13/sep/2024

Day-1

Page No.:

Date: / /

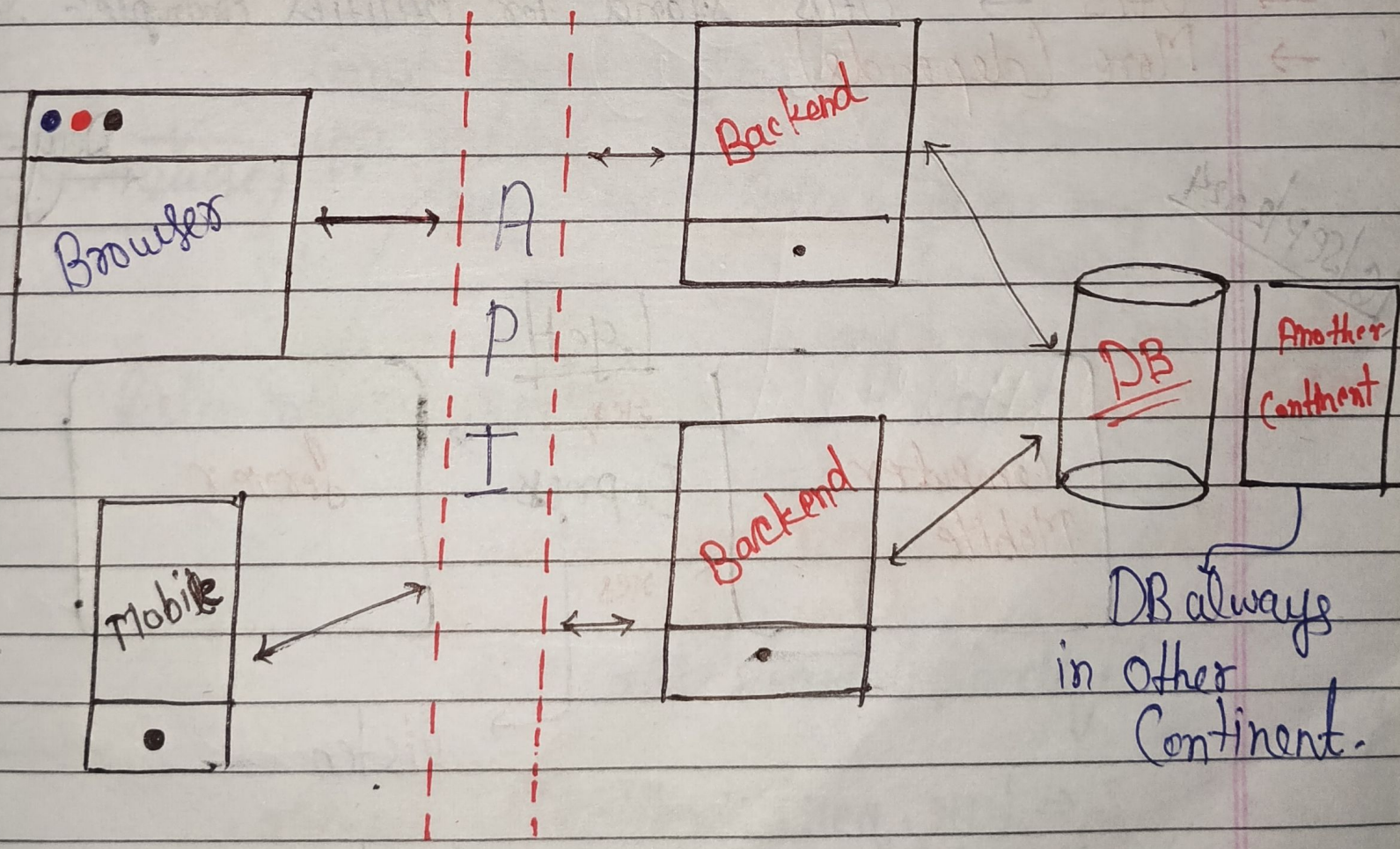
RoadMap Backend

Backend Development.

