**L-8:**

**app.js**

Here's an explanation for each line:

import express from 'express';: Imports the Express framework, which simplifies the process of creating a web server and handling HTTP requests.

import cors from 'cors';: Imports the CORS (Cross-Origin Resource Sharing) middleware, which enables cross-origin requests.

import cookieParser from 'cookie-parser';: Imports the cookie-parser middleware, which parses cookies attached to incoming requests.

const app = express();: Creates an instance of the Express application.

app.use(cors({ ... })): Configures CORS middleware to handle cross-origin requests. It specifies the allowed origin from the environment variable CORS\_ORIGIN and allows credentials (cookies) to be sent with requests.

app.use(express.json({ limit: '16kb' }));: Configures middleware to parse incoming JSON data with a limit of 16 kilobytes.

app.use(express.urlencoded({ extended: true, limit: '16kb' }));: Configures middleware to parse incoming URL-encoded data with the extended option and a limit of 16 kilobytes.

app.use(express.static('public'));: Configures middleware to serve static files from the "public" directory. This line is typically used to serve images, stylesheets, or other static assets.

app.use(cookieParser());: Configures middleware to parse cookies from incoming requests.

export { app };: Exports the configured Express application to be used in other files.

These lines collectively set up a basic Express application with middleware for handling CORS, parsing JSON and URL-encoded data, serving static files, and parsing cookies.

**ApiError.js**

Explanation:

class ApiError extends Error { ... }: Defines a custom error class ApiError that extends the built-in Error class.

The constructor of ApiError takes four parameters:

statusCode: HTTP status code to be sent in the API response.

message: A default error message if none is provided.

errors: An array of error details or additional error information.

stack: Stack trace information for debugging purposes.

Initializes properties such as statusCode, data, message, success, and errors based on the provided or default values.

The if (stack) { ... } else { ... } block checks if a stack trace is provided. If not, it captures the stack trace using Error.captureStackTrace.

export { ApiError };: Exports the ApiError class for use in other files.

ApiError.js: This file provides a custom error class ApiError that can be used to create consistent error responses in your API. It allows you to specify the HTTP status code, error message, and additional error details.

**ApiResponse.js**

Explanation:

class ApiResponse { ... }: Defines a class ApiResponse for structuring API responses.

The constructor of ApiResponse takes three parameters:

statuscode: HTTP status code to be sent in the API response.

data: The actual data to be sent in the response.

message: A default success message if none is provided.

Initializes properties such as statuscode, data, message, and success based on the provided values.

The success property is set based on whether the status code is less than 400, indicating success.

ApiResponse.js: This file provides a class ApiResponse for structuring successful API responses. It helps create a standardized format for sending data back to clients, including the HTTP status code, data payload, and success message.

**asyncHandler.js**

Explanation:

const asyncHandler = (requestHandler) => { ... }: Defines a function asyncHandler that wraps an asynchronous request handler with error handling.

The function takes requestHandler as a parameter, which is assumed to be an asynchronous function.

The inner function (req, res, next) => { ... } is the actual middleware function that wraps the requestHandler.

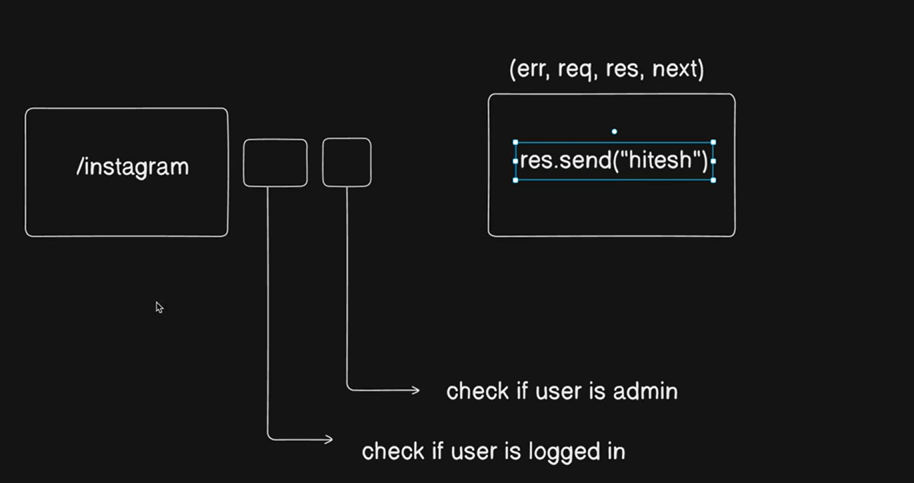
Promise.resolve(requestHandler(req, res, next)): Calls the requestHandler and ensures that it returns a Promise.

