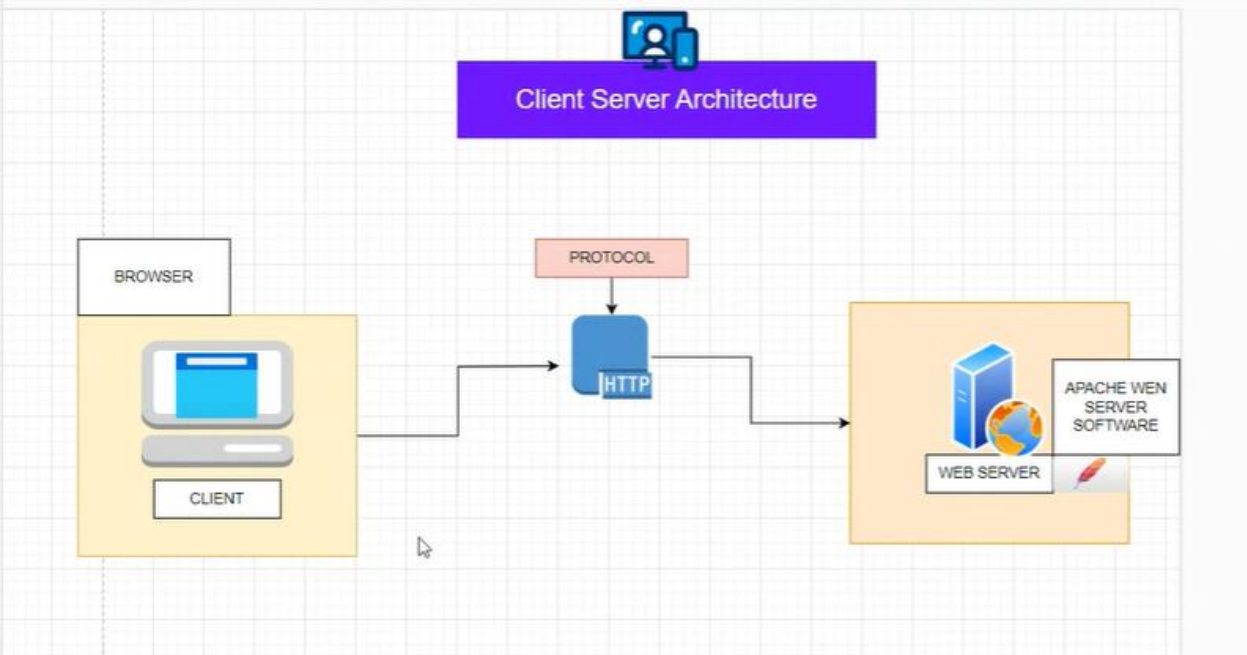
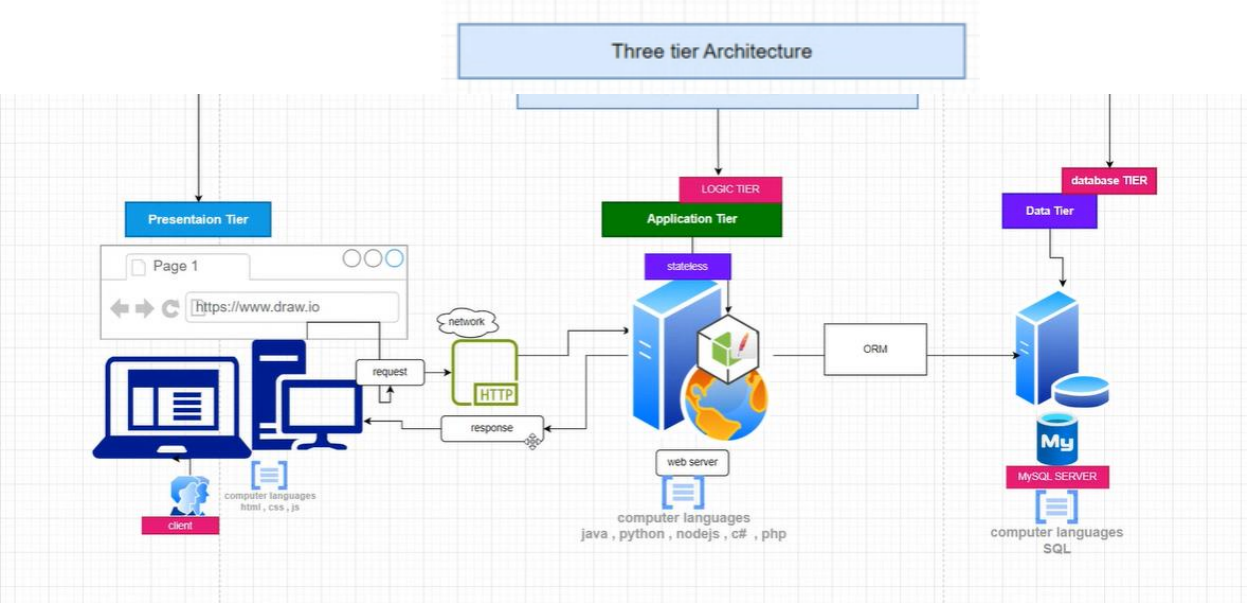
Client Server Architecture



Three Tier Architecture



What is node js ?

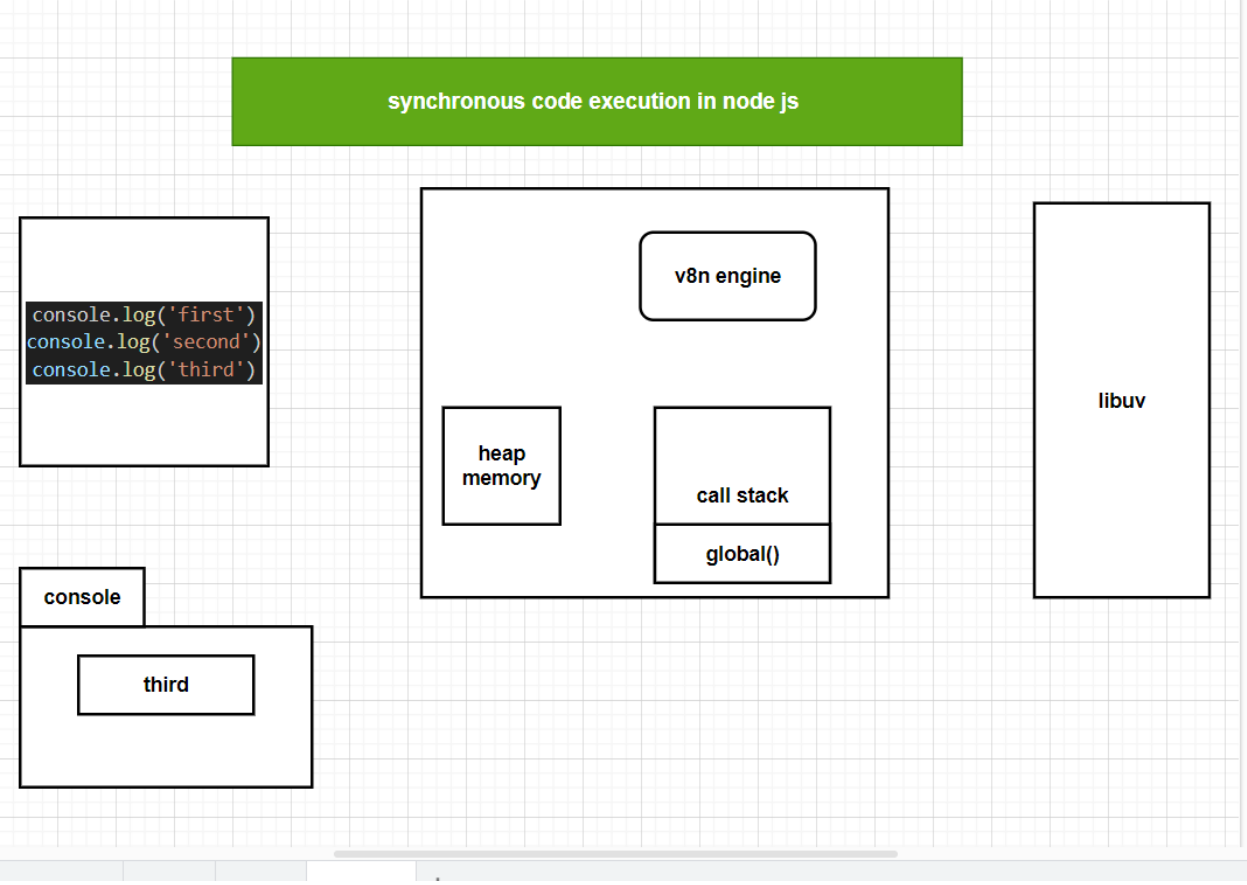
Node.js is an open-source and cross-platform JavaScript runtime environment.

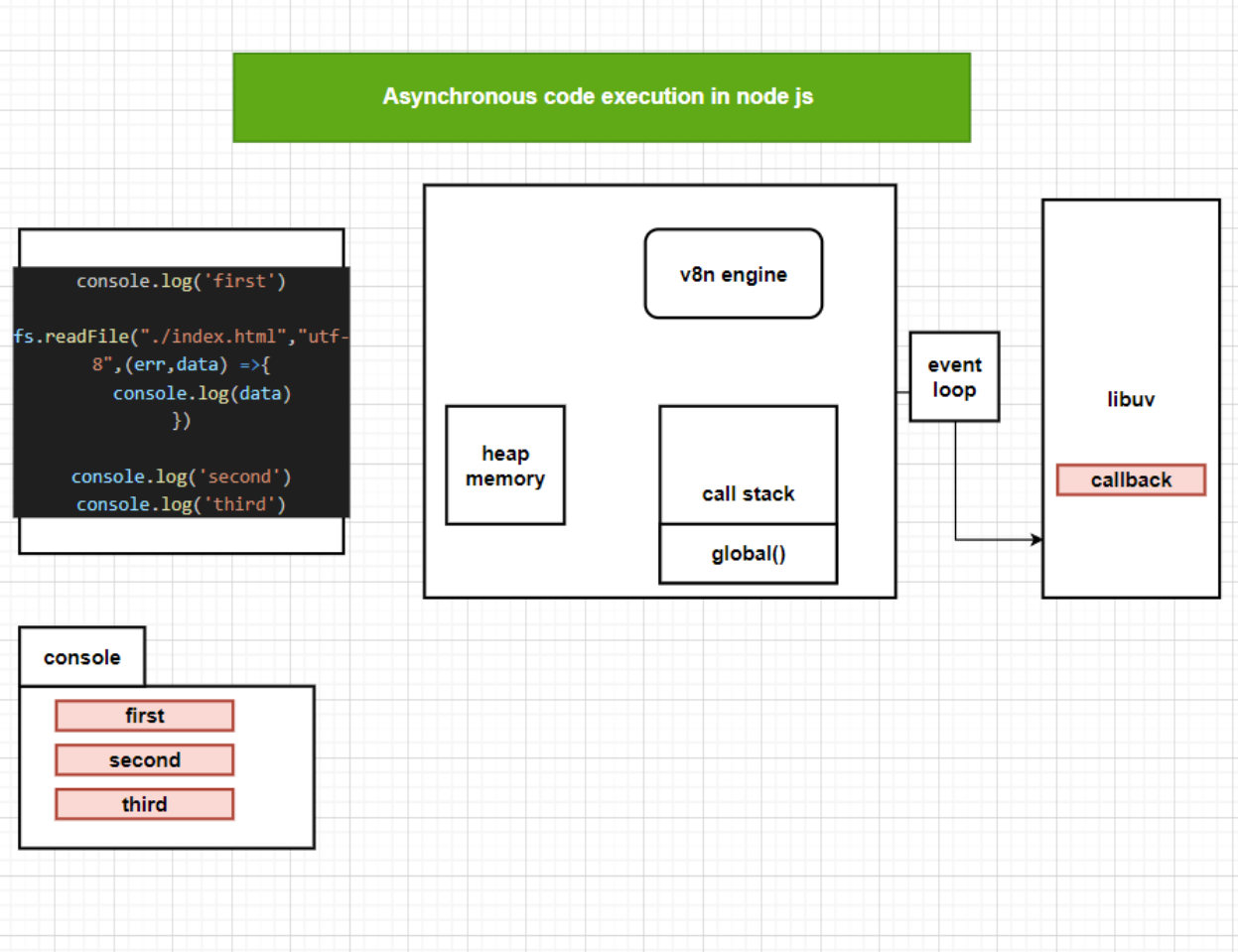
What is open source ?

