Express JS

mongoDB

BACKEND DEVELOPEMNT

HANDWRITTEN NOTES

PREPARED BY

ARNAB SENAPATI

LEARN & GROW

INDEX

INDEX

Day-1

Backend Development

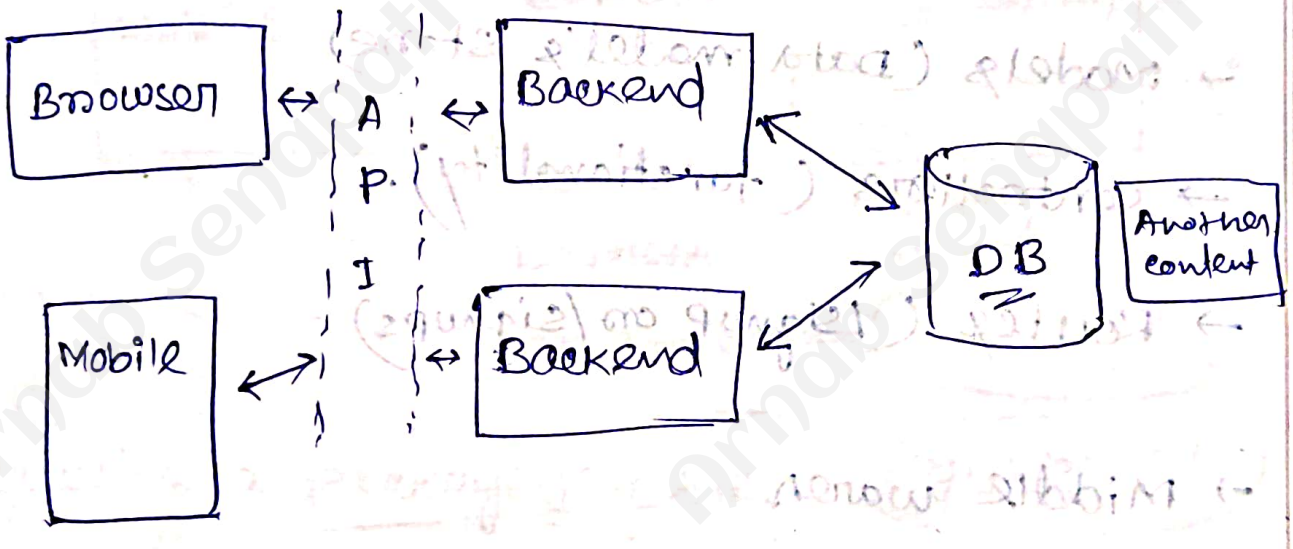
2 major components

A Programming

Java, JS, PHP, golang
C++
⇒ one framework

A Database

Mongo, MySQL, Postgres, sqlite
⇓
ORM: ODM



⇒ A JavaScript based Backend

Data File Third Party (API)

→ A JS Runtime : Nodejs / Deno / Bun

Package.json → src

- → DB
- → Models
- → Controllers.
- → Routes
- → Middlewares
- → Utils

.env (readme, git, lint, Prettier etc).

index → DB connects

→ More (depends

APP → config cookie, unicode,

constants → enums, DB-name.

Files

index → DB connects (entry point)

APP → config cookie, urlencode (configuration)

constants → enums, DB name

Directory structure

→ DB (Data Base) → Actual code connect in DB.

→ Models (Data model's store)

→ controllers (functionality).

→ Routes (/signup or /signups)

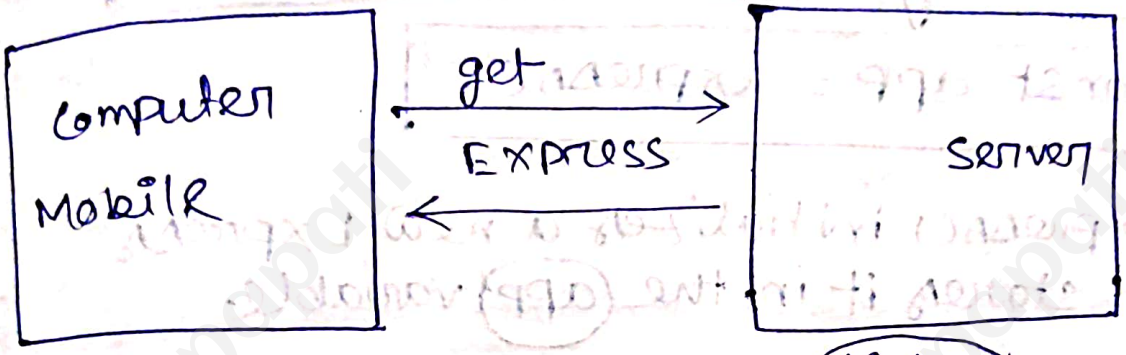
→ Middle wares

→ Utils (utility e.g. mail karna) (file upload).

→ None (depends)

How to deploy backend code in production

1st :- i) Down node JS and intall
ii) open terminal check nodeJs version
`node -v`
iii) check npm version `nmp -v`



Listen

Express

mongoose

/: Home route

/login: login setup

2nd

creating a package.json file.

i) create folder, then open in vs code.
ii) open terminal, (npm init) or (npm init -y)

3rd

i) go to ExpressJs website;
ii) Express :- Fast, unopinionated minimalist web framework for (node.JS)
iii) copy path `$npm install express --save`

next

create a server ... importing Express

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

- this line imports the Express.js framework, which is required to create a server.

2. Creating an Express-Application:

```
const app = express()
```

- express() initializes a new Express and stores it in the (app) variable.

3. Defining the server port:

```
const port = 4000
```

- sets the port number to 4000, meaning the server will run on http://localhost:4000/

4. Defining Routes:

```
app.get('/', (req, res) => {
    res.send('Hello World')})
```

- this defines a GET request handler for the root (/) URL.
- when a user visit http://localhost:4000/ the server responds with "Hello world"

5. Starting the server

```
app.listen(port, () => {
    console.log('Example app listening
        on port ${port}')
})
```

- app.listen(port, callback) start the server on the specified port (4000).
- when the server starts, it logs:

```
Example app listening on port 4000
```

=

Next **dotenv**:

what? ⇒ (dotenv) is a package that allows you to load environment variables from a `.env` file into your Node.js application.

why use?

i) keeps sensitive data (API keys, passwords) out of your source code.

ii) Easily switch between different environments (development, testing, production)

How install dotenv?

1) go to dotenv website
2) copy path → npm i dotenv
3) go to vs code terminal, past & install

How use?

1) require('dotenv').config()
2) where use, there write this → `process.env.PORT`

How to Deploy a Backend Application to production

1st : i) go to github, create a repository

ii) push vs code to github
1) git init
2) git add .
3) git commit -m "first commit"
4) git branch -M main
5) copy reposetory URI & past
6) git push -u origin main.

2nd Deploy free

1) Render
2) Railway
3) Vercel
4) Netlify

How to Connect frontend and Backend in Javascript

1st Open a folder → create two folder
i) Backend ii) frontend

2nd : open terminal → npm init

3rd : npm i express

4th : touch server.js

next
add package.json →
① "type": "module"
② "start": "node server.js"

next :- add server.js

① create 5 joker

```
app.get('/joker', (req, res) => {
  const joker = [
    { id:1,
      title: 'A joke',
      content: 'This is a joke'
    }
  ]
```

like 5 joke

Next go to Frontend

create a react app

~~two~~ next → acces joke backent to frontend.

~~Step~~ How ?

Step-1 : const [jokes, setJokes] = useState()

Step-2 :
```
{ joke.map((joke, index) => {
    <div key={joke.id}>
      <h3>{joke.title}</h3>
      <p>{joke.content}</p>
    </div>
```

Step-3 : install → npm i axios.

- **next** :- it's when run then coming a error.

http://localhost:5173 has been blocked by **CORS** policy

⊛ what is CORS ?
⇒ is a security feature implemented in web browsers to restrict how resources on a web server can be requested from another domain.

Data Modelling for backend with mongoose

step-1 :

1st : go to stackBlitz, create account

2nd :- ~~create~~ go to new terminal, install
→ npm i mongoose

3rd : create models folder., next go to models folder, create a new folder todos (in models folder).

4th : create a file (user.models.js) in todos folder. same create next two file → (todo.models.js) & (sub_todo.models.js)

step-2

1st : import mongoose :

how ?
→ import mongoose from "mongoose" (in sub_todo)

2nd : mongoose help, & create schema
how?
→ new mongoose.Schema({})

3rd : export

how?
→ export const User = mongoose.model()

2.0

4th :

⊛ Important : when create model then its convert into (plural)

If :

```
export const User = mongoose.model
    ("User", userSchema)
```

Users

now go to documentation

step 1 :

```
{ username: String,
  email: String
  isActive: Boolean
}
```

or

```
username: {
  type: String,
  required: true,
  unique: true,
}
```

mongoose timestamps :-

1. createdAt :
2. updatedAt

Day-5

Ecommerce and Hospital management

Data modelling :

Ecommerce

Step-1 :-

1st : create a **ecommerce** folder and create four js file in ecommerce folder.

2nd : i) category.model.js
ii) order.model.js
iii) product.model.js
iv) user.models.js.

user.models.js

```
import mongoose from,
"mongoose"
const userSchema = new
mongoose.Schema({
  username:{
  type: String,
  required: true,
  unique: true,
  lowercase: true
  },
so on email & password
},{timestamps: true})
export const User = mongoose
model("User", UserSchema)
```

* same create
Product.models.js
like :- description,
name, productImage,
price, stock, category.

```
category :{
type: mongoose.Schema.
   Types.ObjectID,
required: true,
ref : "Category",
```

category.model.js

```
import mongoose from "mongoose"
const categorySchema = new mongoose.
                              schema({
  name: {
    type: String,
    required: true
  },
}, {timestamps: true})
export const category = mongoose.model
          ("category", categorySchema)
```

order.model.js

```
import mongrose from "mongoose"
```

create submodel
productId
quantity

```
orderprice
customer
orderitem [type: [ submodel]]
address
status
```

Hospital Management

step-1

1st :- creat a (hospital- managment) folder and create a four.js file in hospital-managment folder.

