Understanding OAuth2 token authentication

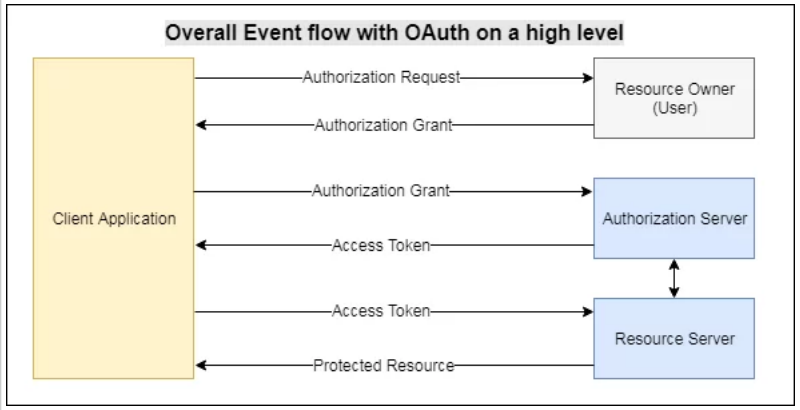
1. Introduction

In this tutorial, we will be understanding **OAuth2 Token Authentication**, such that only authenticated users and applications get a valid access token which can be subsequently used to access authorized APIs (which are nothing but the protected resources in OAuth terms) on the server.

With token based authentication, the users/applications get access to the protected resources for a certain period of time by sharing a valid access token for every interaction with the server.

2. Events involved in Token Authentication

With Token Authentication, the events involved have been clearly depicted in the diagram below –



1. Client application first requests for the authorization grant from the user (resource owner), as we often see an authorization pop-up to authorize or deny access to the data some other application. For example, Goibibo asks for to access friends from your Facebook account.
2. Once the user authorizes by clicking *'Authorize'* on the pop-up window, the client application (Goibibo) receives the authorization grant.
3. The client application (Goibibo) then requests Authorization Server (Facebook) for the access token along with its own identity and the authorization grant it received in the previous step.
4. If the client application is authenticated and the authorization grant is found valid, the Authorization Server (by Facebook) provides/issues an access token to the client application (Goibibo).
5. The client application (Goibibo) then accesses the protected resources (friends from the facebook application) by subsequently passing the access token to the Resource Server (by Facebook), until the token expires after the specified period of time.

3. OAuth2 Roles

Listed below are the delegated roles in OAuth implementation –

1. 1. **resource owner** – Entity capable of granting access to the protected resource. If you are logged-in to Goibibo, and it wants to access friends from your Facebook account, you are the resource owner, who needs to provide an authorization grant.
   2. **resource server** – Server (Facebook) hosting the protected resources, that could accept and respond to protected resource requests with access tokens.
   ```
   <!-- Define protected resources hosted by the resource server -->
   <oauth:resource-server id="resourceServerFilter"
       resource-id="adminProfile" token-services-ref="tokenServices" />
   ```
   This means that friends list is one of the resources being hosted on the server (Facebook) to be exposed for access by other third party apps (Goibibo).
   3. **client** – An application (Goibibo) making protected resource requests on behalf of the resource owner (user) and with its authorization. We define all such authorization grants into the context as –
   ```
   <!-- Define protected resources hosted by the resource server -->
   <oauth:resource-server id="resourceServerFilter"
       resource-id="adminProfile" token-services-ref="tokenServices" />
   ```
   4. This means that an Gobibo as a client application can access the admin friends list on the resource server (Facebook).
   5. **authorization server** – Server that provides access token to the client application based on the client application's identity and Authorization Grant as received by the resource owner (User).

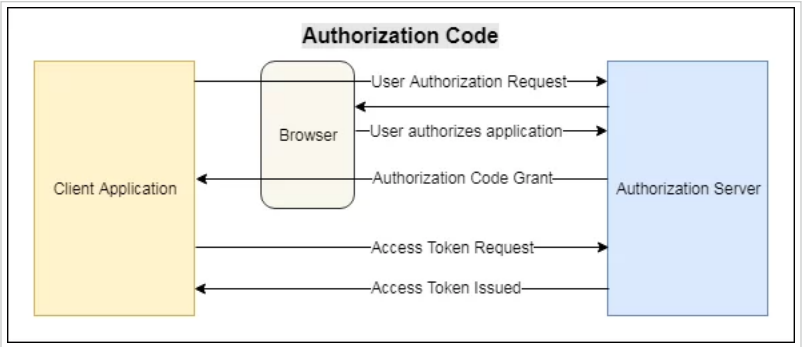
```
<!-- The server issuing access tokens to the client after successfully
authenticating the resource owner and obtaining authorization -->
<oauth:authorization-server
    client-details-service-ref="clientDetails" token-services-ref="tokenServices"
    user-approval-handler-ref="userApprovalHandler">
    <oauth:authorization-code />
    <oauth:implicit />
    <oauth:refresh-token />
    <oauth:client-credentials />
    <oauth:password />
</oauth:authorization-server>
```