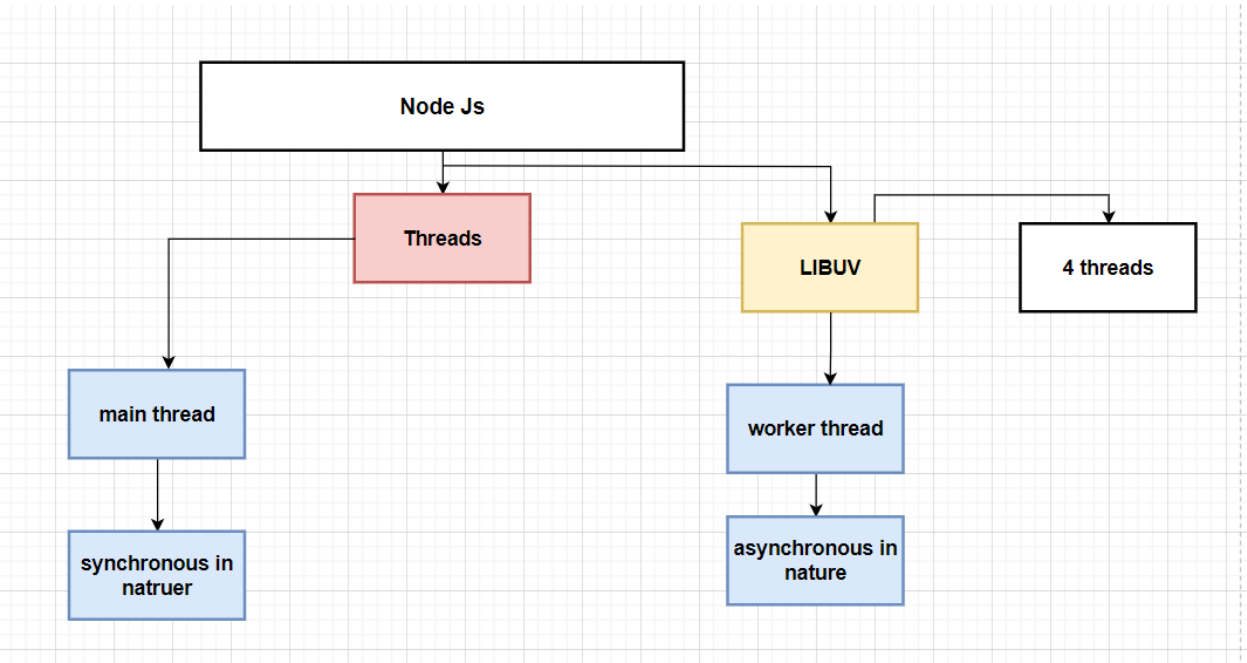
The term open source refers to something people can modify and share because its design is publicly accessible.

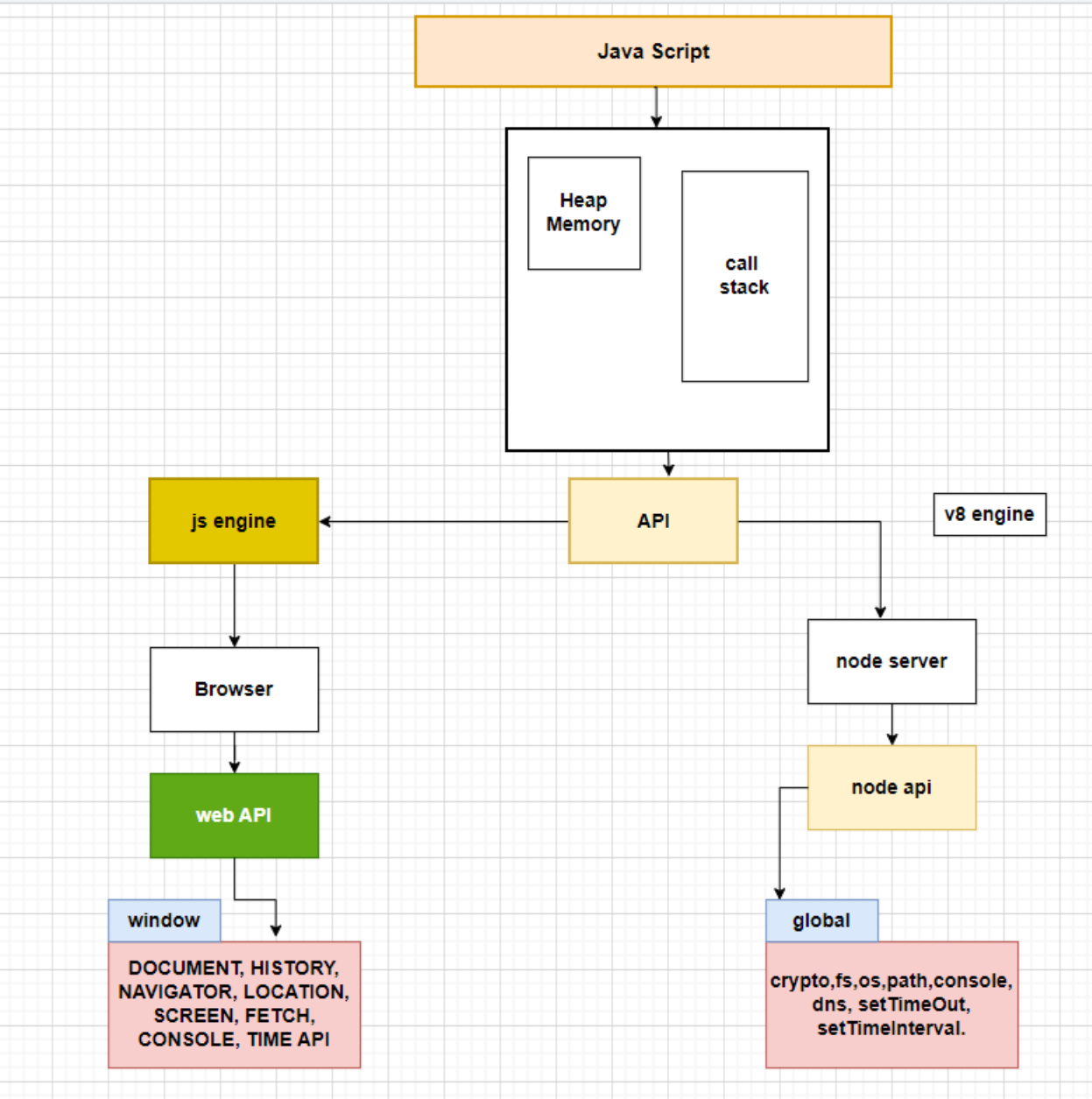
What is Cross-Platform ?

Cross-platform refers to the ability of a software application, framework, or technology to run on multiple operating systems or platforms without requiring significant modification. In the context of software development, platforms typically refer to different operating systems like Windows, macOS, Linux, iOS, Android, etc.

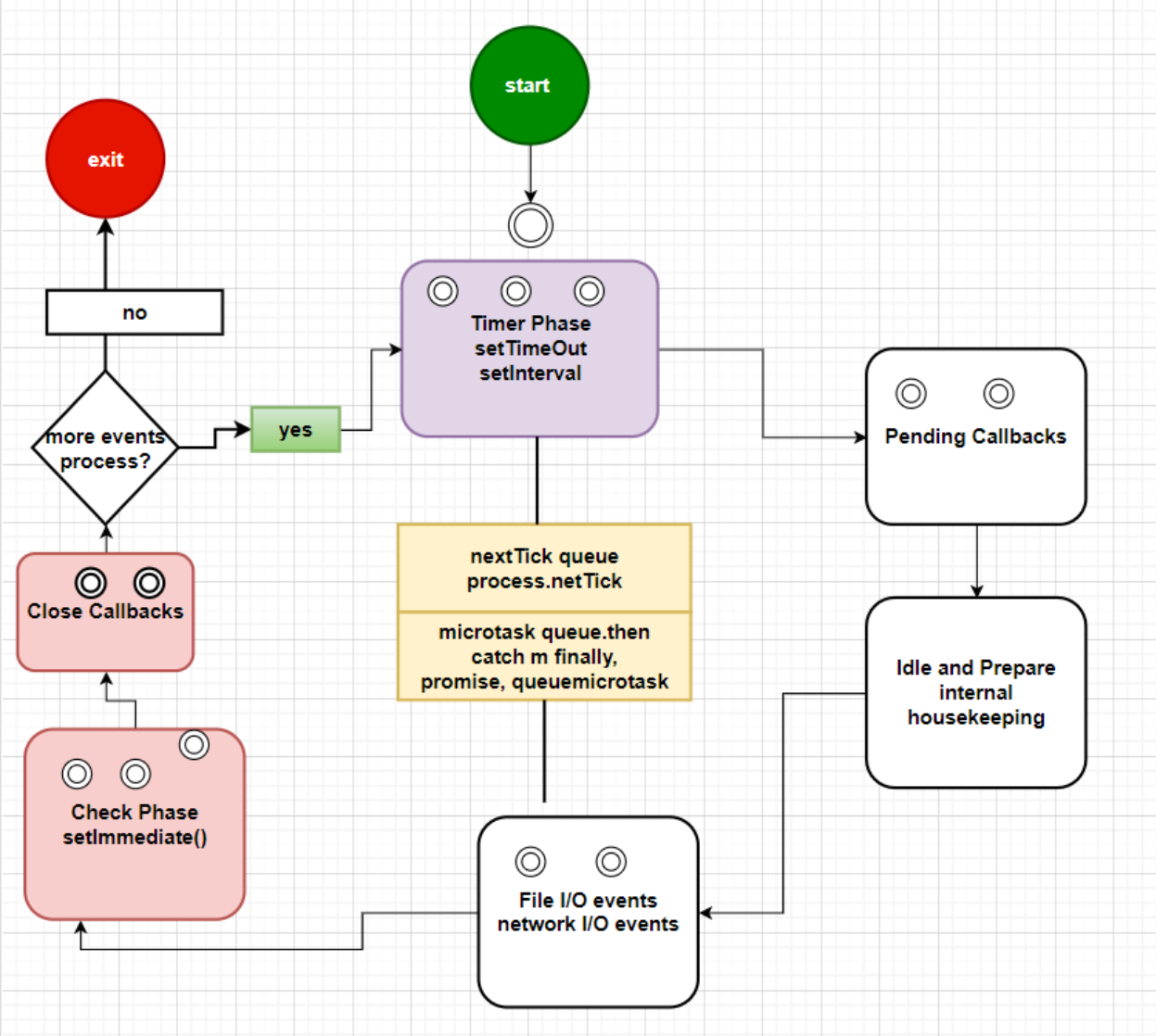



Asynchronous code execution in node js
console.log('first')
fs.readFile("./index.html","utf-8",(err,data) =>{
console.log(data)
})
console.log('second')
console.log('third')
v8n engine
heap memory
call stack
global()
event loop
libuv
callback
console
first
second
third


