

# Session, Jwt, Oauth2 에 대한 정리

## What are the main differences between JWT and OAuth authentication?

Asked 4 years, 4 months ago   Active 6 months ago   Viewed 314k times

 I have a new SPA with a stateless authentication model using JWT. I am often asked to refer OAuth for authentication flows like asking me to send 'Bearer tokens' for every request  
**423** instead of a simple token header but I do think that OAuth is a lot more complex than a simple JWT based authentication. What are the main differences, should I make the JWT authentication behave like OAuth?

 I am also using the JWT as my XSRF-TOKEN to prevent XSRF but I am being asked to keep them separate? Should I keep them separate? Any help here will be appreciated and might lead to a set of guidelines for the community.  


[authentication](#) [oauth](#) [oauth-2.0](#) [jwt](#)

Share [Improve this question](#)

edited Jul 14 '20 at 18:56

Follow



Abhishek Bhagate

5,063 ● 3 ● 9 ● 27

asked Oct 7 '16 at 4:30



Venkatesh Laguduva

10.7k ● 5 ● 26 ● 32

## 8 Answers

[Active](#) [Oldest](#) [Votes](#)

 **395** 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 roll your own system.

 As stated in another answer, JWT ([Learn JSON Web Tokens](#)) is just a token format, it defines a compact and self-contained mechanism for transmitting data between parties in a way that can be verified and trusted because it is digitally signed. Additionally, the encoding rules of a JWT also make these tokens very easy to use within the context of HTTP.

Being self-contained (the actual token contains information about a given subject) they are also a good choice for implementing stateless authentication mechanisms (aka *Look mum, no sessions!*). When going this route and the only thing a party must present to be granted access to a protected resource is the token itself, the token in question can be called a bearer token.

In practice, what you're doing can already be classified as based on bearer tokens. However, do consider that you're not using bearer tokens as specified by the OAuth 2.0 related specs (see [RFC 6750](#)). That would imply, relying on the `Authorization` HTTP header and using the `Bearer` authentication scheme.

Regarding the use of the `JWT` to prevent CSRF without knowing exact details it's difficult to ascertain the validity of that practice, but to be honest it does not seem correct and/or worthwhile. The following article ([Cookies vs Tokens: The Definitive Guide](#)) may be a useful read on this subject, particularly the `XSS and XSRF Protection` section.

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.

(source: [RFC 6819, section 5.4.1](#))

Share

edited Jul 14 '20 at 18:55

answered Oct 7 '16 at 9:33

Improve this answer



Abhishek Bhagat

5,063 ● 3 ● 9 ● 27

João Angelo

51.1k ● 12 ● 126 ● 138

Follow

- 4 Does this mean if I use JWT authentication on a mobile app, I don't need to include CSRF on its POST request? Unlike web interface with forms? – [user805981](#) Oct 12 '17 at 22:55
- 4 Cookies vs Tokens: The Definitive Guide , i.e [auth0.com/blog/cookies-vs-tokens-definitive-guide](#) isn't working This is another great discussion post : [stackoverflow.com/questions/37582444/...](#) – [Siddharth Jain](#) Jun 22 '18 at 3:41
- 2 " they are also a good choice for implementing stateless authentication mechanisms (aka Look mum, no sessions!). " If you need a way to invalidate the token because let's say it was leaked or intercepted or the user simply logged out and removing the token is not secure enough because the token is still valid then you need to store them in some database, so I think there must be some notion of session on the server for security purposes or simple token blacklist. You might say use "refresh" tokens for this. But refresh tokens can be intercepted too and consequences are much worse. – [Konrad](#) Sep 5 '18 at 7:27
- 2 @Konrad, I did implement a similar mechanism which stored the unused valid tokens in a table, release them from there when they expire. For each incoming request, I have written code to cross check the incoming token against the "unused valid tokens". Even though it works, I always had my doubts - there should be a better way to handle unused but still valid tokens. – [Venkatesh Laguduva](#) Sep 11 '18 at 2:22

doubts - there should be a better way to handle unused but still valid tokens. –

Venkatesh Laguduva | Sep 11 '18 at 2:22 

- 3 On the other hand refresh tokens just complicate implementation of the client. Because if your access token expires you need to handle that, user will be pissed if you will just log him out without any possibility of even manual refresh of the session(like banks do). It's slightly more work to do, also using standard ways of authentication (OID) and authorization(OAuth) can very often be an overkill. – Konrad Sep 11 '18 at 7:19



OAuth 2.0 defines a protocol, i.e. specifies how tokens are transferred, JWT defines a token format.

342



OAuth 2.0 and "JWT authentication" have similar appearance when it comes to the (2nd) stage where the Client presents the token to the Resource Server: the token is passed in a header.



But "JWT authentication" is not a standard and does not specify *how* the Client obtains the token in the first place (the 1st stage). That is where the perceived complexity of OAuth comes from: it also defines various ways in which the Client can *obtain* an access token from something that is called an Authorization Server.

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).

Share Improve this answer Follow

edited Aug 5 '19 at 9:21



piet.t

10.9k ● 20 ● 39 ● 49

answered Oct 7 '16 at 7:12



Hans Z.

40.1k ● 9 ● 79 ● 104

- 14 Do oAuth protocol implementations use JWT as the token format, for most cases? If not what is most common? – James Wierzba Nov 10 '17 at 23:30 

- 21 The token format in oauth is undefined, but JWT should work fine – vikingsteve Nov 21 '17 at 14:34



Firstly, we have to differentiate JWT and OAuth. Basically, JWT is a token format. OAuth is an authorization protocol that can use JWT as a token.

157 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.



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.

Therefore sending CSRF token in JWT is unnecessary.

Share Improve this answer Follow

edited Nov 1 '18 at 18:07

answered Oct 7 '16 at 6:05



Melikşah Şimşek

1,933 ● 1 ● 6 ● 11

12 I don't understand why this answer has a lot of upvotes, it states that "OAuth is an authentication framework" and this is completely wrong. OAuth is an authorization protocol and only an authorization protocol. – Michael Oct 30 '18 at 16:14

5 Hi @Michael there is too much misunderstanding about this. I edited my comment thank you. – Melikşah Şimşek Nov 1 '18 at 18:08

2 Are you guys saying that Oauth is just a piece of Standards that developers should follow? Or it really is a framework? – StormTrooper Apr 27 '20 at 12:10

*"If you want to do real logout you must go with OAuth2"* -- Not true. For example, Devise-JWT is a non-OAuth solution that provides "log out" functionality by invalidating tokens: [github.com/waiting-for-dev/devise-jwt#revocation-strategies](https://github.com/waiting-for-dev/devise-jwt#revocation-strategies) – GoBusto Sep 2 '20 at 11:49

2 @Michael, that's not entirely correct. The RFC is titled "The OAuth 2.0 Authorization Framework" so I guess that leaves some confusion as well ;) [tools.ietf.org/html/rfc6749](https://tools.ietf.org/html/rfc6749). I get what you mean though. – hfossli Sep 10 '20 at 7:05

**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).

86

🕒

**OAuth2** - OAuth2 solve a problem that user wants to access the data using client software like browse based web apps, native mobile apps or desktop apps. OAuth2 is just for authorization, client software can be authorized to access the resources on-behalf of end user using access token. *Authorization only*

**OpenID Connect** - 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. *+ Authentication*

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

Share Improve this answer Follow

edited Aug 29 '20 at 2:43

answered Oct 8 '17 at 0:33



Machavity ♦



ManishSingh



28.3k ● 25 ● 73 ● 90



1,259 ● 9 ● 10

- 7 No cookies == No CSRF protection. If you don't use cookies for authorization, then you don't have to worry about CSRF protection. – [niranjan harpale](#) Oct 18 '19 at 11:17

It looks like everybody who answered here missed the moot point of OAUTH

## 66 From Wikipedia

OAuth is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords.<sup>[1]</sup> This mechanism is used by companies such as Google, Facebook, Microsoft and Twitter to permit the users to share information about their accounts with third party applications or websites.

The key point here is access delegation. Why would anyone create OAUTH when there is an id/pwd based authentication, backed by multifactored auth like OTPs and further can be secured by JWTs which are used to secure the access to the paths (like scopes in OAUTH) and set the expiry of the access

There's no point of using OAUTH if consumers access their resources(your end points) only through their trusted websites(or apps) which are your again hosted on your end points

You can go OAUTH authentication only if you are an OAuth provider in the cases where the resource owners (users) want to access their(your) resources (end-points) via a third-party client(external app). And it is exactly created for the same purpose though you can abuse it in general

### Another important note:

You're freely using the word authentication for JWT and OAUTH but neither provide the authentication mechanism. Yes one is a token mechanism and the other is protocol but once authenticated they are only used for authorization (access management). You've to back OAUTH either with OPENID type authentication or your own client credentials

Share Improve this answer Follow

answered Jul 16 '17 at 16:19



manikawnth

1,719 ● 14 ● 29

- 6 OAuth can also be used for your own clients, not necessarily just 3rd party ones. The Password Credentials Grant type does exactly that. – [harpratap](#) Jan 12 '18 at 2:57

- 4 I was looking for google for such a concrete answer but could not find one. Everyone was just talking about definitions e.g. token vs protocol. Your answer explained the true purpose of using one above the other. Thank you so much! – [Vivek Goel](#) Jan 7 '20 at 18:39



# Introduction to JSON Web Tokens

**NEW:** get the JWT Handbook for free (<https://auth0.com/resources/ebooks/jwt-handbook>) and learn JWTs in depth!

## What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519 (<https://tools.ietf.org/html/rfc7519>)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on *signed* tokens. Signed tokens can verify the *integrity* of the claims contained within it, while encrypted tokens *hide* those claims from other parties. When tokens

are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

## When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **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.

## What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots ( . ), which are:

- Header
  - Payload
  - Signature
- 

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

Let's break down the different parts.

## 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.

For example:

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

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

## Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

- **Registered claims** (<https://tools.ietf.org/html/rfc7519#section-4.1>): 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 (<https://tools.ietf.org/html/rfc7519#section-4.1>).

Notice that the claim names are only three characters long as JWT is meant to be compact.

- **Public claims** (<https://tools.ietf.org/html/rfc7519#section-4.2>): 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 (<https://www.iana.org/assignments/jwt/jwt.xhtml>) or be defined as a URI that contains a collision resistant namespace.
- **Private claims** (<https://tools.ietf.org/html/rfc7519#section-4.3>): These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

An example payload could be:

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

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

## **Signature**

To create the signature part 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.

## **Putting all together**

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

If you want to play with JWT and put these concepts into practice, you can use jwt.io Debugger (<https://jwt.io/#debugger-io>) to decode, verify, and generate JWTs.

The screenshot shows the jwt.io debugger interface. At the top, the algorithm is set to HS256. The token is displayed in two columns: 'Encoded' on the left and 'Decoded' on the right.

**Encoded:**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6  
IkpxVCJ9.eyJzdWIiOiIxMjM0NTY  
30DkwIiwibmFtZSI6Ikpvag4gRG9  
lIiwiYWRtaW4iOnRydWV9.TJVA95  
OrM7E2cBab30RMHrHDcEfjoYZge  
FONFh7HgQ
```

**Decoded:**

**HEADER:**

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

**PAYOUT:**

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

**VERIFY SIGNATURE**

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret  
)  secret base64 encoded
```

**Signature Verified**

## How do JSON Web Tokens 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

([https://cheatsheetseries.owasp.org/cheatsheets/HTML5\\_Security\\_Cheat\\_Sheet.html#storage](https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html#storage)).

