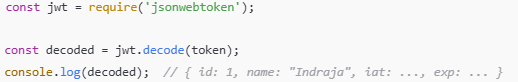
**🔐 JWT Token ante enti?**

JWT = **JSON Web Token**  
It is a **digitally signed string** (token) that is used to **verify the identity** of a user **without storing session data on the server**.

**Functions from the jsonwebtoken (jwt) library in Node.js:**

jwt.sign() – generates token

jwt.decode()- Only **parses** the token payload (header + body). **Does NOT check** if the token is tampered with.

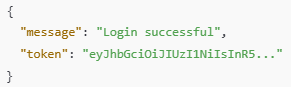
**Debugging or Logging (Use decode): Useful for logs or dev tools**.

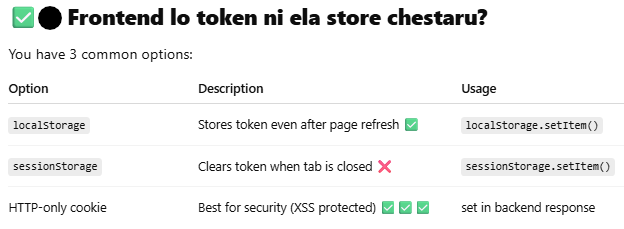
jwt.verify() - **Validates** the token signature using the secret key ( this should be same secret key which is used while generating JWT token). Checks if the token is **expired**, **tampered**, or **invalid**. Best for **authentication** and **authorization** logic.

**⚠️ Real-World Example:**



**✅ Token ante:**

* Oka **unique, encrypted string** ni backend lo generate chesi client ki istharu after successful login.   
  Ikkada, **client** ante **browser** or **frontend app**.  
  When the login is successful, server JWT token generate chesi client ki return chesthadi.  
    
  👉 This token is sent in the **HTTP response** and the **frontend (browser)** receives it.
* A token lo user yoka information untadi (like username, role, id).

  
**Example using localStorage (in React or plain JS):**



* A token ni future requests lo client attach chesthadu — so backend will know “who is requesting.”

✅ **Future Requests lo token attach cheyatam ela?**

When the browser sends any **protected request**, it must send the token in the

**headers** like this:

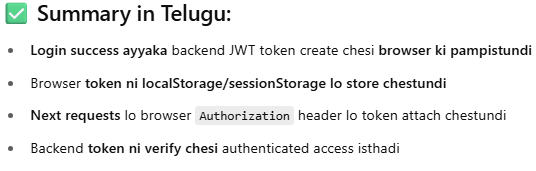


Or



✅ Backend lo token verify chesi allow cheyyadam ela?

Backend lo **authenticateToken** middleware untadi:  
 



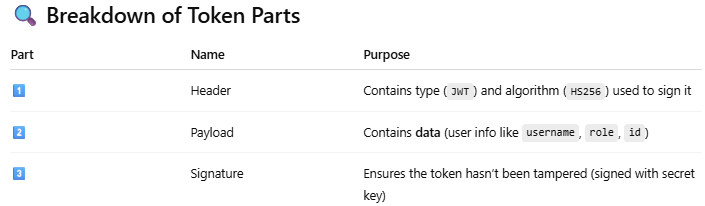
## ✅ Token Format

A JWT has 3 parts, separated by dots (.):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

.eyJ1c2VybmFtZSI6ImluZHJhamEiLCJyb2xlIjoiYWRtaW4ifQ

.7kqGH7Gf7p8lDsD4f0v2pI2UtQjYz7GlGzMlmZP-oWk



🔁 When JWT is Used?

### 🔓 Login lo:

1. User enters username & password
2. Server verifies details
3. Server creates a **JWT token** using secret key
4. Token is sent to the client