Node Js
Threads
LIBUV
4 threads
main thread
worker thread
synchronous in natruer
asynchronous in nature


Java Script
Heap Memory
call stack
API
v8 engine
js engine
Browser
web API
window
DOCUMENT, HISTORY, NAVIGATOR, LOCATION, SCREEN, FETCH, CONSOLE, TIME API
node server
node api
global
crypto,fs,os,path,console, dns, setTimeOut, setTimeInterval.

Event Loop


start
exit
no
more events process?
yes
Timer Phase
setTimeOut
setInterval
Pending Callbacks
nextTick queue
process.netTick
microtask queue.then
catch m finally,
promise, queuemicrotask
Close Callbacks
Idle and Prepare
internal
housekeeping
Check Phase
setImmediate()
File I/O events
network I/O events

Module In NodeJs

- a module is a piece of reusable code .
- it's a logical reusuable Js code.
- it helps the developer to dry (don't repeart yourself) principle in programming.
- they also help to break down complex logics into small piece or small or simple logical and manageble code.
- nodeJs internally uses Commonjs module (it is default module type in nodejs)
- es6 module also work in nodejs but have configuration .

Types Of node modules

1 : built-in module

- the core module or built-in module includes bare minimum functionality of Nodejs.
- the core modules are compiled into binary distribution and load automatically when NodeJs process starts.
- However you need to import core module at the first in order use it in your application.

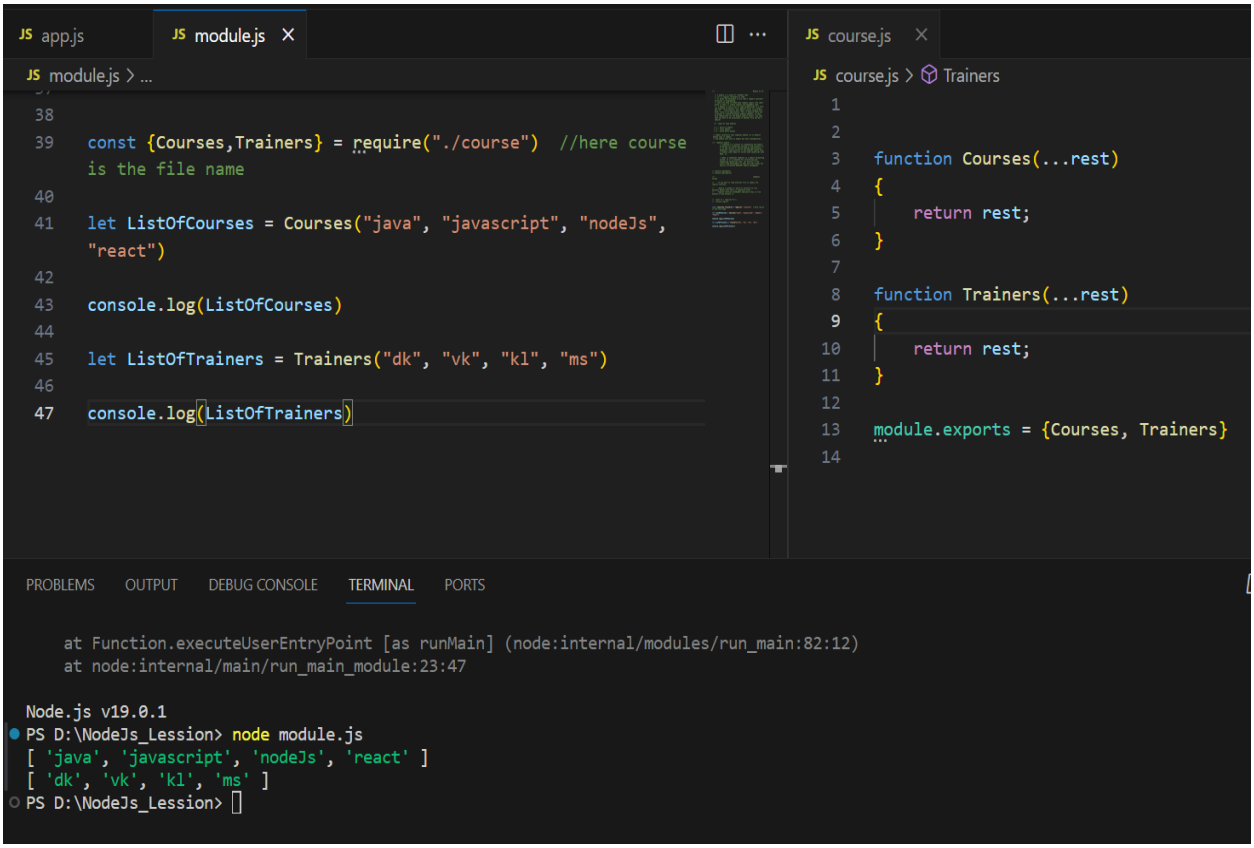
Example : *http , url , file system , path , crypto , util , http , os , events*

commonJs format

- if you want to load external file in nodejs use require function .
- require is global or built-in function to call external module that exits in separtate files .
- require function statements basically reas js file execute it and process it .

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

console.log(fs)
```

2 : local module



module.js

```
const val =
require("./course")

console.log(val.name)
console.log(val.courses)
console.log(val.duration)
console.log(val.availability)
```

course.js

```
module.exports = {
    name : "sashi",
    courses:
["js","nodejs","react"],
    duration : "5 months",
    availability : true
}
```

```
const val =
require("./course") ;

console.log(val.mern())
console.log(val.courses())
```

```
exports.mern = function()
{
    return {
        name : "sashi"
    }
}

// named export

exports.courses = function()
{
    return {
        name : "java",
        trainer : "dixith",
        duration : "4 month"
    }
}
```

```
PS D:\NodeJs_Lession> node module.js
{ name: 'sashi' }
{ name: 'java', trainer: 'dixith', duration: '4 month' }
PS D:\NodeJs_Lession>
```

course.js

```
//! case 1 :

function f1()
{
    return "function 1 "
}
function f2()
{
    return "function 2 "
}
function f3()
{
    return "function 3 "
}

module.exports = {f1,f2,f3};



//! case 2

function f1()
{
    return "function 1 "
}
function f2()
{
    return "function 2 "
}
function f3()
{
    return "function 3 "
}



module.exports.f1 = f1 ;
module.exports.f2 = f2 ;
module.exports.f3 = f3 ;


//! case 3

exports.f1 = function ()
```

module.js

```
const val = require("./course")

console.log(val)

//? case 1 for printing

console.log(val.f1())
console.log(val.f2())
console.log(val.f3())




//? case 2: with the hlp of
destructure

const {f1,f2,f3} = val;

console.log(f1())
console.log(f2())
console.log(f3())
```

```
{
    return "function 1 "
}
 exports.f2 = function ()
{
    return "function 2 "
}
exports.f3 =function ()
{
    return "function 3 "
}
```

3 : third party module :

npm is the world's largest **Software Library** (Registry)

npm is also a software **Package Manager** and **Installer**

npm is the world's largest **Software Registry.**

The registry contains over 800,000 **code packages.**

Open-source developers use **npm** to **share** software.

Many organizations also use npm to manage private development.

A cool thing about using modules in Node.js is that you can share them with others. The Node Package Manager (NPM) makes that possible. When you install Node.js, NPM comes along with it.

With NPM, you can share your modules as packages via the NPM registry. And you can also use packages others have shared.

If you want to install third party module

1. create package.json file

- The **package.json** file is the heart of Node.js system.
- It is the manifest file of any Node.js project and contains the metadata of the project.
- The package.json file is the essential part to understand, learn and work with the Node.js. It is the first step to learn about development in Node.js.

What does package.json file consist of?

The package.json file contains the metadata information. This metadata information in **package.json** file can be categorized into below categories.

1. **Identifying metadata properties:** It basically consist of the properties to identify the module/project such as the name of the project, current version of the module, license, author of the project, description about the project etc.
2. **Functional metadata properties:** As the name suggests, it consists of the functional values/properties of the project/module such as the entry/starting point of the module, dependencies in project, scripts being used, repository links of Node project etc.

Create a package.json file:

A **package.json** file can be created in two ways.

1. **Using npm init:** Running this command, system expects user to fill the vital information required as discussed above. It provides users with default values which are editable by the user.
   Syntax:

```
npm init
```

2. **Writing directly to file :** One can directly write into file with all the required information and can include it in the Node project.

Example: A demo **package.json** file with the required information.

```
{
  "name": "nodejs_lession",
  "version": "1.0.0",
  "description": "we are developing nodejs app",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "nodejs",
    "express"
  ],
  "author": "santanu",
  "license": "ISC"
}
```

How to use third-party packages

To use a third-party package in your application, you first need to install it. You can run the command below to install a package.

```
npm install <name-of-package>
```

For example, there's a package called `capitalize`. It performs functions like capitalizing the first letter of a word.
Running the command below will install the capitalize package:

```
npm install capitalize
```

To use the installed package, you need to load it with the `require` function.

```
const capitalize = require('capitalize')
```

And then you can use it in your code, like this for example:

```
const capitalize = require('capitalize')
console.log(capitalize("hello"))  // Hello
```

This is a simple example. But there are packages that perform more complex tasks and can save you loads of time.