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:

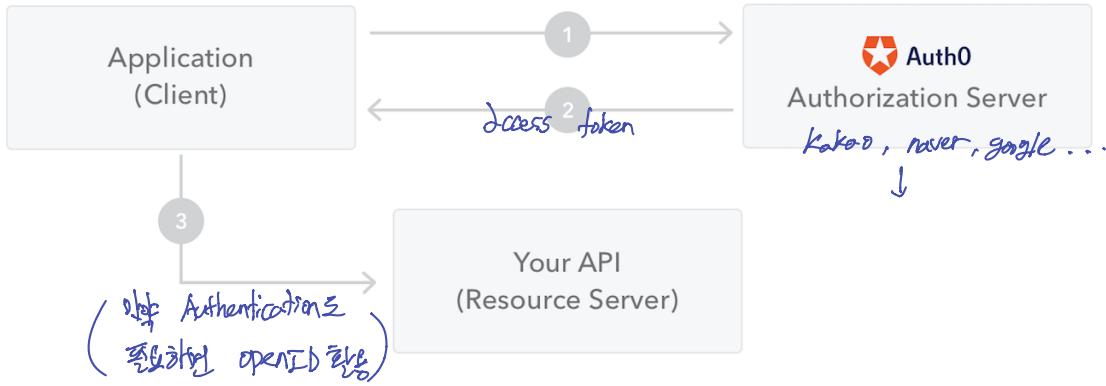
```
Authorization: Bearer <token>
```

This can be, in certain cases, a stateless authorization mechanism. The server's protected 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.

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.

The following diagram shows how a JWT is obtained and used to access APIs or resources:

Authorization header ے تو  
token یا تو CORS ہفت گت. (cookie ے ممکن نہیں)



1. The application or client requests authorization to the **authorization server**. This is performed through one of the different authorization flows. For example, a typical OpenID Connect (<http://openid.net/connect/>) compliant web application will go through the /oauth/authorize endpoint using the authorization code flow ([http://openid.net/specs/openid-connect-core-1\\_0.html#CodeFlowAuth](http://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth)).
2. When the **authorization is granted**, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Do note that with signed tokens, all the information contained within the token is exposed to users or other parties, even though they are unable to change it. This means you should not put secret information within the token.

## Why should we use JSON Web Tokens?

①  $\rightarrow$  Secure

② json  $\rightarrow$  size compact

Let's talk about the benefits of **JSON Web Tokens (JWT)** when compared to **Simple Web Tokens (SWT)** and **Security Assertion Markup Language Tokens (SAML)**.

As **JSON** is less verbose than XML, when it is encoded its size is also smaller, making JWT more **compact** than SAML. This makes JWT a good choice to be passed in HTML and HTTP environments.

Security-wise, SWT can only be symmetrically signed by a shared secret using the HMAC algorithm. However, [JWT and SAML tokens can use a public/private key pair in the form of a X.509 certificate for signing](#). Signing XML with XML Digital Signature without introducing obscure security holes is very difficult when compared to the simplicity of signing JSON.

JSON parsers are common in most programming languages because they map directly to objects. Conversely, XML doesn't have a natural document-to-object mapping. This makes it easier to work with JWT than SAML assertions.

Regarding usage, JWT is used at Internet scale. This highlights the ease of client-side processing of the JSON Web token on multiple platforms, especially mobile.

**Encoded** PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiiTxDjM0NTY3ODkwIiwibmFtZSI6IkpvG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA950rM7E2cBab30RMHrHDcEfjoYZgeFONFh7HgQ
```

**Decoded** EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE	
<pre>{   "alg": "HS256",   "typ": "JWT" }</pre>	
PAYLOAD: DATA	
<pre>{   "sub": "1234567890",   "name": "John Doe",   "admin": true }</pre>	
VERIFY SIGNATURE	
HMACSHA256( <code>base64UrlEncode(header) + "." +</code> <code>base64UrlEncode(payload),</code> <code>secret</code> <input type="checkbox"/> secret base64 encoded	



**SAML**

Debugger

**SAML ENCODED** PASTE A TOKEN HERE

```
PHNhbwXv0Jlc3BvbnnIIihbtG5zOnhbWxwPSJ1cm46b2FzaXM6bmFtZXM6dGM6U0FNtDoyLjA6chJvdG9jb2wiEIPEsJNjlxYzRjNGFINWQ2MGm3NjhiyZlIICBWZXJzaW9uPSlyLjAiElzc3ViWS5zdGFudD0iMjAxNC0xMC0xFNQxNDz0MjoxN1oiCBEZXN0aW5hdGlvbj0iaHR0cHM6Ly9hcAUXXVOaDAuY29l3Rlc3Rlc19zYW1scCl+PHNbWw6SXNzdWVylHtbG5zOnhbWw9lnVybjpYXNpczpuyW1lczp0YzpTQU1MOjluMDphc3NlcnRp24iPnVbjptYXR1Z2l0LmfIdGgwLmNvbTwvc2FtbDpjc3N1ZXI+PHNbWxwOIn0YXR1cz48c2FtbHA6U3RhdhvZq29kZSBWYXw1Z0idJuOm9hc1zOm5hbWwOnRjOINBTUw6M4wOnNOYXR1czpTdWNjZXNzI8+PC9zYWfscDpTdGF0dXM+PHNbWw6QXNzZXJ0aW9ulHtbG5zOnhbWw9lnVybjpYXNpczpuyW1lczp0YzpTQU1MOjluMDphc3NlcnRp24iIIFZlcnNpb249jluMCigSUQ9ll81Vks3TFQ3RmxpVWtrYVF1VzYzNgJyRjBERzVFMIg3NlqgSXNzdWVJbnNOYw50PSyMD E0LTEwLTE0VDE0OjMyOjE3LjIMVoIPjxzyWlsOklz3Vlcj51cm46bWF0dWdpdC5hdXRoMC5jb208L3NhbWw6SXNzdWVypJxTaWduYXR1cmUgeGlsbnM9lmh0dIA6Ly93d3cudzMuB3JnLzwMDAvMDkveGlsZHNPzYMPjxtaWduZWRJbmZvPjxDYw5vbmljWVxpemFoaW9uTWV0aG9kIEFsZ29yaXRobT0iaHR0cDovL3dy53My5cmcvMjAwMS8xMC94bWwtZxhlJWMxNG4jl8+PFNpZ25hdHvZU1ldGhvZCBBGdvcml0sG091mh0dHA6Ly93d3cudzMuB3JnLzwMDAvMDkveGlsZHNPzYNYc2etc2hhMSlvpjxszWZlcmVuY2UgVVJPSljxZvWSzdMVdDGbGIVa2thUVXNnl0YnJGMERHNUUzWdc2lj48VHJhbNmb3Jtc48VHJhbNmb3JtJl...
```

**SAML DECODED**  Prettify (not editable)  Expand

```

1  <samlp:Response
2    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" ID="_621c4e03-130a-4a20-830a-1a2a2a2a2a2a"
3    <saml:Issuer
4      xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">urn:matu...
5    <saml:Subject>
6      <saml:NameID Value="urn:oasis:names:tc:SAML:2.0:status:matu...
7      <saml:Status>
8        <saml:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:...
9      <saml:Assertion
10        xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" Version="...
11        <saml:Issuer>urn:matugit.auth0.com</saml:Issuer>
12        <Signature>
13          xmlns="http://www.w3.org/2000/09/xmldsig#"
14          <SignedInfo>
15            <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-c14n#"/>
16            <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
17            <Reference URI="#_5VK7LT7FlUkkaQuW6r4brF0DG5E3">
18              <Transforms>
19                <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
20                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
21              </Transforms>
22              <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
23              <DigestValue>ZDkfGO3HTu50hawzQVjsACzJwc=</DigestValue>
24            </Reference>
25            <SignedInfo>
26              <SignatureValue>1Fgpt7AaHcME2gTA158achvGQVqDwHSH...
```

**SAML INFO**

Comparison of the length of an encoded JWT and an encoded SAML



# HTTP 인증

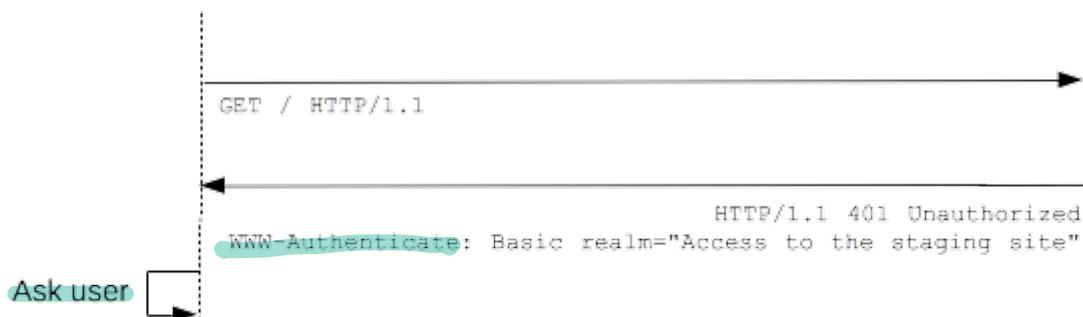
HTTP는 액세스 제어와 인증을 위한 프레임워크를 제공합니다. 가장 일반적인 인증 방식은 "Basic" 인증 방식입니다. 이 페이지에서는 일반적인 HTTP 인증 프레임워크를 소개하고 서버에 HTTP의 Basic 인증 방식으로 접근을 제한하는 것을 보여 줍니다.

## 일반적인 HTTP 인증 프레임워크

☞ RFC 7235는 서버에 의해 클라이언트 요청을 시도(challenge)하고, 클라이언트에 의해 인증 정보를 제공하기 위해 사용될 수 있는 HTTP 인증 프레임워크를 정의합니다. 이러한 시도와 응답 과정은 다음과 같이 작동합니다. 서버는 클라이언트에게 401 (Unauthorized) 응답 코드를 가지고 응답하며, 최소한 한 번의 시도에 포함된 `WWW-Authenticate` 응답 헤더로 권한을 부여하는 방법에 대한 정보를 제공합니다. 서버와 인증을 하기를 원하는 클라이언트는 `Authorization` 요청 헤더 필드에 인증 정보를 포함함으로써 인증을 수행할 수 있습니다. 클라이언트는 대개 사용자에게 비밀번호 프롬프트를 표시할 것이고 정확한 `Authorization` 헤더를 포함하는 요청을 만들 것입니다.

Client

Server





그림에서 보는 것과 같이 "Basic" 인증의 경우, 교환은 안전을 위해 HTTPS (TLS) 연결 위에서 발생하여야 합니다.

## 프록시 인증

동일한 시도 및 응답 메커니즘이 프록시 인증을 위해서도 사용될 수 있습니다. 이 경우, 이것은 인증을 요구하는 중간 프록시입니다. 리소스 인증 및 프록시 인증은 함께 존재할 수 있기 때문에, 헤더와 상태 코드의 다른 세트가 필요합니다. 프록시의 경우, 요청에 대한 상태 코드는 [407](#) (Proxy Authentication Required)이고, [Proxy-Authenticate](#) 응답 헤더는 프록시에 적용 가능한 최소한 하나의 요청을 포함하며, 그리고 [Proxy-Authorization](#) 요청 헤더는 프록시 서버에 인증 정보를 제공하기 위해 사용됩니다.

## 접근 거부

(프록시) 서버가 주어진 리소스에 대한 접근 권한을 얻기 위해 적절하지 않은 유효한 인증 정보를 수신한다면, 서버는 [403](#) [Forbidden](#) 상태 코드로 응답하여야 합니다. [401](#) [Unauthorized](#)나 [407](#) [Proxy Authentication Required](#)와는 다르게, 해당 사용자에 대한 인증은 불가능합니다.

## WWW-Authenticate와 Proxy-Authenticate 헤더

[WWW-Authenticate](#)와 [Proxy-Authenticate](#) 응답 헤더는 자원에 대한 액세스를 얻기 위해 사용되어야 할 인증 방법을 정의합니다. 이들은 인증을 하려는 클라이언트가 인증 정보를 제공할 방법을 알기 위해, 어떤 인증 스킴이 사용될 것인지를 구체적으로 적을 필요가 있습니다. 이들 헤더의 문법은 다음과 같습니다.

```

WWW-Authenticate: <type> realm=<realm>
Proxy-Authenticate: <type> realm=<realm>
  
```

여기서, <type> 은 인증 스킴입니다("Basic"은 가장 일반적인 스킴이며 아래에 소개되어 있습니다). realm은 보호되는 영역을 설명하거나 보호의 범위를 알리는데 사용됩니다. 이는 어떤 공간에 사용자가 접근하려고 시도하는지를 알리기 위하여, "중간 단계의 사이트에 대한 접근"과 같거나 또는 비슷한 메시지가 될 수 있습니다.

## Authorization와 Proxy-Authorization 헤더

Authorization 와 Proxy-Authorization 요청 헤더는 사용자 에이전트가 (프록시) 서버에 인증을 하기 위한 인증 정보를 포함합니다. 여기에서 type은 다시 한 번 필요하며 credentials이 뒤에 따라옵니다. credentials 부분은 어떤 인증 스킴이 사용되는지에 따라 인코딩이 되어 있거나 암호화가 되어 있을 수 있습니다.

