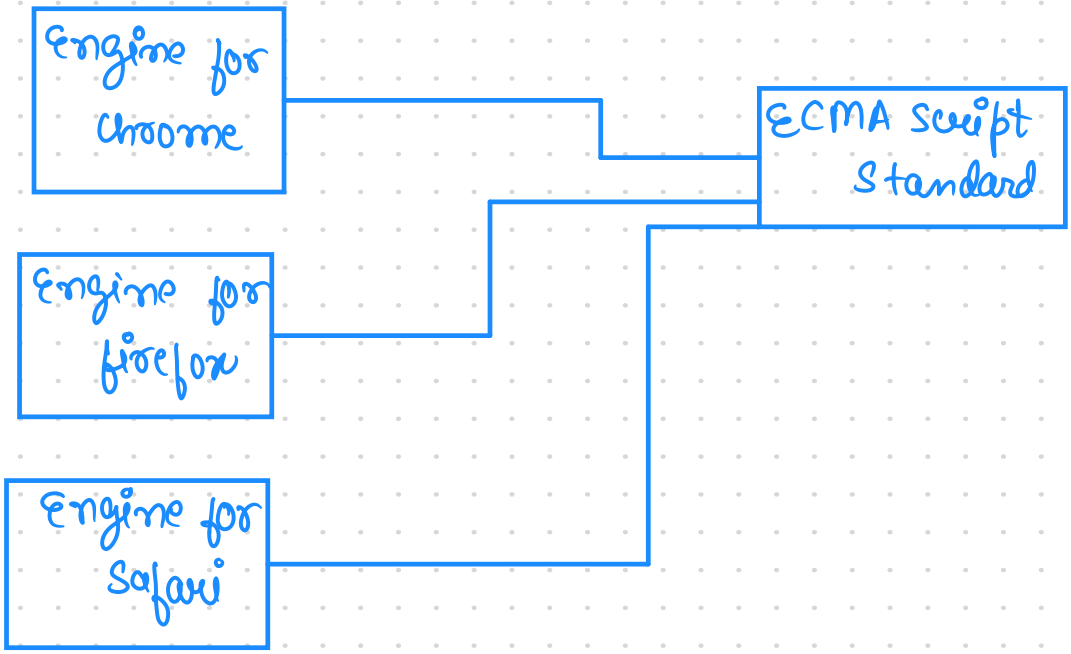
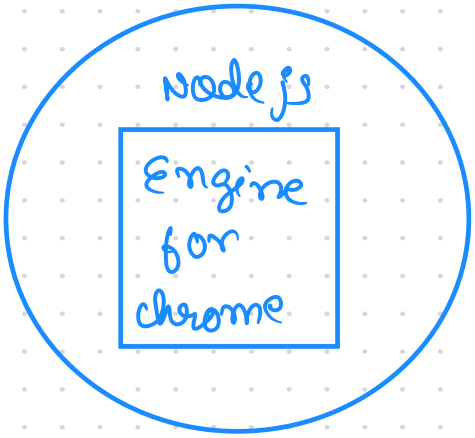
JS Architecture



Node.js

↳ Its a js runtime



JS can run in

Browser
data centers
Flight TV
Mobile devices

→ Single Threaded
↳ Using single core
Single executor
No helpers
↳ Async in nature
↳ Use callbacks

Typical task goes on while other tasks can be done parallerly.

Interpreted → line by line (start)

Loosely typed → no need to define int, var etc.

HTML Stuffs

↳ contain local storage inside browser

F12 → Application → Local Storage

write a backend server that on / route → Prints seem to I/O

Synchronous code

↳ Just calling the function simply and using

↳ executed line by line in order its written each operation waits for previous one to complete before moving to next one.

I/O heavy operations

↳ refers to heavy tasks in a computer program that involves a lot of data transfer btw program and external devices.

↳ waiting for data to be read from sources like disks, networks, databases which is time consuming.

eg

↳ Reading from file

↳ starting a clock

↳ HTTP requests (Hyper-text transfer Protocol req)

fs → file system library

↳ readfile

↳ readfilesync

CPU Bound tasks

Operations that are limited to speed and power of the CPU. require processing power

I/O Bound tasks

Op that are limited by system I/O capabilities, such as disk I/O, network I/O or other data trasfer. Spend time in waiting for I/O operations.

eg → read file

one by one ✗

content switch btw them ✓

↳ Runs Parallely : which is done early then prints.