### 🔐 Protected API lo:

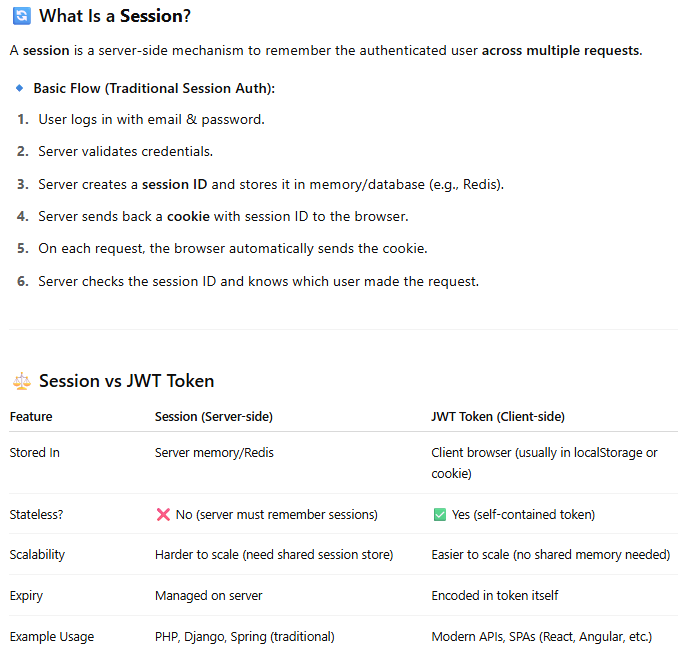
1. Client attaches token in request header:



1. Backend verifies token:

* If valid → allow access
* If invalid/expired → deny access

**What is Session?**



Building a **common** **authentication** and **authorization** **service** which can be leveraged by microservices developed in different programming languages(i.e., one service in python, one service in Nodejs etc)

Break this down clearly with a **real-world example using Netflix:**

✅ Basic Concepts First:

* **Authentication**: Are you really the person you claim to be? (login step)
* **Authorization**: What are you allowed to do? (permissions, roles)

### 📱 Scenario: Netflix App User Journey

### ****Client Use Case:****

You install the Netflix app and open it.

### 🔐 PHASE 1: AUTHENTICATION (Login)

#### A. New User:

1. You click on "Sign Up".
2. App sends **user data** (email, password, name) to the **Auth Service**.
3. Auth Service:
   * Hashes password.
   * Saves details in **User DB**.
   * Sends back a **JWT token** (or session) on success.

#### B. Existing User:

1. You click on "Sign In".
2. App sends **email & password** to the **Auth Service**.
3. Auth Service:
   * Verifies credentials.
   * Generates and returns a **JWT token** with:
     + userId
     + role
     + expiry
4. Client stores the JWT token (in localStorage or memory).

### 🔑 PHASE 2: AUTHORIZATION (Accessing other services)

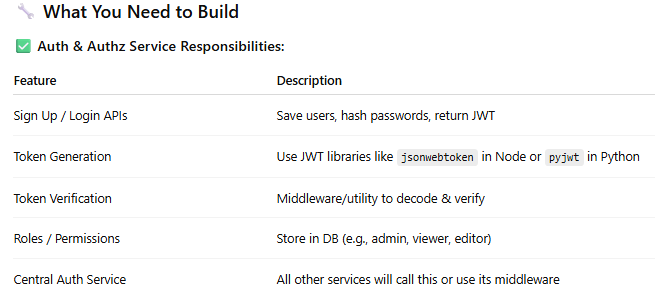
Let’s say Netflix has multiple microservices:

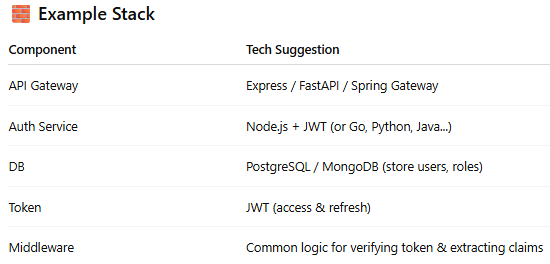
* movie-catalog-service
* billing-service
* profile-service
* watch-history-service

Each service doesn't handle login itself. Instead, they **rely on the token** from Auth Service.

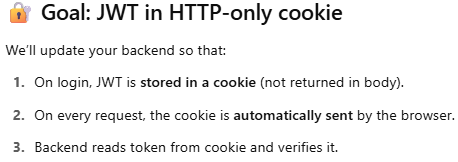
#### 🔁 Flow when making an API call:

1. User clicks **"Continue Watching"** (calls watch-history-service)
2. Request goes with token:  
   Authorization: Bearer <JWT-TOKEN>
3. watch-history-service sends token to **Auth Service** or **verifies JWT itself**.
4. Auth Service or middleware decodes the token:
   * Checks validity, expiry, and permissions.
5. If user is **authorized** → access granted
6. If user is **unauthorized** → 403 Forbidden





How to do **secure JWT-based login using HTTP-only cookies** — this is more secure than localStorage especially for **internal apps.**

