# **Backend**

Server is a software that serves something.

**Working of Backend:**

Browser API (Response) Backend Database(Another Continent)

Data/File/Third Party API ` Handling in Backend

NodeJS -> Js Runtime

**Files in Backend:**

Package.json , .env, (readme,git,..)

Src:

-> index ->DB connect

->App -> configuration(main)

->constants ->enums, DB-name

**Directory that should be made:**

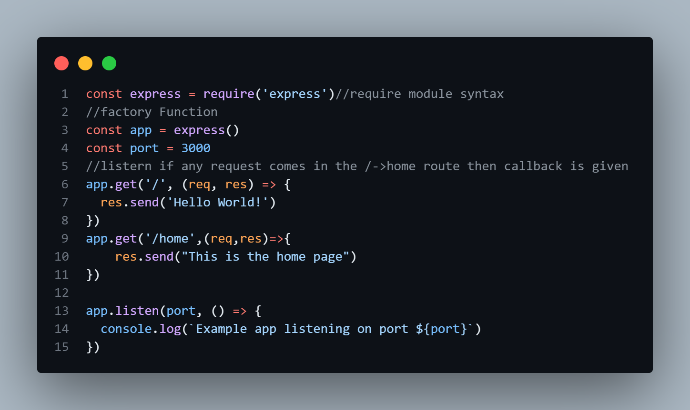
1. DB directory
2. Models -> data structure
3. Controllers ->methods and functionality
4. Routes -> /home or /about
5. Middle wares
6. Utils -> mail

get(request)

Computer Express server

response Listen: /:home route

/login : login setup

 **Express:**

1. Hello world

2. And production application:

Package -> dot env

->install: npm install dotenv

->creating a file .env

A. .env file:

PORT = 4000

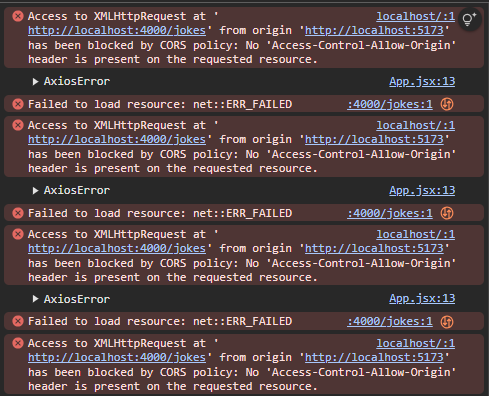
B.

Reads the port number from an environment variable or defaults to 3000



Starts the server and listens on the specified port

.

#Backend and Frontend



->Sends a GET request to the backend API /api/jokes.

->This request is proxied to the backend using Vite’s proxy configuration.

-jokes: Holds the list of jokes fetched from the backend.

-setjokes: A function to update the jokes state.

-Initial state is an empty array [].

useEffect: A React hook that runs side effects (like fetching data).

useState: A React hook to manage component-level state.

axios is used to make HTTP requests.

*$CORS policy means : only allowing which has same origin: not all are allowed:*

*: cross origin request:*

*Url, port then only they have same origin to connect the backend and frontend,*

*We can slove by making making the url/domain white list, or mark \**

***Solving Error:***

1. *Using the proxy in frontend:[vite.config.js]*



server.proxy:

->Redirects frontend requests starting with /api to the backend running at http://localhost:4000.

* This avoids Cross-Origin Resource Sharing (CORS) issues during development.

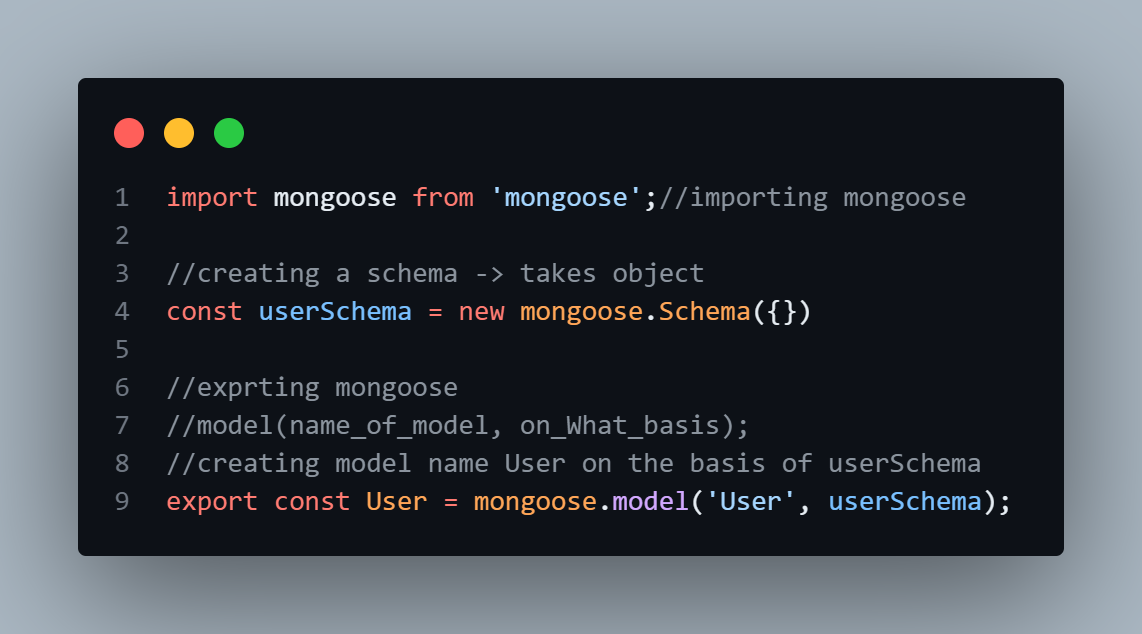


This hook runs after the component renders.

