RE-Visit:

File Operations:

Lot of things in NodeJS are asynchronous. Everything works on the principle of callback. So, **`fs.read(fileName, function)`**. So, scald it reads the content internally the callback function so everything based on callback function. Everything is based on callback. The better way to use is promises which means to use async and await. This helps to structure my code, so that I don't have to deal with callback.

In express also you need callback, but a bit in-core modules in NodeJS supports promises and we will see them now.

```
const fs = require('fs').promises

fr.readFile = ('./sample.txt', 'utf-8').then(function(data){
    console.log(data)
})
```

For each character there is a binary code. Similarly, this will be the same in the case of writeFile as well.

```
fs.writeFile('test.txt', "This is a test write").then(() => console.log
('Done'))
```

This was promise based. Even though we wanted to get rid of callback function, in the .then we are still passing the callback function. The only difference is we are passing it in .then() or .catch().

First thing, we need to use await; which is only allowed inside async function. We cannot directly do that. In the latest ECMA script, it is allowed, but to keep it confusion free, we will declare an async-function.

```
async function read() {
    const data = await fs.readFile('sample.txt', 'utf-8')
    console.log(data)
}

read()
```

This is comparatively much simpler to understand than the previous example. If we remove the 'uft-8' we will see a buffer, it gives a binary data.

Since we know promises we can change our code accordingly.

```
async function write() {
    const data = await fs.readFile('text2.txt', "This is coming from as
ync await")
    console.log(data)
}

write()
```

The output would be undefined because, we are writing content in our file and ultimately, we are not returning anything. If we having an error, then we use the try catch block. Using the try-catch block,

```
async function write() {
    try {
        const data = await fs.appendFile('test2.txt', "This is coming f
rom async await")
        console.log(data)
    } catch (error) {

    }
}
```

If we want to use template to preserve the format,

```
async function write() {
    try {
        const text = `
            Asdasd
        `
        const data = await fs.appendFile('test2.txt', text)
        console.log(data)
    } catch (error) {

    }
}
```

We have translated the callback, await and promises knowledge into NodeJS.

Events:

We know there are DOM specific events or we have window-based events. Events are essentially, something happening at any particular point of time. Since browser does not support NodeJS, then again everything in NodeJS is an event. The only thing is it provides its own sets of events.

It provides a way of emitting events and whatever functions are attached to it will be getting executed. A simple example of legit events inside NodeJS, we need to discuss something as process.

Everything when it is executed in the memory becomes a process. All the individual events are process. A program which is running in the current time is called process. Some of the process as based on the operating system. All the processes are assigned a particular ID. So even when we are running the NodeJS, we can see that NodeJS is running.

Remembering that, if we need to do anything regarding it, there is key word called process. This process is available to the NodeJS, it is referring to the process itself. We are creating a NodeJS process to run the script file. So, when we use **`node index.js`**, the process was running the index.js file.

```
console.log("Some Operations")

console.log(`25 + 22 = ${25 + 22}`)

process.on("exit", function() {
    console.log('exiting')
})
```

The function is the event listener which gets fired on the process object. If we run this particular file, we see the output as

Some Operations
25 + 22 = 41
exiting

```
console.log("Some Operations")

console.log(`25 + 22 = ${25 + 22}`)

process.on("exit", function() {
    console.log('exiting')
```

```
})

process.on("exit", function() {
    console.log('exiting 2')
})
```

Even though the event is fired once, there are multiple event handlers are attached and that is getting executed accordingly. We will be working with express server.

HTTP Methods

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

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

})