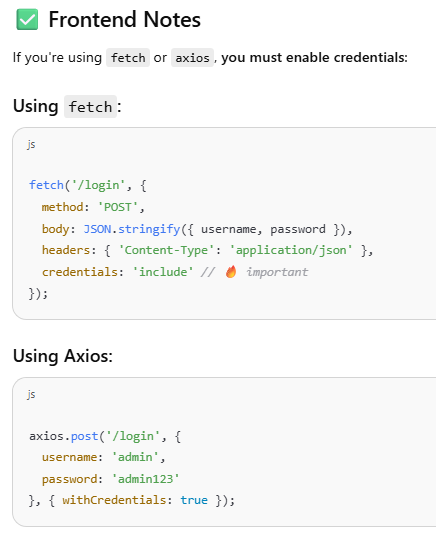


✅ Backend Example (Express.js + cookie-parser):

**NOTE:** Created a folder JWT usage for protected APIs, you can find either in **Git Hub** or **local system.**

1. npm install cookie-parser
2. **Code: index.js**

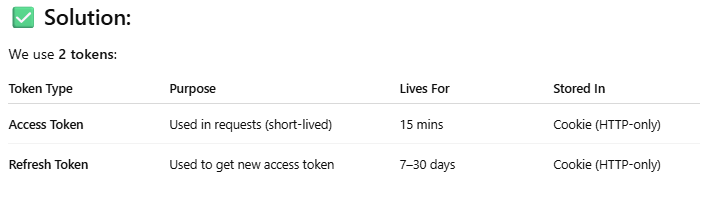
* const express = require('express');
* const bcrypt = require('bcrypt');
* const jwt = require('jsonwebtoken');
* const bodyParser = require('body-parser');
* const cookieParser = require('cookie-parser');
* const app = express();
* app.use(bodyParser.json());
* app.use(cookieParser());
* const SECRET\_KEY = 'supersecretkey';
* const PORT = 3000;
* **// In-memory user table**
* const users = [
* {
* id: 1,
* username: 'admin',
* passwordHash: bcrypt.hashSync('admin123', 10),
* role: 'admin'
* }
* ];
* **// 🔐 Login and set token in cookie**
* app.post('/login', async (req, res) => {
* const { username, password } = req.body;
* const user = users.find(u => u.username === username);
* if (!user) return res.status(404).json({ message: 'User not found' });
* const valid = await bcrypt.compare(password, user.passwordHash);
* if (!valid) return res.status(401).json({ message: 'Invalid credentials' });
* const token = jwt.sign({ id: user.id, role: user.role }, SECRET\_KEY, { expiresIn: '1h' });
* **// Set HTTP-only cookie**
* res.cookie('token', token, {
* httpOnly: true,
* secure: false, // Set true in production (HTTPS)
* sameSite: 'strict',
* maxAge: 3600000 // 1 hour
* });
* res.json({ message: 'Login successful' });
* });
* **// Middleware to check token from cookie**
* function authenticate(req, res, next) {
* const token = req.cookies.token;
* if (!token) return res.status(401).json({ message: 'No token' });
* jwt.verify(token, SECRET\_KEY, (err, decoded) => {
* if (err) return res.status(403).json({ message: 'Invalid token' });
* req.user = decoded;
* next();
* });
* }
* **// Protected route**
* app.get('/dashboard', authenticate, (req, res) => {
* res.json({ message: `Welcome to dashboard`, user: req.user });
* });
* **// Logout route (clear cookie)**
* app.post('/logout', (req, res) => {
* res.clearCookie('token');
* res.json({ message: 'Logged out' });
* });
* app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));



Let's now implement **token expiry and refresh logic** using **JWT + Refresh Tokens** — a best practice for long-lived user sessions.

## 🔐 Problem:

* JWT tokens usually expire in **15 mins to 1 hour** for security.
* If token expires, user is logged out — annoying! 😠



## ✅ Flow Summary

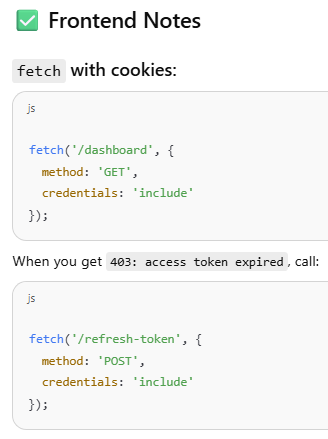
1. **Login success →** server gives 2 cookies: accessToken & refreshToken
2. Frontend uses accessToken (15 min) to access APIs
3. If token expired → client calls /refresh-token
4. Server checks refreshToken and gives **new accessToken**
5. No need to ask for credentials again!

