5...

File System

Objectives...

- To learn the operations on file.
- To learn about I/O operations.
- To study about Handling Exceptions and Errors.

5.1 INTRODUCTION

About Node JS file System:

- In this section, we discuss handling files from the file system which is important for loading and parsing files in your application.
- The file system is big part of any application need to handle files for loading, manipulating or serving data.
- We can handle file operations such as create, read, delete and many more using Node JS. The built-in model provided by Node JS is `fs` (File System).
- Node JS provides functionality of file I/O by providing wrappers around the standard functions.
- Depending upon user requirements. File system operations can have synchronous and asynchronous approach.
- To use this file system (`fs`) module, use the require() method:

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

5.1.1 Synchronous and Asynchronous Approach

S-22

Synchronous Approach:

- We also called Synchronous approach as **"blocking functions"** as after completion of one operation it starts the next operation i.e. to execute next command previous all command must have executed.
- For Example, Imagine the situation of a restaurant. The waiter will take order from table1 give it to the kitchen and will wait until the chef prepares the food. This is called **blocking or synchronous nature.**

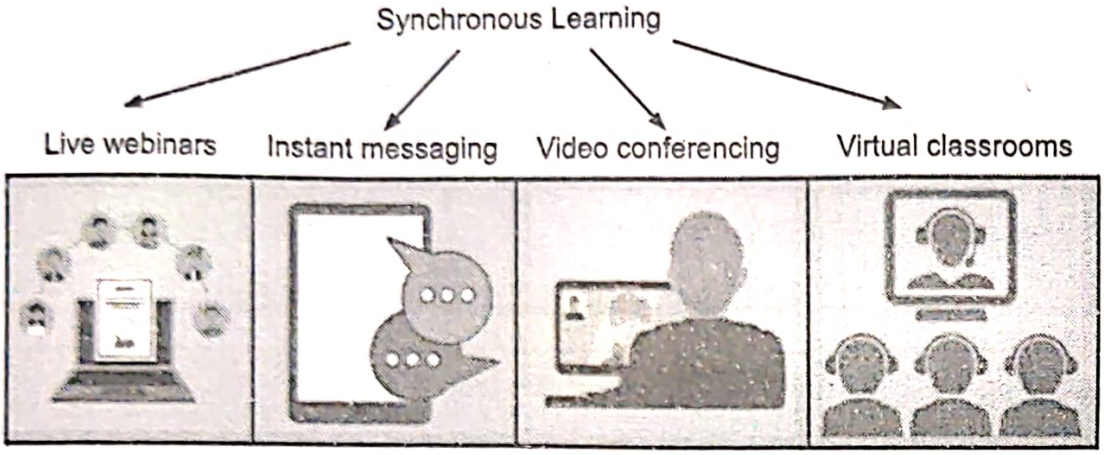

Fig. 5.1: Synchronous Learning

Asynchronous Approach:

- We also called Asynchronous approach as **"non-blocking functions"**. As it never waits other operations to complete. It may happen that several commands are executing in background and several are executing in foreground.
- For Example, in contrast the waiter takes order from table1 and gives the order in the kitchen, now the waiter can serve another table let's say table2 and get their order. This is **asynchronous system**, as the chef is preparing the food, meanwhile a single waiter is serving different tables (taking orders and serving food). The waiter doesn't have to wait for the chef to cook the meal before he has to go to another table. This is what we call a **non-blocking** or **asynchronous** architecture.