2nd : i) doctor.models.js
ii) hospital.models.js
iii) medical-record.models.js
iv) patient.models.js

step-2 :-

patient.models.js

1st :- import mongoose

2nd : create
i) name
ii) digorsewidth
iii) address
iv) age
v) bloodGroup
vi) gender
vii) admitted In

3rd : export mongoose.

doctor.models.js

1st :- import mongoose

2nd : create
i) name
ii) salary
iii) qualification
iv) experience in year
v) works In Hospitals:

3rd : export mongoose.

hospital.models.js

1st :- import mongoose.

2nd :- create
i) name
ii) addressLine1
ii) addressLine2
iii) city
iv) pincode
v) specialized

3rd :- expond mongoose.

Day-6

How to setup a professional backend project

step-1 :-

1st :- create a folder, then open it vs code.

2nd :- open terminal, then write command
i) npm init

3rd :- create a (readme.md) file.

4th :- push all file in github
i) git init
ii) git add .
iii) git commit -m "add initial files for backend"
iv) git branch -M main
v) git remote add origin repository name.git
vi) git push -u origin.

5th :- create a (.gitkeep) file ion (.gitignore)

6th :- creat (.env) & (.env.sample) file.

7th :- create a (src) folder
next create 3 file in (src)
i) app.js
ii) constants.js
iii) index.js

Step-2 :-

1st :- install nodemon

2nd :- open terminal `npm i -D nodemon`

3rd :- when write (index.js) then reload automatic.

4th :- go to (package.json) file and chang (test) command to (dev) and write

→ `"dev": "nodemon src/index.js"`

Step-3 :-

1st :- agin push in github

2nd :- open terminal
- i) git add .
- ii) git commit -m "setup project files - part 1"
- iii) git push

3rd :- go to src folder and open terminal open src folder
- i) mkdir controllers db middlewares models routes utils

Step-4 :- install prettier

1st :- open terminal

2nd :- enter this command

```
npm i prettier
```

why install prettier

i) Automatically formats code for consistancy and readability.

ii) Helps catch syntax error early by enforcing coding standards.

iii) ensure uniform code style across a team, avoiding merge conflicts.

Step-5 :- create a (.prettierrc) file

1st: open file and write

```
{
  "singleQuote": false,
  "bracketSpacing": true,
  "tabWidth": 2,
  "trailingComma": "es5",
  "semi": true
}
```

create a (.prettierignore) file

write => /.vscode, /node_modules, /dist, *.env, .env, .env.*

P-8

DAY-7

HOW to connect database in MERN with debugging

* we use MongoDB

step-1 :- create MongoDB free account

step2 :- create organization if not.

step-3 :- create cluster

1. cluster name
2. Provider choose (aws)
3. choose Region (Hyderabad)
4. create Deployment.

step-4 :- go to SECURITY

1. Database Access (create user)
   - i) Authentication, choose password
   - ii) Enter user name (no special charecter)
   - iii) Enter password (no special charecter)
   - iv) use Description (optional)
   - v) Add user.

step-5 :- go to Network Access.

- i) Add IP Address
- ii) Access List Entry (0.0.0.0/0)
- iii) comment (optional)
- iv) confirm.

step-6 :- go to Data Base

① click cluster.
② see cluster created
③ click connect.
④ Access your data through tools → choose compass.
⑤ use uri → choose 2nd option → copy connection string.

step-7 : go to VS code

① click .env & .env.sample
② past copy string

like :→ mongodb+srv://username:<db_password>@cluster0.whusm.mongodb.net/

=

Next, install → ① npm dotenv

② mongoose

③ Express.

step-1 :- open terminal

write → npm i mongoose express dotenv

How Approach ?

how connect data from database

1st Approach :

① we execute index file. so we all code put in index file. when index file load, which function we write data base code, then it's execute.

2nd Approach :

we make DB folder, and create a index.js then import, then execute this file.

jo connect krai usme likhu.

* we need three package
① dotenv ② mongoose ③ Express.

How connect mongoose ?

* mainly we need try & catch
* database is always another continent.
* async await

1st approach :

① import mongoose

import mongoose from "mongoose"

② create function

```
function connectDB(){}
connectDB()
```

on

we ~~write~~ know in JS → () → it's a function
now it's imediatly executed ()

③ when connect from Data Base
then use try & catch

```
(async () => {
    try {
        await mongoose.connect('${
        {process.env.MONGODB-URL}
        ${DB-NAME}')
    }
    catch (error) {
        console.log error("ERROR:", error)
        throw err
    }
})()
```

④ import DB_NAME

⑤ import express.
↳ const app = express() → create app

⑥ when connect DataBase then see listeners

⑥ app.on("error", (error) => {
    console.log("Err:", error);
    throw error
})

⑦ app.listen(process.env.PORT, () => {
    console.log(`APP is listening
    ${process.env.PORT}`);
})

→

2nd Approach

① import mongoose

② import DB_NAME

③ create function
↳ const connectDB = async() => {
    try {

    }
    catch(error){
        console.log("MONGODB connection error", error);
        process.exit(1);
    }

④

```
try {
    const connectionInstance = await
    mongoose.connect (`${process.env.
    MONGODB_URL}/${DB_NAME}`)
    console.log ('\n MongoDB connected !!
    DB HOST : ${connectionInstance.
     connection.host}`);
    }
```

⑤ next export

export default connectDB

⑥ go to src (index) file.

connectDB() → import & export.

⑦ use (dotenv) it's a environment → As early as possible in your application, import and configure dotenv:

→ require ('dotenv').config()

Day-8

Custom api response and error handling chai aur backend

Step-1 :-

1st : create a app.js then import express and export express.

2nd : when connect DB then ~~then~~ i) .then & .catch

```
.then (() => {
    app.listen (process.env.port || 8000, () => {
    console.log (`server is starting at
                port: ${process.env.PORT}`);
    })
})
.catch ((err => {
    console.log ("Mongo DB connection
                failed !!!", err)
})
```

3rd :- Next create Express.

① go to Express website.
② choose latest API (5.x)
③ next go to npm website
④ search (cookie-parser)
⑤ next search (cors)

4th :- open terminal and write
npm i cookie-parser cors

5th :- Import in (app.js) cors & cookie-parser.

6th :- next how use cors?

⇒ ① app.use(cors({
origin: process.env.CORS_ORIGIN
}))

② next ~~use~~ define (CORS_ORIGIN) in .env file → CORS_ORIGIN = *
and same define (env.sample)

~~Step-2~~ Step-2 :-

1st :- set middleware or req from body.

→ app.use(express.json({limit: "16kb"}))

next use a configration :-

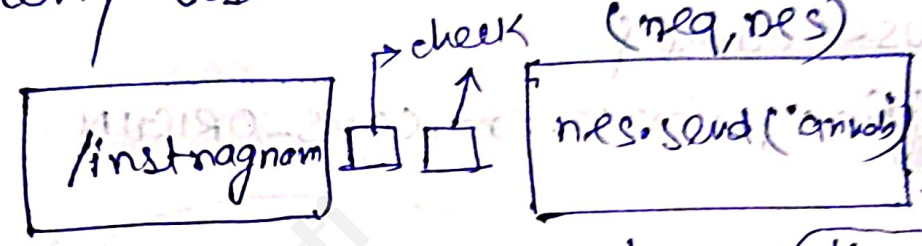
we see a google some link ⊕ use show result.

→ app.use(express.urlencoded(
{extended: true, limit: "16kb"}))

2nd :- next use express configuration when store pdf, doc document

3rd :- what use cookie?
→ when my browser, from user's cookie access or set.

⊛ why use middleware?



⊛ ⇒ when I request from (/instragram) my id then not show bez i not login, show in middle checking process happen it's call middleware.

(req, res) → (err, req, res, next)

Step-3 :-

1st : create a (asyncHandler.js) in utils folder, asyncHandler.js → here create a method then it's export.

1st method

```
const asyncHandler = (fn) => async (req, res, next)
=> {
    try { await fn(req, res, next) }
    catch (error) {
        res.status(error.code || 500).json({
            success : false,
            message: error.message
        })
    }
}
```

2nd method :

```
const asyncHandler = (requestHandler) =>
{ (req, res, next) => {
    promise.resolve(requestHandler
(req, res, next)).catch((err) => next(err))
}}.
```

- asyncHandler is a higher-order function meaning it takes another function ~~request~~ → (requestHandler) as an arguments and returns a new function.
- (req, res, next) => { } → this is an Express middleware function that taken request(req), response(res), and next(next) as arguments.

Handling the Async Function :

~~promise.resolve~~

promise.resolve (requestHandler(req, res, next))
.catch((err) => next(err));

- if (requestHandler) is an async function, it will return a promise. → is executed and wrapped in Promise.resolve().
- if the promise rejects .catch(err) => next(err)) will pass the error the next(), which allows Express to handle it properly.

Step-3

① go to google and search (nodejs) Errors

② creat a file (ApiError.js) in utils

=> (ApiError) class is meant to be a custom error Handler for an API. ~~However, then~~

why use?

1. standarlized API Errors
2. Easier debugging (custom stack Error)
3. custom message for different HTTP errors.

③ next create a file (Api Response.js) in utils.

=> the (ApiResponse.js) class is a standardized response handler for an API.

constructor - parameters

Parameter	Purpose
statuscode	The HTTP status code (eg. 200, 400, 500
data	the actual data to be in the response (e.g. fetched records)
message	A human-readable message (default: "success")
success	A boolean indicating success (true if statuscode < 400, else false).

Breakdown of Each property

(1) this.statusCode = statusCode;

⇒ Stores the HTTP status code (e.g., 200 OK, 404 not Found, 500 internal server Error).

(2) this.data = data;

⇒ stores the actual response data (it could be an object, array, string etc).

(3) this.message = message

⇒ A human-readable response message (defaults to "success" if not provided).

