1] Backend overview :-

⇒ Backend :-

Server : a software which some thing serve. which can ~~anything~~ computers.

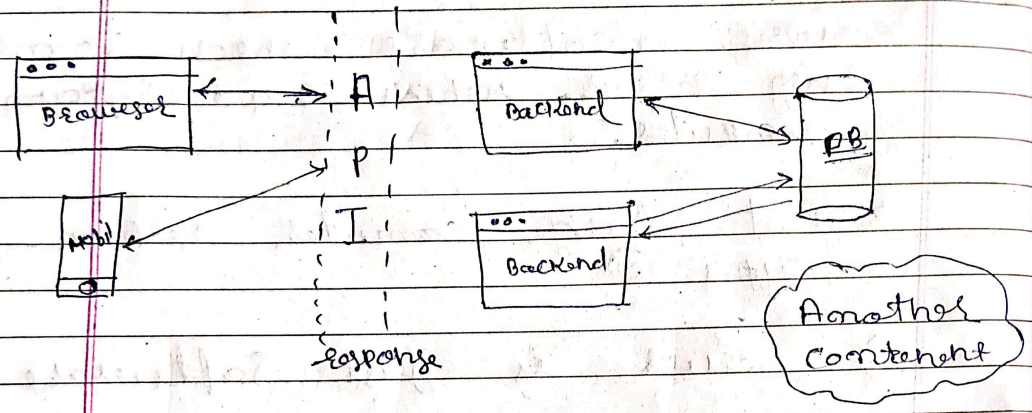
→ Not large comuter which myths ☺

⇒ Server is just software.

⇒ 2 Major components :-

Programming lang.	A database
→ Java, JS, PHP, golang c++, etc & it's framework.	→ Mongo, MySQL, Postgres, sqlite,
⇒ we use lang and its framework for ease.	→ ORM, ODM eg: mongoose,

⇒ How the backend works.



⇒ Backend is the Processing of storing, retriving and manage data through the API and framework for storing data.

⇒ A Java Script based backend

→ whenever we make the backend we deal & only with 3 things.

Data	File	Third Party (API)
string, Num, obj, any	Img, Pdf, etc	google login, AWS, map etc.

⇒ A JS Runtime : Node | Deno | Bun

we can use any of the above to run JS in backend.

⇒ File Structure :-

→ package.json : Dependecies file manage
→ .env : for storing env variable
→ etc (Redme, git, gitignore,)

SRC ⇒

- [ ] DB
- [ ] Models
- [ ] controllers
- [ ] Routes
- [ ] Middleware
- [ ] utils
- [ ] More (depends)

File :-

index
↳ DB conn

App
↳ config
↳ cookie
↳ urlencode

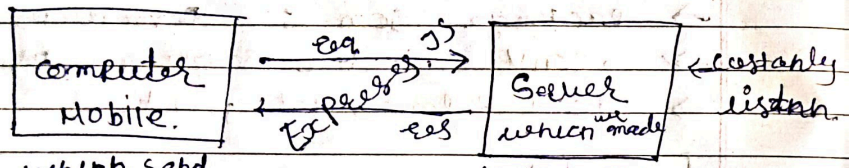
constant
↳ enums
↳ DB-name

2] How to Deploy.

⇒

Just Knowledge :- we use Node.
so check it.
node -v
npm -v

⇒



which send req

listen
/ : Home
/login

⇒ Now, we make Server.

1] ~~npm~~ npm init to make Folder Struct. and description.

npm init (-y) ← for automatically & default.

2] Script :
"start" : "npm | nodemon index.js"
dev
build
etc.

→ It's for starting or running the file or server.

⇒ Express :- install it.

→ we used it for routing.

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

const app = express();
const Port = 4000;
```

⇒ Now we have Passed to use express through app variable.

PORT : where our server is listen.
- it just virtual port like physical port is our comp.. 65535
- there is many ~~6555~~ Port.

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

app.listen(Port, () => {
    console.log(`running on ${Port}`);
});
```

⇒ Test your .env variable and deploy on your favourite host Platform.

