

# Protected Routes with JWT Verification Report

## Aim

To design and implement a secure mechanism for **restricting access to specific API endpoints (Protected Routes)** by verifying the authenticity and authorization of a user through a **JSON Web Token (JWT)**.

## Objective

The primary objective is to:

1. **Implement User Authentication:** Create a login endpoint that, upon successful credentials verification, generates and issues a JWT.
2. **Develop Middleware:** Create an Express middleware function responsible for extracting the JWT from the request header, verifying its signature and expiry, and either granting access or rejecting the request.
3. **Secure an Endpoint:** Apply the developed middleware to a designated API route, ensuring it is only accessible to authenticated users.

## Theory

### JSON Web Token (JWT)

A **JSON Web Token (JWT)** is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure.

A JWT consists of three parts, separated by dots (.):

1. **Header:** Contains the token type (JWT) and the signing algorithm (e.g., HMAC SHA256 or RSA).
2. **Payload:** Contains the claims, which are statements about an entity (typically the user) and additional data (e.g., user ID, username, expiry time exp).
3. **Signature:** Created by taking the encoded Header, the encoded Payload, and a **secret key** known only to the server, and running them through the specified algorithm.

The core principle of security lies in the **Signature**: the server can trust a token because it can verify the signature using its private secret. If a third party tampers with the Header or Payload, the signature check will fail, and the token will be rejected.

### Protected Routes and Middleware

A **Protected Route** is any API endpoint that requires a valid, authenticated token to be accessed. In a Node.js/Express environment, this is implemented using **Middleware**.

The authentication middleware executes *before* the route's primary handler. It performs these

steps:

1. Checks if the Authorization header is present.
2. Extracts the token (usually removing the "Bearer " prefix).
3. Uses the server's secret key to verify the token's signature and confirm it hasn't expired.
4. If verification succeeds, it decodes the payload, extracts the user ID, attaches it to the request object (req.user = decoded.userId), and calls next() to proceed to the main route handler.
5. If verification fails (invalid signature, expired token), it immediately sends a 401 Unauthorized response.

## Procedure (Using Node.js/Express)

1. **Setup:** Initialize a Node.js project and install necessary packages (express, jsonwebtoken, dotenv).
2. **JWT Generation (Login):** Define a /login route. If credentials are valid, use jwt.sign() to create a token containing the user's ID and return it to the client.
3. **Middleware Implementation:** Write the verifyToken middleware function to handle token extraction and verification using jwt.verify().
4. **Route Protection:** Define a sensitive route (e.g., /api/profile) and insert the verifyToken middleware before the route handler.

## Code (Illustrative Node.js/Express Implementation)

This example includes the token generation (Login) and the verification middleware, demonstrating the full cycle.

### 1. Server Setup and Routes ([server.js](#))

```
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();
const PORT = 3000;
const SECRET_KEY = 'your_super_secret_key_12345'; // Use environment variable in production!

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

// --- AUTHENTICATION MIDDLEWARE ---
const verifyToken = (req, res, next) => {
  // 1. Check for Authorization header
  const authHeader = req.headers.authorization;
  if (!authHeader) {
    return res.status(401).json({ message: 'Access denied. Token'

```

```

missing.' });
    }

    // 2. Extract token (remove "Bearer ")
    const token = authHeader.split(' ')[1];

    try {
        // 3. Verify token signature and expiry
        const decoded = jwt.verify(token, SECRET_KEY);

        // 4. Attach user info to request and proceed
        req.user = decoded;
        next();

    } catch (err) {
        // 5. Handle verification failure (expired, invalid signature)
        console.error('JWT verification failed:', err.message);
        return res.status(401).json({ message: 'Invalid or expired token.'
    });
    }
};

// --- PUBLIC ROUTE: Token Generation (Login) ---
app.post('/api/login', (req, res) => {
    // In a real app, you would verify req.body.username and
    req.body.password against the database.
    const { username, password } = req.body;

    if (username === 'testuser' && password === 'pass123') {
        // Payload: Contains claims about the user (e.g., user ID)
        const payload = { userId: 101, role: 'admin' };

        // Generate the token, set to expire in 1 hour
        const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });

        return res.status(200).json({

```

```

        message: 'Login successful',
        token: token
    });
}

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

// --- PROTECTED ROUTE: Requires verifyToken Middleware ---
app.get('/api/profile', verifyToken, (req, res) => {
    // This code only runs if the token is valid.
    // req.user contains the decoded payload (e.g., { userId: 101, role:
    'admin' })
    res.status(200).json({
        message: 'Welcome to your protected profile!',
        data: {
            userId: req.user.userId,
            role: req.user.role,
            accessGrantedAt: new Date().toISOString()
        }
    });
});

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

```

## 1. Unprotected Route (Login)

**Request:** `POST /api/login` with body: `{ "username": "testuser", "password": "pass123" }`

**Response:** Status 200 OK.

```

{
  "message": "Login successful",
  "token":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJlc2VybWludlwiWF0lIj0iLCJleHAiOiE3MDQwNzA4MDB9.EXAMPLE_SIGNATURE_STRING_"

```

```
HjK2t"  
}
```

#### #### 2. Failed Access (Protected Route without Token)

**\*\*Request:\*\*** `GET /api/profile` (No `Authorization` header)

**\*\*Response:\*\*** Status 401 Unauthorized.

```
``json  
{  
  "message": "Access denied. Token missing."  
}
```

#### #### 3. Successful Access (Protected Route with Valid Token)

**\*\*Request:\*\*** `GET /api/profile` with header: `Authorization: Bearer [Token from step 1]`

**\*\*Response:\*\*** Status 200 OK.

```
``json  
{  
  "message": "Welcome to your protected profile!",  
  "data": {  
    "userId": 101,  
    "role": "admin",  
    "accessGrantedAt": "2025-01-01T10:00:00.000Z"  
  }  
}
```

#### ### Learning Outcomes

1. **\*\*JWT Life Cycle Mastery:\*\*** Students will understand the end-to-end flow of JWT authentication, from token creation (`jwt.sign()`) upon login to client-side storage and final server-side verification (`jwt.verify()`).
2. **\*\*Middleware Development:\*\*** The exercise demonstrates the critical role of **\*\*middleware\*\*** in server-side application development, specifically how to intercept requests, enforce security logic, and conditionally allow access to routes.
3. **\*\*Secure API Design:\*\*** Students will learn the principle of securing resources by enforcing token transmission via the `Authorization: Bearer` header, ensuring that sensitive data is only accessed by clients with a valid cryptographic proof of identity.