*When proxy is used, this request will act like the same as the origin then cros error will not occur since: ther are both from the same origin.*

# Mongoose [MongoDB]:

A popular ODM (Object Data Modeling) library for MongoDB and Node.js, used for database operations.



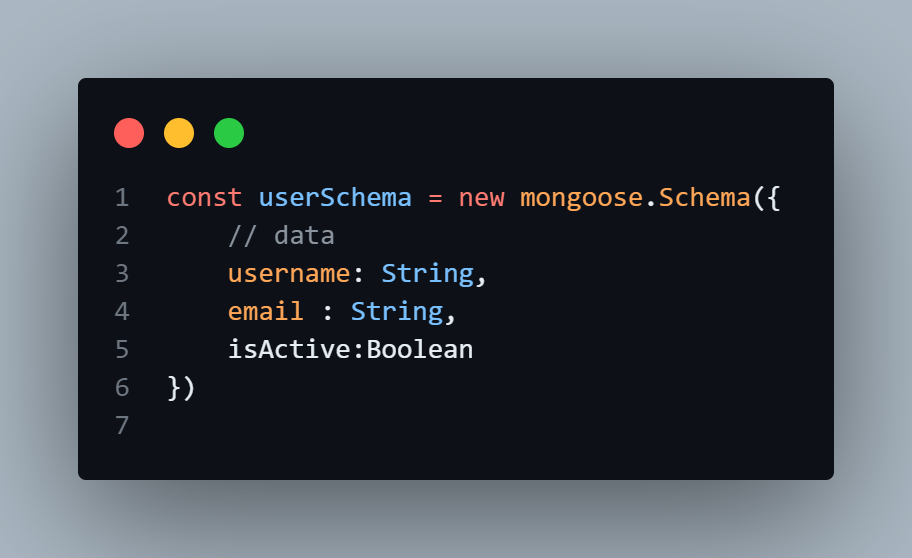
In database User will be converted to users

Data saving in mongoose:



required: [true, 'password is required']

A custom message can also be sent



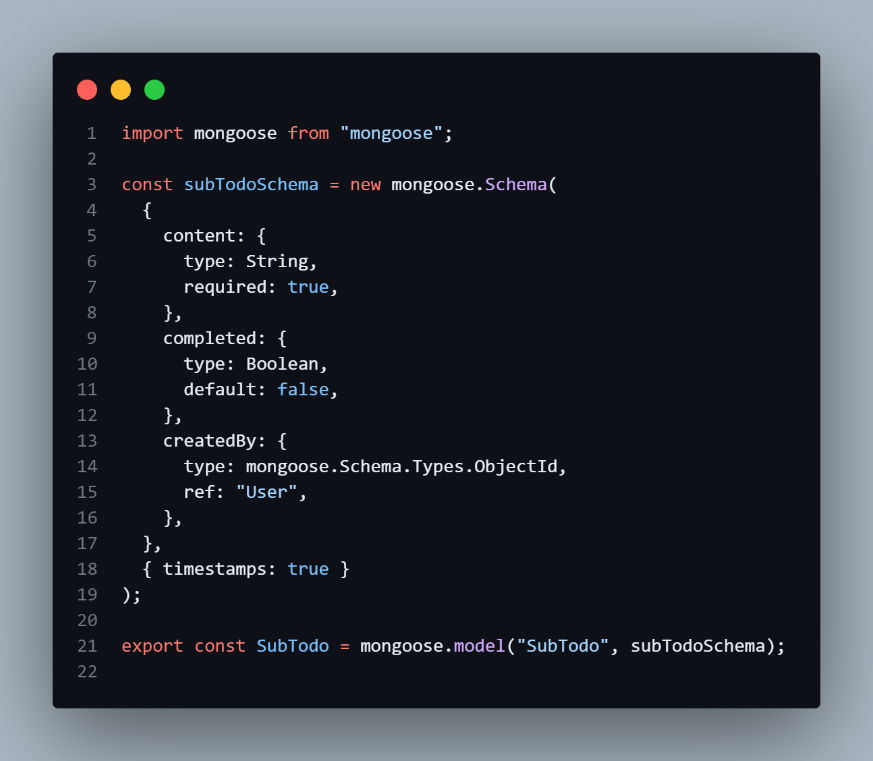
}, {timestamps: true})

To add created at time and updated at time in database

Type 1 OR, Type 2 [Most reliable]

Connecting documents[relate]:

1. User:user.models.js
2. todo.mode.js:
3. sub\_todos.models.js:



Note: Always use asyc await and try catch or promises to handle error in data base connection since it takes time and it’s assumed to be in another continent and error occurs during the connection of database.

