@Jumedkhan Pathan

Page No.
Date

1] Backend overview :-

⇒ Backend :-

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

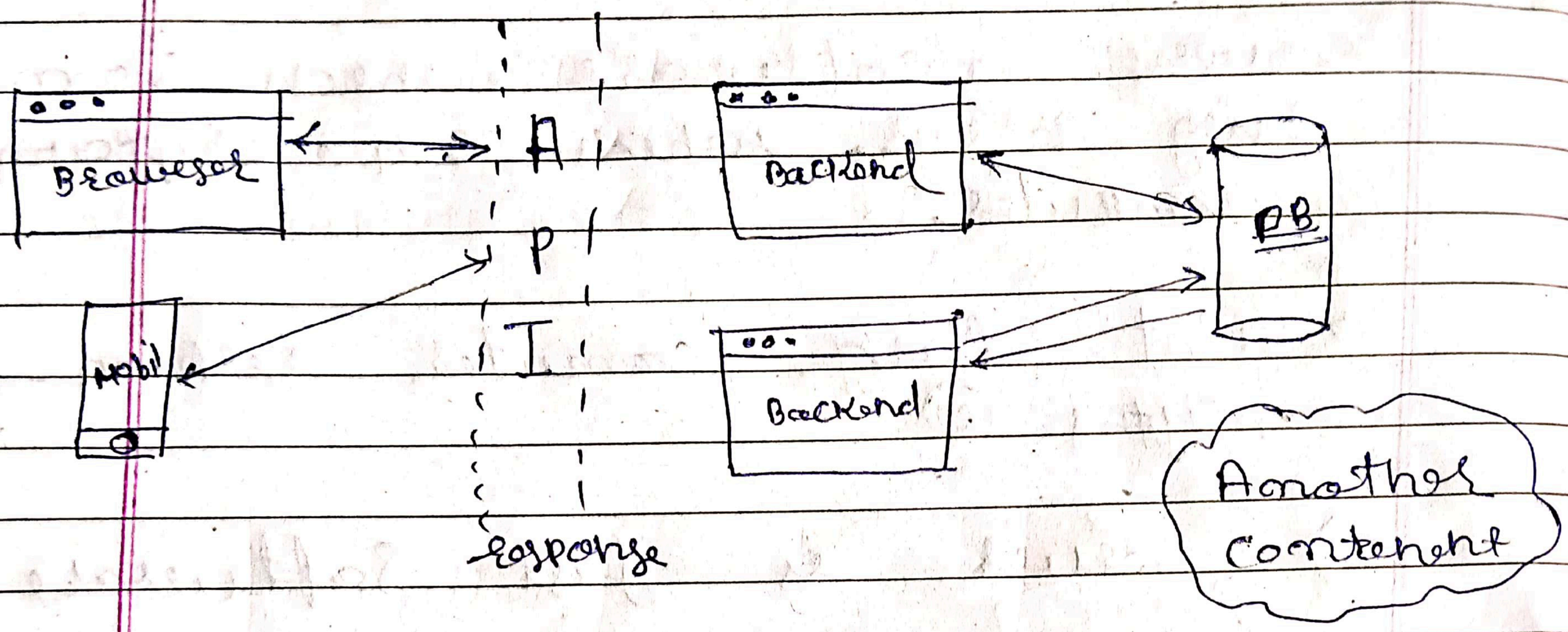
→ Not large computer which myths ☺

⇒ 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 case.	→ ORM, ODM
	exc: mongoose,

⇒ Have 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 only three 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 : Dependencies file manage
→ .env : for storing env variable.
→ etc (Readme, .gitignore, .lint,)

SRC ⇒

- ☐ DB
- ☐ Models
- ☐ controllers
- ☐ Routes
- ☐ Middleware
- ☐ utils
- ☐ More (depends)