3] Connect Backend Frontend

⇒ there is two ways to use files or Package.

① common JS ② module JS.

ex: require("express") ex: import "" from

⇒ If you use ② module JS then you have tell to Package.json file "type": "module" in file.

⇒ So, it treat the files as module.

⇒ If you use it and don't specify in Package.json file it gives error.

Tips:- whenever you get json data for better visualization & understand go the JSON formatter and lively other and post your json data that beautify in tree, graph structure. to see it Proffested way

⇒ Tool chain / bundler :-

→ to make react app from Vite, CRA, Parcel etc which convert the make bundle of HTML, CSS, JS which web understand end of the day.

⇒ CI/CD pipline introduce in this era. mean where this tool chain are came in market.

⇒ Make a front-end.

Create Vite@latest .

Tips:- dot is for create app in only this folder instead of create other folder.

⇒ For fetching & other request action in front end we use axios

→ we not have to Parse data it do automatically.

cross origin resource sharing

⇒ CORS : It provide safety to your application. (Safety.)

⇒ CORS Policy origin error :- you might be face this error.

ex: we not allow in our home to other unknown person same it not allow other API in our origin.

→ whenever our frontend backend are in different domain, port, Hosted this are cross origin.

Solution-1 to say your backend team to make your IP, or API whitelist.

⇒ to install CORS package and use it.

→ and using this package define your origin and also you can define your whitelist api.

4] Data modeling for Backend with mongoose.

TIPS: fresher are just starting work and experienced are asking question on it, what is, why, how at step. for proper requirements.

⇒ In this, lecture overview of the How the application to build which process, best practices and like all things are deciding, before starting build application.

⇒ Data modeling :

→ It is the process and building the application of which type of Data we required.

⇒ How they are relation with the others data

→ In fact, there all team for it to deciding the what we required data

Page No.
Date

⇒ After that they build model with the help of tools like Moon Modeler and etc.

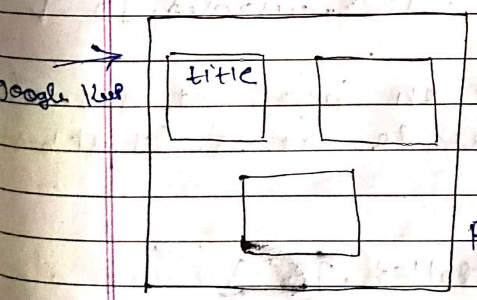
⇒ For Free. eraser.io we can use for data modeling.

Tips: whenever we some build first we have to decide to ~~st~~ How we save data:

Ex: what we first make login or Register
→ login is just validation to which stored in data.

→ In short, first we have decide which we take field required. So like, username, Pass, DOB, photo. etc.

⇒ For example; Rough idea of Todos



For outer box: we need.
title
color
subTodos: [{}, {}, {}]

For inner Todos:
content
markedAsDone
Created date.

⇒ So, first we make modeling of the our app then we use the mongoose, prizma helper for storing data.

Tips: It's called helper which is help to store the data (mongoose, prizma)

⇒ Code phase :-

⇒ After deciding modeling make on making models first in code. with help of mongoose.

→ It is file but most popular conventions and standard are as follow.

user.js ✗ user.model.js ✓

→ name convention bcoz it is model file so we write it as. (Just best practices)

same

user.controller.js ✓

⇒ for creating model schema in mongoose. ③ line are main ~~import~~

import mongoose from "mongoose"

const userSchema = new mongoose.Schema({})

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

⇒ Right way to define field in Schema.

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

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

⇒ Also, they provide Validation in Schema go to Docs → Validation.

⇒ TimeStamp is most common thing in Schema that's why mongoose make it easy to just pass object {timestamps: true} in Schema after defining all fields. So it automatically add createdAt & updatedAt field added.

⇒ Imp in Schema.

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

(it's type of ref)

name of ref database schema model name.

```
subTodos: [ ]   (acceptable)
```

but

```
subtodos: [
  {
    type: mongoose.Schema.Types.ObjectId,
    ref: "SubTodo"
  }
]
```

⇒ There is we can also export make schema in the same file.

ex: ~~export~~ const orderItemSchema

⇒ also we can define enum for choices, in the only given value.

