JWTs

JWT stands for JSON Web Token. Similar to cookies, you can use JWTs to store a certain amount of information. Unlike cookies, which can be easily read by users, JWTs are digitally signed with a secret key or encrypted to hide the data inside.

When to use

- Authorization: The most common scenario to use JWTs. 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.

Structure

JWTs consist of three parts

- Header
- Payload
- Signature

Header: Typically consists of two parts, the type of token and the signing algorithm used (like HMAC SHA256 or RSA). A header might look like this:

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

Payload: Contains the claims. Claims are statements about an entity (like the user) and additional data. There are three types of claims:

- Registered claims: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

- Public claims: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

- Private claims: These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

A payload example would be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

For the signature, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
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.

How they work

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

You also should not store sensitive session data in browser storage due to lack of security.

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

This can be, in certain cases, a stateless authorization mechanism. The server's routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case.

Note that if you send JWTs through HTTP headers, you should try to prevent them from getting too big. Some servers don't accept more than 8 KB in headers. If you are trying to embed too much information in a JWT token, like by including all the user's permissions, you'll need an alternative solution.

If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

Coding Example

server.js

```
import express from 'express';
import jwt from 'jsonwebtoken';
const app = express();
app.use(express.json());

const users = { // Example users database
    admin: { password: "password123" }
};

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    if (users[username] && users[username].password === password) { // if their login credentials are correct
        const token = jwt.sign(
            { username: username },
            'secret_key', // Never expose your secret key in client-side code
            { expiresIn: '24h' } // Expires in 24 hours
        );
        res.json({ token });
    } else {
        res.status(401).send('Credentials are incorrect');
    }
});

app.listen(3000, () => console.log('Server running on http://localhost:3000'));
```

Then, in the LoginPage.js

```
import React, { useState } from 'react';

function Login() {
    function handleLogin() {
        const data = {
            username: username,
            password: password
        }
        try {
            fetch('http://localhost:3000/login', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(data)
            });
            .then(response => response.json())
            .then(response => {
                if (response.status === 200) {
                    localStorage.setItem('token', data.token); // Stores the token
in localStorage
                    setIsLoggedIn(true);
                } else {
                    alert('Login Failed');
                }
            })
        } catch (error) {
            console.error('Login Error:', error);
        }
    };
}

export default Login;
```

Notes:

- The JWT is stored in localStorage. This is practical for demo purposes, but for enhanced security, especially against XSS attacks, consider using HttpOnly cookies or other secure methods to store tokens.
- Never include sensitive logic like key generation or user verification directly in React. This should always be handled by a secure server.
- The stored JWT can be used to make authenticated requests to other server endpoints. Typically, it's included in the HTTP Authorization header as Bearer {token}.