Backend Notes by me

### Two types of dependency in node:

#### normal dependency:

using --save

ex: $npm i express --save

Package.json e always thakbe,amra use korte parbo

—save na dile boro project e package e jhamela hoy, dile git pull korle local machine e same dependency install hobe

#### Dev dependency:

Production e lagbe na, for development purpose only

ex: $npm i express --save-dev

### commands

***$node -v***

***$npm -v***

***$code .***

***$npm init***

***$npm init -y (for not answering any question)***

***$npm i express —save***

***$npm run start (main command $node server.js,jeta server.js file ta k execute kore, but package.json e eta start diye configure kora)***

### Initial code explanation

const express = require('express');

const app = express();

* **Import Express**: express is required to create a web server.
* **Initialize App**: app is an instance of the Express application.

//? Check Connection

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

res.json({ message: 'welcome' });

});

* **Route Definition**: Defines a GET route for /.
* **Request Handler**: When / is accessed, the server responds with a JSON object: { message: 'welcome' }.
* **Purpose**: Useful for a basic health check of the server (confirming it's running and responding). It is a **route handler**, not middleware.

const port = process.env.PORT || 3000;

* **Port Configuration**:
  + process.env.PORT: Retrieves the port number from the environment variables (useful for deployment, where the hosting platform specifies the port).
  + || 3000: Falls back to port 3000 if no environment variable is set (e.g., during local development).

app.listen(port, () => {

console.log(`Server running on port ${port}`);

});

* **Start the Server**:
  + app.listen(port, callback): Starts the Express server, listening on the specified port.
  + The callback function logs a message to the console when the server starts successfully.

### To install nodemon:

***$npm i nodemon –save-dev***

Then package.json => scripts => {”dev”: “nodemon server.js”}

To apply changes on the fly, give following command instead of “**$npm run start**”

**$npm run dev**

### To run debugger:

- make sure that following two line is included in “package.json->{}scripts”  
 **$"start": "node server.js”**

**$"dev": "nodemon server.js"**

- click debug icon above script function

- click on dev from {start,test,dev} options.

- it will run a new javascript debug terminal

### To make API call in postman:

* Click new(left upper one) -> click collection -> add request -> GET [address]

### **CRUD** operation:

*-create [POST] {ex: linkedin account creation}*

*-read [GET] {view profile}*

*-update [PUT/PATCH] {add info}*

*-delete [DELETE] {delete account}*

#### Request er body pete hole

*npm er body-parser package namaste hobe”* ***$npm i body-parser —save***

*Surute body-parser require korte hobe*

#### Routes Overview

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| **POST** | /users | Create a new user |
| **GET** | /users | Retrieve all users |
| **GET** | /users/:id | Retrieve a user by ID |
| **PUT** | /users/:id | Update a user's details |
| **DELETE** | /users/:id | Delete a user by ID |

#### 

## To run debugger:

In terminal, give the command-

$npm run dev

Then server will start.

The line you want to debug, just add a breakpoint.

Reload from user end.  
Then debug environment will appear with some floating options,use them to debug.

Remember: after yellow box appear around breakpoint, postman api call doesn’t work. Remove the breakpoint to make api call again from postman

## Route Implementations WIthout DB

### Create a User (POST /users)

app.post('/users', (req, res) => {

const user = req.body; // Extract user data from the request body

user.id = ++lastid; // Assign a unique ID to the new user

users.push(user); // Add the user to the users array (in-memory storage)

res.status(201).json(user); // Respond with the created user and HTTP status 201

});

### Get All Users (GET /users)

app.get('/users', (req, res) => {

res.json(users);

});

### Get a User by ID (GET /users/:id)

app.get('/users/:id', (req, res) => {

const id = parseInt(req.params.id);

const user = users.find(u => u.id === id);

user ? res.json(user) : res.status(404).json("user not found");

});

### Update a User (PUT /users/:id)

app.put('/users/:id', (req, res) => {

const id = parseInt(req.params.id); // Extract ID from the URL and convert it to an integer

const index = users.findIndex(u => u.id === id); // Find the index of the user with the given ID

if (index === -1) return res.status(404).json("user not found"); // If user not found, return 404

users[index] = { ...users[index], ...req.body }; // Merge existing user data with updated data

res.json(users[index]); // Respond with the updated user data

});

### Delete a User (DELETE /users/:id)

app.delete('/users/:id', (req, res) => {

const id = parseInt(req.params.id); // Extract user ID from URL and convert it to an integer

const index = users.findIndex(u => u.id === id); // Find the index of the user with the given ID

if (index === -1) return res.status(404).json("user not found");

users.splice(index, 1); // Remove the user from the array

res.json({ message: "user deleted" }); // Send success response

});

### Middleware

To parse JSON request bodies:

app.use(express.json());

### Mongodb-compass and mongoose installation

### TO SEE INSTALLATION OF MONGODB-Compass AND CONNECT MONGOOSE to vscode(google by mongoose npm) SEE VIDEO no 045

### To install mongoose:

***&npm i mongoose –save***

## Route Implementations WIth DB

### Folder Structure:

### 

### 

### /server.js:

const express = require('express'); // Import Express framework to create server and handle routing

const app = express(); // Initialize an Express application instance

const bodyParser = require('body-parser'); // Import body-parser to parse incoming request bodies

const connectDB = require('./config/db'); // Import the database connection function

//? Parse Request Body Middleware

app.use(bodyParser.json());

// Middleware that allows the app to parse JSON request bodies, enabling access via req.body

//? Connect to MongoDB

connectDB();