```
C:\Users\subha>npm -l
npm <command>

Usage:

npm install        install all the dependencies in your project
npm install <foo>  add the <foo> dependency to your project
npm test           run this project's tests
npm run <foo>      run the script named <foo>
npm <command> -h   quick help on <command>
npm -l             display usage info for all commands
npm help <term>    search for help on <term> (in a browser)
npm help npm       more involved overview (in a browser)
```

- **Node. js** projects can have two types of dependencies: **production dependencies** and **development dependencies**.
- **Production dependencies** are required for your **application to run**, while **development dependencies** are only needed during **development** (e.g., testing frameworks, build tools)

Fs Module

```
//! the node:fs module enables interacting with the file system in a way modeled on standard POSIX functions.

const fs = require('fs');  // core module

console.log("read file synchronous started");

          //! synchronous way to read file

let readFileSync = fs.readFileSync("./data.txt","utf-8");

                //? utf :=> unicode transformation format

console.log(readFileSync);

         //!  asynchronously reading file

//? i/o queue

fs.readFile("./data.txt","utf-8", (err,data)=>{
    if(err) throw err;

    console.log(data)
})

console.log("read file synchronous ended");
```

```
                //!  reading json data

fs.readFile("./data.json", {encoding: "utf-8"},
(err,data)=>{

    if(err)
    {
        console.log(err)
    }
    else{
        if(data)
        {
            let convertIntoString = JSON.stringify(data);
            console.log(convertIntoString);

            let parseData = JSON.parse(convertIntoString)
            console.log(parseData)
        }
        else{
            console.log("not converted")
        }
    }
})
```

Write File In Node Js

```
//!  synchronous way


const fs = require('fs');

let readFile = fs.readFileSync("./data.json","utf-8")

let writeFile = fs.writeFileSync("person.json",readFile)

// ex: 2

let readFile = fs.readFileSync("./data.json","utf-8")
let personData = Math.floor(Math.random() * 1000)

let writeFile = fs.writeFileSync(`person
                                ${personData}.json`,readFile)
```

```
//!  asynchronous way

let readFile = fs.readFile('./data.json','utf-
8',(err,data)=>
{
    if(err) throw err;
    console.log("successfully file read that file next it
will copy the data and write ")

    fs.writeFile("x.json",data,(err)=>{
        if(err) throw err;

        console.log('file written successfully')
    })
})
```

create directory

```
//? Synchronously


fs.mkdirSync("src")
console.log("successfully directory created");

//? next create a file within x folder

fs.writeFileSync("src/app.js","my name is app.js" )


//! Asynchronously

fs.mkdir("src1",{},(err)=>{
    if(err) throw err;
    console.log('successfully directory created');
    fs.readFile("./data.json","utf-8",(err,data)=>{
        if(err) throw err;
        console.log("file read it successfully");

        fs.writeFile("src/app1.json",data,(err)=>{
            if(err) throw err;

            console.log('file written successfully')
        })
    })
})
```

```
                //!  Nested Directory


fs.mkdir("src3", {}, err =>{
    if(err) throw err;
    console.log("src3 folder created");

    fs.mkdir("src3/components", {}, err=>{
        if(err) throw err;
        console.log("components folder created");

        fs.mkdir("src3/components/navbar", {}, err=>{
            if(err) throw err;
            console.log("navbar folder created");

            fs.writeFile("src3/components/navbar/Navbar.jsx"
          ,"navbar content", err=>{
                if(err) throw err;
                console.log("navbar jsx created")
            })
        })
    })
})
```

delete file

```
//! synchronously

fs.unlinkSync("x.json")
console.log('file deleted successfully')


 //! asynchronously

fs.unlink("c.txt", err=>{
    if(err) throw err;
    console.log("file deleted successfully")
})
```

Delete Folder or directory

```
//! synchronously

fs.rmdirSync("src1")
console.log("directory removed")


//! asynchronously

fs.rmdir("yyy", err=>{
    if(err) throw err;

    console.log("directory removed");
})
```

Delete Nested Directory

```
fs.unlink("src3/components/navbar/Navbar.jsx", err =>{
    if(err) throw err;
    console.log("Navbar.jsx is deleted");

    fs.rmdir("src3/components/navbar", err=>{
        if(err) throw err;
        console.log("navbar folder is removed")

        fs.rmdir("src3/components", err=>{
            if(err) throw err;
            console.log("compnent folder is removed");

            fs.rmdir("src3", err=>{
                if(err) throw err;
                console.log("src3 is removed");
            })
        })
    })
})
```

Rename a File

```
//! synchronously

fs.renameSync('sa.js',"san.html")
console.log("rename successfully done ")


//! asynchronously

fs.rename("san.html","index.html",err=>{
    if(err) throw err;
    console.log("successfully file named ")
})
```

Rename Folder

```
fs.rename("src3", "public", err=>{
    if(err) throw err;

    console.log("successfully folder renamed")
})
```

Handling File System With the Help Of Promise (asynchronous)

Rename Folder

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

fs.rename("public","src3")
 .then(_=>
    console.log("successfully folder nenamed")
    ).
    catch(err=>
        console.log(err)
       )
```

Read File

```
fs.readFile("data.json","utf-8")
 .then(data=>{
    console.log(data)
 })
 .catch(err=> console.log(err))
```

Write File With Promise

```
fs.writeFile("san.txt","hello santanu")
 .then(_=>
    console.log("data written successfully")
    )
    .catch(err => console.log(err))
```

Create directory with promise

```
fs.mkdir("public")
 .then(_=>
    console.log("folder created")
    )
    .catch(err => console.log(err))
```

Promise way to delete file and folder

```
fs.unlink("public/san.txt")
 .then(_=>{
    console.log("file deleted")
    fs.rmdir("public")
    .then(_=>{
        console.log("folder deleted")
    })
    .catch(err => console.log(err))
 }).catch(err=>console.log(err))
```

async and await in fs module

read file using async and await

```
 let readFile= async () =>{
    const data = await fs.readFile("data.txt", "utf-8");
    console.log(data);
}

readFile()
```

write file using async and await

```
let writeFile = async()=>{
    let data = await fs.readFile("data.txt", "utf-8");
    await fs.writeFile("hello.txt",data)
    console.log("successfully written a file")
}

writeFile()
```

create directory using async and await

```
let createDirectory = async()=>{
    await fs.mkdir("public")
    console.log("successfully folder created ")
}

createDirectory()
```

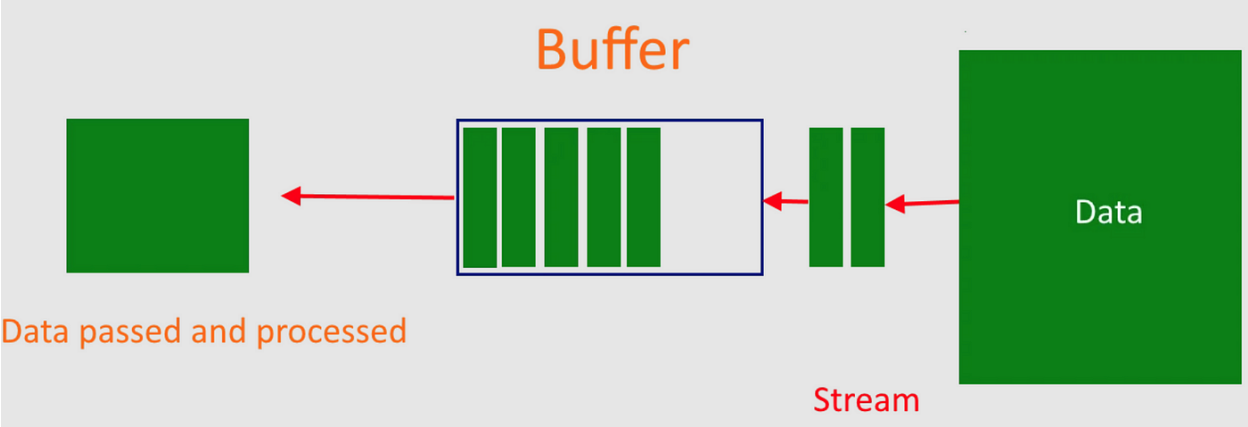
```
        create nested directory with async and await

let createNestedDirectory = async()=>{
    await fs.mkdir("public");
    await fs.mkdir("public/component")
    await fs.mkdir("public/component/auth")
    await
fs.writeFile("public/component/auth/login.jsx","login
component")

    console.log("successfully created nested folder ")

}

createNestedDirectory()

          delete nested folder with async and await

let removeNestedDirectory = async ()=>{

    await fs.unlink("public/component/auth/login.jsx")
    await fs.rmdir("public/component/auth")
    await fs.rmdir("public/component")
    await fs.rmdir("public")

    console.log("removed directory ")
}

removeNestedDirectory()
```

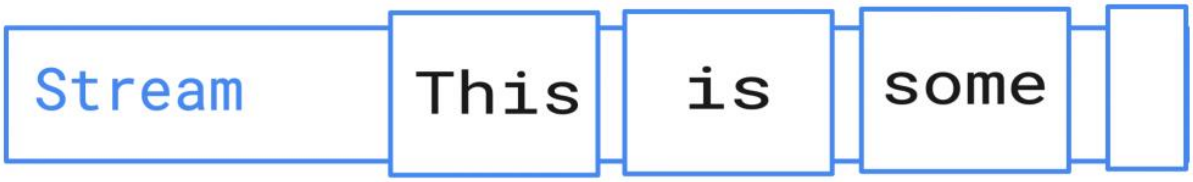
Append File

```
//! synchronous

const fs = require("fs")

fs.appendFileSync("data.txt","I am teaching nodeJs")
console.log("successfully appended")


//! asynchronously

fs.appendFile("data.txt","very good", err=>{
    if(err) throw err;
    console.log("data appened ")
})


//!  using async and await

const fs = require("fs").promises;

let appendData = async ()=>{
    await fs.appendFile("data.txt","how are you")
    console.log("with async await data is appened")
}

appendData()
```

Buffer

- a Buffer is a way to store and manipulate binary data in Node. js.
- Binary data refers to data that consists of binary values, as opposed to text data, which consists of characters and symbols.
- Examples of binary data include images, audio and video files, and raw data from a network.





Buffer

- Nodejs Buffer is a fixed size area of memory allocated outside of the V8 Javascript engine.
- They store sequences of integers similar to Javascript arrays, however, the difference is that once the buffer size has been allocated it cannot be changed, unlike javascript arrays.
- The Buffer class in Nodejs is designed to handle raw binary data, which leads to the next questions; what are binary data and why binary data.
- Binary data is a type of data represented in the binary numeral system (Base 2 number system).
- The circuits in computers are made up of billions of transistors.
- A transistor is a tiny switch that is activated by the electronic signals it receives. The digits 1 and 0 used in binary data reflect the on and off states of a transistor.
- Computers don't understand high-level languages (such as our everyday English), hence, before a computer can process a set of instructions it is encoded into binary data.

The Buffer Class in Node.Js

As mentioned earlier The buffer class in Node.Js is designed to handle raw binary data. It is a global class, so there is no need in using the **required()** method to import it.