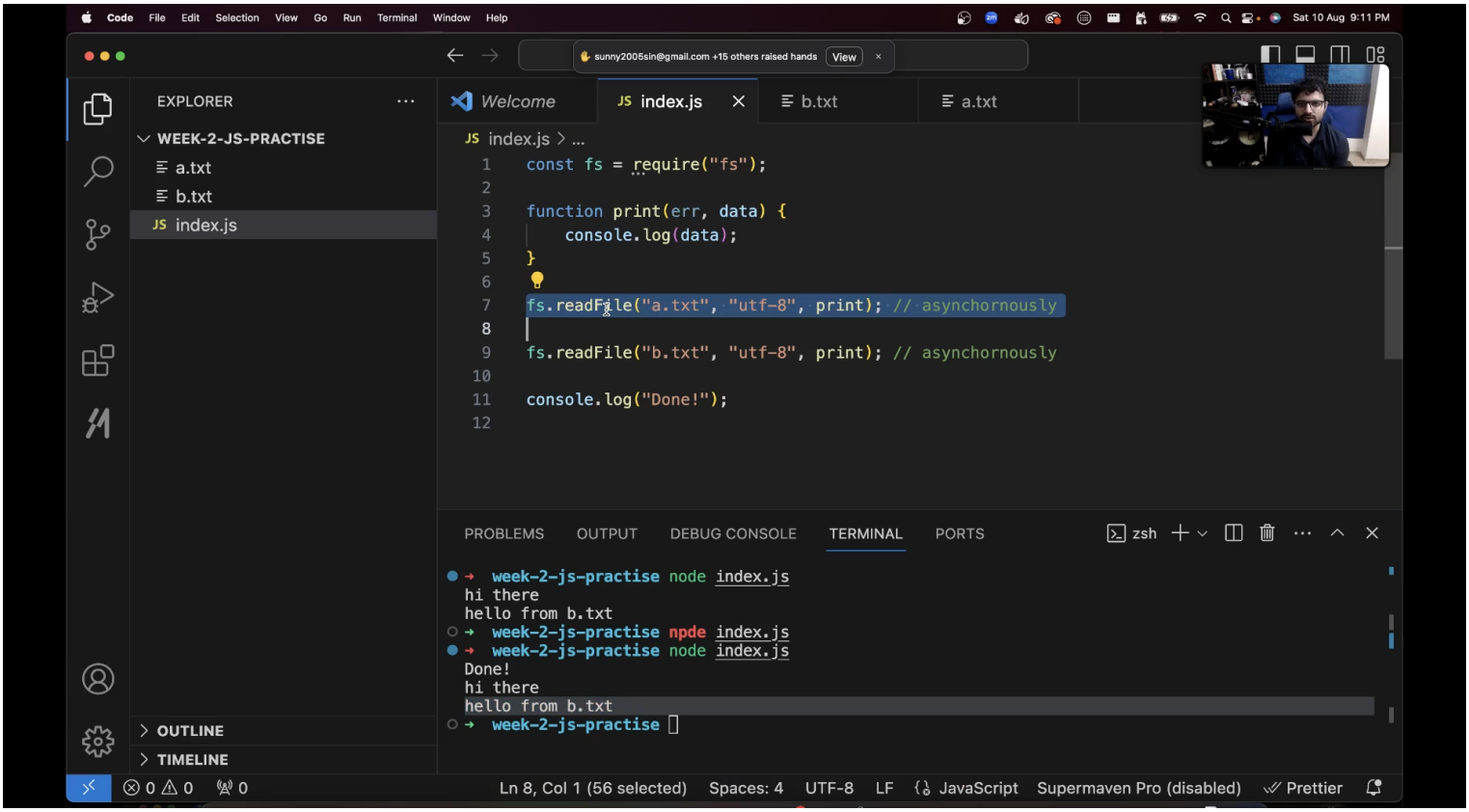
Functional arguments

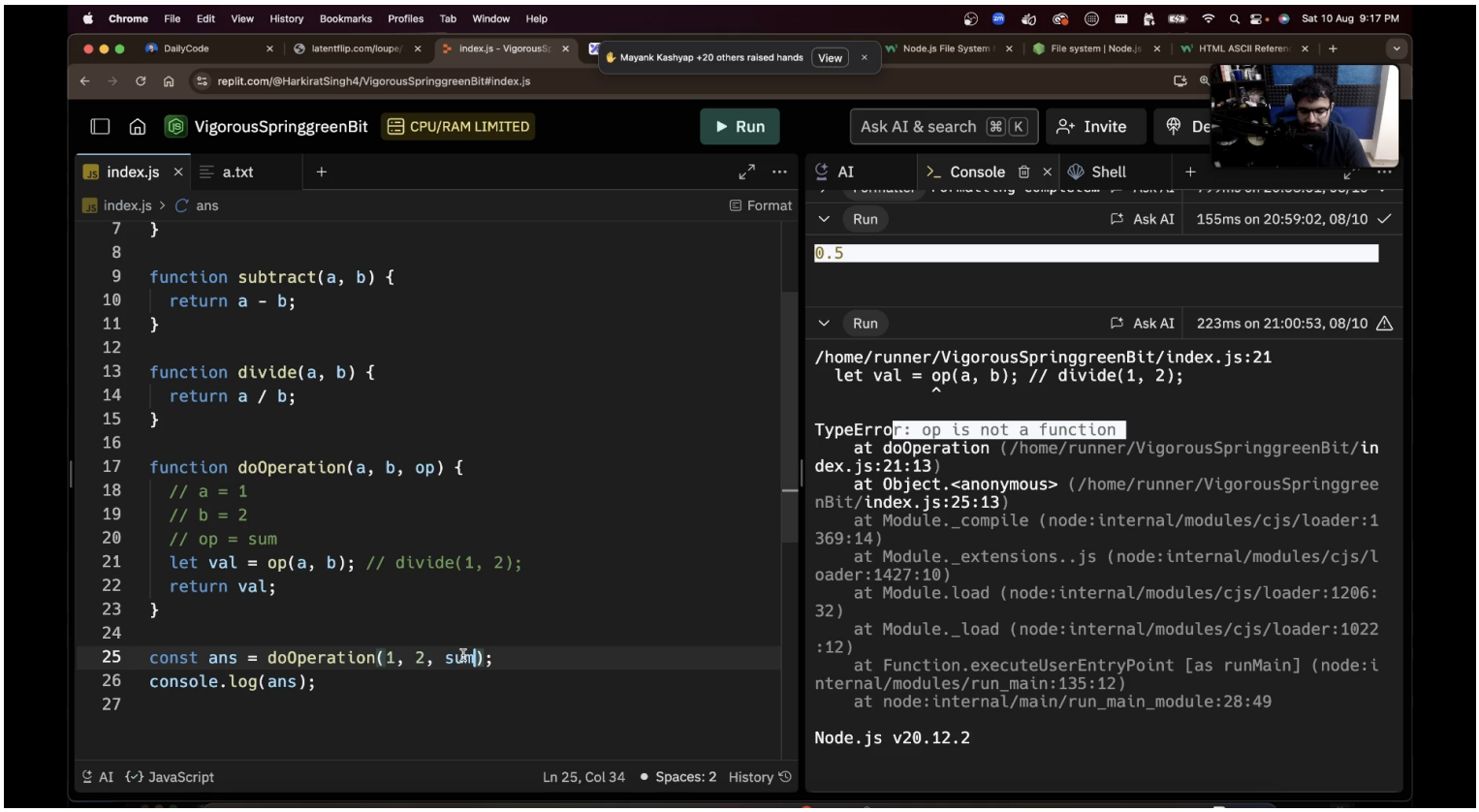
↳ Passing a () to another () as an argument

```
function doOperation(a, b, op){
    return op(a, b)
}
console.log( doOperation (1, 2, sum))
```

