Code Node Modules

We have seen third party packages written by someone else which can be installed and used in our code.

Reading Files

Let's create a samplefile.txt, which contains text which we will access in our node project. To open this file and to read content in it, we need to import filesystem which is pre-build or preinstalled in node. We would use **require('fs')** fs – file system module. We have two ways to do this, we have readFile and readFileSync.

It is a synchronous version of read file. Some times when we need a https request coming in with file operation, so you cannot wait for it to finish and then move to the next request. Rather, we would create this asynchronous readFile so that as long as the server is checking for the file, it can go for the next request.

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

fileSystem.readFileSync
```

In the current directory, we can find the file samplefile.txt

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

const textFromFile = fileSystem.readFileSync('./samplefile.txt')

console.log(textFromFile)
```

By default, we are getting something called as buffer. This is understood by computer. Buffer is a collection of binary bites. There is an encoding that we can provide so that the output would be in a textual format. So, if we add utf-8, then the output would be textual format.

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

const textFromFile = fileSystem.readFileSync('./samplefile.txt', 'utf-8')


console.log(textFromFile)
```

Now, if we add,

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

console.log('Reading')
const textFromFile = fileSystem.readFileSync('./sample.txt', 'utf-8');
console.log('Finished')
console.log(textFromFile)
```

Only after the file is properly read, ‘Finished’ will be logged in.

For an asynchronous one,

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

fileSystem.readFile('./samplefile.txt', 'utf-8', function(err, data) {
    if (err) {
        console.log("Error Occurred")
        console.log(err)
    }
    console.log(data)
})

console.log("Finished")
console.log("Finished")
```

The callback function receives two parameters; first parameter is **error** and the second parameter would be actual **data**. The error can be checked using the if condition. In case of error, we need to handle it. If there is not error, then we can read the data.

Output:

Finished
Finished
File_Data_Will_Be_Displayed_Next

For **synchronous** version, the **output** would be,

File_Data_Will_Be_Displayed_Next
Finished
Finished

If we want to read a file from another directory, then we can give the file path in readFile(' ') and it would show the content of the file.

We will now delete the samplefile.txt file. If we run the code now, it will show an error. So instead of writing the whole code we can just mention **`if (err) throw err`**

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

fileSystem.readFile('./samplefile.txt', 'utf-8', function(err, data) {
    if (err) throw err

    console.log(data)
})

console.log("Finished")
```

The moment we upload this code to a server, then the system will become that server's system.

Writing Files:

```
const textToWrite = "This is going to be written in the file"

console.log("Writing")
fileSystem.writeFileSync('./myFile.txt', textToWrite)
console.log("Written", textToWrite)
```

When we run this, then the **output** would be

Writing
Written

And we can see the myFile.txt being created in the folder.

For async,

```
const textToWrite = "This is going to be written in the file"

console.log("Writing")
fileSystem.writeFile('./myFile.txt', textToWrite, function(err) {
    if (err) throw err
    console.log("Finished")
```

```
})
console.log("Written")
```

Output:

Writing
Written
Finished

And then the file was created.

Delete a File:

To delete myFile.txt, we would say

```
fileSystem.unlinkSync('./myFile.txt')
```

This code will delete the file. This is an synchronous version.

```
try {
    fileSystem.unlinkSync('./myFile.txt')
    console.log('Deleted')
} catch (error) {
    console.log(error)
}
```

For error handling we can use try …catch.

For asynchronous,

```
fileSystem.unlink('./myFile.txt', function(err) {
    if (err) throw err
    console.log("Deleted Second File")
})
```

The benefit of try catch block is that even though there is an error we can still run the remaining program.

Now, if we need to read a .json file, then we can use the filesystem module to read the entire content, but a very easy path would be,

```
const fileSystem = require('fs')
const package = require('./package.json')
console.log(package.license)
```

Internally, it is doing the same thing, but this is a short hand way.

Until now, we have never seen anything which is similar to the input function of Python. But since now we are using NodeJS, this is one feature that we can build upon.

```
const readline = require('readline')

// to create an interface.
const rl = readline.createInterface(process.stdin, process.stdout)
```

Process.stdin – standard input – keyboard

Process.stdout – standard output – terminal

We have created an interface, and we need to use the interface to take input and see the output.

We have an option as rl.quesiton(), which is an asynchronous function.

```
const readline = require('readline')

// to create an interface.
const rl = readline.createInterface(process.stdin, process.stdout)

rl.question("What is your name?", function(name) {
    console.log(`Hi, nice to meet you ${name}`)
})
```

Output:

What is your name?

The cursor is still active for us to give in an input. So, once we enter the name ‘Yash’, in the next line, we can see

Hi nice to meet you Yash.

To close the function, we can use rl.close(). We are closing the function after we have run the program.

```
const readline = require('readline')

// to create an interface.
const rl = readline.createInterface(process.stdin, process.stdout)

rl.question("What is your name?", function(name) {
    console.log(`Hi, nice to meet you ${name}`)
    rl.close()
```

```
})
```

If we need to ask another question then

```
const readline = require('readline')

// to create an interface.
const rl = readline.createInterface(process.stdin, process.stdout)

rl.question("What is your name?", function(name) {
    console.log(`Hi, nice to meet you ${name}`)
    rl.question("What is your age?", function(age) {
        console.log(`You are ${age} old`)
        rl.close()
    })
})
```

We cannot put the second question outside, then both questions would be triggered as the same time.

It would get messy if we need to ask multiple questions. So we can use another npm package – readlineSync. We need to install the package,

`npm install readline-sync`

Once it is installed, we need to import the code in the package to our source code.

```
const rs = require('readline-sync');

const name = rs.question('What is your name?')
const age = rs.questionInt('What is your age?')

console.log(`Hi, ${name}. I see you are ${age} years old`)
```

This is very simple comparatively. We are able to minimize the call-back function. Most of the time, there will be a module or package already present in the npm packages. All we need to do is import it and use it in our program.

Sending Files to the Server

`npm install express`

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {

    res.send('Works')
})


app.listen(3000, () => console.log('Sever Started'))
```

We will use res.sendFile(./index.html) → we cannot directly access index.html file. So, we need to call res.sendFile(./index.html). we need to restart the server and then the output would be seen in the html.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.sendFile('index.html')
})


app.listen(3000, () => console.log('Sever Started'))
```

Since we are stuck with restarting our server, we can use another npm package,

`npm istall -g nodemon`

This is only for development purpose and so it should not come into the dependencies list. All we need to do is,in the terminal,

`Nodemon index.js`

If we make changes, to it,

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    // res.sendFile('index.html')
    res.send('hello')
})
```

```
app.listen(3000, () => console.log('Sever Started'))
```

Once we save it, the output changes.

```
const express = require('express');
const path = require('path');
const app = express();

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname+'index.html'))
})


app.listen(3000, () => console.log('Sever Started'))
```

We are able to send html back through my server.

There is a text.txt, which has content as, “this is inside a text file”.

```
const express = require('express');
const path = require('path');
const app = express();

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname+'index.html'))
})

app.get('/txt', (req, res) => {
    res.sendFile(path.join(__dirname+'/test.txt'))
})


app.listen(3000, () => console.log('Sever Started'))
```

Now, if we say localhost:3000/txt, in the browser we can see, this is inside a text file. It will automatically provide margin as well to the content.