How to create a Buffer

There are few ways to create a new Buffer which includes Buffer.from(), Buffer.alloc(size), Buffer.allocUnsafe(size).

Buffer.from()

```
   const buf1 = Buffer.from("Hello, welcome to Node.Js")
console.log(buf1) //<Buffer 48 65 6c 6c 6f 2c 20 77 65 6c 63
6f 6d 65 20 74 6f 20 4e 6f 64 65 2e 4a 73>

console.log(buf1[0]) //72
console.log(buf1[1]) //101
console.log(buf1[2]) //108
console.log(buf1[3]) //108

const buf2 = Buffer.from([23,78,12,9,89])
console.log(buf2) //<Buffer 17 4e 0c 09 59>

console.log(buf2[0]) //23
console.log(buf2[1]) //78
console.log(buf2[2]) //12
console.log(buf2[3]) //9
```

When creating a new Buffer from a string or array of integers, the corresponding response will be encoded in UTF-8. For example, **H** in **Hello welcome to Node.Js** has been converted to its Unicode equivalent of **48**
The numbers in each index of the Buffer indicate the position of the character in the Buffer. For example in **Buffer1**, the position of **H** in **Hello welcome to Node.Js** is **72**

To view the content of a string in Buffer, use **buf.toString()** as seen below

```
const buf = Buffer.from("Hello, welcome to Node.Js")
console.log(buf.toString()) //Hello, welcome to Node.Js
```

Buffer.alloc(size, fill)

Buffer.alloc(size, fill) allocates a new Buffer of size bytes. If fill is undefined it will fill the arrays with 0

```
const buf1 = Buffer.alloc(8)
 console.log(buf1) //<Buffer 00 00 00 00 00 00 00 00>

 const buf2 = Buffer.alloc(6, 2)
 console.log(buf2) //<Buffer 02 02 02 02 02 02>

 const buf3 = Buffer.alloc(9, 3)
 console.log(buf3) //<Buffer 03 03 03 03 03 03 03 03 03>
```

From the above, we created three buffers, **buf1** can only contain 8 bytes pre-filled with 0s, **buf2** can only contain 6 bytes pre-filled with 2s as specified and **buf3** can only contain 9 bytes pre-filled 3s as specified.
Let us try to write into **buf1**, you will notice that if you try to write into **buf1** more than its allotted size, it will only fill in the available space and discard the others. See example below

```
 const buf1 = Buffer.alloc(8)
 buf1.write("Hello, welcome to Node.Js")

 console.log(buf1.toString())//Hello, w
```

Buffer.allocUnsafe(size, fill)

Buffer.allocUnsafe is similar in operation with Buffer.alloc; they are both used to create a new buffer and allocate default byte size and values, but the difference is that in terms of performance Buffer.allocUnsafe is faster in creating the buffers. However, the trade-off is that the allocated segment of memory might contain old data that is potentially sensitive. A more detailed answer can be found here.

Other fun parts of Buffer

Buffer.isBuffer

This is used to check if a variable is a buffer similar to the Array.isArray (for checking if a variable is an array) as seen below

```
const buf = Buffer.allocUnsafe(8, 1)
  buf.write("Hello, welcome to Node.Js")

  console.log(Buffer.isBuffer(buf))//true
  console.log(Buffer.isBuffer("buf"))//false
```

buffer.length

This is used to check the length of a buffer

```
  const buf = Buffer.from("Hello, welcome to Node.Js")
  console.log(buf.length)//25
```

buffer.copy

buffer.copy(target, targetStart=0, sourceStart=0, sourceEnd=buffer.length)
It allows you to copy the content from one buffer to another. You can only specify the starting and the end position of the contents that are to be copied in the bufferCopy. Some examples will shed more light on the features of buffer.copy

```
 const buf = Buffer.from("Hello, welcome to Node.Js")
   const bufCopy = Buffer.alloc(26)

   buf.copy(bufCopy)
   console.log(bufCopy.toString()) //Hello, welcome to
Node.Js
```

```
   const buf = Buffer.from("Hello, welcome to Node.Js")
   const bufCopy = Buffer.alloc(26)


   buf.copy(bufCopy, 0, 12, 26)
   console.log(bufCopy.toString()) //me to Node.Js
```

buffer.slice

buffer.slice allows the ability to extract a section of the contents in the buffer

```
const buf = Buffer.from("Hello, welcome to Node.Js")

const newBuf = buf.slice(0, 5)
console.log(newBuf.toString()) //Hello
```

Streams in NodeJs

- Streams are objects that let us read data from a source or write data to a destination in continuous fashion.
- Streaming means listening to music or watching video in ‘real time’, instead of downloading a fie to your computer and watching it later.
- Stream is the method of transferring large amounts of data in an efficient way.
- They are used to read or write input into output sequentially. Furthermore, they are used to handling reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient manner.

Example :

A popular example to explain streaming is YouTube, whenever you play a video on YouTube. You will observe that the video is not available all at once, the player downloads the video in bits while you watch, peradventure your bandwidth is low, or you are experiencing a bad network, you will notice that you will get to a point where you have watched all the available downloaded frames and the YouTube player shows a loading animation trying to download the next available frames for you to watch.

Streaming is like the concept of Pay as You Go, you only pay for what you use now. Imagine YouTube did not implement the Streams concept, what this signifies is that for you to watch a 1-hour video on YouTube, you will need to wait for the player to download the video from the YouTube servers before it can become available for you to watch. Furthermore, if you lose interest in the video like 10 minutes into an hour video, the data used in downloading the other parts of the video is wasted.

Advantages of streams

1. **It is time-efficient** – Because data is available in bits, it takes less time to start processing the data than waiting for the whole data to be available.

2. **It is memory efficient** – It requires less memory to process the data because you don't need all the data to be available in the memory before processing it

Types of NodeJs Streams

1. **Readable stream –** A readable stream is an abstraction for a source from which data can be read, in other words, it lets you read data from a source

2. **Writeable stream –** A writable stream is an abstraction for a destination to which data can be written

3. **Duplex stream –** You can both read and write into it, in other words, it is a combination of both readable and writeable streams. Example – net.Socket

4. **Transform stream –** It is similar to a duplex stream, but it can modify the data as it is being written and read. Example – zlib.createGzip

Each type of **Stream is an EventEmitter** instance and throws several events at

different instance of times. For example, some of the commonly used events are:

data : this event is fired when there is data is available to read .

end : this event is fired when there is no data to read .

error : this event is fired when there is any error receiving or writing data .

finish : this event is fired when the data has been flushed to underlying system.

how to create Readable Stream

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

 let readStream = fs.createReadStream("./data.txt","utf-8");

   //access stream data we need nodejs event

   readStream.on('data', chunk =>{
    console.log(chunk)
   })


// stream also use for communication for client and server

 let readHtml = fs.createReadStream("./index.html","utf-8")
   readHtml.on("data", chunk =>{
    console.log(chunk);
   })
```

Writable Stream

```
let WritableStream = fs.createWriteStream("login.html")

// read stream

let readStream = fs.createReadStream("./index.html","utf-8")

readStream.on('data' , chunk =>{

    WritableStream.write(chunk,err =>{

        if(err) throw err;
        console.log(chunk)
        console.log("successfully data written")
    })
})

-------------------------------------------------------------
```

Printing the Data chunk by chunk

```
const readable = fs.createReadStream('./data.txt',
{highWaterMark : 20});

//? highWaterMark determines how much data buffer inside the
stream.
//? by default it is 64kb

readable.on('data' , chunk =>{
    console.log(`read ${chunk.length} bytes
${chunk.toString()} `)
})
```

```
    Lorem ips
read 20 bytes um dolor sit amet, c
read 20 bytes onsectetur adipisici
read 20 bytes ng elit. Corporis ea
read 20 bytes que minima magni ea
read 20 bytes sed! Suscipit in vol
read 20 bytes uptatum repudiandae
read 20 bytes neque ea aliquid at
read 20 bytes quae sed! Ex, ipsa.
read 20 bytes At facilis nostrum p
read 20 bytes erferendis accusanti
read 20 bytes um sed, pariatur sun
read 20 bytes t, consequatur verit
read 20 bytes atis repellendus adi
read 20 bytes pisci architecto neq
read 20 bytes ue ullam a sint nihi
read 20 bytes l sit ea quos esse r
read 20 bytes epellat cumque expli
read 20 bytes cabo aliquid ad. Dig
read 20 bytes nissimos quas dicta
read 20 bytes mollitia dolorum ips
```

Duplex Streams

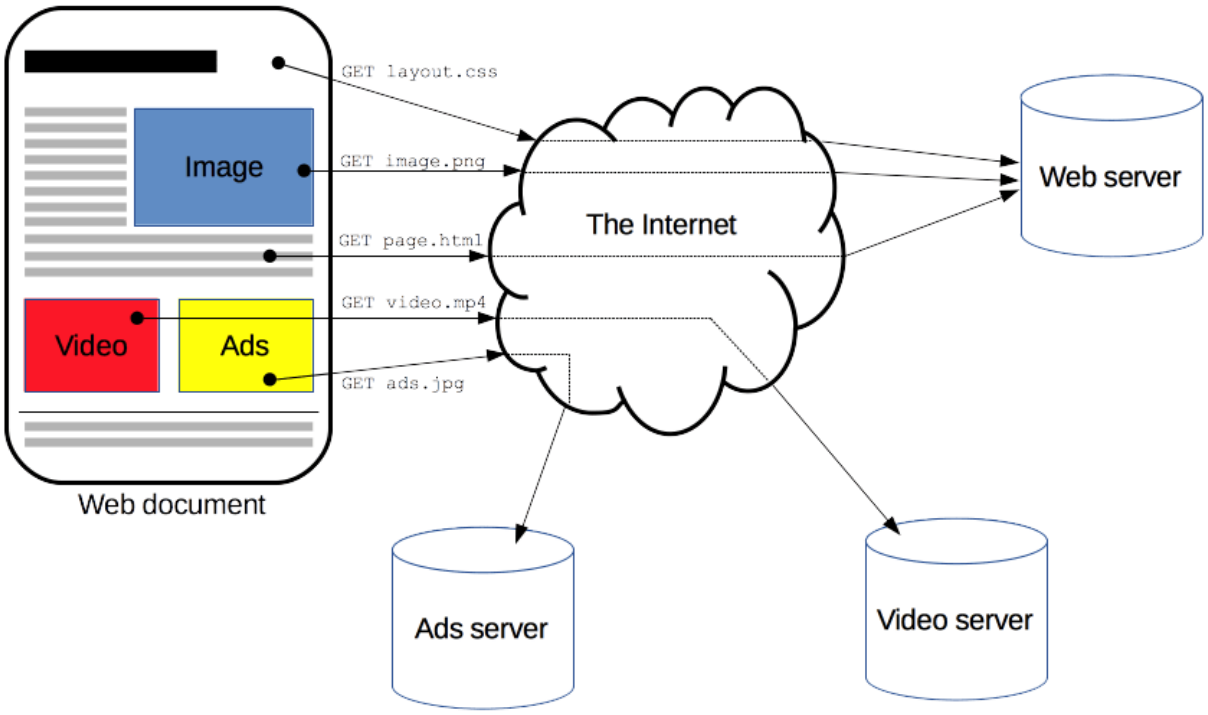
- These streams represent both a readable and a writable stream.
- It means you can both read from and write to these streams simultaneously.
- Duplex streams are bidirectional and are commonly used for tasks like network communication.

Stream.pipe() : It is the method used to take a readable stream and connect to a writeable stream.

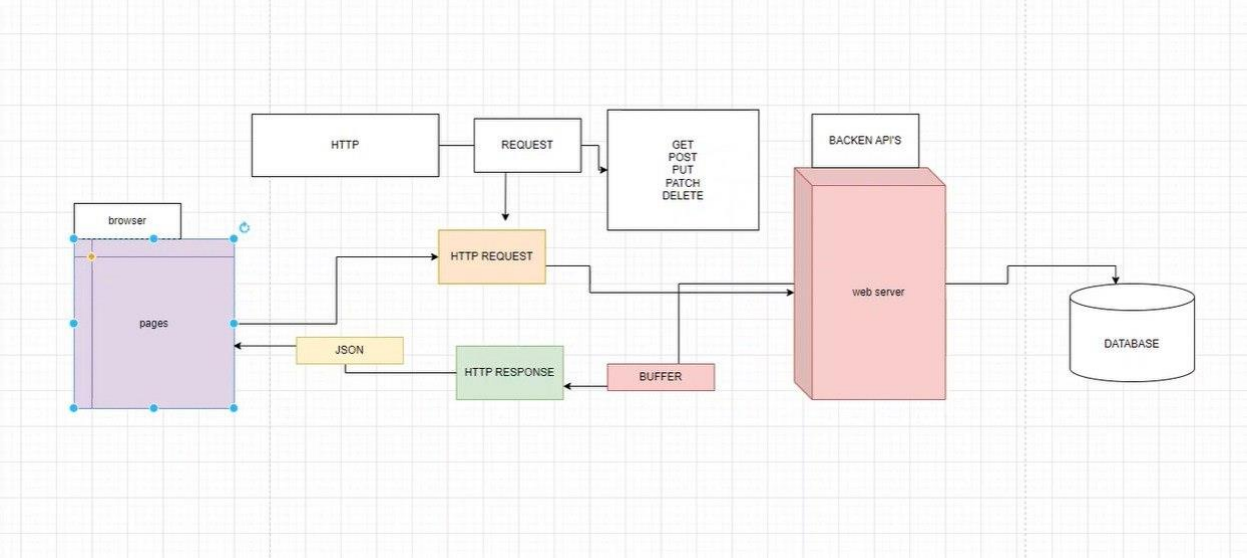
```
const fs = require('fs');