⇒ default also property there.

5] Set up a Professional backend Project

→ 100% Production grade
→ most Company's Stadelize approach.

→ git track only files, so when we create folder or inner folder without any files so, we can't Push that folder.

→ Fodr Structure is properly follow by the company according it's requirment but mostly are as discussed & others are follows.

⇒ Prettier: code beauteby & formatter.
- it required because we not only work but in company there is People which work on same Project so, it's other defined How much Spaces, semicolon are we keep in our app code.

→ it's setting other depend by Project manager or head for code consistency.

⇒ For prettier file configuration make file with `.Prettierrc` name and define all requirment configuration.

esc: in file:

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

⇒ there is also one it's file which name is `.Prettierignore` which is used for which file not apply Prettier format.

esc: in file:

```
*.env
.env
.env.*
```

⇒ There are also, loader generator for it all file. but it's all are depend on our requirement.

6] How to connect database in MERN with debug.

① make mongoDb atlas account for cloud.
Caution:
- always or there is only one machine IP is provided where our backend served.

- 'allow from anywhere' is not good and caution. but if our use can use debugging purpose.

② after set environment Variable in .env

PORT = 8000
MONGODB_URI = mongodb+srv://..... .net

→ after we give name to our app database in constant.js as below.

export const DB_NAME = "videotube"

③ There is two approach to connect mongoDB.

④ Professional and modular approach where all required connection func. code are in DB → file. and called by the index.js file.

⑤ directly all connection code are write in the index.js file.

→ dotenv Package : it's require to load env Variable in our app. it use require syntax.

Tips: Database about some points...
- whenever we connecting database problems are necessary.
- make sure we use try catch or promise
- Database is always in other continent make sure async await used.

① → So, now we use first approach to write code in index.js file with IIFE func.

Page No.
Date

```
;( async () => {

  try {
     await mongoose.connect(`${P.env.
 Mongo_URI}/${DB_NAME}`)

    app.on('error', (error) => {
       console.log("Error", error);
       throw error
    })

    app.listen(process.env.PORT, () => {
       console.log("APP listining ${PORT}")
    })

  } catch (error) {
       console.error("Error", eror)
       throw err

  })()
```

express event for listner error handling

// for run on port

② Second dB connection approach.
to write all code in modular & in DB folder file.

Page No.
Date

⇒ In [dB] ⇒ Index.JS file.

```
const connecDB = async () => {
  try {
     const connInstance = await
  mongoose.connect(`${p.e.URI}/${DB_NAME}`)
  console.log(`\n MongoDB connected !!
   ${connectionInstanc.connection.host}`);

  }catch (error) {
     console.log("MongoDB conn Failed", error)
       process.exit(1);

export default connetDB
```

mongo DB return obj for success connection we can check orsee the other it host

process is from node which means terminate process

↳ 0 means without failure 1 mean with failure.

⇒ In main Index.JS file.

```
require('dotenv').config().
import .....
import .....
ConnectDB();
```

} It break code consistency so we use import statement with flag.

import ...
import dotenv from 'dotenv'

dotenv.config ({
    path : './env'
})

connectDB()

⇒ in package.json
nodemon -r dotenv/config --experimental-json-modules src/index.js "

⇒ This all things are only for dotenv package error to all the app when it used and for import statement instant require

⇒ dotenv package used for the provide error to all in app.

⇒ when ever error occure read properly because in backend even .js extension error are problem. it will not given.

7] Custom API response & error Handling

⇒ Now, we propose into express.js file.

after, connection DB file it return promise. so we write app listen on this then logic. like as.

⇒ index.js
connectDB()
.then(() => {
app.listen(process.env.PORT, () => {
console.log("app listen....")
})
.catch((e) => {
e.log("Failed....")
})

⇒ In app.js file we write all express related code.

import express from 'express'

const app = express();

export {app}

← Here all middleware and other package config are write which we discuss ahead.