```
|Authorization: <type> <credentials>
|Proxy-Authorization: <type> <credentials>
```

### 인증 스킴 *Authenticate*

일반적인 HTTP 인증 프레임워크는 여러 인증 스킴에 의해 사용됩니다. 스킴은 보안 강도와 클라이언트 또는 서버 소프트웨어에서 사용 가능성에 따라 달라질 수 있습니다.

가장 일반적인 인증 스기는 아래에서 좀 더 자세하게 소개할 "Basic" 인증 스기는 IANA는 ↗ [인증 스기의 목록](#)을 유지하고 있으나, Amazon AWS와 같은 호스트 서비스에 의해 제공되는 다른 스기도 존재합니다. 일반적인 인증 스기는 아래를 포함합니다.

- **Basic** (↗ [RFC 7617](#)를 보십시오. base64-encoded credentials. 더 많은 정보는 아래를 확인하십시오.),
- **Bearer** (↗ [RFC 6750](#)를 보십시오. bearer tokens to access OAuth 2.0-protected resources),
- **Digest** (↗ [RFC 7616](#)를 보십시오. Firefox에서는 md5 해싱만 지원합니다. SHA 암호화 지원을 위하여 ↗ [bug 472823](#)을 확인하십시오.),
- **HOBA** (↗ [RFC 7486](#) (draft)를 보십시오. HTTP Origin-Bound Authentication, digital-signature-based),

- Mutual ([draft-ietf-httpauth-mutual](#)를 참조하십시오),
- AWS4-HMAC-SHA256 ([AWS docs](#)를 참조하십시오).

# Basic 인증 스킴

"Basic" HTTP 인증 스킴은 [RFC 7617](#)에 정의되어 있는데, 이는 base64를 이용하여 인코딩된 사용자 ID/비밀번호 쌍의 인증 정보를 전달합니다.

## Basic 인증의 보안

사용자 ID와 비밀번호가 평문으로 네트워크를 통해 전달되기 때문에 (이것은 base64로 인코딩 되어 있으나, base64는 복호화가 가능한 인코딩이므로), Basic 인증 스킴은 안전하지 않습니다. HTTPS / TLS는 basic 인증과 함께 사용되어야 합니다. 이러한 추가적인 보안상의 향상이 없이는, basic 인증은 민감하거나 귀중한 정보를 보호하는 데 사용되어서는 안 됩니다. *기억하세요!*

## Apache와 Basic 인증으로 접근 제한하기

Apache 서버에서 디렉터리를 비밀번호로 보호하기 위해서는, `.htaccess` 와 `.htpasswd` 파일이 필요할 것입니다.

`.htaccess` 파일은 일반적으로 이렇게 생겼습니다.

```
AuthType Basic
AuthName "Access to the staging site"
AuthUserFile /path/to/.htpasswd
Require valid-user
```

`.htaccess` 각각의 줄이 콜론(":")으로 나뉘어져 있는 사용자 이름과 비밀번호를 포함하는 `.htpasswd` 파일을 참조합니다. 여러분은 그들이 [암호화](#)되어 있기 때문에 (이 경우에는 md5) 실제 비밀번호를 보지 못할 수도 있습니다. 여러분이 원한다면 `.htpasswd` 파일의 이름을 바꿀 수 있지만, 이 파일은 누구에게도 접근 가능해서는 안 됨을 유의하십시오. (Apache는 일반적으로 `.ht*` 파일에 대한 접근을 제한하도록 설정되어 있습니다)

```
aladdin:$apr1$ZjTqBB3f$IF9gdYAGlMrs2fuINjHsz.
```

## nginx와 Basic 인증으로 접근 제한하기

nginx에서 여러분은 보호하려는 위치와 비밀번호로 보호될 영역의 이름을 나타내는 `auth_basic` 명령어를 적어줄 필요가 있습니다. 위의 Apache 예제에 있는 것과 같이, `auth_basic_user_file` 명령어는 암호화된 사용자 인증 정보를 가지고 있는 `.htpasswd` 파일을 가리킵니다.

