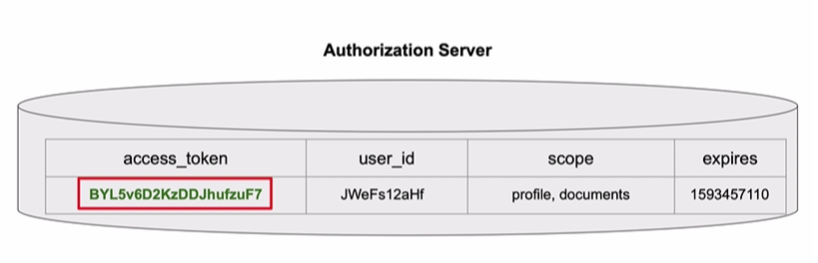
1. **Identifier Type Access Token**

(Preferable in our case as we are already generating the session id and we will generate the identity Token along with Session ID to make it more secure)

One Way is we can use the same Session ID but along with that we can add a table on Authorization Server ,

Where Table is like having details of user\_id , scope and expires...

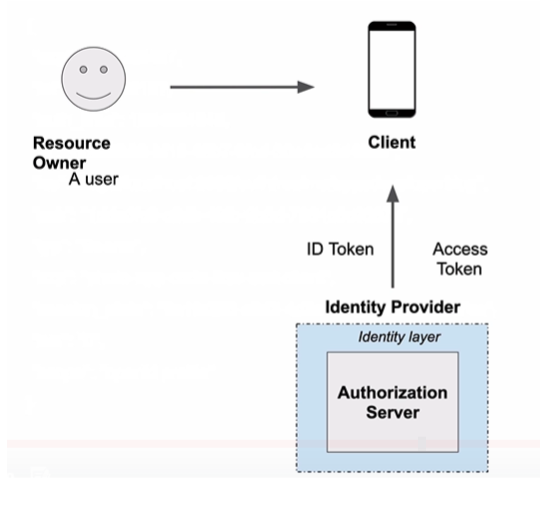
Also we can get integrate the ehcache to get the details between our Auth server and the client.



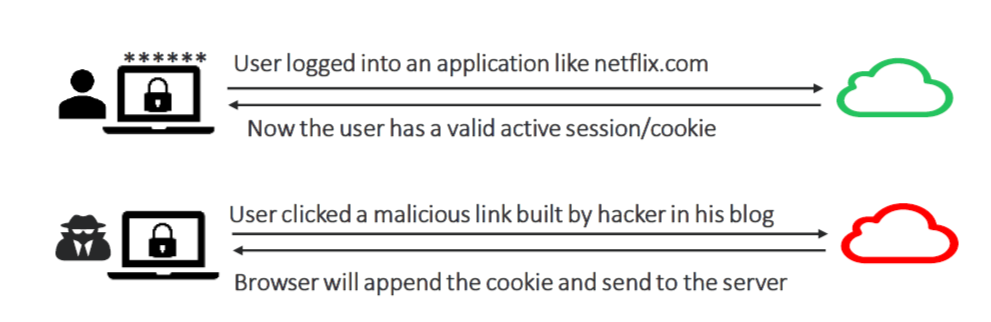
So the Client application will have now 2 tokens