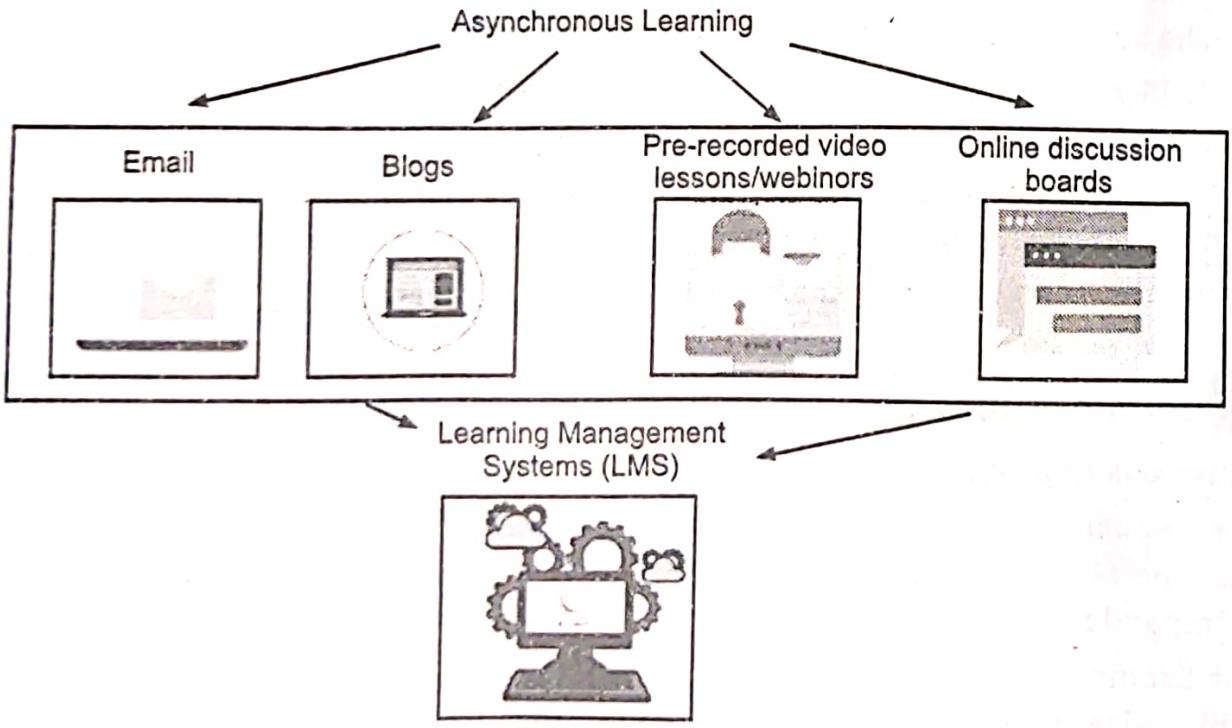

Fig. 5.2: Asynchronous learning

Difference between Synchronous and Asynchronous Approach:

Table 5.1: Difference between Synchronous and Asynchronous

Sr. No.	Synchronous	Asynchronous
1.	Students learn at the same time.	Students learn at different time.
2.	Communication happens in real time.	Communication is not live.
3.	Possibly more engaging and effective.	Possibly more convenient and flexible.
4.	Allows for instant feedback and clarification.	Allow students to work at their own speed.
5.	Examples: Video conferencing, live chat, live streamed videos.	Examples: Email, Screencasts, Flipgrid, Videos, Blog posts/comments.

Implementation of Synchronous and Asynchronous Modes in Various Applications:

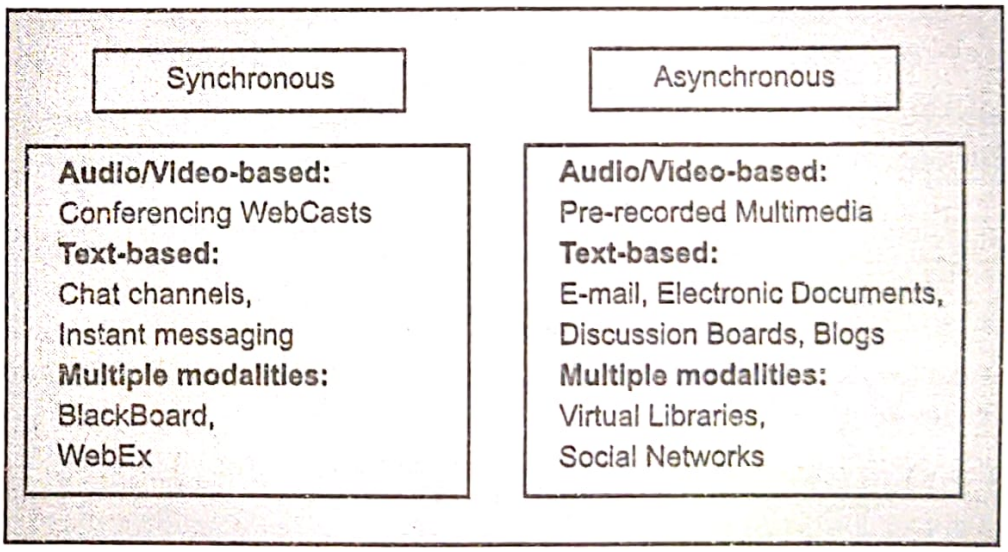

Fig. 5.3: Synchronous and Asynchronous Modes in Various Applications

5.2 OPERATIONS ON FILE [S-22, W-22, S-23]

5.2.1 Read a File (fs.readFile) [W-22, S-23]

1. Read a File Synchronously:

- The `fs.readFileSync()` method is an inbuilt application programming interface of `fs` module which is used to read the file and return its content.
- In `fs.readFileSync()` method, we can read files in a synchronous way, i.e. we are telling Node JS to block other parallel process and do the current file reading process.

- **Syntax:**

```
fs.readFileSync(path, options)
```

Where,

path: It takes the relative path of the text file.

options: It is an optional parameter which contains the encoding and flag.

Program 5.1: Node JS program to demonstrate the fs.readFileSync() method.

```
const fs = require('fs');
// Call readFileSync() method to read 'programming.txt' file
const data = fs.readFileSync('./programming.txt',{encoding:'utf8', flag:'r'
});
// Display the file data
console.log(data);
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
This is a file containing a collection of programming languages.
1. C
2. C++
3. JAVA
4. Angular JS
5. Node JS
```

2. Read a File Asynchronously:

- To read contents of a file into memory is very common programming task. There are many file system methods in fs module. Let's first start with **fs.readFile().**
- **Syntax:**

```
var fs = require('fs');
fs.readFile(file, [encoding], [callback]);
```

Where,

file: File path of the file or file name.

encoding is an optional parameter that specifies the type of encoding to read the file. Types of encodings are 'ascii', 'utf8', and 'base64'. The default encoding is null.

callback is a function to call when the file has been read and the contents are ready. It is passed two arguments, error and data.

- For example, suppose we have predefined file calc.js having some content. Now we will open this file and read file by writing code into app.js file.

Program 5.2: Program for fs.readFile().

```
fs = require('fs');
fs.readFile('programming.txt', 'utf8', function (err,data) {
   if (err) {
       return console.log(err);
   }
   console.log(data);
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
This is a file containing a collection of programming languages.
1. C
2. C++
3. JAVA
4. Angular JS
5. Node JS
```

5.2.2 Writing a File Data

- File I/O operation implemented by `'fs'` module of Node JS. There are synchronous as well as asynchronous methods in `fs` module. The asynchronous function has a callback function. The last parameter which indicates the completion of the asynchronous function.
- There are 2 ways to write in file synchronously and asynchronously.

1. Synchronously Writing Data to a File:

- If we want to write data synchronously we use `fs.writeFileSync()` method. The `fs.writeFileSync()` creates a new file if the specified file does not exist.
- **Syntax:**

```
fs.writeFileSync(file, data, option)
```

Where,

`file`: It is a string, Buffer, URL or file description integer that indicates the path of the file where it has to be written. Using a file descriptor will make the work similar to fs.write() method.

`data`: Buffer, TypedArray or DataView that will be written to the file. The data is of string type.

`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 which specifies the encoding of the file. The default value is 'utf8'.
- `mode`: It is an integer which specifies the file mode. The default value is 0o666.
- `flag`: It is a string which specifies the flag used while writing to the file. The default value is 'w'.