```
location /status {
    auth_basic      "Access to the staging site";
    auth_basic_user_file /etc/apache2/.htpasswd; 암호화된 사용자 인증정보 저장
}
```

## URL에 인증 정보를 사용하여 접근하기

또한 많은 클라이언트들은 아래와 같이 사용자 이름과 비밀번호를 포함하는 인코딩된 URL을 사용하여 로그인 프롬프트를 피하게 합니다.

`https://username:password@www.example.com/`

이러한 방식의 URL은 더 이상 사용되지 않습니다. Chrome에서, URL의 `username:password@` 부분은 보안 상의 이유로 ↗ 제거됩니다. Firefox에서는, 해당 사이트가 진짜로 인증이 필요한지를 체크하며, 그렇지 않으면 Firefox는 프롬프트로 "You are about to log in to the site “www.example.com” with the username “username”, but the website does not require authentication. This may be an attempt to trick you."와 같이 경고합니다.

## 함께 보기

- `WWW-Authenticate`
- `Authorization`
- `Proxy-Authorization`
- `Proxy-Authenticate`
- `401, 403, 407`

# 인증 방식과 프로세스 비교

ljinsk3 · 2020년 5월 19일

0

authentication



## 개발 공통 지식 (웹, 인프라, 아키텍처)

▼ 목록 보기

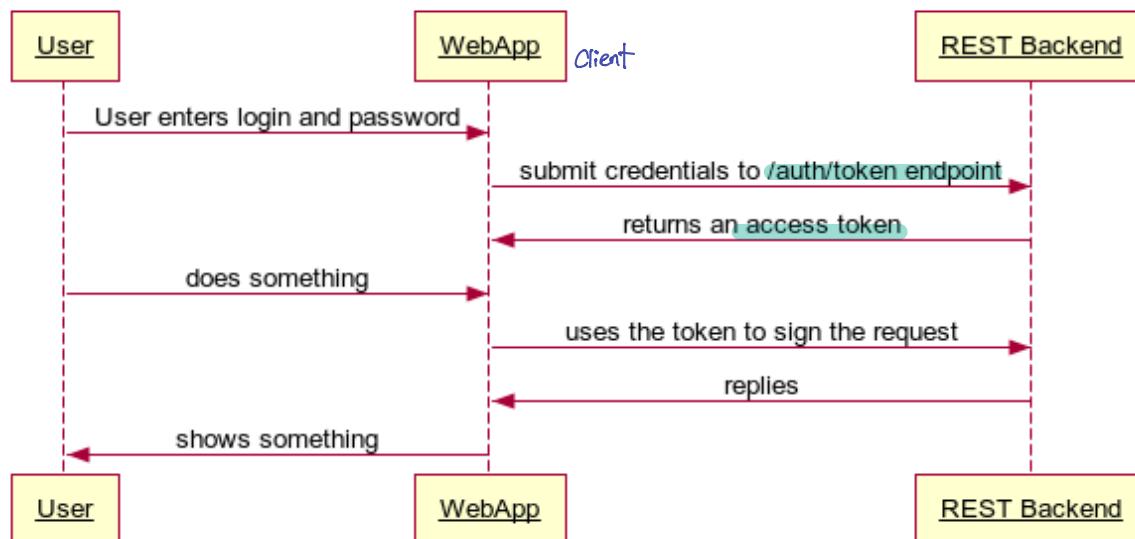
3/4



## 인증 프로세스

- 시퀀스 다이어그램

### Authentication Sequence



- 유저가 계정과 비밀번호로 로그인을 시도
- (서버에서 패스워드 확인 등으로) 인증이 완료되면 클라이언트로 accessToken을 발급
- 이후 요청 시마다 해당 accessToken을 사용하여 인증 절차 생략

## 인증 authentication

*401 Unauthorized  
왜 auth하고 했을까?*

- 방문자가 자신이 회사 건물에 들어갈 수 있는지 확인받는 과정
- 인증된 유저인지 아닌지를 확인하는 작업
- 인증되지 않으면 401 status code를 내려보낸다.

## 인가 authorization

*Level to Access, 403 Forbidden*

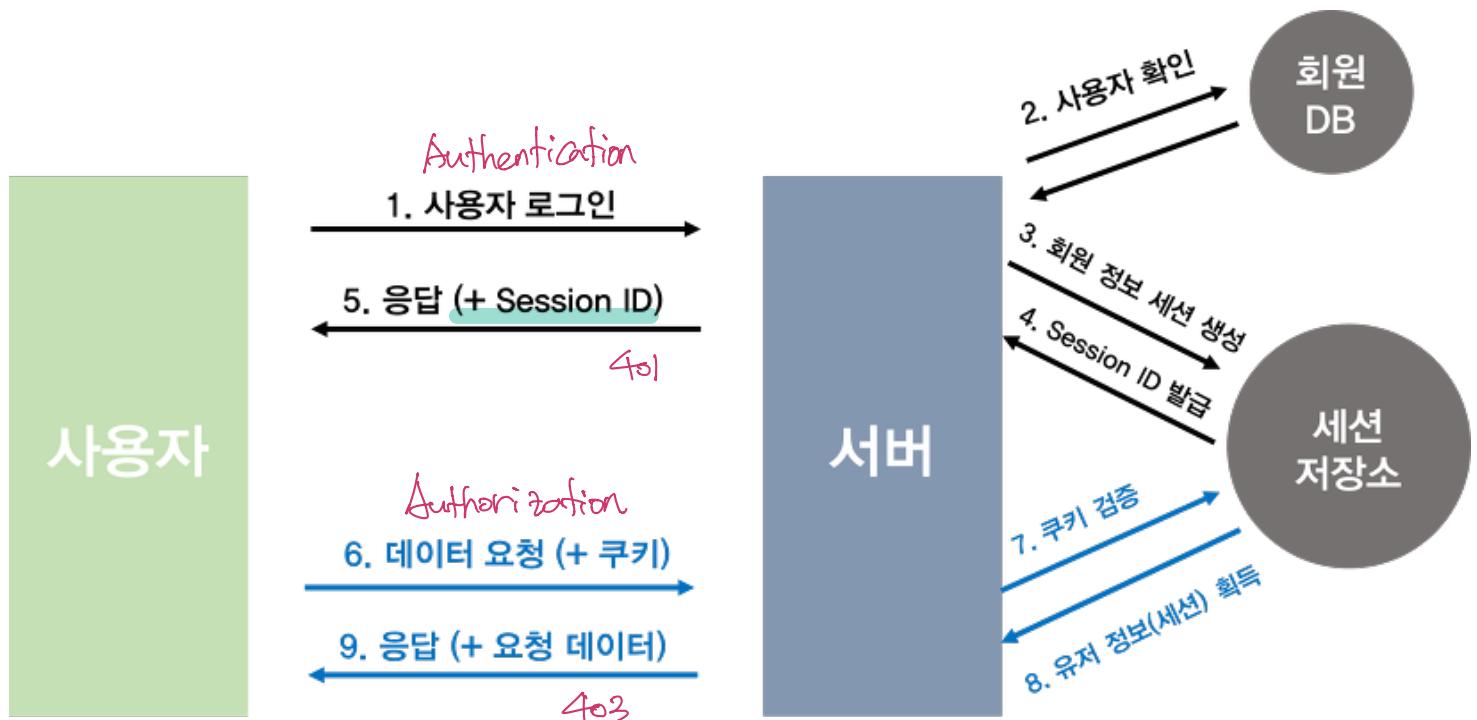
- (인증된) 방문자가 회사 건물에 방문했을 때, 허가된 공간에만 접근 가능하다.
- 특정 서비스, 페이지에 접근 권한을 부여하는 것을 의미한다.
- 'JWT'를 쓰면 인가가 수월함. 서버가 유저 정보를 갖고 있으므로 데이터에서 유저의 권한 정보도 읽어들이면 됨.
- 인가되지 않으면 403 status code를 내려보낸다. ~~Authentication~~ means checking the identity of the user making a request.

**Authorization** refers to the set of rules that is applied to determine what a user is allowed to see / do.

## 1. Session/Cookie

- Session/Cookie 방식은 가장 기본적이면서 전통적인 방식이다.
- 사용자의 정보를 서버(세션저장소)에 세션이라고 불리는 사용자의 정보를 저장한다. (*stateful*)
- 클라이언트에게는 Session ID를 발급하고 클라이언트는 이 정보를 쿠키에 저장한다.
- 이제 매 요청마다 이 Session ID를 함께 보내고, 서버는 세션저장소에서 유저 인증을 수행한다.
  - 인증이 성공하면 요청을 핸들러로 보낸다.
  - 인증이 실패하면 응답으로 예외를 반환한다.
- Session ID와 같은 인증 기록은 서버 및 클라이언트 측 모두에 유지된다. 주요한 정보가 다 서버 쪽에 남을 수 있다는 것이 큰 장점이기도 하지만, 요청때마다 매번 세션 저장소에 접근하여 확인해야 한다는 것이 단점이 될 수도 있는 방식이다.

*# 실제 구현할 땐, 세션 세이브 따로 구현해야 하잖아?*



### cookie란?

stateless한 HTTP 프로토콜 특성 때문에 유저의 정보를 유지할 수 없는 한계를 극복하기 위해 나온 웹 브라우저에 존재하는 작은 텍스트 파일이다. 이 파일은 브라우저가 관리하게 되고 일반적으로 쿠키에는 만료일이 있다. 브라우저는 쿠키를 통해 접속자의 장치를 인식하고, 접속자의 설정과 과거 이용내역에 대한 일부 데이터를 저장한다. key-value 형식으로 저장된다.

## 2. Basic HTTP Authentication

Authentication이 어떤  
이 헤더를 쓰는 이유?

- HTTP는 다른 인증 프로토콜에 맞추어 확장할 수 있는 **Authorization** 제어 헤더를 제공한다.
- HTTP Authorization 헤더는 서버의 사용자 에이전트임을 증명하는 **자격을 포함**하여, 보통 서버에서 401 Unauthorized 상태를 **WWW-Authenticate** 헤더로 먼저 알려준다.
- 클라이언트는 **username**과 **password** 값을 **: (콜론)**으로 합친 뒤, 통채로 **Base64** 인코딩을 통해 **credentials** 생성하여 서버로 보낸다.
- 서버는 **credentials** 값을 **Base64**로 디코딩하여 **username**과 **password**를 분리하여 인증한다.



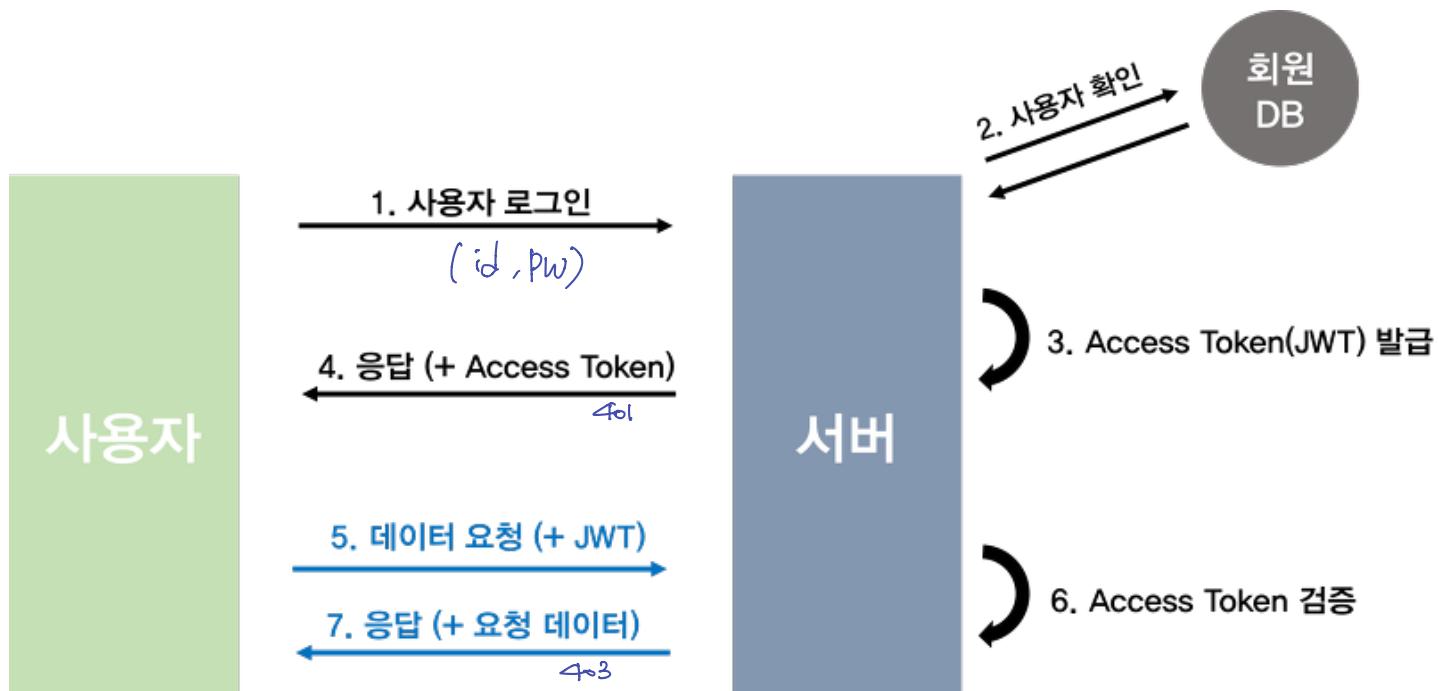