⇒ 2 major Components:-

A Programming language — A Database

- Java, Js, PHP, golang, c++
  - ↳ One framework
- Mongo, MySQL, Postgres, sqlite
  - ↳ ORM, ODM



⇒ A Javascript based Backend

Data file — Third Party (API)

→ A Js Runtime : Nodejs | Deno | Bun

Page No.:
Date: / /

(2)

Package.json .env (Readme, git, lint, Prettier

→ Src

index → DB Connects

APP → Config
Cookie, urlencode

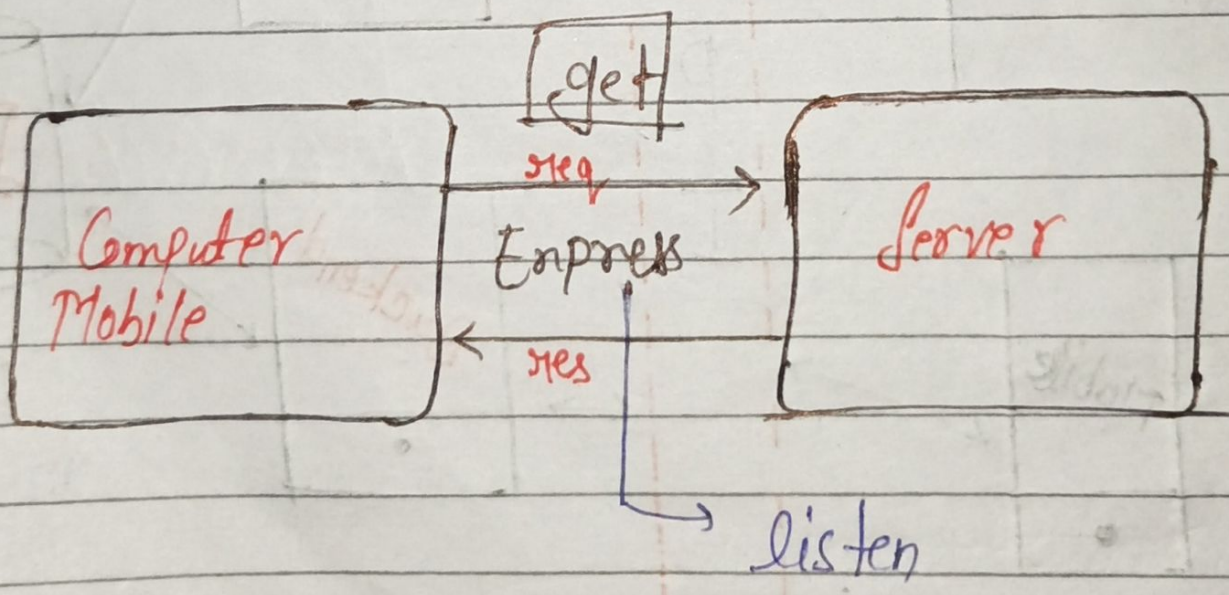
Constants → enums, DB-name

Directory Structure

→ DB → actual code which connect the database.
→ Models → create structure to keep Data.
→ Controllers → It is a functionality.
→ Routes → Routes mean that slash (/) like: (/signup, /lo
→ Middle wares
→ Utils → Utils stand for utilities Example:- Send Email up
→ More (depends)

Day-2

16/sep/2024



/ : home route

/login : login setup

③

Note :- const express = require('express') → common js
import express from "express" → module.js → different

Page No.:
Date:

the difference is styling of js.

etc) Let's make a server.
So simply let's go to the express js and copy the code and paste it in js file in VS. code.

file ⇒ index.js

//code :-

require module System

require('dotenv').config() //from dotenv

Const express = require('express')

Const app = express()

Using Express we created a variable 'app'

Const port = 4000 //It's listen on 4000 and run on my server

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

get request

listen on Home route

So lets go to our my browser and write localhost:4000

if we get any request, we will send Hello world! in response
It's call request handling

app.get('/twitter', (req, res) => {
res.send('Khushidat.com')
}) //listen on :- localhost:4000/twitter