.catch((err) => next(err)): Catches any errors that occur during the execution of the requestHandler and passes them to the Express next function to be handled by error middleware.

export { asyncHandler };: Exports the asyncHandler function for use in other files.

**asyncHandler**.js: This file provides a middleware function asyncHandler that simplifies error handling for asynchronous route handlers. It ensures that any errors thrown within the asynchronous function are properly caught and forwarded to Express's error-handling middleware.

These utility files contribute to a cleaner and more organized codebase by encapsulating common functionality that can be reused across different parts of your application.



**L-9:**

**userSchema.pre('save', async function (next) {**

**if (!this.isModified('password')) return next();**

**this.password = await bcrypt.hash(this.password, 10);**

**next();**

**});**

This block of code is a Mongoose middleware that runs before saving a user instance to the database. Specifically, it is a pre-save hook (pre('save', ...)) that executes some logic before the user document is saved. Let's break down each part:

1.Condition Check:

**if (!this.isModified('password')) return next();**

This line checks if the password field of the user document has been modified. If the password hasn't been modified, it means that some other field is being updated, and there's no need to re-hash the password. In such cases, the middleware exits by calling next() to proceed with the save operation without any further action.

2.Password Hashing:

**this.password = await bcrypt.hash(this.password, 10);**

If the password has been modified, this line hashes the password using the bcrypt hashing algorithm with a cost factor of 10. The hashing is done asynchronously using the bcrypt.hash function, and the result is assigned back to the password field of the user document.

3.Call to Next Middleware or Save:

**next();**

Finally, this line calls the next() function, allowing the middleware to proceed to the next middleware in the chain or to the actual save operation if there are no more middlewares. This is essential to ensure that the Mongoose middleware pipeline continues to execute.

In summary, this pre-save middleware ensures that the user's password is hashed before saving it to the database, but only if the password field has been modified. This is a common practice for securing user passwords in a database by storing only their hashed values rather than plaintext.

This method, generateAccessToken, is part of the Mongoose schema and is designed to generate an access token for a user. Let's break down the code:

**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,**

**}**

**);**

**};**

**Syntax: jwt.sign(payload, secretOrPrivateKey, [options, callback])**

The jwt.sign function is from the json-webtoken library, and it is used to generate a JSON Web Token (JWT). In this case, the payload consists of information about the user, including their \_id, email, username, and fullname.

1.Payload:

{

\_id: this.\_id,

email: this.email,

username: this.username,

fullname: this.fullname,

}

This is the information that will be encoded into the JWT. It typically includes user-specific data that can be used to identify the user when the token is later decoded.

2.Secret Key:

process.env.ACCESS\_TOKEN\_SECRET

The second parameter to jwt.sign is the secret key or private key used to sign the JWT. It's crucial to keep this key secure, as it is used to verify the authenticity of the token during the decoding process.

3.Options:

{

expiresIn: process.env.ACCESS\_TOKEN\_EXPIRY,

}

The third parameter is an options object, which, in this case, includes the expiration time for the token. The expiresIn option is set to the value of process.env.ACCESS\_TOKEN\_EXPIRY, which presumably specifies the duration for which the token is valid.

Return Value:

The generateAccessToken method returns the generated JWT.

In summary, this method generates an access token for a user based on their information, signs it using a secret key, and includes an expiration time specified in the environment variables. This token can be used for authentication and authorization purposes in a web application.

**L-10:**

**src/middlewares/multer.middlewares.js**

import multer from "multer": This line imports the Multer middleware, which is a Node.js middleware used for handling multipart/form-data, primarily used for uploading files.

const storage = multer.diskStorage({ ... }): This line creates a storage engine using multer.diskStorage. This storage engine defines how files should be stored. In this case, it specifies the destination directory and the filename.

destination: This function determines the destination directory where uploaded files will be stored. In this example, it uses a callback (cb) to specify that the destination is the "./public/temp" directory.

filename: This function determines the filename for the uploaded file. In this example, it uses the original filename of the uploaded file.

export const upload = multer({ storage }): This line creates a Multer instance with the specified storage engine. The upload constant is then exported for use in other parts of your application.

multer({ storage }): This creates a Multer middleware instance with the storage engine you defined earlier. The storage parameter specifies how files should be stored.