주의할 점은 Base64 인코딩은 암호화나 해싱을 의미하는 것이 아니다. 이 방법은 인증에 대해서 문자를 그대로 보내는 것과 동일하다고 할 수 있기 때문에 반드시 https를 사용해야 한다. 인코딩을 해서 암호처럼 보이지만, 인코딩 디코딩 방식이 정해져있기 때문에 어느 누가 중간에 통신을 탈취하여 해킹할 위험이 존재하는 것이 단점이다.

### 3. Bearer HTTP Authentication

- 로그인 시 서버로부터 token을 발급받고, 이후 요청 시 Authentication 헤더에 토큰을 실어 보낸다.
- token을 만들 때 JWT 방식으로 생성하는 것이 특징이다.
- 세션 정보가 클라이언트/서버 각각의 저장소에 흩어져있지 않고, 토큰 자체에 내장되어 있다.
- 무상태 HTTP를 유지할 수 있기 때문에 속도도 빠르고 보안도 뛰어나다고 할 수 있다.

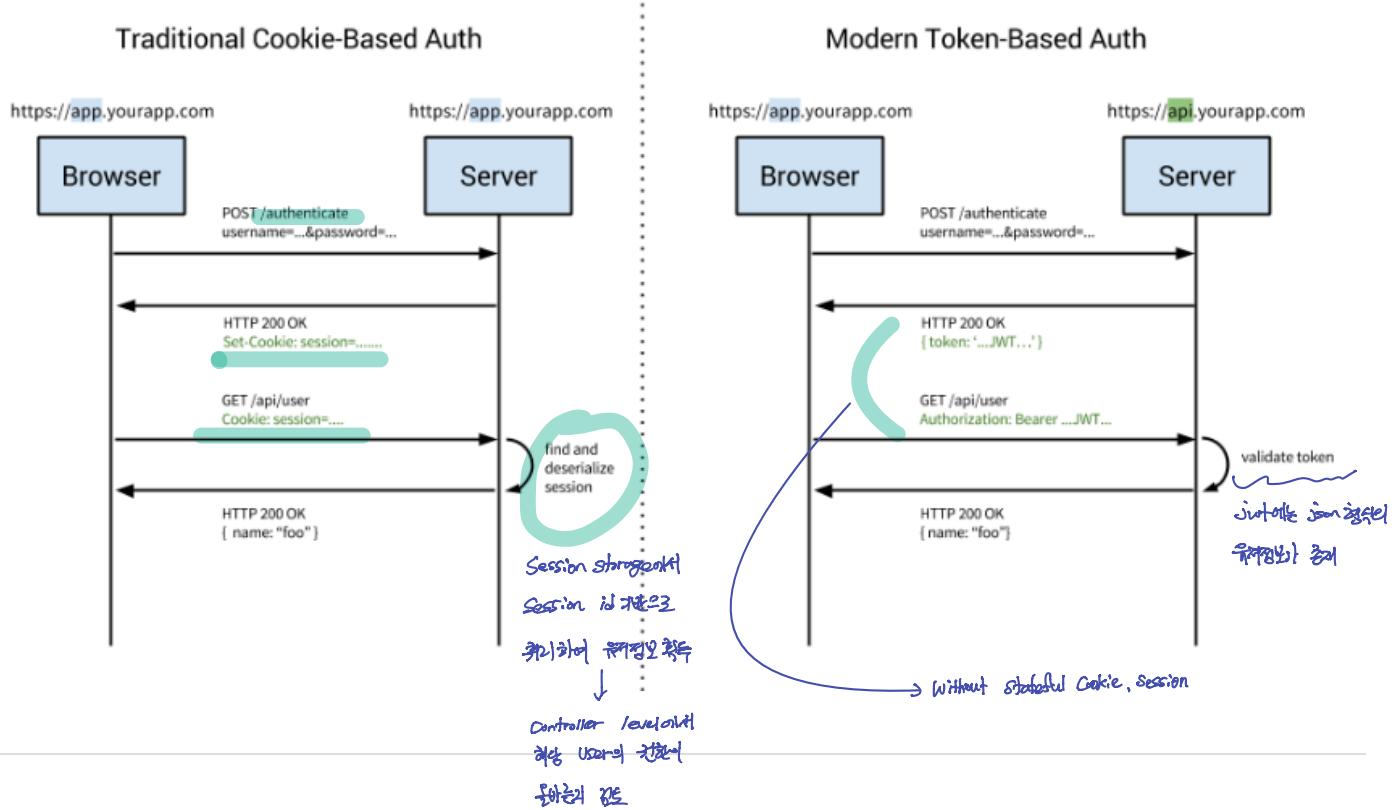
\* stateless

→ Bearer는 OAuth2 방식이랑 어떤 차이인가?



## Session/Cookie vs Token

- Session/Cookie 방식은 sessionId를 매번 세션 저장소에서 찾아야 하는 번거로움이 있지만, token은 해독하여 유효성만 검증하면 된다.
- 여기서 유효성이란 토큰의 만료기간, 토큰 안의 세션정보 등을 확인할 수 있다.





216 Followers

About

Follow

# Session vs Token Based Authentication

Why do we need session or token for authentication?



Sherry Hsu Jun 30, 2018 · 3 min read

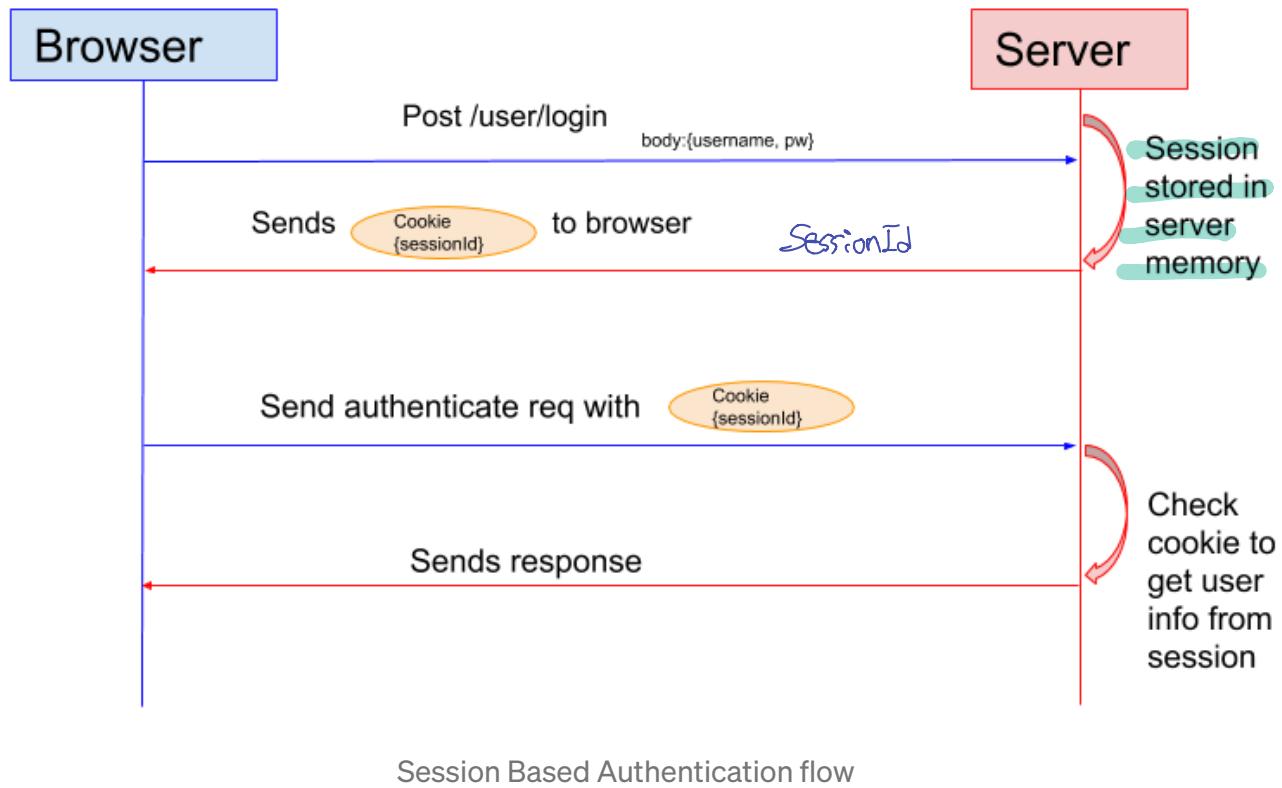
HTTP is stateless. All the requests are stateless. However, there are situations where we would like our states to be remembered. For example, in a on-line shop, after we put bananas in a shopping cart, we don't want our bananas to disappear when we go to another page to buy apples. ie. we want our purchase state to be remembered while we navigate through the on-line shop!

To overcome the stateless nature of HTTP requests, we could use either a session or a token.

## Session Based Authentication

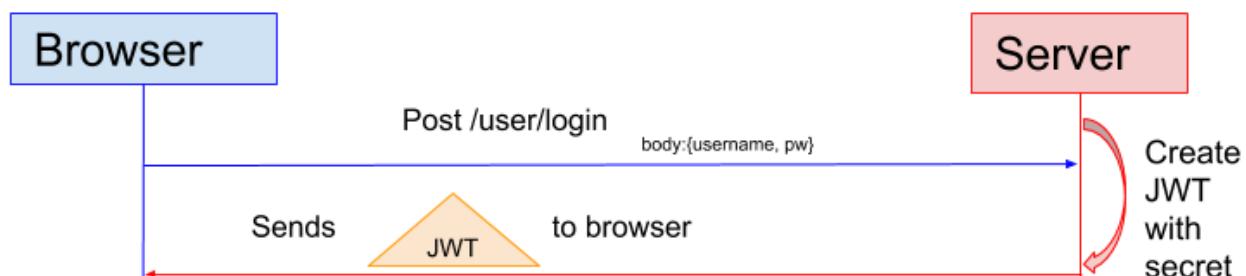
In the session based authentication, the server will create a session for the user after the user logs in. The session id is then stored on a cookie on the user's browser. While the user stays logged in, the cookie would be sent along with every subsequent request. The server can then compare the session id stored on the cookie against

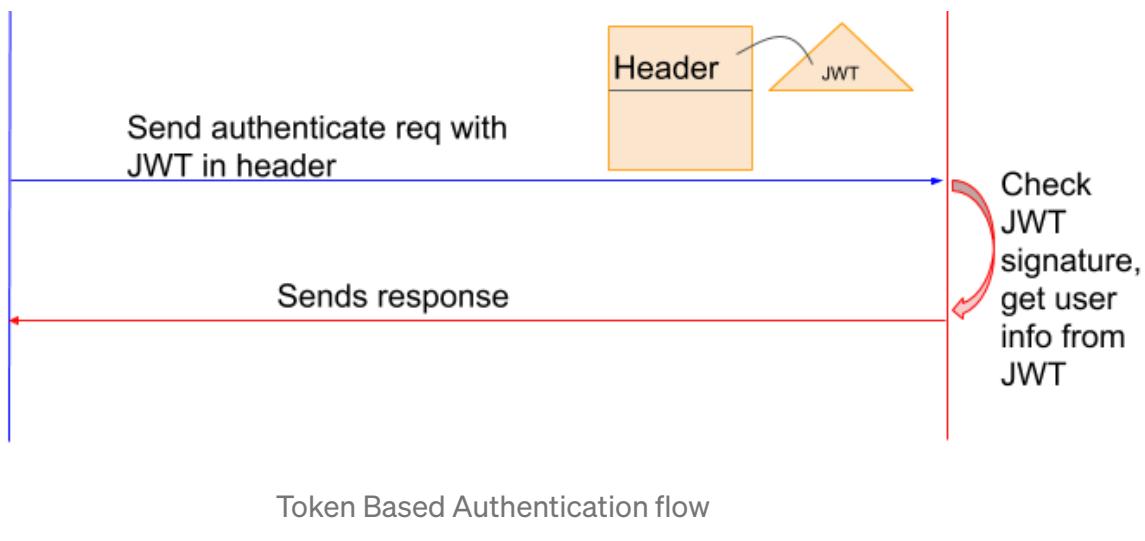
the session information stored in the memory to verify user's identity and sends response with the corresponding state!



## Token Based Authentication

Many web applications use JSON Web Token (JWT) instead of sessions for authentication. In the token based application, the server creates JWT with a secret and sends the JWT to the client. The client stores the JWT (usually in local storage) and includes JWT in the header with every request. The server would then validate the JWT with every request from the client and sends response.





The biggest difference here is that the user's state is not stored on the server, as the state is stored inside the token on the client side instead. Most of the modern web applications use JWT for authentication for reasons including scalability and mobile device authentication.

## Node Modules for JWT

jsonwebtoken library can be used to created the JWT token on the server. Once the user is logged in, the client passes the JWT token back on the header.authorization.bearer attribute.