const stream = require('stream');

const duplex = new stream.Duplex({
    read(size){
        this.push("simple data can read it from readable stream");
    },

    write(chunk, encoding, callback){
        console.log('written chunk to strream', chunk.toString());
        variable = chunk.toString()
        callback()
    }
});

global.process.stdin.pipe(duplex).pipe(global.process.stdout)


//? ex : 2

let readableStream = fs.createReadStream("./index.html","utf-8")
let WritableStream = fs.createWriteStream("login2.html")

//? Duplex stream
readableStream.pipe(WritableStream);

console.log("successfully read the file and wrote it in file ")
```

HTTP Module

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response. HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests.



https://developer.mozilla.org/en-US/docs/Web/HTTP



▼ HTTP request methods

CONNECT

DELETE

GET

HEAD

OPTIONS

PATCH

POST

PUT

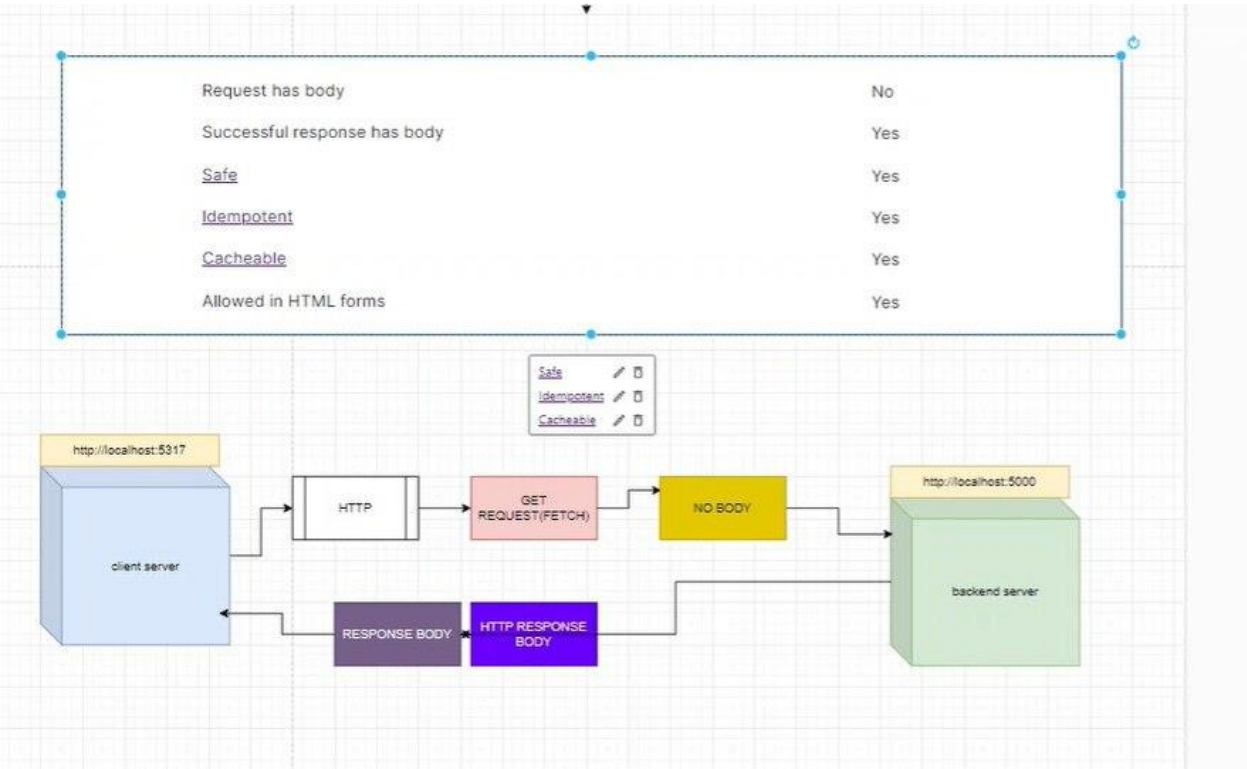
TRACE

GET

The **HTTP GET method** requests a representation of the specified resource. Requests using GET should only be used to request data (they shouldn't include data).

Note:

Sending body/payload in a GET request may cause some existing implementations to reject the request — while not prohibited by the specification, the semantics are undefined. It is better to just avoid sending payloads in GET requests.



Safe (HTTP Methods)

An HTTP method is **safe** if it doesn't alter the state of the server. In other words, a method is safe if it leads to a read-only operation. Several common HTTP methods are safe: `GET`, `HEAD`, or `OPTIONS`. All safe methods are also idempotent, but not all idempotent methods are safe. For example, `PUT` and `DELETE` are both idempotent but unsafe.

Idempotent

An HTTP method is **idempotent** if the intended effect on the server of making a single request is the same as the effect of making several identical requests.

This does not necessarily mean that the request does not have *any* unique side effects: for example, the server may log every request with the time it was received. Idempotency only applies to effects intended by the client: for example, a POST request intends to send data to the server, or a DELETE request intends to delete a resource on the server.

Cacheable

A **cacheable** response is an HTTP response that can be cached, that is stored to be retrieved and used later, saving a new request to the server. Not all HTTP responses can be cached; these are the constraints for an HTTP response to be cacheable:

Post Request

The **HTTP POST method** sends data to the server. The type of the body of the request is indicated by the Content-Type header.

The difference between PUT and POST is that PUT is idempotent: calling it once or several times successively has the same effect (that is no *side* effect), where successive identical POST may have additional effects, like passing an order several times.

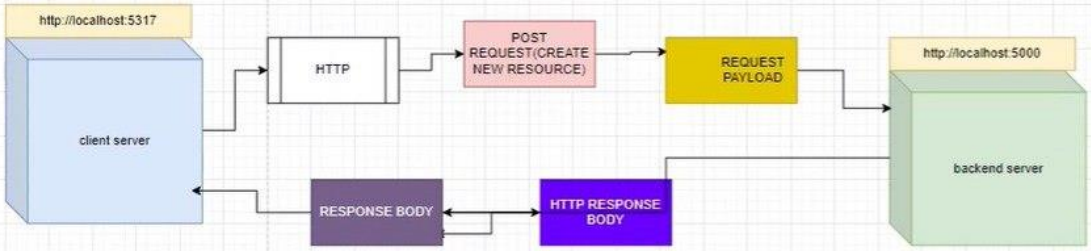
A POST request is typically sent via an HTML form and results in a change on the server. In this case, the content type is selected by putting the adequate string in the enctype attribute of the <form> element or the formenctype attribute of the <input> or <button> elements:

- application/x-www-form-urlencoded: the keys and values are encoded in key-value tuples separated by '&', with a '=' between the key and the value. Non-alphanumeric characters in both keys and values are URL encoded: this is the reason why this type is not suitable to use with binary data (use multipart/form-data instead)
- multipart/form-data: each value is sent as a block of data ("body part"), with a user agent-defined delimiter ("boundary") separating each part. The keys are given in the Content-Disposition header of each part.
- text/plain

When the POST request is sent via a method other than an HTML form, such as a fetch() call, the body can take any type. As described in the HTTP 1.1 specification, POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Adding a new user through a signup modal;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

Request has body	YES
Successful response has body	Yes
Safe	NO
Idempotent	NO
Cacheable	Only if freshness information is included
Allowed in HTML forms	Yes



Put Request

The **HTTP PUT request method** creates a new resource or replaces a representation of the target resource with the request payload.

The difference between PUT and POST is that PUT is idempotent: calling it once or several times successively has the same effect (that is no *side* effect), whereas successive identical POST requests may have additional effects, akin to placing an order several times.

Request has body	Yes
Successful response has body	May
Safe	No
Idempotent	Yes
Cacheable	No
Allowed in HTML forms	No

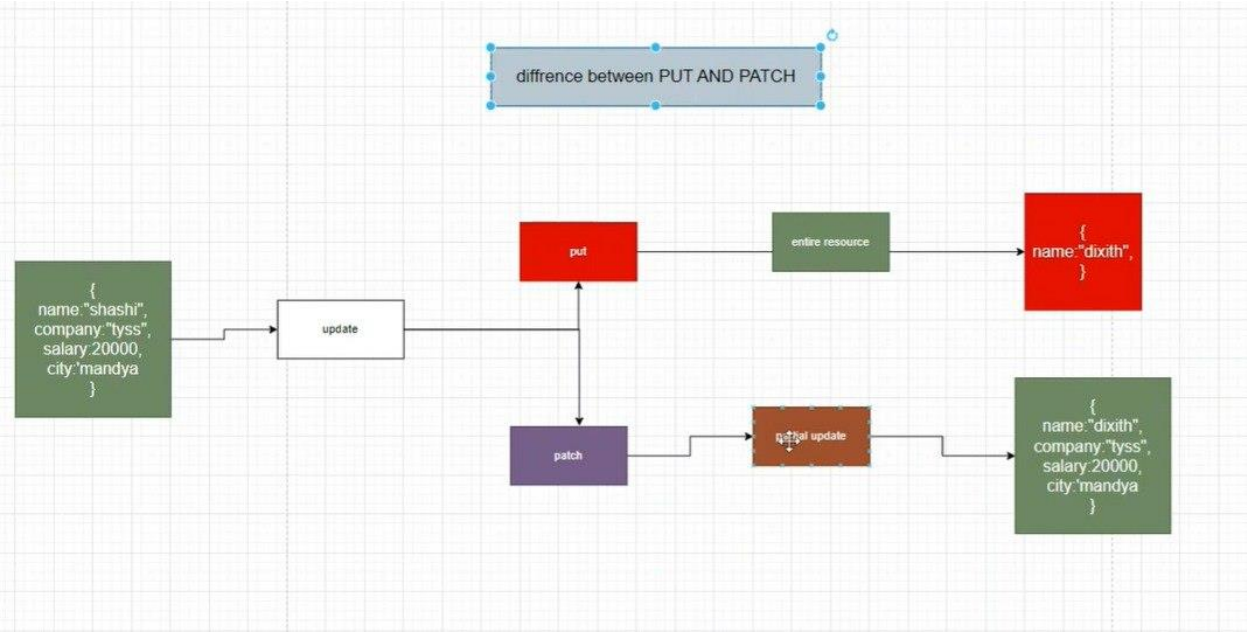
Patch Request

The **HTTP PATCH request method** applies partial modifications to a resource.

PATCH is somewhat analogous to the "update" concept found in CRUD (in general, HTTP is different than CRUD, and the two should not be confused).

A PATCH request is considered a set of instructions on how to modify a resource. Contrast this with PUT; which is a complete representation of a resource.

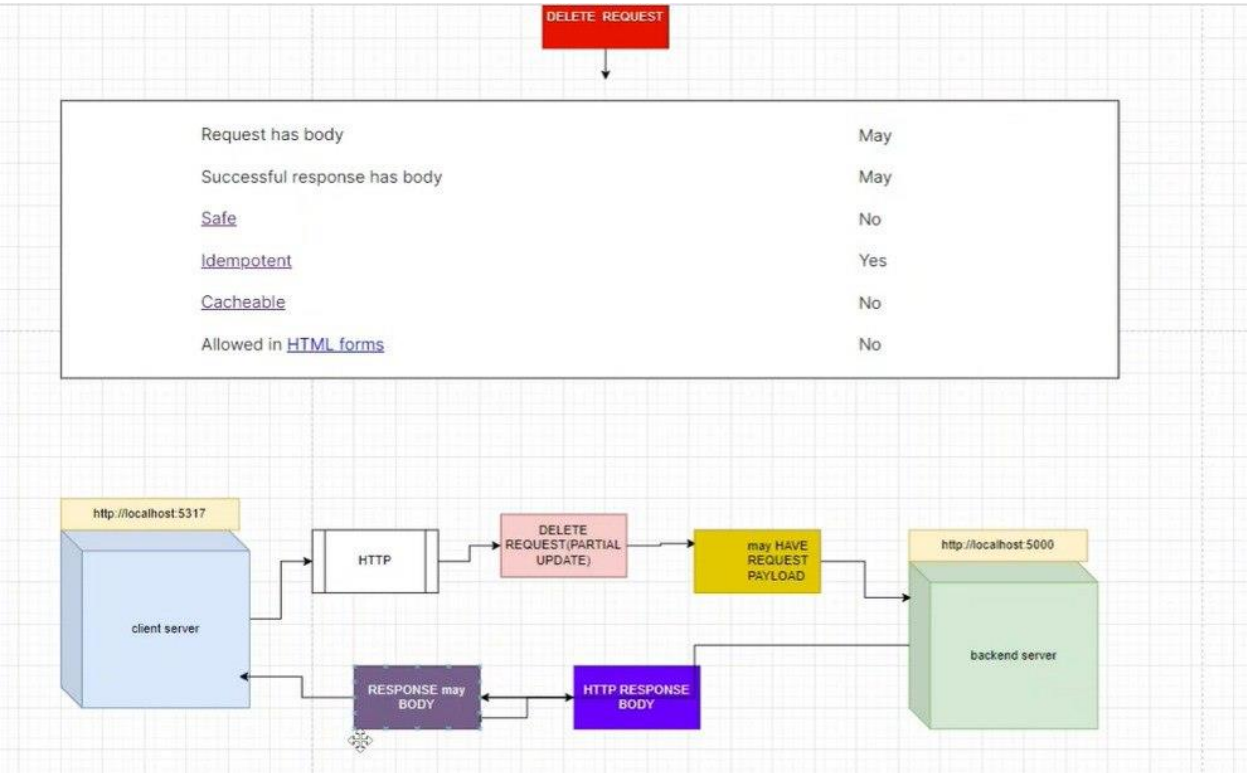
Request has body	Yes
Successful response has body	May
Safe	No
Idempotent	No
Cacheable	Only if freshness information is included
Allowed in HTML forms	No



Delete Request

The **HTTP `DELETE` request method** deletes the specified resource.

Request has body	May
Successful response has body	May
Safe	No
Idempotent	Yes
Cacheable	No
Allowed in HTML forms	No



Status Code