// Calls the function to establish a connection with the MongoDB database

//? API Routes

app.use('/api/users', require('./routes/api/users'));

// This line mounts the user routes (CRUD operations on users) under the '/api/users' endpoint

//? Check Connection Endpoint

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

res.json({ message: 'welcome' });

// Sends a JSON response to the root URL ('/') to confirm the server is running

});

//? Server Port Configuration

const port = process.env.PORT || 3000;

// Sets the server to listen on the port specified in the environment variable or defaults to 3000

app.listen(port, () => {

console.log(`Server running on port ${port}`);

// Starts the server and logs a message indicating the port it's running on

});

### /config/db.js:

const mongoose = require('mongoose'); // Import Mongoose for database interaction

const uri = 'mongodb+srv://nodejs-c1:nodejs-c1@nodejs-c1.530gk.mongodb.net/'; // MongoDB connection URI

const connectDB = async () => {

try {

await mongoose.connect(uri); // Connect to MongoDB using the provided URI

console.log('Mongoose Connect'); // Log success message if connection succeeds

}

catch (error) {

console.error(error.message); // Log the error message if the connection fails

}

}

module.exports = connectDB; // Export the connectDB function to be used elsewhere in the application

### /models/User.js:

const mongoose = require('mongoose'); // Import Mongoose to interact with MongoDB

const UserSchema = new mongoose.Schema({

fname: {

type: String // Field 'fname' should be of type String

},

lname: {

type: String // Field 'lname' should be of type String

},

email: {

type: String // Field 'email' should be of type String

}

}, {

timestamps: true // Automatically adds 'createdAt' and 'updatedAt' fields to the schema

});

module.exports = mongoose.model('User', UserSchema); // Export the Mongoose model named 'User'

### /routes/api/users.js:

const express = require('express'); // Import Express framework to handle routing functionalities.

const router = express.Router(); // Create an instance of the Express router to define API routes.

const User = require('../../models/User'); // Import the User model to interact with the MongoDB collection.

//? Create a new user (POST request)

router.post('/', async (req, res) => {

try {

// Extract user details from the request body.

// Why: This helps to separate the incoming data and prepare it for database storage.

const userObj = {

fname: req.body.fname, // First name is extracted from the request body.

lname: req.body.lname, // Last name is extracted from the request body.

email: req.body.email, // Email is extracted from the request body.

};

// Create a new user instance using the Mongoose model.

// Why: This step helps in ensuring data adheres to the defined schema before saving.

const user = await User(userObj);

// Save the new user to the database.

// How: The `save()` function writes the new user to the database.

await user.save();

// Send a response with HTTP status 201 (Created) and return the created user.

// Why: HTTP 201 indicates successful resource creation.

return res.status(201).json(user);

}

catch (err) {

// Handle any errors and respond with a generic message.

// Why: To avoid exposing sensitive error details to the client.

res.status(500).json({ message: "Something went wrong" });

}

});

//? Get all users (GET request)

router.get('/', async (req, res) => {

try {

// Retrieve all users from the database.

// How: The `find({})` method fetches all documents from the collection.

const users = await User.find({});

// Return all users with HTTP status 200 (OK).

// Why: HTTP 200 indicates a successful data retrieval.

return res.status(200).json(users);

}

catch (err) {

// Handle any errors that occur during database query execution.

res.status(500).json({ message: "Something went wrong" });

}

});

//? Get a specific user by ID (GET request)

router.get('/:id', async (req, res) => {

try {

// Extract the user ID from the request parameters.

// How: `req.params.id` retrieves the ID from the URL.

const id = req.params.id;

// Find the user by ID in the database.

// How: `findById(id)` searches the collection for a matching document.

const user = await User.findById(id);

if (user) {

// If user is found, return the user data.

return res.json(user);

} else {

// If no user is found, return a 404 (Not Found) status.

return res.status(404).json("user not found");

}

}

catch {

// Catch block to handle any unexpected errors.

res.status(500).json({ message: "Something went wrong" });

}

});

//? Update an existing user (PUT request)

router.put('/:id', async (req, res) => {

try {

// Extract the user ID from the request parameters.

// Why: The ID is needed to identify which record to update.

const id = req.params.id;

// Extract updated user data from the request body.

const userBody = req.body;

// Find the user by ID and update with new data.

// How: `findByIdAndUpdate` takes ID, new data, and `{ new: true }` to return updated data.

const updatedUser = await User.findByIdAndUpdate(id, userBody, { new: true });

// Optional comments to clarify update options

// Only update `fname` field: `User.findByIdAndUpdate(id, { fname: req.body.fname }, { new: true })`

// Show old values after update: `User.findByIdAndUpdate(id, { fname: req.body.fname })`

if (updatedUser) {

// If update is successful, return the updated user data.

return res.json(updatedUser);

} else {

// If no user is found, return a 404 (Not Found) response.

return res.status(404).json("user not found");

}

}

catch {

// Handle any errors that might occur during the update process.

res.status(500).json({ message: "Something went wrong" });

}

});

//? Delete a user

router.delete('/:id', async (req, res) => {

try {

const id = req.params.id; // Extract the user ID from the request URL parameter

// Attempt to delete the user by ID from the database

const deletedUser = await User.findByIdAndDelete(id);

if (deletedUser) {

// If user is found and deleted, return success message with deleted user details

return res.json({ "following user deleted": deletedUser });

} else {

// If no user is found with the given ID, return 404 Not Found status

return res.status(404).json("user not found");

}

} catch (err) {

// Handle any errors that may occur during the operation and respond with 500 Internal Server Error

res.status(500).json({ message: "Something went wrong", error: err.message });

}

});

module.exports = router; // Export the router to be used in the main server file.