So, in summary, this code sets up a Multer middleware with a disk storage engine. When a file is uploaded, it will be stored in the "./public/temp" directory with its original filename. The upload constant can be used as middleware in your routes to handle file uploads.

**Let's break down the code in src/utils/cloudinary.js:**

import {v2 as cloudinary} from "cloudinary";: This line imports the Cloudinary SDK, specifically version 2 (v2), and aliases it as cloudinary for easier use in the code.

import fs from "fs";: This line imports the Node.js built-in module fs (file system), which is used for working with the file system, including file operations.

cloudinary.config({ ... });: This sets up the configuration for Cloudinary using the provided environment variables. The process.env allows access to environment variables, and CLOUDINARY\_CLOUD\_NAME, CLOUDINARY\_API\_KEY, and CLOUDINARY\_API\_SECRET are assumed to be set in your environment.

const uploadOnCloudinary = async(localFilePath) => { ... }: This defines an asynchronous function named uploadOnCloudinary that takes a localFilePath parameter. This function handles the uploading of a file to Cloudinary.

try { ... } catch (error) { ... }: This sets up a try-catch block to handle errors that might occur during the file upload process.

if (!localFilePath) return null;: Checks if localFilePath is falsy (e.g., undefined or empty), and if so, the function returns null.

const response = await cloudinary.uploader.upload(localFilePath, { resource\_type: "auto" });: This line uploads the file to Cloudinary using the upload method from the cloudinary.uploader object. It specifies resource\_type as "auto" to automatically detect the type of resource being uploaded.

return response;: Returns the Cloudinary response object, which contains information about the uploaded file.

}

catch (error) { ... }: Catches any errors that occurred during the file upload process.

fs.unlinkSync(localFilePath);: If an error occurs, this line attempts to remove the locally saved temporary file using fs.unlinkSync. This ensures that if the upload fails, the local file is cleaned up.

return null;: Returns null in case of an error during the file upload process.

export { uploadOnCloudinary };: Exports the uploadOnCloudinary function to make it available for use in other parts of your application.

In summary, this module provides a function, uploadOnCloudinary, for uploading a file to Cloudinary. It handles the Cloudinary configuration, file upload, and cleanup in case of an error.

**L-11:**

HTTP crash course provides a broad overview of how things interact

The main difference between HTTP and HTTPS is that data sent through HTTPS is encrypted and becomes readable only on the server or client.

HTTP headers are a way to send additional information about the request or response

HTTP headers can be categorized into request headers, response headers, and representation headers.

User agents provide information about the user and can trigger app downloads

HTTP methods are used to perform specific operations

HTTP Methods: PUT and PATCH

HTTP status codes are used to communicate the success or failure of a request.

The difference between a good programmer and an average programmer is the amount of resources and information they gather.



**src/controllers/user.controllers.js**

The code defines a function generateAccessAndRefreshTokens

that fetches a user from the database by ID,

generates access and refresh tokens for the user,

updates the user object with the refresh token,

saves the user to the database,

and returns the generated tokens.

If any errors occur during this process, it throws a custom API error.

**Login**

// send cookies

const loggedInUser = await User.findById(user.\_id).

select("-password -refreshToken")

const options = { // modifiable by the server

httpOnly:true,

secure:true

}

It fetches the logged-in user from the database, excluding the password and refreshToken fields. It sets options for cookie configuration, including httpOnly (to prevent client-side access) and secure (to only send cookies over HTTPS).

return res.status(200).cookie("accessToken",accessToken,options).cookie("refreshToken",refreshToken,options).json(

new ApiResponse(

200,

{

// data

user: loggedInUser ,accessToken,refreshToken

},

"User logged in Successfully"

)

)

It sets cookies for accessToken and refreshToken in the response using the .cookie method. Then, it sends a JSON response using the ApiResponse class with a status code of 200 (OK), including the logged-in user, access token, and refresh token in the response data. The message is "User logged in Successfully."