```
STATUS_CODES: {
  '100': 'Continue',
  '101': 'Switching Protocols',
  '102': 'Processing',
  '103': 'Early Hints',
  '200': 'OK',
  '201': 'Created',
  '202': 'Accepted',
  '203': 'Non-Authoritative Information',
  '204': 'No Content',
  '205': 'Reset Content',
  '206': 'Partial Content',
  '207': 'Multi-Status',
  '208': 'Already Reported',
  '226': 'IM Used',
  '300': 'Multiple Choices',
  '301': 'Moved Permanently',
  '302': 'Found',
  '303': 'See Other',
  '304': 'Not Modified',
  '305': 'Use Proxy',
  '307': 'Temporary Redirect',
  '308': 'Permanent Redirect',
  '400': 'Bad Request',
  '401': 'Unauthorized',
  '402': 'Payment Required',
  '403': 'Forbidden',
  '404': 'Not Found',
  '405': 'Method Not Allowed',
  '406': 'Not Acceptable',
  '407': 'Proxy Authentication Required',
  '408': 'Request Timeout',
  '409': 'Conflict',
  '410': 'Gone',
  '411': 'Length Required',
  '412': 'Precondition Failed',
  '413': 'Payload Too Large',
  '414': 'URI Too Long',
  '415': 'Unsupported Media Type',
  '416': 'Range Not Satisfiable',
  '417': 'Expectation Failed',
  '418': "I'm a Teapot",
  '421': 'Misdirected Request',
  '422': 'Unprocessable Entity',
  '423': 'Locked',
  '424': 'Failed Dependency',
  '425': 'Too Early',
  '426': 'Upgrade Required',
  '428': 'Precondition Required',
  '429': 'Too Many Requests',
  '431': 'Request Header Fields Too Large',
  '451': 'Unavailable For Legal Reasons',
  '500': 'Internal Server Error',
  '501': 'Not Implemented',
  '502': 'Bad Gateway',
  '503': 'Service Unavailable',
  '504': 'Gateway Timeout',
  '505': 'HTTP Version Not Supported',
  '506': 'Variant Also Negotiates',
  '507': 'Insufficient Storage',
  '508': 'Loop Detected',
```

what is http ?

1. hyper text transfer protocol , needs to transfer data between client and server .
2. it is an application and communication protocol
3. In NodeJs http is a built-in module so no need to install it use import and create server .

4. http is a protocol which allows the fetching resources, such as html documents.
5. http is the foundation of any data exchange on the web and it is a client-server protocol which means requests are initiating by the client or recipients, usually the web browser.
6. To create web server we need http module in nodejs

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

fetch all http methods

```
let methods =http.METHODS;
console.log(methods)
```

//! important methods => GET , POST, PUT, DELETE, PATCH

CRUD

```
//? => CREATE , READ , UPDATE , DELETE

//? C => POST       => create new resources
//? R => GET        => fetch data from the source or read
                       data use http GET method
//? U => PUT/PATCH  => to new resources or existing
                       resources to modify or update
//? D => DELETE     => deleting resources
```

Create Server

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

const server = http.createServer((req, res)=>{

    //? set header

    //syntax :  res.writeHead(statusCode, headers)

    res.writeHead(200,'ok',{"Content-Type":"text/plain"});

   //? body and ending cycle

   res.end("welcome to nodejs world")

})
.listen(5000, err=>{
    if(err) throw err;

    console.log("server is running on port number 8000")
})
```

Serve html file

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

http.createServer((req,res)=>{

    fs.readFile('./index.html',"utf-8" , (err,data)=>{
        if(err) {
            console.log(err);
            res.end();
        }
        else{
            res.writeHead(200, 'ok', {"Content- Type":
                                                "Text/html"});
            res.write(data);
            res.end();
        }
    })

})
.listen(5000, err =>{
    if(err) throw err;

    console.log("server is runing on port number 5000")
})
```

Serve Html And Other Resources With Stream

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

const server = http.createServer((req,res)=>{

 res.writeHead(200, 'ok', {"Content-Type":"text/html"});

let readStrem = fs.createReadStream("./index.html","utf-8");

   readStrem.pipe(res)

})


// listen a port

let PORT = 5000;

server.listen(PORT , err=>{
    if(err) throw err;

    console.log("App is running on port number",PORT)
})
```

Serve Css File From Server

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

http.createServer((req,res)=>{
    res.writeHead(200,'ok',{"Content-Type":"text/css"})

 let readStrem = fs.createReadStream('./style.css','utf-8');

    readStrem.pipe(res)
})
.listen(5000, err=>{
    if(err) throw err;

    console.log("running on port number 5000")
})
```

Routing In Node

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

let server = http.createServer((req,res)=>{

    if(req.url ==='/'){

        res.writeHead(200,'ok',{"Content-Type":"text/html"})
        let index = fs.createReadStream("./public/index.html","utf-8");
        index.pipe(res);
    }
    else{
        res.writeHead(400,{"Content-Type":"text/html"})
        let notFound = fs.createReadStream("./routes/PageNotFound.html");
        notFound.pipe(res);
    }
})

let PORT = 5000;
server.listen(PORT , err=>{
    if(err) throw err;

    console.log("web server is running on port number",PORT)
})
```

```
let http = require('http')
let fs = require('fs');

const server = http.createServer((req,res)=>{

    // setHeader

    res.setHeader('Content-Type',"text/html");
    let path ="public/";

    switch(req.url)
    {
        case '/' :
                path += 'index.html'
                break;
        case '/login':
                path += 'login.html'
                break;
        case '/about':
                path += 'about.html'
                break;
        case '/style/css':
                path += 'style.css'
                res.setHeader("Content-Type", "text/css")
                break;
        case '/api':
                path += 'users.json'
                res.setHeader("Content-Type","application/json")
                break;
        default :
                path += 'pageNotFound.html'
                res.statusCode = 404;
                break;
    }

    fs.createReadStream(path, 'utf-8').pipe(res)

}).listen(5000, err=>{
    if(err) throw err;

    console.log('sever is running on port number 5000')
})
```

The Node.js Event emitter

- If you worked with JavaScript in the browser, you know how much of the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on.
- On the backend side, Node.js offers us the option to build a similar system using the events module.
- This module, in particular, offers the EventEmitter class, which we'll use to handle our events.

