e. r-read, w-write, r+ -readwrite. It sets to default as readwrite.
- **err:** If any error occurs.
- **data:** Contents of the file. It is called after the open operation is executed.

Example: Let us create a js file named **main.js** having the following code to open a file **input.txt** for reading and writing.

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

// Asynchronous - Opening File

console.log("opening file!");

fs.open("input.txt", "r+", function (err, fd) {

   if (err) {

      return console.error(err);
```

```
  }
  console.log("File open successfully");
});
```

Output:

```
opening file!
File open successfully
```

Reading a File

The fs.read() method is used to read the file specified by fd. This method reads the entire file into the buffer.

Syntax:

```
fs.read(fd, buffer, offset, length, position, callback)
```

Parameters:

- **fd:** This is the file descriptor returned by fs.open() method.
- **buffer:** This is the buffer that the data will be written to.
- **offset:** This is the offset in the buffer to start writing at.
- **length:** This is an integer specifying the number of bytes to read.
- **position:** This is an integer specifying where to begin reading from in the file. If the position is null, data will be read from the current file position.
- **callback:** It is a callback function that is called after reading of the file. It takes two parameters:
  - **err:** If any error occurs.
  - **data:** Contents of the file.

Example: Let us create a js file named **main.js** having the following code:

```
const fs = require("fs");
const buf = new Buffer(1024);
console.log("opening an existing file");
fs.open("input.txt", "r+", function (err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("reading the file");
  fs.read(fd, buf, 0, buf.length, 0, function (err, bytes) {
    if (err) {
      console.log(err);
    }
    console.log(bytes + " bytes read");
    // Print only read bytes to avoid junk.
    if (bytes > 0) {
      console.log(buf.slice(0, bytes).toString());
    }
  });
});
```

Writing to a File

This method will overwrite the file if the file already exists. The fs.writeFile() method is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists. The ‘options’ parameter can be used to modify the functionality of the method.

Syntax:

fs.writeFile(path, data, options, callback)

Parameters:

- **path:** It is a string, Buffer, URL, or file description integer that denotes the path of the file where it has to be written. Using a file descriptor will make it behave similarly to fs.write() method.
- **data:** It is a string, Buffer, TypedArray, or DataView that will be written to the file.
- **options:** It is a string or object that can be used to specify optional parameters that will affect the output. It has three optional parameters:
  - **encoding:** It is a string value that specifies the encoding of the file. The default value is ‘utf8’.
  - **mode:** It is an integer value that specifies the file mode. The default value is 0o666.
  - **flag:** It is a string value that specifies the flag used while writing to the file. The default value is ‘w’.
- **callback:** It is the function that would be called when the method is executed.

- **err:** It is an error that would be thrown if the operation fails.

Example: Let us create a js file named **main.js** having the following code:

```
const fs = require("fs");
console.log("writing into existing file");
fs.writeFile("input.txt", "Hello all", function (err) {
  if (err) {
    return console.error(err);
  }
  console.log("Data written successfully!");
  console.log("Let's read newly written data");
  fs.readFile("input.txt", function (err, data) {
    if (err) {
      return console.error(err);
    }
    console.log("Asynchronous read: " + data.toString());
  });
});
```

Appending to a File

The fs.appendFile() method is used to synchronously append the data to the file.

Syntax:

```
fs.appendFile(filepath, data, options, callback);
// or
fs.appendFileSync(filepath, data, options);
```

Parameters:

- **filepath:** It is a String that specifies the file path.
- **data:** It is mandatory and it contains the data that you append to the file.
- **options:** It is an optional parameter that specifies the encoding/mode/flag.
- **Callback:** Function is mandatory and is called when appending data to file is completed.

Example 1: Let us create a js file named **main.js** having the following code:

```
const fs = require("fs");
let data = "\nLearn Node.js";
// Append data to file
fs.appendFile(
  "input.txt", data, "utf8",
  // Callback function
  function (err) {
    if (err) throw err;

    // If no error
    console.log("Data is appended to file successfully.");
  }
);
```

Example 1: For synchronously appending

```
const fs = require("fs");
const data = "\nLearn Node.js";
// Append data to file
fs.appendFileSync("input.txt", data, "utf8");
console.log("Data is appended to file successfully.");
```

Closing the File

The fs.close() method is used to asynchronously close the given file descriptor thereby clearing the file that is associated with it. This will allow the file descriptor to be reused for other files. Calling fs.close() on a file descriptor while some other operation is being performed on it may lead to undefined behavior.

Syntax:

```
fs.close(fd, callback)
```

Parameters:

- **fd:** It is an integer that denotes the file descriptor of the file for which to be closed.
- **callback:** It is a function that would be called when the method is executed.

  - **err:** It is an error that would be thrown if the method fails.

Example: Let us create a js file named **main.js** having the following code:

```
// Close the opened file.
fs.close(fd, function (err) {
   if (err) {
      console.log(err);
   }
   console.log("File closed successfully.");
});
```

Delete a File

The fs.unlink() method is used to remove a file or symbolic link from the filesystem. This function does not work on directories, therefore it is recommended to use fs.rmdir() to remove a directory.

Syntax:

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

Parameters:

- **path:** It is a string, Buffer or URL which represents the file or symbolic link which has to be removed.
- **callback:** It is a function that would be called when the method is executed.
  - **err:** It is an error that would be thrown if the method fails.

Example: Let us create a js file named **main.js** having the following code:

```
const fs = require("fs");
console.log("deleting an existing file");
fs.unlink("input.txt", function (err) {
   if (err) {
      return console.error(err);
   }
   console.log("File deleted successfully!");
});
```

File- ReadStream, WriteStream

```
var fs = require('fs');
var grains = ['wheat', 'rice', 'oats'];
var options = { encoding: 'utf8', flag: 'w' };
var fileWriteStream = fs.createWriteStream("grains.txt", options);
fileWriteStream.on("close", function(){
console.log("File Closed.");
});
while (grains.length){
var data = grains.pop() + " ";
fileWriteStream.write(data);
console.log("Wrote: %s", data);
}
fileWriteStream.end();
var options = { encoding: 'utf8', flag: 'r' };
var fileReadStream = fs.createReadStream("grains.txt", options);
fileReadStream.on('data', function(txt) {
console.log('Grains: %s', txt);
console.log('Read %d bytes of data.', txt.length);
});
```

Node.js Events

Node.js is built on an event-driven architecture that allows you to build highly scalable applications. Understanding the event-driven nature of Node.js and how to work with events is important for building efficient and responsive applications.

What Are Events in Node.js?

In **Node.js**, an event is an action or occurrence that the program can detect and handle. The event-driven architecture allows asynchronous programming, and your application becomes able to perform non-blocking operations. This means that while waiting for an operation to complete (like reading a file or making a network request), the application can continue processing other tasks.

EventEmitter Class

At the core of the Node.js event system is the EventEmitter class. This class allows objects to emit named events that can be listened to by other parts of your application. It is included in the built-in events module.

Key Features of EventEmitter:

- **Event Registration:** You can register listeners for specific events using the on() method.
- **Event Emission:** Use the emit() method to trigger an event and call all registered listeners for that event.
- **Asynchronous Execution:** Listeners can execute asynchronously, allowing other operations to continue while waiting for events.

Syntax:

```
const EventEmitter=require('events');

var eventEmitter=new EventEmitter();
```

Working with Events in Node.js

Step 1: Importing the Events Module

To start using events in your application, you need to import the events module and create an instance of the EventEmitter class.

```
const EventEmitter = require('events');

const myEmitter = new EventEmitter();
```

Step 2: Registering Event Listeners

You can register listeners for specific events using the on() method. The first argument is the event name, and the second argument is the callback function to be executed when the event is emitted.

```
myEmitter.on('event', () => {
  console.log('An event occurred!');
});
```

Step 3: Emitting Events

To trigger an event, use the emit() method with the event name as the first argument.

```
myEmitter.emit('event');  // Output: An event occurred!
```

Listening events

Before emitting any event, it must register functions(callbacks) to listen to the events.

Syntax:

```
eventEmitter.addListener(event, listener)

eventEmitter.on(event, listener)

eventEmitter.once(event, listener)
```

Removing Listener

The **eventEmitter.removeListener()** takes two argument event and listener, and removes that listener from the listeners array that is subscribed to that event.
While **eventEmitter.removeAllListeners()** removes all the listener from the array which are subscribed to the mentioned event.

Syntax:

```
eventEmitter.removeListener(event, listener)

eventEmitter.removeAllListeners([event])
```

Note:

- Removing the listener from the array will change the sequence of the listener’s array, hence it must be carefully used.
- The **eventEmitter.removeListener()** will remove at most one instance of the listener which is in front of the queue.

Program

```
// Importing events

const EventEmitter = require('events');


// Initializing event emitter instances

var eventEmitter = new EventEmitter();


var fun1 = (msg) => {

   console.log("Message from fun1: " + msg);

};
```

```
var fun2 = (msg) => {
   console.log("Message from fun2: " + msg);
};

// Registering fun1 and fun2
//eventEmitter.on('myEvent', fun1);
//eventEmitter.on('myEvent', fun2);

// Listening to myEvent with fun1 and fun2
eventEmitter.addListener('myEvent', fun1);
// fun2 will be inserted in front of listeners array
eventEmitter.prependListener('myEvent', fun2);

console.log(eventEmitter.listeners('myEvent'))
console.log(eventEmitter.listenerCount('myEvent'))
 // Triggering myEvent
eventEmitter.emit('myEvent', "Event occurred");
// Removing listener fun1 that was
// registered on the line 13
eventEmitter.removeListener('myEvent', fun1);
// Triggering myEvent
eventEmitter.emit('myEvent', "Event occurred");

// Removing all the listeners to myEvent
eventEmitter.removeAllListeners('myEvent');