→ In between the app.js

```
app.use(cors({
    origin: process.env.CORS_ORIGIN,
    credentials: true
}))
```

It is cross origin resource sharing for authentication & cookies setting.

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

For accepting the json data from users.

```
app.use(express.static("public"))
```

It's for static assets accessing.

```
app.use(cookieParser())
```

We install this cookie-parser pkg for the sending and storing cookie and performing CRUD operation to accessing set cookies.

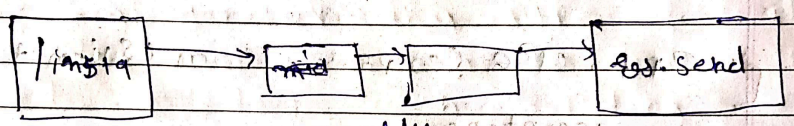
```
app.use(express.urlencoded({extended: true, limit: "16kb"}))
```

for handling the url data from the user.

⇒ Middleware overview :-

→ middleware work is this process between req & res to performing some task.

→ ex: to user hit the /instagram url but we want to check is logged user so we made middleware to check before sending response.



middleware

⇒ There is next argument for passing to the next middleware.

→ There is always 4 argumen not only (req, res) but (error, req, res, next)

→ middleware are always write in sequences.

⇒ Utils Folder use & implementation :-

→ In utils all utility files and reusable code are written.

⇒ For try catch for promises we use in all connection & request so we use it handler.

→ we make file in utils → asyncHandler.js

① with try catch.

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

② with promises.

```
const asyncHandler = (requestHandler) => {
    return (req, res, next) => {
        Promise.resolve(requestHandler(req, res, next))
        .catch((error) => next(err))
    }
}
export {asyncHandler}
```

⇒ In general this type of writing code better approach to reduce the code of try catch repeating.

→ for attaching try catch we made wrapper.

⇒ Same way for catch or error part there is also one better approach to customize the error code.

→ Node provide error class for Handling error.

→ b with help of it we make our custom error api, like as.....

→ In utils → ApiError.js file.

```
class ApiError extends Error{

    constructor(
        statusCode,
        message = "something went wrong",
        errors = [],
        stack = ""
    )
    {
        super(message)
        this.statusCode = statusCode
        this.data = null;
        this.message = message;
        this.success = false
        this.errors = errors

        if (stack) {
            this.stack = stack
        }else{
            Error.captureStackTrace(this,
            this.constructor)
        }
    }
}

export {ApiError}
```

⇒ Also same kaam as we make file for success response.

→ for binding the response with the proper manner & response we write this code as error we modified file.

→ In, utils → ApiResponse.js file.

```
class ApiResponse{

    constructor(statusCode, data, message=
    "success")
    {
        this.statusCode = statusCode
        this.data = data
        this.message = message
        this.success = statusCode < 400
    }
}

export {ApiResponse}
```

8] User & Video model with hooks & JWT

⇒ in the user & Video model we need first it's model file and define all regular field, name, email, pass etc. but the main things are written below of that model and myths.

⇒ index: true, it's property in define on our model schema field for searching optimizing way. so we define index true for that field.
→ remember give more index is affect the app.

⇒ avatar or Thumbnail or any image or Video we use third party api to storing this and they provide us link and some data for accessing it. ex: cloudinary, AWS,

⇒

⇒ so in model we just define it's type as a String but it's link of cloudinary or third party api.

⇒ mongoose-aggregate-paginate-V2 :-

- it's allow us to make the aggregate query efficiently and give more power to write advance aggregates query.

→ It's inject like Plugin in our Project. (in Schemas)

→ ex: with explanation

```
import mongooseAggregatePaginate from
    "mongoose-aggregate-....";

const videoSchema = new Schema({
    // our property with field.
});

videoSchema.plugin(mongooseAggregatePaginate)
```

schema variable we give to add plugin and other pre, post method to perform task.

⇒ bcrypt:
- It is a package which allows us to encrypt our password with hashes.

