Anil Kumar Follow
Sep 18, 2017 · 14 min read · Listen

JSON Web Tokens vs Oauth 2.0

Table of Contents

Brief introduction to JWT

- A JWT technically is a mechanism to verify the owner of some JSON data.
- It's an encoded string, which is URL safe, that can contain an unlimited amount of data (unlike a cookie), and it's cryptographically signed.
- When a server receives a JWT, it can guarantee the data it contains can be trusted because it's signed by the source. No middleman can modify a JWT once it's sent.
- It's important to note that a JWT guarantees data ownership but not encryption; the JSON data you store into a JWT can be seen by anyone that intercepts the token, as it's just serialized, not encrypted.

For this reason, it's highly recommended to use HTTPS with JWTs (and HTTPS in general, by the way).

We're not going to cover how JWTs are generated in detail. There are various guides about this, and I suggest this one: "What the heck is JWT anyway?"

TL;DR: What are they good for?

JWT is a great technology for **API authentication** and server-to-server authorization.

It's not a good choice for sessions.

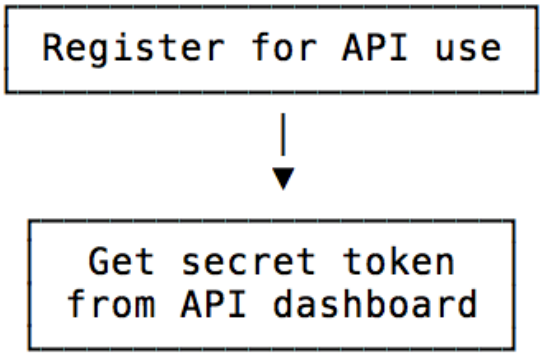
Read on to understand the nitty gritty details about those affirmations.

Using JWT for API authentication

A very common use of a JWT token, and the one you should probably *only* use JWT for, is as an **API authentication mechanism**.

Just to give you an idea, it's so popular and widely used that Google uses it to let you authenticate to their APIs.

The idea is simple: you get a secret token from the service when you set up the API.



On the client side, you create the token (there are many libraries for this), using the secret token to sign it.

You pass it as part of the API request, and the server will know it's that specific client because the request is signed with its unique identifier:

JWT (JSON Web Token)

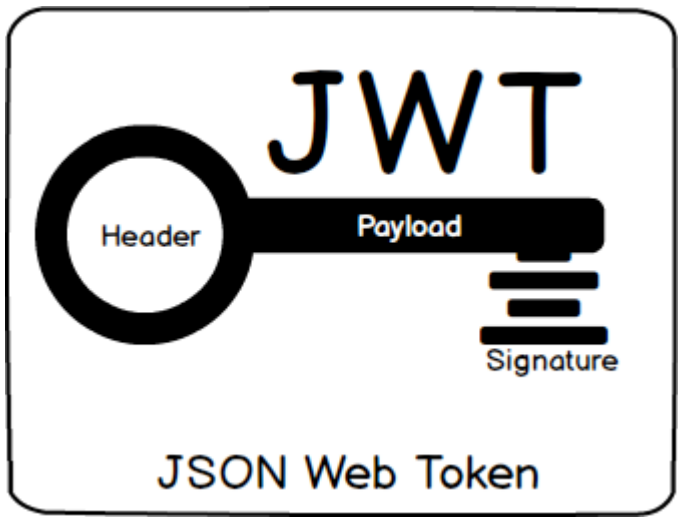
As you may already know JWT is a popular JSON based format of a security token. So, lets start from the formal definition of JSON web tokens;

> JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties.

Simply put, a JWT is a JSON based format of a **security token** which is basically a **base64 url-**encoded string which is used as a means of transferring secure content between two applications. They are used to secure request data in Web APIs. These are included in **Authorization** HTTP headers as part of the bearer authentication scheme.

> A JWT token is composed of a header, a payload, and a signature and has the format **header.payload.signature.**

period (‘.’) characters. When signed, the three parts of the JWT are the three parts of a JWS (JSON Web Signature) used to represent the JWT.