```
{
  method: "GET",
  headers:{
    "Authorization": "Bearer ${JWT_TOKEN}"
  }
}
```

this is for JWT

Middleware, express-jwt, can be used to validate the JWT token by comparing the secret.

## Scalability

**Session based authentication:** Because the sessions are stored in the server's memory, scaling becomes an issue when there is a huge number of users using the system at once.

**Token based authentication:** There is no issue with scaling because token is stored on the client side.

## Multiple Device

Session-based는 Cookies를 사용하며, 이는 일반적으로 Single-domain에서 활용된다.  
(CORS 문제)

**Session based authentication:** Cookies normally work on a single domain or subdomains and they are normally disabled by browser if they work cross-domain (3rd party cookies). It poses issues when APIs are served from a different domain to mobile and web devices.

**Token based authentication:** There is no issue with cookies as the JWT is included in the request header.

Token Based Authentication using JWT is the more recommended method in modern web apps. One drawback with JWT is that the size of JWT is much bigger comparing with the session id stored in cookie because JWT contains more user information. Care must be taken to ensure only the necessary information is included in JWT and sensitive information should be omitted to prevent XSS security attacks. JWT는 XSS에서 안전하지 않다. 민감정보는 사용X

## Reference

# OAuth2와 JWT, 웹기반 SSO 인증

클라우드컴퓨팅, 2017-09-13

언어 선택 ▼

인터넷 기반 서비스, 특히 Cloud Computing 환경에서 개발되는 Application이라면 웹이든 모바일이든 꼭 사용하게 되고야 마는 OAuth2와 JWT, 그것을 바탕으로 하는 인증/인가 체계의 개념에 대하여 최대한 쉽게 정리한다. (사실, 쉽게 쓴다고 해도 쉬운 내용이 아니지만 일반적으로 접할 수 있는 Protocol 자체에 대한 관점보다는 사용하는 입장과 왜 이렇게 쓰는지를 이해하기에 조금이라도 수월한 글이 되었으면 좋겠다)

아무튼, 거의 일년만의 포스팅인데 느낌은 10년 만인 것 같다. 이제 가을이 오고 그동안의 바빴던 일들도 대충 정리가 되어 숨통이 트여가고 있으니, 그간 남겨놓은 기록들을 웹으로 옮길 시간이 된 것 같다. 오늘은 그 첫번째 시간으로 OAuth2의 이해. 왜 이 글이 Reboot의 시작이 되었냐면, 얼마 전에 이 기술을 기반으로 한 프로젝트를 하나 마무리했기 때문이기도 하다. 아무튼, 시작.

IT 세상이 발전하면서, 그리고 특히나 Web을 중심으로 한 Cloud 세상이 오면서 더욱 중요해진 것 중의 하나가 “어떻게하면 흩어져 있는 Application들의 인증 관리를 중앙에서 쉽게 할 것인가” 하는 것인데, 이것을 푸는 방식 중 대표적인 것이 바로 OAuth2이다. Google, Facebook 등을 비롯한 대부분의 인터넷 기반 Application들이 이것 또는 그 변종을 사용하여 스스로를 인증하거나 누군가에게 인증 서비스를 제공하고 있다. 또, 근래에는 OpenID Connect라는(예전의 URL 기반 OpenID와는 다르다) 것이 등장하여 OAuth의 개념을 보다 편리하게 쓸 수 있도록 한 기술도 등장한 상태이다.

# 네안데르탈 API Key

근래의 Web Application은 초기 인터넷처럼 하나의 큰 Layer(3-Tier 구조와는 조금 다른 얘기)로 구성하기 보다는 Layer를 나누고 Sector를 나누는 방식으로 이루어진다. 풀어서 말하면, 화면 구성은 앞쪽 Layer의 'U'가 하고, 인증은 그 옆에 선 'A'가 하고, 업무 하나는 'J'가, 다른 하나는 'K'나 'L'이 나란히 뒤에 서서 처리하는 모양이 된다.

이렇게 분할하여 개발(MSA와 유사한 개념이나, MSA의 개념이 정립되기 전, 또는 SOA가 정립되는 과정에서도 그래왔음)을 하게 되면, 각각의 부분을 전체의 이해 없이도 개발할 수 있다는 점이라든지, 뭐랄까 흔히 얘기하는 Divide and Conquer가 가능해지기 때문에 개발과 유지보수가 매우 편리해진다. 그런데 현실은 그냥 그림으로 바기와는 좀 다르다. 이 서로 떨어진 것들이 어떻게 화합과 조화를 이루며 공존할 것인가가 매우 큰 과제가 되기 때문이다. **핵심과제**



어쨌든, 그것을 푸는 가장 첫 번째 방식은 Application을 쪼개되, DBMS는 공유하는 것이었다. 앱을 따로 만들어도 DBMS(또는 그에 준하는 무언가)를 통해 자료를 공유하면 매우 간단하게 “통합”을 할 수 있다. 그런데 이 핵심적인 부분을 공유하다 보니, 만약 내 앱을 고치가가 그 중 하나의 Table을 주정하고 싶을 경우, 그 영향도를 쉽게 알 수 없으니 막막해지게 된다.

이렇게, DBMS를 공유하는 방식은 얼핏 생각해도 “쪼갠 게 쪼겐 게 아니다.”라는 결론에 이른다.(고는 하지만 지금도 그런 경우가 꽤 많은 것 같다)

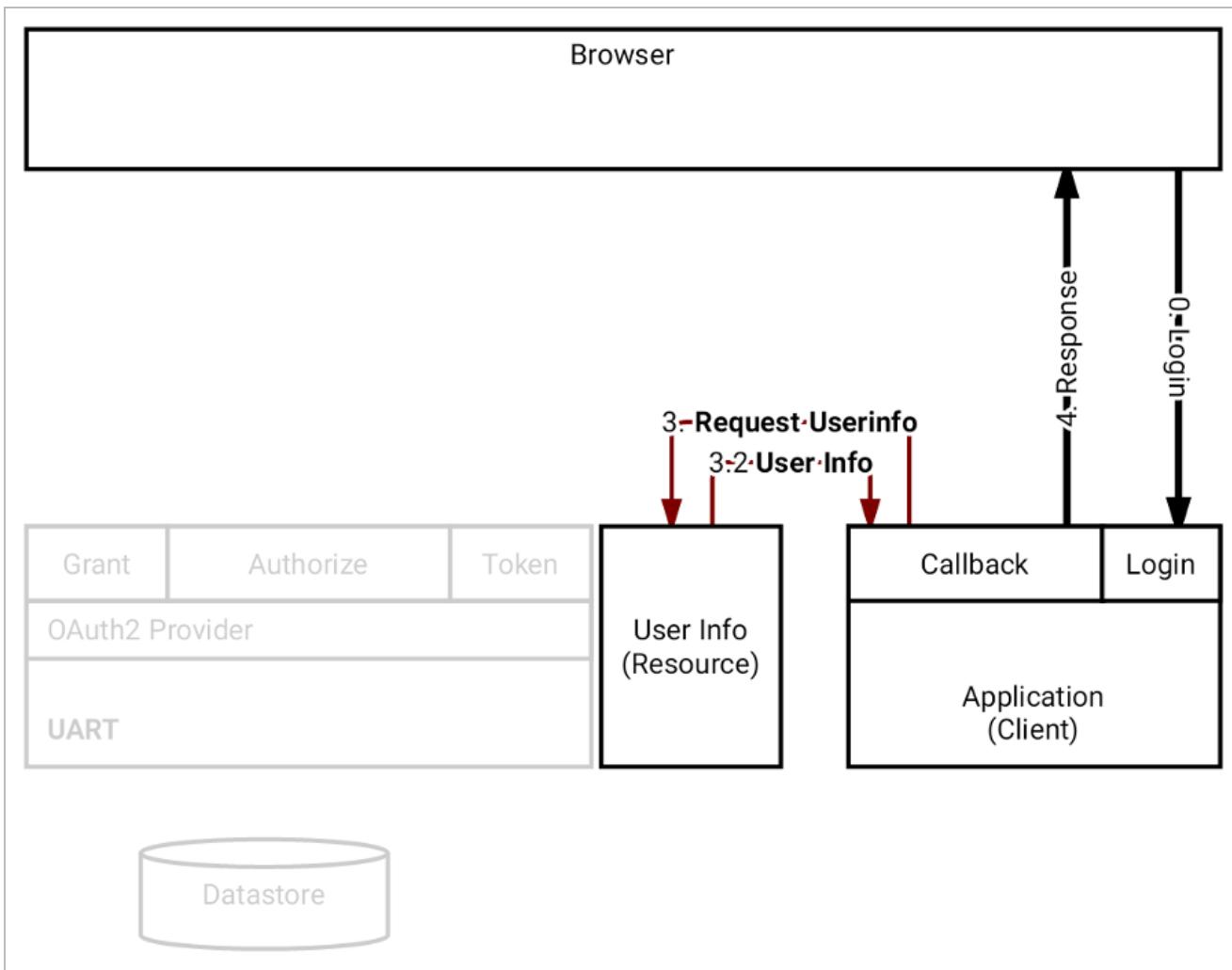


그래서 등장한 것이, Data Tier에서 공유하지 않고 Logic Tier에서 공유하는 방식인데, 그게 일반적으로 API 방식으로 이루어진다. “사용자 정보든, 뭐든, 필요로 하는 녀석이 API로 요청해라. 그럼 내가 알아서 찾아서 줄게” 뭐, 이런 얘기다. 내 DBMS는 내가

알아서 관리하고 뒤지되, 정해진 API 규역으로 요청한 것만 잘 지키면 서로 다른 Module이나 Application 간의 공유가 간편해지고 각각의 독립성으로 보장할 수 있게 된다.

그런데 이렇게 API를 열고 보니... 접속자의 인증과 인가 관리가 필요해진다. 여기서 잠깐, 인증과 인가, 영어로는 Authentication과 Authorization이라고 표현하는데, “인증”이라 함은 “나 맞아”를 증명하는 단계이고 “인가”는 인증이 된 상태에서 “나 반장이야” 뭐 이런거? 좋지 않은 군대의 예를 들면 나는 그 시절에 “비취인가”라고 불리는, 비밀을 취급하도록 “허용된 자격”을 가지고 있었다. 뭐, 흔한 3급이었지만. 이게 인증과 인가다.

이런 API 통신의 인증과 인가 관리를 위해 가장 먼저 등장한, 그리고 가장 널리 보편적으로 쓰이는 기술이 바로 (네안데르탈) API Key 이다.



그림에서 보는 바와 같이, Application(Client)라고 써있는 박스가 업무를 담당하는 App 인데, 이 App이 사용자 인증정보를 알고 싶으면 User Info(Resource)라고 써있는 인증 App에게 물어본다.(번호가 이상하지만 3번) 그런데 이렇게 물어볼 때, 자신을 증명하기 위해 API Key라는 허가증을 같이 보내게 된다. 인증 App은 요청과 함께 온 API Key를 자신의 인증 저장소와 맞춰봐서, 이게 누구의 Key인지, 그가 가진 권한이 뭔지 확인한 후에 이에 부합하는 응답을 준다.

훌륭하다. 몇 가지 부분만 빼면, 정말 훌륭하고 아름다운 그림이다. (아, 내가 머리를 자르러 가서 애써 그린 이 그림 말고, 그 동작 방식의 그림 말이다.)

문제가 되는 부분은, 아무리 통신구간 암호화를 한다고 해도... 이 Key가 유출된 경우를 대비하지 않을 수가 없다는 점이다. 그래서 주기적으로 Key를 업데이트해야 한다는 문제도 있고, 이 업데이트가 양쪽에서 잘 맞아 이루어지지 않으면 서비스 장애로 이어진다는 문제도 있다. (깜박하고 안 바꿨는데요, 뭐 이런 거) 또, Key 하나로 이런 절차를 감당하다 보니... 별도의 ACL 등을 활용한 접속제어 등을 하지 않을 경우,(가령, 허가된 IP로부터의 접근에 대해서만 허용한다든지...) 보안 문제로 이어질 수 있다. 그런데 그걸 다 관리하려면 얼마나 복잡해... 특히나 Cloud 의 상징인 Auto Scaling 등으로 서버의 수가 변하고 IP가 유동적이라면 사실상 관리가 되지 않는다.

1. Authorize Code : 단기 Browser (client)  $\leftrightarrow$  Auth. Server

2. Access token : 장기, APP Server  $\leftrightarrow$  Auth. Server

## 돌돌한 OAuth2

항상 가려운 부분이 있으면 긁을 수 있는 효자손이 등장해 왔으니, API Key의 단점을 보완해줄 누군가가 나타났을 것인데, 그렇게 OAuth2의 시대가 열렸다. 앞서 가려웠던 부분을 조금 다른 방향으로, 그리고 단지 Application 대 Application의 약속을 넘어 사용자(관리자 말고 사용자)가 적당히 개입하는 방식으로 발전한 것인데...

앞서 본 API Key가 요청하는 서버와 요청받는 서버 정도로 단순한 구성이라면, 여기서는 요청하는 서버, 요청받는 서버, 그리고 인증하는 서버 등으로 조금 세분화가 이루어진다. (그래야, 중립적인 위치에서 인증을 전담하는 구조가 성립하므로...)

1