Async code, callbacks

↳ Not waiting for finish move to next on





```
JS index.js > print
1   const fs = require("fs");
2
3   function print(err, data) {
4       if (err) {
5           console.log("File not found!")
6       } else {
7           console.log(data);
8       }
9   }
10
11  fs.readFile("aas.txt", "utf-8", print); // asynchornously
12
13  // fs.readFile("b.txt", "utf-8", print); // asynchornously
14
```

adhere = sunna padega

setTimeout
↳ async

readFileSync → will stay

```
function timeout() {
    console.log("Click the button!");
}

console.log("Hi!");

setTimeout(timeout, 1000);

console.log("Welcome to loupe.");

let c = 0;
// 3-4s
for (let i = 0; i<10000000000; i++) {
    c = c + 1;
}

console.log("Expensive operation done");
```

→ Hi
→ welcome to loupe
→ expop
→ click the button

Boom!!

eg
↳ Washing Machine
10mins Beep

Running a treadmile
30minutes without completing you will go ??
NO Right :)

If anything or something happening in Call Stack the code will wait until it is completed

↳ (Web APIs, callback Queue)

WebAPIS → Async ()

↳ washing machine

When the call stack is free the data which was finished from WebAPIs and was sent to callback queue now can be printed. (You can see in loupe)

for reading data use

↳ async code

Classes in JS

Primitive
- number
- String
- Boolean

Complex types
- Objects
- Arrays

Date Class
- `const date = new Date();`
  `console.log(date);`
  - `date.getFullYear();`

Promise class
- Promise class gives you a promise that I will return u something in the future

Two approaches

Callback based approach

Promise based approach

Promise in JS is an object that represents the eventual completion (or failure) of an async operation and its resulting value.

Callback code ⟶ Promises (more Cleaner Code)
- need this due to callback hell
- superior way to write Callback

```
function main(){
}
setTimeout(main, 3000)
```
callback

⟶ `setTimeoutPromisified(3000).then(callback)`

Promise returns object of the Promise class

```
function setTimeoutPromisified(ms){
    const d= new Promise();
    return d;
}
let p= setTimeoutPromisified(5000);
console.log(p);
```

```
function waitFor3S(resolve) { // resolve = main
    setTimeout(resolve, 3000)
}

function main() {
    console.log("main is called")
}

waitFor3S(main);
```

O/P → main is called

```
function waitFor3S(resolve) {
    setTimeout(resolve, 10000)
}

function setTimeoutPromisified() {
    return new Promise(waitFor3S);
}

function main() {
    console.log("main is called")
}

setTimeoutPromisified().then(main); // promise based approach
```

O/P → main is called

Promise takes one () as a input iss () ka jo pehla argument hoga whenever its called usse baad i will call whatever is passed in .then (main)

Promise () → random () → resolve () → data

↳ Yeah complete hoga fir next kaam hoga

```
function random(resolve) { // resolve is also a function
    resolve();
}

let p = new Promise(random); // supposed to return u something eventually



// using the eventual value returned by the promise
function callback() {
    console.log("promise succeded");
}
p.then(callback);
```

O/P → promise succedded

Promise → random → resolve ← resolve

.then ← resolve ← resolve

nae finish

callback → Print hua

Ko callKara

```
function random(resolve) { // resolve is also a function
    setTimeout(resolve, 3000);
}

let p = new Promise(random); // supposed to return u something eventually



// using the eventual value returned by the promise
function callback() {
    console.log("promise succeded");
}
p.then(callback);
```

O/P → after 3sec

Promises

Whatever is passed to first argument of Promise class, usKa jon first argument hai, whenever that called, .then (callback), this will be called

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

function readTheFile(sendTheFinalValueHere) {
    fs.readFile("a.txt", "utf-8", function(err, data) {
        sendTheFinalValueHere(data);
    })
}

function readFile(fileName) {
    // read the file and return its value
    return new Promise(readTheFile);
}

const p = readFile();

function callback(contents) {
    console.log(contents);
}

p.then(callback)
```

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

console.log("-----top of the file-----");

function readTheFile(resolve) {
    console.log("readTheFile called");
    setTimeout(function() {
        console.log("callback based setTimeout
        completed");
        resolve();
    }, 3000);
}

function setTimeoutPromisified(fileName) {
    console.log("setTimeoutPromisified called");
    // read the file and return its value
    return new Promise(readTheFile);
}

const p = setTimeoutPromisified();

function callback() {
    console.log("timer is done");
}

p.then(callback)

console.log("-----end of the file-----")
```

washing machine

```
→  week-2-js-practise node index.js
-----top of the file-----
setTimeoutPromisified called
readTheFile called
-----end of the file-----
callback based setTimeout completed
timer is done
```

```
function setTimeoutPromisified(time) {

    return new Promise(function () {
        setTimeout(resolve, time);
    })
}

function callback() {
    console.log("Time has passed")
}

const p = setTimeoutPromisified(10000)
p.then(callback);
```