Program 5.3: Node JS program to demonstrate the fs.writeFileSync() method. **W-22, S-23**

```
const fs = require('fs');
let data = "This is a file containing a collection"
          + " of programming languages.\n"
          + "1. JAVA\n2. C++\n3. R";
fs.writeFileSync("programming.txt", data);
console.log("File written successfully\n");
console.log("The written has the following contents:");
console.log(fs.readFileSync("programming.txt", "utf8"));
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
File written successfully
The written has the following contents:
This is a file containing a collection of programming languages.
1. JAVA
2. C++
3. R
```

2. Asynchronously Writing Data to a File:

- If we want to write data asynchronously we use `fs.writeFile()` method.
- **Syntax:**

```
fs.writeFile( file, data, options, callback )
```

Where,

`file`: It is a string, Buffer, URL or file description integer that denotes the path of the file where it has to be written.

`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.

Program 5.4: Node JS program to demonstrate the fs.writeFile() method.

```
const fs = require('fs');
let data = "This is a file containing a collection of books.";

fs.writeFile("books.txt", data, (err) => {
  if (err)
    console.log(err);
  else {
    console.log("File has been written successfully\n");
    console.log("With following contents:");
    console.log(fs.readFileSync("books.txt", "utf8"));
  }
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
File has been written successfully
With following contents:
This is a file containing a collection of books.
```

5.2.3 Opening a File

- To open a file, create file, to write to a file or to read a file `fs.open()` method is used. `fs.open()` method performs several operations on a file. Initially we have to upload the `fs` class, which is module to access physical file system.
- **Syntax:**

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

Where,

filename: It holds the name of the file to read or the entire path if stored at other location.

flag: The operation in which file has to be opened.

`mode`: Sets the mode of file i.e. r-read, w-write, r+ -readwrite. It sets to default as readwrite.

`callback`: It is a callback function that is called after reading a file. It takes two parameters:

- `err`: If any error occurs.
- `data`: Contents of the file. It is called after open operation is executed.

- All types of flags are described below:

Table 5.2: Flags

FLAG	DESCRIPTION
r	This flag will open file to read and throws exception if file doesn't exist.
r+	This flag will open file to read and write. Throws exception if file doesn't exist.
rs+	This flag will open file in synchronous mode to read and write.
w	It is use to open file for writing. File is created if it doesn't exist.
wx	It is same as 'w' but fails if path exists.
w+	It is use to open file to read and write. File is created if it doesn't exist.
wx+	It is same as 'w+' but fails if path exists.
a	It is use to open file to append. File is created if it doesn't exist.
ax	It is same as 'a' but fails if path exists.
a+	It is use to open file for reading and appending. File is created if it doesn't exist.
ax+	It is same as 'a+' but fails if path exists.

Program 5.5: Node JS program to demonstrate the fs.open() Method in write mode.

```
var fs = require('fs');
// To open file in Write mode, create file if doesn't exists.
fs.open('demo.txt', 'w', function (err, f) {
    if (err) {
        return console.error(err);
    }
      console.log(f);
      console.log("File opened!!");
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
3
File opened!!
```

Program 5.6: Node JS program to demonstrate the fs.open() Method for read mode.

```
var fs = require('fs');
//open file in read mode, exception occurs if the file does not exist
fs.open('demo.txt', 'r', function (err, f) {
   if (err) {
      return console.error(err);
   }
   console.log(f);
   console.log("File opened!!");
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
3
File opened!!
```

5.2.4 Deleting a File

- Whenever we want to delete a file we use `fs.unlink()` method. This method does not work on directories. To delete directories we have to use `fs.rmdir()`.
- **Syntax:**

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

Where,

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.

Program 5.7: Program to remove a file from the file system.

```
const fs = require('fs');
// Get the files in current directory before deletion
getFilesInDirectory();

// Delete books.txt
fs.unlink("books.txt", (err => {
  if (err) console.log(err);
```

```
    else {
      console.log("\nDeleted file: books.txt");
      // Get the files in current directory after deletion
      getFilesInDirectory();
    }
  }));
  //Function to get current filenames in directory with specific extension
  function getFilesInDirectory() {
    console.log("\nFiles present in directory:");
    let files = fs.readdirSync(__dirname);
    files.forEach(file => {
      console.log(file);
    });
  }
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
Files present in directory:
app.js
books.txt
demo.txt
node_modules
package-lock.json
package.json
programming.txt

Deleted file: books.txt

Files present in directory:
app.js
demo.txt
node_modules
package-lock.json
package.json
programming.txt
```