아래 그림의 번호를 따라 흐름을 보면, 먼저 0번의 로그인 요청이 있다. 꼭 로그인이 아니더라도, 가령, “상품 목록을 보여줘” 이런 것이 될 수도 있다. 그러면 업무서버(Application)는 일단 이 사용자가 자신에게 로그인이 되어 있는지를 본다. 그런데 “로그인이 되어있지 않다”, 그렇다면 파란 선의 1.을 따라서 인증서버에게 사용자를 돌려보낸다(Redirection). 단지 로그인이 아니라 인가가 없는 상태를 포함해서

하는 이야기이다. (참고로, 그림에서 파란선은 모두 Redirection을 표시한 것이고, 빨간 선은 사용자나 사용자의 브라우저가 개입하지 않는 서버간 직접 통신, 굵은 선은 사람이 개입하는 선이다.)

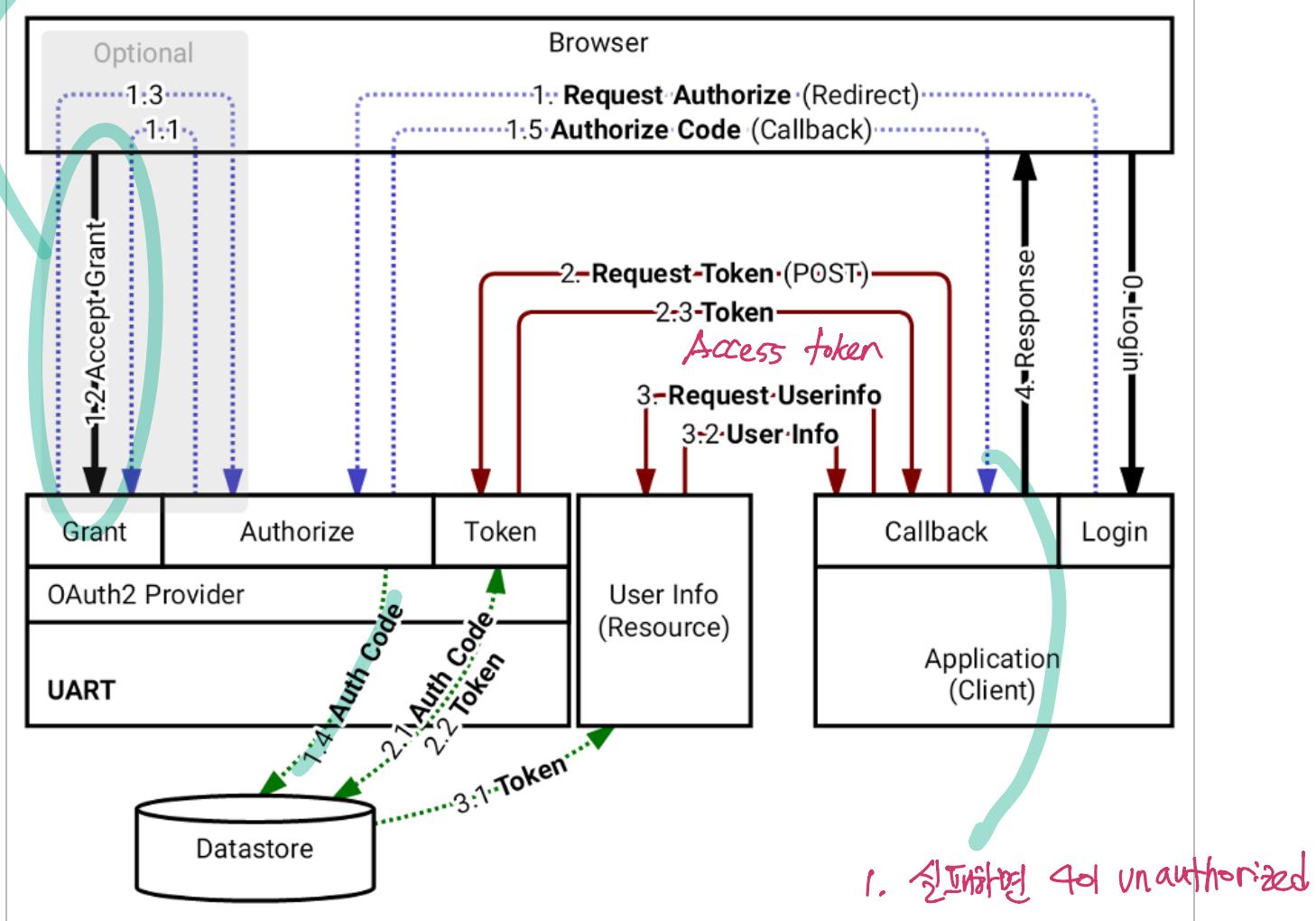
이렇게 1번을 통해 간접적으로 Authorize 요청을 받은 인증서버는, 이 사용자가 회원인지, 그리고 인증서버에 로그인이 되어있는지를 확인한다. (이 과정은 그림에는 생략되어 있다.) 인증을 거쳤다면 이 사용자가 최초에 인가를 요청한 서버(업무)에 대한 사용의사가 있는지(또는 권한이 있는지) 확인한다. (이 과정을 **Grant**라고 하는데, 대체로 인증서버는, 사용자의 의지를 확인하는 수준에서 Grant 처리를 하게 되고, 각 업무 서버는 다시 권한 관리를 할 수도 있다.) 만약, 사용자가 아직 Grant한 상태가 아니라면 사용자에게 Grant를 요청하게 된다. (흐름 1.1)

### ⚡ **Grant는 사용자의 의지** *Grant vs Authorization*

Grant는 인가와는 조금 다르다. 인가는 서비스 제공자의 입장에서 사용자의 권한을 보는 것이지만, Grant는 사용자가 자신의 인증 정보(이름, 메일주소, 전화 번호 등 개인정보가 포함되어 있을 수 있다)를 자신이 사용하려는 App에게 제공할지 말지를 결정하는 과정이다. (인가와 관련된 얘기는 다음에)

이렇게 사용자가 Grant 요청을 받게 되면, 이 순간에 한하여 사용자 개입이 일어난다. “업무서버가 내 인증정보를 읽을 수 있도록 허용해”라는 의미로 사용자가 직접 Grant를 해줘야 하기 때문인데, 이렇게 “응, 허가해”라고 답을 하고 나면 흐름 1.3을 타고 다시 인증서버의 인가 처리기에 되돌려 보내진다. 그러면 이 처리기는 흐름 1.5를 따라 인증과 인가가 되었다는 결과와 이후의 과정에서 사용하게 될 인가 코드(Authorize Code)를 업무서버에게 전달하게 된다. 그런데 주의해서 볼 부분은, 1.3과 1.5 사이의 흐름 1.4인데, 이 인가 코드를 자신의 저장소에 저장해두게 된다.

구글에서 사용자에게 허용할지 물어봄. (사용자는 허용을 누르는 단계)



이렇게 Authorize Code를 받은 업무서버는 이 코드를 이용하여 사용자의 인증 정보에 접근할 수가 있다...가 아니라, 이 코드는 보안을 위해 매우 짧은 기간 동안에만 유효하고, 그 기간 안에 업무서버는 다시 Access Token이라는 것을 인증서버에게 받아가야 한다. 이 Access Token은 앞서 본 API Key와 유사하게 사용되게 된다.

Access Token으로 넘어가기 전에, 중요한 포인트가 있는데, 이 Authorize Code의 수명과 전달 방식이다. 그림에서 보듯, 이 전달은 웹브라우저를 경유하게 된다. (흐름 1.5; Browser 박스를 지나는 흐름은 모두 그런 것임) 이 때에도 역시 HTTP Redirect가 활용되는데, 이 Redirect할 위치를 업무서버와 인증서버가 알고 있으면서 그것을 검증하고, Client인 업무서버의 주도적인 전달이 아닌 Redirect를 활용한 인증서버

주도의 흐름이 되다 보니, 다른 곳으로 샐 가능성이 상대적으로 줄어든다.(말하자면, 인증서버가 업무서버를 IP가 아닌 논리적 URL을 가지고 인증하는 것으로, API Key와 함께 자주 쓰이는 “접속 IP 제한”과 유사한 역할을 하는 것이다.

2

아무튼, Authorize Code를 받은 Application, 업무서버는 다시 이 코드를 이용해서 빠른 시간 안에 인증을 해준 서버에게 “네가 보내준 이거 잘 받았어. 이거 맞지? 이제 물건 줘”라고 요청하게 되는데, 이게 2번 흐름, Request Token 단계이다. 뭔가... 아버지가 남긴 칼날 조각을 들고 고구려로 찾아가는 유류... 같다. 인증서버는 이 Code를 다시 흐름 2.1을 따라 자신의 저장소에 저장해둔 정보와 일치하는지 확인하고, 이번에는 긴 유효기간을 갖는, 그리고 앞으로 실제 리소스 접근에 사용하게 될 Access Token을 업무서버에게 전달한다. 그리고 그 전에, 이번에도 이 Token을 자신의 저장소에 저장한다.

*Access token.*

이제, Application은 이 Token을 들고 실제 사용자 정보가 있는 서버에 찾아가서 “나 공증된 Token 가지고 왔어, 리소스 줘”라고 요청한다. (흐름 3) 그러나 리소스 서버 역시 호락호락하지는 않아서, 단지 정부 발급 증명서라고 해서 그냥 믿기 보다는 얼른 무전기를 꺼내서 중앙에 물어보게 된다. “번호 XYZ9876 맞나요?” 왜냐하면, 이 Token이라는 것은 그 자체로는 의미가 없는 복잡한 문자열일 뿐이며, 그것의 유효성은 인증서버의 저장소에만 있기 때문이다. 이렇게 확인을 거친 후, 리소스 서버는 업무서버가 원하는 답을 건내준다.

이미 말한 바와 같이, 이 Token은 무의미한 문자열로, 기본적으로 정해진 규칙에 의해 발급된 것이 아니다. 그래서 증명확인이 필요할 때마다, 인증서버에 어떤 식이든, DBMS 접근이든 또다른 API를 활용하든 접근하여 유효성을 확인해야 한다. (심지어 공증여부 뿐만 아니라, 유효기간 문제도 있다.)

조금 복잡하긴 하지만, 키 분실 위협이라든지 보안 관점에서는 많은 부분이 개선되었지만 여전히 만족스럽지 못한 부분이 있다. (물론, 이 정도면 꽤나 훌륭하다.)

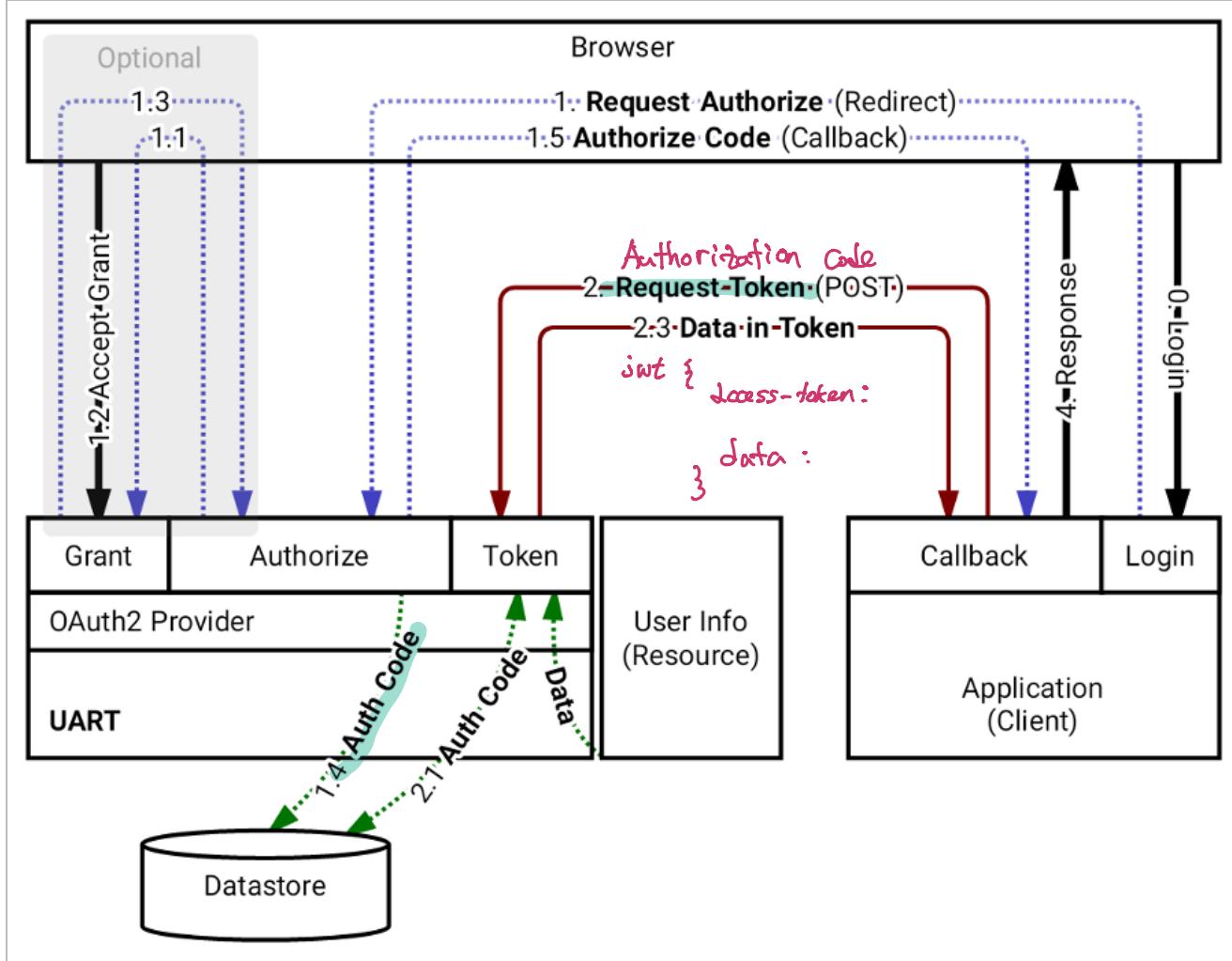
JWT는 그걸 OAuth2와 함께 쓰는가?

## 문자 쓰는 ~~JWT, JSON Web Token~~

그러다가 등장한 것이 JWT, JSON Web Token이다. 이 규약은, 인증 흐름의 규약이 아닌 Token 작성에 대한 규약이다. 이미 말한 바와 같이, 기본적으로 Access Token은 의미가 없는 문자열로 이루어져 있으며, Token의 진위나 유효성을 매번 확인해야 하는 것임에 반하여, 이 ~~JWT는 Token 그 안에 위조여부 확인을 위한 값, 유효성 검증을 위한 값, 심지어는 인증정보 자체를 담아서 제공함으로써, Token 확인 단계를 인증서버에게 묻지 않고도 할 수 있도록 만든 것이다.~~ (이 글에서는 JWT의 3단 구조와 의미, 그 데이터의 세부적인 부분은 생략하고, OAuth2의 어떤 부분이 개선된 것인지만 설명한다.)

아래, JWT를 이용한 흐름을 그려봤는데, 뭐가 다른가? 이미 Token 안에 인증정보, 사용자 정보를 넣어버렸기 때문에, 그리고 그와 함께 Token 발급자 정보와 서명, 유효기간 등의 모든 관련 정보를 담고 있기 때문에, 아예 3번 대의 흐름이 그림에서 완전히 사라졌다. 아무튼, 다시 한 번 말하지만, 중요한 것은 ~~일단 Token을 받으면 다시 “이 신분증이 맞나요?” 하며 확인 과정을 거칠 필요가 없다는 점이다.~~

App Server 와 Auth Server 추가 통신 불필요

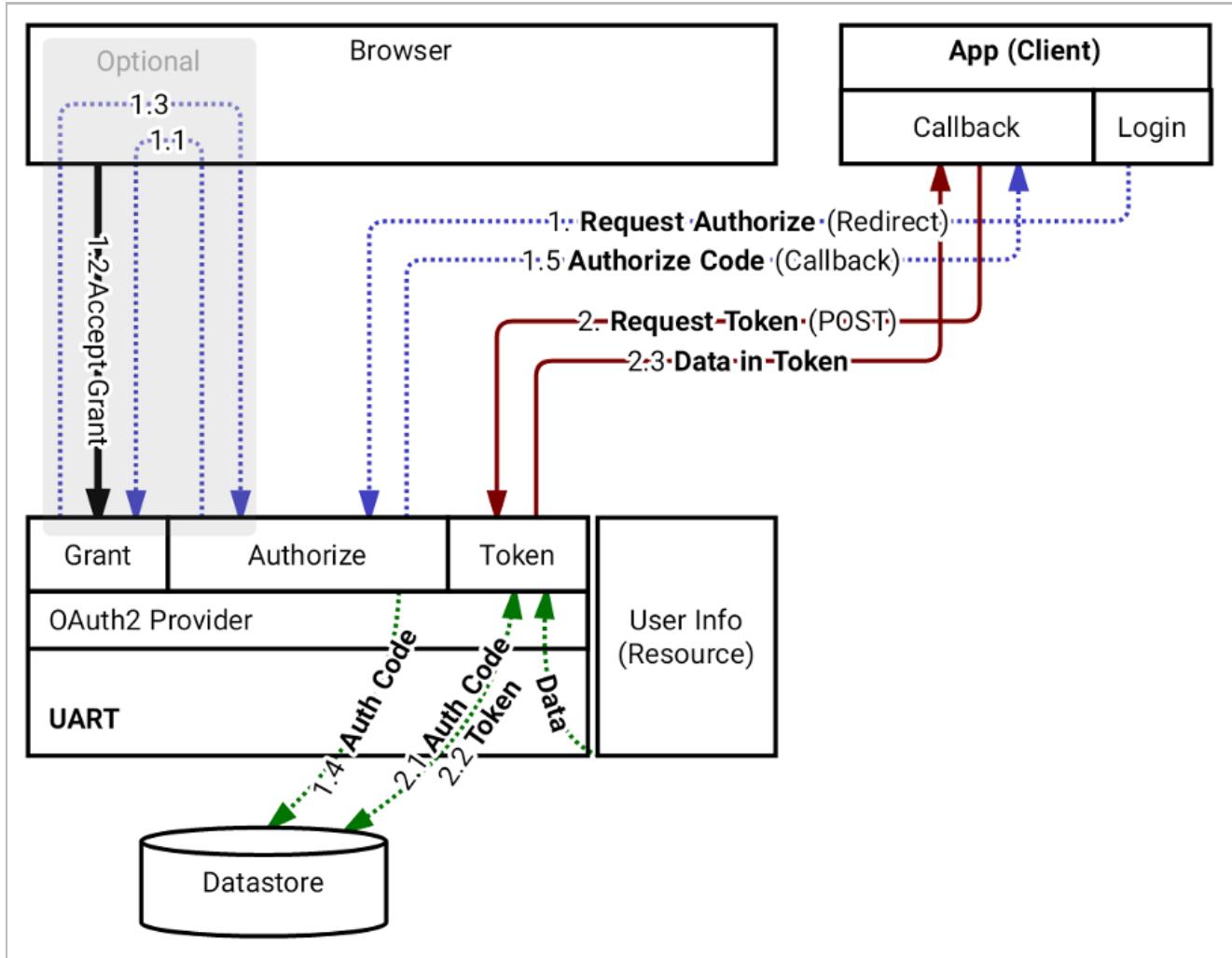


이렇게, 서버에 직접 연결하여 인증을 확인하지 않아도 되게 되면 장점이 많다. 만약, Access Token 검사를 위해 인증서버 저장소에 접근해야 한다면, 그 부하도 부하지만 Latency 발생도 피할 수 없고, 업무서버를 여러 대 두거나 지리적으로 멀리 두는 것이 조금 힘들어진다.

## 모바일 시대

인증이 간단해지니 이제 모바일 앱에서 이 인증을 적용하게 되면, 또는 Browser 기반 Framework을 사용한 Client Side App에서도 이 인증을 적용하여 비교적 간단하고 안전하게 인증/인가 처리를 할 수 있게 된다.

아래 그림은 다시 그린 그림이 아니라 위의 그림을 조금 밀고 당겨본 것이다. 이렇게, App의 위치가 사용자 쪽으로 이동해도 앞선 흐름이 전혀 깨지지 않고 큰 문제 없이 동작한다는 것을 직관적으로 느낄 수 있다.



Cloud Computing 시대를 살면서, 이 OAuth2는 제공자와 사용자 모두에게 매우 중요하다는 생각이 든다. 제공자의 관점에서 Cloud 를 사용하는 사용자에게 “서버가 가끔 죽기도 해요”, “Auto Scaling 해야죠”라고 말할 때, 문제가 되는 부분 중 하나가 바로 인증정보와 세션정보의 공유 또는 유지이다. 간단하지 않은 부분이다. 이것을 단지 사용자의 몫으로 넘긴다면 사용자로써는 플랫폼을 선택하면서 숙제를 떠안게 되는 격이므로, 사용자가 그것을 쉽게 풀 수 있는 길을 제공하는 것이 중요하다 할 수 있다.

또한, 사용자의 관점에서도, 만약 사용자가 중급 이상 규모의 서비스를 제공한다고 한다면, 그것을 하나의 통으로 개발/관리하는 것은 효율적일 수 없다. 서비스의 연속성을 지키면서 동시에 유지보수성이나 즉응성을 높이려면 서비스의 분산 처리 및 유기적인 연계 관리가 필수적인데, 이를 위한 첫 번째 단계가 인증과 세션정보에 대한 유연한 관리일 것 같다.

느슨한 인증 통합을 제공하는 OAuth2와 JWT, 그리고 그와 관련된 기술과 사상은 이런 양쪽의 입장에서 매우 중요한 요소가 아닐 수 없다.

글이 길어지고 힘드니, 얼른 대충 맺는다. ㅠ.ㅠ

--  
*"And in the end, the love you take is equal to the love you make." -- the Beatles*



클라우드컴퓨팅 UART 인증 OAuth2 JWT Single Sign On API

묶음글, 관련된 글, 잊지 마세요~!

잘못된 부분의 제보나 의견, 요청 있으시면 댓글! 환영합니다.

소용환의 생각저장소의 다른 댓글.

<p><b>블로그, Tistory로부터 Github Pages로 이주</b></p> <p>6 years ago • 댓글 2건</p> <p>얼마 전에 Ember.js와 Semantic-UI를 사용한 프로젝트를 진행하고 나서, 발표 ...</p>	<p><b>Docker: Getting Started with Docker</b></p> <p>3 years ago • 댓글 2건</p> <p>예상하지 못했던 여유가 생겨서, 미루고 또 미뤘던 주제를 하나 정리하려고 한다. 이미 많은 ...</p>	<p><b>Docker Swarm의 고가용성</b></p> <p>3 years ago • 댓글 7건</p> <p>Docker의 기본 Orchestration 도구인 Swarm의 기본적인 구성에 대하여 설명한 "Getting ...</p>	<p><b>Kibana Head 차원으로 펼쳐</b></p> <p>3 years ago • 1 comment</p> <p>"모니터링은 경! “에서 들었던 문제점 중 하나는</p>