Note :-
Install Express.js.
Open terminal and write npm i express

```
app.get('/login', (req, res) => {
    res.send('<h1> Please login at github </h1>')
})

// we can use html tags
// listen on :- localhost:4000/login

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

↳ evaluate and embed expressions dynamically in template literals.

⑤ 17/Sep/2024

Day-3

Page No.:
Date: / /

→ How to connect frontend and backend in Javascript / Fullstack Proxy and CORS

⇒ We have two folders → Backend
→ frontend

- first make a server in Backend.

file:- Server.js :-

⇒ import express from "express"; → module.js
↳ imports the Express framework to build the server

const app = express(); ↳ creates and Express application instance to handle server and routing.

app.get('/', (req, res) => {
res.send('Server is ready');
}); // handles get requests to the root URL (/) and responds with 'Server is ready'.

app.get('jokes', (req, res) => {
. . .
}); // Handles GET requests to /jokes and responds with a list of jokes in JSON format.

const jokes = [...]; // Define an array of joke objects
res.send(jokes); // sends the list of jokes as a respons

Page No.:
Date: / /

Const Port = process.env.PORT || 3000;

↳ sets the port for the server.
Uses environment variable PORT (For Prod...
or defaults to 3000 (for local develop...

app.listen(port, () => {
    console.log(`Server at https://... ${port}`);
}); // starts the server on the specified port and logs a message with the server URL

Routs Summary

- / Returns "server is ready".
- /jokes Returns a list of jokes in JSON formate.

- After creating the server in the Backend let's now write the code in the frontend to connect both together.

...er: src/App.jsx :-

import { useEffect, useState } from 'react'; ← These hooks manage and fetch.

Initialize an ~~arr~~ empty array to store jokes from the backend.

Automatically runs when the component loads to make the api call.

```
import axios from 'axios';
function App() {
  const [ jokes, setJokes ] = useState([]);
  useEffect(() => {
    axios.get('/api/jokes');
```

↳ empty array

// Proxy (handled in vite config) forwards this to the backend at http://localhost:3000/api/jokes

Makes an API request to /api/jokes.

```
    .then((response) => {
      setJokes(response.data)
    });
```

↳ Updates the state with the list of jokes from the Backend

```
    .catch((error) => {
      console.log(error)
    });
  });

  return (
    <>
      <h1>Welcome to My Fullstack Journey</h1>
      <p>JOKES: {jokes.length}</p>

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

Page No.:
Date: / /

// loops through the jokes array and displays each joke's title and content on the page.

{jokes.map((joke, index) => (...))}

↳ Rerenders each joke dynamically.

))

}

)

}

export default App;

CORS :- (Cross-Origin Resource Sharing) error occurs when a web application tries to access data from a different domain without the correct permissions.

Backend Communication (vite proxy)

vite configuration (vite.config.js):

- Proxy Setup :- server.proxy : This is used to avoid CORS issue by proxying the frontend API requests to the Backend.
- '/api' → http://localhost:3000 : Any request starting with /api from the frontend will be forwarded to the Backend running on http://localhost:3000.

⇒ How Frontend and Backend are connect

Quick Summary

Step-1 Frontend makes a request to /api/jokes in App.jsx.
Step-2 vite proxy intercepts this request and forward to the backend (running on localhost:3
Step-3 Backend (Express server) responds with a list of jokes.
Step-4 Frontend receives the jokes and renders them on the page.

(9) 1sept2024

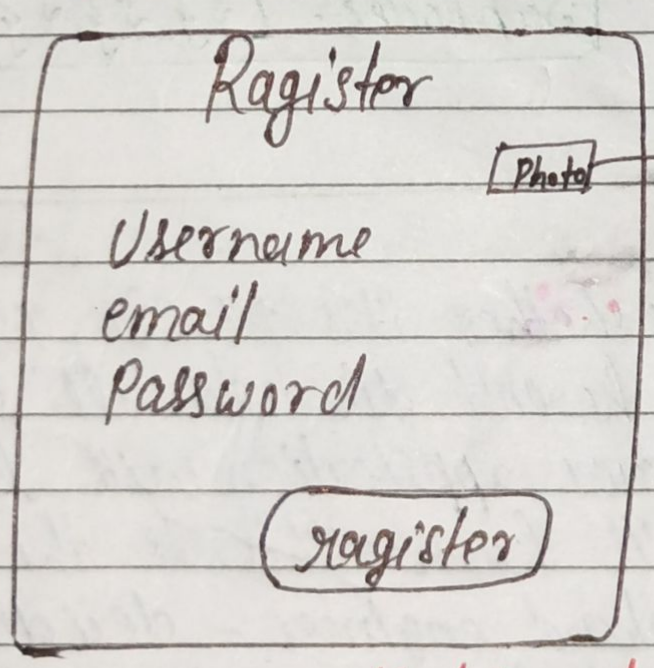
Day-4

Page No.:
Date: / /

→ Data modelling for backend with mongoose

#Note!- Where the data is stored is not important, but what data being stored is important. So, which data points can be important, like; username, password, date of birth .etc.

