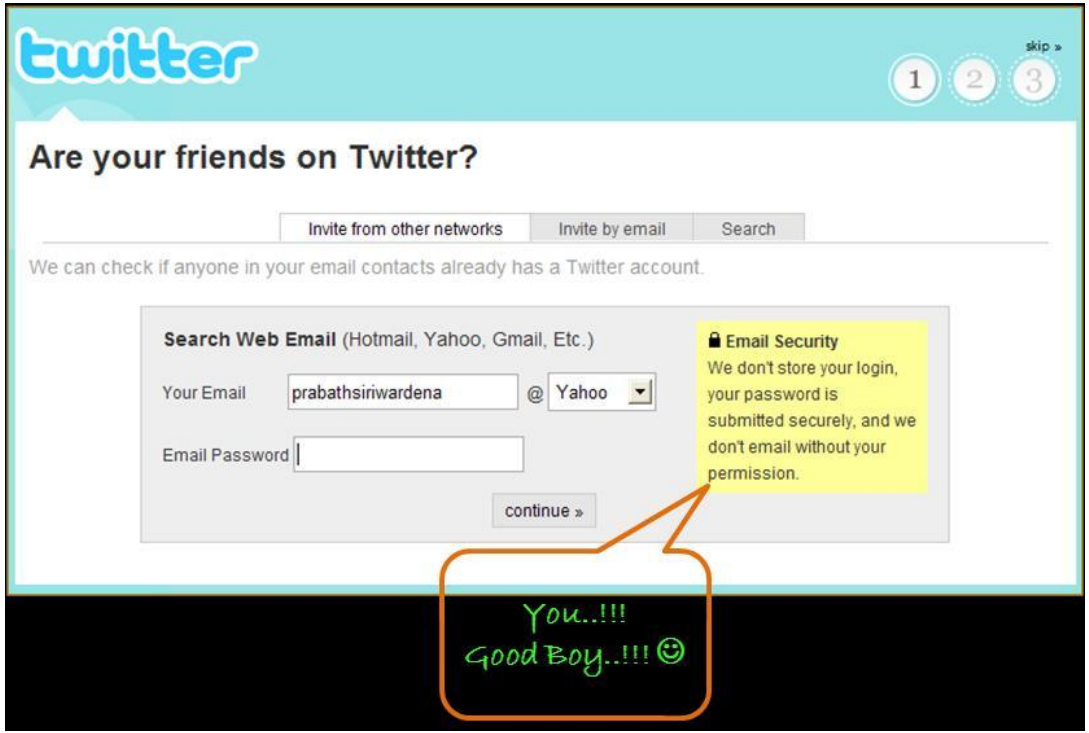
OAUTH 2.0 FOR WEB AND MOBILE APPLICATION DEVELOPERS

Prabath Siriwardena
Senior Director - Security Architecture
prabath@wso2.com | prabath@apache.org

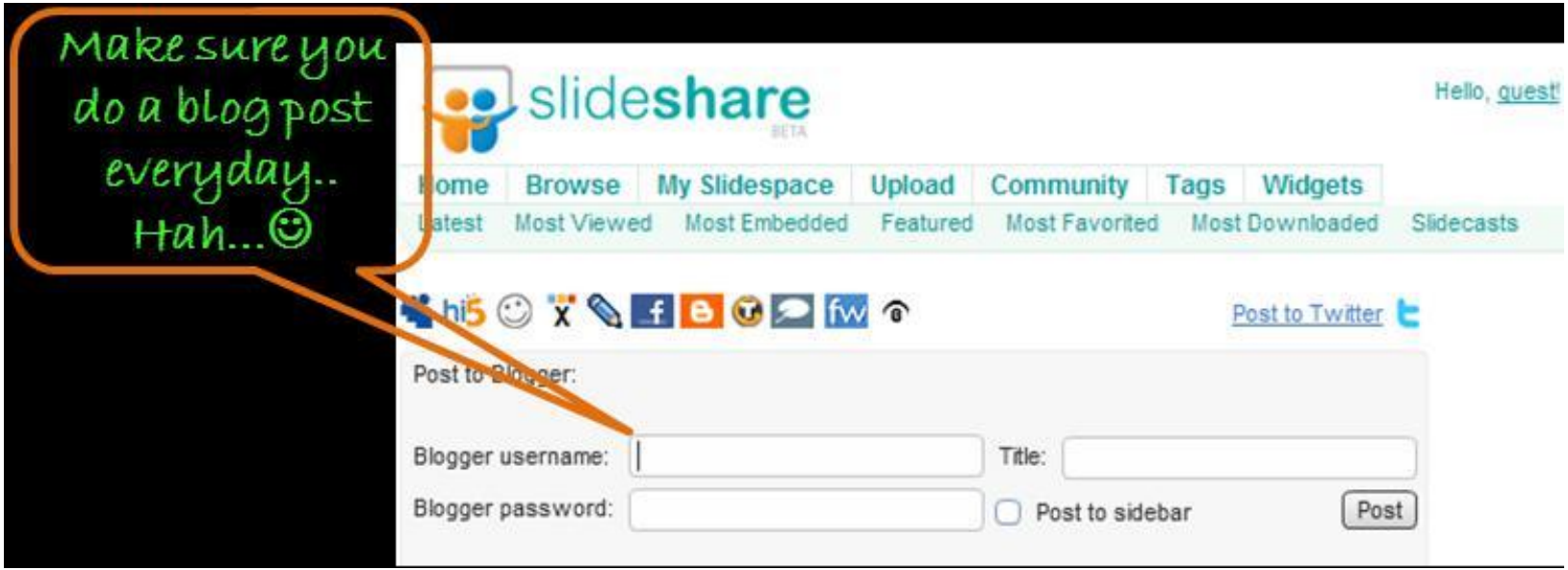
OAuth 2.0 & Access Delegation

- OAuth 2.0 provides a way of delegating access to a third party to access a resource on behalf of the delegator.
- Access delegation via credential sharing (pre-OAuth Era)
- Access delegation via no credential sharing (non-standardized)
- Access delegation via no credential sharing (standardized)

Pre-OAuth Era



Pre-OAuth Era



Pre-OAuth Era

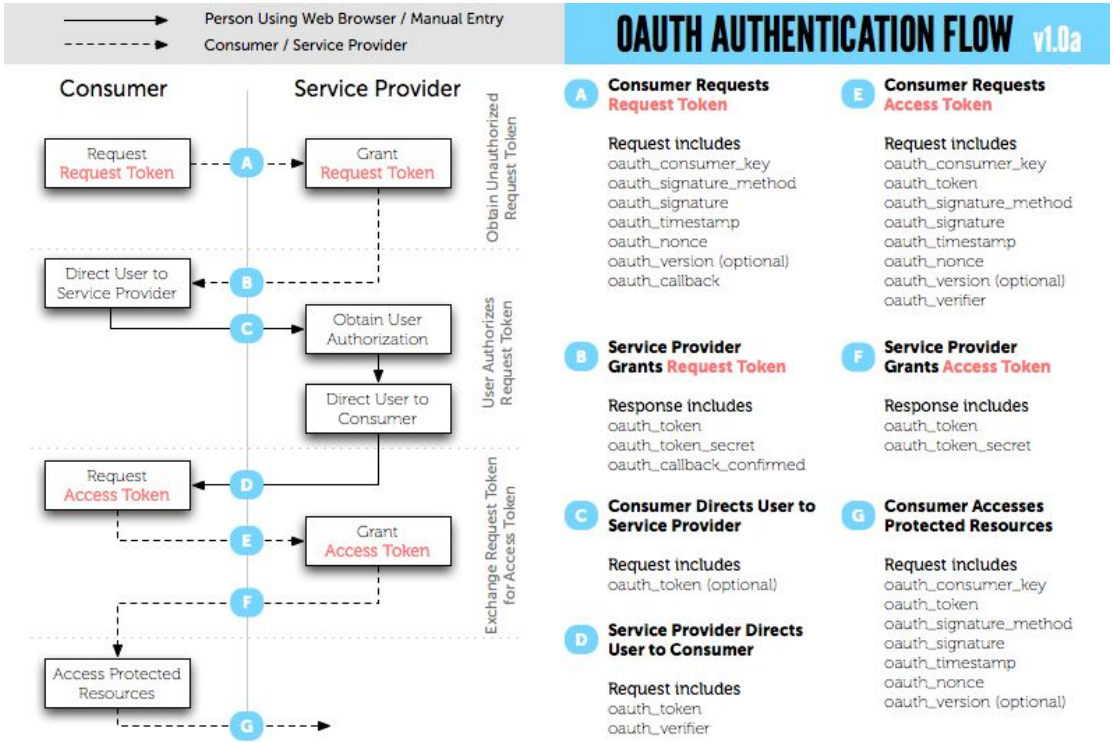
Pre-OAuth Era

Google calendar

Pre-OAuth Era

- Google ClientLogin / AuthSub
- Flickr Auth
- Yahoo BBAuth

OAuth 1.0


Person Using Web Browser / Manual Entry
Consumer / Service Provider
OAUTH AUTHENTICATION FLOW v1.0a
Consumer
Service Provider
Request Request Token
Grant Request Token
Obtain Unauthorized Request Token
Direct User to Service Provider
Obtain User Authorization
Direct User to Consumer
User Authorizes Request Token
Request Access Token
Grant Access Token
Exchange Request Token for Access Token
Access Protected Resources
A Consumer Requests Request Token
Request includes
oauth_consumer_key
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_callback
B Service Provider Grants Request Token
Response includes
oauth_token
oauth_token_secret
oauth_callback_confirmed
C Consumer Directs User to Service Provider
Request includes
oauth_token (optional)
D Service Provider Directs User to Consumer
Request includes
oauth_token
oauth_verifier
E Consumer Requests Access Token
Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_verifier
F Service Provider Grants Access Token
Response includes
oauth_token
oauth_token_secret
G Consumer Accesses Protected Resources
Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)

OAuth WRAP

- In Nov 2009, OAuth WRAP was introduced as a draft specification for access delegation, built on top of OAuth 1.0.
- WRAP was later deprecated in favour of OAuth 2.0.
- WRAP is not based on a signature scheme (like OAuth 1.0)
- In 2009, Facebook add OAuth WRAP support FriendFind.
- Introduced multiple profiles (autonomous client profiles and user delegation profiles).
- Client Account & Password / Assertion / Username & Password / Web App / Rich App profiles.

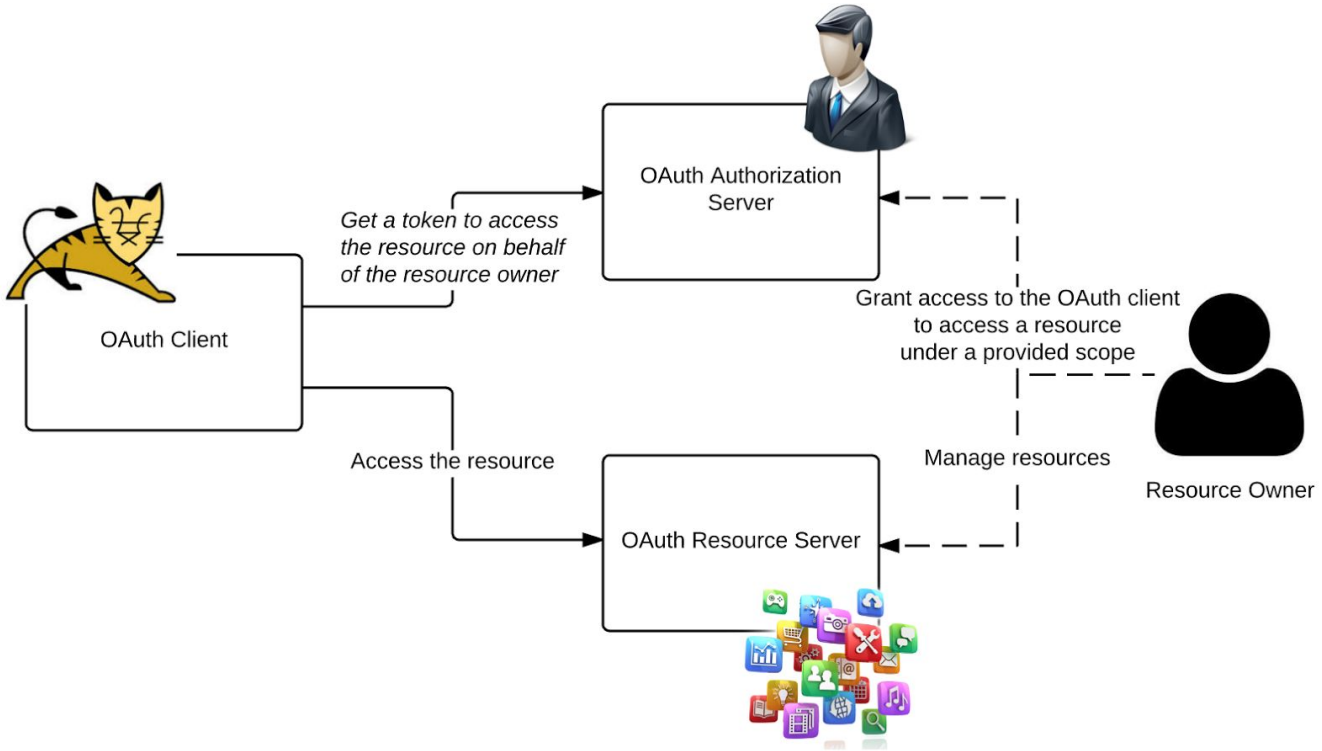
OAuth 2.0

- The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

OAuth 1.0 vs. OAuth 2.0

- OAuth 1.0 is a concrete protocol for access delegation, while OAuth 2.0 is an authorization framework.
- OAuth 1.0 is signature based (HMAC-SHA256, RSA-SHA256) - while OAuth 2.0 supports multiple token profiles.
- OAuth 1.0 is less extensible.
- OAuth 1.0 is less developer friendly.
- OAuth 1.0 requires TLS only for the initial handshake - but OAuth 2.0 requires TLS through the flow.