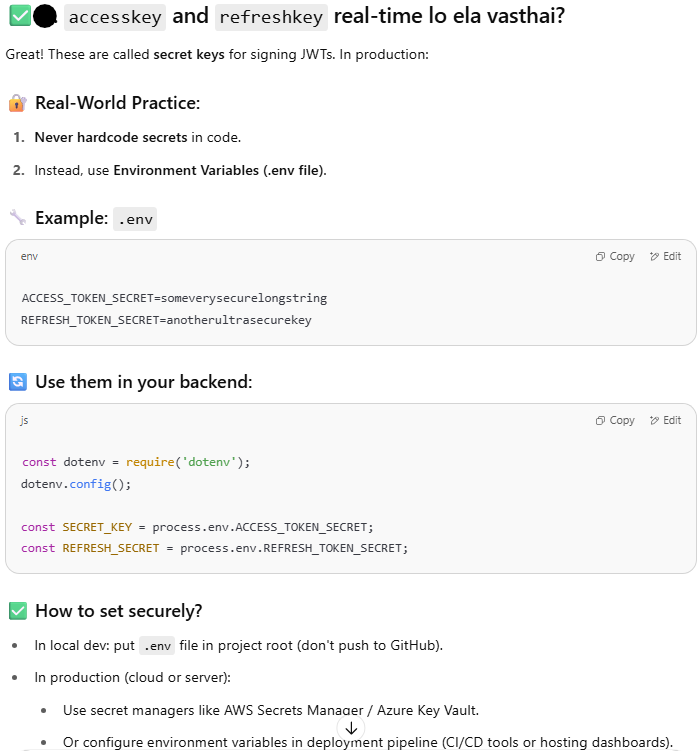
✅ Full **Node.js** Backend Code with Refresh Token

**NOTE:** Created a folder JWT usage for protected APIs, you can find either in **Git Hub** or **local system.**

1. **npm install express bcrypt jsonwebtoken cookie-parser**
2. Code (index.js)

* const express = require('express');
* const bcrypt = require('bcrypt');
* const jwt = require('jsonwebtoken');
* const cookieParser = require('cookie-parser');
* const app = express();
* app.use(express.json());
* app.use(cookieParser());
* const SECRET\_KEY = 'accesskey';
* const REFRESH\_SECRET = 'refreshkey';
* const PORT = 3000;
* // Simulate DB
* const users = [
* {
* id: 1,
* username: 'admin',
* passwordHash: bcrypt.hashSync('admin123', 10),
* role: 'admin'
* }
* ];
* let refreshTokens = []; // Store valid refresh tokens (in production use DB)
* function generateAccessToken(user) {
* return jwt.sign({ id: user.id, role: user.role }, SECRET\_KEY, { expiresIn: '15m' });
* }
* function generateRefreshToken(user) {
* const token = jwt.sign({ id: user.id }, REFRESH\_SECRET, { expiresIn: '7d' });
* refreshTokens.push(token);
* return token;
* }
* // 🟢 Login
* app.post('/login', async (req, res) => {
* const { username, password } = req.body;
* const user = users.find(u => u.username === username);
* if (!user || !(await bcrypt.compare(password, user.passwordHash))) {
* return res.status(401).json({ message: 'Invalid credentials' });
* }
* const accessToken = generateAccessToken(user);
* const refreshToken = generateRefreshToken(user);
* res.cookie('accessToken', accessToken, {
* httpOnly: true,
* maxAge: 15 \* 60 \* 1000
* });
* res.cookie('refreshToken', refreshToken, {
* httpOnly: true,
* maxAge: 7 \* 24 \* 60 \* 60 \* 1000
* });
* res.json({ message: 'Logged in successfully' });
* });
* // 🔒 Auth middleware
* function authenticate(req, res, next) {
* const token = req.cookies.accessToken;
* if (!token) return res.status(401).json({ message: 'Access token missing' });
* jwt.verify(token, SECRET\_KEY, (err, user) => {
* if (err) return res.status(403).json({ message: 'Access token expired' });
* req.user = user;
* next();
* });
* }
* // 🔄 Refresh Token Route
* app.post('/refresh-token', (req, res) => {
* const refreshToken = req.cookies.refreshToken;
* if (!refreshToken || !refreshTokens.includes(refreshToken)) {
* return res.status(403).json({ message: 'Invalid refresh token' });
* }
* jwt.verify(refreshToken, REFRESH\_SECRET, (err, user) => {
* if (err) return res.status(403).json({ message: 'Refresh token expired' });
* const newAccessToken = generateAccessToken(user);
* res.cookie('accessToken', newAccessToken, {
* httpOnly: true,
* maxAge: 15 \* 60 \* 1000
* });
* res.json({ message: 'Access token refreshed' });
* });
* });
* // 🔐 Protected route
* app.get('/dashboard', authenticate, (req, res) => {
* res.json({ message: 'Welcome to dashboard', user: req.user });
* });
* // 🚪 Logout
* app.post('/logout', (req, res) => {
* refreshTokens = refreshTokens.filter(t => t !== req.cookies.refreshToken);
* res.clearCookie('accessToken');
* res.clearCookie('refreshToken');
* res.json({ message: 'Logged out' });
* });
* app.listen(PORT, () => console.log(`Server running at [http://localhost:${PORT}`)](http://localhost:$%7bPORT%7d%60)));