5.2.5 Truncate a File

- The `fs.ftruncate()` method is used to change the size of the file i.e. either increase or decrease the file size.
- It changes the length of the file at the path by `len` bytes. If `len` is shorter than the file's current length, the file is truncated to that length. If it is greater than the file length, it is padded by appending null bytes (x00) until `len` is reached. It is similar to the truncate() method, except it accepts a file descriptor of the file to truncate.
- **Syntax:**

```
fs.ftruncate(fd, len, callback)
```

Where,

fd: This is the file descriptor returned by fs.open().

len: This is the length of the file after which the file will be truncated.

callback: This is the callback function no arguments other than a possible exception are given to the completion callback.

Program 5.8: Node JS program to demonstrate the fs.ftruncate() Method.

```
var fs = require("fs");
var buf = new Buffer.alloc(1024);

console.log("Going to open an existing file");
fs.open('programming.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to truncate the file after 10 bytes");

   // Truncate the opened file.
   fs.ftruncate(fd, 10, function(err) {
      if (err) {
         console.log(err);
      }
      console.log("File truncated successfully.");
   });
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
Going to open an existing file
File opened successfully!
Going to truncate the file after 10 bytes
File truncated successfully.
```

5.2.6 Append a File

- It is used to asynchronously append the given data to a file using **appendFile()** method. If file does not exist this create new file.
- **Syntax:**

```
fs.appendFile( path, data[, options], callback )
```

Where,

path: It is source filename or file descriptor that will be appended to.

data: It is data that has to be appended.

options: It is an string or an object that can be used to specify optional parameters. It has three optional parameters:

- **encoding:** It is a string which specifies the encoding of the file. The default value is 'utf8'.
- **mode:** It is an integer which specifies the file mode. The default value is '0o666'.
- **flag:** It is a string which specifies the flag used while appending to the file. The default value is 'a'.
- **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.

Program 5.9: Node JS program to demonstrate the fs.appendFile() method.

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

// Get the file contents before the append operation
console.log("\nFile before append:",
  fs.readFileSync("programming.txt", "utf8"));

fs.appendFile("programming.txt", "World", (err) => {
  if (err) {
    console.log(err);
```

```
    }
    else {
      // Get the file contents after the append operation
      console.log("\nFile after append:",
        fs.readFileSync("programming.txt", "utf8"));
    }
  });
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
File before append: This is a file containing a collection of programming
languages.
1. JAVA
2. C++
3. R
File after append: This is a file containing a collection of programming
languages.
1. JAVA
2. C++
3. R
World
```

5.3 OTHER I/O OPERATIONS

1. Stat():

- In Node, we can use the `stat()` method to get statistics about a file. The `stat()`, takes a path string and callback function as arguments.
- The callback function also takes two arguments. The first argument to the callback is the err object and the second is an object containing the stats found for the file.
- **Syntax:**

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

- `stat()` methods are given in the following table.

Table 5.3: stat() methods

Method	Description
`stats.isFile()`	It returns true if file type of a simple file.
`stats.isDirectory()`	It returns true if file type of a directory.

Contd...

stats.isBlockDevice()	It returns true if file type of a block device.
stats.isCharacterDevice()	It returns true if file type of a character device.
stats.isSymbolicLink()	It returns true if file type of a symbolic link.
stats.isFIFO()	It returns true if file type of a FIFO.
stats.isSocket()	It returns true if file type of a socket.

Program 5.10: Program for get statistics about file.

```
var fs = require("fs");
console.log("Get file info!");
fs.stat('programming.txt', function (err, stats) {
   if (err) {
      return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");

   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
Get file info!
Stats {
  dev: 3865741643,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 844424930132133,
  size: 90,
  blocks: 0,
```