---	--	--	---

# Basic steps

All applications follow a basic pattern when accessing a Google API using OAuth 2.0. At a high level, you follow five steps:

## 1. Obtain OAuth 2.0 credentials from the Google API Console.

Visit the [Google API Console](#) to obtain OAuth 2.0 credentials such as a client ID and client secret that are known to both Google and your application. The set of values varies based on what type of application you are building. For example, a JavaScript application does not require a secret, but a web server application does.

## 2. Obtain an access token from the Google Authorization Server.

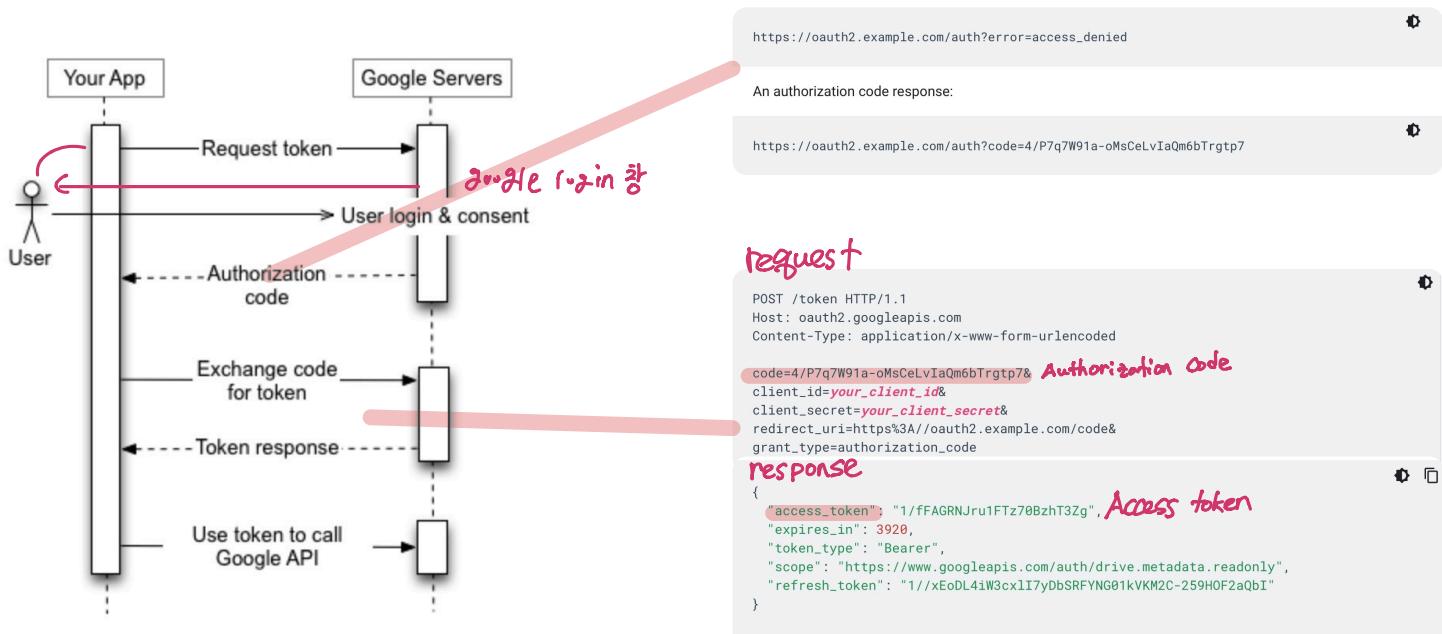
Before your application can access private data using a Google API, it must obtain an access token that grants access to that API. A single access token can grant varying degrees of access to multiple APIs. A variable parameter called scope controls the set of resources and operations that an access token permits. During the access-token request, your application sends one or more values in the scope parameter.

There are several ways to make this request, and they vary based on the type of application you are building. For example, a JavaScript application might request an access token using a browser redirect to Google, while an application installed on a device that has no browser uses web service requests.

Some requests require an authentication step where the user logs in with their Google account. After logging in, the user is asked whether they are willing to grant one or more permissions that your application is requesting. This process is called user consent.

If the user grants at least one permission, the Google Authorization Server sends your application an access token (or an authorization code that your application can use to obtain an access token) and a list of scopes of access granted by that token. If the user does not grant the permission, the server returns an error.

It is generally a best practice to request scopes incrementally, at the time access is required, rather than up front. For example, an app that wants to support saving an event to a calendar should not request Google Calendar access until the user presses the "Add to Calendar" button; see [Incremental authorization](#).



### 3. Examine scopes of access granted by the user.

Compare the scopes included in the access token response to the scopes required to access features and functionality of your application dependent upon access to a related Google API. Disable any features of your app unable to function without access to the related API.

The scope included in your request may not match the scope included in your response, even if the user granted all requested scopes. Refer to the documentation for each Google API for the scopes required for access. An API may map multiple scope string values to a single scope of access, returning the same scope string for all values allowed in the request. Example: the Google People API may return a scope of `https://www.googleapis.com/auth/contacts` when an app requested a user authorize a scope of `https://www.google.com/m8/feeds/`; the Google People API method `people.updateContact` requires a granted scope of `https://www.googleapis.com/auth/contacts`.

### 4. Send the access token to an API.

After an application obtains an access token, it sends the token to a Google API in an [HTTP Authorization request header](#). It is possible to send tokens as URI query-string parameters, but we don't recommend it, because URI parameters can end up in log files that are not completely secure. Also, it is good REST practice to avoid creating unnecessary URI parameter names. Note that the query-string support will be deprecated on June 1st, 2021.

Access tokens are valid only for the set of operations and resources described in the `scope` of the token request. For example, if an access token is issued for the Google Calendar API, it does not grant access to the Google Contacts API. You can, however, send that access token to the Google Calendar API multiple times for similar operations.

### 5. Refresh the access token, if necessary.

Access tokens have limited lifetimes. If your application needs access to a Google API beyond the lifetime of a single access token, it can obtain a refresh token. A refresh token allows [your application to obtain new access tokens](#).

★ **Note:** Save refresh tokens in secure long-term storage and continue to use them as long as they remain valid. Limits apply to the number of refresh tokens that are issued per client-user combination, and per user across all clients, and these limits are different. If your application requests enough refresh tokens to go over one of the limits, [older refresh tokens stop working](#).

## 구글 jwt 2lo방식

<https://developers.google.com/identity/protocols/oauth2/service-account#creatinganaccount>