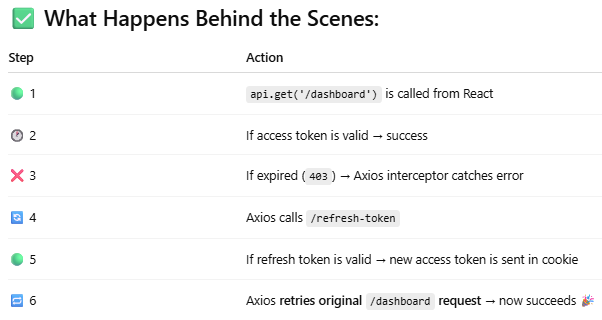


Access token expire ayinapudu refresh token ki call auto avvali frontend nundi — ela?

This should happen **automatically** from frontend using an **Axios Interceptor**. Interceptor observes responses. If token expired (403), it silently calls /refresh-token and retries the original request.✅ **Scenario Recap:**

1. **Login successful** → Server sends 2 cookies:
   * accessToken (expires in 15 mins)
   * refreshToken (expires in 7 days)
2. You call /dashboard API → ✅ works if access token is valid
3. After 15 mins, you call /dashboard → ❌ 403 - Access token expired
4. Axios **automatically** calls /refresh-token, gets new accessToken, and **retries the original request**

**🔧 Axios Interceptor Setup (React, etc.):**   

## ✅ Backend Code (Important Endpoints)

You must ensure your backend:

* Sends **accessToken** & **refreshToken** as HTTP-only cookies
* Implements /refresh-token route which:
  + Verifies refresh token
  + Sends new access token as a cookie

## 🔒 Security Tip

* Never store tokens in localStorage when using cookies
* Use httpOnly**,** secure: true in production
* Clear cookies on logout

Example:

🔄 How to Use Token Between APIs via Postman

Assume there is no frontend, Postman acts as the client. Here’s how to use the JWT token from Node API in a request to the Flask API.

The following is the setup involves creating a common authentication system where:

* Node.js API handles login and JWT token generation
* Flask API hosts protected resources
* There is no frontend, and you’re using Postman to simulate client requests.

✅ Objective:

You want to:

1. Login using /login on Node.js (port 3000)

2. Get a JWT token in the Postmen response

3. Send a Postmen request to Flask **/protected** (port 5000), using the token generated by Node /login API, to authorize the request.

✅ Step-by-Step:

1. Login to Node.js /login API

• POST to <http://localhost:3000/login> in Postmen

• In the response, you’ll receive:

• JWT token in:

• Response body OR

• Cookie (if you’re setting it using res.cookie())

**Example Node.js response in Postmen (with token):**

{ "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..." }

1. Use Token in Postmen Request to Flask API  
   Depending on how you’re sending the token (via headers or cookies), there are two options:

✅ Option A: Sending token as json response

In Node.js /login API code:

You send token as response like:



**NOTE:** Below is the code for JWT Authentication for Node /protected API:



**JWT Authentication for Flask /protected API code:**

**NOTE:** In **Python (using PyJWT)**, the method is always jwt.decode(), and it performs

both **decoding** and **verification**.