(4) this.success = statusCode < 400

⇒ Automatically determines if the response indicates success.

- if statusCode is below 400 (like 200 or 201), success = true.
- if statusCode is 400 or above (like 404, 500), success = false.

why use?

(1) Consistent API response.
(2) Easier debugging.
(3) Automatically determines success/failure.
(4) Reduces repetitive code in controllers.

Day-9 :

User and video model with hooks and JWT :

Step-1 :

1st :- create two file (user.model.js) & (video.model.js) in [models] folder.

Step-2 :

1st : (user.model.js) → import mongoose and create → i) username, ii) email iii) fullname, iv) avatar v) coverimage vi) watchHistory vii) password viii) refreshToken.

2nd : (video.model.js) → import mongoose. and create → i) vidoFile ii) thumbnail iii) title iv) description v) duration vi) views vii) isPublished viii) owner

(next) → export mongoose.

3rd : install (mongoose-aggregate) package →

go to vs code → open terminal →

> npm i mongoose-aggregate-paginate-v2

4th :- import mongoose-aggregate in (video.model.js) file

(and) → export this.

what is?

⇒ A page based custom aggregate pagination library for Mongoose with customizable labels.

usage :- i) Adding the plugin to a schema:

when use :- i) when need pagination with aggregation queries.

ii) when dealing with large datasets.

iii) when using MongoDB aggregation operators like $group, $lookup, $match.

step-3 :-

1st :- go to ~~bcrypt~~ npm website → search bcrypt

what is bcrypt? ⇒ is a password-hashing function that is widely used for securely storing password.

install :- copy path → `npm i bcrypt`

step-4 :-

1st :- ~~go to~~ go to npm → search : jsonwebtoken

what? : is a compact, self-contained format used for securely transmitting information between parties.

install :- copy path → npm i jsonwebtoken

2nd :- direct encript not possible. so helps mongoose hooks

* we use (pre) hook,

how work pre hook?

⇒ when user save data, first of all change user password to encript.

```
userSchema.pre("save", asyne function (next) {
  if (!this.isModified("password")) return next();
```

↳ it use for, who not change password then no save encript.

```
  this.password = bcrypt.hash(this.password, 10)
  next()
})
```

↳ when anyone save or change password then password save on encript format.

* I save encript password but user save (1,2,3,4) like password, how match?

⇒

```
userSchema.methods.isPassword(connect =
asyne function (password){
  return await bcrypt.compare(password,
  this.password) };
```

* JWT → is a Beaer token, jske pas Beaer token hai uske me Bear. Bhej dunga.

day-10

How to upload file in backend

multer :-

step-1 :- go to cloudinary.com website and sign up & create account.

step-2 :- ① go to getting started → choose an SDK → Node.js → copy npm path

② go to multer-npm → copy path → npm i multer

③ open vs code terminal & past & and install.

multer :- used for uploading files.

→ multer is a Node.js middleware for handling multipart/form-data primarily used for uploading files.

step-3 : create a file cloudinary.js in utils folder.

① import v2 cloudinary

② import fs from "fs"

what is fs :- in Node.js fs (short for file system) is a built-in module that allows you to work with the file system on your computer. It provides function to read, write, update, delete. ~~and~~

what is fsPromises.unlink ?

⇒ If path refers to a symbolic link, then the link is removed without affecting the file or directory to which that link refers.

③ next import cloudinary → cloud-name, api-key & api-secret.

how properly organize?

① create a method.
② in method, local file ka path doge.
③ me usko upload kar dunga.
④ successfully file upload then unlink.

```
                                   → time ke liye     → parameter
const uploadOnCloudinary = async (localFilePath) => {
    try {
        if (!localFilePath) return null;
        // upload the file on cloudinary
        cloudinary.uploader.upload(localFilePath)
    } catch(error) {
        fs.unlinkSync(localFilePath)
        return null;
    }}
```

✱ How file upload in Express.js using configures multer, a Node.js middleware

1. Defining storage strategy

```
const storage = multer.diskStorage({
```

- storage → the storage object defines where and how files will be stored.
- diskStorage → this function tells multer to save file on disk (local storage)

2. Destination Folder :

```
destination : function (req, file, cb){
    cb(null, '/tmp/my-uploads')
}
```

- cb → call back function
- '/tmp/my-uploads' → storage location.

- this function determines where to store the uploaded files.

~~file~~

3. ~~filename~~ Generating a Unique Filename

```
filename : function (req , file, cb) {
    const uniqueSuffix = Date.now() + '-' +
    Math.round(Math.random() * 1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix);
}
```

- Ensure uniqueness
- Math.random() * 1E9 → to add random no.
- file.fieldname → original field name of the uploaded file
- cb(null, ...) → passes the new filename to multer.

4. Creating the upload Middleware

```
const upload = multer({ storage : storage });
```

- upload → is now a middleware that can handle uploads.
- multer({ storage }) → initializes Multer with the specified storage configuration.

DAY-11 HTTP crash course HTTP Methods HTTP headers :

HTTP (HyperText Transfer Protocol)

HTTP(S) → just a layer, that data encript.

What are HTTP headers

metadata ← key-value sent along with request & response.

→ caching, authentication, mange state

X- Prefix → 2012 (x-deprecated)

- Request Headers → from client
- Response Headers → from server
- Representation Headers → encoding/compression
- Payload Headers → data

Most Common Header
Accept: application/json User-Agent Authorigation content-Type cookie cache-Control

CORS
Access-Control- Allow → origin ↳ credentials ↳ Method

Security
cross-Origin-Embedder → policy
cross → origin → opener policy
content → security → policy
X-XSS-Protection

HTTP methods

Basic set of operation that can be used to interact with server.

- GET : retrieve a resource.
- Head : No message body (response headers only).
- OPTIONS : what operations are available
- TRACE : loopback test (get same data)
- DELETE : remove a resource
- PUT : replace a resource
- POST : interact with resource (mostly add)
- PATCH : change part of a resource.

HTTP status code	
1XX	Informational
2XX	Success
3XX	Redirection
4XX	Client error
5XX	Server error

100 → Continue
102 → Processing
200 → OK
201 → created.
202 → accepted
307 → temporary redirect
308 → permanent redirect

400 → Bad request
401 → Unauthorized
402 → Payment required
404 → Not Found

500 → Internal server Error
504 → Gate way time out.

Day-12

Complete guide for router and controller with debugging

step-1 : create a file (user.controller.js) in Controller folder.

① Import (asyncHandler) - i helper file that's help → req, res

② create a method

```
const registerUser = asyncHandler
    (async (req, res) => {
      res.status(200).json {
          message: "ok"
      }
    })
```

③ next step → create routes

```
import {Router} from "express";
const router = Router()
export default router
```

④ import router in (app.js) and router declaration.

```
import userRouter from "./routes/user.routes.js"
```

⑤ next routes declaration in app.js

```
app.use ("/api/v1/users", userRouter)
```

when type user control → userRouter

⑥ now what do userRouter ?

```
router.route("/register").post(
                registerUser)
```

Step-2 :- open vs Terminal

→ npm run dev

✱ many times server not connected error :- ① check MongoDB connect

② port number not same.

③ check properly router import & declaration or not?

④ server all ready running.

fix this error, next server connected

Step-3 : API Testing

① go to google → Thunder client
(use for color theme).

or

② Postman → Download postman → create account

openpostman → go to collection → click (+) → write → http://localhost:8000/api/users/register

this write properly. → left side show (get) → choose (POST) → click send → get message (OK)

DAY-13

Logic building Register Controller

* challenge is → user register.

now breakdown user register step

1. // get user details from frontend.
2. // validation - not empty.
3. // check if user already exists: username, email.
4. // check for images, check for avater.
5. // upload them to cloudinary, avater.
6. // create user object - create entry in DB.
7. // remove password and refresh token field from response.
8. // check for user creation
9. // return res

* now one by one step write code.

1. const {fullname, email, username, password} = req.body

   console.log("email", email);

* now we get user details from body.

(2) validation check?

⇒ check one by one field empty or not?

field means

```
if ([fullname, email, username, password].
  some((field) => field?.trim() == "")
);
```

The code checks if any of the fields (fullname, email, username, password) are empty.

`.some` → this method is used here to efficiently check if any of the given fields (fullname, email, username, password) are empty.

`field?.trim()` : `?.` → prevents error if field is undefined or null.

`.trim()` → removes leading and trailing spaces.

```
throw new ApiError(400, "All fields are required")
```

ApiError → is likely a custom error class

400 → Bad request

D-22

next:

hold a variable. → any one find

cost existedUser = User.findOne({

③ User Already Exist?.({

$or: [{username}, {email}]

find any one username or email.

next check (existedUser) present or not

if (existedUser) {

throw new

---

④ // check for images and avatar?

```
const avatarLocalPath = req.files?.
                        avatar[0]?.path;

const coverImageLocalPath = req.files?.
                        coverImage[0]?.path;

if (!avatarLocalPath) {
    throw new ApiError(400, "Avatar
        file is required")
}
```

⑤ next upload coverImage, and avatar in → (cloudinary)

1st → import (coloudinary.js) in →

user.controller.js

2nd :- upload in cloudinary

```
i) const avater = await uploadOnCloudinary
   (avatarLocalpath.
ii) const coverImage = await uploadOnCloudinary
   (coverImageLocalPath)
```

3rd :- check if avatar not present, so create a Error.

```
if (!avater) {
   throw new APIError(400," Avatar
   file is required")
}
```

=

⑥ create user object - create entry in db

```
const user = await User.create({
   fullname,
   avater : avater.url,
   coverImage : coverImage.url || "",
   email,
   password,
   username: username.toLowerCase()
   })
```

(7) //remove password and refresh token field from response :

```
const createdUser = await User.findById
    (user._id).select("-password
                       -refreshToken")
```

`User.findById(user._id)`

→ This is a mongoose query that searches for a user in the database using their unique `_id`.

`user._id` → refers to the ID of the user that was just created.

(8) this method a document (user data) if found. otherwise null.