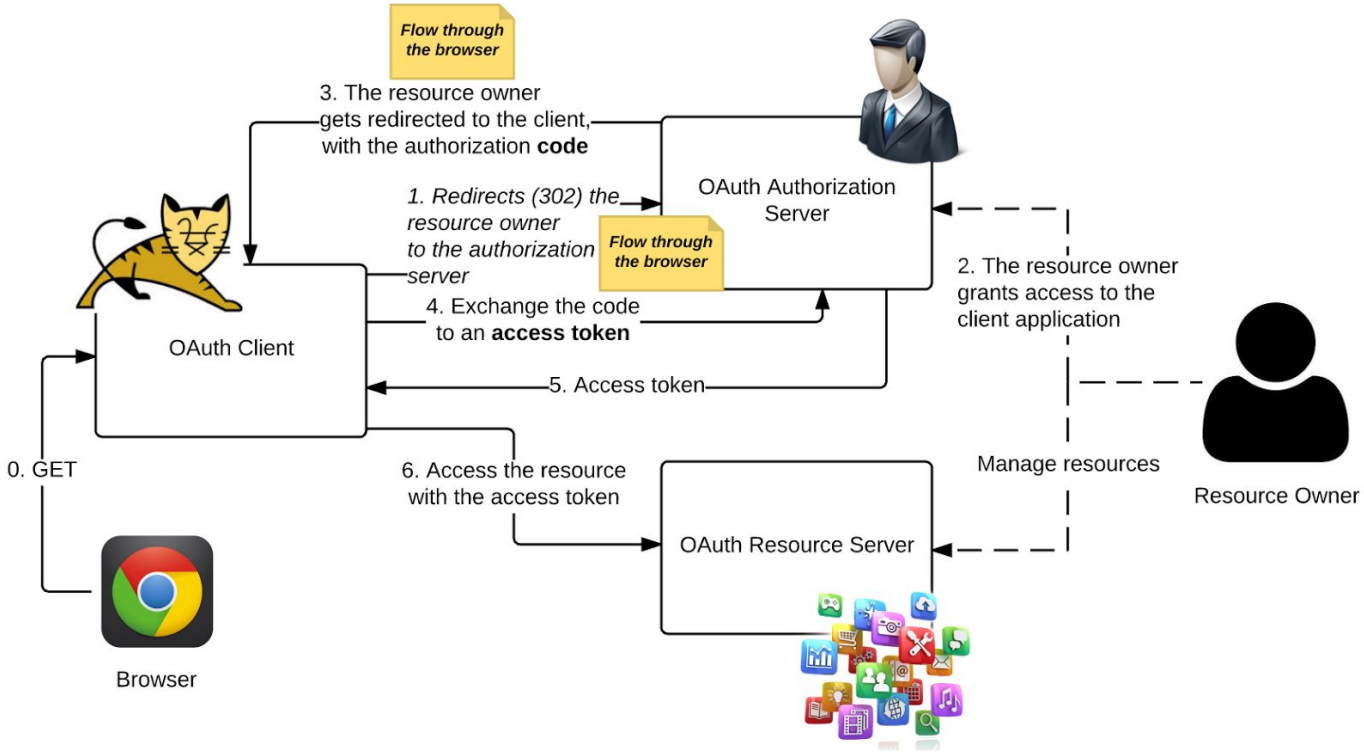
OAuth 2.0


OAuth Authorization Server
Get a token to access the resource on behalf of the resource owner
OAuth Client
Grant access to the OAuth client to access a resource under a provided scope
Resource Owner
Access the resource
Manage resources
OAuth Resource Server

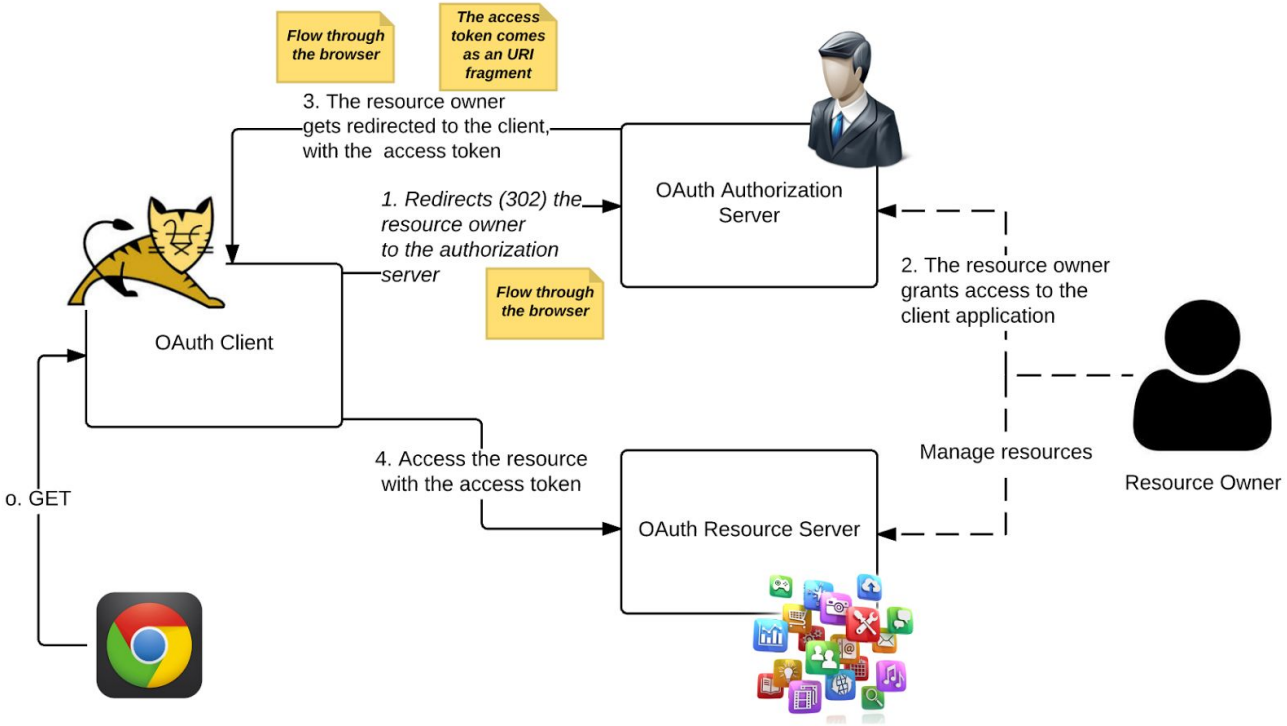
Grant Types

- A grant type defines how a client could obtain an authorization grant from the authorization server on behalf of the resource owner.
- A grant type is a powerful extension point in OAuth 2.0.