In Postman:

* Send POST request to <http://localhost:3000/login>, you get token as response
* Send a GET or POST request to <http://localhost:5000/protected>
* Go to Headers
* Add: Authorization: Bearer <your\_token\_from\_login>

✅ Option B: Cookies (if token sent via Set-Cookie)

In Node.js /login API:

Send token in cookie as:



In Flask:

Reading token in cookie as:



**In Postman:**

* After hitting /login, check **Cookies** tab → Node server should send token in Set-Cookie
* When making /protected call to Flask:
* Go to **Cookies** section in Postman
* Manually add the same cookie (token=your\_token) for localhost:5000

NOTE:

Cookie-based tokens need special attention to domains/ports and CORS settings.

Browser automatically attaches cookie to all requests.

**Testing Authentication and Authorization on APIs from frontend React APP instead of postmen:**

✅ Your Setup should be like

* Node.js API (port 3000) – for /login and token generation
* Flask API (port 5000) – protected route, needs token for auth
* React frontend (e.g., port 5173 with Vite or 3001 with CRA)

🔄 Goal

1. User logs in through React frontend
2. React sends credentials to Node /login
3. Node generates a JWT and returns it
4. React stores the token (either in cookies or localStorage)
5. When React calls Flask APIs, it sends the token in the Authorization header or via cookies
6. Flask verifies the token and allows or denies access

🔐 Two Approaches

✅ Option 1: Accessing JWT from Authorization Header (Recommended for APIs)



📦 Where token is stored:

• In memory or localStorage (temporary persistence)

⚙️ How React uses it:

• Send token in Authorization header as Bearer <token>



**  
Here,** React saves JWT from Node /login API into browsers localStorage

****

**Here,** When calling **Flask API**, React attaches the JWT in the Authorization header.

****

**Here,** Flask extracts and verifies the token from the header.

****

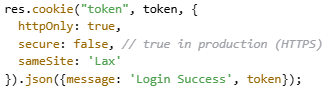
🍪 If you want to Use browser’s HTTP-only Cookie to store JWT Instead of LocalStorage?

✅ **Option 2: JWT in HTTP-Only Cookie (More secure)**

**NOTE:**

* JWT is sent from Node API in an HTTP-only cookie
* React doesn’t touch the token directly
* Browser automatically attaches cookie to all requests.
* Flask reads the token from the cookie.

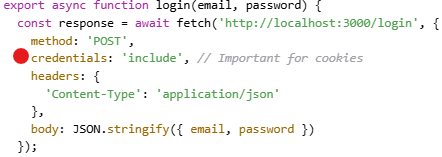
**Change above Node.js code to:**

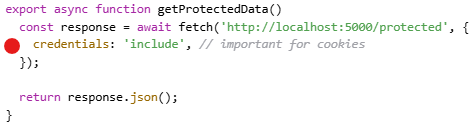
****

Change above Flask to:

****

In above React, use:

****

****

✅ Recommendation :

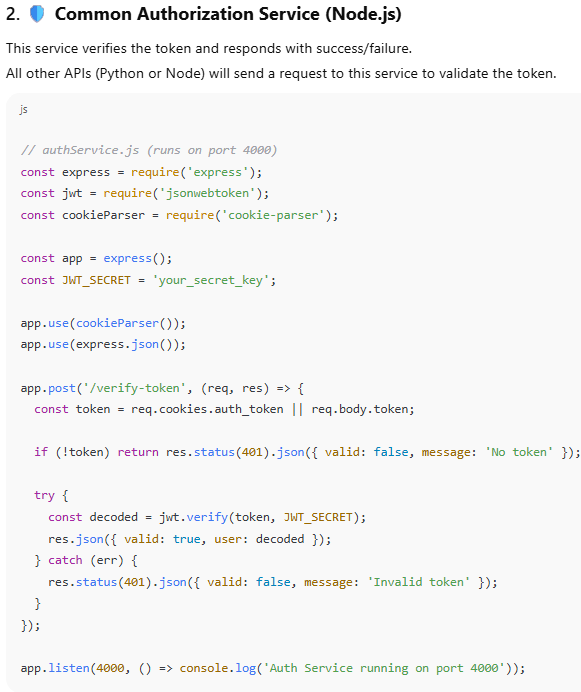
For APIs and token handling across different services (Node & Flask), use:

🔑 Header-based JWT approach

Header-based JWT : use browsers **localStorage**

Cookie-based JWT : use **HTTP-only Cookie**





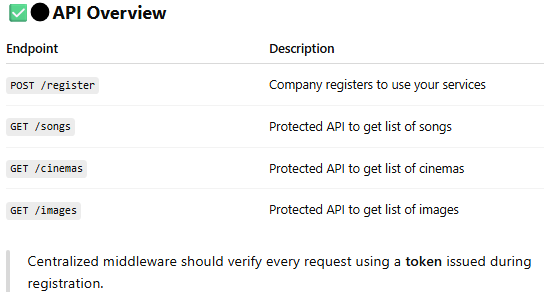




**Service Autherization – When someone can use our APIs in their application:**

Building a real-world, scalable **multi-tenant API service** where external companies like spotless.com can consume your APIs (like /songs, /cinemas, /images) **only after authorization using a token**.