`.select("-password -refreshToken")`

- `select` is used to include or exclude specific fields from the retrieved document
- `-password` means exclude these fields from the response.

(9) The ensures sensitive data (like password & refreshToken) is not exposed.

(8) // check for user creation.

```
if (!createdUser) {
    throw new ApiError(500, "Something
    went wrong while registering the
    User")
}
```

(9) // return response

```
return res.status(201).json(
    new ApiResponse(200, createdUser,
    "User Registered Successfully")
)
```

DAY-14

Access Refresh Token, Middleware and cookies in Backend :

① What is an Access Token?

⇒ • An access token is a short-lived credential issued by an authentication server after a user successfully log in.

• It is used to access protected resource (like APIs) on behalf of the user.

• Typically encoded as a JWT (json web Token)

Why is it needed?

• provides a secure and efficient way to authenticate API request without requiring username and password repeatedly.
• Reduces database queries for checking user session.
• It expires quickly (e.g. 15-30 minUtes) to minimize sequrity risks.

② What is a Refresh Token?

i) A refresh token is a long lived credential used to generate a new access token without requiring the user to log in Again.

ii) It is used along with the access token but stored securely (usually in HTTP-only cookies).
iii) Unlike access tokens, refresh token are not sent with every request.

Why is it needed?

i) since access token expire quickly, refresh token allow to stay logged in without repeatedly entering credentials.
ii) Enhances security by limiting access token lifespan while ensuring a smooth user experience.

How They Work Together?

1. User logs in → Authentication server issues an access token and a refresh token.
2. Client uses the access token to access protected APIs.
3. When the access token expire, the client sends the refresh token to request a new access token.

4. If the refresh token is valid, the server issues a new access token (without requiring login).
5. If the refresh token is expired or revoked, the user must log in again.

Where to store These Tokens ?

1. Access Token : Store in memory (frontend).
2. Refresh Token : store in HTTP-only secure cookies.

Now Create Login User

1st : Create a #todo, what and how create ?

i) req body → data
ii) username or email.
iii) find the user.
iv) password check.
v) access and refresh token.
vi) send cookie.

2nd one by one write code.

i) req body → data
const { email, username, password } = req.body.

ii) username or email :-

```
if (!username || !email) {
    throw new ApiError(400, "username or password is required")
}
```

⇒ when (username) & (email) not find then throw a APIError.

iii) find the user :

```
user.findOne({
    $or: [{username}, {email}]
})
```

mongodb operators.

⇒ when User does not exist then throw a ApiError.

```
if (!User) {
    throw new ApiError(404, "User does not exist")
}
```

iv) password check :

```
const isPasswordValid = await user.isPassword
                                    -Correct

if (!isPasswordValid) {
    throw new ApiError(401, "Invalid user credentials")
}.
```

v) **Access and refresh token** :-

1st creat a method bez it's User many times

```
const generateAccessAndRefreshTokens =
async (userId) => {
  try {
    const User = await User.findById(userId)
    const accessToken = User.generateAccessToken()
    const refreshToken = User.generateRefreshToken()
  }
catch (error) {
  throw new ApiError(500, "Something went
wrong while generating refresh and access
token"); }
  }
```

* Save accesstoken and Refreshtoken.

```
user.refreshToken = refreshToken.
await user.save({ validateBeforeSave: false})
return { accessToken, refreshToken}
}
```

* now access method :

```
Const {accessToken, refreshToken} = await
generateAccessAndRefreshTokens(user._id)
```

(VI) Send cookie :

```
return res.status(200)
  .cookie("accessToken", accessToken, option)
  .cookie("refreshToken", refreshToken, option)
  .json(
    new ApiResponse(200, {
      user: loggedInUser, accessToken, refreshToken
    }, "User logged in Successfully")
  )
```

Now Create logout User

1st: create a file (auth.middleware.js) in (middleware) folder. this file check user present or not.

i) import asyncHandler :-

```
import {asyncHandler} from "../utils/asyncHandler";
```

ii) Declaring the (verifyJWT) Middleware

```
export const verifyJWT = asyncHandler
(async(req, res, next) => {
```

iii) Extracting the JWT Token

```
const token = req.cookies?.accessToken ||
req.header("Authorization")?.replace("Bearer ", "")
```

iv) Checking if Token Exists

```
if (!token) {
    throw new ApiError(401, "Unauthorized
                                @request");
```

v) Attaching User to the Request

```
req.user = user;
next();
```

Important : select (const token) to (next()) write try → press enter

vi) Error Handling

```
catch (error) {
    if (error.name === "TokenExpiredError"){
        throw new ApiError(403, "Access token
    expired, Please log in again");
    }
    else {
        throw new ApiError(401, "Invalid access
    token");
    }
}
```

Create a Router :-

```
router.route("/login").post(loginUser)
//second routes
routes.route("/logout").post(logout)
```

User.Controller.js :- logout user

1. **Middleware Definition :**

```
const logoutUser = asyncHandler(async
                    (req,res) => {
```

2. **Removing the Refresh Token from DB**

```
await User.findByIdAndUpdate(
    req.user._id,
    {$set: {refreshToken: undefined}},
    {new: true}
)
```

3. **Defining Cookie option**

```
const option = {
    httpOnly: true,
    secure: true
}
```

4. **Clearing Cookie & sending Response**

```
return res.status(200)
    .clearCookie("accessToken", option)
    .clearCookie("refreshToken", option)
    .json(new ApiResponse(200, {}, "User logged out"))
```

DAY 15

Access token and refresh token in Backend

1st: go to postman & create a register User and check log in User or not, then logout User

2nd: go to (user.controller.js) file and import jwt

```
import jwt from "jsonwebtoken"
```

Why Need Accessstoken & Refreshtoken

1. Access Tokens are short-lived, reducing the risk if stolen, while Refresh Tokens stay secure on the server.
2. No need to store User session in the database; tokens handle authentication efficiently.
3. Users stay logged in without frequent re-authentication, thanks to Refresh Tokens.
4. Works well in distributed system without relying on session storage.

3rd: create successfully register, login & logout user, next create Accesstoken & Refresh Token.

i) 1st extract Refresh Token.
ii) check Refresh Token exist or not.
iii) verify Refresh Token.

(1) Extract Refresh Token :-

```
const incomingRefreshToken = req.cookies.
refreshToken || req.body.refreshToken
```

* `req.cookies.refreshToken` → It tries to get it from cookies

* `req.body.refreshToken` → request from body.

(2) check if Refresh Token Exists :-

```
if (!incomingRefreshToken) {
    throw new ApiError(401, "Unauthorized
                                request");}
```

⇒ when reject check request then show a Api Error

(3) Verify the Refresh Token :-

```
try {
    const decodedToken = jwt.verify(
        incomingRefreshToken,
        process.env.REFRESH_TOKEN_SECRET);
```

→ JWT verify

`decodedToken` → if it's valid will contain user info (_id).

* if invalid, it will throw an error (handle in catch block).

(4) Find the User :-

```
const user = await User.findById(decodedToken?._id)
```

⇒ fetch user from the database using the (-id) from the (decoded) token.

(5) Validate the Refresh Token :-

```
if (!user) {
  throw new ApiError(401, "Invalid refresh token");
}
```

⇒ check if the user exists - if not, the token is invalid

```
if (incomingRefreshToken !== user?.refreshToken) {
  throw new ApiError(401, "Refresh token is expired or used")
}
```

⇒ compare the stored Refresh Token with the incoming one. if mismatched then show Error.

(6) Generate New Tokens :-

```
const {accessToken, newRefreshToken} =
await generateAccessAndRefreshTokens(user._id)
```

⇒ calls a function (generateAccessAndRefreshTokens) to generate : i) New Access Token ii) Refresh Token.

(7) Set HTTP-Only Cookies :-

```
Const option = {
    httpOnly : true,
    secure : true, };
return res
    .status(200)
    .cookie("accessToken", accessToken, options)
    .cookie("refreshToken", newRefreshToken,
                                options)
    .json(new ApiResponse(200,
      {accessToken, refreshToken: newRefresh
                                Token },
      "Access token refreshed"
    ));
```

(8) ~~Json~~ Handle Errors :-

```
catch(error){
 throw new Error(401, error?.message ||
                    "Invalid refresh token");
}
```

DAY - 16

Writing Update controllers for User Backend with JS

P-30

1st

Create Subscription :-

1. When user subscription a chenel then and chenel also a chenel
2. When a user subscribes to a chenel, it creates a subscription record linking the two the chenel (user) can have multiple subscriptions (other users).
3. User Table (id, name, email etc) and subscriptions Table (id, subscriber_id, channel_id, created_at)
4. User clicks "subscribe" → Adds an entry in Subscription Table. Unsubscribing removes the entry from the Subscriptions Table.

2nd

Create (subscription.model.js) in (models) folder.

1. **Import Dependencies :-**

```
import mongoose, {schema} from "mongoose";
```

⇒ (mongoose) → Used to interact with MongoDB.
{schema}: Extracting (Schema) from Mongoose to define a structured data model.

2. **Defining the Schema :**

```
const subscriptionSchema = new Schema({
```

⇒ This creates a new schema for the (subscription) model.

③ Defining Fields :

```
subscriber: {
  type: Schema.Types.ObjectId,
  ref: "User"
},
```

⇒ stores the subscriber's ID (who is subscribing)
⇒ schema.Types.ObjectId → Refers to another document (the user).
⇒ ref: "User" → Links to the User model.

```
channel: {
  type: Schema.Types.ObjectId,
  ref: "User" },
```

⇒ Stores the channel's ID (the user being subscribed to).
⇒ Again, links to the User model.

④ Creating and Exporting the Model

```
Export const Subscription = mongoose.model
  ("Subscription", SubscriptionSchema);
```

⇒ Creates the Subscription model from the schema.

3rd ✱ create a function (changeCurrentPassword) is a backend API endpoint that allows an authenticated user to update their password securely.

How work this :-
It takes the old password and new password from the user, verifies if the old password is correct and if valid, updates it with the new one.