⇒ JWT (JSON web tokens):
- It ~~allows~~ is based on cryptography.
- It is used for send token to clients.
⇒ There is three parts when encoded with secret keys.

① Header ② payload & data ③ Secret key

encoded data based on it.

⇒ Middleware hook: it is the same kind of method or hook which used to perform on specific requirements.

→ ex: Pre, Post, etc (mongoose docs).

→ ~~use of pre & bcrypt in user schema~~

```
UserSchema.pre("save", async function(
next){
this.password
```

⇒ Use of pre & bcrypt in Schema

⇒ We use pre for we want to perfor encrypt the password before they save in DB. So we do it as...

~~User.Sc~~ Here, we don't use arrow func due to the this keyword is reserved

```
UserSchema.pre("save", async function(
  next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password
    , 10)
  next();
});
```

⇒ For Validation checking we use the password is correct or not we use it as below.

⇒ We make isPasswordCorrect name function in the UserSchema.method as like below.

Here, we don't use arrow fun because it not provide context of this keyword

userSchema.methods.isPasswordCorrect = async function(password) {

return await bcrypt.compare(password, this.password)

}

→ userSchema.methods in we add our methods as above write isPasswor...

⇒ JWT :-

- JWT is bearer token.
- means who have this token it would be right. (જેવાકે પાસરકોળ ચે ચાંરહી)

→ In .env we define secret key as like below.

ACCESS_TOKEN_SECRET = OT4XPWZ........ UXPNX
ACCESS_TOKEN_EXPIRY = 1d
REFRESH_TOKEN_SECRET = CTKTOU......PTXNY
REFRESH_TOKEN_EXPIRY = 10d

→ So, as above we write fun in method for bcrypt same we writ for our JWT token.

userSchema.methods.generateAccessToken = function() {

return jwt.sign(
{
id: this._id,
email: this.email,
userName: this.userName,
fullName: this.fullname
},
process.env.ACCESS_TOKEN_SECRET,
{
expiresIn: process.env.ACCESS_TOKEN_EXPIRY
}
)
}

} as we want we can write prope

⇒ As, same way we write for REFRESH TOKEN...

userSchema.methods.generateRefreshToken = function() {

} There only we pass _id

}

9] Q Have to upload file in backend (Multer)

⇒ Cloudinary : It's provide storage for our video & files assets.
- Most companies use this third party for its assets and also we used it for our Projects.

⇒ 1] create account on it.

⇒ 2] we use also middware for file uploading is multer.

→ express-fileupload is earlier used but today most popular is multer.

⇒ lets start working.

⇒ npm i cloudinary multer

⇒ Strategy for file uploading

→ we take file from user and temporary keep in our server with using of multer.

→ After, with using cloudinary we take file from our local server and upload in cloudinary.

⇒ why this 2 steps instead of direct uploading in cloudinary ?

→ because, we give first it's production grade and recommended every in industry for attaching the reuploading if any problem occurs.

⇒ In, [utils] we create utility of cloudinary as name cloudinary.js.

→ In this file we give the path of local files where are in uploaded the local server.

→ After upload we unlink/remove from our local server for cleaning

⇒ fs module in nodejs:
- it's predefined module in nodejs for file operation.

→ unlink :
- when we want do delete file so we unlink it. (It's terminology for delete file in os)

→ So after uploading file we unlink from our server. using this fs module method.

→ In, file. code....

Import {v2 as cloudinary} from 'cloudinary';
import fs from 'fs'

cloudinary.config({
cloud_name: P.e.CLOUDINARY_CLOUD_NAME,
api_key: P.e.CLOUDNARY_API_KEY,
api_secret: P.e.CLOUDINARY_API_SECRET.
})

⇒ Now is uploading is too easy in cloudinary just use .V2.uploader.upload method but,
→ we make it some additional to more efficient and for unlink the file from local.
→ So, for that we make func. as...

const uploadOnCloudinary = async (localFilePath) => {
try {
if (!localFilePath) return null
// File upload on the cloudinary.
const response = await cloudinary.uploader.upload(localFilePath, {
resource_type: "auto"
}) → For any format and jpg, video etc
return response;
// we also we remove the file after uploading
} catch (error) {
This is also some property.
fs.unlinkSync(localFilePath)
return null;
}
}
remove uploaded file. if the upload is failed.

export {uploadOnCloudinary}

⇒ Multer middleware writing code:

→ whenever we perform file uploading we use multer middleware so we use one middleware for it.

