1. Create server.js
2. // Import required modules
3. const express = require("express");
4. const fs = require("fs/promises"); // Async file handling
5. const jwt = require("onwebtoken");
6. const path = require("path");
7. const app = express();
8. const PORT = 3000;
9. const SECRET\_KEY = "secret"; // JWT signing key
10. app.use(express.json()); // Middleware to parse ON
11. // Serve static files from 'public' folder
12. app.use(express.static(path.join(\_\_dirname, "public")));
13. // Read & Write database (database.on)
14. const loadUsers = async () => {
15. try {
16. const data = await fs.readFile("database.on", "utf-8");
17. return ON.parse(data);
18. } catch (err) {
19. console.error("Error reading database.on", err);
20. return [];
21. }
22. };
23. const saveUsers = async (users) => {
24. try {
25. await fs.writeFile("database.on", ON.stringify(users, null, 2));
26. } catch (err) {
27. console.error("Error writing to database.on", err);
28. }
29. };
30. // Register Route
31. app.post("/signup", async (req, res) => {
32. const { name, password, work } = req.body;
33. const users = await loadUsers();
34. if (users.some(user => user.name === name)) {
35. return res.status(400).on({ error: "User already registered" });
36. }
37. users.pu({ name, password, work });
38. await saveUsers(users);
39. res.on({ message: "Successfully Registered" });
40. });
41. // Login Route
42. app.post("/login", async (req, res) => {
43. const { name, password } = req.body;
44. const users = await loadUsers();
45. const user = users.find(u => u.name === name && u.password === password);
46. if (!user) {
47. return res.status(401).on({ error: "Invalid name or password" });
48. }
49. const token = jwt.sign({ name: user.name, work: user.work }, SECRET\_KEY, { expiresIn: "1h" });
50. res.on({ token });
51. });
52. // JWT Middleware
53. const verifyToken = (req, res, next) => {
54. const token = req.headers["authorization"]?.split(" ")[1];
55. if (!token) return res.status(403).on({ error: "Access denied, token missing" });
56. jwt.verify(token, SECRET\_KEY, (err, decoded) => {
57. if (err) return res.status(403).on({ error: "Invalid token" });
58. req.user = decoded;
59. next();
60. });
61. };
62. // Protected Profile Route
63. app.get("/profile", verifyToken, (req, res) => {
64. res.on({ message: "Welcome to your profile", user: req.user });
65. });
66. // Start Server
67. app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));

**1️⃣ What is server.js?**

* server.js is a **backend program** that runs on a **computer** instead of a browser.
* It listens to requests from a web page and gives **responses**.
* It **stores** and **retrieves** user details from a file called database.on.
* It makes sure only the **right users** can log in.

📝 **Analogy**:  
Imagine a **school office** where students submit a form to register.

* The office checks if the student is **already registered**.
* If not, it **adds their name to the student list**.
* If yes, it **rejects the registration**.

**2️⃣ Code Explanation (Line by Line)**

**Step 1: Import Required Modules**

const express = require("express");

const fs = require("fs/promises"); // Async file handling

const jwt = require("webtoken");

const path = require("path");

🔹 **What is happening here?**

* express: A **framework** that makes it easy to create a server.js
* fs/promises: Reads and writes files **without stopping** the server.js
* jsonwebtoken (JWT): Used for **login security**.
* path: Helps find files on the computer.

📝 **Analogy**:  
Think of express as a **school teacher** who listens to students and replies.  
Think of fs as the **school register book**, which stores student details.

**Step 2: Create a Server**

const app = express();

const PORT = 3000;

const SECRET\_KEY = "secret"; // JWT signing key

🔹 **What is happening?**

* app is like the **server manager**.
* PORT = 3000 means the server will listen on **port 3000**.
* SECRET\_KEY is a **password** for login security.