## Some Explanation

#### 1. Understanding async and await:

#### **async (Asynchronous Function)**

* The async keyword is used to define a function that returns a **Promise** implicitly.
* It allows the function to handle asynchronous operations in a readable way.
* Functions declared with async can contain the await keyword.

#### **await**

* The await keyword is used inside an async function to pause execution until the Promise is resolved.
* It ensures that the function waits for an asynchronous operation to complete before moving to the next line of code.
* Instead of using traditional .then() promise chains, await makes the code look synchronous and cleaner.

#### **Example:**

async function fetchData() {

const data = await someAsyncFunction(); // Waits until someAsyncFunction resolves

console.log(data);

}

**How it works:**

1. The function is marked async to indicate it contains asynchronous operations.
2. await makes the function wait until someAsyncFunction() completes, then assigns the result to data.
3. The next line executes only after the Promise resolves.

#### 

#### 2. findByIdAndUpdate() Function Structure

#### **Syntax:**

Model.findByIdAndUpdate(id, updateObject, options);

#### **Explanation of Parameters:**

1. id: The unique identifier (usually \_id) of the document to be updated.
2. updateObject: An object containing the fields to be updated.
3. options (optional):
   * { new: true }: Returns the updated document instead of the old one.
   * { upsert: true }: If the document does not exist, create a new one.
   * { runValidators: true }: Runs schema validators before updating.

#### **Example:**

const updatedUser = await User.findByIdAndUpdate(

"65a3b4cde78", // Example document ID

{ fname: "Updated Name" }, // Fields to update

{ new: true, runValidators: true } // return updated data and validate

);

**How it works:**

* The method searches for a user by the given ID.
* If found, it updates the fname field with "Updated Name".
* It returns the updated document because of { new: true }.

#### 3. find({}) Function Structure

#### **Syntax:**

Model.find(query, projection, options);

#### **Explanation of Parameters:**

1. query: An object that specifies conditions to filter documents (empty {} means no filters, returning all documents).
2. projection (optional): Specifies which fields to include/exclude.
3. options (optional): Additional query options such as sorting, limiting, and pagination.

#### **Example:**

const users = await User.find({}, "fname lname", { limit: 5 });

**How it works:**

* {} as query means all records will be retrieved.
* "fname lname" projection means only fname and lname fields are returned.
* { limit: 5 } option means only 5 records will be returned.

#### 4. Clarifying const users = await User.find({})

#### **Why it uses User.find({}) and not users.find({})?**

* In your code:

const users = await User.find({});

**Explanation:**

* User is the **Mongoose model**, which represents the MongoDB collection.
* The find({}) method is called on the User model to retrieve data from the users collection in the database.
* The result of this query is stored in the variable users, which contains the list of retrieved user documents.

#### **Why not users.find({})?**

* The variable users only stores the data returned by User.find({}), but it does not have access to Mongoose methods like .find().
* The correct syntax is always Model.find() where Model is the schema model, such as User in this case.

#### **Example Breakdown:**

const users = await User.find({});

console.log(users); // Array of user objects from the database

**Step-by-step explanation:**

1. User.find({}) → Fetches all records from the database.
2. The result is stored in the variable users.

users now contains an array of all user documents, e.g.:  
[

{ \_id: '1', fname: 'John', lname: 'Doe' },

{ \_id: '2', fname: 'Jane', lname: 'Doe' }

]

1. If you tried users.find({}), it would result in an error because users is just an array, not a Mongoose model.

| **Concept** | **Explanation** |
| --- | --- |
| async | Declares an asynchronous function that returns a Promise implicitly. |
| await | Waits for an asynchronous operation to complete before continuing. |
| findByIdAndUpdate() | Updates a document by ID and returns the updated document if { new: true } is passed. |
| find({}) | Retrieves documents from the database based on filter criteria (empty {} returns all). |
| User.find({}) | Correct usage, because User is the Mongoose model that interacts with the database. |

### To install dotenv:

**$npm i dotenv –save**

### To install encryption package to protect password:

**$npm i bcrypt –save**

### To install jsonwebtoken to generate token to access password:

**$npm i jsonwebtoken –save**

### Bcrypt Password Encryption - Key Notes

1. Why Encrypt Passwords?

* Protects against unauthorized access.
* Prevents rainbow table and brute-force attacks.
* Ensures passwords are stored securely (hashed, not plain text).

2. Steps in Password Encryption

Step 1: Generate Salt

const salt = await bcrypt.genSalt(10);

* Adds randomness to prevent duplicate hashes.
* 10 is the salt round (higher = more secure, but slower).

Step 2: Hash the Password

const hashedPassword = await bcrypt.hash(req.body.password, salt);

* Combines salt with password and applies hashing.
* Hashed password stored securely in the database.

