Implementing Auth with JWT

JWT

JSON Web Token (JWT) is an open standard. It defines a compact, self-contained way to securely transfer information between parties as JSON objects. This information can be verified and approved because it is digitally signed. They are signed using a public/private key pair.
JWTs are commonly used for authentication and authorization purposes, and can be used in various contexts, such as APIs, single sign-on, and mobile applications.

When should you use JSON Web Tokens?

- **Authorization** : This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- **Information Exchange** : JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

A JWT typically looks like the following.
`xxxxx.yyyyy.zzzzz`
Consisting of Header, payload, signature in respective order.

Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
Ex :

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JSON is Base64Url encoded to form the first part of the JWT.

Payload

It contains the claims. Claims are statements about an entity (typically, the user) and additional data.
Ex :

```
{
  "sub": "0123456789",
  "name": "barbarik",
  "admin": true
}
```

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, & pass it to a function which will return the "*signed*" (in tech terms) value of it
Ex : here HMACSHA256() is a function to sign it

```
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

The output is three Base64-URL strings separated by dots that can be easily passed in HTML over HTTP
Ex :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Implementing JWT in NodeJS

1. You will need to install the `jsonwebtoken` package to use JWT in your Node.js application. You can do this by running `npm i jsonwebtoken`
2. Create a secret key : A secret key is used to sign and verify JWTs. You should generate a unique secret key for your application and keep it secure. You can generate a secret key using a package like crypto:
   const crypto = require('crypto');
   const secretKey = crypto.randomBytes(32).toString('hex');

3. Create a JWT for authentication
   To create a JWT for authentication, you will need to use the sign() method provided by the `jsonwebtoken` package. Ex :

   ```
   const jwt = require('jsonwebtoken');
   const user = {
     id: 123,
     name: 'Siri Alexa',
     email: 'siriAlexa@GoogleAssistant.com'
   };
   const token = jwt.sign(user, secretKey, { expiresIn: '1h' });
   ```

   In this example, we're signing the user object with the secretKey and setting the expiration time for the token to 1 hour.

4. Verify the JWT
   To verify a JWT, you will need to use the verify() method.Ex :

   ```
   const token =
   'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzLCJuYW1lIjoiSm9obiBEb2UiLCJlbWFpbCI6ImpvaG4uZG9lQGV4YW1wbGUuY29tIiwiaWF0IjoxNTE2MjM5MDIyLCJleHAiOjE1MTYyMzkxMjJ9.Bs2sV4jX4xR7VupPvzjIHm6-EDeUP8txHlU1v6U1c6s';

   jwt.verify(token, secretKey, (err, decoded) => {
     if (err) {
       console.error(err);
       return;
     }

     console.log(decoded);
   });
   ```

   In this example, we're verifying the token string with the secretKey and logging the decoded payload to the console.

Here's the complete example of using JWT for authentication in a Node.js API:

```
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();

const secretKey = 'mysecretkey';

app.post('/login', (req, res) => {
  // Check if the user's credentials are valid
  const user = {
    id: 123,
    name: 'Siri Alexa',
    email: 'siriAlexa@GoogleAssistant.com'
  };

  // Create a JWT for the user
  const token = jwt.sign(user, secretKey, { expiresIn: '1h' });

  res.json({ token });
});

app.get('/protected', (req, res) => {
  // Get the JWT from the authorization header
  const authHeader = req.headers.authorization;
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  // Verify the JWT
  jwt.verify(token, secretKey, (err, decoded) => {
    if (err) {
      return res.status(401).json({ message: 'Unauthorized' });
    }

    // If the JWT is valid, proceed with the protected request
    res.json({ message: 'Protected resource' });
  });
});

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In this example, we're creating a simple API with two routes: `/login` and `/protected` . When a user logs in, we create a JWT for them and return it as a JSON response. When a user tries to access a protected resource, we check if they have a valid JWT in the authorization header, and if so, we verify it and allow them to access the resource. If the JWT is invalid or missing, we return a 401 Unauthorized response.