📝 **Analogy**:  
The **server** is like the **school office**.  
The PORT is like the **office door number**.  
The SECRET\_KEY is like the **principal’s special key**—only trusted people can enter.

**Step 3: Middlewares (Helpers)**

app.use(express.json()); // Middleware to parse JSON

app.use(express.static(path.join(\_\_dirname, "public")));//Serve frontend

files

🔹 **What is happening?**

* express.json() allows the server to **understand data** from forms.
* express.static("public") allows the server to sh**ow HTML files**.

📝 **Analogy**:  
Imagine a **teacher who can read both English and Hindi**.  
This middleware helps the server **understand JSON data**.

**Step 4: Load and Save Users**

const loadUsers = async () => {

try {

const data = await fs.readFile("database.json", "utf-8");

return JSON.parse(data);

} catch (err) {

console.error("Error reading database.json", err);

return [];

}

};

const saveUsers = async (users) => {

try {

await fs.writeFile("database.json", JSON.stringify(users, null, 2));

} catch (err) {

console.error("Error writing to database.json", err);

}

};

🔹 **What is happening?**

* loadUsers(): Reads database.json (a file storing user details).
* saveUsers(users): Saves new users in database.json.

📝 **Analogy**:

* loadUsers() is like a **teacher checking the attendance book** to see which students are present.
* saveUsers() is like **writing new students' names** in the book.

**3️⃣ User Registration (Signup)**

app.post("/signup", async (req, res) => {

const { name, password, work } = req.body;

const users = await loadUsers();

if (users.some(user => user.name === name)) {

return res.status(400).json({ error: "User already registered" });

}

users.push({ name, password, work });

await saveUsers(users);

res.json({ message: "Successfully Registered" });

});

🔹 **What is happening?**

1. **User enters name, password, and work** in the signup form.
2. **Server checks if the user already exists** in database.json.
3. If **already exists**, it sh**ows an error**.
4. If **not exists**, the **user is saved** in database.json.

📝 **Analogy**:  
Imagine a **student trying to register** in school.

* If the student **already exists** in the register book, the office says **"Already registered!"** ❌
* If the student **is new**, the office writes their name down and says **"Registration successful!"** ✅

**4️⃣ User Login**

app.post("/login", async (req, res) => {

const { name, password } = req.body;

const users = await loadUsers();

const user = users.find(u => u.name === name && u.password === password);

if (!user) {

return res.status(401).on({ error: "Invalid name or password" });

}

const token =

jwt.sign({name:user.name,work:user.work},SECRET\_KEY,{expiresIn:"1h"});

res.json({ token });

});

🔹 **What is happening?**

1. The user enters their **name** and **password**.
2. The server **checks** if the details exist in database.on.
3. If wrong, it sends **"Invalid name or password"**.
4. If correct, the server **creates a token** (like a special key) to allow the user inside.

📝 **Analogy**:  
Logging in is like **a student entering the school gate**.

* If their **name and password are correct**, they **get an ID card** (token) 🎫.
* If incorrect, the guard says **"You cannot enter!"** ❌.

**5️⃣ Protect Profile Page (Only for Logged-in Users)**

const verifyToken = (req, res, next) => {

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

if (!token) return res.status(403).on({ error: "Access denied, token missing" });

jwt.verify(token, SECRET\_KEY, (err, decoded) => {

if (err) return res.status(403).on({ error: "Invalid token" });

req.user = decoded;

next();

});

};

🔹 **What is happening?**

* If a user **tries to access the profile page**, the server **checks if they have a token**.
* If no token, **access is denied**.
* If token is valid, they **can see their profile**.

📝 **Analogy**:  
A **student with an ID card (token)** can enter the **library**. 📚  
If they **forget their ID card**, the librarian **does not allow them inside**. ❌

**6️⃣ Profile Route (Only for Logged-in Users)**

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

res.on({ message: "Welcome to your profile", user: req.user });

});

🔹 **What is happening?**

* If the user **has a valid token**, they can see their profile.
* Otherwise, **they are blocked**.