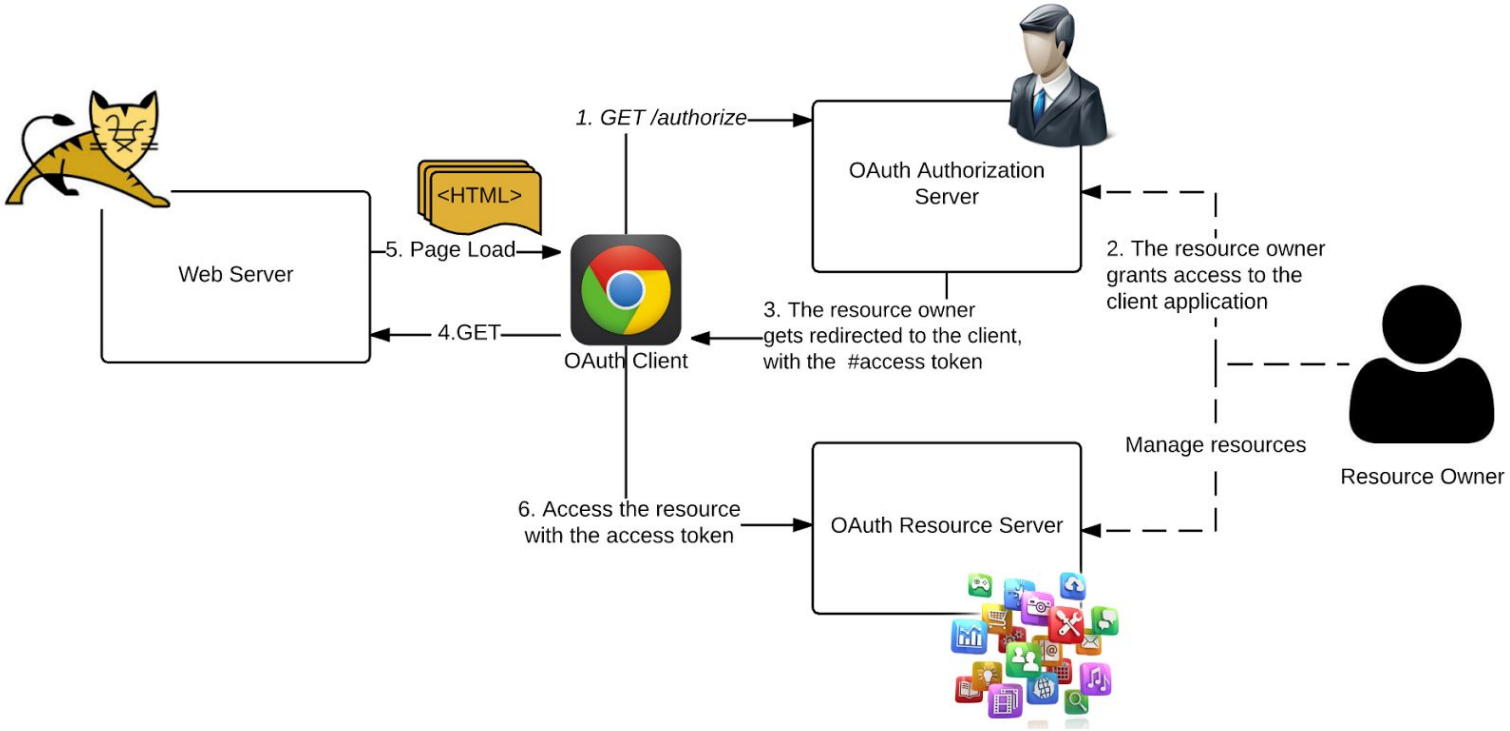
Authorization Code Grant Type



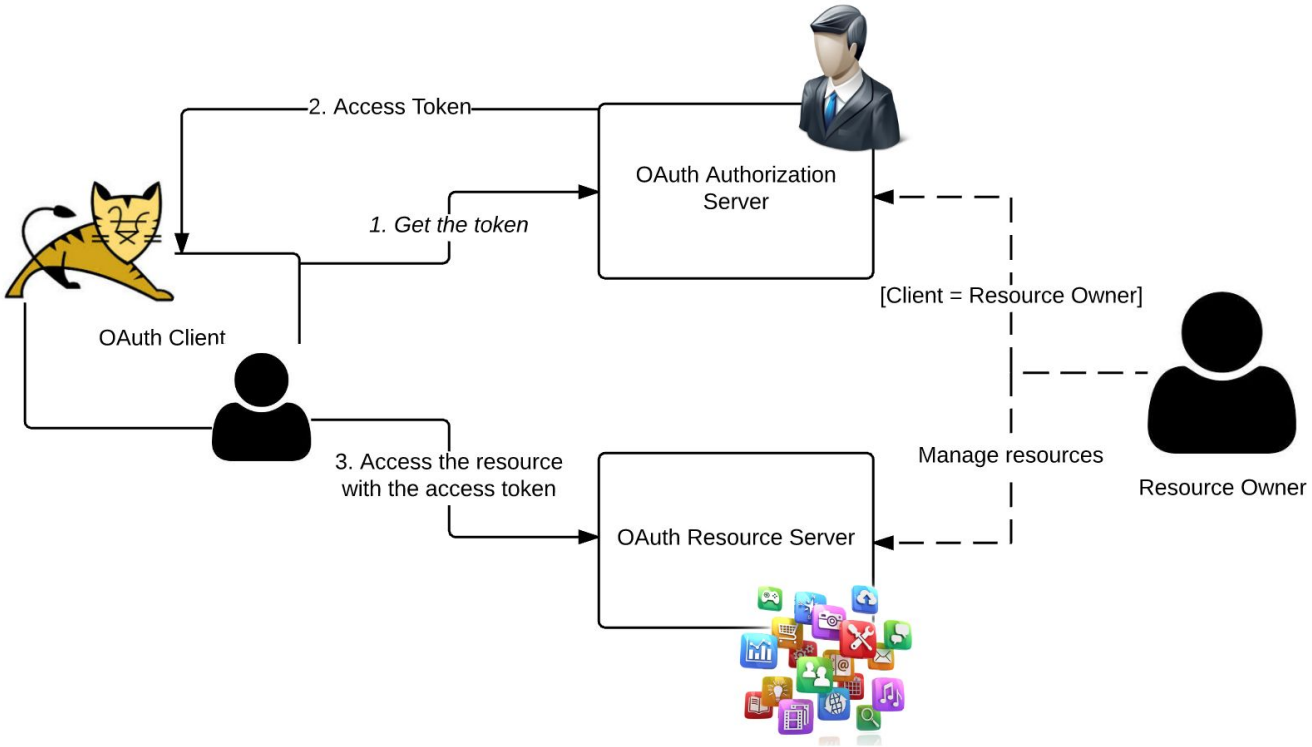
Implicit Grant Type (I)



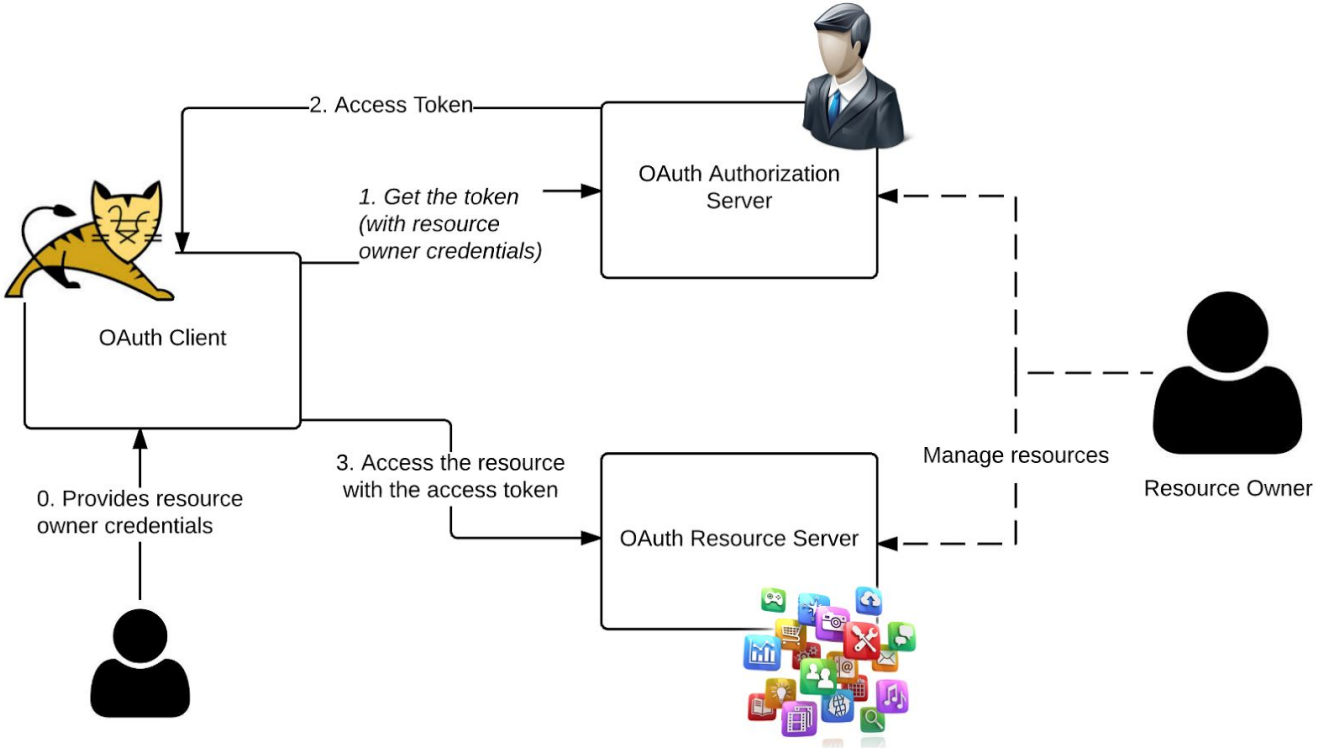
Implicit Grant Type (II)



Client Credentials Grant Type



Password Grant Type



Scope

- Defines the scope of the access token - what can be done with the access token.
- The authorization and token endpoints allow the client to specify the scope of the access request using the "scope" request parameter.
- In turn, the authorization server uses the "scope" response parameter to inform the client of the scope of the access token issued.
- The value of the scope parameter is expressed as a list of space- delimited, case-sensitive strings.

Token Types

- Neither OAuth 1.0 nor WRAP supported custom token types.
- OAuth 2.0 does not mandate any token type.
- OAuth 2.0 Bearer Token / OAuth 2.0 MAC Token
- Almost all the OAuth 2.0 deployments are based on Bearer token profile.

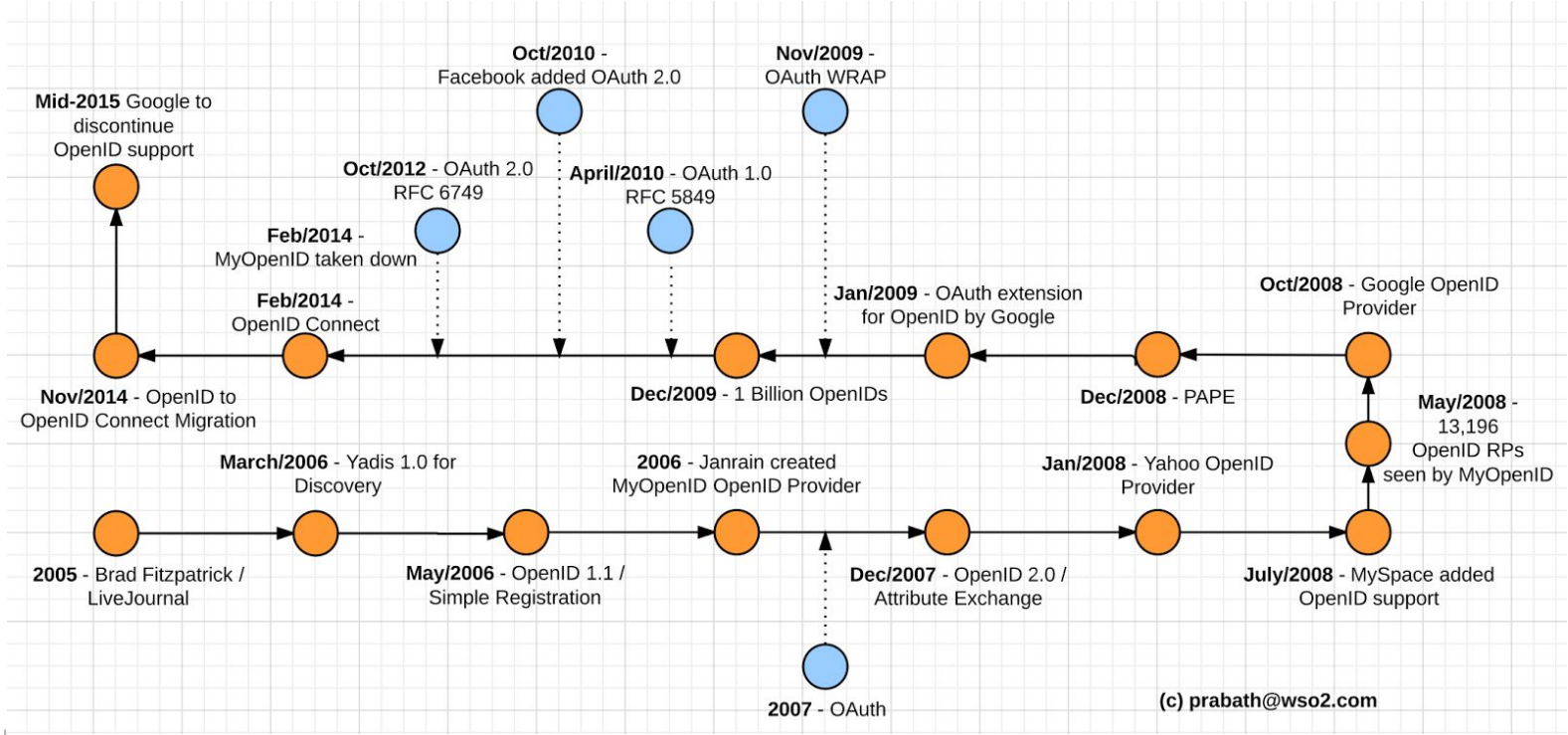
Client Types

- OAuth 2.0 identifies two types of clients: confidential clients and public clients.
- A confidential client is capable of protecting its own credentials while not a public client.
- Web app is a confidential client, while a native mobile app and an SPA are public clients.

OpenID Connect (OIDC)

- An identity layer on top of the OAuth 2.0.
- Enables clients to verify the identity of the end-user based on the authentication performed by an Authorization Server.
- Use to obtain basic profile information about the End-User in an interoperable and REST-like manner.

OpenID Connect (OIDC)



JWT (JSON Web Token)

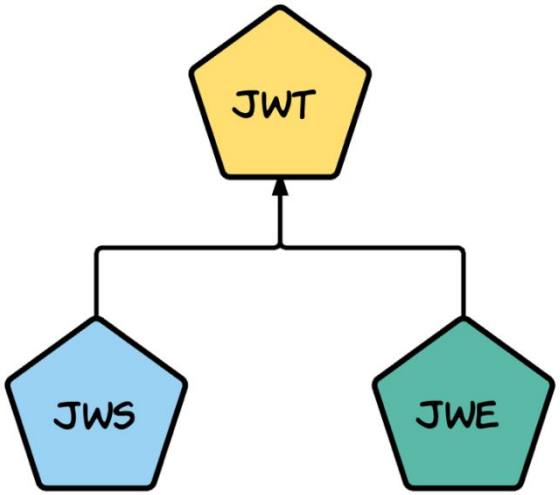
- Defines a container to transport data between interested parties.
- There are multiple applications of JWT - in OpenID Connect the id_token is represented as a JWT.
- Propagate one's identity between interested parties.
- Propagate user entitlements between interested parties.
- Transfer data securely between interested parties over a unsecured channel.
- Assert one's identity, given that the recipient of the JWT trusts the asserting party.