3. Password Verification

**const isMatch = await bcrypt.compare(req.body.password, storedHashedPassword);**

* **Compares input password with stored hash.**
* **Returns true if they match, else false.**

4. Key Benefits of Bcrypt

* **Adaptive hashing (slows down with increasing computation power).**
* **Automatic salting (no need to manually add salt).**
* **Slowness for security (mitigates brute-force attacks).**

5. Important Considerations

* **Always use app.use(express.json()); to parse request bodies.**
* **Never store plain-text passwords.**
* **Recommended salt rounds: 10-12 for a balance between security and performance.**
* **Use async/await to avoid blocking the server.**

## Authentication: /routes/api/users.js:

const express = require('express');

const router = express.Router();

const User = require('../../models/User')

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

//? Create a new user

router.post('/', async (req, res) => {

try {

// 🔐 Password Security Steps:

// 1. Make random 'salt' to mix with password (like special seasoning)

const salt = await bcrypt.genSalt(10); // 10 = security strength

// 2. Hash password = password + salt → scrambled text

const password = await bcrypt.hash(req.body.password, salt);

// 📦 Prepare user data package

const userObj = {

fname: req.body.fname,

lname: req.body.lname,

email: req.body.email,

password: password, // Store HASHED password, never raw password!

}

// 💾 Save to database

const user = await User(userObj) // Create new user document

await user.save(); // Actually save to database

// ⚠️ SECURITY WARNING: We're sending back hashed password!

// Should remove password before sending response

return res.status(201).json(user) // 201 = Created success status

} catch (err) {

// 🚨 Handle errors (like duplicate email or database issues)

res.status(500).json({ message: "Something wrong " }); // 500 = Server error

}

});

// 1. Route Setup

// router.post('/', async (req, res) => {

// What it does: Handles POST requests to the website's root URL (/)

// Like: A registration form submission handler

// 2. Password Security (Most Important Part!)

// const salt = await bcrypt.genSalt(10);

// const password = await bcrypt.hash(req.body.password, salt);

// Step 1: Makes a "salt" - random data for password protection

// genSalt(10) = Security level (higher number = more secure but slower)

// Step 2: Mixes the user's password with the salt to create a secure hash

// Why: Never store passwords as plain text!

// 3. Building the User

// const userObj = {

// fname: req.body.fname,

// lname: req.body.lname,

// email: req.body.email,

// password: password, // This is the hashed password now

// }

// What's happening: Creates a user object with:

// First name

// Last name

// Email

// Securely hashed password (not the real password!)

// 4. Saving to Database

// const user = await User(userObj)

// await user.save();

// Step 1: User(userObj) creates a new user document

// Step 2: user.save() stores it in the database

// 5. Success Response

// return res.status(201).json(user)

// 201 Status: "Created" success message

// Sends back: The created user data (but we'll talk about this later)

// 6. Error Handling

// catch (err) {

// res.status(500).json({ message: "Something wrong " });

// }

// Catches: Any errors that happen in the process

// Responds: Generic error message (500 = Internal Server Error)

//? Login

router.post('/login', async (req, res) => {

try {

const { type, email, password, refreshToken } = req.body;

// Currently only handling email/password login

if (type === 'email') {

// 🔍 Find user by email (like looking up in phonebook)

const user = await User.findOne({ email: email });

if (!user) {

return res.status(404).json({ message: "user not found" }); // 404 = Not found

}

else {

// 🔑 Verify password & handle login

await handleEmailLogin(password, user, res)

}

}

else {

if (!refreshToken) {

return res.status(404).json({ message: "Refresh token not found" });

}

else {

await handleRefreshToken(refreshToken, res);

}

}

} catch (error) {

res.status(500).json({ message: "Something wrong " });

}

});

// This is a login route handler:

// router.post('/login', async (req, res) => {

// It handles POST requests to the /login URL

// async means it contains asynchronous operations (like database queries)

// req is the incoming request, res is the response we'll send back

// Getting user input:

// const { type, email, password, refreshToken } = req.body;

// Extracts data from the request body (what the user sent)

// Possible fields: type, email, password, refreshToken

// Check the login type:

// if (type === 'email') {

// The code currently only handles email/password login

// There might be other types (like 'google' or 'facebook') in the future

// Find the user:

// const user = await User.findOne({ email: email });

// Searches the database for a user with this email

// await means we wait for the database response before continuing

// Handle user not found:

// if (!user) {

// return res.status(404).json({ message: "user not found" });

// }

// If no user exists with that email:

// Send back 404 status (Not Found)

// Return a JSON error message

// return stops further execution

// Handle existing user:

// else {

// await handleEmailLogin(password, user, res)

// }

// If user exists, call a helper function handleEmailLogin

// This function would typically:

// Compare passwords

// Generate authentication tokens

// Send response back to client

// Error handling:

// } catch (error) {

// res.status(500).json({ message: "Something wrong " });

// }

// Catches any errors that occur in the try block

// Sends a generic 500 error (Internal Server Error)

// Note: In real applications, you might want to log the actual error

//?Get user profile

router.get('/profile', authenticateToken, async (req, res) => {

try {

// Step 1: Get the user's ID from the authenticated request

const id = req.user.\_id;

// Step 2: Find the user in the database using their ID

const user = await User.findById(id);

// Step 3: If the user exists, return their profile

if (user) {

return res.json(user);

}

// Step 4: If the user doesn't exist, return a 404 error

else {

return res.status(404).json("user not found");

}

}

// Step 5: Handle any unexpected errors

catch {

res.status(500).json({ message: "Something went wrong" });

}

});

//? Get all users

router.get('/', async (req, res) => {

try {

const users = await User.find({})

return res.status(200).json(users)

}

catch (err) {

res.status(500).json({ message: "Something went wrong" });

}

})

//?Get one user

router.get('/:id', async (req, res) => {

try {

const id = req.params.id

const user = await User.findById(id)

if (user) {

return res.json(user)

}

else {

return res.status(404).json("user not found")

}

}

catch {

res.status(500).json({ message: "Something went wrong" });

}

})