* now, go to (user.controller.js) file → write this function.

① Define an Asyne Handler :-
const changeCurrentPassword = asyncHandler
(async (req, res) => {
⇒ This function is wrapped inside (asyncHandler) which is likely a middleware to handle errors in asynchronous function automatically.

② Extract old & New Password from Request
const { oldPassword, newPassword } = req.body
⇒ oldPassword → The current password the user is using
⇒ newPassword → The new password the user wants to set.

③ Find the User in Database :-

```
const user = await User.findById(req.user?._id)
```

⇒ (req.user?._id) → This is coming from authentication middleware (like JWT or session-based auth)

(User.findById()) → Finds the user in the database using their (_id).

④ Verify if Old password is Correct :-

```
const isPasswordCorrect = await user.isPassword
                          -Correct(oldPassword)
```

⇒ calls (isPasswordCorrect) method

⇒ this compares the provided (oldPassword) with the stored password in the database.

⑤ Handle Incorrect Password :-

```
if (!isPasswordCorrect){
    throw new ApiError(400, "old password
                        is incorrect");}
```

⇒ if the old password is incorrect, return a 400 Bad Request Error.

(6) Update User Password :-

```
user.password = newPassword;
await user.save({validateBeforeSave: false});
```

⇒ updates (user.password) with (newPassword).

⇒ ~~eatts~~ `await user.save(...)`

- saves the modified (user) object to the database
- Uses (await) because (.save()) is an asynchronous operations.

`validateBatoneSave: false`

⇒ Disable Mongoose's built-in schema validation before saving

(7) Send Success Response :-

```
return res.status(200).json(new ApiResponse
(200, {}, "password changed successfully").
```

NOTE: **Final Summary :-**

1. User provides old and new Password.
2. Check if user exists.
3. Verify old password.
4. If incorrect → update password and save.
5. Return success response.

4th Get current User :-

(1) ~~asyn~~
```
const getCurrentUser = asyncHandler
(async (req, res) => {})
```

⇒ (asyncHandler) is a middleware that catches error in async function and passes them to the error handler.

(2)
```
return res.status(200).json({
    status: 200,
    data: req.user,
    message: "current user fetched
              successfully" });
```

⇒ this function retrives the currently logged-in user's details and returns them in a structured JSON response. It is useful when you need to get the user's profile info on the frontend after authentication.

5th Update Account Details :-

(1) Function Declaration :-

```
const updateAccountDetails = asyncHandler
                    (async (req,res) =>{
```

⇒ `updateAccountDetails` → Defines an asynchronous function to update user details.

⇒ `asyncHandler` :→ Wraps the function in a middleware that automatically handles errors.

(2) Extracting Data from Request Body :-

```
const {fullname, email} = req.body;
```

⇒ Extracts fullname and email from the request body.

(3) Input Validation :-

```
if (!fullName || !email) {
    throw new ApiError(400, "All fields are
                        required");
```

⇒ checks if fullname or email is missing

⇒ if missing, it throws an error using ApiError.

(4) Finding & Updating User in Database :-

```
const user = User.findByIdAndUpdate(
    req.user?._id,
```

`req.user?._id` → Retrieves the logged-in user's ID.

`findByIdAndUpdate` → Finds the user by ID and updates the fields.

next:
```
{ $set: { fullname: fullname, email: email } },
{ new: true }
).select("-password -refreshToken");
```

($set) → MongoDB operator that updates only specified fields.
⇒ update (fullname) and (email) fields without changing other data.
⇒ ({new: true}) → Ensure that the function returns the updated user object.

(6th) Update User avatar :-

(1) Function Declaration :-

```
const updateUserAvatar = asyncHandler
    (async (req, res) => {
```

(2) Extracting Avatar File Path :-

```
const avatarLocalPath = req.file?.Path
```

⇒ (req.file?.Path) → this assumes the image is uploaded using Multer.
If (req.file) exists, it retrieves the file's local path.

(3) Validating Avatar File Presence :-

```
if (!avatarLocalPath) {
    throw new ApiError(400, "Avatar file is missing");
}
```

(A) Uploading the Avatar to Cloudinary :-

```
const avatar = await upbadOnCloudinary
                    (avatarLocalPath);
```

(B) Validating Upload Success :-

```
if (! avatar.url) {
    throw new ApiError(500, "Error while
uploading avatar"); }
```

⇒ checks if Cloudinary returned a URL.
if (avatar.url) is missing, it throws an error

(C) Updating the User's Avatar in the Database:

```
const user = await User.findByIdAndUpdate(
    req.user?._id,
  { $set : { avatar : avatar.url } },
   { new : true } ⇒ (return the updated user object)
  ), select ("-password");.
```

⇒ find the user in the database using (req.user?._id)
⇒ update the (avatar) field with the cloudinary URL

(.select("-password")) ⇒ Excludes password from the response for security.

(7) Sending Response to Client :-

return res.status(200).json(new ApiResponse(200, {}, "Avatar updated successfully"));

NOTE :- same create coverImage update :

Why are these functions needed ?

(1) Authenticated user to update their password securely → Ensures only authorized users can change their password, preventing unauthorized access.

(2) Get current user → Allows users to fetch their profile details for personalization and session validation.

(3) Update account details → Enables users to modify their profile information like name and email for account management.

(4) Update user Avatar → Lets users set or change their profile picture, improving their profile.

DAY -17

Understand the Subscription Schema

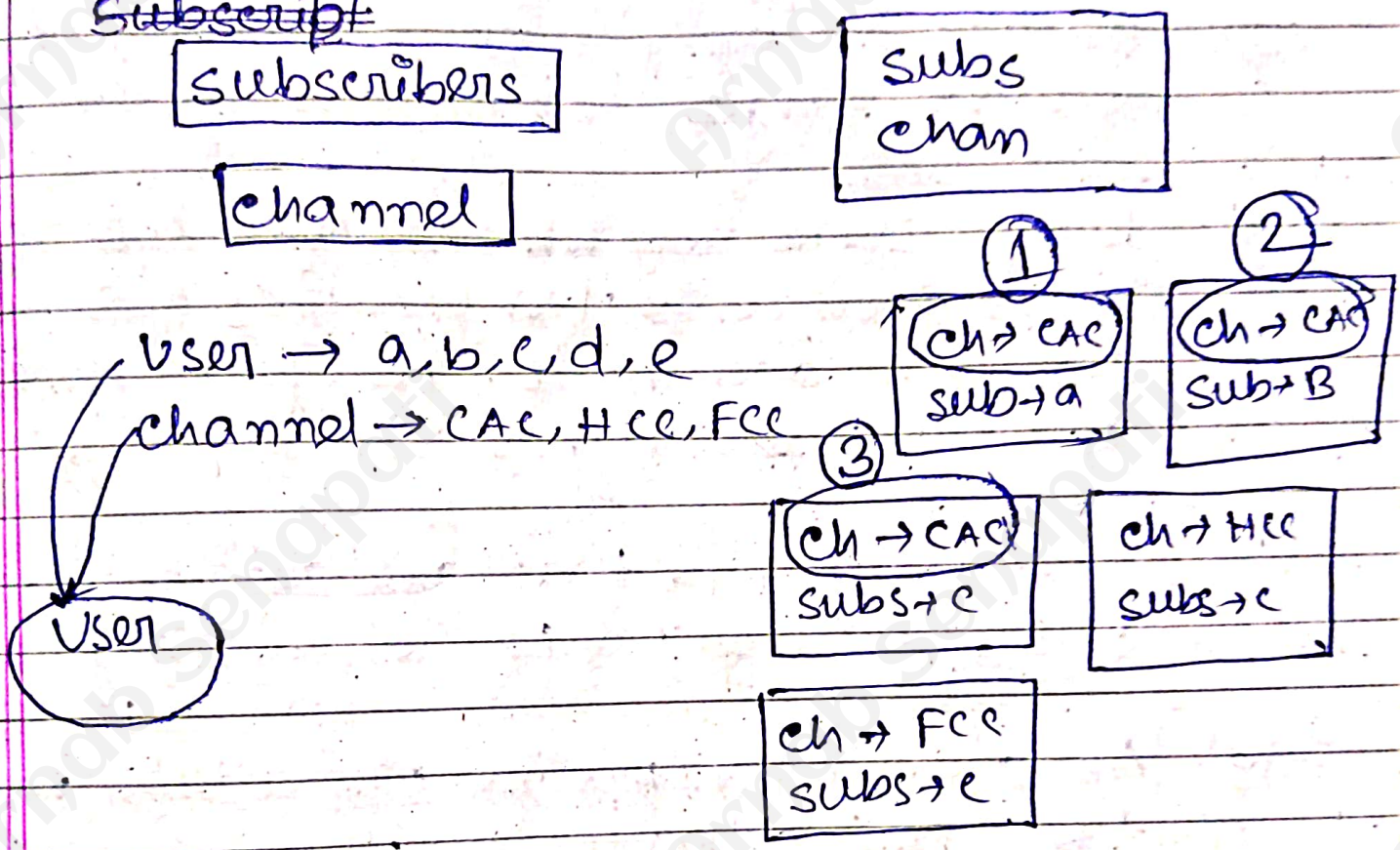
p-35

Subscription Schema

~~Subscript~~

subscribers

channel



* when User subscribe then a document create.

How a channel find subscriber?

~~when~~ 1st find channel name then select ↓ count

How find a User how many subscribe chenel

subscriber value select → suppose value (c) go to document, where value (c)? → next find chenel value → suppose get (3) so, ~~User~~ (c) User subscribe → (3) chenel.

Learn Mongodb aggregation pipelines

What is Aggregation Pipeline :-

⇒ An aggregation pipeline consists of one or more stages that process documents.

① Each stage perform an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.

② The documents that are output from a stage are passed to the next stage.

③ An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum and minimum values.

How write Aggregation Pipeline :-

```
db.orders.aggregate([
  // stage 1: Filter pizza order documents by
     pizza size
    $match : { size: "midium" }
  },
  // stage 2: Group remaining documents by pizza
     name and calculate total quany.
  { $group:{ _id: "$name", totalQuatity : {$sum:
                                  "$quatity" } }
```

Why Use Aggregation Pipeline in MongoDB?