JWT (JSON Web Token)

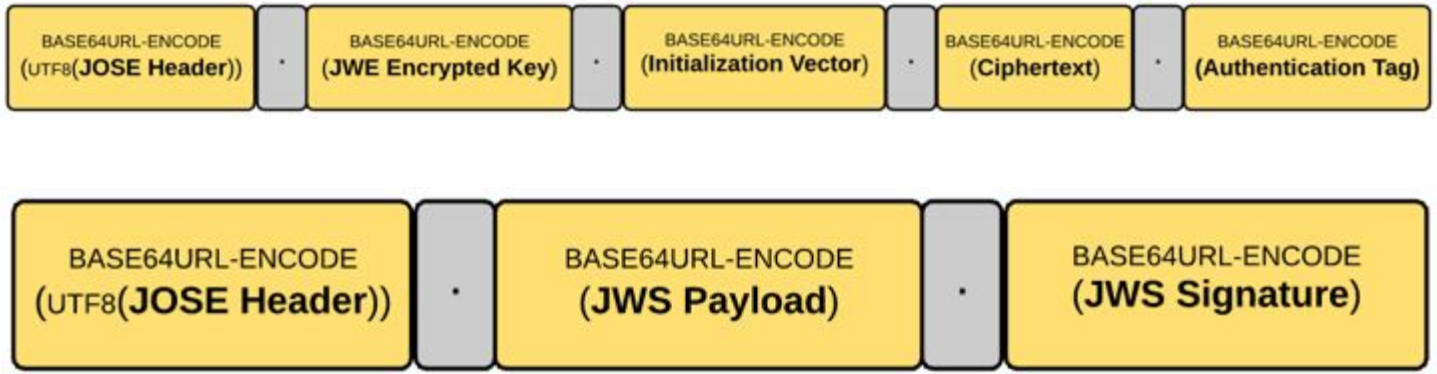
eyJhbGciOiJSUzI1NiIsImtpZCI6Ijc4YjRjZjIzNjU2ZGMzOTUzNjRmMWI2YzAyOTA3NjkxZjJjZGZmZTEifQ.eyJpc3MiOiJhY2NvdW50cy5nb29nbGUuY29tIiwic3ViIjoiMTEwNTAyMjUxMTU4OTIwMTQ3NzMyIiwiYXpwIjoiODI1MjQ5ODM1NjU5LXRlOHFnbDcwMWtnb25ub21ucDRzcXY3ZXJodTEyMTFzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiwiZW1haWwiOiJwcmFiYXRoQHdzbzIuY29tIiwiYXRfaGFzaCI6InpmODZ2TnVsc0xCOGdGYXFSd2R6WWciLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiYXVkIjoiODI1MjQ5ODM1NjU5LXRlOHFnbDcwMWtnb25ub21ucDRzcXY3ZXJodTEyMTFzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiwiaGQiOiJ3c28yLmNvbSIsImlhdCI6MTQwMTkwODI3MSwiZXhwIjoxNDAxOTEyMTcxfQ.**TVKv-pdyvk2gW8sGsCbsnkqsrS0T-H00xnY6ETklfglxfotvFn5IwKm3xyBMpy0FFe0Rb5Ht8AEJV6PdWyxz8rMgX2HROWqSo_RfEfUpBb4iOsq4W28KftW5H0IA44VmNZ6zU4YTqPSt4TPhyFC9fP2D_Hg7JQozpQRUfbWTJI**

JWT (JSON Web Token)

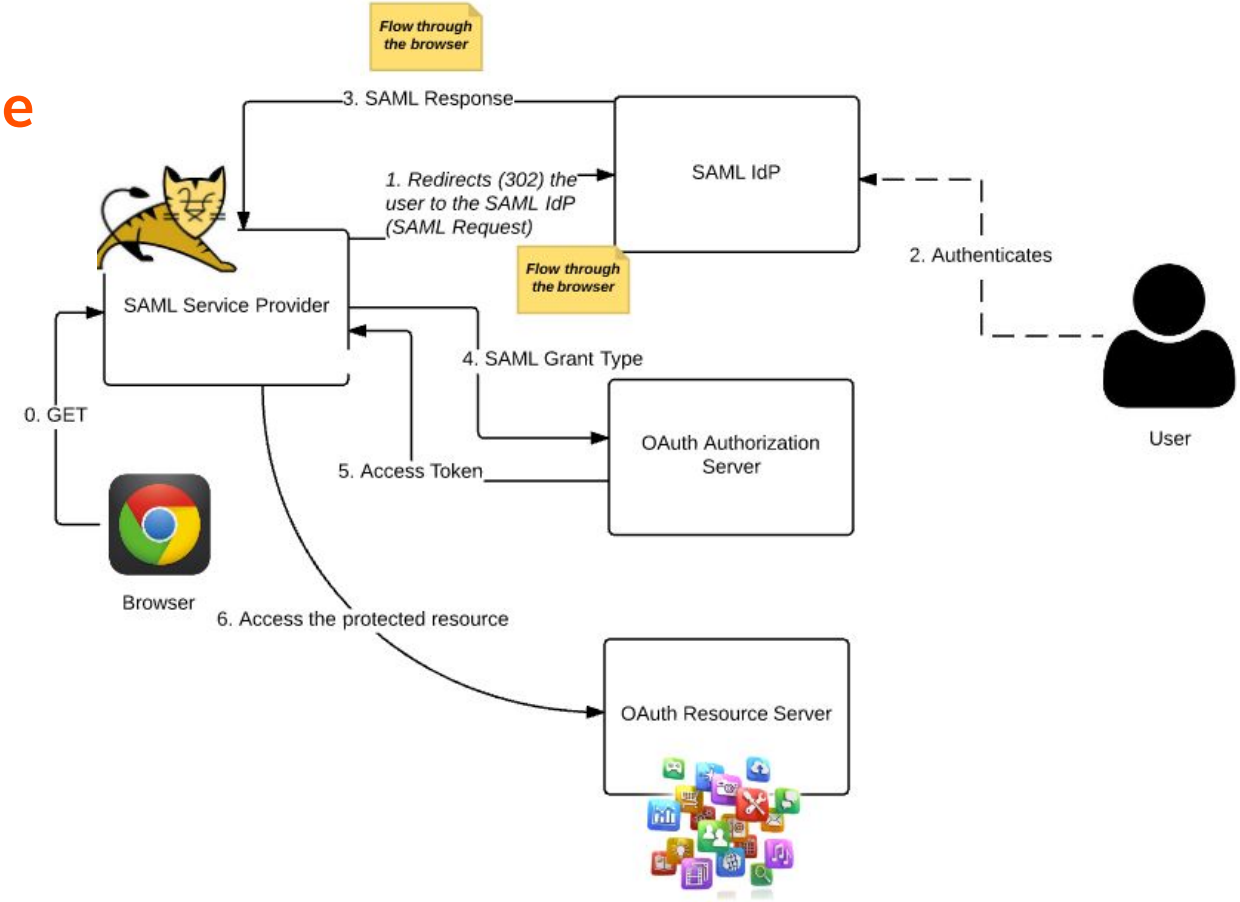
- A JWT does not exist itself — either it has to be a JWS or a JWE (JSON Web Encryption).
- It's like an abstract class — the JWS and JWE are the concrete implementations.
- We call a JWS or JWE, a JWT only if it follows the compact serialization.



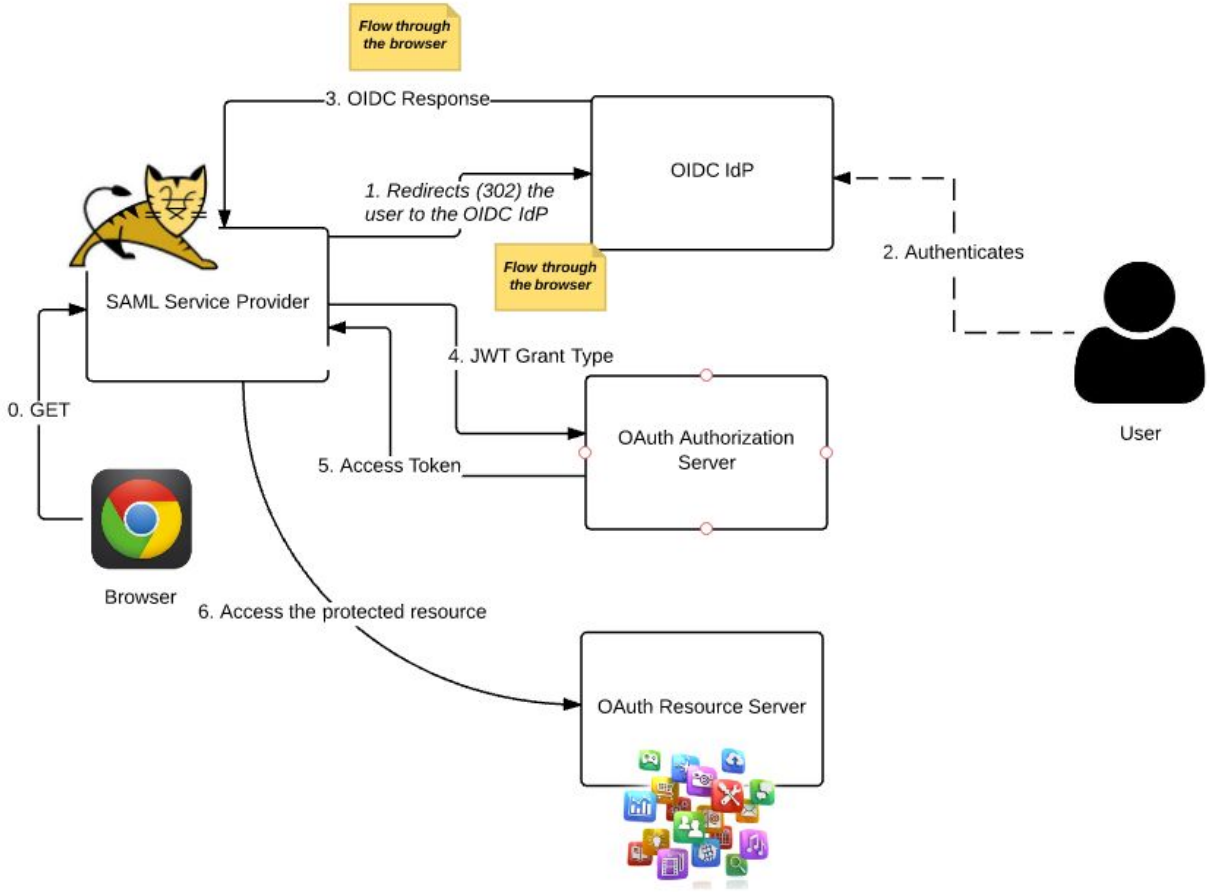
JWT (JSON Web Token)