Want to centralize this token-based auth logic and handle registration securely. Let's break this down and show how to implement it.

****

🔐 **How Token Auth Will Work:**

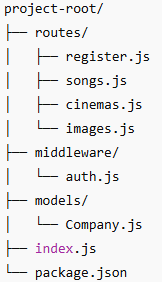
During /register, you’ll:

* Accept company details.
* Generate a **JWT token** (like API key).
* Store the token along with the company in DB.

For protected endpoints (/songs, /cinemas, etc.):

* Use centralized authMiddleware that:
  + Reads token from request header.
  + Validates it.
  + If valid, proceeds to service.

🔐 **Folder Structure :**

****

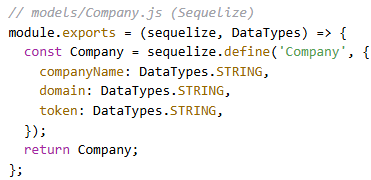
**📥 /register API Design:**

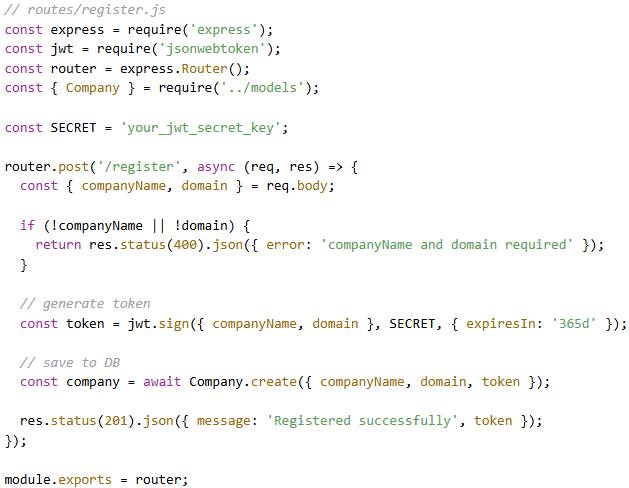
* Spotless admin visits your site: [www.brish.com/register](http://www.brish.com/register)
* The company (e.g., spotless.com) **fills the form on your site (brish.com) and submit**.

Expected Postmen Request:



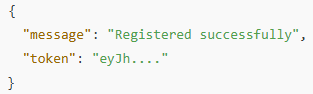
**Code: /register Code (Node.js + Express + JWT + Sequelize for SQL DB)**

****

****

**🧠 Logic Explanation:**

* + Save companyName and domain to DB.
  + We store the same token in our DB **to verify it later** when they make any request to /songs, /cinemas, etc.
  + Generate a JWT token with secret.
  + **Here, token will be their API Key** — it’s the key to access our protected services.
  + Save it with the company.
  + Return token as response.
  + **Where and How Will Company Store the Token?**
    - When you return the token in the /register response:

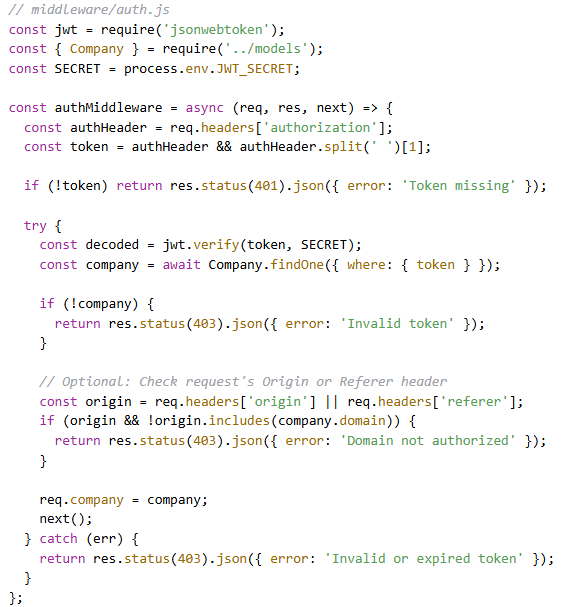
****

* + - Example, In spotless.com Web App, they will store this token in **environment config** or **backend secrets** like:



* + - **Now it's their responsibility** (the spotless.com admin) to **save this token securely** and **use it in all future API calls** to your system.
    - And whenever spotless wants to call your API, they **send this token as part of the request header**.