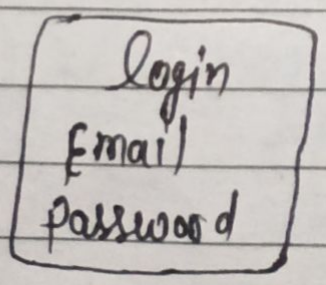
For Example:- We have two options to create a login field or a register field, So the main question is which field to create first. We need to consider how how the data will be saved Login:- login is data validation to check if the data you entered matches the data in the database or not. So first we create register field here.



Now, if even one more field is added like: Photo, the entire structure will change, and the logic will change as well.

→ that's why it's important to consider what information to take from the user and store in the data, and then all the work be based on that.

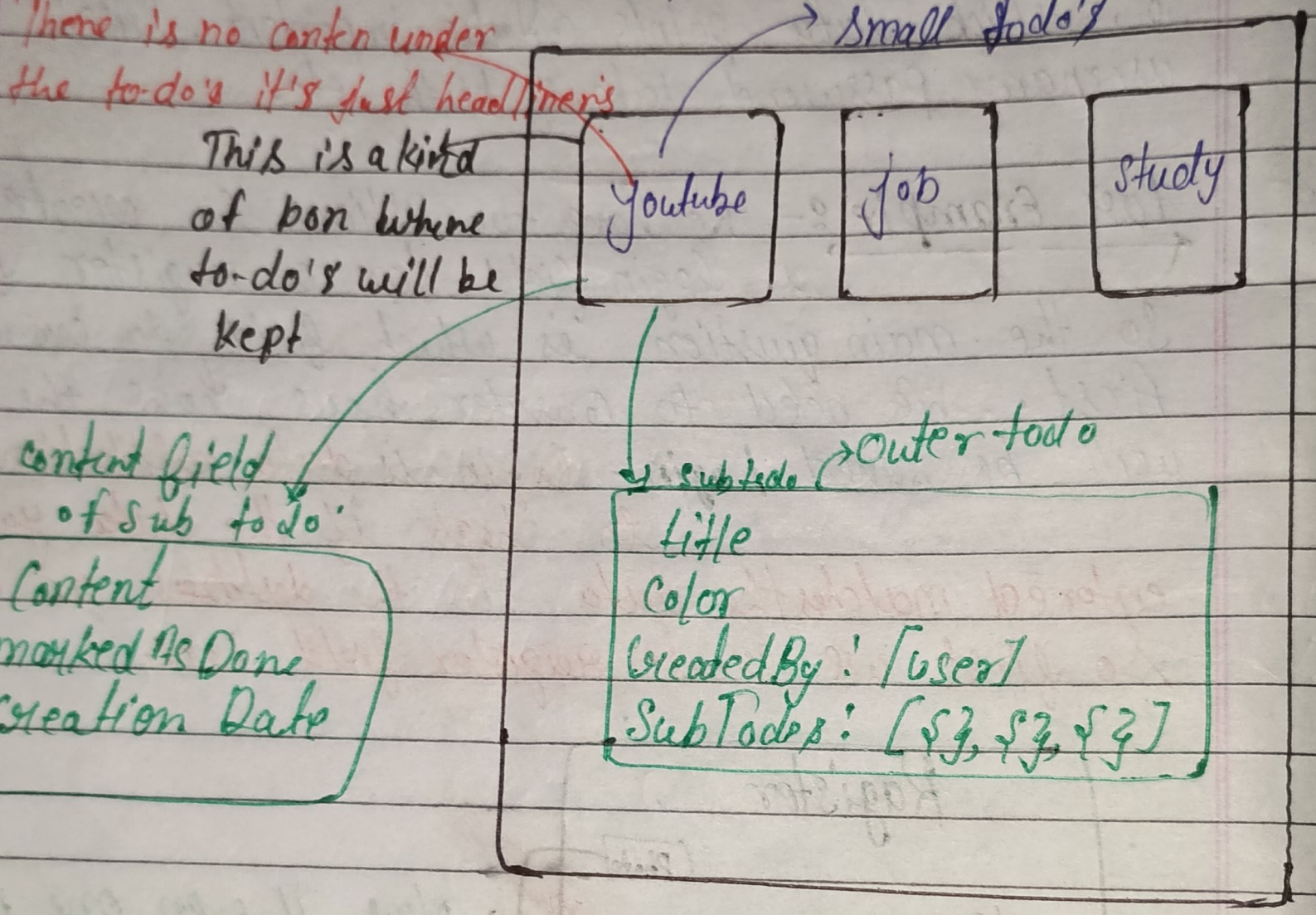
(host!3000)



Page No.:
Date: / /

Structure of todo:

→ first, let's think about what all we are going to store.