app.listen(3000, () +> console.log('Server Started'))
```

We have seen the get method till now. But there are quite a lot of methods that are in the HTTP Methods. There is some caching which can be done on some methods.

Get helps in getting a particular resource from the server to the browser or to the client.

Post is used to send a set of data. Server receives the data and creates a new resource based on the request. This creates data in the server.

Put request is similar to post, however, we use put to update something in the server that is already created. This updated the data that has been created using the post method.

Delete, as the name suggests deletes a particular request. We can either send data without binding it to the request or there are other ways to specify what data we are supposed to delete.

These are the basic requests that are most heavily used. These operations are called **CRUD** (***create-read-update-delete***). This is a simple idea.

In a URL:

http://abc.com/categories?name=test

After the domain name, or after the '/' are called path. And anything after the question mark(?) is query parameters or key-value pairs.

Now the question arises, how do we use this information in our express parameters.

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

app.get('/category', (req, res) => {
    res.send("Works")

})

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

Inside the category, we can define,

localhost:3000/category → will give an output as **works**.

So how to access query parameters?

localhost:3000/category?name=yash&age=23

This will give us output as work. But to get those query parameters, we would use request object. We can access it using

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

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

    console.log(req.query)
    res.send("Works")

})

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

In the console we will get the query parameters as

{ name: 'yash', age: 23 }

So, based on this query parameter, we can write logic as well. Let's say,

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

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

    const {name} = req.query

    if (condition) {
        res.send(`Your name that server received = ${}`)
        return
    }
    res.send("Works")

})

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

The name in the query parameter is what we are concerned. There are other parameters attached, but we are only extracting the name parameter. Using the get request only we are able to send data to the server.

If we remove the return keyword, even though we get our response back, we see error in terminal. The reason why the return key word is important is, when we do anything on the response object, after doing it, it closes the http request connection. So if we do res.send after the http request connection closed, which is stateless.

Returning a value is something different. For now, res.send can be called only once. Once it is called, the function should not call res.send again. This is how we access the query parameters.

If we have a category books; based on the category, we can display list of books. Based on the category, we will fetch data from the database and send it to the front end. There will not be any issue, but the question is how dry will you keep your code.

One way of doing it, is multiple paths. We can define

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

    res.send("Sending Mobiles")

})

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

    res.send("Sending Books")

})

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

If there are 100 categories, we cannot create 100 categories. The express will understand it to be path parameter and will supply it the request query. In the object req.params, there will be the path variable, ‘categoryName’ being declared. We can add one more for req.params object as well.

```
app.get('/category/:categoryName/', (req, res) => {

    console.log(req.params)
    console.log(req.params.categoryName)

})
```

If we enter category/books then it would create a category of books. If we create another subCategory, such as category/books/fiction.

```
app.get('/category/:categoryName/:subCategory', (req, res) => {

    console.log(req.params)
    console.log(req.params.categoryName)

    res.send("Called from sub category")
})
```

The output would through an error as one path is lost. SO inorder to set it right we need to have 3 paths,

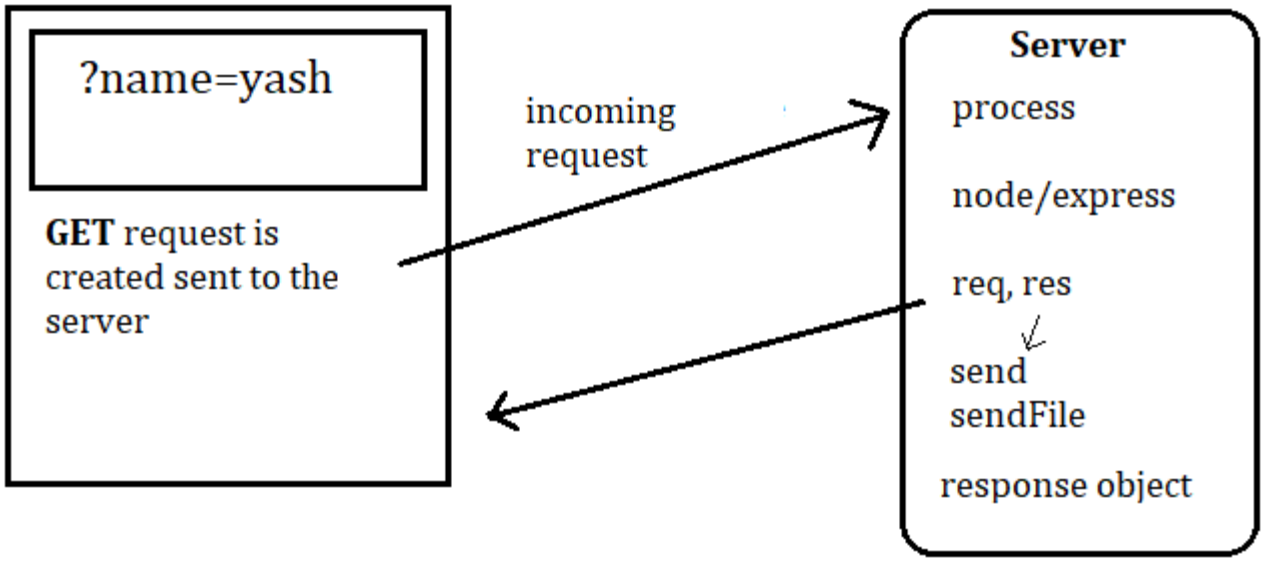
```
const express = require('express')
const app = express()