When encrypted, the three parts of the JWT are the three parts of a JWE (JSON Web Encryption) used to represent the JWT. JWT Header specifies the cryptographic operations performed on the JWT claims. If the JWT Header is a JWS Header, the claims are signed and if the JWT Header is a JWE Header, the claims are encrypted.

sample JWT token includes the following elements.

Header : Algorithm and token type

```
{
 “alg”: “HS256”,
 “typ”: “JWT”
}
```

Payload : data

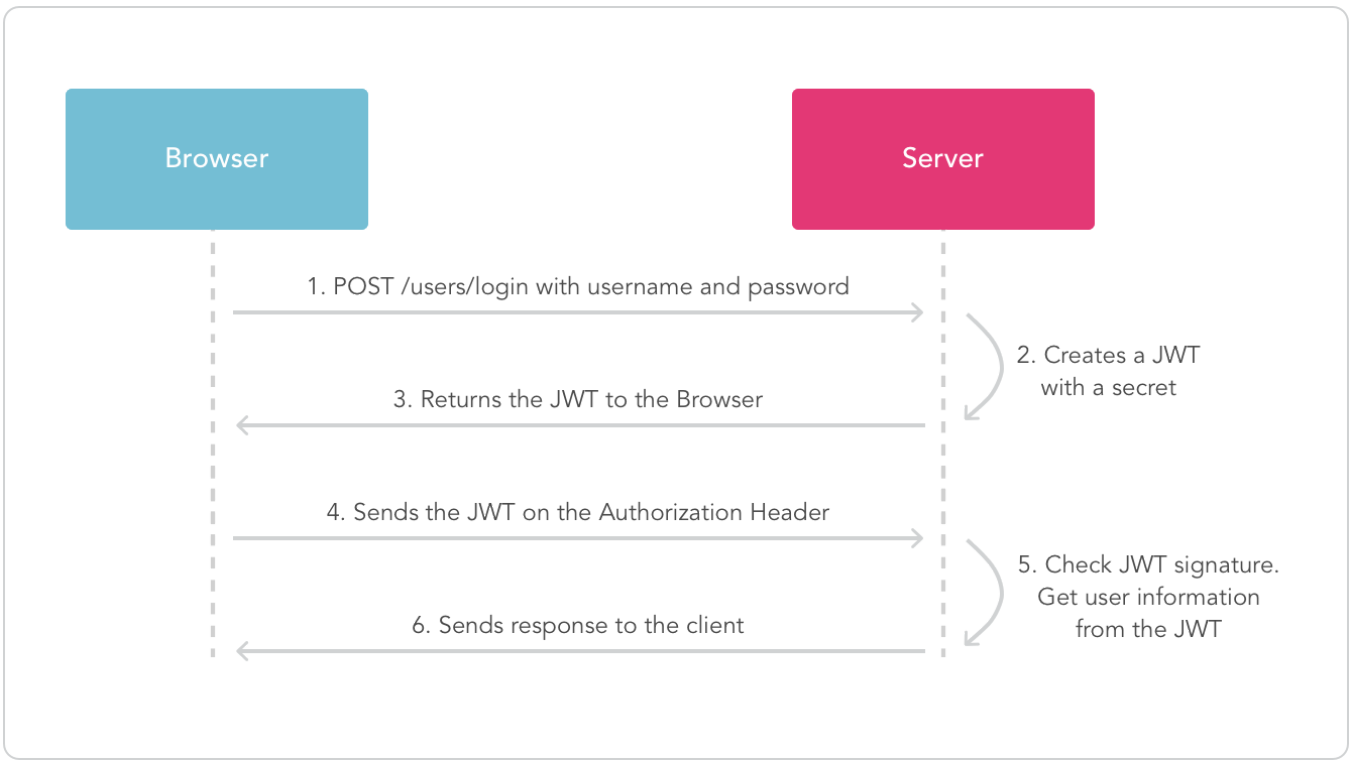
```
{
  “sub”: “1234567890”,
 “name”: “John Doe”,
 “admin”: true
}
```

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

Final output:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
     .
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
     .
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Using **JWT.IO** we can easily decode, verify and generate JWT tokens.