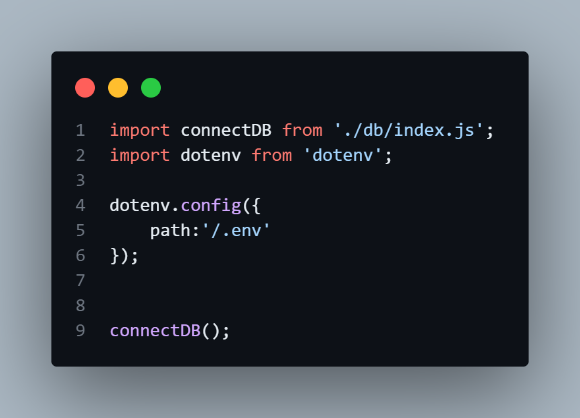
# Connection to Database:

* + - 1. First Approach:
      2. Better Approach

db/index.js->



connectionInstance.connection.host contains the host name (e.g., cluster0.mongodb.net), so this confirms the connection.

index.js -> main file constants.js

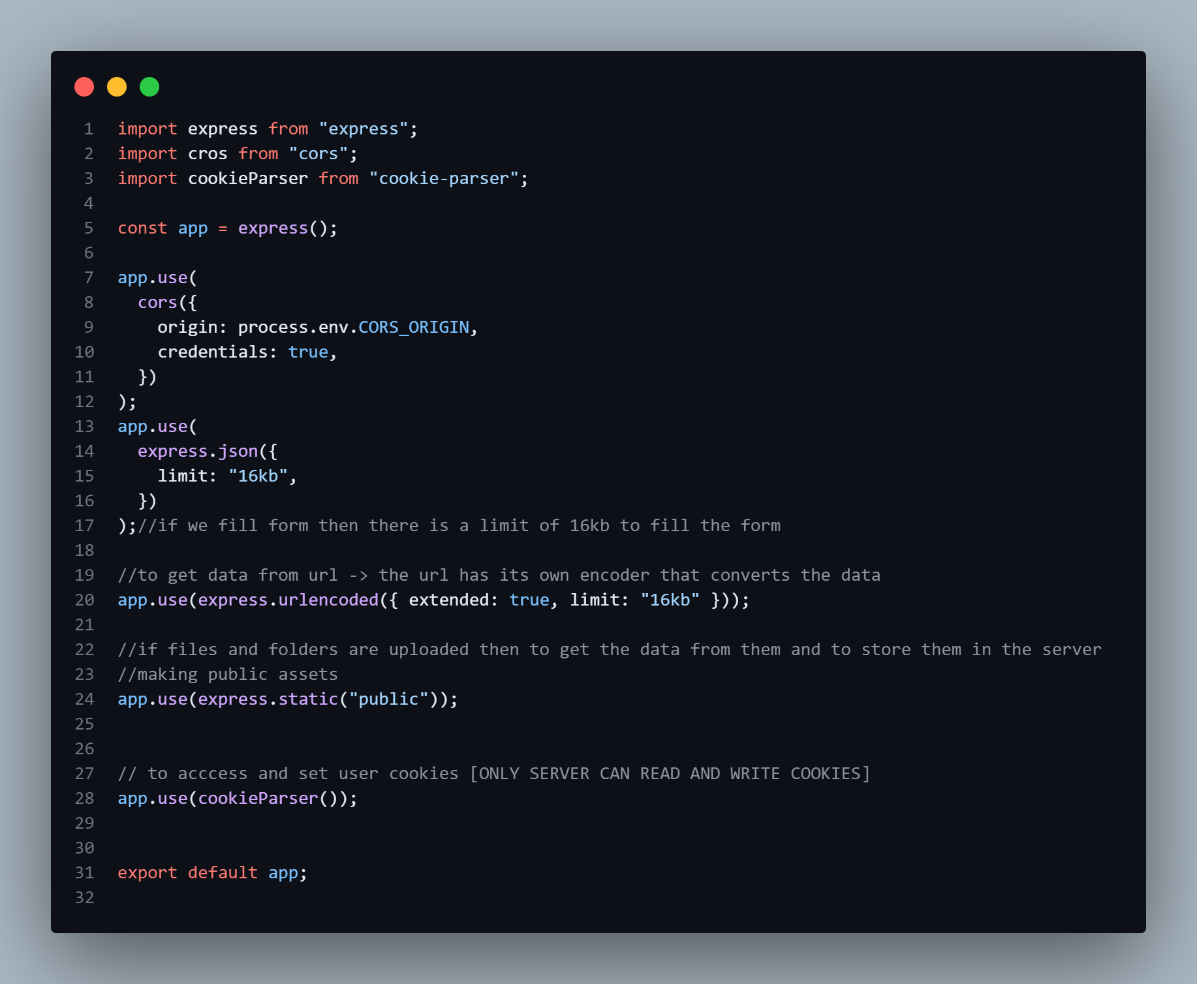
# Cookies:

Cookies are small pieces of data stored on a user's device by a web browser while they are browsing a website. They are used to remember information about the user or the session, enabling websites to provide a more personalized and consistent experience.

1. App.js

CORS\_ORIGIN = \*

This accepts all the request..



 **Purpose**:

* This is the foundation of your server. It defines the Express app and configures the middleware stack to handle requests, process incoming data, and serve static files.