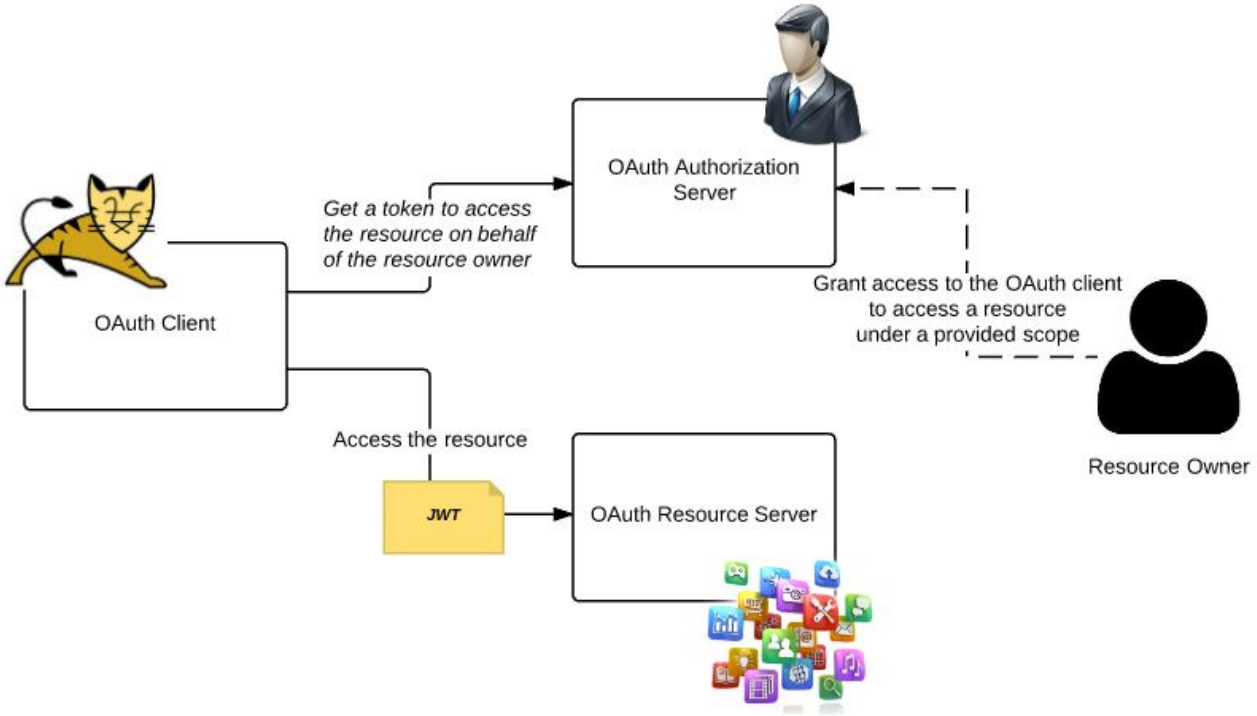
SAML Grant Type for OAuth 2.0



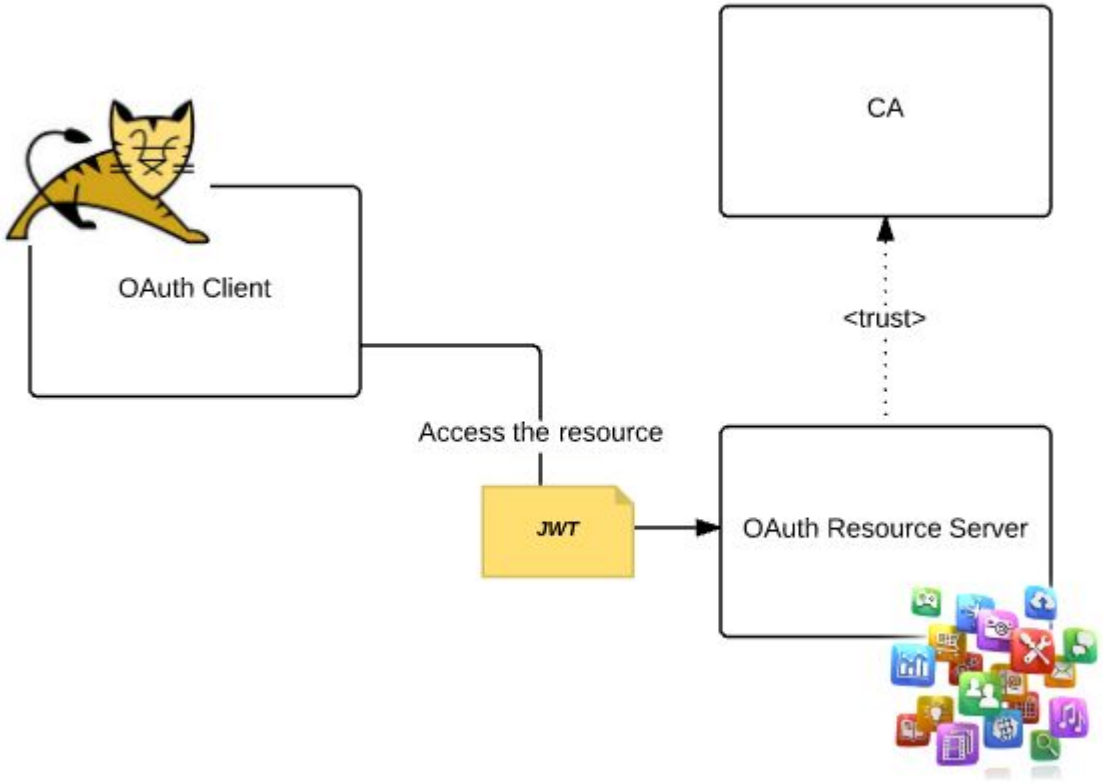
JWT Grant Type for OAuth 2.0



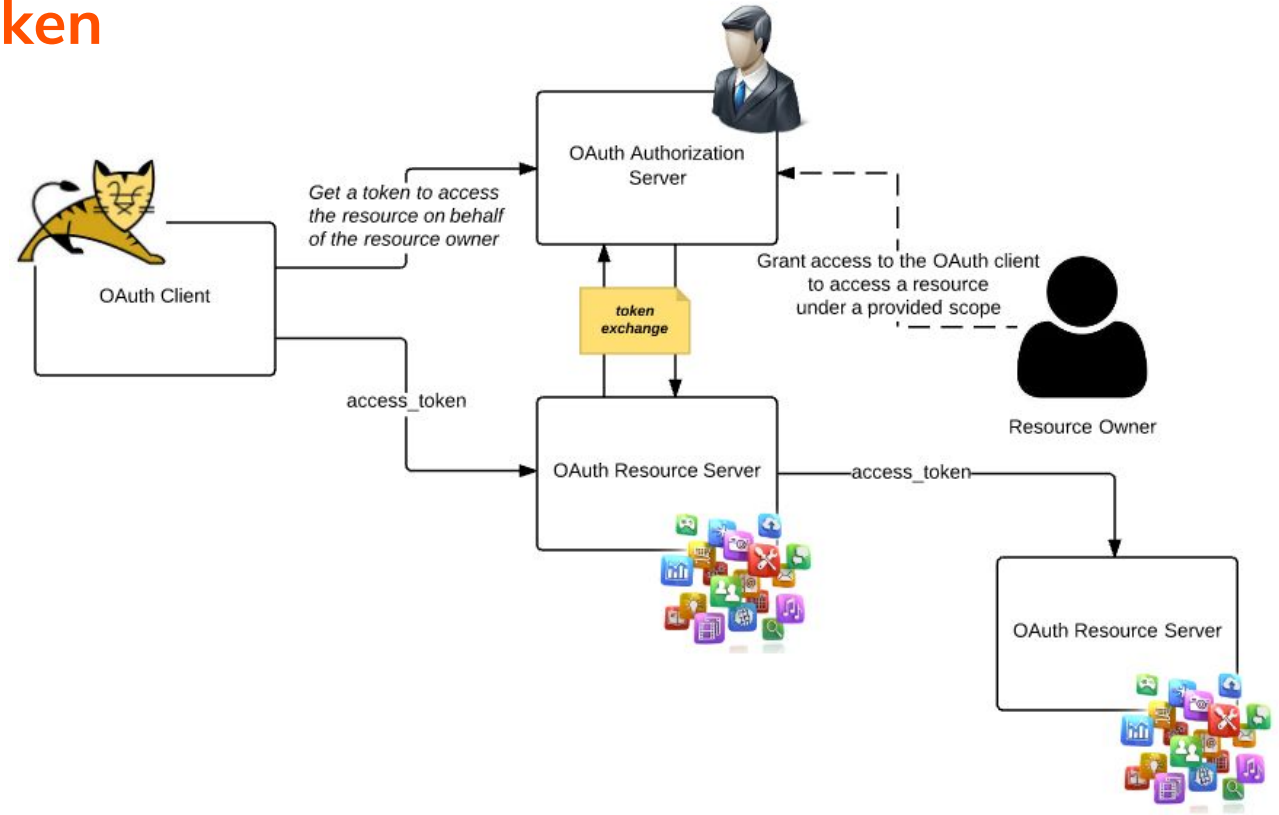
Self-Contained Access Tokens



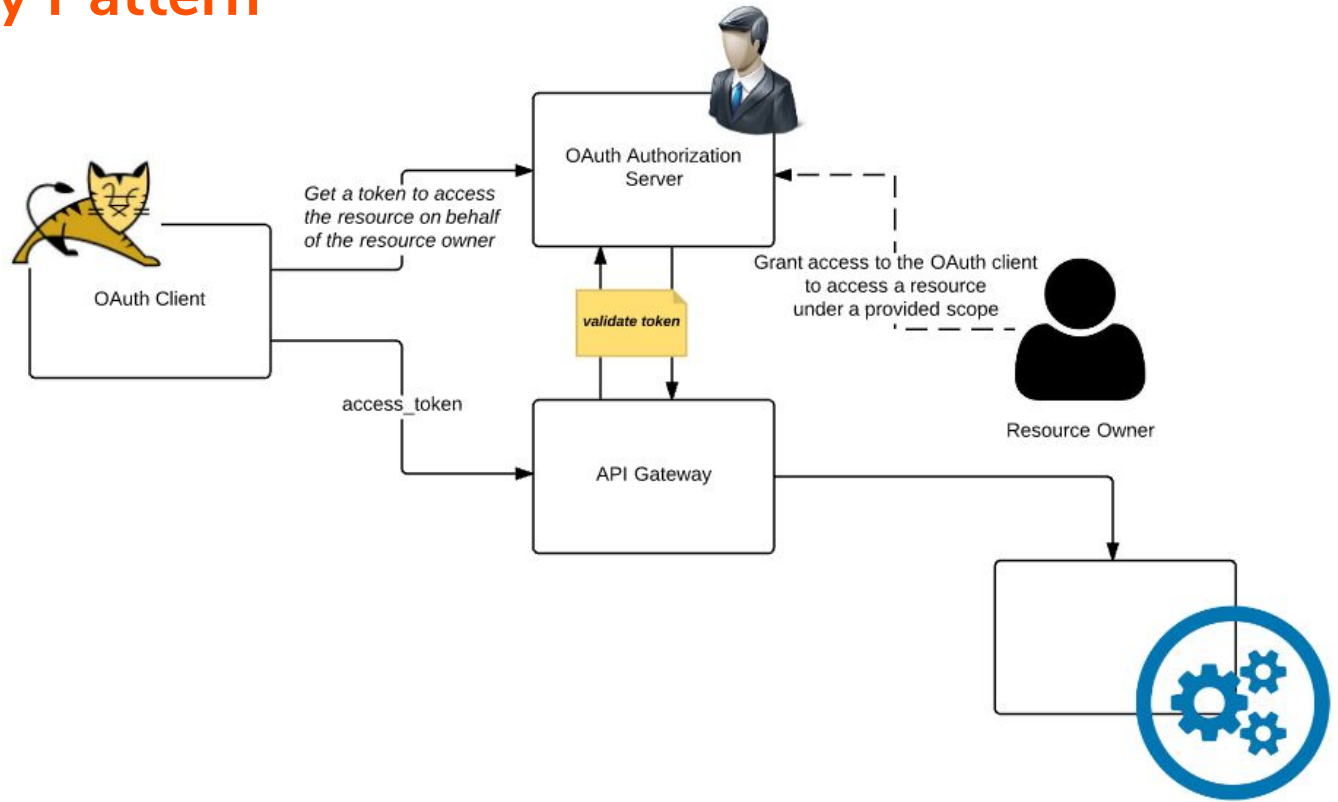
Self-Issued Access Tokens



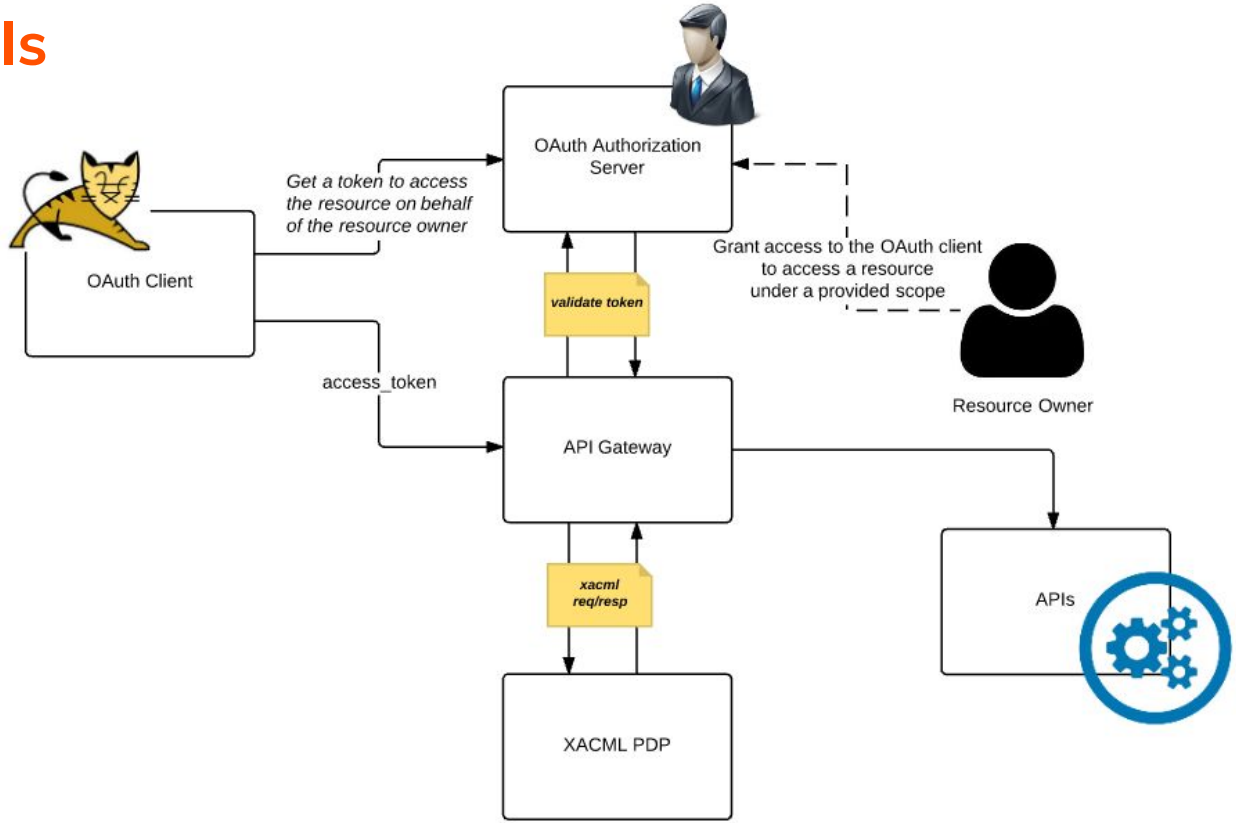
OAuth 2.0 Token Exchange



API Gateway Pattern



Fine-grained Access Control for APIs

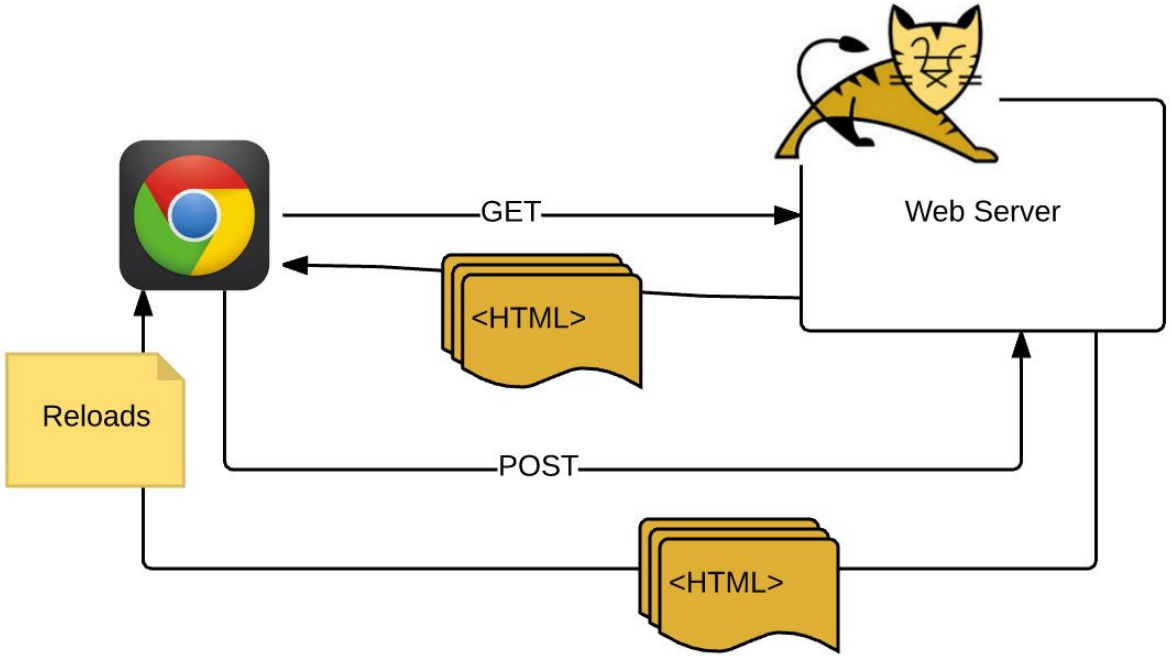


Single Page Applications

- Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.
- An SPA is an application delivered to the browser that doesn't reload the page during use.
- SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads.
- The "page" in SPA is the single web page that the server sends to the browser when the application starts. It's the server rendered HTML that gets everything started. No more, no less. After that initial page load, all of the presentation logic is on the client.