📝 **Analogy**:  
A **teacher checking the ID card** before allowing a student into a **special class**.

**7️⃣ Start the Server**

app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));

🔹 **What is happening?**

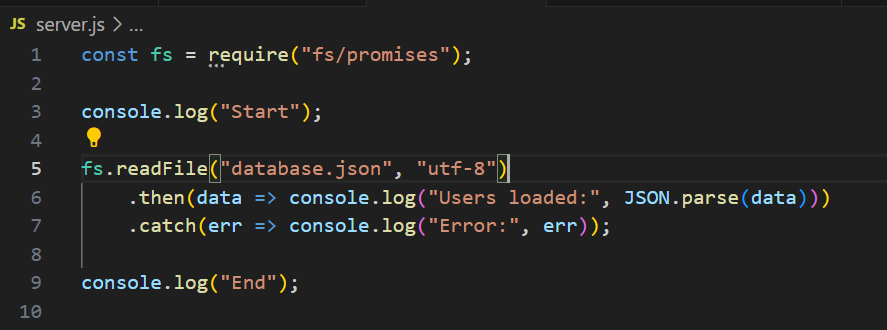
* The server **starts running on port 3000**.

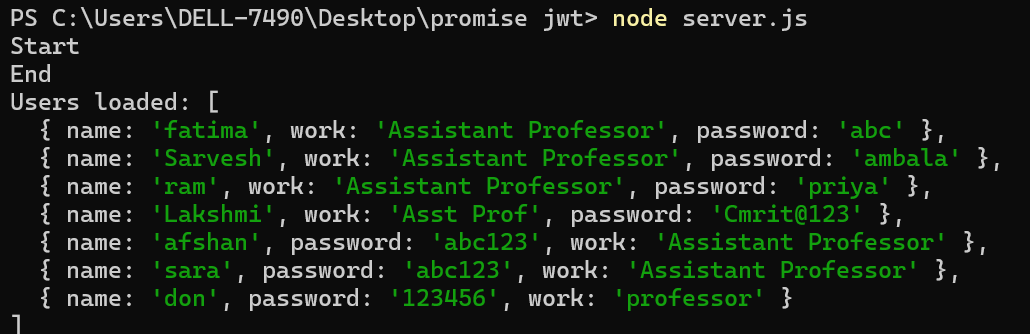
📝 **Analogy**:  
The **school office opens** at **9:00 AM** so students can register.

## **1️⃣ What is a Promise?**

A **Promise** is an object that represents a task that will be completed **later**.  
It can have **three states**:

* **Pending** (Not finished yet)
* **Resolved** (Finished successfully ✅)
* **Rejected** (Failed ❌)

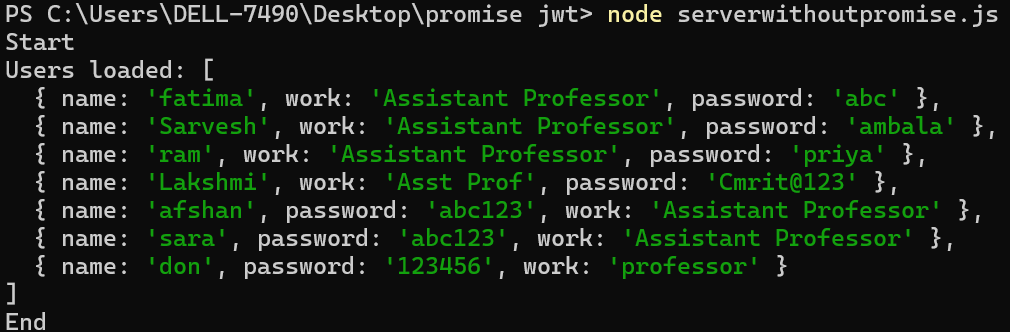




Here because of promise javaScript didn’t stop at the step of reading file, instead it went for another step that is to console.log 🡪 “end”and then continued with reading file step.