Callback based

fs.readfile("a.txt", "utf-8", fn)

Promise based

fs.readfile("a.txt", "utf-8").then(fn)

Npm modules should not be send to other just have package.json and then run

↳ npm install

commander

↳ used to make code more pretty by giving description

↳ node no-ofwords.js -h

↳

↳ shows all the functions used with name and ()

```
const fs = require('fs');
const { Command } = require('commander');
const program = new Command();
program
  .name('counter')
  .description('CLI to do file based tasks')
  .version('0.8.0');
program.command('count') //command name
  .description('Count the number of words in a file')
  .argument('<file>', 'file to count')
  .action((file) => {
    fs.readFile(file, 'utf8', (err, data) => {
      if (err) {
        console.Log(err);
      } else {
        const words = data.split(' ').Length ;
        console.Log(`There are ${words} words in ${file}`);
      }
    });
  });
```

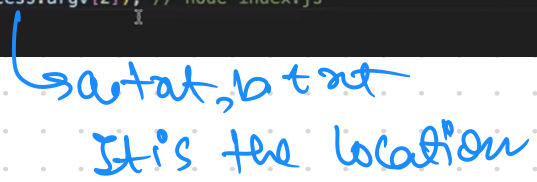
```
  program.command('count_sentences') //command name
  .description('Count the number of lines in a file')
  .argument('<file>', 'file to count')
  .action((file) => {
    fs.readFile(file, 'utf8', (err, data) => {
      if (err) {
        console.Log(err);
      } else {
        const lines = data.split('\n').Length ;
        console.Log(`There are ${lines - 1} lines in ${file}`);
      }
    });
  });
program.parse();
```

↳ node commander.js count a.txt

↳ 2

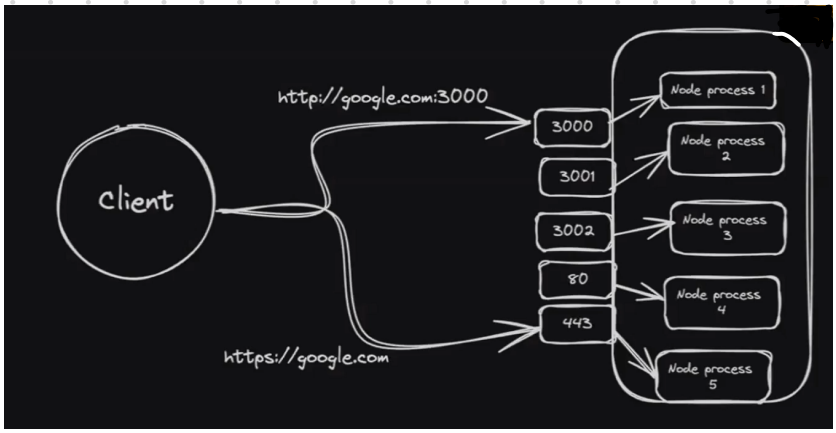
↳ mainly used due to -h capability

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

function main(fileName) {
    fs.readFile(fileName, "utf-8", function(err, data) {
        /// hello world
        let total = 0;
        for (let i = 0; i < data.length; i++) {
           if (data[i] === " ") {
               total++;
           }
        }
        console.log(total + 1 );
    })
}

main(process.argv[2]); // node index.js
```



212.2.242.58:443
↳ Port

Ports
↳ logical endpoints used by protocols to identify specific processes running on a computer or server.
↳ they help direct network traffic to correct application or service on a system.



single VM can run multiple websites

VM: 22.11.44.33

22.11.44.33:3001 → 3001
22.11.44.33:3002 → 3002
22.11.44.33:3003 → 3003

projects.com
app.com
100x.com

Most websites are deployed on 443.

If url starts with Https then port → 443
↳ Secure

HTTP = 80
HTTPS = 443
SSH = 22

Meathods

CRUD
Create → POST
Read → GET
Update → PUT
Delete → DELETE

It's used to specify the type of action that the client wants to perform on a resource on the server.

Response

↳ whatever servers response to the request you sent.

↳ plain text

↳ HTML

↳ JSON Data (Javascript Object Notation Data)

when you scroll in LinkedIn server responses somehow in this way.