1. Processes large datasets efficiently without fetching unnecessary data.
2. Performs filtering, grouping, sorting and transformations in a single query.
3. Returns only required results, minimizing data transfer.

What is $lookup fields ?

`$lookup` is used to perform a left outer join between two collection, allowing you to fetch related data from another collection.

Key Fields in `$lookup` :-

1. `from` → the target collection to join with.
2. `localField` → the field in the current that matches the foreign collection
3. `foreignField` → The field in the target collection to match with `localField`
4. `as` → The output array field where the joined documents will be stored.

Example Scenario :-

① Users Collection :

```
[
  { "_id": 1, "name": "Alice" },
  { "_id": 2, "name": "Bob" }
]
```

② Subscription Collection :

```
[
  { "UserId": 1, "channel": "TechWorld" },
  { "UserId": 1, "channel": "CodeDaily" },
  { "UserId": 2, "channel": "NewsTime" }
]
```

Aggregation with $lookup :-

```
db.users.aggregate([
  {
    $lookup: {
      from: "subscription",
      localField: "_id",
      foreignField: "userId",
      as: "subscriptions"
    }
  }
])
```

What is $addFields ?

$addFields is used to add new fields in documents within the aggregation pipeline.

Key Features of $addFields :-

1. Adds new computed fields to documents.
2. Modifies existing fields without overwriting the original data source.
3. Support expressions.
4. Work at any stage in the pipeline.

Example Scenario :

User

```
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 2, "name": "Bob", "age": 30 }
]
```

Now use $addFields

```
db.user.aggregate([
  {
    $addFields: {
      status: { $cond: { if: { $gte: ["$age", 30] },
      then: "Senior", else: "Junior" }
    }
  }
])
```

Now create get User channel Profile

⇒ go to (user.controller.js) file → write (getUserChannelProfile) function

- This function retrives a user's channel profile using MongoDB Aggregation Pipeline. It gathers subscription details, count subscribers, and checks if the logged-in User subscribed.

(1) **Extract username from Request Parameters**

```
const {username} = req.params;
```

⇒ Retrives username from the request
⇒ Validates if username exists and in not empty.

(2) **Validate Username**

```
if (!username?.trim()) {
    throw new ApiError (400, "Username
                        is missing"); }
```

(4) **MongoDB Aggregation Query on User Collection**

```
const channel = await User.aggregate()
```

⇒ The pipeline starts, filtering and fetching user details.

MongoDB Aggregation Pipeline Steps :

(5) Match User by (username) :-

```
{
  $match: {username: username?.toLowerCase()}
}
```

⇒ Finds the user document where (username) matches the given value.

(6) [Step-2]: Lookup subscription collection to Get Subscribers

```
{
  $lookup: {
    from: "subscriptions",
    localField: "_id",
    foreignField: "channel",
    as: "subscribers"
  }
}
```

⇒ [User._id] matches [subscriptions.subscribers]

⇒ Stores the result in the [subscribedTo] array.

(7) Step-3: Lookup subscriptions Collections to Get Subscribed Channels

```
{
  $lookup: {
    from: "subscriptions",
    localField: "_id",
    foreignField: "subscribers",
    as: "subscribedTo"
  }
}
```

⇒ User._id matches subscriptions.subscribers
⇒ Stores the result in the subscribedTo array

(8) Step-4: Add Computed Fields ($addFields)

```
{
  $addFields: {
    subscriberCount: { $size: "$subscribers" },
    SubscribedCount: { $size: "$subscribedTo" },
    isSubscribed: {
      if: { $in: [req.user?._id, "$subscribers.
                                  subscriber"] },
      then: true,
      else: false
    }
  }
}
```

⇒ `subscriberCount` → Counts the number of subscribers.

⇒ `subscribedToCount` → Counts the number of subscribedTo channels.

⇒ `isSubscribed` → Checks if the logged-in User (req.user._id) is in the subscribers list.

(9) **Step-5: Select Only Required Fields ($project)**

```
{
  $project: {
    fullName: 1,
    username: 1,
    subscriberCount: 1,
    subscribedToCount: 1,
    isSubscribed: 1,
    avatar: 1,
    coverImage: 1,
    email: 1,
    createdAt: 1,
    updatedAt: 1
  }
}
```

DAY-18

How to Write sub pipeline and routes

How get Users Watch History :

⇒ go to (user.controller.js) file → create (getWatchHistory) ~~folder~~ function

NOTE: This function retrieves the watch history of a user by aggregating data from the user collection and populating the watchHistory field with video details. It also populates the video ~~ow~~ owner details.

(1) Match the User by ID :

```
{
  match: {
    _id: new mongoose.Types.ObjectId(req.user._id)
  }
}
```

⇒ Filters ($match) the User collection to find the specific user by (_id) (converted to objectId)

⇒ This ensures that the logged-in user's data is fetched.

HOW $match work ?

⇒ ① $match acts like a filter in aggregation
② $match ~~only~~ ~~vs~~ filter documents just like find().

NEXT

③ Ensure only one user is processed.
④ prevents unnecessary processing of all user, improving performance.
⑤ Converts (req.user._id) to (objectId) for correct matching.

② **Step-2: Lookup Watch History**

```
{
  $lookup: {
    from: "videos",
    localField: "watchHistory",
    foreignField: "_id",
    as: "watchHistory",
    pipeline: [ ]
  }
}
```

⇒ ($lookup) is used to fetch the videos from the videos collection.
⇒ It matches the (watchHistory) array (which stores video (_id)'s) with (_id) in the videos collection.
⇒ The matched documents are stored in the (watchHistory) array.

⊛ How Work $lookup :

① Works like a SQL JOIN - fetches related data from another collection.

② Matches (watchHistory) video IDs with (_id) in (videos).

③ Replaces (watchHistory) with actual video objects.

④ Returned videos are in an array.

(3) Step-3 : Lookup Video Owners

```
{
  $lookup : {
    from : "users",
    localField : "owner",
    foreignField : "_id",
    as : "owner",
    pipeline : [ ]
  }
}
```

⇒ This lookup is inside the (watchHistory) pipeline.

⇒ It fetches the details of the user who owns each video (owner field in videos refers to the uploader).

⇒ The (owner) field from (videos) is matched with (_id) in the users collection.

④ ~~Prac~~ Step-4 : Project Required Owner fields

```
{
  $project: {
    fullName: 1,
    username: 1,
    avatar: 1
  }
}
```

⇒ select only specific fields (fullName, username, avatar) from the (user) collection for optimization.

How $project works?

① $project acts like a SELECT statement in SQL.
② It removes unwanted fields and keeps only the required ones.
③ Fields set on ① remain, while unspecified fields are removed.

⑤ Step-5 : Flatten Owner Array

```
{
  $addFields: {
    owner: { $first: "$owner" }
  }
}
```

⇒ since ($lookup) results in an array, this step ensures owner is stored as a single object

⑤ Final Response :-

```
return res.status(200).json(
    new ApiResponse(200, user[0]?.
    watchHistory, "watch history fetched
    successfully")
);
```

✱ Update all Router in (User.router.js) ?

① router.route("/change-password").post(verifyJWT, changeCurrentPassword)
② updateAccountDetails
③ updateUserAvatar.
④ updateUserCoverImage.
⑤ getUserChannelProfile.
⑥ getWatchHistory.

Day-19 Summary of Our Backend Series P-42

1. Core Backend Concepts Covered :-

a) Prerequisite Topic :
i) Deployment basic and overcoming deployment fears.
ii) Connecting fronted and backend.
iii) Understanding CORS and proxies.
iv) HTTP crash course
v) Node.js & Express.js basics
vi) Setting up routes and environment variables

b) Database and Models :-
i) Connecting to MongoDB (including real-world database consideration).
ii) Database structure, schema design and different model types.
iii) Demo models for e-commerce, hospital managment, etc.
iv) Understanding user models, password handling and token generation.

c) Controllers & Routing :-
i) Writing production-grade controllers.
ii) setting up user, video and subscription models
iii) Advanced aggregation pipelines.

iv) Handling authentication, user registration and logic logic.
v) Managing file uploads to the cloud.
vi) CRUD operation (Create, Read, Update, Delete).

d) Authentication & Security:-
i) JWT tokens (Access & Refresh Tokens).
ii) Middleware for authentication and authorization.
iii) Handling password changes securely.
iv) Managing user sessions and logout processes.
v) Implementing role-based access control.

e) Advanced Backend Concepts:-
i) Writing professional-level routes using controllers.
ii) API versioning.
iii) Handling errors and debugging.
iv) Pipeline and lookup operations.
v) Complex database queries and indexing.
vi) Rate limiting and security best practices.

(2.) **Encouraging Independence in Coding** :-

i) Viewers are encouraged to experiment on their own.
ii) Mistakes are part of the learning process; debugging enhances problem solving skills.
iii) Trying to build controllers and features independently strengthens backend skills.
iv) Next steps: Implementing video upload and subscription features based on learned concepts.
v) practice creating your own APIs and backend systems.

~~(3)~~ (3) **Final Thoughts** :-

i) Writing controllers and routes is the backbone of backend development.
ii) Keep practicing with different models and controllers.
iii) The series will continue, but independent learning is highly encouraged.
iv) More content coming soon - stay engaged and keep coding!

The structured summary makes it easy to understand and revise the key concepts covered in the series.

DAY-20 MongoDB models for like playlist and twee

P-14

Fixed Errors & Successful API Responses

1. **ReferenceError: getCurrentUser is not defined**

- **Cause**: The getCurrentUser function was not import ans defined in (user.routes.js)
- **Solution**: Import the missing function from (user.controller.js)

2. **Other Potential Issues Fixed**

- **Routing issues**: in (user.routes.js) were solved.
- **Middleware (verifyJWT)**: was correctly applied.
- **Controller functions**: in (user.controller.js) were debugged and corrected.