🔐 **Centralized** **Auth Middleware:**

****

**⚠️ Note:** This works best if their backend is making the request. But if they call directly from frontend (like browser), Origin/Referer can be spoofed, so prefer **server-to-server API usage.**

**🎵 Protected API Example: /songs**

****

✅ **Final Notes:**

* Store token securely.
* Use dotenv for secrets.
* Add token expiry, rate limiting later for security.

**Example code: How Does a Company Call Your Protected API?**

Here’s the code sample how spotless.com would call your /songs endpoint from their backend:

****

**What If They Share Their Token with Other Companies?**

API Security **loophole:**

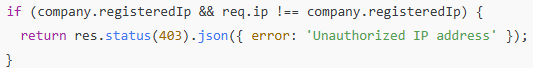
* What If Company A (spotless.com) shares its token and endpoint with Company B (which didn't register), Company B can still access your service!

**How to Fix This?**

* 1. When you generate the token, also include the company’s **domain** (e.g., spotless.com) inside the token payload.

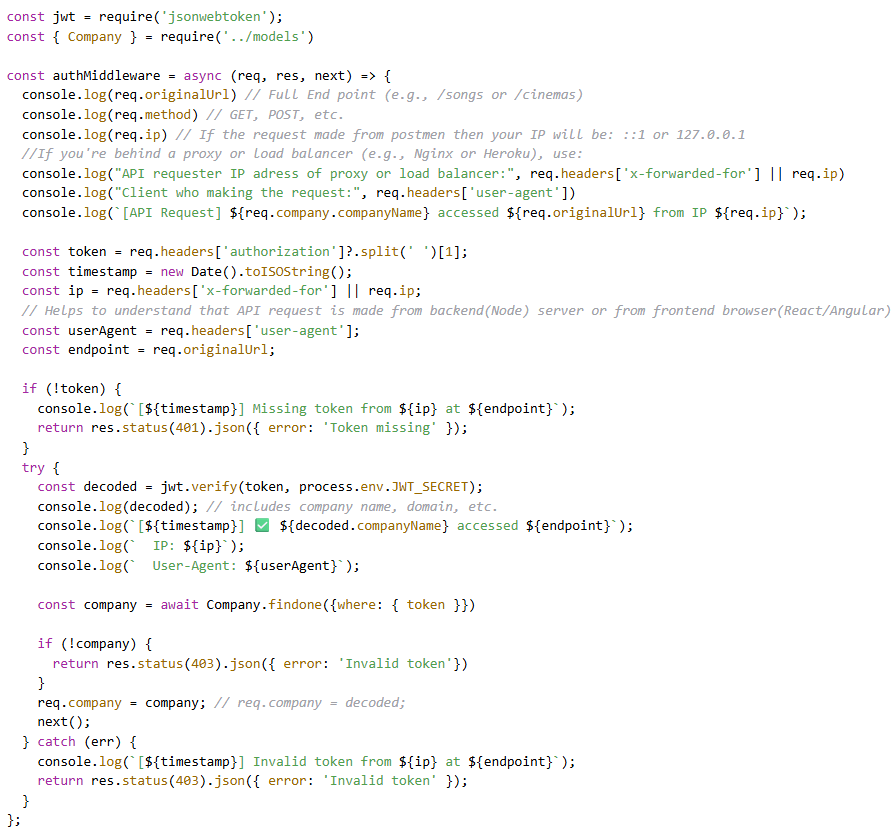
Then in your authMiddleware,check:

1. Is the token valid?
2. Is the **request origin domain** matching the domain we stored during registration?
   1. ✅ Bonus: Even Safer Options

* **IP Whitelisting** 
  + At the time of registration, ask companies to **provide their backend server IP address**.Allow each company to register their server IP and accept requests only from that IP.
  + Save that IP along with their token in your DB.
  + Then, in your middleware, allow only requests from that IP.  
    Code Sample:  
    ****

**Secured Autherization code:**

when a company’s backend server calls your API, **you’ll receive a full request and response object** in your Express route or middleware.

**Using these objects we can securely authorize and verify that our API Requester is our registered client or not with the following code:  
**