Simple procedure on JWT token generation and validation at the same server

As depicted in the above diagrams after obtaining a JWT token from the server, the clients (eg: browser) can include those tokens in bearer authorization scheme in the authorization header. (Authorization: Bearer <token>). Bearer schemes are an

used since token sent in the request can be stolen by third parties and they can use it to access to API until the token expires.

And also to ensure integrity, information contained in the token should be signed by a private key which is owned by the server. When the server gets the token back from the client, it just has to compare the signature sent by the client with the one it will generate with its private key. If the signatures are identical, the token is valid and the JWT claims(user information) contained in the token can be evaluated.

Following are some of the important claims that should be included in a JWT shared between two parties.

iss (Issuer) Claim
Identifies the principal that issued this JWT

sub (Subject) Claim
Identifies the principal that is the subject of the JWT. The claims in a JWT are usually statements about this subject. The subject value must be either scoped to be locally unique in the context of the issue or it should be globally unique.

aud (Audience) Claim
Identifies the recipients that the JWT is intended for. Each principal intended to process the JWT must identify itself as a value in the audience claim of the token. If its not identified as a value in the case where this aud claim is present in the token then the JWT must be rejected by the principal.

exp (Expiration Time) Claim
Identifies the expiration time on or after which the JWT must not be accepted for processing.

Lastly in this post I'm going to show some of the pros and cons of JWT tokens as per my understanding.

Pros

- JWT tokens are stateless in the information is not stored server-


ensures trust and security between applications and its users.

- JWT tokens can be used to authenticate users on multiple applications. For this, applications will be required to share the same private key to sign and verify the tokens. In addition, each application's token validation endpoints should be specified under the audience claims in order to gain access to this shared token. As a result, users can authenticate one time on the authorization server and if authorization is granted he can seamlessly use other applications that use the same private key to verify JWT tokens.

Cons

- Since we specify a token expiry, in the case where a user account needs to be removed then the application has to wait for the token to expire for the user account deactivation to be effective.
- Another major issue is that JWT tokens require re-authentication at the time of token expiration since no refresh tokens are specified in their implementation.
- In the case of account hijacking a user may need to change their password ASAP. But with JWT if an authentication is done beforehand, then a token generated for previous password would be still valid even after user password is changed until the expiry time.
- It is not possible to forcefully destroy a token because even if the token is deleted from the browser, it is still going to be valid until the expiry time. Hence user log out is not enforced until the token is expired.

One more caution when using JWTs is, they might become a disadvantage if you store a lot of data inside it. Since it is sent back and forth with every request, it can slow down the communication, whereas with opaque tokens you are just sending a reference and looking up the data on the server side, which makes the transfer over the network faster.

Store JWTs securely

A JWT needs to be stored in a safe place inside the user's browser.

Don't store it in local storage (or session storage). If any of the third-party scripts you include in your page gets compromised, it can access all your users' tokens.

The JWT needs to be stored inside an **httpOnly cookie**, a special kind of cookie that's only sent in HTTP requests to the server, and it's never accessible (both for reading or writing) from JavaScript running in the browser.

Using JWT to securely exchange information between two servers

Since JWT are signed, the receiver can be sure the client is really who it thinks it is.

Authentication vs Authorization

Prior to introducing what OAuth2 is first, let me give you a brief idea on the concepts of authentication and authorization.

Authentication is the process of verifying the identity of a user by obtaining some sort of credentials for example his username password combination, and using those credentials to verify the user's identity.

Authorization is the process of allowing an authenticated user to access his resources by checking whether the user has access rights to the system. You can control access rights by granting or denying specific permissions to an authenticated user. So, If the authentication was successful, the authorization process starts. Authentication process always proceeds to Authorization process.

What is OAuth

To begin at a high level, OAuth is *not* an API or a service: it's an **open standard** for **authorization** and anyone can implement it.

More specifically, OAuth is a standard that apps can use to provide client applications with "*secure delegated access*". OAuth works over HTTPS and authorizes devices, APIs, servers, and applications with access tokens rather than credentials.