We haven't decided yet whether it's MySQL, MongoDB, or what it is. We haven't discussed all that; it's just about how our application will look, what structure it will have. This is the most important role of backend engineer - deciding what data to store, how to retrieve it, and how to validate it. These are things for later.

Note: userSchema defines the structure of the data for the user collection in MongoDB

(11)

→ Let's move on the practical

user.model.js
~~Sub.todo.model~~

import mongoose from "mongoose"

Mongoose is a library used to connect and interact with MongoDB in Node.js.

↳ Mongoose helps to create Schema

const userSchema = new mongoose.Schema({})

↳ Schema is a method which takes an object.

↓

we can't export it like:
(export (const userSchema)

there is a way to export it
→ It is Schema (Schema of mongoose)

↓

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

Model is a Method
It takes two Parameters
- what model to build
- and on what basis to build it.

Interview based question → As soon as the User enters the database, this model converts from to plural and become lowercase, like: "users"

Best approach

We know that Schema takes object

```
firstObject new mongoose.Schema({
    username: {
        type: String,
        required: true,
    },
```

Note:- one field More

createdBy : {
type : mongoose.Schema.Types.ObjectId,
ref: 'user',
}

19/sep/2024

email: {
type : String
" "
" "

Password: {
type: String
required: [true, "password is required"]
}, we can pass messages
and it's accept (array)

Secondary object
{ Timstamps : true }
→ createdAt : a date representing when this document was created.
→ updatedAt : a date representing when this document was last updated.

#

Why

Orde

Choice

Page No.:
Date: / /

Day - 5

→ Ecommerce And Hospital management Data Modeling

Ecommerce Data modeling

```
const CategorySchema = new mongoose.Schema({}, {timestamps:true})
export const Category = mongoose.model("Category", CategorySchema)
```

it's not standard approach
categories ✗ Not standard approach

if we use "categories" instead of "Category" here the data will still be stored in the database under the name "Categories". Mongoose automatically detects whether it is singular or plural. So No extra "s" will be added at the add.

Category ✓
categories ✗

Why → The reason why it is necessary to use "Category" instead of "categories" here is because we provide references. When we give references to other Models, using. "categories" in the plural form has never been a standard practice.
So solely for this reason, we use the singular form with an uppercase letter

Order Status

```
# mongoose.schema ({
    status: {
        type: String,
        enum: ["PENDING", "CANCELLED", "DELIVERED"],
        default: "PENDING"
    },
});
```

Choices ← we can choose in these three options
spelling mistake ✗
otherwise the value will not go to the Status
required field but Min

(17)

20/sep/2024

Day 6

What is the difference between dependencies and dev dependencies.

→ Dev dependencies are dependencies that we use during development and do not take them to production because they are only used in development.

dev dependency is a development dependency.

⇒ Middlewares:- The code we want to run between, where we receive a book request, and the server fullfills it, but if we want to add some checks before that, we can do that inside middleware.

for example:- you receive a request and are asking the server for some information, but we will put a middleware in between to check if you are eligible to receive that information by asking for your cookies first.

(15)
21/sep/2024

Day-7

Note :- 2 important things

(i) When we try to interact with the database, ✓ issues can arise, which means that wrapping the code in a try-catch block is a much better approach, or we can use promises as well so, we definitely have to choose one of the two approaches.

(ii) Database is always in another continent So, whenever you communicate with a database, it takes time, The main point is that you must use async/await. It's always a better approach that whenever you interact with the database, you use try-catch and keep async/await in mind.

```
import express from "express"
const app = express()

(async () => {
    try {
        await mongoose.connect(`${process.env.MONGODB_URI}/${DB_NAME}`)
        app.on("error", (error) => {
            console.log("ERR", error);
            throw error
        })

        app.listen(process.env.PORT, () => {
```

IIFE a function that execute as soon as it's called or invoked in the event loop

Bad consistency

```
//require('dotenv').config({path: './env'})
import dotenv from "dotenv"
import connectDB from "./db/index.js";

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

connectDB()
```

and all code in db folder in ~~inde~~ index.js file.

CORS:- Cross origin resource sharing!