//? Update 1 user

router.put('/:id', async (req, res) => {

try {

const id = req.params.id;

const userBody = req.body

const updatedUser = await User.findByIdAndUpdate(id, userBody, { new: true })

// only fname update korte chaile (id, fname, {new: true})

// update kore old value e dekhate chaile (id, fname)

if (updatedUser) {

return res.json(updatedUser)

}

else {

return res.status(404).json("user not found")

}

}

catch {

res.status(500).json({ message: "Something went wrong", error: err.message });

}

})

//? Delete a user

router.delete('/:id', async (req, res) => {

try {

const id = req.params.id;

const deletedUser = await User.findByIdAndDelete(id)

if (deletedUser) {

return res.json({ "following user deleted": deletedUser })

}

else {

return res.status(404).json("user not found")

}

} catch (err) {

res.status(500).json({ message: "Something went wrong", error: err.message });

}

});

module.exports = router

// 🔑 Helper function for email/password login

async function handleEmailLogin(password, user, res) {

// 🔐 Compare user input with stored hash

const isValidPassword = await bcrypt.compare(password, user.password)

if (isValidPassword) {

// ✅ Correct password: Create token package

const userObj = await generateUserObj(user)

return res.json(userObj);

}

else {

// ❌ Wrong password: Unauthorized access

return res.status(401).json({ message: "login failed" }); // 401 = Unauthorized

}

}

// What it does:

// Takes the password you entered and the user's stored password (which is encrypted)

// Uses bcrypt.compare() to check if they match

// If correct: Creates a special user object with security tokens

// If wrong: Sends "login failed" error (401 = Unauthorized)

// Key Concepts:

// Never store passwords as plain text (always encrypted/hashed)

// bcrypt is a library for safe password comparison

// 🎟️ Token Creation Helpers

function generateUserObj(user) {

// Create access/refresh tokens (like special event tickets)

const { accessToken, refreshToken } = generateToken(user);

// Convert MongoDB user document to plain object

const userObj = user.toJSON()

// Add tokens to user object

userObj['accessToken'] = accessToken // Short-lived token (1 day)

userObj['refreshToken'] = refreshToken // Long-lived token (20 days)

return userObj;

}

// What it does:

// Gets security tokens from generateToken()

// Converts database user data to a simple JSON object

// Adds the tokens to this object

// Returns the final package ready to send to the client

// Why this matters:

// Tokens act like temporary digital keys for accessing protected features

// Separates database data from what we send to the client

// 🔏 Token Generation (JWT)

function generateToken(user) {

// Access Token (daily use)

const accessToken = jwt.sign(

{ email: user.email, \_id: user.id }, // Payload (user info)

process.env.JWT\_SECRET, // Secret key (like password for tokens)

{ expiresIn: '1d' } // Expires in 1 day

);

// Refresh Token (for getting new access tokens)

const refreshToken = jwt.sign(

{ email: user.email, \_id: user.id },

process.env.JWT\_SECRET,

{ expiresIn: '20d' } // Expires in 20 days

);

return { accessToken, refreshToken };

}

// What it does:

// Uses jsonwebtoken (JWT) library to create two tokens:

// Access Token: Short-lived (1 day) for daily use

// Refresh Token: Long-lived (20 days) to get new access tokens

// Key Concepts:

// jwt.sign(payload, secret, options) creates a token

// Tokens are like digital ID cards with expiration dates

// process.env.JWT\_SECRET is a secret key stored in your server's environment (like a password for your tokens)

function handleRefreshToken(refreshToken, res) {

jwt.verify(refreshToken, process.env.JWT\_SECRET, async (err, payload) => {

if (err) {

return res.status(401).json({ message: "UnauthorizedError" })

}

else {

const user = await User.findById(payload.\_id);

if (user) {

const userObj = generateUserObj(user);

return res.status(200).json(userObj);

} else {

return res.status(401).json({ message: "UnauthorizedError" })

}

}

})

}

### What the “jwt\_secret” actually do?

**What is JWT\_SECRET?**

It’s a secret key (like a password) that only your server knows.

It’s used to sign and verify JWTs.

Example: "mySuperSecretKey123!" (but much longer and more random in real apps).

**What does JWT\_SECRET actually do?**

The JWT\_SECRET has two main jobs:

**1. Signing Tokens (Creating JWTs)**

When a user logs in, the server creates a JWT using the JWT\_SECRET.

The JWT\_SECRET is used to generate a signature for the token.

The signature ensures the token hasn’t been tampered with.

Example:

const token = jwt.sign(

{ email: "alice@example.com" }, // Payload (user data)

process.env.JWT\_SECRET, // Secret key

{ expiresIn: '1d' } // Expires in 1 day

);

**2. Verifying Tokens (Validating JWTs)**

When the client sends a JWT back to the server, the server uses the JWT\_SECRET to verify the token.

The server checks:

Is the token’s signature valid? (Was it signed with the correct JWT\_SECRET?)

Has the token expired?

Has the token been tampered with?

Example:

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

**How does JWT\_SECRET work under the hood?**

**Step 1: Creating a JWT**

The server takes the payload (user data, e.g., email, ID).

It adds a header (describes the token type and algorithm).

It combines the header and payload into a string.

It uses the JWT\_SECRET to generate a signature for the string.

The final JWT is a combination of:

Header

Payload

Signature

Example JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFsaWNlQGV4YW1wbGUuY29tIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

**Step 2: Verifying a JWT**

The server receives the JWT from the client.

It splits the JWT into its three parts:

Header

Payload

Signature

It uses the JWT\_SECRET to re-calculate the signature for the header and payload.