→ In [middlewares] folder we make file multer.middleware.js

→ There is three storage memoryStorage & diskStorage but we diskStorage for prevent to load on memory.

→ code:

```
import multer from "multer"

const storage = multer.diskStorage({

    destination : function (req, file, cb) {
        cb(null, "./public/temp")
    },

    filename : function (req, file, cb) {
        cb(null, file.originalname)
    }
})

export const upload = multer({storage})
```

↳ learn more about

10] HTTP crash Course

→ HTTP | HTTPS : HyperText transfer protcol | Secured:

→ In, HTTP data traffic in plain text & in HTTPS it encrypted.

→ URL | URN | URI : uniform resource Locator | Name | Identifier.

→ It's just address that where we want.

⇒ Headers :-

→ It is metadata. which keep the data about data.

→ Headers are sent along with req & res.

→ Headers are used for many ways some of are belows.

Caching : recently network req can data in header used for any purpose.

→ authtication : all this session cookies, bearer token, refresh token, any kind of auths.

→ manage State : for checking is it logged user, guest user, have they perform any action (add to cart, or etc)

TIPS : ⇒ the caller X used for as prefix in the header name. X-name, etc.
~~⇒~~ it deprecated

⇒ There are no any types of headers but we categorize as below.

→ Request Headers : It's come from the client browsers.

→ Respond Headers : It's come from the servers. there some specific rule of success data (200 code) not found (404) etc. for standerdize.

→ Representational Headers : ~~encoding~~ / com encoding / compression
→ it's define based on our used data.
ex: Zerodha, razorpay, etc. used graph chart, so there data is too much.

so, we compressed because there is some limit data for transfer otherwise page issue occurs.

→ Payload Headers : it's just fancy name of our passed data.
→ an example : we passed some of id, name, email,

⇒ There are many types of headers. Security Headers etc....

⇒ most common Headers :

→ Accept : application/json.
Html/text.
- mostly now used app/json due to api's but we can accept html.

→ User-Agent : It's give data that where is come from: means from, mobile, ios, android, mac, postman... basis of it data we got some times suggestion of it app for downloads. etc.

→ Authorization : mostly used for front-end authentication.
ex: token : Bearer xyzPz.....

→ content-type : when type data are send pdf, img, text etc.

→ cookie :- object which store data in key-value pairs.

→ cache-control : for controlling the network data.
(ex: in which time it caches this data)

⇒ CORS Headers :-
- Access-control-Allow-origin : from which origin to allow. like as operation.
- Access-control-Allow-credential,
- Access-control-Allow-Methods etc.

⇒ Security Headers:
- Cross-origin-Embedded-Policy
- Cross-origin-Opener-Policy
- Content-Security-policy
- X-XSS-Protection.

⇒ HTTP methods:
- Basic set of operations that can be used to interact with server.

- GET : for retrieve resource.
- HEAD : No message body response headers only.
- OPTIONS : what actions are available
- TRACE : loopback test
  It most used for debugging to finding which proxy to pass or come from
- DELETE : delete resource
- PUT : replace whole resource.
- PATCH : 11 specific part of resource
- POST : for new resource add.

⇒ HTTP Status code:
X-mean any digit
- 1xx - Informational
- 2xx - Succes
- 3xx - Redirect
- 4xx - Client error
- 5xx - Server Error

101 - continue
102 - Processing
200 - OK
201 - created
202 - accepted
307 - temporary redirect
308 - Permanent "

400 - Bad req
401 - unauthorized
404 - ~~Payment~~ Not Found ~~required~~
500 - Internal ser. Err
504 - Gateway Time out
402 - payment required

11] Routes & Controllers

⇒ Here, all ~~production~~ Theory & production grade setup are done mostly.