File :-

index
↳ DB conn

App
↳ config
↳ cookie
↳ urlencode

constant
↳ enums
↳ DB-name

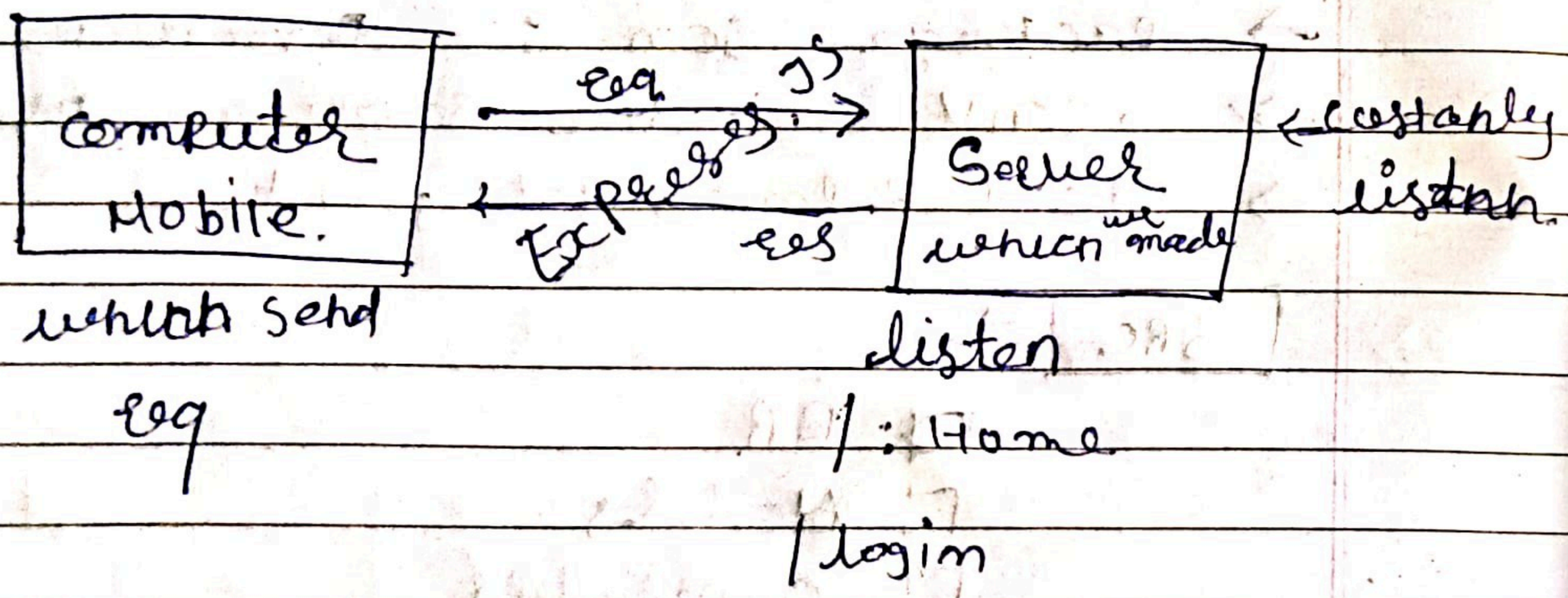
Page No.
Date

2] Have to Deploy.

⇒

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

⇒



⇒ Now, we make Server.

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

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

2] Script:
"start" : "nem /nodemon index.JS"
dev
build
etc.

Page No.
Date

→ 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 Power to use express through app variable.

→ PORT : where our server is listen.
- it just virtual port like physic all port is our comp..
- there is many 65535 port.