Page No.:
Date: / /

```
console.log(`App is listening
on port ${process.env.PORT}`);
})

} catch (error){
    console.error("ERROR:", error)
        throw err

    }
})()
```

22/sep/2024

Day - 8 → Custom api response and Error handling

Asynchronous method:- When an ashynchronous method complete .technically it also returns a promise.

app.use() :- We use App.use when we ~~configure~~ need to apply middleware or configure settings.

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

// And the CORS_ORIGIN is set here so that it doesn't need to be change every where

There is an option called "origin" that specifies which origins you are allowing.

And usually this origin is allowed using this

credentials: true;

allowing Cookies.

(17)

Parse incomming JSON request.
The limit of 16kb restricts the size of JSON payloads.

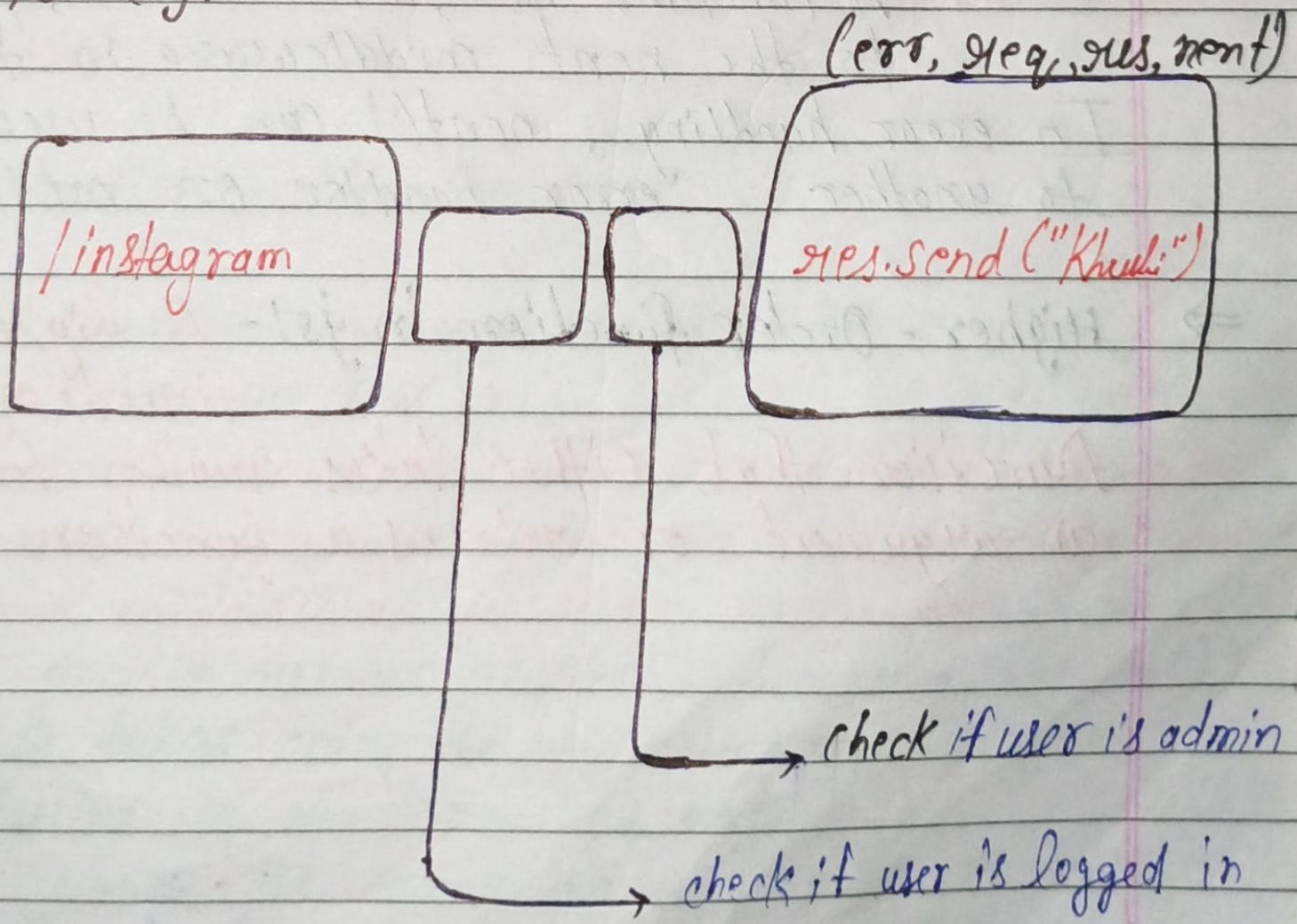
app.use(express.json({limit: "16kb"}))
app.use(express.urlencoded({extended: true, limit: "16kb"}))
Parse URL-encoded data (like form submission) with a limit of 16kb and extended set to true, allowing nested objects.