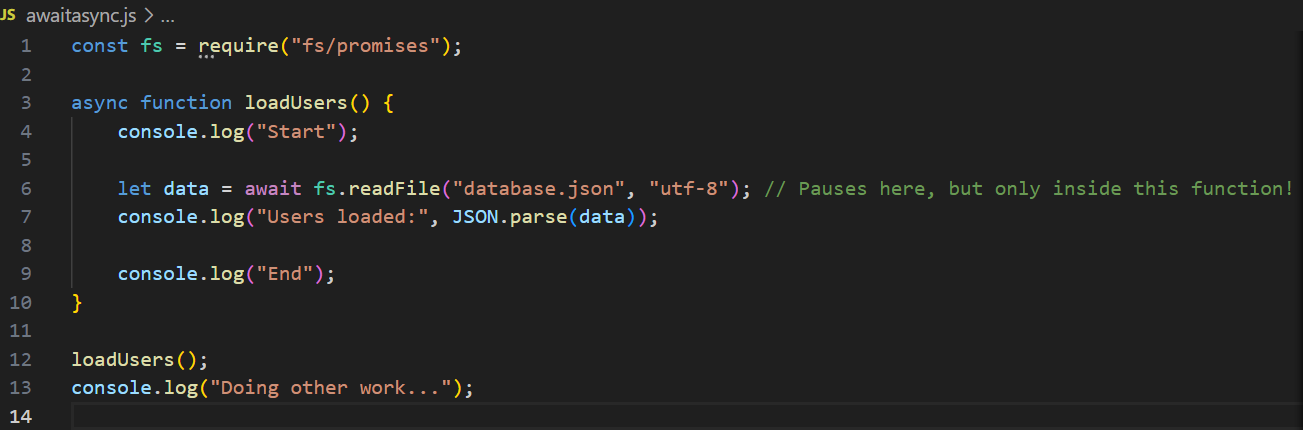
Below is the example of program without using promise

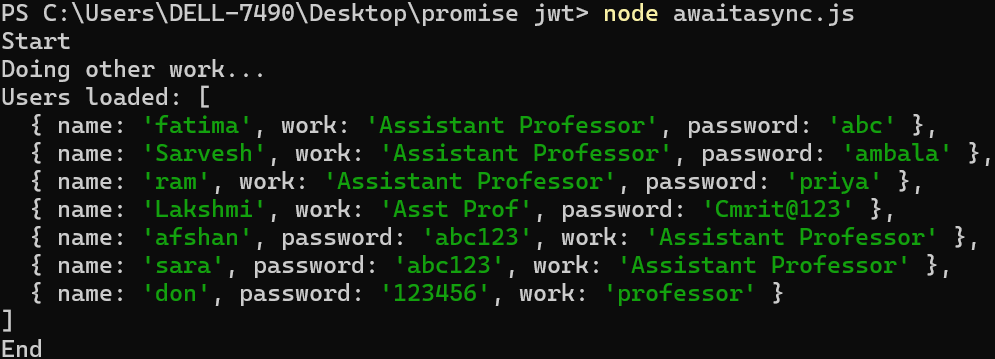




## **2️⃣ What is Async and Await?**

🔹 async makes a function **always return a Promise**.  
🔹 await **pauses execution** inside an async function **until the Promise is resolved**.





### ****What is**** await ****doing?****

await is a special keyword in JavaScript that **pauses the execution** of an async function **until a Promise is resolved or rejected**. It makes **asynchronous code look and behave like synchronous code**.

### ****🛠 How does this work?****

1. "Start" is printed.
2. The function **pauses at await fs.readFile(...)** until the file is completely read.
3. Only after the file is read, "Users loaded:" is printed.
4. "End" is printed.