// Triggering myEvent
eventEmitter.emit('myEvent', "Event occurred");
```

Asynchronous events

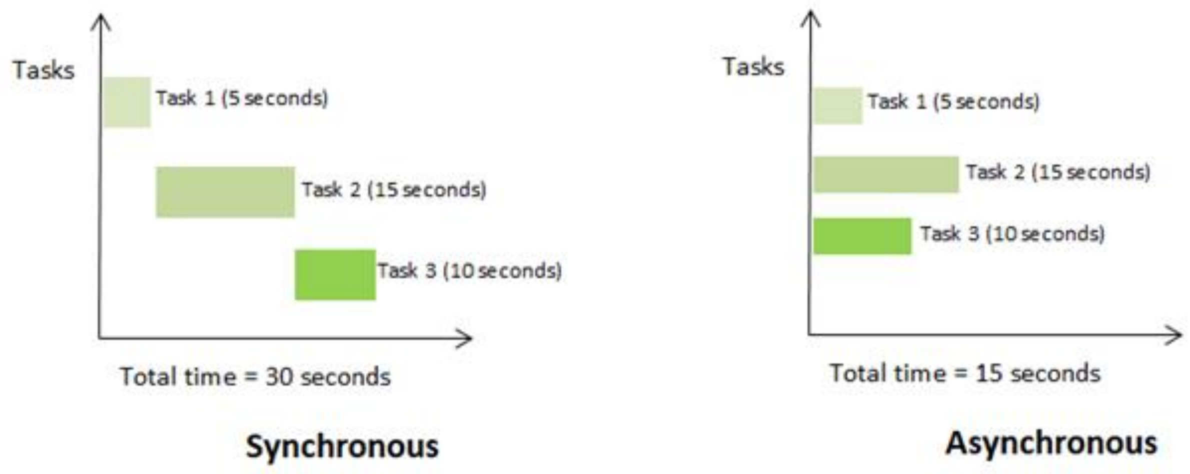
The EventEmitter calls all listeners synchronously in order to which they were registered. However, we can perform asynchronous calls by using **setImmediate()**

```
// Importing events

const EventEmitter = require('events');

// Initializing event emitter instances

var eventEmitter = new EventEmitter();

// Async function listening to myEvent

eventEmitter.on('myEvent', (msg) => {

  setImmediate( () => {

    console.log("Message from async1: " + msg);

  });

});

// Async function listening to myEvent

eventEmitter.on('myEvent', (msg) => {

  setImmediate( () => {

    console.log("Message from async2: " + msg);

  });

});


// Declaring listener fun to myEvent

var fun1 = (msg) => {

  console.log("Message from fun1: " + msg);

};

// Declaring listener fun to myEvent

var fun2 = (msg) => {

  console.log("Message from fun2: " + msg);

};

// Listening to myEvent with fun

eventEmitter.on('myEvent', fun1);

eventEmitter.on('myEvent', fun2);

// Triggering myEvent

eventEmitter.emit('myEvent', "Event occurred");
```

Synchronous and Asynchronous Programming in Node.js

Synchronous code is also called “blocking” as it halts the program until all the resources are available. Synchronous execution usually uses to code executing in sequence and the program is executed line by line, one line at a time. When a function is called, the program execution waits until that function returns before continuing to the next line of code.

Asynchronous code is also known as “non-blocking”. The program continues executing and doesn’t wait for external resources (I/O) to be available. Asynchronous execution applies to execution that doesn’t run in the sequence it appears in the code. The program doesn’t wait for the task to complete and can move on to the next task.



Synchronous code wastes around 90% of CPU cycles waiting for the network or disk to get the data, but the Asynchronous code is much more performing.

Using Asynchronous code is a more efficient to have concurrency without dealing with multiple execution threads.

Example

File Handling Sync Async

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

console.log("Reading synchronously");

data = fs.readFileSync("sample.txt");

console.log(data.toString());

console.log("Reading asynchronously");

fs.readFile('sample.txt', function (err, data) {

  if (err) {    return console.error(err); }

    console.log("Asynchronous read: " + data.toString());

});

console.log('Read operation complete.');
```

How to Write Asynchronous Function for Node.js ?

The asynchronous function can be written in Node.js using ‘async’ preceding the function name. The asynchronous function returns an implicit Promise as a result. The async function helps to write promise-based code asynchronously via the event loop. Async functions will always return a value.

Await function can be used inside the asynchronous function to wait for the promise. This forces the code to wait until the promise returns a result.

Install async using the following command:

```
npm i async
```

A Node. js **Promise** is a placeholder for a value that will be available in the future, allowing us to handle the result of an asynchronous task once it has completed or encountered an error. Promises make writing asynchronous code easier. Promises and async/await both handle asynchronous code in Node.js.

Promise has 3 states – resolved, rejected and pending.

Examples:

1. **Creating async function to add two numbers**

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

function add(a, b) {

   return new Promise(resolve => {

      setTimeout(() => {

         resolve(a + b);

      }, 3000);

   });

}

async function output(a, b) {

   const ans = await add(a, b);

   console.log(ans);

}

output(10, 20);

console.log("async programming")
```

2. **Create an asynchronous function to calculate the square of a number**

```
const async = require("async");
function square(x) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(Math.pow(x, 2));
    }, 2000);
  });
}
async function output(x) {
  const ans = await square(x);
  console.log(ans);
}
output(10);
```