It compares the re-calculated signature with the signature in the JWT:

If they match → The token is valid.

If they don’t match → The token is invalid (tampered with or signed with the wrong key).

**Why is JWT\_SECRET necessary?**

Prevents Tampering:

Without the JWT\_SECRET, anyone could modify the payload (e.g., change the email or ID) and create a fake token.

The signature ensures the token hasn’t been altered.

Ensures Authenticity:

Only your server knows the JWT\_SECRET, so only your server can create valid tokens.

Clients can’t forge tokens.

Stateless Authentication:

The server doesn’t need to store tokens. Instead, it embeds user info in the token and verifies it using the JWT\_SECRET.

**Real-World Analogy:**

Imagine you’re issuing tickets for a concert:

JWT\_SECRET: Your special ink stamp.

Token: A ticket with:

Header: "This is a valid ticket."

Payload: "Alice can enter until 8 PM."

Signature: Your stamp on the ticket.

Verification:

At the door, the bouncer checks:

Is the stamp real? (Uses JWT\_SECRET to verify the signature.)

Is the ticket still valid? (Checks the expiration time.)

### Use of access token:

We can view users profile by using access token which is generated at login time

### Differentiate between guest and registered user by access token

Explanation of Access Tokens for Guest and Registered Users

What is an Access Token?

An access token is a digital credential (like a temporary key) that grants a user permission to interact with specific resources or services in an application. It is often used in APIs and web services for authentication and authorization.

Why Access Tokens Are Needed

| **User Type** | **Purpose of Access Token** | **Example Use Cases** |
| --- | --- | --- |
| **Registered Users** | Grants full access to personalized resources after authentication (e.g., account data). | - Viewing profiles  - Making purchases  - Accessing private data |
| **Guest Users** | Provides limited access for basic interactions (no authentication required). | - Browsing public content  - Adding items to a cart  - Accessing trial features |

Key Differences

| **Feature** | **Registered Users** | **Guest Users** |
| --- | --- | --- |
| **Token Scope** | Full access to personal data and features. | Limited access to public/unrestricted features. |
| **Token Lifespan** | Long-lived (e.g., hours/days). | Short-lived (e.g., minutes/hours). |
| **Security Level** | High (linked to user identity). | Low (anonymous, no personal data). |

How It Works

1. **Registered Users**:
   * **Step 1**: Log in with email/password.
   * **Step 2**: Server issues an access token (e.g., JWT).
   * **Step 3**: Token is sent with every request to access protected resource

// Example: Token generation for registered users

**const token = jwt.sign(**

**{ userId: "123", role: "user" },**

**process.env.JWT\_SECRET,**

**{ expiresIn: "1d" });**

1. **Guest Users**:
   * **Step 1**: Visit the app without logging in.
   * **Step 2**: Server issues a temporary guest token (optional).
   * **Step 3**: Token allows limited actions (e.g., browsing

// Example: Token generation for guests

**const guestToken = jwt.sign(**

**{ sessionId: "abc", role: "guest" },**

**process.env.JWT\_SECRET,**

**{ expiresIn: "1h" });**

Why Guests Might Need Tokens

* **Session Management**: Track guest activity (e.g., cart items).
* **Rate Limiting**: Prevent abuse of public APIs.
* **Security**: Enforce token-based policies even for anonymous users.

### /middleware/auth.js:

// Import the jsonwebtoken library to work with JWTs (JSON Web Tokens)

const jwt = require('jsonwebtoken');

// Export a middleware function that will be used to authenticate requests

module.exports = function (req, res, next) {

// Step 1: Check if the request has an Authorization header

const authHeader = req.headers.authorization;

// If there's no Authorization header, block the request

if (!authHeader) {

// Send a 401 Unauthorized response with an error message

res.status(401).json({ message: 'Invalid authorization' });

}

// If the Authorization header exists, proceed to extract the token

else {

// Step 2: Extract the token from the Authorization header

// The header usually looks like: "Bearer <token>"

// Split the header by spaces and take the second part (the actual token)

const token = authHeader && authHeader.split(' ')[1];

// Step 3: Check if the token exists

if (token) {

// Step 4: Verify the token using the JWT\_SECRET

// jwt.verify() checks if the token is valid and not tampered with

jwt.verify(token, process.env.JWT\_SECRET, (err, payLoad) => {

// If there's an error (e.g., token is expired or invalid), block the request

if (err) {

// Send a 401 Unauthorized response with an error message

res.status(401).json({ message: 'Invalid authorization' });

}

// If the token is valid, proceed to the next step

else {

// Step 5: Attach the payload (user data) to the request object

// The payload contains the user's information (e.g., userId, email)

req.user = payLoad;

// Step 6: Call next() to pass control to the next middleware or route handler

next();

}

});

}

// If the token doesn't exist, block the request

else {

// Send a 401 Unauthorized response with an error message

res.status(401).json({ message: 'Invalid authorization' });

}

}

};

### Explanation:

Step 1: Check for the Authorization Header

const authHeader = req.headers.authorization;

* **What**: The client sends a token in the Authorization header (e.g., Bearer eyJhbGci...).
* **Why**: This is the standard way to send tokens in HTTP requests.
* **How**: Extract the header value to check for a token.

Step 2: Handle Missing Token

if (!authHeader) {

res.status(401).json({ message: 'Invalid authorization' });

}

* **What**: If there’s no token, block the request.
* **Why**: No token = No proof of identity.
* **How**: Send a 401 Unauthorized error.