✅ **Why is this better?**

* The code **waits properly** before moving ahead.
* No need for .then() – it's easier to read.
* If something goes wrong, we can handle it with try...catch.

## **Final Answer: Why Use** await **If We Want Asynchronous Code?**

✅ **Promises (Async)** allow JavaScript to keep working while waiting for slow tasks (like file reading).  
✅ **await makes sure we wait only inside a specific function**, without stopping the entire program.  
✅ **This prevents messy .then() chains** while keeping **the server responsive** for multiple users.

### 🚀 ****Step-by-Step Procedure to Execute Your JWT-Based Node. Server****

Follow these steps to **run your authentication server** using **Node., Express, and JWT**.

### ✅ ****Step 1: Install Node. (Skip if already installed)****

If you haven't installed Node., download and install it from:  
🔗 [https://node.org](https://nodejs.org)

Check installation by running:

node -v

npm -v

### ✅ ****Step 2: Initialize the Project****

If you haven’t already initialized a Node. project, run:

npm init -y

This creates a package. file.

### ✅ ****Step 3: Install Required Packages****

In your project folder, install the necessary dependencies:

npm install express webtoken fs

* express → To create the web server
* webtoken → For JWT authentication
* fs → For reading and writing user data

### ✅ ****Step 4: Ensure Your Project Files Are Correct****

You should have the following files:

📄 server.js → (Your Node. server)  
📄 database. → (User database for login/register)  
📄 package. → (Handles dependencies)  
📂 public/ → (Frontend files if needed)

* 1. **Index.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Home</title>

</head>

<body>

    <center>

        <h2>Welcome!</h2>

        <a href="/signup.html">Sign Up</a> |

        <a href="/login.html">Login</a>

    </center>

</body>

</html>

* 1. signup.html
* <!DOCTYPE html>
* <html lang="en">
* <head>
* <meta charset="UTF-8">
* <title>Sign Up</title>
* <script>
* async function registerUser(event) {
* event.preventDefault();
* const name = document.getElementById("username").value;
* const password = document.getElementById("password").value;
* const work = document.getElementById("work").value;
* const response = await fetch("/signup", {
* method: "POST",
* headers: { "Content-Type": "application/on" },
* body: JSON.stringify({ name, password, work })
* });
* const data = await response.json();
* alert(data.message);
* window.location.href = "/login.html";
* }
* </script>
* </head>
* <body>
* <center>
* <h2>Sign Up</h2>
* <form onsubmit="registerUser(event)">
* <input type="text" id="username" placeholder="Name" required><br>
* <input type="text" id="work" placeholder="Profession" required><br>
* <input type="password" id="password" placeholder="Password" required><br>
* <button type="submit">Register</button>
* </form>
* </center>
* </body>
* </html>

### ****🔹 JavaScript Code (Handles Signup)****

async function registerUser(event) {

✅ Defines an **async function** called registerUser().  
✅ This function will be called **when the user submits the form**.

event.preventDefault();

✅ **Prevents the default page reload** when submitting the form.  
✅ This allows us to **handle the signup process using JavaScript** instead of the default browser behavior.

const name = document.getElementById("username").value;

const password = document.getElementById("password").value;

const work = document.getElementById("work").value;

✅ **Gets the values** entered in the form fields and stores them in variables.

const response = await fetch("/signup", {

method: "POST",

headers: { "Content-Type": "application/on" },

body: JSON.stringify({ name, password, work })

});

✅ **Sends user data** (name, password, work) to the **server (/signup route)**.  
✅ **Uses fetch()** to make a **POST request** (because we are **sending data**).  
✅ **Headers:** "Content-Type": "application/on" tells the server **we are sending JSON data**.  
✅ **await waits for the response** from the server before moving ahead.

const data = await response.node json();

✅ **Converts the server response into JSON format** so we can **read it easily**.

alert(data.message);

✅ **Shows a pop-up message** with the response from the server.js  
✅ Example: If registration is successful, it may show **"Successfully Registered"**.

window.location.href = "/login.html";

✅ **Redirects the user** to login.html after successful registration.