```
{ posts: [
    { creator: harkirat,
      description: "Nothing",
      reactions: {
          likes: 200
      }
    }
]
}
```

JSON Data

Status Codes

↳ 3 digit no. to indicate outcome of client's request.

200 → OK

204 No Content

300 → moved to new URI per

304 → not modified using cache version

400 → Bad Req (Invalid syntax)

401 → Unauthorized (admin)

403 → Understood req but refuses to authorize

404 → Not found (Not found in server)

500 → Internal Server error

502 ↳ Bad Gateway

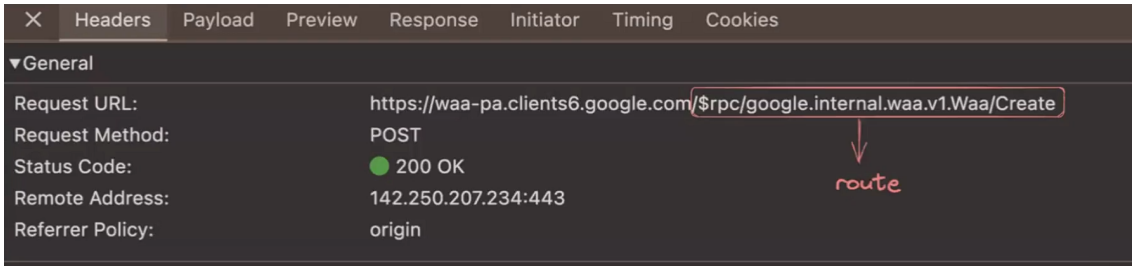
Paytm used to do 200 always body → error

Body

↳ Part of HTTP req that contain the actual data to be sent.

Routes

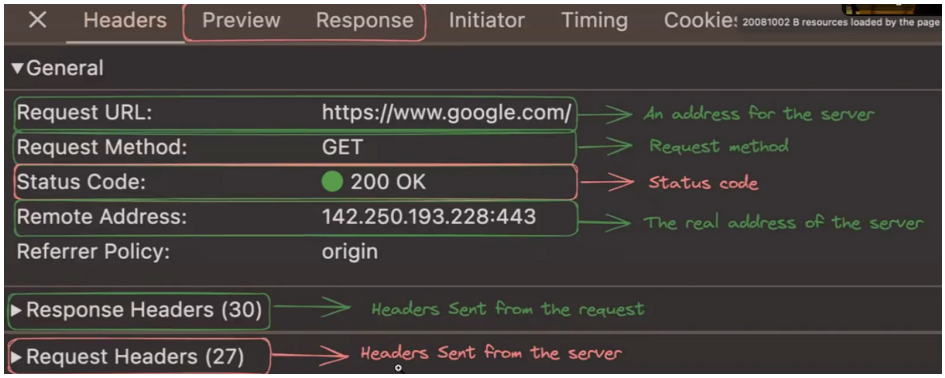
↳ what exactly you want to do



accessing courses

↳ /courses

↳ /calender



http → website sends data in text

can be taken out

https → secures that

Async Await Syntax

```
function setTimeoutPromisified(duration) {
  return new Promise(function (resolve) {
    setTimeout(resolve, duration);
  });
}

// Promise chaining
setTimeoutPromisified(1000).then(function () {
  console.log("hi");
  return setTimeoutPromisified(3000)
}).then(function () {
    console.log("hello");
    return setTimeoutPromisified(5000)
}).then(function () {
    console.log("hi there");
});

console.log("outside the callback hell");
// promise chaining

// const sortedString = str1.split("s").sort().join("").toLowerCase();
// asd => ["a", "s", "d"] => ["a", "d", "s"] => "asd"
```

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function solve() {
    await setTimeoutPromisified(1000);
    console.log("hi");
    await setTimeoutPromisified(3000);
    console.log("hello");
    await setTimeoutPromisified(5000);
    console.log("hi there");
}

solve();
```

Not stuck here

console.log("later");

↑ same thing

Provides a way to write async code that looks and behaves like sync code make it easier to read and maintain.

syntactic sugar on top of promises.

O/P
↳ later
hi
hello
hi there

{looks like sync code but async} (It's a promise only)

To get data dynamically

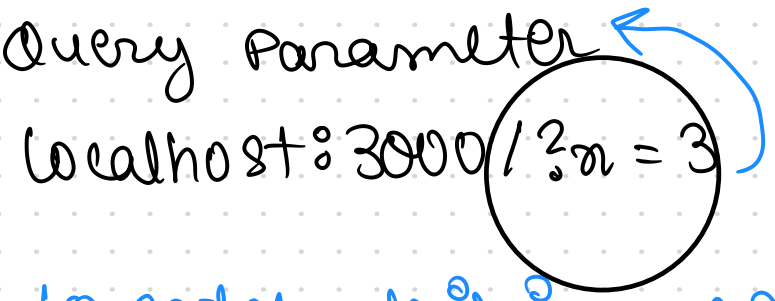
↳ localhost:3000/a/b

↳ get("/add/:a/:b", ...)

Use `const a = req.params.a;`

query → params

Query Parameter



to catch this ?n code use

```
const n = req.query.n;
```

req → request res → response

Middle ware ① It may or may not change the req object

② It will → either stop request right there

↳ forward the req to real route handler.

↳ ticket checker in a park

middleware () are () that have access to the request object, response object and the next middleware () in the application's request response cycle. The next middleware () is commonly denoted by variable named next.

() are

↳ execute any code

↳ make changes to req and response object

↳ end the req-res cycle

↳ call the middleware () in the stack.

express app is essentially a series of middleware calls.

```
const express = require("express");
const app = express();
let requestCount = 0;
function requestIncreaser(req,res,next){
    requestCount+=1;
    console.log("Total number of req " + requestCount);
    next();
}
function realSumHandler(req,res){
    const a = req.query.a;
    const b = req.query.b;
    res.json({
        answer: parseInt(a)+parseInt(b)
    })
}
app.get("/sum",requestIncreaser,realSumHandler);
app.listen(3000);
```

app.use(requestIncreaser);
↳ will be exec in every () automatically

Common used Middlewares
↳ express.json
↳ built in middleware() in express.js used to parse incoming req bodies that are formatted as JSON.
↳ Used in PUT, POST Req
To send JSON data you need to Parse the data using express or body Parser.
app.use(express.json());
↳ CORS (Cross origin resource sharing)
↳ cors origin requests by default blo

It simply means that if you are on facebook.com under the hood you are not allow to access api of any other company such as ndtc.
But by default its blocked, to use it we can do it.

It a security feature that implemented by web browsers to control how resources on a web server can request on another domain.

Headers

↳ key value pair between a client and a server in HTTP req or res. They convey metadata about req, res such as → content type, auth info.