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

    res.send("Hello World");
}

app.listen(Port, () => {
    cl("running on{Port} }
```

Page No.
Date

=> PEST Your .env Variable and deploy on your fave. host plat form.

Page No.
Date

3] Connect Backend Frontend.

⇒ there is two ways to using 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 is and dont specify in Package.Json file it gives error.

Tips:- whenever you get JSON data for bette visualization & understand go the JSON formatter and lintly other and post your JSON data that beautify in tree, graph structure. to see a Proffesional way

Page No.
Date

⇒ Toal chain / bundler :-

→ to make react app from Vite, CRA, Parcel etc which ~~convert the~~ make bundler of HTML, CSS, JS which web under stand end of the day.

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

⇒ Make a front-end.

Create Vite @ latest .

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

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

→ we not have to parse data, it do automatically.

cross origin resource sharing

Page No.
Date

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

⇒ CORS Policy origin error :- you ~~[illegible]~~ 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, ~~[illegible]~~ or API white-list.

~~[illegible]~~ ⇒ to install CORS package and ~~[illegible]~~ 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 author 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 are 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.

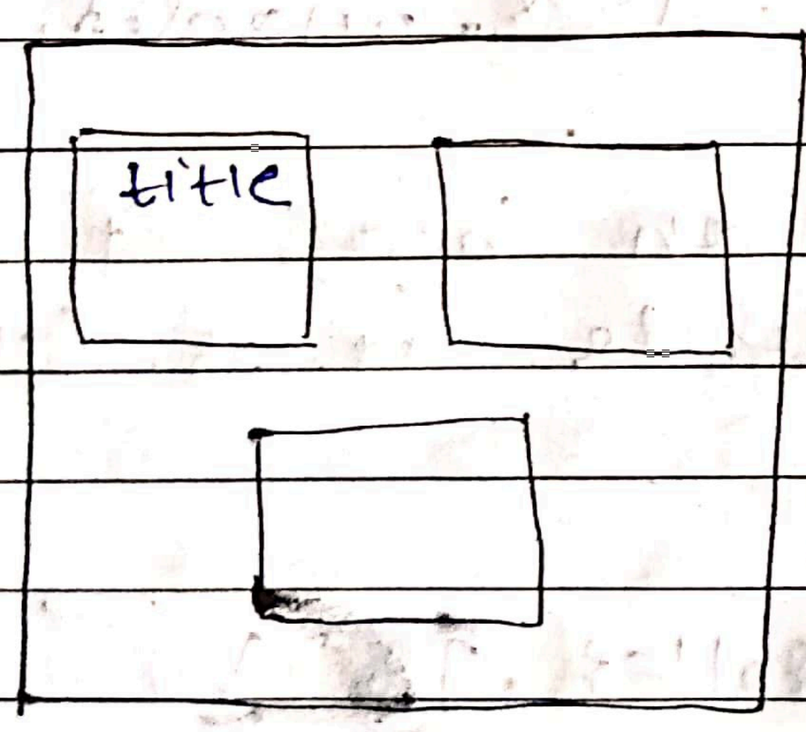
⇒ For Free. eraser.io we can use for data modeling.

TIPS: whenever we some build first we have to decide to How we save data:

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

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

⇒ For example; Google Keep idea of TODOS



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

For inner Todos:
content
markedAsDone
Created date.

Page No.
Date

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

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

⇒ Code phase :-

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

→ It's file but most popular conventions and standerized are as follow.

User.js ✗ User.model.js ✓

→ name convention beoz it is model file so we write it as (Trust best practices)

same

User.controller.js ✓

Page No.
Date

⇒ 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,          username: {
  email: String              type: String,
  Password: String           required: true,
  isActive: Boolean          unique: true
}                            lowercase: true
                             ...more
                           }
```

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

⇒ TimeStamp is most common thing in Schema that's why mongoose made 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.

```
                                          its type
                                          of ref
createdBy: {
    type: mongoose.Schema.Types.ObjectId
    ref: "User"
```

↑ name of ref database schema model name.

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

but

```
subtodos : [
    {
        type: mongoose.Schema.Types.ObjectId
        ref = "subTodo"
    }
]
```

⇒ There is we can also ~~used~~ make schema in the same file.
~~for exa~~ ex: ~~OrderItem~~ 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 standlize approach.

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

→ Folder structure is properly follow by the company according it's requirement but mostly are as discussed & other are follows.

⇒ Prettier: code beauty & 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 all code.

→ It's setting other defined by project manager or head for code consistency.

Page No.
Date

⇒ For Prettier file configuration make file with [.Prettierrc] name and define all requirment configuration.

ex: in file:

```
{
  "singlequote" : false,
  "bracketSpacing": true,
  "tabWidth" : 2,
  "trailingComa": "es5"
  "semi" : true
}
```

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

ex: in file:

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

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

6] How to connect database with in MERN with debug.

① make mongoDb atlas account for cloud ~~to~~ cloud.

Caution:

- always ~~at~~ there is only one machine IP is provided where our backend served.

- 'allow from anywhere' is not good and caution. but ~~a~~ we 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 approch to connect mongoDB.

Page No.
Date

(3) Professional and modular approach where all mongoDB connection func. code are in DB → file and called by the index.JS file.

(2) 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 ~~to~~ 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('${process.env.MONGO_URI}/${DB_NAME}')
```

express event listner error handling

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

// for run on port

```
        app.listen(process.env.PORT, () => {
            console.log("App listening ${PORT}")
        })

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

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

Page No.
Date

⇒ In $\boxed{dB}$ ⇒ 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 connn Failed", error
    process.exit(1);

export default connetDB
```

mongoDB return obj for success connection we can check or see the other it host

process is from node which means terminate process

↳ 0 means without failure 1 means ~~to~~ with failure.

⇒ In main Index.JS file.

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

It break code consistency so we use empty ~~this~~ statement with flag.

Page No.
Date

import ...
import dotenv from 'dotenv'

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

connectDB()

⇒ in package.JSON
nodemon -e dotenv/config --experimental-json-modules src/index.JS "

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

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

⇒ when ever error occure read properly because in backend even .75 extension error are problem. it we not given.

7] Custom API response & error Handling

⇒ Now, we response into server.js file.

after, connection DB file et return promise. some write app listen on his then logic, like as.

⇒ index.js