You initialize that using

```
const EventEmitter = require('node:events');
const eventEmitter = new EventEmitter();
```

This object exposes, among many others, the on and emit methods.

emit is used to trigger an event

on is used to add a callback function that's going to be executed when the event is triggered

For example, let's create a start event, and as a matter of providing a sample, we react to that by just logging to the console:

```
eventEmitter.on('start', () => {
    console.log('started');
  });

  eventEmitter.emit('start');
```

You can pass arguments to the event handler by passing them as additional arguments to emit():

```
eventEmitter.on('start', number => {
    console.log(`started ${number}`);
  });
  eventEmitter.emit('start', 23);
```

Multiple arguments:

```
eventEmitter.on('start', (start, end) => {
    console.log(`started from ${start} to ${end}`);
  });
  eventEmitter.emit('start', 1, 100);
```

The EventEmitter object also exposes several other methods to interact with events, like

- once(): add a one-time listener
- removeListener() / off(): remove an event listener from an event
- removeAllListeners(): remove all listeners for an event

https://nodejs.org/en/learn/asynchronous-work/the-nodejs-event-emitter

```
//? event emitter object

const event = require("events");
                                        // method 1

const dog = new event();

console.log(dog)

// ! Basic event handling methods

// on , once , off

// ! creating event name and sounding

dog.on('bark', ()=>{
    console.log("Dog is barking ... like bow bow ")
})


// ! emit events by unsing emit method

dog.emit("bark")


class Person extends event.EventEmitter{}

                                            // method 2
let person = new Person();
// console.log(person)

//? create event for person

person.on('speech', ()=>{
    console.log("hello i am speacking english...")
})

person.emit('speech')
```

```
class Cat extends event.EventEmitter{}

let cat = new Cat();

cat.on("pet", ()=>{
    console.log("hello i'm cat .. sounding like mew mew ")
})

cat.emit("pet")


class Student extends event.EventEmitter{};

let student = new Student();

student.on("study",()=>{
    console.log("i'm student , i'm studying")
})

student.emit("study")
```

What is urlencoded ?

You might be referring to application/x-www-form-urlencoded, which is a common content type used when submitting data through HTML forms on the web. It is a way of encoding key-value pairs as a string in the format of key1=value1&key2=value2.

formData.js

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

const queryString = require("querystring");

let server = http.createServer((req,res)=>{

    //! HTTP method for sending data to the server from client

    if(req.method ==='POST')
    {
            // console.log(req.method);
            // console.log("data triggered")

  let FORM_URLENCODED = "application/x-www-form-urlencoded"
  if (req.headers["content-type"] === FORM_URLENCODED){

                //! event

                // console.log(req)

                let body ="";
                req.on("data", chunk=>{

                 let value = chunk.toString();
                 body += value;

                });
```

```
                //! Close  event

          req.on('end',_=>{
          let payload = queryString.parse(body.valueOf());
          let email = payload.email;
          let password = payload.password;
          res.end(`Thank You ${email} For Subscription...
           your password is ${password} `)
                })
            }
            else{
                res.end(null)
            }
    }else{

      if(req.url ==="/"){

      let readHtml =
            fs.createReadStream("./formData.html","utf-8");
            readHtml.pipe(res);
        }
        else if(req.url ==="/stylesheet") {

       res.writeHead(200, {"Content-Type":"text/css"})
     fs.createReadStream("./formData.css","utf-8").pipe(res)
       }
       else{
            res.end("<h1> Page Not Found 404 </h1>")}
    }
})

server.listen(5000, err=>{
    if(err) throw err;
    console.log("web server is connected and listen 5000
Port number")
})
```

How To Connect With MongoDB

```
const mongodb = require("mongodb").MongoClient;

//? The MongoClient class is a class that allows for making
Connections to MongoDB. remarks. The programmatically provided options
take precedence over the URI options. example // Connect using a
MongoClient instance const MongoClient = require('mongodb').

//!  Connect to the mongoDB server  ==> mongodb://localhost:27017

mongodb.connect("mongodb://localhost:27017")
      .then(()=>{
        console.log("mongodb server connected successfully")
      })
      .catch(err => console.log(err))
```

CRUD Operation In MongoDb using NodeJs

```
const mongodb = require("mongodb").MongoClient;
let URL = "mongodb://localhost:27017"

//? connnect mongodb server

let ConnectDb = async()=>{
    try{

        let db = await mongodb.connect(URL);
        console.log("database server connected")

        //! Create a database

            let database = db.db("testYantra");

            //! create collection

            let collection = await database.createCollection("users");
```

```
            console.log("collection has been created ")

            //! insert data

            collection.insertOne({name:"shashi", company:"Qspider"})
            console.log("data inserted successfully")

           //! insert more data from git

            let data = await fetch("https://api.github.com/users")
            let finalData = await data.json();
            collection.insertMany(finalData)
            console.log("data inserted successfully")

    }
    catch(err)
    {
        console.log(err)
    }
}

ConnectDb()



                    //! find all documents

let users = await collection.find({}).toArray();
console.log(users)

                    //! find One Document

let user = await collection.findOne();
console.log(user)

                //! find perticular document

let filterUser = await collection.findOne({login: "pjhyett"});
console.log(filterUser)

                    //! find data in range

let usersData = await collection.find({id: { $lte: 10}}).toArray();
console.log(usersData)
```

```
//! Delete one document

let removeDocument = await collection.deleteOne({login: "pjhyett"});
console.log("user deleted successfully")

//! Delete All Documents

let removeAllDocument = await collection.deleteMany({});
console.log("All users deleted successfully")

//!  Update one  Documents

let updateDocument = await collection.updateOne(

{id:1},
{ $set : {login:"shashi"}},
{upsert : true}
)
console.log('successfully updated',updateDocument)

//! update many

let updateManyDocument = await collection.updateMany(
{
id: { $lte:10},
},
{
$set: { login : `user - ${Math.round(Math.random() * 100)}`}
},
{
upsert : true
}
)
console.log("successfully updated",updateManyDocument)
```

Path Module

The path module provides utilities for working with file and directory path.

Syntax

⇨ var path = require('path');

Path Properties and Methods

Method	Description
basename()	Returns the last part of a path
delimiter	Returns the delimiter specified for the platform
dirname()	Returns the directories of a path
extname()	Returns the file extension of a path
format()	Formats a path object into a path string
isAbsolute()	Returns true if a path is an absolute path, otherwise false
join()	Joins the specified paths into one
normalize()	Normalizes the specified path
parse()	Formats a path string into a path object
posix	Returns an object containing POSIX specific properties and methods
relative()	Returns the relative path from one specified path to another specified path
resolve()	Resolves the specified paths into an absolute path
sep	Returns the segment separator specified for the platform
win32	Returns an object containing Windows specific properties

```js
const path = require("path");

console.log(__filename);
// D:\NodeJs_Lession\Module\path.js

console.log(__dirname)
// D:\NodeJs_Lession\Module


                //!  Methods Of Path Module

//! 1. basename

console.log(path.basename(__filename));     //? path.js
console.log(path.basename(__dirname));      //? module


//! 2. extname

console.log(path.extname(__filename));     //? .js
console.log(path.extname(__dirname));      //?


//! 3. parse()

console.log(path.parse(__filename))

{
       root: 'D:\\',
       dir: 'D:\\NodeJs_Lession\\Module',
       base: 'path.js',
       ext: '.js',
       name: 'path'
     }
```

```
 //! 4. format()

console.log(path.format(path.parse(__filename)))
// D:\NodeJs_Lession\Module\path.js

 //! 5. isAbsolute()

console.log(path.isAbsolute(__filename))       //? true
console.log(path.isAbsolute("./fs.js"))        //? false


//! 6. join()

console.log(path.join("folder1","folder2","index.js"))
// folder1\folder2\index.js

console.log(path.join("/folder1","folder2","index.js"))
// \folder1\folder2\index.js

console.log(path.join("/folder1","//folder2","index.js"))
// \folder1\folder2\index.js

console.log(path.join("folder1","//folder2","../index.js"))
  // \folder1\index.js

console.log(path.join(__dirname,"data.json"))
    // D:\NodeJs_Lession\Module\data.json
```

Note :

- **in join method if we are giving double (//) also it will consider only one.**
- **When we are using (../) it will not take the previous path.**

```
                        //! 7. resolve

console.log(path.resolve("folder1","folder2","index.js"))
// D:\NodeJs_Lession\Module\folder1\folder2\index.js

// when we are not taking (/) at the beginning it will take
absolute path

console.log(path.resolve("/folder1","folder2","index.js"))
// \folder1\folder2\index.js

console.log(path.resolve("/folder1","//folder2","index.js"))
// \folder2\index.js

// if we give (//) it will consider that as root

console.log(path.resolve("folder1","//folder2","../index.js"
))
// \index.js


console.log(path.resolve(__dirname,"data.json"))
// D:\NodeJs_Lession\Module\data.json
```

OS Module

The OS module provides information about the computer's operating system.

Syntax

⇨ var os = require('os');

OS Properties and Methods

Method	Description
arch()	Returns the operating system CPU architecture
constants	Returns an object containing the operating system's constants for process signals, error cotes etc.
cpus()	Returns an array containing information about the computer's CPUs
endianness()	Returns the endianness of the CPU
EOL	Returns the end-of-line marker for the current operating system
freemem()	Returns the number of free memory of the system
hostname()	Returns the hostname of the operating system
loadavg()	Returns an array containing the load averages, (1, 5, and 15 minutes)
networkInterfaces()	Returns the network interfaces that has a network address
platform()	Returns information about the operating system's platform
release()	Returns information about the operating system's release
tmpdir()	Returns the operating system's default directory

	for temporary files
totalmem()	Returns the number of total memory of the system
type()	Returns the name of the operating system
uptime()	Returns the uptime of the operating system, in seconds
userInfo()	Returns information about the current user