Simply put: it's a standard to securely access stuff with randomized tokens.

The specification describes **five grants** for *acquiring an access token*:

- **Authorization code grant**

This flow redirects you to log in directly with a 3rd party, meaning the client never gets access to your username/password that you type in. That very important secret is not shared in another database somewhere, it remains between you and the credential provider you trust (such as Facebook, although not sure I would trust them too much).

That 3rd party provider that you login with generates your JWT that the client actually uses to fetch data for you. Based upon the configuration, in most cases, it's a short-lived Access Token (Access Token is a JWT) meaning the client only can act on your behalf for a certain time period.

- **Implicit grant**
- **Resource owner credentials grant**
- **Client credentials grant**
- **Refresh token grant**

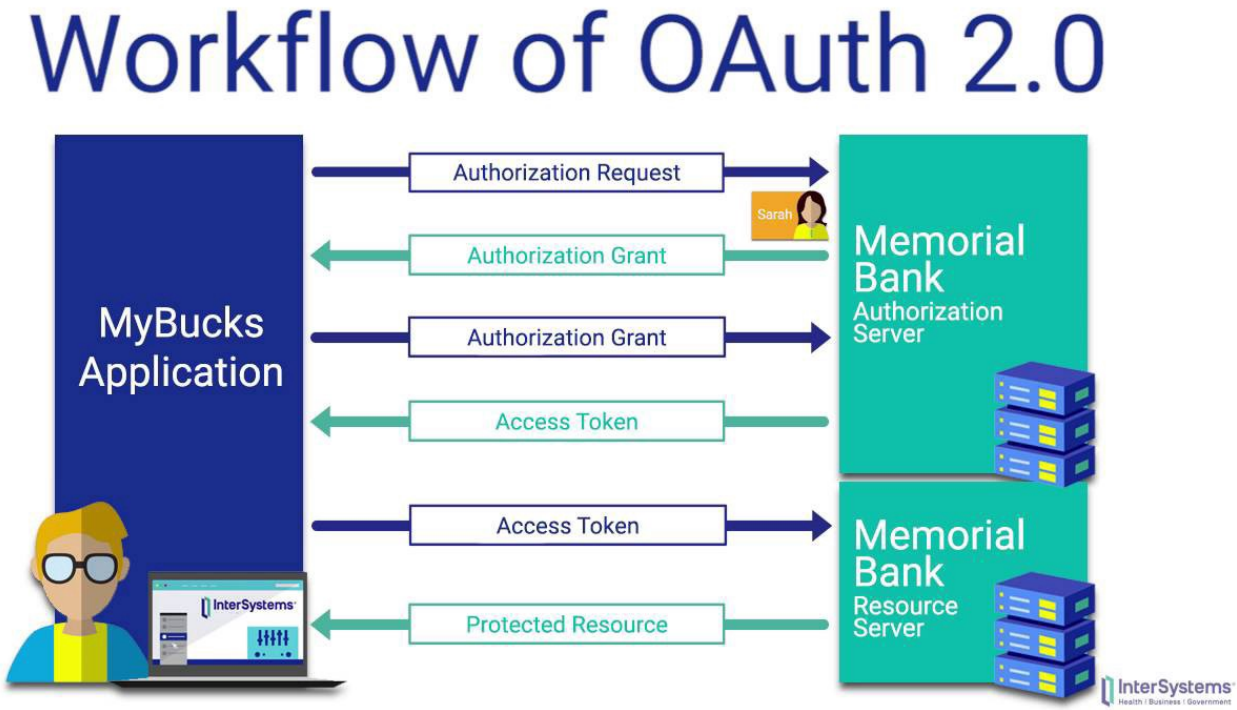
OAuth 2.0

This is the version 2 of the OAuth protocol. It can be referred to as a authorization framework as well. Version 2 simplifies the previous version of the protocol and facilitates interoperability between different applications. Even Google and popular social websites like Facebook and Twitter also uses OAuth2 protocol for authentications and authorizations.