```
connectDB().
.then( () => {
    app.listen (PORT, () => {
        console... ( "app listen....) }
.catch ( (e) => {
    c.lg ( " Failed.... )
})
```

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

```
import express from 'express'

const app = express();
```

} ← Here all middlewares and other package configs are write which we discuss ahead.

Page No.
Date

→ In between the app.js

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

It is either during resource sharing for authentication & cookie listing.

→ 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"}))

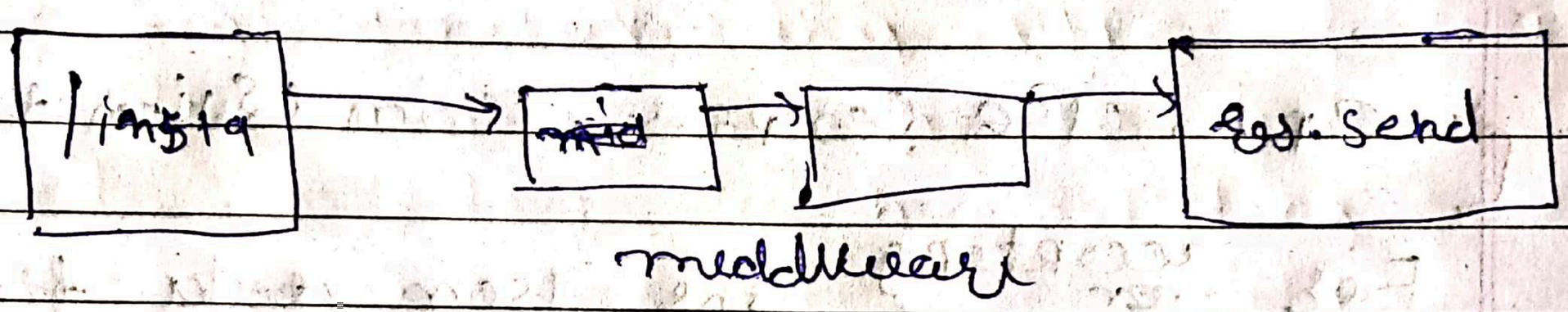
for handling the url data from the user.

Page No.
Date

⇒ Middleware overview :-

→ middleware works is the process between req & res. to performing some task.

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



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

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

→ middleware are always write in sequences.

Page No.
Date

⇒ Utils Folder use of implementation:-

→ In utils all utility fns and ~~code~~ 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 asynHandler = (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) => {
        Promises.resolve(requestHandler(req,
    res, next))
        .catch((error) => next(error))
    }
}
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.

→ 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}
```

Page No.
Date

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

→ for binding the response with the proper manner ~~to~~ response we ~~to~~ write this code as error we made file.

→ In, utils → ApiResponse.js file.

```
class ApiResponse {

constructor (statusCode, data, message =
"sucess !!").
{
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 in 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.

Page No.
Date

⇒ mongoose-aggregate-paginate-V2 :-

- it's allows us to make a the aggregate query efficently 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-aggre....";

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.

Page No.
Date

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

⇒ JWT (JSON web tokens) :
- It ~~allows~~ ~~us to~~ 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~~

~~User.Schema.pre("save", async function(next) {
this.password~~

Page No.
Date

⇒ 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

UserSchema.pre("save", asymc function(
next) {

~~if (this.Password~~

Parallel checking isModified property

→ if (!this.isModified("password")) ~~return~~ 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.methods as like below.

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

Page No.
Date

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

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

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

⇒ JWT :-

- JWT is bearer token.
- means who have this token it would be right. (જેને પાસ્વર્ડ તે બરાબર)

→ .env are define secret key as like below.

ACCESS_TOKEN_SECRET = 0T4XPWZ.....UXPNX
ACCESS_TOKEN_EXPIRY = 1d
REFRESH_TOKEN_SECRET = CJKTOU.....PTXNY
REFRESH_TOKEN_EXPIRY = 10d

→ So, as above we write ~~to meth~~ fun in method for bcrypt same we writ for our Jwt token.

Page No.
Date

```
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 is only we pass _id

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

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

⇒ 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.

Page No.
Date

→ 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 its production grade and recommended way in industry for attaining 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 ~~our~~ local ~~local~~ ~~server~~ ~~file~~ file which are in upload the local server.