 **Middleware explained**:

1. **CORS (cors)**: Allows cross-origin requests. It enables the server to accept requests from a specific origin (set by process.env.CORS\_ORIGIN) and share credentials like cookies.
2. **express.json and express.urlencoded**: These handle parsing of JSON and URL-encoded data in incoming requests with a data size limit of 16 KB.
3. **express.static**: Serves static files (like images, stylesheets, or JavaScript) from the public folder.
4. **cookie-parser**: Enables the server to read/write cookies in HTTP requests.

 **Usage**: This configured app is exported for use in your main server entry point (e.g., server.js or index.js). It processes and routes all incoming HTTP requests.

We can also user the white list to accept the specific **URL** only.

1. asyncHandler.js
2. apierror

return



 A custom error class to create detailed, consistent error responses.

 Extends the built-in Error class, adding properties like statusCode, errors, and a stack trace.

The ApiError class **extends** the built-in Error class, making it a subclass of Error.

#### **Why** super **is needed:**

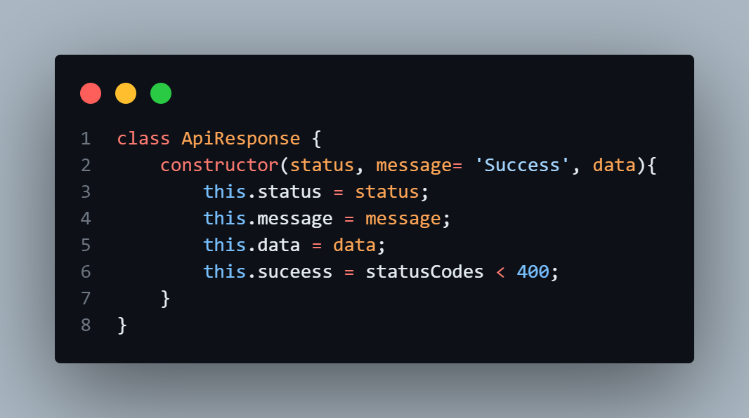
* When extending a parent class (Error), you must call the parent's constructor using super() to properly initialize the inherited properties.



 Ensures consistent error handling for asynchronous request handlers (e.g., when using async/await).

 If an error occurs in a request handler, it will automatically be passed to Express' error-handling middleware using next(err).

If the handler throws an error (e.g., due to a failed database call), the error is caught and forwarded to the error-handling middleware.

1. Apirespose

 A standardized way to send success responses.

 Encapsulates HTTP status, message, and data into a consistent format.

# Tokens

Tokens are unique pieces of information (usually strings) used in software systems for **authentication, authorization, and secure communication**. They act as temporary, verifiable identifiers that represent a user, application, or entity during interactions with a system.

Using jsonwebtoken(bwt):

## Bcrypt: A library to help you hash passwords.

Password should not be in plane text format so bcrypt or bcryptjs helps in encryption and decreption , comparing the password.

# Pre-middleware:

Since encrypton and decreption cannot be done directly we use pre middleware

Pre-middleware functions are executed one after another, when each middleware calls next

|  |
| --- |
|  |
| | **Feature** | **Tokens** | **Sessions** | **Cookies** | | --- | --- | --- | --- | |
| |  |  |  |  | | --- | --- | --- | --- | | **Definition** | A piece of data (usually JWT) used to authenticate and authorize users. | A server-side mechanism that stores user state. | Small data stored in the browser to remember user info. | |  | | | | |
| |  |  |  |  | | --- | --- | --- | --- | | **Storage** | Stored on the client-side (e.g., browser local storage, session storage, or memory). | Stored on the server, with only a session ID stored on the client (usually in a cookie). | Stored in the browser (client-side). | |
| |  |  |  |  | | --- | --- | --- | --- | | **Usage** | Commonly used for API authentication (stateless). | Used to maintain user state and login status. | Used to store user preferences, authentication info, or tracking data. | |
| |  |  |  |  | | --- | --- | --- | --- | | **Security** | Can be signed and encrypted (e.g., JWT). Must be securely stored to prevent XSS. | More secure since session data is on the server. Can be hijacked if the session ID is stolen. | Vulnerable to XSS and CSRF attacks if not set properly. | |
| |  |  |  |  | | --- | --- | --- | --- | | **Lifetime** | Usually has an expiration time (short-lived or long-lived). | Exists until the user logs out or session expires (configurable on the server). | Can be set to expire when the browser is closed (session cookies) or persist longer (persistent cookies). | |
| |  |  |  |  | | --- | --- | --- | --- | | **Scalability** | Scalable for distributed systems (stateless, no need for session storage). | Requires session management, less scalable without proper setup (e.g., sticky sessions, database storage). | No direct impact on scalability but depends on browser storage limitations. | |
| |  |  |  |  | | --- | --- | --- | --- | | **Example Use** | API authentication, Single Sign-On (SSO). | Maintaining login state for a web app. | Remembering user preferences (e.g., dark mode, language settings). | |

# mongoose-aggregate-paginate-v2

Pagination is the process of **splitting large sets of data into smaller, manageable chunks (pages)** instead of displaying everything at once. This is commonly used in databases, APIs, and web applications to improve **performance** and **user experience**.