Step 3: Extract the Token

const token = authHeader.split(' ')[1];

* **What**: Split the header value to get the token.
  + Example: Bearer abc123 → abc123.
* **Why**: The token is prefixed with Bearer by convention.
* **How**: Split the string by spaces and take the second part.

Step 4: Verify the Token

jwt.verify(token, process.env.JWT\_SECRET, (err, payload) => { ... });

* **What**: Use the server’s secret key (JWT\_SECRET) to validate the token.
* **Why**: Ensures the token wasn’t tampered with and is still valid.
* **How**:
  + jwt.verify decodes the token.
  + If valid → payload contains user data (e.g., userId, email).
  + If invalid → err explains why (expired, fake, etc.).

Step 5: Grant or Deny Access

if (err) {

res.status(401).json({ message: 'Invalid authorization' });

} else {

req.user = payload; // Attach user data to the request

next(); // Allow access to the protected route

}

* **Valid Token**:
  + Attach the user’s data to req.user (e.g., req.user.email).
  + next() passes control to the next middleware or route handler.
* **Invalid Token**:
  + Block access with a 401 Unauthorized error.

Key Concepts Explained

What is a JWT?

* A **JSON Web Token** (JWT) is a secure way to transmit user data between the client and server.
* Structure: Header.Payload.Signature
  + **Header**: Algorithm used (e.g., HS256).
  + **Payload**: User data (e.g., userId, email).
  + **Signature**: Ensures the token is valid (created using JWT\_SECRET).

What is JWT\_SECRET?

* A secret key **only the server knows** (stored in .env).
* Used to:
  + **Sign tokens**: Create the token’s signature.
  + **Verify tokens**: Confirm the token is authentic.

Why Use Bearer in the Header?

* It’s a convention to prefix tokens with Bearer to indicate the type of authentication.
* Example: Authorization: Bearer abc123.

Example Flow

1. **User Logs In**:
   * Server creates a JWT and sends it to the client.
2. **User Requests Protected Data**:
   * Client sends the JWT in the Authorization header.
3. **Middleware Checks Token**:
   * Valid token → Access granted.
   * Invalid token → Access denied.

Common Questions

Q: What if the token expires?

* The jwt.verify check will fail, and the user must log in again.

Q: Where is JWT\_SECRET stored?

* In a .env file (never in code!) to keep it secure:
* env
* Copy
* JWT\_SECRET=your\_super\_secret\_key\_here

Q: Why attach payload to req.user?

* So downstream routes/middleware can access the user’s data (e.g., req.user.email).

### Clarification:

#### Why attach payload to req.user?

The line req.user = payload; is a crucial part of the middleware. It attaches the **decoded payload** (user data) from the JWT to the req.user object. Let’s break down **why this is done** and **how it works** in simple terms.

What is payload?

* The payload is the **decoded data** from the JWT.
* It typically contains user information, such as:

{

"userId": "123",

"email": "user@example.com",

"role": "user"

* }
* This data is embedded in the JWT when the token is created (e.g., during login).

Why Attach payload to req.user?

1. Share User Data Across Middleware and Routes

* In Express, the req object is passed through all middleware and route handlers.
* By attaching the payload to req.user, you make the user’s data available to any subsequent middleware or route handler.
* Example:

app.get('/profile', (req, res) => {

console.log(req.user); // { userId: "123", email: "user@example.com" }

res.send(`Welcome, ${req.user.email}!`);

* });

2. Avoid Repeated Token Decoding

* Without attaching payload to req.user, you’d need to decode the JWT in every route handler that requires user data.
* Example of **bad practice**:

app.get('/profile', (req, res) => {

const token = req.headers.authorization.split(' ')[1];

const payload = jwt.verify(token, process.env.JWT\_SECRET); // Decode again

res.send(`Welcome, ${payload.email}!`);

* });
* This is inefficient and redundant.

3. Simplify Access Control

* You can use req.user to check permissions or roles in downstream middleware or routes.
* Example:

app.get('/admin', (req, res) => {

if (req.user.role === 'admin') {

res.send('Welcome, Admin!');

} else {

res.status(403).send('Access Denied');

}

* });

How Does It Work?

1. **Token Verification**:
   * The middleware decodes the JWT using jwt.verify.
   * If valid, the payload contains the user’s data.
2. **Attach Payload to req.user**:
3. req.user = payload;
   * This adds the user’s data to the req object.
4. **Use req.user in Routes**:
   * Any route handler or middleware after this can access req.user.

#### why this ?

const id = req.user.\_id;

why not:

const id = req.user.id;??

Great question! The difference between req.user.\_id and req.user.id depends on how the JWT payload is structured and how your database schema is designed. Let’s break it down:

**1. Why req.user.\_id?**

In MongoDB (and Mongoose), the unique identifier for a document is stored in the \_id field by default.

When you create a JWT payload, it’s common to include the user’s \_id from the database:

const token = jwt.sign(

{ \_id: user.\_id, email: user.email }, // Payload

process.env.JWT\_SECRET,

{ expiresIn: '1d' }

);

So, when the token is decoded in the middleware, req.user will contain:

req.user = { \_id: "123", email: "user@example.com" };

Therefore, you access the user’s ID using req.user.\_id.

**2. Why req.user.id?**

Some developers prefer to use id instead of \_id in the JWT payload for simplicity.

If the payload is structured like this:

const token = jwt.sign(

{ id: user.\_id, email: user.email }, // Payload

process.env.JWT\_SECRET,

{ expiresIn: '1d' }

);

Then, when the token is decoded, req.user will contain:

req.user = { id: "123", email: "user@example.com" };

In this case, you would use req.user.id.