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

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

→ unlink :

- when we want to 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 mod module method.

→ In, file code...

```
import fs2 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 .V.2.uploader.upload method but,

→ we make it some additional to more efficent and for unlink the file from local.

→ So, for that we make func. as...

Page No.
Date

```
const uploadOnCloudinary = async (localFilePath) => {

try {
    if (!localFilePath) return null

// File upload on the cloudinary.

const response = await cloudinary.uploader.upload(localFilePath, {
        resource_type : "auto"
    })
return response;
```

→ For any format and jpg, video etc

// we also here remove the file after uploading

```
} catch (error) {
    fs.unlinkSync(localFilePath)
    return null;
}
}
```

There is also same property.

remove uploaded file if the upload is failed.

```
export {uploadOnCloudinary}
```

Page No.
Date

⇒ Multer middleware writing code:

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

→ In [middleWare] 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

Page No.
Date

10] HTTP crash Course

→ HTTP | HTTPS : Hyper text transform protocol | 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 interest.

⇒ Headers :-

→ It is metadata. means keep the data about data.
→ Headers are sent along with req & res.

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

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

→ authentication : are 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 earlier 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 that some specific rule of success data (200 code) not found (404) etc. for standardize.

→ Representational Headers : ~~encoding~~ / comm encoding / compression

→ it's define based on our used data. ex: Zerodha, razorpay, etc used graph chart, so there data is too much.

Page No.
Date

So, we compressed because there is some limit data for transfer otherwise large 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 get some times suggestion of it app for downloads. etc.

Page No.
Date

→ Authorization : mostly used for front-end authorization.
ex: token : Bearer xyVPZ.....

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

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

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

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

⇒ Security Headers:
- cross-origin-Embedded-Policy
- cross-origin-opener-Policy
- content-security-policy
- X-xss-protection.

Page No.
Date

⇒ 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 options are available
- TRACE : loopback test
- It most used for debugging for finding which proxy to pass or come from

- ~~Delete~~ DELETE : delete resource
- PUT : replace whole resource.
- PATCH : " specific part of resource
- POST : for new resource add.

⇒ HTTP Status Code:

x-mean 2 digit

- 1 xx - Informational
- 2 xx - Succes
- 3 xx - Redirect
- 4 xx - Client error
- 5 xx - Server Error.

Page No.
Date

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 ~~[illegible]~~ Theory & production grad setup ~~are~~ done mostly.

- ~~Now~~ Now, we have create the code to using that all ~~a~~ early discussed assets.

⇒ In, Controllers folder we write first controller.

→ file in | controllers | → user.controller.js.

→ Import the asynHandler which we made for ~~to~~ error Handling.

import {asynchandler} from "../utils/asy.

→ 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.

→ $\boxed{\text{routes}}$ → user.routes.js file.

- import {Router} from "express";
- import {registerUser} from "control/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 middle ware like as...

→ In, app.js

{ { } }

all other early config middlewares (JSON, static, urlencoded) etc

Page No.
Date

after ut

→ // route import

import userRouter from './routes/user.route';

// 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 ~~route~~ when "/api/v1/users" in the url so ~~ut~~ pass to user.route.JS

where further api are /register, /login, /forgot 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 algorithm (STEPS)

1. get user data from front-end or postman
2. Validation - not empty.
3. check email or username already exist
4. check for images, avtar 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~~ to send front end
8. ~~return~~ check for user creation
9. return res.

⇒ 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.

→ Req. body i give the data at front end.

→ For files we used multer middleware which we mad use it use routes. JS file.

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

The field when we give frontend input name & backend

there are other Profile 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, "Full Name required")
}
```

Page No.
Date

⇒ Tips for checking error, conditional.

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

(8) ⇒ Checking the user is exist in DB?

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

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

(9) ⇒ Handeling files / image :-

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

→ like req.body give form data multar add in req.file give one file data.

→ so, we ~~handle~~ ~~data~~ ~~as~~ checking and uploading ~~on~~ on cloudinary.

(5) const avatarLocalPath = req.files?.avatar[0]?.path;

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

we check because it required field.

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

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

If (!avatar) {
throw new ...
}

(6) const user = await User.create({
fullName,
email,
Password,

from cloudinary res.url

avatar: avatar.url
coverImage: coverImg?.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 man~~ Postman.

- The professionally it used by sending the api collection between front-end backend.

- Set up the environment Variable & repeating url to make Variables.

- making collection with nested folder.
- ex apa collection → users api folder, product api folder etc.