For example, when you browse an **e-commerce website**, instead of loading **thousands of products** at once, you see **a limited number per page (e.g., 10 or 20 items)**, with options to go to the **next or previous page**.

* npm install mongoose-aggregate-paginate-v2

# jsonwebtoken:

* Algorithm.Data[payload].24-bit-secet-code
* Hashes & verifies passwords securely.

### **Uses of JWT in Website Design**

1. **Stateless Authentication** – JWT allows users to stay logged in without storing session data on the server, making it scalable.
2. **API Authentication & Authorization** – Tokens verify users before allowing access to protected routes or API endpoints.
3. **Single Sign-On (SSO)** – JWT enables users to authenticate once and access multiple services without needing to log in repeatedly.
4. **Role-Based Access Control (RBAC)** – User roles (e.g., admin, user) can be included in JWT payloads to enforce access control.
5. **Secure Client-Server Communication** – Tokens ensure that only authenticated users can make API requests, reducing CSRF risks.

* **Cross-Site Request Forgery (CSRF)** is a web security vulnerability that allows an attacker to trick a user into executing unwanted actions on a trusted website where they are authenticated.

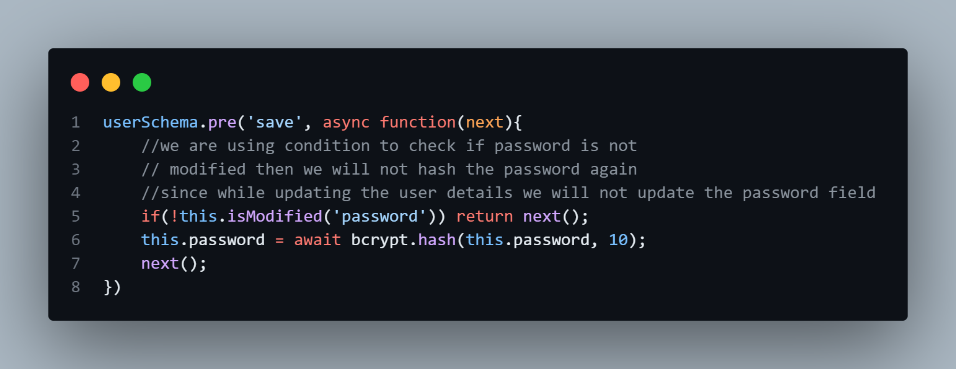
1. **Session Management Without Cookies** – Unlike traditional authentication, JWTs can be stored in localStorage or HTTP-only cookies to maintain user sessions.

# Mongoose pre hook (Middleware)

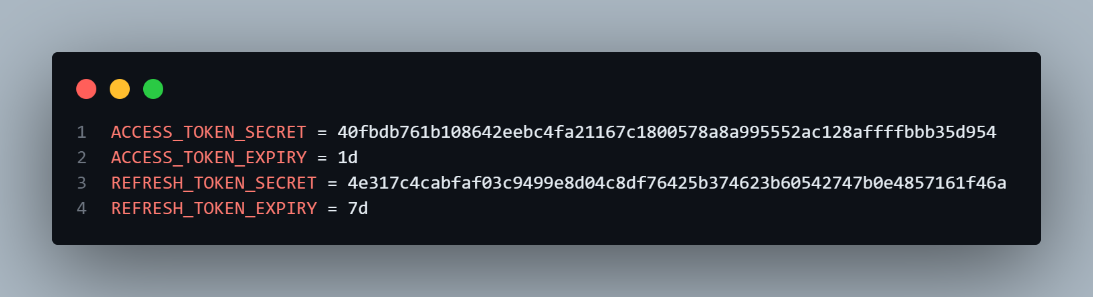
A **pre-hook** (or middleware) is a function that runs before a certain operation, such as saving or updating a document. These hooks allow you to modify the data, perform validation, or execute custom logic before Mongoose carries out the operation.

* Use normal function instead of the regular function since: arrow function doesn’t allow this feature since we need this feature in pre-hook.

1. Encrypting password.
2. Checking password entered by user and password in database.



* If condition not used then every time when we change something the password will be encrypt so to only encrypt password we check .
* This condition checks if there is any modification in the password field.

Jwt:

1. User.model.js

## **User Model (Mongoose Schema)**

The **Mongoose schema** (user.models.js) defines the structure of a user document.

### **Key Fields in the Schema**

* **timestamps: true**: Automatically adds createdAt and updatedAt fields.

## **4. Password Hashing Using Bcrypt**

* **pre('save')**: This middleware runs **before** saving a user.
* **bcrypt.hash()**: Hashes the password before storing it.

## **5. User Authentication Methods**

### **Checking If Password is Correct**

This method is used during login to compare the entered password with the hashed password.

javascript

CopyEdit

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

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

};

### **Generating JWT Tokens**

JWT is used for authentication. The following methods generate an **access token** and a **refresh token**.

#### **Access Token (Short-lived)**

javascript

CopyEdit

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 }

);

};

#### **Refresh Token (Longer-lived)**

javascript

CopyEdit

userSchema.methods.generateRefreshToken = function() {

return jwt.sign(

{ \_id: this.\_id },

process.env.REFRESH\_TOKEN\_SECRET,

{ expiresIn: process.env.REFRESH\_TOKEN\_EXPIRY }

);

};

* The **access token** is used for API authentication.
* The **refresh token** allows users to request a new access token when the old one expires.