- Much of the work happens on the client side, in JavaScript.
- User Agent-based Application

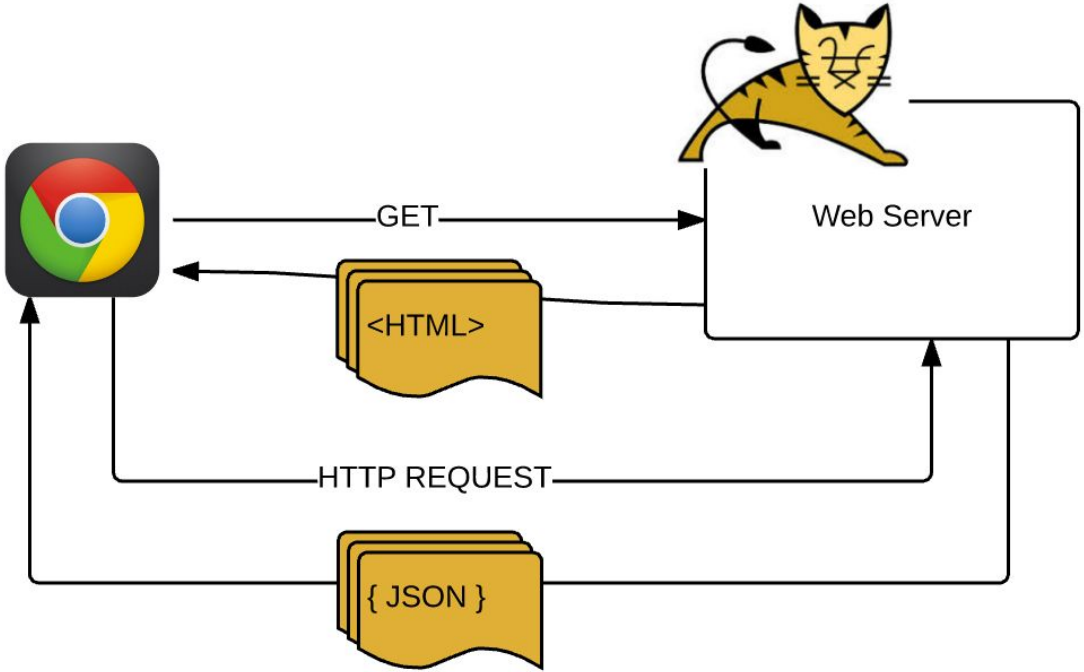
Single Page Applications

TRADITIONAL PAGE LIFECYCLE



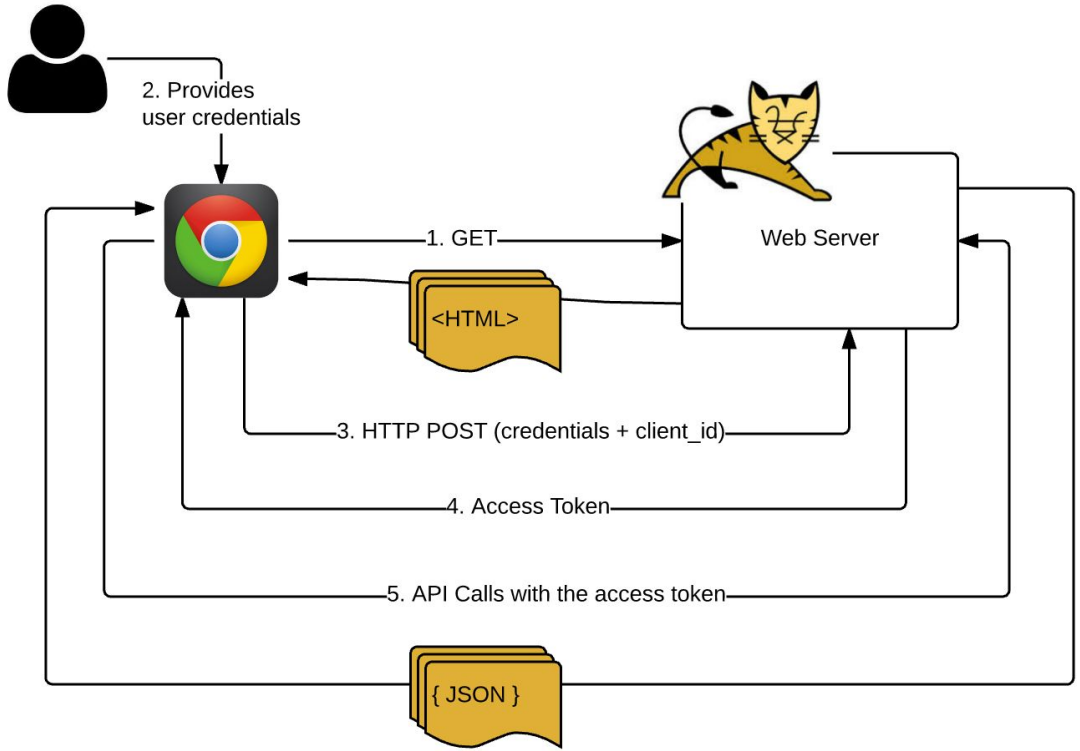
Single Page Applications

SPA LIFECYCLE



Securing SPAs (I)

PASSWORD GRANT TYPE



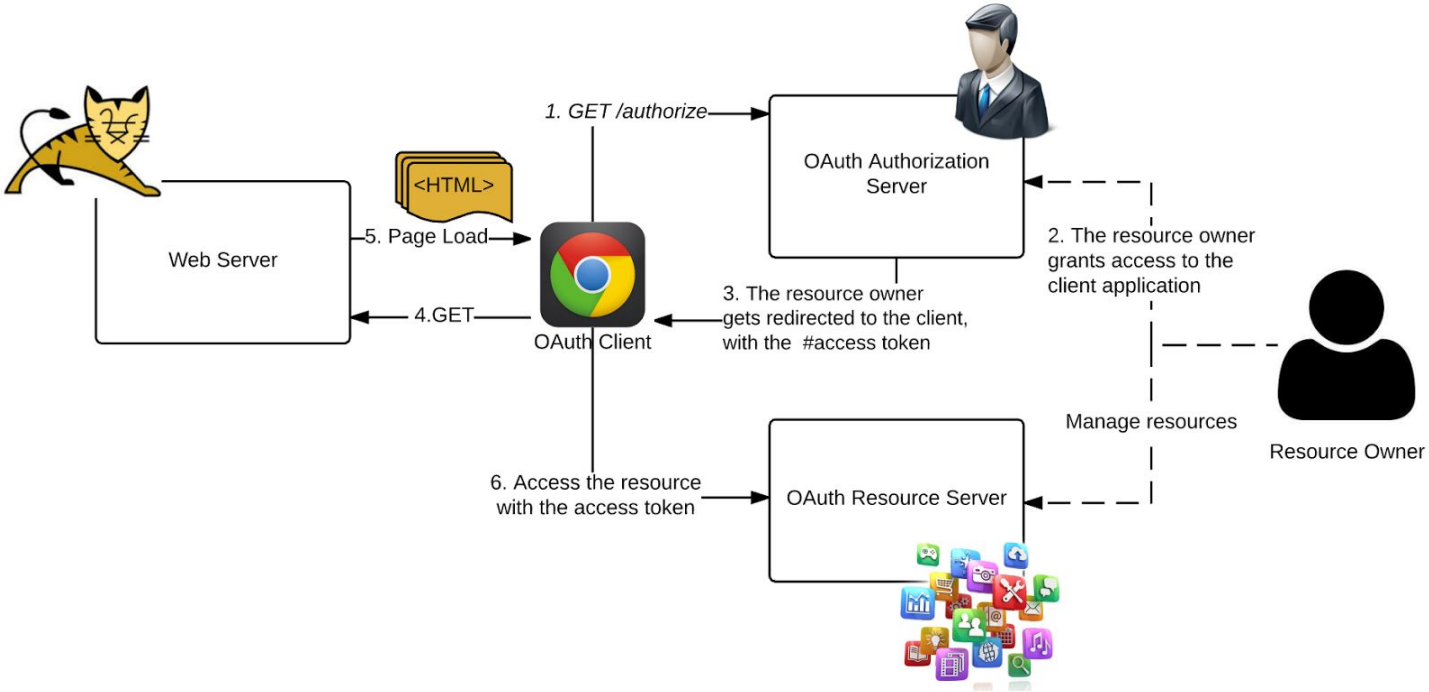
Securing SPAs (I)

PASSWORD GRANT TYPE

- Two fundamental issues you find in any ‘pure’ SPA application.
  - *An SPA accessing an OAuth secured API is - the client cannot be authenticated in a completely legitimate manner.*
  - *An SPA accessing an OAuth secured API is - the access token cannot be made invisible to the end-user.*
- No single sign on experience.
- Users have to provide their credentials directly to the SPA - rather than to the identity provider. Must trust the SPA.
- No UI redirections.