Common headers

↳ Authorization (Info) → no in address ↳ but through headers

↳ Content Type

↳ Referer → url req coming from

↳ Cookies

↳ Request headers

```
async function getRecentPost() {
  console.log("before sending request");
  const response = await fetch("https://google.com")
  const data = await response.json();
  console.log(data);
  console.log("request has been processed")
  document.getElementById("posts").innerHTML = data.body
}

async function getRecentPost() {
  console.log("before sending request");
  await fetch("https://google.com")
    .then(function(response) {
        response.json().then(function(data) {

          console.log(data);
           console.log("request has been processed")
          document.getElementById("posts").innerHTML = data.body
        })
    })

}
```

1 2

} same as promise only

→ to convert data in json (Promise)

fetch is used under the hood to get more data like in utube as we scroll new things get added below.

Axios → used to make syntax easier

HTTP Client

convert to json by default

need to import module

AJAX → Async JSON and XML

Arrow ()

```
function sum(a, b) {
  return a + b;
}

const sum = (a, b) => {
  return a + b;
}

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

})

app.get("/", funciton(req, res) {

})
```

Map()

```
const input = [1, 2, 3, 4, 5];

/// solution
// const newArray = [];
// for (let i = 0; i < input.length; i++) {
//   newArray.push(input[i] * 3);
// }
// console.log(newArray);
/// other solution
function transform(i) {
  return i * 2;
}
const ans = input.map(transform);
console.log(ans);
```

Filter()

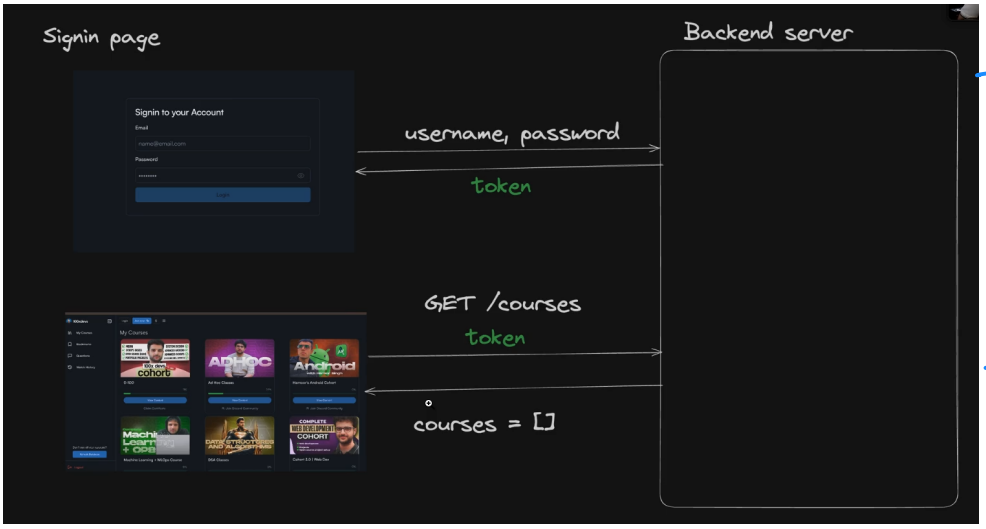
```
const arr = [1, 2, 3, 4, 5];

// ans
function filterLogic(n) {
  if (n % 2 == 0) {
    return true;
  } else {
    return false;
  }
}

const ans = arr.filter(filterLogic);
console.log(ans);
```

Axios vs fetch

Authentication

↳ Process of sign in / sign out making sure routes are protected and users can get back their own data and not the data from diff users.

Things
- ↳ Auth basics
  - ↳ JWT (JSON web tokens)
  - ↳ Authorization header
  - ↳ Creating own auth middleware
  - ↳ local storage



→ The user comes to website
→ user sends request /signin with their username and password
→ The user gets back a token.
→ In every subsequent req, the user sends the token to identify it self to the backend.

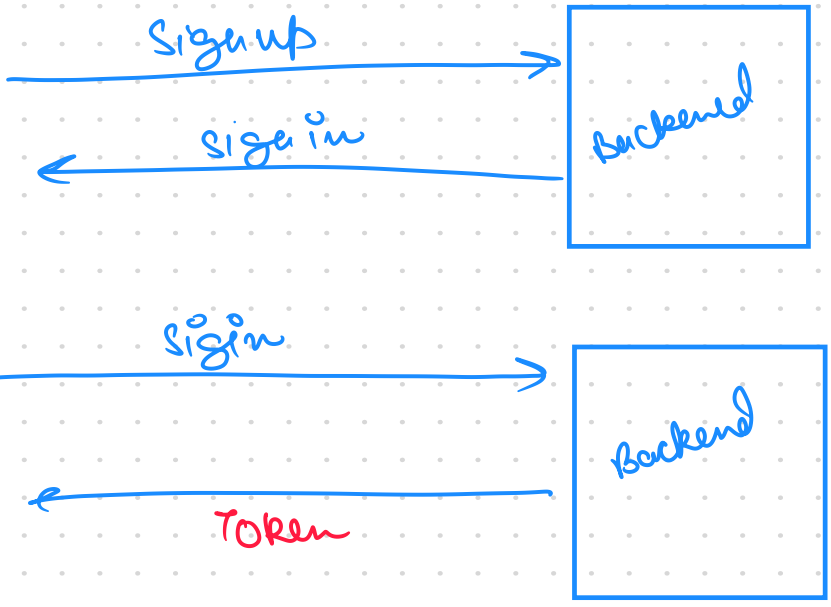
express.json help to parse body of json file

```
function generateToken() {
    let options = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
    'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A',
    'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
    'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2', '3', '4',
    '5', '6', '7', '8', '9'];

    let token = "";
    for (let i = 0; i < 32; i++) {
        token = token + options[Math.floor(Math.random() * options.length)];
        //Math.random() * options.length => gives a number btw 0-42
        //floor converts 11.3 -> 11
    }
    return token;
}
```

```
for(let i=0;i<users.length;i++){
    if(users[i].username == username && users[i].password == password){
        foundUser = users[i]
    }
}
const user = users.find(function(u){
    if(u.username == username && u.password == password){
        return true;
    }
    else{
        return false;
    }
})
```