Video.models.js

## uploading file in coudinary using multer

cloudinary.js

multer.middleware.js

#### **1️. Multer Setup (Handles Local File Upload)**

* **Used for:** Temporarily storing files before uploading to Cloudinary.
* **Storage:** Saves files in the ./public/temp/ folder.
* **Filename Handling:** Uses file.originalname, which can cause overwrites (should be fixed using unique names).
* **Exports:** upload middleware for handling file uploads in Express.

#### **2️. Cloudinary Setup (Uploads Files to Cloud)**

* **Configuration:** Uses cloud\_name, api\_key, and api\_secret from environment variables for security.
* **Resource Type:** resource\_type: 'auto' allows images, videos, and other file types.
* **Deletes Local File:** After uploading to Cloudinary, the temporary file is removed using fs.unlinkSync(localFilePath).

#### **3️. Error Handling**

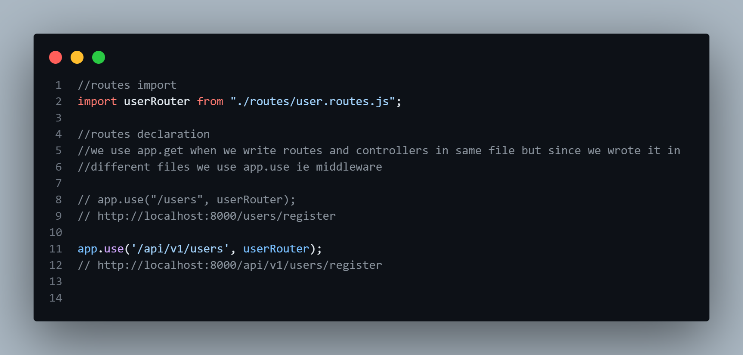
* **Prevents Upload of Null Files:** If localFilePath is missing, the function returns null.
* **Logs Errors:** If Cloudinary upload fails, it logs an error message.
* **Deletes Files on Failure:** Ensures local files are deleted even if the upload fails.

#### **4️. Exported Functions**

* **upload (Multer middleware):** Handles file uploads in Express routes.
* **uploadOnCloudinary() (Cloudinary function):** Uploads files and returns Cloudinary's response.

Routes:

user.controller.js user.routes.js

app.js

Routes in a backend application define how the server should respond to client requests (like GET, POST, PUT, DELETE). They act as endpoints that allow communication between the frontend and backend.

For example, in an **Express.js** backend, routes handle API requests such as:

* **GET** /users → Fetch all users
* **POST** /users → Create a new user
* **PUT** /users/:id → Update a user by ID
* **DELETE** /users/:id → Delete a user

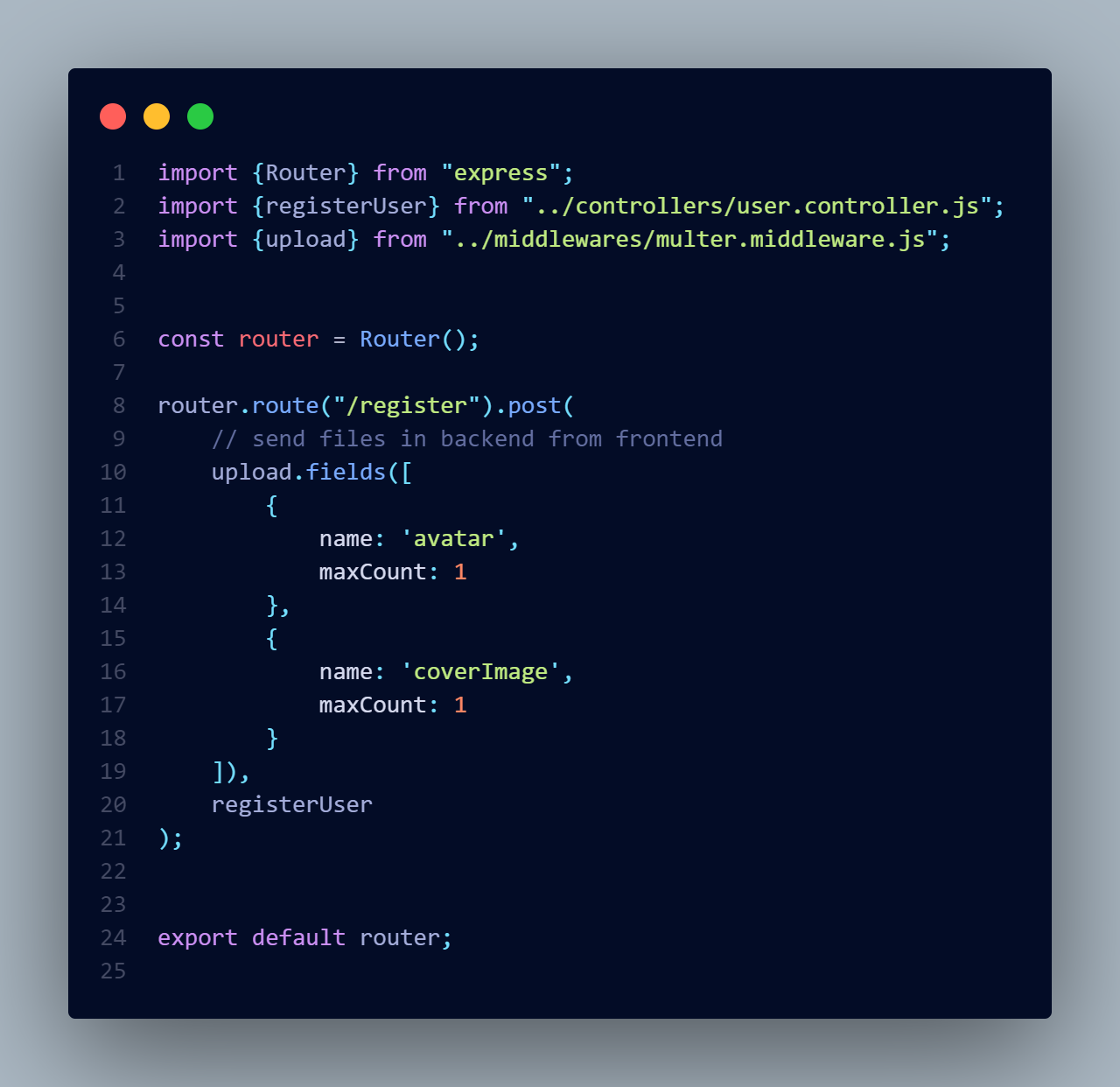
### Registering user:

When a user tries to register, the frontend sends a **POST request** to the /register route. The request body contains:

fullName,email,username,password,avatar (profile picture),coverImage (optional background image)

**upload.fields([...])**: This uses **Multer** (a middleware for handling file uploads) to extract avatar and coverImage from req.files.

User.routes.js



steps for register user

1. get data from frontend
2. validate data-> not empty
3. check if user already exists: username, email
4. check for images, check for avatar
5. upload them to Cloudinary: check avatar in cloudinary
6. Create user object - create entry in database
7. remove password and refresh token field from response
8. check for user creation
9. return response

user.controller.js

The .some() method checks if **at least one** element in the array meets a condition

 **field?.trim()**:

* ?. (optional chaining) ensures that if field is null or undefined, it won’t cause an error.
* .trim() removes extra spaces at the beginning and end of the string.

 **=== ""**:

* If the trimmed value is an empty string, it means the field was either empty or contained only spaces.

### **Mongoose Syntax Summary**



| **Mongoose Method** | **Purpose** |
| --- | --- |
| User.create({...}) | Creates and saves a new user |
| new User({...}).save() | Alternative way to create and save a user |
| User.findById(id) | Finds a user by their \_id |
| .select("-field1 -field2") | Excludes specific fields from the result |

user.router.js

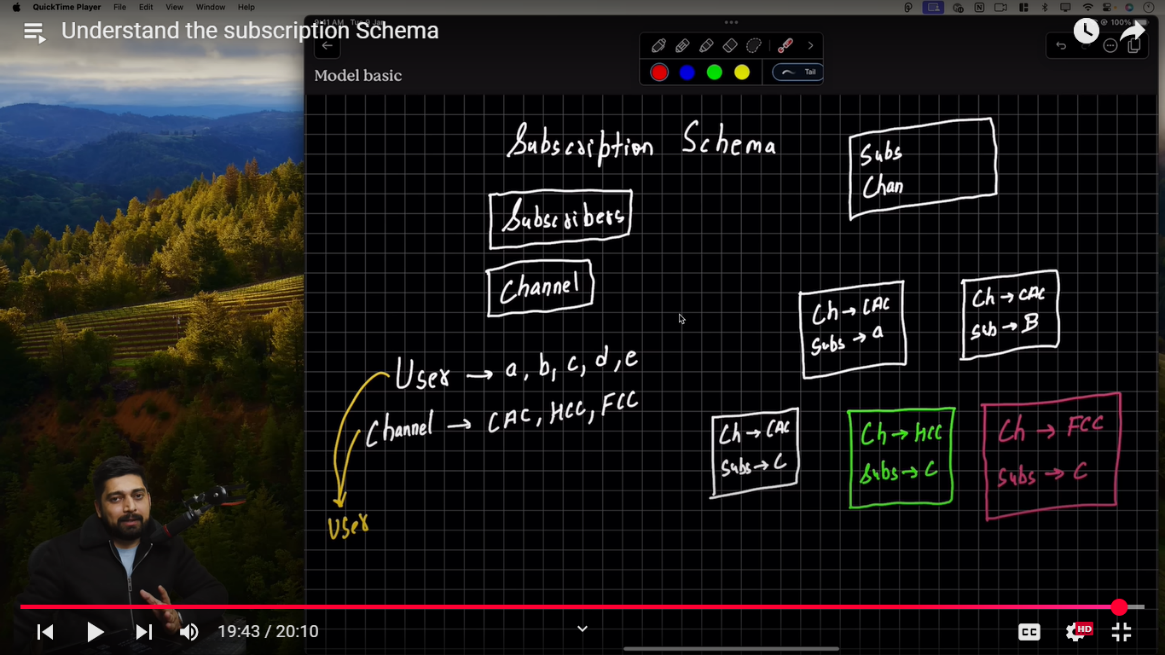
auth.middleware.js



user.controller.js

# Subscription Schema

We could create a subscriber’s array in user also But, if we do this then there may be millions of subscribers then we have millions of arrays then deletion of a subscriber from the array will be problematic. So, we create subscription models.