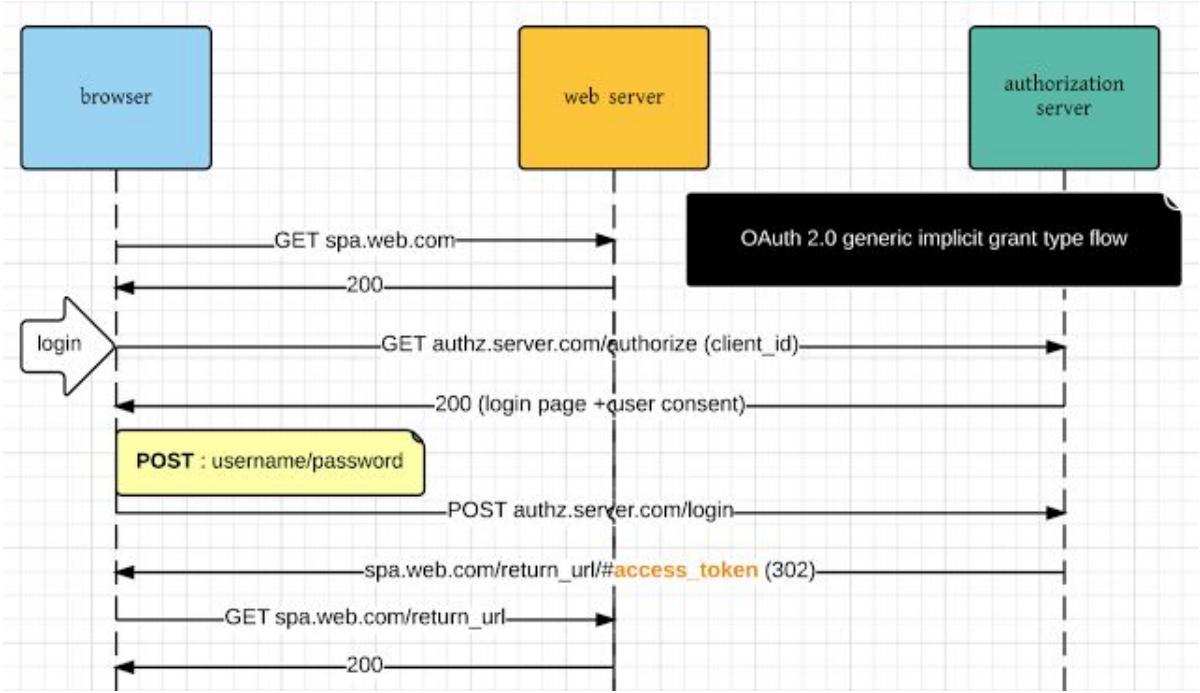
Securing SPAs (II)

IMPLICIT GRANT TYPE



Securing SPAs (II)

IMPLICIT GRANT TYPE



Securing SPAs (II)

IMPLICIT GRANT TYPE

- Two fundamental issues you find in any ‘pure’ SPA application.
  - *An SPA accessing an OAuth secured API is - the client cannot be authenticated in a completely legitimate manner.*
  - *An SPA accessing an OAuth secured API is - the access token cannot be made invisible to the end-user.*
- Single sign on experience.
- Users do not need to provide credentials to the SPA, rather to the identity provider.
- UI redirections.

Overcoming Security Challenges

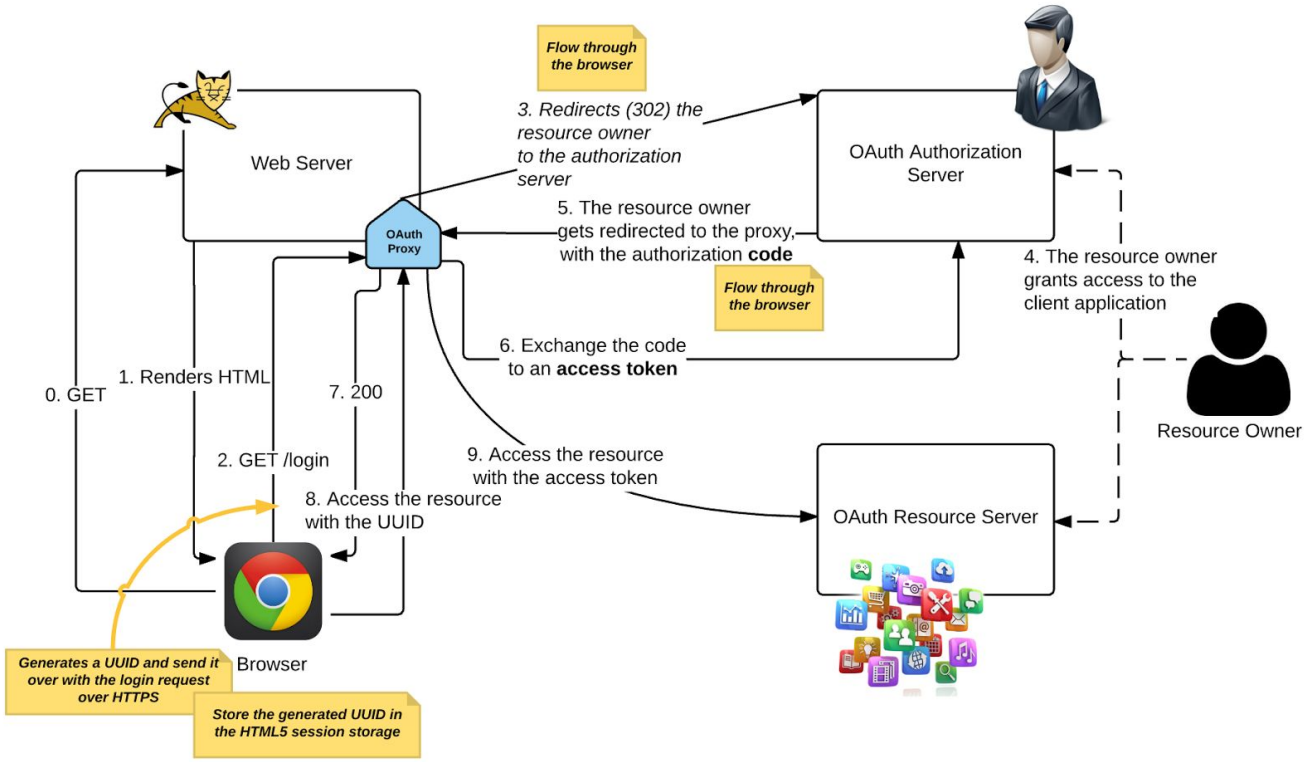
- An SPA accessing an OAuth secured API and the client cannot be authenticated in a completely legitimate manner.
  - Impact
    - Invoke APIs protected with client_credentials grant type.
    - Impersonate a legitimate client application and fool the user to get his consent to access user resources on behalf of the legitimate user.
  - Recommendations
    - Reject any tokens used to access APIs, which are issued under client_credentials grant type.
    - The authorization should do proper validations on the redirect_url.

Overcoming Security Challenges

- An SPA accessing an OAuth secured API and the access token cannot be made invisible to the end-user.
  - Impact
    - Can eat-out throttling limits associated with an API per application.
    - Access token returned backed from the implicit grant type is in browser history. Can be used by illegitimate users.
  - Recommendations
    - Enforce per user per application throttling limits.
    - Make the access tokens short-lived.
    - One time access token - discard the access token in its first use (access token chaining).

Securing SPAs (III)

OAuth Proxy



Securing SPAs (II)

OAuth Proxy

- ~~Two fundamental issues you find in any 'pure' SPA application.~~
  - ~~*An SPA accessing an OAuth secured API is - the client cannot be authenticated in a completely legitimate manner.*~~
  - ~~*An SPA accessing an OAuth secured API is - the access token cannot be made invisible to the end-user.*~~
- Single sign on experience.
- Users do not need to provide credentials to the SPA, rather to the identity provider.
- UI redirections.
- Not pure SPA - all the API calls from the SPA should go through the SPA.

Securing SPAs (IV)

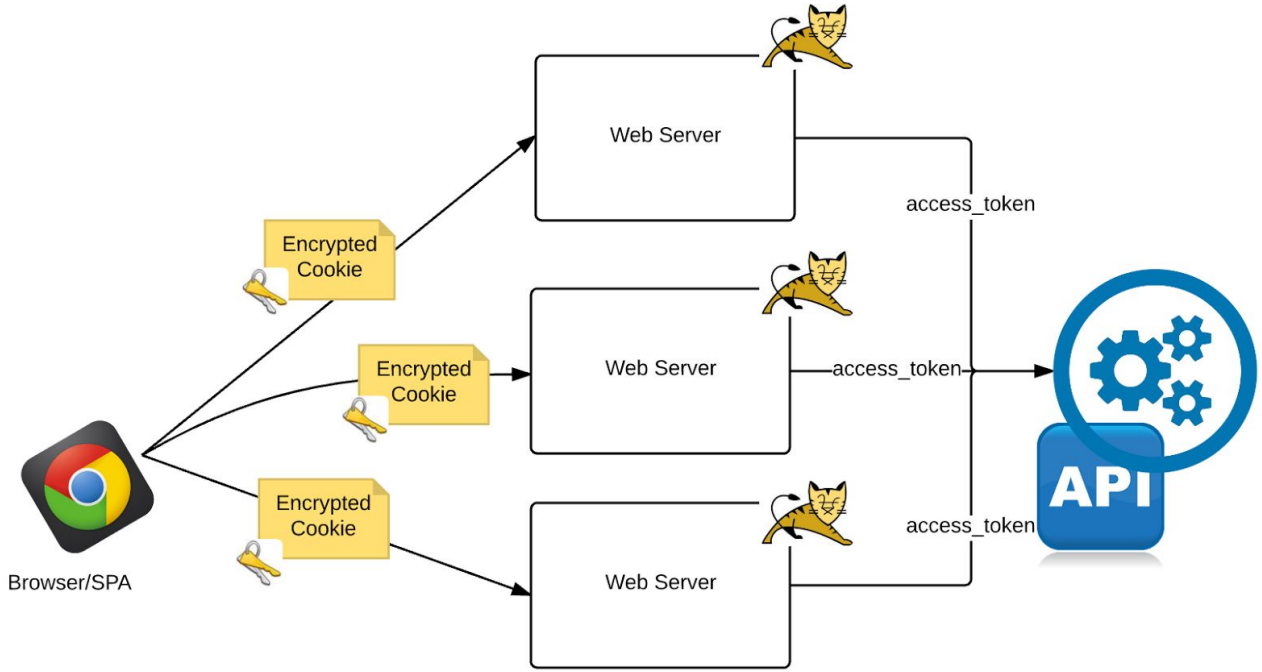
Stateless OAuth Proxy

- Create a JWE with the access token, user info - encrypt and set it in a session cookie, in the response to the login
- All the API calls from the SPA to the proxy, this cookie will be included.

```
 var xmlHttp = new XMLHttpRequest();
      xmlHttp.withCredentials = true;
      xmlHttp.onreadystatechange = function() {
            if (xmlHttp.readyState == 4 && xmlHttp.status == 200) {
                  var obj = JSON.parse(xmlHttp.responseText);
                  document.getElementById('name-id').innerHTML = obj.sub;
            }
};
xmlHttp.open("GET",
                  "https://localhost:9443/oauth2-proxy/users?code="
                        + sessionStorage.getItem("guid"), true);
xmlHttp.send();
```

Securing SPAs (IV)