```
  atimeMs: 1608309396710.3035,
  mtimeMs: 1608309396643.2324,
  ctimeMs: 1608309396643.2324,
  birthtimeMs: 1608283886442.5571,
  atime: 2020-12-18T16:36:36.710Z,
  mtime: 2020-12-18T16:36:36.643Z,
  ctime: 2020-12-18T16:36:36.643Z,
  birthtime: 2020-12-18T09:31:26.443Z
}
Got file info successfully!
isFile ? true
isDirectory ? false
```

- Depending on the type of file or operating system you are using, the stats returned may differ than above. Below is a table that summarizes the stats that returned by the stat method.

Table 5.4: stats that returned by the stat method

Property	Description
dev	ID of the device containing the file.
mode	The file's protection.
nlink	The number of hard links to the file.
uid	User ID of the file's owner.
gid	Group ID of the file's owner.
rdev	The device ID, if the file is a special file.
blksize	The block size for file system I/O.
ino	The file's inode number. An inode is a file system data structure that stores information about a file.
size	The file's total size in bytes.
blocks	The number of blocks allocated for the file.
atime	Date object representing the file's last access time.
mtime	Date object representing the file's last modification time.
ctime	Date object representing the last time the file's inode was changed.

- There are more methods to check whether a path is pointing to a file or is a directory The stats object returned from the callback function contains methods for finding this out:

```
var fs = require("fs");
var path = "C:\shiv\nodeexmaple\abcde.exe";
fs.stat(path, function(error, stats) {
    console.log(stats.isFile());
    console.log(stats.isDirectory());
    console.log(stats.isBlockDevice());
    console.log(stats.isCharacterDevice());
    console.log(stats.isFIFO());
    console.log(stats.isSocket());
});
```

2. Using Read():

- Use the `fs.read()` method to read file data. You must first use the open method before attempting to read from the file. The `read` method takes numerous arguments, but the most important ones to know are that the `read` method uses the file descriptor (obtained from the `open` method), along with the file size (obtained from stats).
- **Syntax:**

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

Where,

fd: This is the file descriptor returned by fs.open().

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 position is null, data will be read from the current file position.

callback – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Program 5.11: Node JS program to demonstrate the fs.read() method.

```
const { Buffer } = require("buffer");
var fs = require("fs");
var buf = new Buffer.alloc(1024);

console.log("Open an existing file");
```

```
fs.open('programming.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Read 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());
      }
   });
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
Open an existing file
File opened successfully!
Read the file
90 bytes read
This is a file containing a collection of programming languages.
1. JAVA
2. C++
3. R
World
```

3. Using Close:

- To close a file, simply use the `close()` method and pass in the file descriptor (obtained from the open method).
- **Syntax:**

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

Program 5.12: Node JS program to demonstrate the fs.close() method.

```
var fs = require("fs");
fs.open('programming.txt', "w+", function(error, fd) {
   if (error) {
      console.error("open error: " + error.message);
   }
   console.log("File opened successfully!");
   fs.close(fd, function (error) {
      if (error) {
         console.error("close error: " + error.message);
      }
      else {
         console.log("File was closed!");
      }
   });
});
```

Output:

```
PS D:\Prajakta\BBA CA\Node JS\programs> node app.js
File opened successfully!
File was closed!
```

5.4 HANDLING EXCEPTIONS AND ERRORS

- Our requests can sometimes fail, for a variety of reasons - an error occurring in the fetch() function, internet issues, internal server errors, and others. We need a way to handle these situations, or in the very least be able to see that they occurred.
- We can handle runtime exceptions by adding catch() at the end of the promise-chain. Let's add a simple catch() function to our program above:

```
const fetch = require('node-fetch');
let todo = {
    userId: 123,
    title: "loren impsum doloris",
    completed: false
}
fetch('https://jsonplaceholder.typicode.com/todos', {
    method: 'POST',
```

```
        body: JSON.stringify(todo),
        headers: { 'Content-Type': 'application/json' }
    }).then(res => res.json())
      .then(json => console.log(json))
      .catch(err => console.log(err))
```

- Ideally, you shouldn't simply ignore and print errors, but instead have a system in place for handling them.
- We should keep in mind that if our response has a 3xx/4xx/5xx status code, the request either failed or additional steps need to be taken by the client.
- Namely, 3xx HTTP status codes indicate that additional steps need to be taken by the client, 4xx codes indicate an invalid request, and 5xx codes indicate server errors. All these status codes tell us that our request wasn't successful in practical terms.
- catch() won't register any of these cases because communication with the server went well, i.e. we made a request and got a response successfully. This means that we need to take additional steps to make sure we cover the situation when the client-server communication was successful, but we didn't receive any of the successful (2xx) HTTP status codes.
- A common way to make sure that failed requests throw an error is to create a function that checks the HTTP status of the response from the server. In that function, if the status code doesn't indicate success, we can throw an error and catch() will catch it.
- We can use the previously mentioned ok field of Response objects, which equals true if the status code is 2xx.
- Let's see how this works:

```
const fetch = require('node-fetch');

function checkResponseStatus(res) {
    if(res.ok){
        return res
    }
    else {
       throw new Error('The HTTP status of the reponse: ${res.status}
       (${res.statusText})');
    }
}
```

```
fetch('https://jsonplaceholder.typicode.com/MissingResource')
    .then(checkResponseStatus);
    .then(res => res.json());
    .then(json => console.log(json));
    .catch(err => console.log(err));
```

- We used the function at the beginning of the promise-chain (before parsing the response body) in order to see whether we encountered an issue. You can also throw a custom error instead.
- Again, you should have a strategy in place for handling errors like this instead of just printing to the console.
- If everything went as expected, and the status code indicated success, the program will proceed as before.

Summary

- The file system is big part of any application needing to handle files path for loading, manipulating or serving data.
- We can handle file operations such as create, read, delete and many more using Node JS. The built-in model provided by Node JS is `fs` (File System).
- The `fs.readFileSync()` method is an inbuilt application programming interface of `fs` module which is used to read the file and return its content.
- To read contents of a file into memory is very common programming task. There are many file system methods in `fs` module. Lets first start with `fs.readFile()`.
- If we want to write data synchronously we use `fs.writeFileSync()` method.
- If we want to write data asynchronously we use `fs.writeFile()` method.
- To open a file, create file, to write to a file or to read a file `fs.open()` method is used. `fs.open()` method performs several operations on a file.
- We want to delete a file we use `fs.unlink()` method. This method does not work on directories. To delete directories we have to use `fs.rmdir()`.
- The `fs.ftruncate()` method is used to change the size of the file i.e. either increase or decrease the file size.
- It is used to asynchronously append the given data to a file using `appendFile()` method. If file does not exist this create new file.
- In Node, we can use the `stat()` method to get statistics about a file.
- Use the `fs.read()` method to read file data.

Check Your Understanding

1. Which method of fs module is used to delete a file?
   (a) fs.delete(fd, len, callback) (b) fs.remove(fd, len, callback)
   (c) fs.unlink(path, callback) (d) None of the above.
2. Which of the following is the correct way to get an extension of a file?
   (a) fs.extname('main.js') (b) path.extname('main.js')
   (c) os.extname('main.js') (d) None of the above.
3. A stream fires data event when there is data available to read.
   (a) false (b) true
4. What does the fs module stands for?
   (a) File Service (b) File System
   (c) File save (d) File Store
5. Which statement is valid in using a Node module fs in a Node based application?
   (a) Import fs (b) Package fs
   (c) var fs = require("fs") (d) var fs = import("fs")
6. Which of the following is true about readable stream?
   (a) Readable stream is used for read operation
   (b) Output of readable stream can be input to a writable stream
   (c) Both of the above
   (d) None of the above

ANSWER KEY

1. (c)	2. (b)	3. (b)	4. (b)	5. (d)
6. (c)				

Practice Questions

Q.I: Answer the following questions in short.

1. What is Synchronous and Asynchronous approach?
2. Write use of close()?
3. Write syntax and use of read().
4. Write use of stats function().
5. Why we use require method?

Q.II: Answer the following questions.

1. Explain with help of suitable example readFile() method.

2. Explain 2 ways to write in a file.
3. How to write synchronous data to a file explain with suitable example.
4. How to write asynchronous data to a file explain with suitable example.
5. Write a note on opening a file in Node JS.
6. How to delete file in Node.JS? Explain with suitable example.
7. Explain difference between blocking and non-blocking calls in Node JS.
8. Write a note on reading from file in Node JS.
9. Write short note on exception handling.