Since we have created the subscription schema then when the user subscribes to certain channel /user then a new document is created then we use the channel name not the user name to count the channel to find subscription to your channel.

To find the subscriber of that channel we use the channel to match from the document created. But, to find the how many channels that subscriber has subscribed we use the **subscriber's username** and search for matching documents based on the channel field.

# MongoDB Aggregation Pipelines.

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

* $project
  + Reshapes each document in the stream, e.g., by adding new fields or removing existing fields. For each input document, output one document.
* $match
  + Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. For each input document, the output is either one document (a match) or zero document (no match).
* $group
  + Groups input documents by a specified identifier expression and apply the accumulator expression(s), if specified, to each group. $group consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field (group id) and, if specified, accumulated fields.
* $sort: Reorders the document stream by a specified sort key. The documents are unmodified, except for the order of the documents. For each input document, the output will be one document.
* **$addFields: Is an aggregation stage in MongoDB that adds new computed fields to the documents without modifying the original data**.



 **Filters** the User collection to **find the user** whose username matches the provided username (converted to lowercase).

 Ensures that only relevant documents move to the next stages.

 Performs a **left join** with the subscriptions collection.

 **Matches** \_id of the user (channel owner) with the channel field in subscriptions.

 Stores **all matching documents** in an array called "subscribers"

 **Includes** (1) only the required fields in the final output.

 **Excludes** unnecessary fields like \_id, subscribers, and subscribedTo, which were only used for calculations.

$addFields is used to **calculate and add three new fields**:subscribersCount → **Total number of subscribers**.

channelsSubscribedToCount → **Total number of channels the user subscribed to**.isSubscribed → **Whether the logged-in user is subscribed to this channel**.

# User watch history:

We used aggregation pipelines to get the user watch history. Since we are getting watch history from the video and in video we have owner in which we are getting user details we are using sub-pipelines to get the user details from user models and to limit the data from user we use another pipeline ($project).

**Steps**:

 Fetching **the User's Watch History:**

* The User.aggregate() method is used to perform an aggregation pipeline query on the User collection.
* The first stage, $match, filters the user by \_id, which is extracted from req.user?.\_id and converted to a mongoose.Types.ObjectId. This is necessary because \_id in MongoDB is an ObjectId, and it must match the correct type.

 Joining **with the "Videos" Collection:**

* The $lookup stage fetches related video data from the videos collection.
* It matches the user's watchHistory array with the \_id field in the videos collection and stores the result in watchHistory.
* A pipeline is included inside the $lookup to further process each video document.

 Fetching **the Owner Details of Each Video:**

* Inside the $lookup pipeline, another $lookup operation is performed on the users collection.
* This joins each video's owner field with the \_id field in users, retrieving the owner's details.
* The $project stage limits the owner’s details to only fullName, avatar, and username.

 Restructuring **the Data for Easier Frontend Use:**

* The $addFields stage adds a new owner field to each video document.
* It extracts the first (and only) element from the owner array using $first, converting it from an array to a single object.

 Sending **the Response:**

* The response is sent back with a status code of 200.
* The ApiResponse wrapper is used to format the response, containing the user's watch history along with video and owner details.
* The response message "Watch history fetched successfully" is included.



Adding routes….

# Backend Project Steps

1. **Database Connection**
   * Set up the database connection in index.js.
2. **Application Setup**
   * Configure paths and connection in index.js.
   * Set up routing, JSON parsing, file size limits, URL encoding, and cookie parser in app.js.
   * Implement API versioning.
   * Import and configure routes (user.routes.js).
3. **Models**
   * Create UserModel with:
     + Pre-save hooks.
     + Password correction hooks.
     + JWT token generation hooks.
     + Save and retrieve data from the database.
   * Create VideoModel and SubscriptionModel using aggregation pipelines.
4. **Authentication**
   * Implement refresh token and access token mechanisms.
5. **Controllers and Routes**
   * User-related routes:
     + Register user (multer -> Cloudinary).
     + Login user.
     + Logout user.
     + Refresh access token.
     + Change password.
     + Get current user.
   * Video-related routes:
     + Upload videos.
     + Manage subscriptions.
     + Get channel profile.
     + Get watch history using aggregation pipelines ($lookup, $addFields, $match, $project).
6. **Middleware & File Management**
   * Implement authentication middleware.
   * Use multer for file handling.
   * Upload and manage files using Cloudinary.

## Steps in backend: [Rough sketch]

* Database connection:
* We setup paths and connection in index.js
* In app.js: setting up routing -> json, limiting in file, URL encode, cookie parser and routes (user.routes.js) and writing controllers, api versioning
* Models: user model, and its hooks: pre, password correction hooks, generate jwt token hooks, and put all in database and receive and save
* Refresh token and access token
* User model controller
* Video models and subscription model -> aggregation pipelines
* Controllers -> routes: register user->multer->cloudinary , login user, logout, refresh access token , change password - >CRUD operation, get current user , auth middleware,multer middleware, file management and updating, get channel profile and get watch history ->aggregation pipelines[$lookup, $addfields, $match.