☑ **Successful Postman Respon**:-

1. User Registration (POST/register)
2. Login (POST/login)
3. Logout (POST/logout)
4. Refresh Token (POST/refresh-token)

⑤ Get Current User (GET / current-user)
⑥ Change Password (POST / change-password)
⑦ Get User channel Profile (GET / getUserChannelProfile)
⑧ User History (Get / history).

Next Create ⇒ ① comment model
② like model
③ playlist model
④ tweet model.

⇒ go to vs code ⇒ open terminal next go to (models) folder ⇒ write this command ⇒ touch comment.model.js like.model.js playlist.model.js tweet.model.js ⇒ enter

⇒ now one by one model create.
1st create [comment.model.js] ~~mode~~

[Comment.model.js] :-

① Import Required Modules

```
import mongoose, {schema} from "mongoose";
```

⇒ mongoose → Library to interact with MongoDB.

(2) Defining the Comment Schema :-

```
const commentSchema = new Schema(
  {
    content: {
      type: String,
      required: true
    },
    video: {
      type: Schema.Types.ObjectId,
      ref: "Video",
      required: true
    },
    owner: {
      type: Schema.Types.ObjectId,
      ref: "User",
      required: true
    }
  },
  {timeseries: true}
);
```

⇒ content (String, Required) → Stores the text of the comment.

⇒ Video (objectId, Required, Reference: "Video") → Associates the comment with a specific video.

=> owner: {objectId, Required, Reference: "User"} → Identifies the user who posted the comment.

=> `timestamps: true` → is the correct option to track created/updated times.

(3) Creating & Exporting the Model :-

```
export const Comment = mongoose.model
("comment", commentSchema);
export default Comment;
```

✱ Next Create Playlist Model :-

(1) Importing Required Model :-

```
import mongoose, {Schema} from "mongoose";
```

=> mongoose → Library to interact with MongoDB
Schema → Used to define the structure of playlist document.

(2) Defining the playlist schema :-

i) name
ii) description
iii) videos
iv) owner.

P-AG

1. **name** (String, Required) → Stores the name of the playlist.
2. **descriptions** : (String, Required) → Stores the description of the playlist
3. **videos** : (Array of objectIds, Reference: "videos") → An array of video references (objectId) associated with this playlist.
4. **owner** (objectId, Required, Reference, "User") → The user who created and owns the playlist

(3) Creating & Exporting the Model :-

```
Const playlist = mongoose.model("playlist", playlistSchema);
export default playlist;
```

⇒ Creates the playlist model using the defined schema.

⇒ Export the model so it can be used in other parts of the application.

Next Create Tweet Model

(1) Importing Required Modules :-

i) mongoose → Library for working with MongoDB.
ii) Schema → User to structure (Tweet) document.

(2) Defining the Tweet Schema :-

i) (content) (String, Required) → Stores the Tweet text.
ii) (owner) (objectId, Required, Reference: "User") → The user who created the tweet.
iii) (comments) (Array of objectIds, Reference: "Comment") → Stores IDs of comments on the tweet.
iv) (timestamps: true) → Automatically createdAt & updateAt

(3) Creating & Exporting the Model :-

```
export const Tweet = mongoose.model("Tweet", tweetSchema)
```

Next Create Like Model :-

(1) Importing Required Modules :-

```
import mongoose, {Schema} from "mongoose";
```

(2) Defining the Like Schema :-

i) video (objectId, Reference: "video", Required)
Stores the ID of the video that receive a like

ii) owner (objectId, Reference: "user", Required)
Represents the owner of the content being like (video or tweet)

iii) tweets (objectId, Reference: "user", Required)

iv) likeBy (objectId, Reference: "user", Required)

Why Do We Need these Four Models

(1) Comment Model (comment)

purpose:

i) Allows users to comment on a video.

ii) Stores who posted the comment and which video it belongs to.

Why Needed?

i) Enables user engagement on videos.
ii) Tracks which user commented on which video.
iii) Allows pagination for efficient comment retrival.

Example Usage:

=> A user comments "This is a great video!"
=> The system saves the comment and links it to the correct video & user.

(2) Playlist Model :

Purpose :-

i) Allows user to create and manage playlists of videos.
ii) stores playlist name, description, and video list.

Why Needed?

i) Helps users organize their favorite videos.
ii) Supports "Watch Later" and "Favorite" lists.
iii) User can group videos into categories.

Example

=> A user creates a "Coding Tutorials" playlist.
=> The system stores the playlist and likes the added videos.

③ Tweet Model (Tweet)

purpose :-
i) Allows users to post tweets (like on Twitter)
ii) Supports likes and comments.

Why Needed :-
i) Users can express opinions and discuss videos.
ii) Enables social interaction on the platform.
iii) Allows tracking of who posted what.

Example :-
i) A user tweets: "This new feature in JS is amazing".
ii) The system saves the tweet, links it to the user, and allows others to like/comment.

④ Like Model (Like) :-

purpose :-
⇒ Tracks who liked what (videos or tweets).

Why Needed ?
i) Increase user engagement (likes boost content visibility)
ii) Helps personalized recommendations (user see content they like).

iii) Allows analytics (track most-liked videos/tweets).

Example Usage :-

=> A user likes a video or tweet.

=> The system records the like and updates the like count.

How These Model Work Together

1. Users create and like tweets
2. Users comment on video.
3. Users create playlist and add videos.
4. Likes are tracked from videos and tweets.

=> Together, these models build a complete video-sharing + social interacting system!

DAY-21 Covered Essential controllers for comments like, playlist, tweets, videos and more p-49

1st: Create comment Controller, Health check, Dashboard, Like, Playlist, Tweet & video controller → go to vs code terminal → go to controller folder → touch comment.controller.js Helalthcheck.controller.js Dashboard.controller.js Like.controller.js playlist.controller.js Tweet.controller.js video.controller.js

2nd Write One by one controller :-

① Comment Controller :-

purpos : Manages user comments on tweets/video

Functions :-
- addComment (req, res) : Adds a new comment to a tweet or video.
- deleteComment (req, res) : Removes a comment by ID.
- getComments (req, res) : Fetches all comments for a specific tweet/video.

MongoDB operations : Uses find(), findByIdAnd Delete(), and populate()

(2) Health Check Controller :-

Purpose :- Ensures the backend server is running properly.

Functions :
- healthCheck (req, res) : Return a success message with status code 200.

MongoDB Operations :- Re Not required, simple response.

(3) Dashboard Controller :-

purpose :- Manages like and unlike action on tweets/videos

Function :-
- likePost (req, res) : Adds a like to a post
- unlikePost (req, res) : Removes a like from a post.
- getLikes (req, res) : Fetches all likes for a post.

MongoDB operations : Uses findOneAndUpdate(), deleteOne() and countDocuments()

(4) Like Controller :-

Purpose :- Manages like and unlike actions on tweets/videos.

Functions:

- likePost (req, res): Adds a like to a post.
- unlikePost (req, res): Removes a like from a post
- getLikes (req, res): Fetches all likes for a post.

MongoDB operations :- Uses findOneAndUpdate(), deleteOne() and countDocuments()

(5) Playlist Controller

Purpose: Handles playlist creation, updates and retrieval.

Functions:

- createPlaylist (req, res): Creates a new playlist.
- deletePlaylist (req, res): Deletes a playlist
- getPlaylists (req, res): Retrieves all playlists for a user.

MongoDB operations: Uses insertOne(), findByIdAndDelete(), and populate().

⑥ Tweet Controller :-

Purpose : Manages user tweets/posts.

Functions :
- createTweet(req, res) : Creates a new tweet.
- deleteTweet(req, res) : Deletes a tweet.
- getTweets(req, res) : Retrieves all tweets.

MongoDB Operations :- Use save(), find(), and populate().

⑦ Video Controller :-

Purpose :- Handles video-related actions.

Function :-
- uploadVideo(req, res) : Uploads a new video.
- deleteVideo(req, res) : Deletes a video.
- getVideos(req, res) : Fetches all video

MongoDB Operations :- Uses save(), findByIdAndDelete(), and populate().

3rd Next update in routes → go to routes terminal create → comment, dashboard, healthcheck, like, playlist, subscription tweet, ~~ts~~ and video routes.

Routes Folder - API Routing Structure :-

In the routes folder, we define the API endpoints for different feature of our backend. Each route file handles specific functionalities and maps to corresponding controllers.

1. comment.js - Handles routes for adding, deleting, and fetching comments.
2. healthcheck.js - Provides a route to check if the server is running and healthy.
3. like.js - Manages like/unlike operations for tweets and videos.
4. playlist.js - Handles playlist creation, updates and retrieval.
5. tweet.js - Defines routes for posting, fetching and deleting tweets.
6. subscription.js - Manages user subscriptions to other users or channels.
7. video.js - Handles video upload, retrieval and metadata updates.

