**JWT (JSON Web Token)**

JWT (JSON Web Token) is a compact, URL-safe token format that is used to securely transmit information between parties as a JSON object. It is often used in authentication and authorization systems to verify the identity of users and provide secure access to resources.

JWT is typically used for:

1. **Authentication**: After a user logs in, the server generates a JWT that contains information about the user (e.g., user ID, role). This token is sent back to the client, which stores it (usually in localStorage or cookies) and includes it in the header of subsequent API requests. The server can then verify the token to identify the user.
2. **Authorization**: The token can contain claims (information about the user) and the server can check those claims to authorize the user to access specific resources.

**Structure of JWT**

A JWT consists of three parts:

1. **Header**: Contains metadata about the token, such as the type of token (JWT) and the algorithm used to sign the token (e.g., HMAC SHA256 or RSA).
2. **Payload**: Contains the claims. Claims are statements about an entity (typically the user) and additional data. There are three types of claims:
   * **Registered claims**: Predefined claims like iat (issued at), exp (expiration), sub (subject), etc.
   * **Public claims**: Claims that can be defined by anyone, but should be collision-resistant.
   * **Private claims**: Claims used by parties that agree on them (like user\_id, role, etc.).
3. **Signature**: Ensures the integrity of the token. The signature is generated using the header, payload, and a secret key (or private key) with the specified algorithm. The signature verifies that the data has not been altered.

The structure of a JWT looks like this:

HEADER.PAYLOAD.SIGNATURE

**Example JWT**

A typical JWT looks like this:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

* The first part (before the first dot) is the **Header**.
* The second part (between the dots) is the **Payload**.
* The third part (after the last dot) is the **Signature**.

**JWT Usage Flow**

1. **User Login**:
   * The user sends their login credentials (username, password) to the server.
   * If the credentials are correct, the server generates a JWT with claims like user\_id, role, and an expiration time (exp), then sends it back to the client.
2. **Storing the JWT**:
   * The client (usually a web browser or mobile app) stores the JWT (typically in localStorage or a secure cookie).
   * The client includes the token in the Authorization header of subsequent requests, using the Bearer schema (e.g., Authorization: Bearer <token>).
3. **Server Verification**:
   * The server receives the token in the Authorization header of incoming requests.
   * It verifies the token by checking the signature using the secret key (or public key if using asymmetric algorithms).
   * If the token is valid (signature matches, and not expired), the server processes the request. Otherwise, the request is rejected.

**JWT Example in Node.js (Express)**

1. **Install Required Packages**: You’ll need jsonwebtoken for creating and verifying JWT tokens, and bcryptjs (or bcrypt) for hashing passwords.

npm install jsonwebtoken bcryptjs

1. **Creating JWT Tokens**: This example shows how to create a JWT after user login and send it back to the client.

import express from 'express';

import jwt from 'jsonwebtoken';

import bcrypt from 'bcryptjs';

const app = express();

const secretKey = 'your-secret-key'; // Store securely in environment variables

app.use(express.json());

const users = []; // Example user store, should be replaced with a database

// Register endpoint

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

    const { username, password } = req.body;

    const hashedPassword = await bcrypt.hash(password, 10);

    // Save user to the database

    users.push({ username, password: hashedPassword });

    res.status(201).send('User registered successfully');

});

// Login endpoint

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

    const { username, password } = req.body;

    // Find user in the database

    const user = users.find(user => user.username === username);

    if (!user) {

        return res.status(400).send('User not found');

    }

    // Check password

    const isMatch = await bcrypt.compare(password, user.password);

    if (!isMatch) {

        return res.status(400).send('Invalid credentials');

    }

    // Generate JWT

    const token = jwt.sign(

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

        secretKey,                    // Secret key for signing

        { expiresIn: '1h' }           // Token expiration time

    );

    // Send token to client

    res.json({ token });

});

// Protected Route

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

    const token = req.headers['authorization']?.split(' ')[1]; // Extract token from header

    if (!token) {

        return res.status(403).send('Access denied');

    }

    // Verify the token

    jwt.verify(token, secretKey, (err, decoded) => {

        if (err) {

            return res.status(403).send('Invalid token');

        }

        res.send(`Welcome ${decoded.user\_id}, you have access to this route`);

    });

});

app.listen(3000, () => console.log('Server running on port 3000'));

**Steps in the Code:**

* **Registration**:
  + User registers by sending a POST request to /register with their username and password.
  + The password is hashed using bcryptjs and stored (in this case, in the users array, but it should be in a real database).
* **Login**:
  + User sends their login credentials to /login.
  + If the credentials are correct, a JWT is generated with the user\_id as part of the payload.
  + The token is sent back to the client.
* **Protected Route**:
  + The /protected route requires the user to send the JWT in the Authorization header (in the format Bearer <token>).
  + The server verifies the token using jwt.verify(). If the token is valid, the server responds with a message indicating that the user has access.

**Verifying JWT in Middleware (for protected routes)**

You can create middleware that checks if a request contains a valid JWT:

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

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

    if (!token) {

        return res.status(403).send('Access denied');

    }

    jwt.verify(token, secretKey, (err, decoded) => {

        if (err) {

            return res.status(403).send('Invalid token');

        }

        req.user = decoded; // Attach decoded info (e.g., user\_id) to the request object

        next();

    });

};

// Apply to a protected route

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

    res.send(`Welcome ${req.user.user\_id}, you have access to this route`);

});

**Summary**

* **JWT** is widely used for **authentication** and **authorization** in web applications.
* It consists of **Header**, **Payload**, and **Signature**.
* JWT is created and verified using a **secret key** (symmetric) or **public/private keys** (asymmetric).
* Once generated, JWT is stored on the client-side (usually in localStorage or cookies) and included in the Authorization header of subsequent requests to protected resources.

By using JWT, you can create secure and scalable authentication systems for your web or mobile applications.