4. Authorization Grants

OAuth2 provides four types of Authorization Grants –

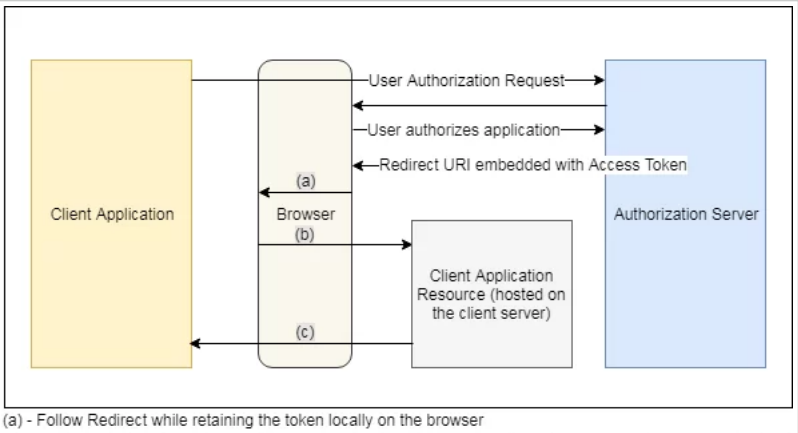
4.1 Authorization Code

- Used in server side applications, hence the privacy at both sides are maintained.
- Most common of all.
- Redirection-based flow.
- The User Authorization request routes through the User Agent (Web Browser), and the client application must be capable enough to interact with the User Agent and receive the Authorization Code from the Authorization server at the other end.



4.2 Implicit

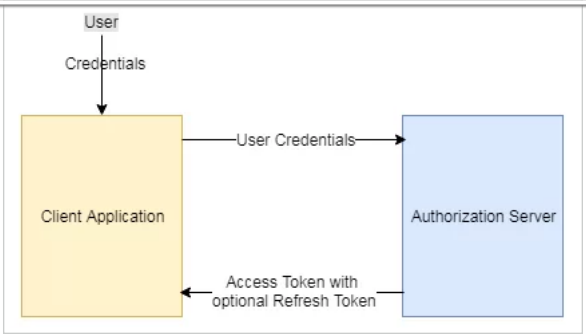
- Somewhat similar to Authorization Code Grant Type.
- More commonly used with for mobile applications and web applications.
- Privacy at risk at the client side, as the Authorization Code is stored locally with the User Agent before handing it over to the client application. This can be exposed to other applications on the user's device.
- Does not support Refresh Tokens .



(a) - Follow Redirect while retaining the token locally on the browser
(b) - The returned web page (hosted on the client server) has the **token extract script** that extracts the access token from the redirect URI retained previously
(c) - Passes the token to the client

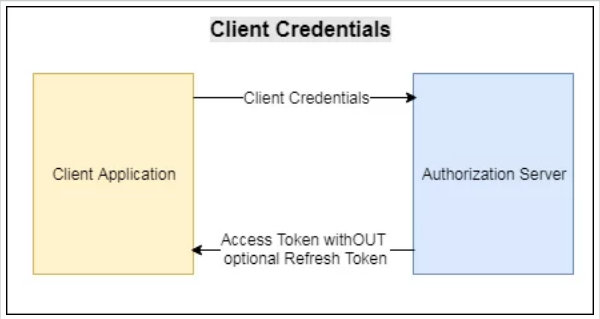
4.3 Resource Owner Password Credentials

- Used between trusted applications.
- The user (Resource Owner) shares the credentials directly with the client application, which requests the Authorization Server to return the access token after successfully authenticating the user credentials and further authorizing the user to access limited resources on the server.
- **This is the Authorization Grant Type we will be using for our demo application.**



4.4 Client Credentials

- Used when an application needs to access its own service account segment, by which it can fetch resources only under its control.
- So there may be other trusted confidential clients to the same application, who have their own individual service accounts of the application, and might be having having different protected resources under their control.
- An application requests for the access token by sending its credentials, client ID and client secret, to the authorization server.



5. Token Store

Another important component involved is the Token Store, that defines how the generated tokens need to be stored. The default store is an **in-memory implementation**, but there are some other implementations also available, which can be used as per the requirement –

1. **InMemoryTokenStore** – Token is stored in the server memory, hence there is a risk of losing the tokens on authorization server restart.
2. **JwtTokenStore** – All authorization and access grant data is encoded into the token itself, and such tokens are not persisted anywhere. Such tokens are validated on-the-fly using the decoder and has a dependency on *JwtAccessTokenConverter*.
3. **JdbcTokenStore** – The token data is store into the relational database. With this token store, we are safe in case of Authorization server restart. The tokens can also be easily shared among the servers and can be revoked. Note that to use the JdbcTokenStore we will be needing "spring-jdbc" dependency in the classpath.

```
<!-- This defines the token store. We have currently used in-memory token
    store but we can instead use a user defined one -->
<bean id="tokenStore"
    class="org.springframework.security.oauth2.provider.token.InMemoryTokenStore" />
<!-- If need to store tokens in DB
<bean id="tokenStore"
    class="org.springframework.security.oauth2.provider.token.store.JdbcTokenStore">
    <constructor-arg ref="jdbcTemplate" />
</bean> -->
```

6. Refresh Token Flow

Once the access token expires after the designed period of time (119 as per below example), we can regenerate a new valid access token using the **refresh token**. Refresh token comes along when the original access token is issued, as we can see below –

```
{
   "access_token":"04f12761-501b-415b-8941-52bce532ce60",
   "token_type":"bearer",
   "refresh_token":"fc348b4f-de62-4523-b808-9935b1f2e46f",
   "expires_in":119
}
```