Day-22

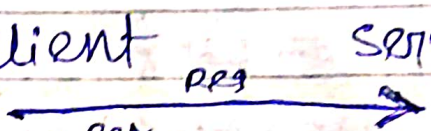
Socket.io

* communication Protocol => HTTP / Web socket / FTP / SMTP

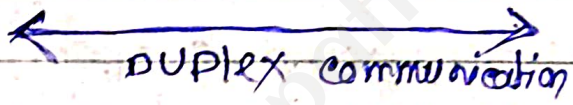
↓

use this make API

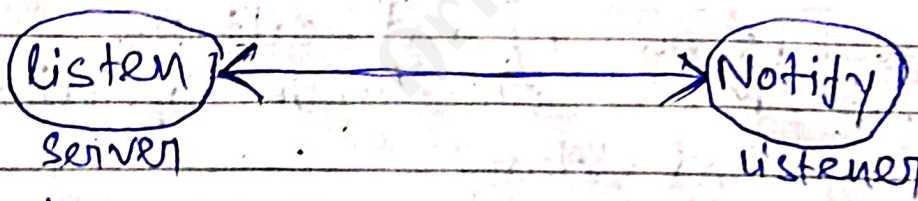
HTTP →

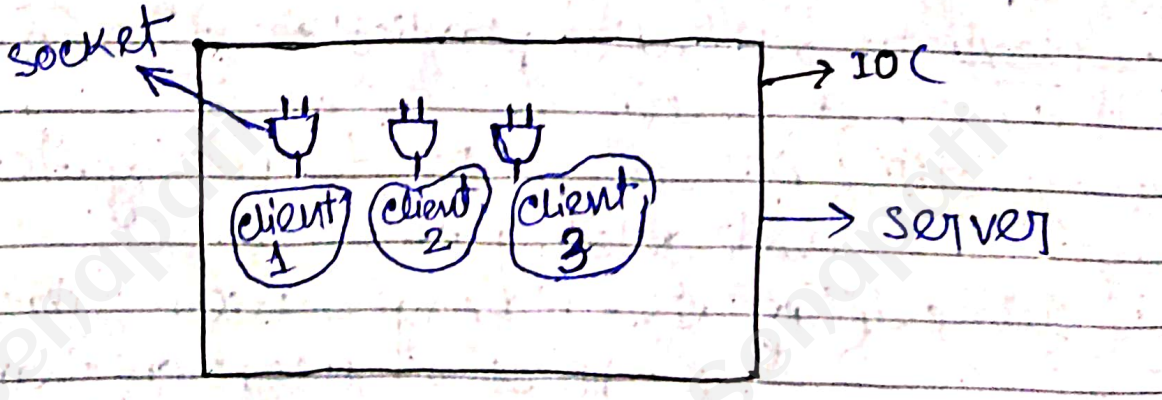


webSocket →



NOTE: suppose we make a server and suppose create a event there listen, here server sent data on listener sent data





What is ~~soket~~ socket :-

A socket is basically an endpoint for communication between two machines (or two process). It's used when you want real-time, two-way communication between the client (like a browser, mobile app) and the server (your backend).

Example :-

When we chat on WhatsApp or see live cricket or football scores on a website — the information is constantly updating without you refreshing the page. This is often done using sockets.

In Technical Terms :

1. Sockets are part of network programming
2. They allow data to be sent and received over a network.
3. They enable persistent connection, meaning the client and server stay connected instead of connecting again and again like in normal HTTP request.

What are the difference HTTP vs socket

HTTP	socket
Client sends a request, server sends a response	Both client and server can send message anytime
2) Connection closes after each request	2) Connection stays open (persistent)
3) NOT real time	3) Real-time communication

IN Node.JS (Backend) we often use :-

1. WebSockets (The protocol)
2. Socket.io (The library to easily use sockets).

EX:

```
const io = require('socket.io')(3000);
io.on('connection', socket => {
    console.log('User connected');
    socket.on('chat message', msg => {
        io.emit('chat message', msg);
    });
});
```

What is io in Backend?

⇒ Think of io as a manager that handles all real-time communication between the backend and multiple users (clients). It allows sending and receiving data instantly without needing to refresh the page.

How Do We Get io?

In Node.js when we import socket.io, we create an io instance like this:

```
const socketIO = require('socket.io');
const io = socketIO(server);
```

This io object is the WebSocket server that: Listens for new connections
- Sends message to all connected

Understanding (io) in Simple words :-

① `io` → A server that manages all webSocket connection.

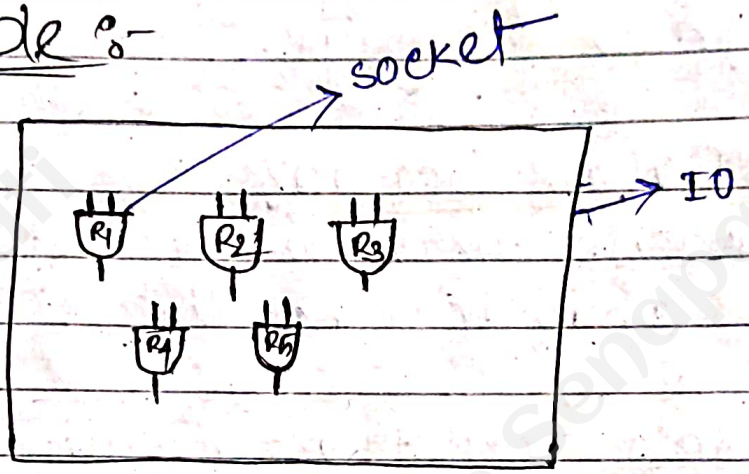
② `io.on('connection', callback)` → Listens for new users connecting

③ `Socket.on('event', callback)` → Listens for messages from a user.

④ `io.emit('event',data)` → sends a message to all users.

⑤ `socket.emit('event',data)` → Sends a message to only one user

Example :-



suppose we have 5 socket
R1, R2, R3, R4 & R5

① I want send a message to one specific socket or all 5 socket
⇒ use this function → emit()
⇒ socket1.emit('message', 'Hello User1!")

② Each socket listens for an event
⇒ use this function → on()
⇒ socket2.on('message', (data) ⇒ console.log(data));

③ Suppose 5 users are connected to the server, and User 1 sends a message. User 1 will not receive the message, but all ~~the~~ other users (user 2, 3, 4 and 5) will receive it.

⇒ use this function → `broadcast.emit()`

④ Suppose:
- User 1, User 2 and User 3 join Room 1.
- User 4 and user 5 do not join Room 1

If User 1 sends a message to Room 1, only User 1, User 2 and User 3 will receive the message, but User 4 and User 5 will not receive it.

⇒ use this function → `to(room).emit()`

⑤ Suppose:
- User 1 connects and joins Room 1
- User 2 connects and joins Room 1
- User 3 connects and joins Room 2

If a message is sent to Room 1, only one User 1 and User 2 will receive it, User 3 will not.

⇒ use this function → `join()`

Next

① create a folde → open it in vs code → open vs code Terminal → write (mkdir server) & (mkdir client)

② Next separate two terminal → client & server

⇒ write command for server →

1. npm init -y
2. npm i express soket.io
3. npm i --save-dev nodemon
4. npm i cors

⇒ Write command for client

1. npm create vite@latest
2. npm i
3. npm install @mui/material @emotion/react @emotion/styled.

1st Create a client, server there see server ~~req~~ & client request and response upon.

⇒ 1st create server APP.JS

(1) setting up the Express server & webSocket :-
i) import required modules: express, socket.io, http and cors
ii) Create an Express app and an HTTP server.
iii) Attach a socket.io server (io) to server.

(2) Handling CORS (Cross-Origin Resource Sharing)
i) Allow the client (http://localhost:5173) to communication with the server using "GET" and "POST" request.
ii) Enable credentials: true for cookies/auth

(3) Defining Routes:
GET /route : Returns "Hello World" when accessed in the browser.

(4) Starting the server:
The server.listen(port, callback) starts the server on port 3000.

NOTE
(*) i) When a client connects, it logs their socket ID.
ii) It broadcasts a "welcome" message to all other connected users.

2nd Create Client (APP.jsx) :-

① **Importing Dependencies :-**
i) React for rendering UI
ii) socket.io-client for WebSocket communication

② **Creating the Socket Connection :-**
i) Calls io("http://localhost:3000") to connect to the server
ii) This connects automatically when the app loads.

③ **Handling WebSocket Events :-**
i) socket.on ("connect")
⇒ Logs "Connected to server" and the assigned socked.id
⇒ Sends a "message" event to the server (currently not handled).
ii) socket.on ("welcome")
⇒ Listens for the "welcome" message from the server.
⇒ Logs it to the console.

NOTE :
i) Connects to the WebSocket server on load
ii) Logs "Connected to server" + socket.id.
iii) sends "Hello from client"
iv) Listens for "welcome" messages and logs them.

Part-2
D-23

Socket.io

P 56

What You Learn from Socket.io :

① **Real-Time Communication:**
- Enables instant messaging between server and clients using WebSockets.
- Supports bi-directional event-based communication

② **Connecting Users :**
- io.on ("connection", socket => {...}); detects when a new user connects.
- Each user gets a unique socket.id.

③ **Emit & Listen :**
- socket.emit ("event", data): send data
- Socket.on ("event", callback): Receive data.
- ~~socket.on ("event", callback): Receive data~~
- socket.broadcast.emit(): send to all except sender.

④ **Joining Rooms :**
- socket.join ("roomName"): User joins a specific room.
- io.to ("roomName").emit(...): send message to users in that room only.

⑤ **One-to-One or Private Chat :**
To chat between specific users:
- Maintain a map of users and their socket IDs
- Use io.to(socketID).emit(...) to send a message to a specific user.

Example :

```
io.to(targetSocketId).emit("privateMessage",
message);
```

⑥ Cleanup & Disconnection :

- socket.on("disconnect",...): Detect when user leaves.
- socket.disconnect(): Close connection on unmount (client-side).

HOW JWT tokens and secret keys Work

⇒ especially in the context of your backend project.

✱ What is a Token and Secret key in JWT?

① JWT (JSON Web Token) :

A JWT is a secure, compact way to transmit user data (like user ID, email, etc) between client and server.

It looks like this ⇒ xxxx.yyyy.zzzz

- Header : Info about the algorithm used
- payload : The actual user data.
- signature : Ensures the token is not tampered with.

P.57

2. What is a Secret key?

The secret key is used to sign the token so the server knows it's genuine.

EX:

```
const token = jwt.sign({id: "asdgdsgsjasjhgaas"},
secretkeyJWT);
```

- jwt.sign(payload, secret) creates the token.
- Later, you can verify it using:

=> `jwt.verify(token, secretkeyJWT);`

3. How it Works in a Flow

On Login:-

1. Server creates token → jwt.sign(userData, secretKey
2. Token is sent to client (via cookie or response)
3. Client stores it (cookie/localStorage)

On Protected Routes:-

1. Client sends token with request
2. Server verifies → jwt.verify(token, secretkey)
3. If valid → access granted; else → denied.

Important:-

- Keep your secretkey JWT safe and private.
- Use httpOnly cookies for better security (prevents xss attacts).