⇒ Now, we have write the code to using that all discussed assets.

⇒ In, Controllers folder we write first controller.

→ file in controllers → user.controller.js

⇒ Import the asyncHandler which we made for ~~to~~ error Handling.

import {asyncHandler} from "../utils/asyncHandler.js"

→ make example of register user to see how we write routes & api.

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

export {registerUser}
```

⇒ Routes : Now we, have write routes for that controllers.

→ routes → user.routes.js file.

- import {Router} from "express";
- import {registerUser} from "controllers/user.co"
const router = Router()

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

router.route("/login").post(loginUser)

so, on

export default router

⇒ after made route and controller where we declare it ?

→ so, the best way to declare is in app.js file where all middle ware defined and it also define by middleware like as...

→ Ex, app.js

{ } { }

all other early config middleware (json, static, urlencoded) etc

after it

→

// route import

import userRouter from "./routes/user.ro..

// routes declaration

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

→ standard practice to write api to define version.

→ the declaration define that they pass to user router when "api/v1/users" in the url so pass to user.route.js where further api are /register, /login, /logout etc,

→ whole url makeup like as

http://localhost:8000/api/v1/users/register

12] Logic Building

⇒ we have to make register controller it is problem ? (How we do it ?)

→ So, we do it by the defining algo Ethm (steps)

① get user data from frontend or postman
② Validation - not empty.
③ check email or username already exist
④ check for images, avatar is there ?
⑤ upload them to cloudinary. C.
⑥ create user obj - create entry in db
⑦ remove pass and refToken field from response to send front end
⑧ check for user creation
⑨ return. Es.

⇒ Now, we have to implement all above point in code in user.controller.JS between the below function,
const registerUser = asyncHandler(...

⇒ we just write here the same imp notes instant of whole code

→ As, discuss this is the process that How we decide before writing controller.

→ Eg. body : give the data at front end.

⇒ For files we used multer middleware, when we need we use it use.routes.JS file.

router.route("/register").post(
upload.fields([
{
name: "avatar",
maxCount: 1
},
{
name: "coverImage",
maxCount: 1
}
]),
registerUser
)

The first when we give from and input name & backend

other people we can use according to our requirement

→ So, we added multer in our register controller as middleware.

② ⇒ For Validation we use our ApiError

if (fullName === "") {
throw new ApiError(400, "FullName is required")
}

⇒ Tips for checking error conditional.

```
if ([fullName, email, userName, password]
    .some((field) => field?.trim() === "")
) {
    throw new ...
}
```

(3) ⇒ Checking the user is exist in DB!

```
const existUser = User.findOne({
    or: [{username}, {email}]
})

if (existUser) {
    throw new ApiError(409, "User already")
}
```

(4) ⇒ Handeling files / image

→ when we use multer as middleware it injects req.files object and we can access all details of the image as given below / example below.

→ like req.body give form data multer add in req.file give our file data.

→ so, we handle checking and uploading on cloudinary.

(5)

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

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

if (!avatarLocalPath) {
    throw new ApiError(400, "Avatar file req")
}

const avatar = await uploadOnCloudinary(avatarLocalPath)
const coverImage = await uploadOnCloudinary(coverIm
```

(we check because it req field.)

```
if (!avatar) {
    throw new ...
}
```

(6)

```
const user = await User.create({
    fullName,
    email,
    password,
```

from cloudinary res.url

avatar: avatar.url
coverImage: coverImage?.url || "",
username: username.toLowerCase()
})

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

→ The above query is fit for checking the user created by it's _id with findById methods. more on it give chaining on Select method we give we Not want "" field so it give user data except "-field".

(9) return res.status(201).json(
new ApiResponse(200, createdUser,
"User registered successfully")
)

⇒ So, we can make as all router.

13] ~~Post~~ Postman.

- The Professionally it used by senders the api collection between front-end backend.
- Set up the environment Variable & repeat url to small Variables.
- making collection with endwise folder.
- eg: apa collection → user api folder, product api folder etc.