**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 syntax for creating a **JWT (JSON Web Token)** is straightforward, typically using the jsonwebtoken library in Node.js. Below is the general syntax for creating a JWT using

jwt.sign():

### ****JWT Creation Syntax****

jwt.sign(

payload,

secretOrPrivateKey,

{expiresIn :process.env.ACCESS\_TOKEN\_EXPIRY}

)

### ****Parameters:****

1. **payload**: The payload is the data you want to include in the token. This can include information about the user or session (such as user ID, email, role, etc.). This data will be **encoded** into the token.

Example:

{

        \_id : this.\_id,

        email : this.email,

        userName : this.userName,

        fullName : this.fullName

    }

1. **secretOrPrivateKey**: The secret key (for HMAC algorithms) or the private key (for RSA or ECDSA algorithms) used to **sign** the token. This key is known only to the server and is used to verify the integrity of the token. It should be stored securely, usually in environment variables.

Example:

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9yJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQSflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

"

1. **options** (Optional): These are configuration options, such as the token expiration time, issuer, audience, etc.
   * **expiresIn**: Sets the expiration time of the token (e.g., "1h", "30m", "2d").
   * **issuer**: The issuer of the token (usually your application name).
   * **audience**: Intended recipient of the token.
   * **algorithm**: The signing algorithm, e.g., "HS256", "RS256", etc.
2. **callback** (Optional): A callback function that is invoked with the token once it is created (can be used for asynchronous handling).

**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.

userSchema.methods.generateAccessToken = function(){

    return jwt.sign({

        \_id : this.\_id,

        email : this.email,

        userName : this.userName,

        fullName : this.fullName

    },

    process.env.ACCESS\_TOKEN\_SECRET,

    {

        expiresIn : process.env.ACCESS\_TOKEN\_EXPIRY

    }

)

}

userSchema.methods.generateRefreshToken = function(){

    return jwt.sign(

        {

            \_id : this.\_id,

            email : this.email

        },

        process.env.REFRESH\_TOKEN\_SECRET,

        {

            expiresIn : process.env.REFRESH\_TOKEN\_EXPIRY

        }

    )

}

The code you've provided defines two **custom instance methods** in Mongoose: generateAccessToken and generateRefreshToken. These methods are used to generate **JWT tokens** (JSON Web Tokens) for **authentication**. They are intended to be used for **access tokens** and **refresh tokens** in a typical token-based authentication flow.

### ****Code Explanation****

#### ****1.**** generateAccessToken ****Method****

This method generates a **JWT Access Token** for a user. The access token is used to authenticate the user and provide access to protected routes. It usually has a short expiration time (e.g., 15 minutes to 1 hour).

* **Payload**: This includes the user data (e.g., \_id, email, userName, fullName). These are the claims that will be encoded in the token. When the token is decoded later, you can access this information (without the need to query the database).
* **Secret Key**: The process.env.ACCESS\_TOKEN\_SECRET is used to sign the token. It is a secret key that only the server knows and is used to verify the authenticity of the token. The secret should be stored securely in environment variables.
* **Expiration**: The token will expire after the time set in process.env.ACCESS\_TOKEN\_EXPIRY (e.g., 1 hour). After this time, the token will no longer be valid, and the user will need to request a new one (usually using a refresh token).

#### ****2.**** generateRefreshToken ****Method****

This method generates a **JWT Refresh Token** for the user. Refresh tokens are usually long-lived (e.g., weeks or months). They are used to obtain new access tokens once they expire without requiring the user to log in again.

userSchema.methods.generateRefreshToken = function(){

    return jwt.sign(

        {

            \_id : this.\_id,

            email : this.email

        },

        process.env.REFRESH\_TOKEN\_SECRET,// secret key

        {

            expiresIn : process.env.REFRESH\_TOKEN\_EXPIRY , // options

        }

    )

}

* **Payload**: This token typically contains only the user’s \_id and email (because you generally don’t need more data in a refresh token). You don’t want to put sensitive information (like password, userName, etc.) in a refresh token since it lasts longer.
* **Secret Key**: Similar to the access token, the refresh token is signed with process.env.REFRESH\_TOKEN\_SECRET. The refresh token secret should also be stored securely.
* **Expiration**: The refresh token has a longer expiration time (process.env.REFRESH\_TOKEN\_EXPIRY) compared to the access token. This could be days, weeks, or even months. However, it's a good practice to expire refresh tokens after a longer period to limit exposure if they are compromised.

### ****Example Flow:****

1. **Login / User Authentication**:
   * When a user logs in with their credentials (e.g., username and password), the server checks if the credentials are valid. If valid, the server generates an **access token** and a **refresh token**.
   * These tokens are sent to the client. The **access token** is used to authenticate subsequent requests, while the **refresh token** is stored securely (usually in httpOnly cookies or in local storage).
2. **Access Token**:
   * The **access token** is included in the Authorization header of protected requests in the format:

Bearer <access\_token>

* + The server verifies the token with the secret key, and if valid, it allows access to the requested resource.

1. **Refresh Token**:
   * When the **access token** expires, the client sends the **refresh token** to the server to obtain a new **access token**.
   * The server verifies the refresh token, and if it's valid, it generates a new access token and sends it back to the client.
2. **Logout / Token Revocation**:
   * If the user logs out or the refresh token needs to be revoked, the server invalidates the refresh token (e.g., by deleting it from a database or marking it as invalid).

### ****Summary of Key Points:****

* **Access Tokens**: Used for short-term authentication. Typically expire after a short period (e.g., 15 minutes to 1 hour).
* **Refresh Tokens**: Used to get new access tokens after the access token expires. Typically last longer (e.g., weeks or months).
* **JWT Generation**: jwt.sign() is used to generate both access and refresh tokens, with different payloads and expiration times.
* **Security**: Access tokens are often included in the Authorization header for subsequent requests, and refresh tokens can be stored securely in cookies. Make sure to use **secure**, **httpOnly** cookies for refresh tokens to prevent XSS attacks.

By using this system, users can maintain long-term authentication without having to log in repeatedly while keeping access to protected resources secure.