Same same but different

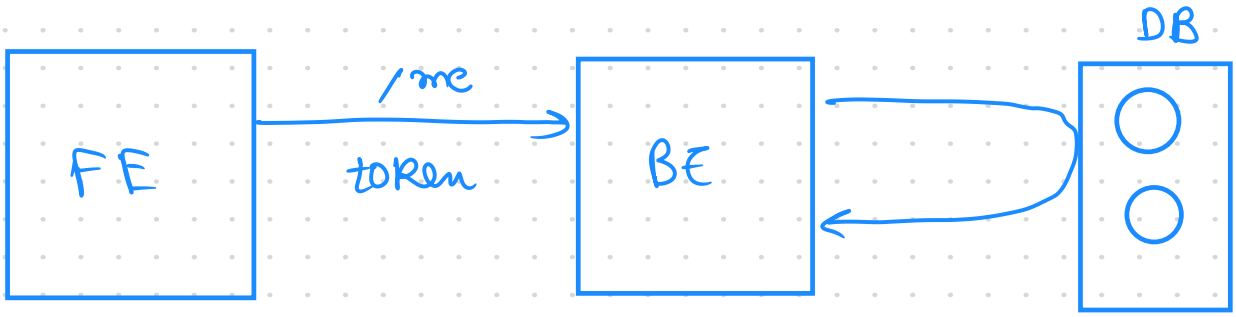
What we have done



Token is set in a type of metadata in headers
↳ for getting my courses

```
const express = require("express");
const app = express();
app.use(express.json());

const users=[];
//will be stored like
//[
//{username: "harkirat",password:"iloveneha",token: "afjaslkjdfsalkjas"}
//]
function generateToken() {···
}
app.post("/signup",function(req,res){···
})
app.post("/signin",function(req,res){···
})
app.get("/me",function(req,res){···
})

app.listen(3000);
```

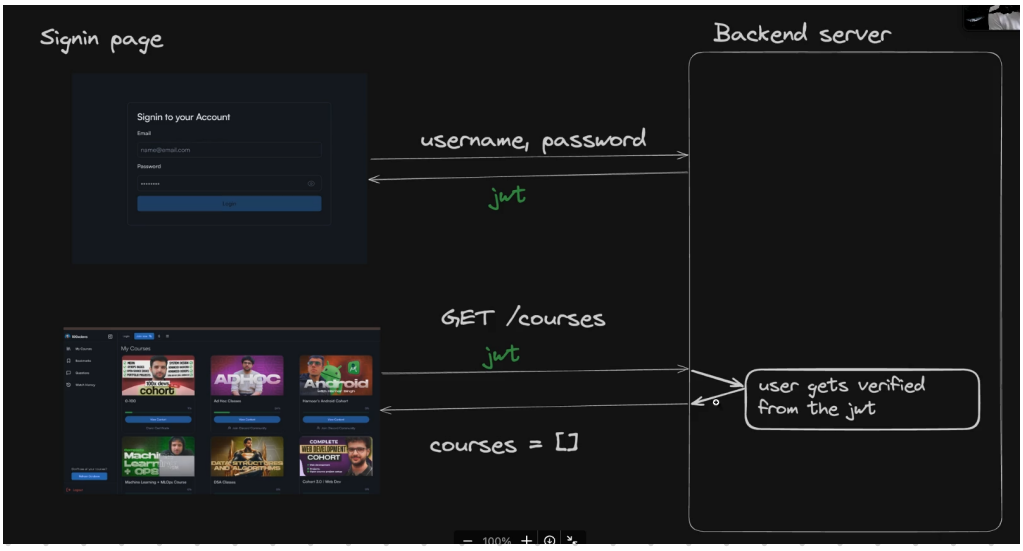
Problem with tokens is:



To many req are sent to Databases.
to get rid of that we use JWTs.

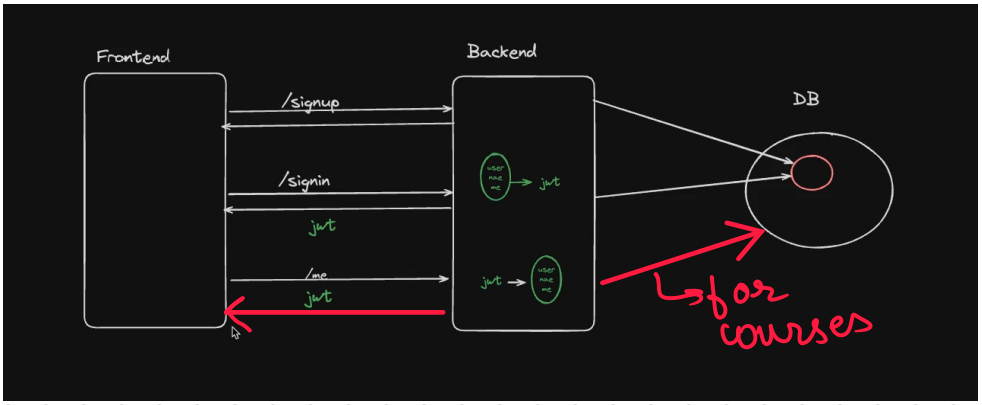
JWTs → Encrypted token which contain ID, pass which will help in reducing calls to DB.

↳ JWTs are stateless: Contains all the info needed to authenticate the request, so the server doesn't store the session data. All data stored in token itself.



- Don't want to overwhelm the DB.
- Prevents 1 round trip from the BE server and DB for any user req

This is the thing napenning under the hood



→ If anyone has jwt mean they have logined earlier so authenticated.