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

```
    const {name} = req.query

    if (name) {
        res.send (`Your name that server received = ${name}`)
        return
    }

    res.send("Works")
})

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

    console.log(req.params)
    console.log(req.params.categoryName)

    res.send("Called from Main Category")

})

app.get('/category/:categoryName/:subCategory', (req, res) => {

    console.log(req.params)
    console.log(req.params.categoryName)

    res.send("Called from Sub Category")
})

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

In the browser there is an address bar and accordingly how we write the address, certain parts can be accessed. In the form of query parameters, we are able to access or share some date with the server. When we share the query parameters, it translates into specific route that is defined properly. if we change anything in terms of path it will not change anything.

So, when we write, “?name=yash” and hit enter, browser understand it is a get request because anything you write in the address bar is by default a GET request. Even though we are not creating a post request, we are able to send some data. In HTTP, get request is created behind the scenes. This is like the letter in the postal system; get request is created, query parameter is added and send to the server. Based on the URL that is provided, HTTP request is sent to the server process. There

are different layers. The layer in picture now is NodeJS with express. This is the final layer through which entire request gets passed through.



We have access to something called request object and response object, which are created by express internally and sent back to you (developer). This query parameter and all the path name that we have discussed is passed in the browser. The request object has all the information related to the server. If we want to send some data back to the client, then we would have to use the response object. On this we have multiple function and properties, but we are concerned about send and sendFile. So, when we call this function, then it tells the server that the developer wants this data to be send back, and so all the data get specified and is sent back to the client browser. In the browser we are able to read or do DOM manipulation.

How to send huge amount of Data?

We need to create a post request.

```
app.post('/userInfo', (req, res) => {
    res.send("post route works")
})
```

The output would be cannot get, because we have not a route with get request. We are expecting the URL to be a post request and not the get request. But our info in the address bar would always be a get request. We need to define the route for the get method. So,

```
app.get('/userInfo', (req, res) => {
    res.send("get user info")
```

```
})

app.post('/userInfo', (req, res) => {
    res.send("post route works")
})
```

We can create multiple type of request for the same path. HTTP method is different but the URL is same.

We defined the post route and is internally working, but we cannot call the post request. We can write a script in the console; inside we can use some javascript in which we can create a post request. But while developing a backend environment, this will be a tedious process.

There are special software such as https client, rest client which allows us to test everything without actually writing the browser based javascript code. One of the popular is **postman**.

Postman has become a bit complicated so we can install insomnia. This is also a rest client in which we can provide our URL and then test them.

In postman, we have to go inside my workspace area, add a new space. In the bar we can write all the url. We need to go to the Body and select raw and the input type as JSON. In this, we can write,

```
{
    "name": "Yash",
    "Age": 36
}
```

This is the data that we are going to send the server. All the though the data is sent to the server, we cannot access it. So, at the top, we will write

```
app.use(express.json())
```

This is to tell the express that we are going to send data in json format. We can add 100s of property, but it would sent to the server.

```
app.post('/userInfo', (req, res) => {
    console.log(req.body)
    res.send(`Server changed your name to ${res.body.name} Sinha`)

})
```

In the postman when we click send, we can see,

Server changed your name to Yash Sinha. → from backend.

To send a json back,

```
app.post('/userInfo', (req, res) => {
    console.log(req.body)

    const newData = {...req.body}
    newData.appendedValue = "Express"
    console.log(newData)
    res.json(newData)
})
```

Now, when we click on send the json data is being sent back to use using json format.