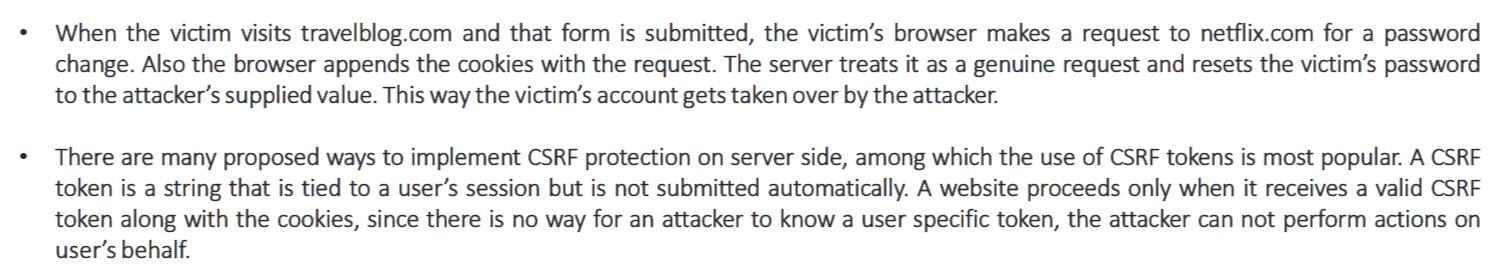
1. ID Token – to get the basic Details

2. Access Token – to get the Authorization Details



Limitation with only using Session is there may be a CSRF (Cross Site Request Forgery) attacks

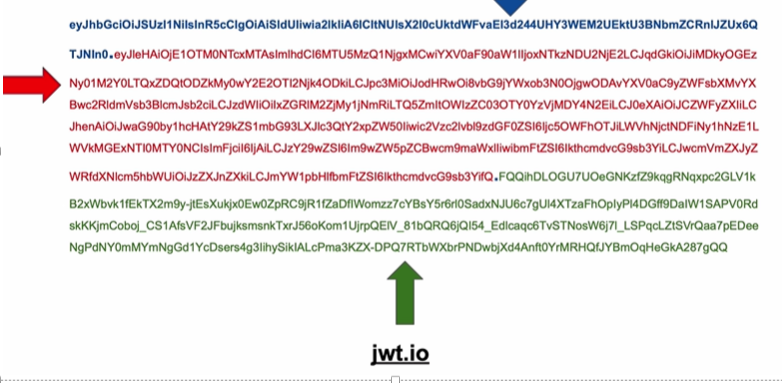




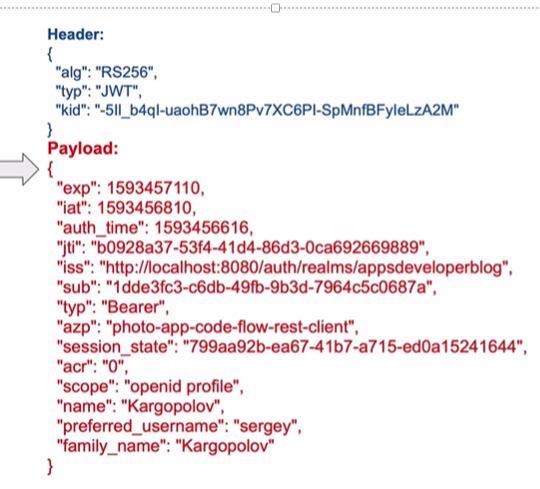
We can address this by using a xsrf token like along with jsession id



2. **Second way is to use self contained type**



Where we have Header , payload and Signatures for more security.



**1) Suppose a session expiry time is 12 hours. A user logged in at 8 am. His roles changed at 11 am. This should be detected and he should be immediately logged off. We should not wait until session expiry i.e 8pm in this case. So Can we run a background thread which does this?**

**Solution 1:**

Common Solution for this is to reduce the duration of the JWT and revoke the refresh token so that the user can’t generate a new JWT token..

May be 5-10 minutes max.

**Solution 2:**

To revoke a JWT is by leveraging a distributed **event system** that notifies services when refresh tokens have been revoked.

The identity provider broadcasts an event when a refresh token is revoked and other backends/services listen for the event.

When an event is received the backends/services update a local cache that maintains a set of users whose refresh tokens have been revoked.

**Solution 3:**

We can use any db or in-memory where,

**Step 1:** As soon as the token is generated for the first time for a user, store it in a db with the token and it's "**issuedAt()**" time.