- So, **OAuth** as it name suggests is simply a standard for **Authorization**.
- With OAuth, you can **log into third party websites** with your Google, Facebook, Twitter or Microsoft accounts without having the necessity to provide your passwords. This way you can avoid creating accounts and remembering passwords on each and every web application that you use on the Internet.
- OAuth is based on an **access token concept**. When you authenticate yourself using


access your data hosted in the resource server. In the case of Google, your Gmail inbox, contacts, photos etc. are the resources. So, any third party application can access those resources, for an example view your Gmail inbox using OAuth. Hence, OAuth is a simple way to publish and interact with protected resource data. It's also a safer and more secure way for people to give you access to their resource data.

- **OAuth2 uses HTTPS** for communication between the client and the authorization server because of confidential data for example client credentials. passing between the two applications.



Workflow of OAuth 2.0

Tokens

When the client application is **authorized** by the resource owner, the authorization server issues an **access token**. The client application can use that token to access resource server APIs. For an example a third party application can request an access token from Google server to use Google contacts API. These tokens are random string generated by the authorization server.

There are 2 types of token:

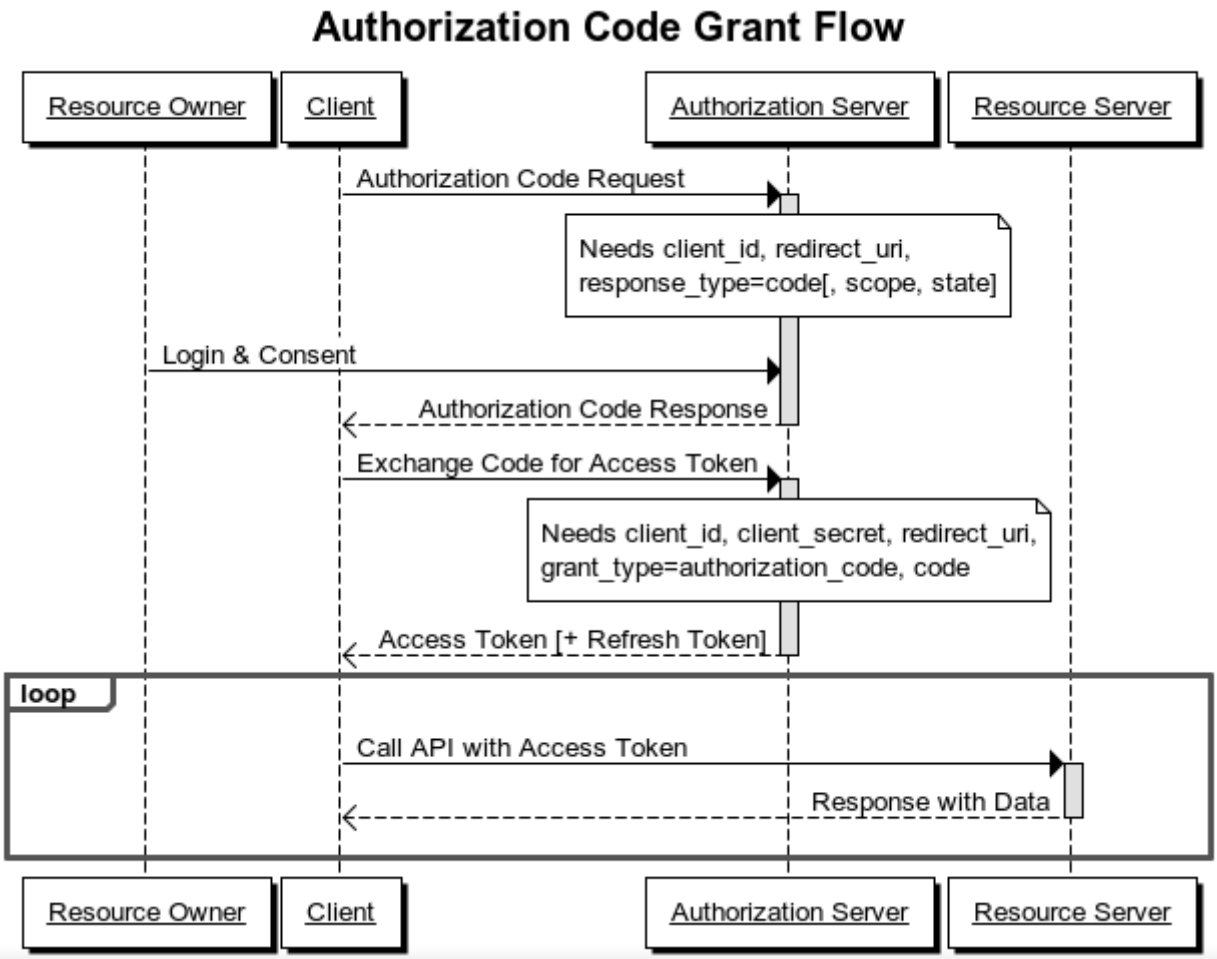
confidential. The authorization server can define the access token scope. This is a parameter which can be used to limit the access rights for the third party application. The client application should provide the scopes it requires with the request to the authorization server. If the scope is minimum there is a greater chance of being authorized by the user.