JWT flaw is it can be decoded on the website by anyone but verified by me

jwt.decode ⟶ raman ko bss to encode karega wahi correct mana jayega

jwt.verify ⟶ It should be signed by our specific Jwt secret not any random secret.

__dirname = Current direc

Database → NO SQL Databases

Earlier using NO Memory DB

transient → if one goes down other works

Server should be stateless

DB must be replicated enough.

Full stack app

↳ you persist data in DB

→ Schemaless

Mongo DB and NO SQL Databases

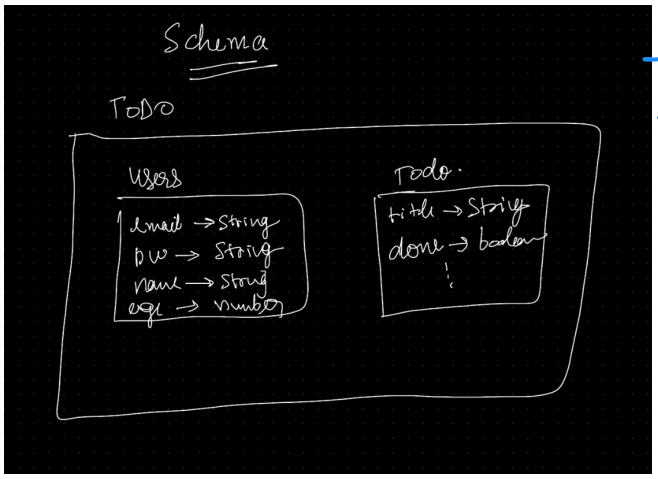
↳ Designed to handle variety of data models and workloads that may or may not fit neatly into tabular schema of relational db.

→ Nested structure

↳ lectures in weeks

Properties

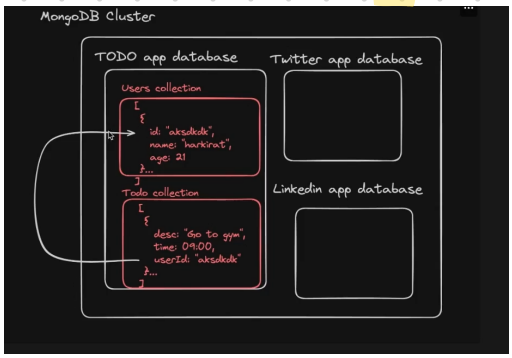
↳ Schema flexibility → How does the end data looks alike



→ Schema, SQL DB

→ But Mongo allows flexible schema, better for unstructured data

↳ Scalability: ↑ no. of Instances, adding serv



Making different db file for more clarity.
To use the Schema we define that

```
const UserModel = mongoose.model('users', User);
```

To use the file we export it also

```
module.exports = {
    UserModel: UserModel
};
```

Model is sending request to a server very far away which means async ().
↳ Use await

We are making async. bcos while sending data it may fail to check that we need to wait which means await need to be used.
↳ Before letting user know you are logged in

For using req.body → use
↳ app.use(express.json());

All db calls must be awaited

await
↳ wait for the process to happen before proceeding

mongoose → requires Schema
mongo DB → NO Schema
ObjectId default provided to the user.

Hashing is not encryption

① first we learnt jsonwebtoken

② Hashing means converting password to gibberish and storing that in db so that anytime if get leaked noone can not get password.

↳ Then to check again if the password is correct just run hashing algo again and check correct or not.

Problem

↳ If two person has same passwords then the other person can logen. to fix that we use

Salting

↳ Take the input add some random gibberish and then you hash it.

↳ Safe :)

In DB
↳ Store → hash + random gibberish

Bcrypt

↳ Incorporates a salt and is designed to be computationally exp, making brute force attack more difficult.

Rainbow Table / Pre computed table
↳ Passwords with its hashings

ZOD

input validation → user can send anything so we need to check that

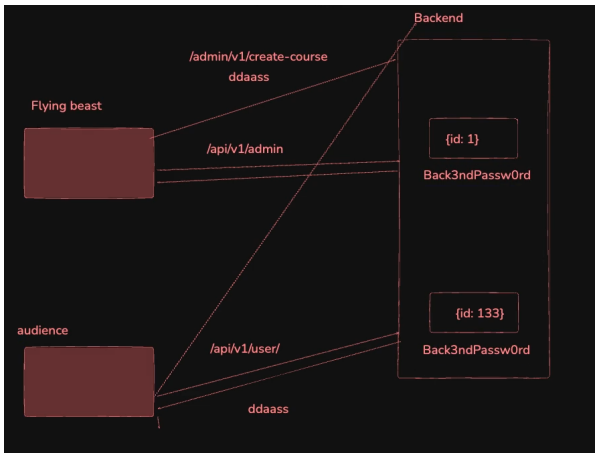
Schema validation ✓

describe schema to ZOD

Express Routing

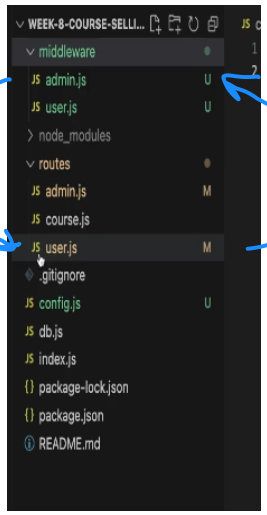
/user if mentioned in index.js no need to mention in the other code

route name can be handled in index.js



Why we keep different just pass points for admin and user?

↳ To avoid users to hit admin user points



some config files

some variables

Dont get into circular dependency

↳ means get a data from different files and giving back data to that file also

- .env → to save creds
- .env.example → share the schema of env file
- .gitignore → telling git to ignore certain files

Dotenv → No dependency