**Module III**

**AIM:**

To demonstrate the Fs module file paths

**Objective**

Explantion of Fs module file paths

**Theory**

n Node.js, file handling is handled by fs module. You can read more about it here. We can check the path for file or directory in Node.js in both Synchronous and Asynchronous way.

Note: Asynchronous version is usually preferable if you care about application performance.

Synchronous method: Synchronous operations are great for performing one-time file/directory operations before returning a module. To check the path in synchronous mode in fs module, we can use statSync() method. The fs.statSync(path) method returns the instance of fs.Stats which is assigned to variable stats. A fs.Stats object provides information about a file. The stats.isFile() method returns true if the file path is File, otherwise returns false. The stats.isDirectory() method returns true if file path is Directory, otherwise returns false.

**Explanation**

Example

**// Require the given module**

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

**// Use statSync() method to store the returned**

**// instance into variable named stats**

var stats = fs.statSync("/Users/divyarani/Documents/geekforgeeks/geeks.js");

**// Use isFile() method to log the result to screen**

console.log('is file ? ' + stats.isFile());

var stats = fs.statSync("/Users/divyarani/Documents/geekforgeeks/geek");

**// Use isDirectory() method to log the result to screen**

console.log('is directory ? ' + stats.isDirectory());

Output

is file ? true

is directory ? true

**AIM:**

To demonstrate the how to read, write, & close file

**Objective**

Explantion of the how to read, write, & close file in node.js

**Theory**

**Reading From Files**

Being able to read from files on your local file system can be hugely useful and there are a number of different things you can build on top of this. A log reader, importing information from spreadsheets and xml files or whatever you can think of, being able to read from files is hugely useful

the file path is File, otherwise returns false. The stats.isDirectory() method returns true if file path is Directory, otherwise returns false.

**Explanation**

**app.js**

var fs = require("fs");

fs.readFile("temp.txt", function(err, buf) {

console.log(buf.toString());

});

**Create a temp.txt within the same directory and write in it anything you’d like. Run your script using node app.js and you should see in the console the contents of your file.**

**Understanding the Code**

We’ll step through this with comments.

var fs = require("fs");

**This line does the job of importing the fs package and allowing us to utilize it within our own code.**

fs.readFile("temp.txt", function(err, buf) {

console.log(buf);

});

**Writing To Files**

Now that you’ve got the reading of files down, it’s time to start modifying these files. To do this we’ll be using the same FS package we used in part one.

The Code:

Again create a new file within your current directory and call it write.js and then add the following javascript code:

write.js

var fs = require("fs");

var data = "New File Contents";

fs.writeFile("temp.txt", data, (err) => {

if (err) console.log(err);

console.log("Successfully Written to File.");

});

Run this code by executing node write.js in the terminal and then open up temp.txt in your editor, you should now see the new contents of the file.

Creating New Files - The above code will successfully create new files for you should the path to the file not already exist. This is handy as it means you can succinctly create, and write to a new file in one promise.

**AIM:**

Demonstrate node JS $http service

**Objective**

Create an application that uses the Angular JS $http service to make a request to a server & display a response

**Theory**

AngularJS provides **$http** control which works as a service to read data from the server. $http is an [AngularJS](http://www.c-sharpcorner.com/technologies/angularjs) service for reading data from remote servers. The**$http** is a core AngularJS service that is used to communicate with the remote HTTP service via browser’s XMLHttpRequest object or via JSONP.

The **$http**service is function that takes a configured object to generate a HTTP request and return the response. This response contains data, status code, header, configuration object and status text. In $http the first function executes on successful callback and the second function xeecutes on error.

**Setting HTTP Header**  
The **$http** service automatically set svalues for the header property for all requests. We can manually define the values for the property of headers. In the below example we defined the same property of header.

The **$http**service is used for reading data from remote servers. Data from remote servers can be retrieved in simple text or JSON format.

**types of HTTP request get(), post(), put(), delete().**

**Explanation**

The $http service is a core AngularJS service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.

The $http API is based on the deferred/promise APIs exposed by the $q service. While for simple usage patterns this doesn't matter much, for advanced usage it is important to familiarize yourself with these APIs and the guarantees they provide.

The $http service is a function which takes a single argument — a configuration object — that is used to generate an HTTP request and returns a promise that is resolved (request success) or rejected (request failure) with a response object.



Text

Description automatically generated

**AIM:**

Demonstrate node JS piping

**Objective**

Explanation of node js stream.pipe() methode

**Theory**

**What Are Streams?**

A stream is an abstract interface that lets you perform specific tasks continuously. A stream is an EventEmitter that implements different methods. A user can use streams to perform a variety of tasks, like read, write, and transform functions.

In a Node.js environment, streams are used to work with streaming data. It provides users with an API that helps them in creating the streaming interface. The data here is received in parts and is read in parts as well.

**Piping Streams**

As the name suggests, piping is a process that ensures data flows without any hindrance. Piping feeds the output of a stream as an input to another stream and helps maintain a smooth workflow. How long will the piping continue depends on what a user feeds into the system.

**Explanation**

var fs = require("fs");

var data = '';

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error

readerStream.on('data', function(chunk) {

data += chunk;

});

readerStream.on('end',function() {

console.log(data);

});

readerStream.on('error', function(err) {

console.log(err.stack);

});

console.log("Program Ended");

**Now run the main.js to see the result −**

$ node main.js

**Verify the Output.**

Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

You can use additional logic to allow switching between modes automatically. Using piping, users can create bidirectional streams that can perform various functions. The piping mechanism stops as soon as the end functions run.

**AIM:**

Demonstrate how to shut down server using node js

**Objective**

Write code to shut down node js server

**Theory**

server.stop() stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a 'close' event.

i.e. when you call server.close(), server stops accepting new connections, but it keeps the existing connections open indefinitely. This can result in your server hanging indefinitely due to persistent HTTP connections or because of the ongoing requests that do not produce a response. Therefore, in order to close the server, you must track creation of all connections and terminate them yourself.Explanation

**Explanation**

The basic principle of forcefully closing a HTTP server looks like this:

import {

createServer,

} from 'http';

const server = createServer(() => {});

server.listen();

const sockets = new Set();

server.on('connection', (socket) => {

sockets.add(socket);

server.once('close', () => {

sockets.delete(socket);

});

});

/\*\*

\* Forcefully terminates HTTP server.

\*/

const close = (callback) => {

for (const socket of sockets) {

socket.destroy();

sockets.delete(socket);

}

server.close(callback);

};