Example of sending an access token to the resource server using HTTP GET:

https://example.com/profile**?access_token=MzJmNDc3M2VjMmQzN**

- **Refresh Token**: this token is issued with the access token but unlike the latter, it is NOT sent in each request from the client to the resource server. The client application can simply use a refresh token to renew access token when it expires.

The below diagram depicts the basic sequence flow for an Oauth2 authorization process.



will be required to login and consent at the server. After successful authentication, user has to authorize the client at the authorization server. The server responds to the request with an authorization code. The client can then exchange the authorization code with an access token by using client secret. In return the authorization server will issue an access token and a refresh token in addition to that.
With the acquired access token, the client can send requests to the resource server and access resource server API and user data presenting the access token as the means of authentication.

- Oauth
- Oauth2

So the real difference is that JWT is just a token format, OAuth 2.0 is a protocol (that *may* use a JWT as a token format or access token which is a bearer token.).

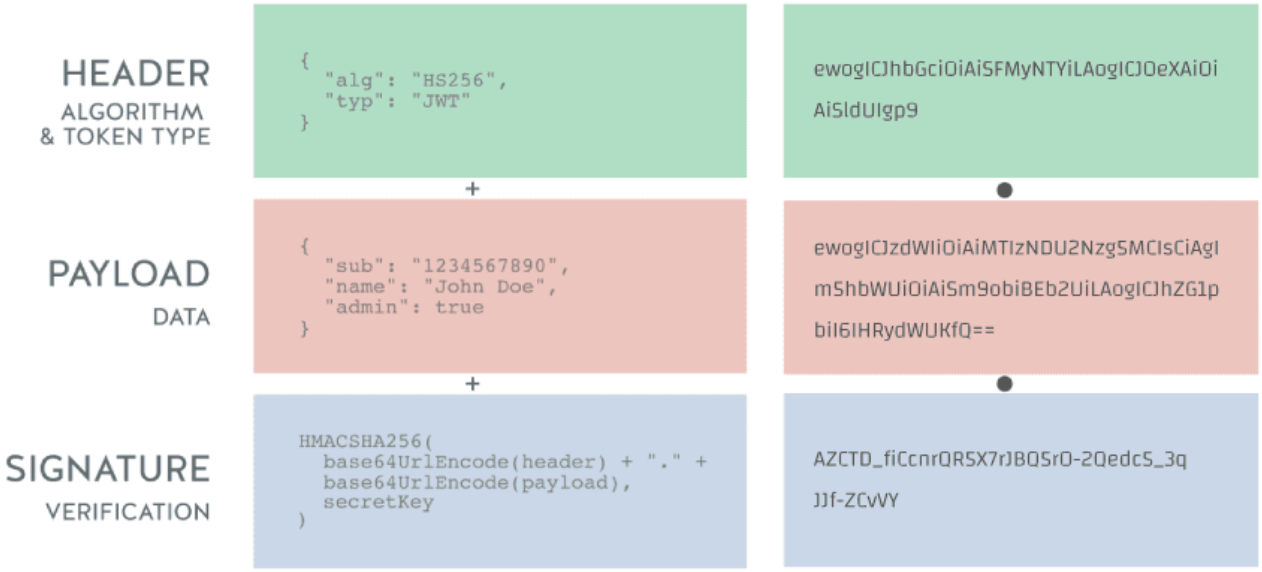
OpenID connect mostly use JWT as a token format.