I stored it in DB in this JSON format,

Ex:

{

"username" : "username",

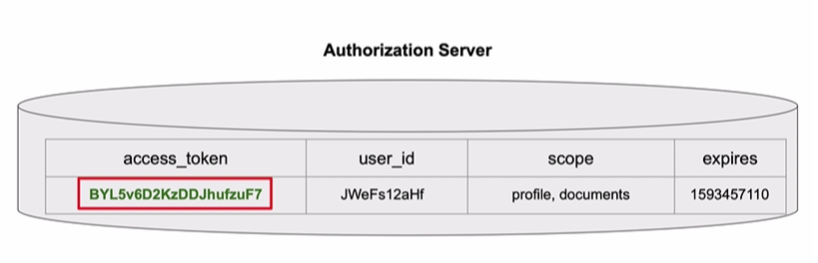
"token" : "token",

"issuedAt" : "issuedAt"

}

Step 2**:** Once you get a web service request for the same user with a token to validate, fetch "issuedAt()" timestamp from the token and compare it with stored(DB/in-memory) issued timestamp.

**Step 3**: If stored issued timestamp is new (using after()/before() method) then return that the token is invalid (in this case we are not actually expiring the token but we are stop giving access on that token).



**2) How to detect a client is calling our Auth Server or is it some malicious attacker trying to call? Example: A user usually logs on the client and client will call Auth Server right. But there is a chance some attacker calls Auth Server directly. So How can we detect this and solve?**

**1.** If we are using the self contained type JWT token then we can add some sort of signature used by both client and server in agreement to verify the authenticity of the payload.

**Example**: Signature for JWT token is

HS256(Base64(header) + “.” + Base64(payload), “A secret API example”) = TseARzVBAtDbU8f3TEiRgsUoKYaW2SbhCWB0QlKpdp8

2. If we are using identity type access token then no need as we are storing the access token in the database.. and we have only token which client and server has only..

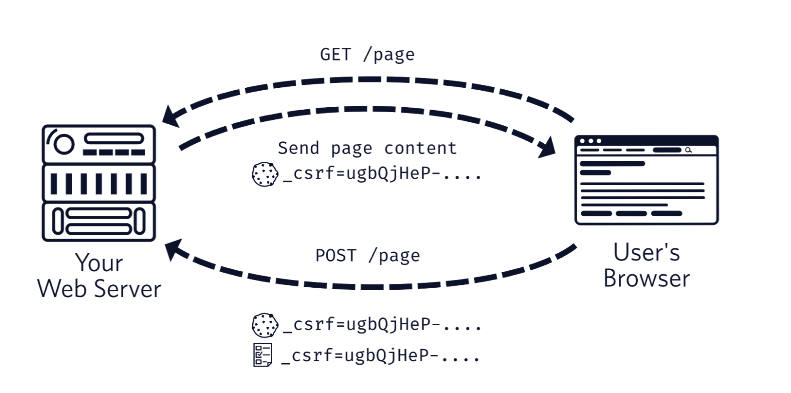
To make it even more secure we can use any type of signature as well like

HS256(Base64(accesstoken) , “A secret Key“) = TseARzVBAtDbU8f3TEiRgsUoKYaW2SbhCWB0QlKpdp8

**3) How to solve CSRF(Cross Site Request Forgery)?**

We can protect ourselves from this attack by using **CSRF token**. The concept is that when the browser gets a page from the server, it sends a randomly generated string as CSRF token as a cookie.

 Later, when your page performs a POST request it will send the CSRF token as a cookie and also in another way such as a parameter in the body or via an HTTP header like X-CSRF-Token.



An attacker will not be able to reproduce the same behavior with their hidden form since they won’t be able to access the cookie to retrieve the value and send it along with their malicious POST request.

As an attacker would serve its malicious script from another origin, his script is not allowed to read data contained in another origin (I.e. the token).

Using XSRF

XSRF are saving in cookies and should set Same-Site Policy as Lax or Strick to make browser not giving it to another script.

<https://stackoverflow.com/questions/39536888/steal-csrf-token>