**JSON.stringify()**

**JSON.stringify({ name, password, work })** is used to **convert the JavaScript object into a JSON string**, so it can be sent to the server in a request.

🔹 **JavaScript Object (Before conversion)**

{ name: "Alice", password: "mypassword", work: "Teacher" }

🔹 **JSON String (After JSON.stringify())**

'{"name":"Alice","password":"mypassword","work":"Teacher"}'

✅ **Without JSON.stringify(), the server won’t understand the data correctly.**

3. Login.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Login</title>

    <script>

        async function loginUser(event) {

            event.preventDefault();

            const name = document.getElementById("username").value;

            const password = document.getElementById("password").value;

            const response = await fetch("/login", {

                method: "POST",

                headers: { "Content-Type": "application/json" },

                body: JSON.stringify({ name, password })

            });

            const data = await response.jsjson();

            if (data.token) {

                localStorage.setItem("token", data.token);

                window.location.href = "/profile.html";

            } else {

                alert("Invalid Credentials");

            }

        }

    </script>

</head>

<body>

    <center>

        <h2>Login</h2>

        <form onsubmit="loginUser(event)">

            <input type="text" id="username" placeholder="Username" required><br>

            <input type="password" id="password" placeholder="Password" required><br>

            <button type="submit">Login</button>

        </form>

    </center>

</body>

</html>

## **🔹 Understanding** loginUser(event) **Function**

async function loginUser(event) {

✅ **Defines an asynchronous function (async function)**.  
✅ This function will handle the **login process** when the form is submitted.  
✅ Since it is **asynchronous**, it can use await to wait for the **server response** without blocking the page.

### ****🔹 Prevent Default Form Submission****

event.preventDefault();

✅ This **prevents the default browser behavior** (which is to reload the page when the form is submitted).  
✅ **Why?** We want to handle the form **using JavaScript**, not let the browser reload.

### ****🔹 Get User Input from the Form****

const name = document.getElementById("username").value;

const password = document.getElementById("password").value;

✅ document.getElementById("username").value → **Gets the value** the user entered in the "Username" field.  
✅ document.getElementById("password").value → **Gets the value** from the "Password" field.

👀 **Example:**  
If a user enters:

* **Username:** "Alice"
* **Password:** "mypassword"

Then, these values are stored as:

const name = "Alice";

const password = "mypassword";

### ****🔹 Sending Login Data to the Server****

const response = await fetch("/login", {

method: "POST",

headers: { "Content-Type": "application/on" },

body: ON.stringify({ name, password })

});

✅ fetch("/login", { ... }) → Sends a **POST request** to the /login API endpoint on the **server.js**  
✅ method: "POST" → This tells the server that we are **sending data**.  
✅ headers: { "Content-Type": "application/on" } →

* This tells the server, **"Hey, I'm sending ON data!"**
* It ensures that the **server can correctly read the request**.  
  ✅ body: ON.stringify({ name, password }) →
* Converts the **JavaScript object** into a **ON string**.
* **Example Before ON.stringify():**

{ name: "Alice", password: "mypassword" }

* **After ON.stringify():**

on

'{"name":"Alice","password":"mypassword"}'

* **Why is this needed?** The server **expects data in ON format**.

### ****🔹 Processing Server Response****

const data = await response.json();

✅ response.json() → Converts the **server’s response** into a JavaScript object.  
✅ await ensures we **wait** until the response is fully received.

👀 **Example Response from the Server:**  
If login is successful, the server might return:

json

{

"token": "abcdefg123456"

}

Then, data will contain

{ token: "abcdefg123456" }

### ****🔹 Checking If Login Was Successful****

if (data.token) {

✅ **Checks if the server sent a token** (meaning login was successful).  
✅ If data.token exists, the user is **authenticated**.