TL;DR If you have very simple scenarios, like a single client application, a single API then it might not pay off to go OAuth 2.0, on the other hand, lots of different clients (browser-based, native mobile, server-side, etc) then sticking to OAuth 2.0 rules might make it more manageable than trying to rolling your own system.

One final piece of advice, even if you don't need to go full OAuth 2.0, I **would strongly recommend on passing your access token within the `Authorization` header instead of going with custom headers**. If they are really bearer tokens follow the rules of RFC 6750, if not, you can always create a custom authentication scheme and still use that header.

> Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus, the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general, and especially Authorization headers.

XSRF token is always sent to the client in every response header. It does not matter if a CSRF token is sent in a JWT token or not, because the CSRF token is secured with itself.

Difference between JWT and OAuth

Firstly, we have to differentiate JWT and OAuth.

Basically, JWT is a token format.

OAuth is an standardised **authorization** protocol that can use JWT as a token.

OAuth uses server-side and client-side storage. If you want to do real logout you must go with OAuth2. Authentication with JWT token can not logout actually. Because you don’t have an Authentication Server that keeps track of tokens.

If you want to provide an API to 3rd party clients, you must use OAuth2 also. OAuth2 is very flexible. JWT implementation is very easy and does not take long to implement. If your application needs this sort of flexibility, you should go with OAuth2. But if you don’t need this use-case scenario, implementing OAuth2 is a waste of time.

JWT (JSON Web Tokens)- It is just a token format. JWT tokens are JSON encoded data structures contains information about issuer, subject (claims), expiration time etc. It is signed for tamper proof and authenticity and it can be encrypted to protect the token information using symmetric or asymmetric approach. JWT is simpler than SAML 1.1/2.0 and supported by all devices and it is more powerful than SWT(Simple Web Token).

just for authorization, client software can be authorized to access the resources on-behalf of end user using access token.

- Security protocols like **OAuth2** use JWT tokens to secure APIs.
- At this point, most, but not all, Identity Provider vendors are using JWT tokens as OAuth2 Access Tokens.

OpenID Connect —

> "OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner."

As mention in the definition, **OpenID** Connect builds on top of OAuth2 and add authentication. OpenID Connect add some constraint to OAuth2 like UserInfo Endpoint, ID Token, discovery and dynamic registration of OpenID Connect providers and session management. JWT is the mandatory format for the token.

- **OpenID Connect** uses JWT tokens to authenticate web applications, but stores the token in a cookie. But they are different from the session cookies which is sent to the user in the request automatically whenever a user logs in.

CSRF protection — You don't need implement the CSRF protection if you do not store token in the browser's cookie.

3 Ways to Secure Your Web API for Different Situations

3 Ways to Secure Your Web API for Different Situations

Security is a important part in any software development and APIs are no exception. Even for a public API, having...

medium.com

- JWT can be seen not but modifiable once it's sent.
- JWT is just serialised, not encrypted.
- OAuth is *not* an API or a service: it's an **open standard** for **authorization** .
- OAuth is a standard set of steps for obtaining a token. There are 5 different flow patterns.
- Authorization code grant is the most secure OAuth grant type
- Resource Owner grant type is the least secure

References:

https://medium.com/@technospace/understanding-json-web-tokens-jwt-a9064621f2ca

An Introduction to OAuth 2.0

Prior to introducing what OAuth2 is first let me give you a brief idea on the concepts of authentication and...

medium.com

https://stackoverflow.com/questions/39909419/what-are-the-main-differences-between-jwt-and-oauth-authentication#:~:text=OAuth%202.0%20defines%20a%20protocol,JWT%20defines%20a%20token%20format.&text=So%20the%20real%20difference%20is,JWT%20as%20a%20token%20format).