app.use(express.static("Public"))

static
↓
serves static files (e.g., HTML, CSS, image) from the Public folder

app.use(cookieParser())
↓
Middleware that parses cookies from the HTTP request, making them easily accessible in your server.

What's the exact meaning of middleware:
for e.g:-



so that
ery where

middleware functions that takes (err, req, res, next) as parameters are error-handling middleware.

→ err:→ Represent the error object if an error occurs, it's passed here for handling.

→ req (request):→ Represent the HTTP request from the client. Contains data like headers, parameters, and body.

→ res (response):→ Represents the HTTP response that will be sent back to the client.

→ next:→ A function that, when called, passes control to the next middleware in the stack. In error handling, next() can be used to move to another error handler or middleware

⇒ Higher-Order function in js:- A higher order function is a function that either takes another function as an argument or returns a function as it's result.

(19)

asyncHandler with try.catch.

Higher-order function.

```
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
        })
    }
}
```

Arrow function return asynchronous function.

// asyncHandler → a wrapper function that handles asynchronous operations in route handlers.

- asyncHandler = (fn) → A function (asyncHandler) that takes another function (fn) as an argument.
- This is a higher order function that return another function.

asyncHandler with Promises!

```
const asyncHandler = (requestHandler) => {
    (req, res, next) => {
// This is the returned function from asyncHandler. It takes the usual Express.js request (req), response (res), and next middleware function as argument.
        Promise.resolve(requestHandler(req, res, next))
// It ~~takes~~ runs the requestHandler function, passing in req, res, and next.
// It wraps the result in a promise, ensuring the request handlers return value is treated as asynchronous.
```

Date: / /

- catch ((err) => next(err))

// if the request Handler throws an error or rejects a promise, this catch will catch the error and pass it to the next middleware using next(err)

Status Code

Information response (100 - 199)

Successful responses (200 - 299)

Redirection messages (300 - 399)

Client error responses (400 - 499)

Server error responses (500 - 599)

} # Best Practices

3/Sep/2024

Page No.:
Date: / /

Day- 9

For user → User and video model with hooks and JWT.

#

Just like MongoDB automatically generates a unique ID for the user, It saves data in BSON formate rather than JSON. That's why we don't include the ID within the user's data → BSON ✓
JSON ✗

Mongoos-aggregate-paginate:-

is a plugin that enables pagination for Moongoose aggregate quiries, allowing you to break large datasets into manageable pages. It's impromes performance and enhences user experience by making data navigation smoother in applications.

- importing the plugin →

```
import mongooseAggregatePaginate from
"mongoos-aggregate-paginate-v2"
```

- Imports the plugin for pagination with aggregate queries.

- Using the plugin →

```
videoSchema.plugin(mongooseAggregatePaginate);
```

- Attaches the pagination capability to the videoSchema

Page No.:
Date: / /

Key Concept :-

Mongoose :- A popular ODM (object Data Modeling) library for MongoDB and Node.js, simplifying database interactions.

Aggregate Queries :- These are used to process data records and return computed results. They can perform operations like filtering, grouping, and sorting.

Pagination :- It helps manage large ~~database~~ datasets by diving them into pages, making it easier to handle and display data.

Bcrypt library :- A library to help you hash passwords

JSON web Token (JWT) :- JWT is a compact, URL-Safe token format for Securely transmitting information between parties.

Structure

- Header : Contain token type (JWT) and signing (eg. HMAC SHA256)
- Payload : Hold claims (user data, permissions)
- Signature : Ensure authenticity by signing

Usage:-
→ Authentication: After login, a JWT is issued to the client for accessing protected resources.
→ No need to store session data on the server.

Advantages
→ Compact! Easily transmitted in URL or header
→ Self-Contained: Reduce database queries by storing user info in the token.