**Which One Should You Use?**

It depends on how the JWT payload is created:

If the payload uses \_id:

Use req.user.\_id.

If the payload uses id:

Use req.user.id.

**Best Practice**

Be consistent with how you structure your JWT payload.

If you’re using MongoDB/Mongoose, it’s common to use \_id because that’s the default field name for the unique identifier in the database.

Example:

const token = jwt.sign(

{ id: user.\_id, email: user.email }, // Payload

process.env.JWT\_SECRET,

{ expiresIn: '1d' }

);

Then, in your route:

const id = req.user.\_id; // Access \_id

**Why Not Both?**

You can include both \_id and id in the payload if you want flexibility:

const token = jwt.sign(

{ \_id: user.\_id, id: user.\_id, email: user.email }, // Include both

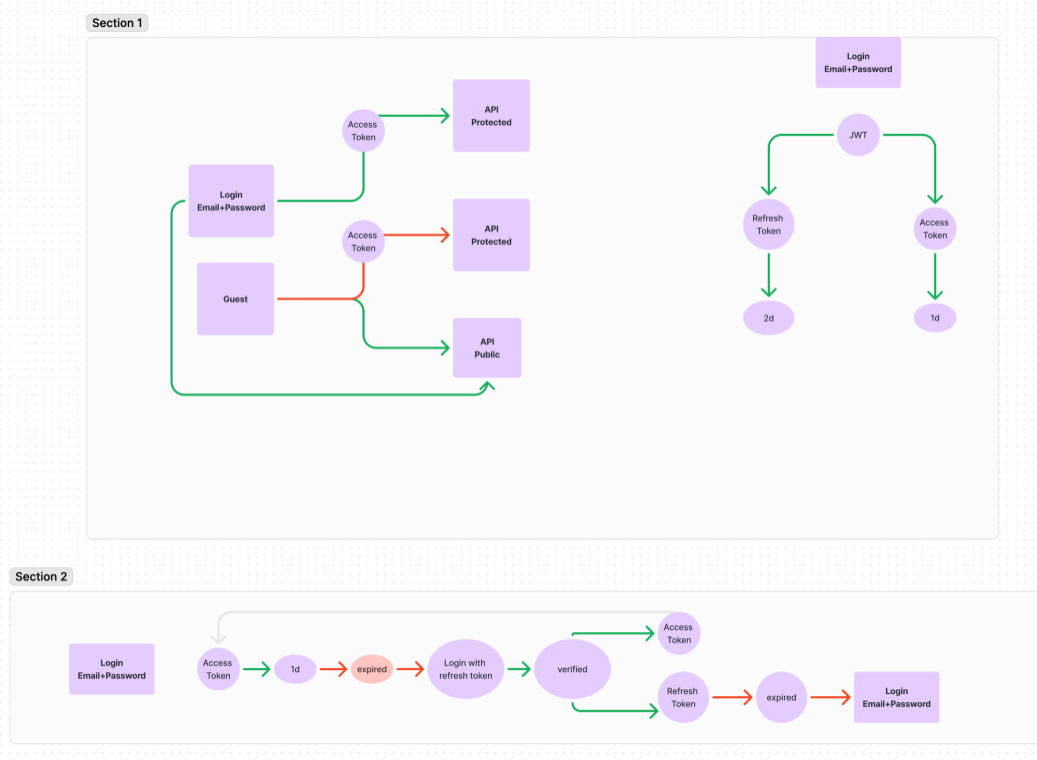
process.env.JWT\_SECRET,

{ expiresIn: '1d' }

);

Then, you can use either req.user.\_id or req.user.id in your code.

### Auth Flow:



### 

### Example for using another Object Reference:

We should write like this:  
**productId: {**

**type: mongoose.Types.ObjectId,**

**ref: "Product",**

**},**

The ref: "Product" establishes a relationship between the Order schema and the Product schema

const mongoose = require("mongoose");

const OrderSchema = new mongoose.Schema(

{

qty: {

type: Number,

},

total: {

type: Number,

},

userId: {

type: mongoose.Types.ObjectId,

ref: "user",

},

// productId: {

// type: mongoose.Types.ObjectId,

// ref: "Product",

// },

deliveryLocation: {

type: String,

},

expectedDeliveryDate: {

type: Date,

},

purchaseDate: {

type: Date,

},

deliveryStatus: {

type: String,

enum: ["delivered", "cancelled", "in-progress"],

default: "in-progress",

},

},

{

timestamps: true,

}

);

module.exports = mongoose.model("Order", OrderSchema);

### **Multer Definition:** Multer is a middleware for handling multipart/form-data, primarily used for uploading files in Node.js applications.

**Key Features:**

* Works with Express.js
* Supports single and multiple file uploads
* Allows file filtering and size limits
* Stores files in memory or disk

**Installation:**

npm install --save multer

**Basic Usage:**

// Configure storage settings for multer

const storage = multer.diskStorage({

// Set the destination folder for uploaded files

destination: function (req, file, cb) {

// Save files in the "/tmp/my-uploads" directory

cb(null, "/tmp/my-uploads");

},

// Define the naming convention for uploaded files

filename: function (req, file, cb) {

// Generate a unique suffix using the current timestamp and a random number

const uniqueSuffix = Date.now() + "-" + Math.round(Math.random() \* 1e9);

// Set the file name as the field name plus the unique suffix

cb(null, file.fieldname + "-" + uniqueSuffix);

},

});