### ****🔹 Storing the Token in Local Storage****

localStorage.setItem("token", data.token);

✅ **Saves the token in localStorage** so it can be used later (e.g., for accessing the profile page).  
✅ **Why store the token?**

* This allows the user to stay logged in without needing to enter the password every time.
* The token is used in future API requests to prove **"I'm logged in"**.

👀 **Example:**  
Now, the browser stores:

Key: "token"

Value: "abcdefg123456"

### ****🔹 Redirecting to Profile Page****

window.location.href = "/profile.html";

✅ If the login is successful, this **redirects the user to profile.html**.  
✅ **Why?** The profile page is **only for logged-in users**, so we take them there after a successful login.

### ****🔹 Handling Invalid Login****

else {

alert("Invalid Credentials");

}

✅ If data.token **does not exist**, the login **failed**.  
✅ The user sees a **pop-up alert**:  
**"Invalid Credentials"** 🚫  
✅ This means **either the username or password was incorrect**.

1. Profile.html
2. <!DOCTYPE html>
3. <html>
4. <head>
5. <meta charset="UTF-8">
6. <title>Welcome</title>
7. </head>
8. <body>
9. <center><h2>Welcome to your profile!</h2></center>
10. </body>
11. </html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### ✅ ****Step 5: Run the Server****

Start the server by running:

node server.js

OR  
If you added a **start script** in package.:

"scripts": {

"start": "node server.js"

}

**Expected Output:**

Server running on http://localhost:3000

### ✅ ****Step 6: Test the API Using Postman or CURL****

1️⃣ **Register a User**

* **Method:** POST
* **URL:** http://localhost:3000/register
* **Body ():**

{

"name": "testuser",

"password": "password123",

"work": "Engineer"

}

* **Expected Response:**

{ "message": "Successfully Registered" }

2️⃣ **Login to Get a JWT Token**

* **Method:** POST
* **URL:** http://localhost:3000/auth
* **Body ():**

{

"name": "testuser",

"password": "password123"

}

* **Expected Response:**

{ "token": "your\_jwt\_token\_here" }

3️⃣ **Access a Protected Route**

* **Method:** GET
* **URL:** http://localhost:3000/profile
* **Headers:**

makefile

Authorization: Bearer your\_jwt\_token\_here

* **Expected Response:**

{

"message": "Welcome to your profile",

"user": { "name": "testuser", "work": "Engineer" }

}

### ✅ ****Step 7: Debugging Issues****

If the server doesn't start:

* Run node server.js and check for errors.
* If port 3000 is busy, change PORT in server.js (e.g., 4000).
* If database. is missing, create it manually:[]
* If JWT token verification fails, ensure you send the token correctly.

## **Verification by client through the browser**

If you're using **Node.js with Express**, you need to serve your frontend files correctly.

### ****Step 1: Ensure Your Files are in the Right Folder****

* Put your frontend files (HTML, CSS, JS) inside a folder named **public** (or any other name you choose).

project-folder/

├── public/

│ ├── index.html

│ ├── signup.html

│ ├── login.html

│ ├── profile.html

│ ├── styles.css

│ ├── script.js

├── server.js

├── package.json

### ****Step 2: Modify**** server.js ****to Serve Static Files****

Add this line to your server.js:

const express = require("express");

const app = express();

const path = require("path");

// Serve static files from 'public' folder

app.use(express.static(path.join(\_\_dirname, "public")));

app.listen(3000, () => {

console.log("Server running on http://localhost:3000");

});

This tells Express to serve all files inside the public folder.

### ****Step 3: Restart the Server****

Stop the server (Ctrl + C) and restart it:

node server.js

### ****Step 4: Access Your Pages****

Now open your browser and visit:

* **http://localhost:3000/** (for index.html)
* **http://localhost:3000/signup.html** (for signup.html)