Security:- use HTTP and keep the secret key secure to prevent forgery

userschema.pre("Save" async function (next){

if (!this.isModified("Password"))
return next()

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

A pre-save middleware function for hashing a user's password before saving it to the database

Function Definition:-
This defines a middleware that runs before the save operation on the userschema

Condition Check: Checks if the password field has been modified. if not, it skips the hashing process and proceeds to the next middleware

Key point:- • Ensures passwords are securely hashed before storage.
• Prevent unnecessary hashing, if the password hasn't changed.

Password Hashing :-
this.password = bcrypt.hash(this.password, 10)
uses bcrypt to hash the password with a salt round of 10.

next Middleware:- next():
• calls the next middleware in the stack after the password has been hashed

4/Sep/2024

Day-10

⟶ How to upload file in backend
Multer

⇒ Whenever we uploaded a file, there are always 2 packages involved in the backend, and we choose on of them technically.
→ Express-fileupload
→ Multer

So the first thing we will do is allow the user to upload a file. We use Multer to upload files;
Cloudinary is a service, Not a direct file upload method.

First step
1. using Multer, we will take the file from the user and temporarily store it on our local server.

Page No.:
Date: / /

Next Step

2. After that we will use Cloudinary to take the file from local Storage and then upload it to the server.

fs (File-System):- is a built in node.js module that allows you to interact with the file system on your server. It provides methods to read, write, delete and manipulate files and directories, commonly used for operations like creating files, reading or updating file contents etc.

unlink:- is a method provided by the fs module, and it is used to delete a file from the file system. It takes the file path and a callback function, removing the file asynchronously.

upload a file to Cloudinary and delete the local file after successful upload

Cloudinary is a cloudbased service that provides an easy way to upload, store, and manage images and videos.

Cloudinary Configuration:

```
cloudinary.config({
    cloud_name: process.env.CLOUDINARY_CLOUD_N
    api_key: process.env.CLOUDINARY_API_KEY,
    api_secret: process.env.CLOUDINARY_API_SECRET
});
```

Page No.:
Date: / /

- Configures Cloudinary using environment variables (process.env) for security purposes
- Cloud_name, api_key, and api_secret are required to authenticate and connect to your Cloudinary account.

⇒ Update function:-

Const uploadOnCloudinary = async (localFi...
=> {...}

↳ this is an asynchronous function that takes localFilePath (the Path of the local file to be uploaded) as an argument.

⇒ Try-catch Block

if (!localFilePath) retu... null;

↳ if the localFilePath is not provided or is nul... the function returns nu...

- Const response = await cloudinary.uploader.upload (localFilePath, { resource_type: "auto" })
  ↳ uploads the file to Cloudinary. the resource_type: "auto" allows the upload of any type of file (image, video, etc).
- fs.unlinkSync (localFilePath);
  ↳ Deletes the local file after it has been successfully uploaded to save storage space.
- return response;
  ↳ Return the response from Cloudinary, which includes information like the uploaded

Page No.:
Date: / /

⇒ Catch Block:

- fs.unlinkSync(localFilePath);
  ↳ if an error occurs during the upload, the local file is still deleted to prevent unused temporary files from being left on the server.

- return null;
  ↳ Returns null to indicate that the upload failed.

Why use use this code

→ File Management:- uploads file to to cloudinary for secure storage and easy access.
→ Resource optimization:- Deletes the local files after upload to save server storage.
→ Error Handling:- Ensure the local file is deleted even if the upload fails. Keeping the system clean.

Multer middleware:

```
import multer from "multer"
const storage = multer.diskStorage({
  cb(null, "./public/temp")
},
  filename: function (req, file, cb) {
    cb(null, file.originalname)
  }
})
export const upload = multer({
  storage
```

multer::
Storage Configuration: Defines how and where the uploaded files will be stored.
destination: specifies the folder ./public/temp where the file will be temporarily sa
filename: specifies that the uploaded file will retain its original name

upload. Export → Export a configuration multer instance with the custom storage settings.

Why we use this code?

*. File Handling: To manage file uploads from users by temporarily storing them locally

• Customization: Allows control over the file's destination and name before further processing

Diskstorage:-

• Storage files on disk (local file system) with customizable options like the destination and file name.
• Useful when you need files to be saved and accesible on the server, such as for later processing or refrence.

Memorystorage:

• Stores files in memory as buffers
• Useful for temporary storage where you don't need to save files to disk. Typically used when files need to be processed immediately (e.g. image processing).

Note :- Both are the storage engines by multer. for handling files uploads in different scenarios