Stateless OAuth Proxy



OAuth 2.0 for Native Apps

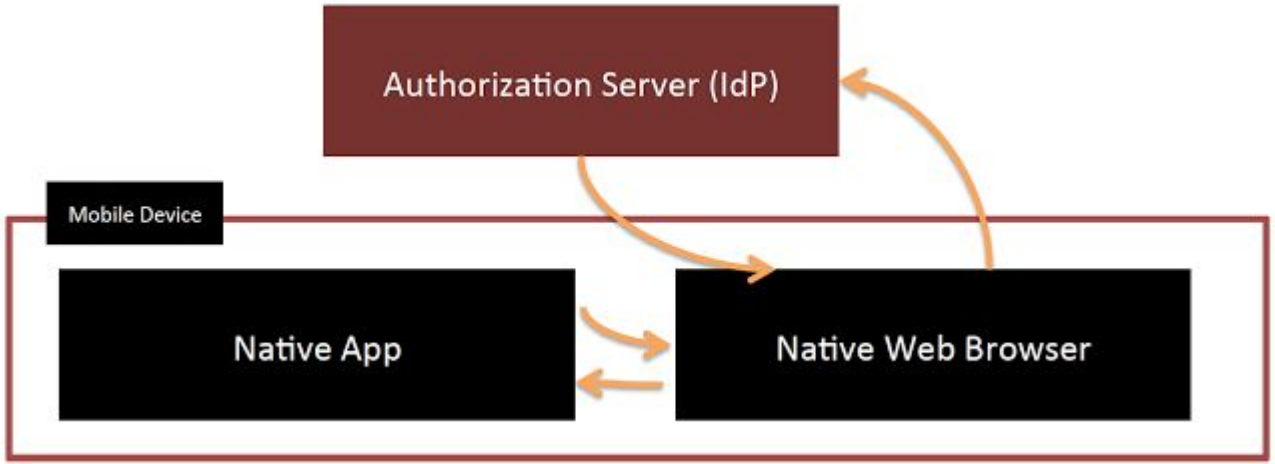
- Use the web view
- Session not shared among multiple native apps.
- Possible phishing attacks

OAuth 2.0 for Native Apps

- Use password grant type
- Session not shared among multiple native apps.
- Possible phishing attacks

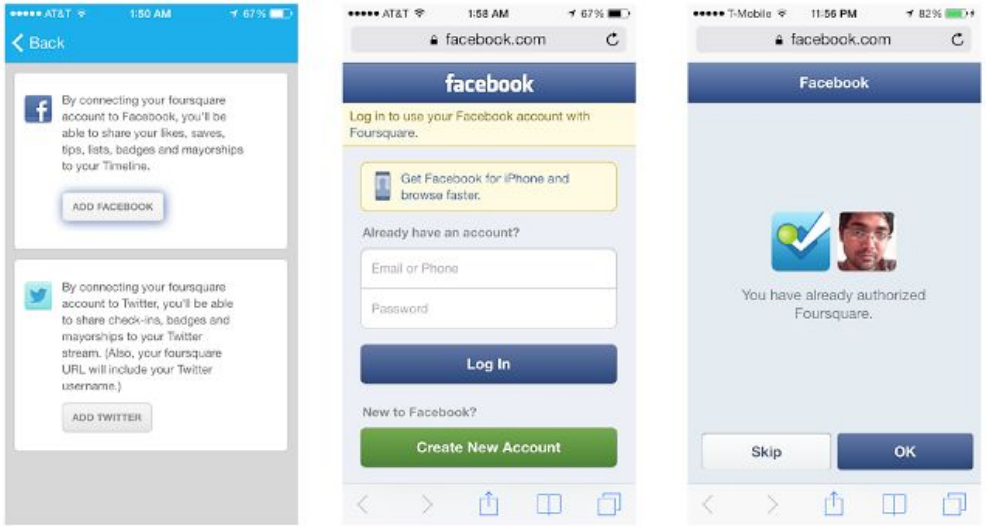
OAuth 2.0 for Native Apps

- Use the system browser
- Session shared among multiple native mobile apps



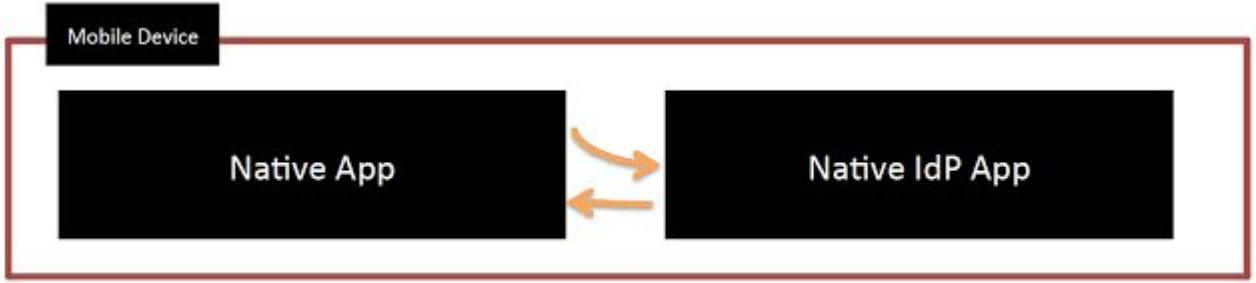
OAuth 2.0 for Native Apps

- Use the system browser
- Session shared among multiple native mobile apps



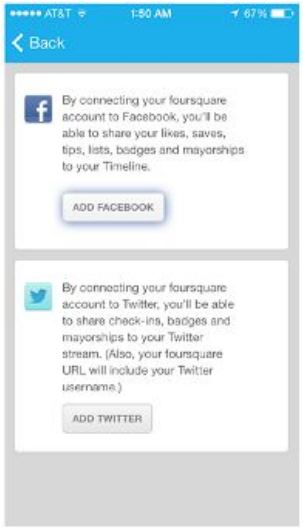
OAuth 2.0 for Native Apps

- Use an IdP proxy
- Session shared among multiple native apps
- NAPPS working group under OpenID foundation
- No more.



OAuth 2.0 for Native Apps

- Use an IdP proxy
- Session shared among multiple native apps
- NAPPS working group under OpenID foundation
- No more.



OAuth 2.0 for Native Apps

- Apple (iOS9+ - SFSafariViewController) and Google (Chrome 45+ - Chrome Custom Tabs)
- This web controller provides all the benefits of the native system browser in a control that can be placed within an application.
- Session shared between apps.

OAuth 2.0 for Native Apps

- The best practices draft under OAuth IETF working group ‘OAuth 2.0 for Native Apps’ recommends that OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser.

PKCE (Proof Key for Code Exchange)

- Authorization Code Grant are susceptible to the authorization code interception attack.
- The PKCE introduces a technique to mitigate against the threat.
- In this attack, the attacker intercepts the authorization code returned from the authorization endpoint within a communication path not protected by TLS.
- The attacker has access to the "client_id" and "client_secret" (if provisioned).
- Uses a cryptographically random string that is used to correlate the authorization request to the token request.

Dynamic Client Registration Profile

- Defines mechanisms for dynamically registering OAuth 2.0 clients with authorization servers.
- The resulting registration responses return a client identifier to use at the authorization server and the client metadata values registered for the client.
- The client can then use this registration information to communicate with the authorization server using the OAuth 2.0 protocol.

Token Introspection Profile

- Defines a method for a protected resource to query an OAuth 2.0 authorization server to determine the active state of an OAuth 2.0 token and to determine meta-information about this token.

```
 POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=mF_9.B5f-4.1JqM&token_type_hint=access_token
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
       "active": true,
        "client_id": "l238j323ds-23ij4",
        "username": "jdoe",
        "scope": "read write dolphin",
       "sub": "Z5O3upPC88QrAjx00dis",
        "aud": "https://protected.example.net/resource",
        "iss": "https://server.example.com/",
        "exp": 1419356238,
       "iat": 1419350238,
       "extension_field": "twenty-seven"
}
```

Token Revocation Profile

- Allows clients to notify the authorization server that a previously obtained refresh or access token is no longer needed

```
POST /revoke HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

Resource Set Registration Profile

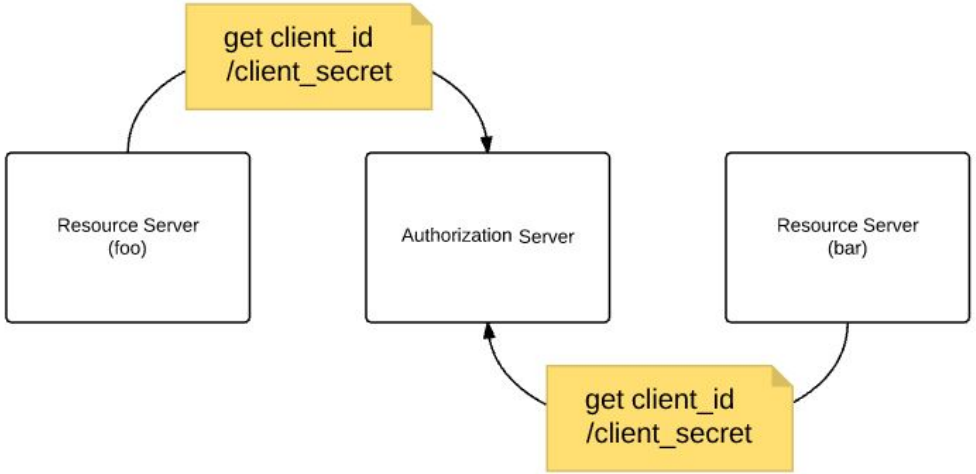
- Defines a resource set registration mechanism between an OAuth 2.0 authorization server and resource server.
- A resource set is one or more resources that the resource server manages as a set, abstractly.
- A resource set may be a single API endpoint, a set of API endpoints, a classic web resource such as an HTML page.
- A set of scopes can be associated with a resource set during the registration.

User Managed Access (UMA)

- Defines how resource owners can control protected-resource access by clients operated by arbitrary requesting parties, where the resources reside on any number of resource servers, and where a centralized authorization server governs access based on resource owner policies.
- User-Managed Access (UMA) is a profile of OAuth 2.0.

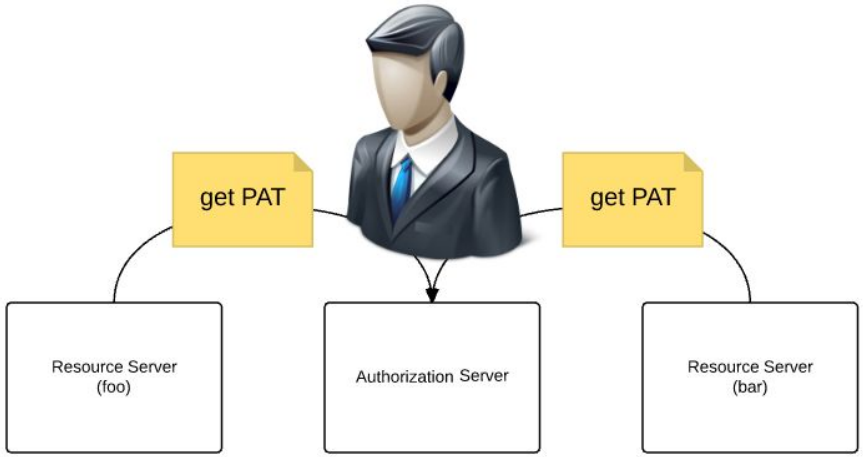
User Managed Access (UMA)

- Each resource server has to register itself with the centralized authorization server.
- This one time operation between the resource server and authorization server.



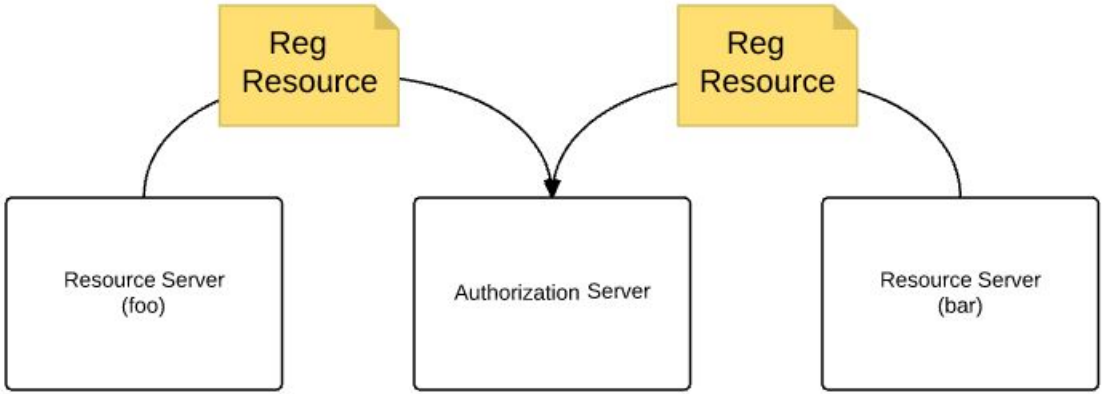
User Managed Access (UMA)

- The resource server get a PAT (Protection API Token) from the authorization server on behalf of the resource owner.
- PAT is an OAuth 2.0 access token with the **uma_protected** scope.



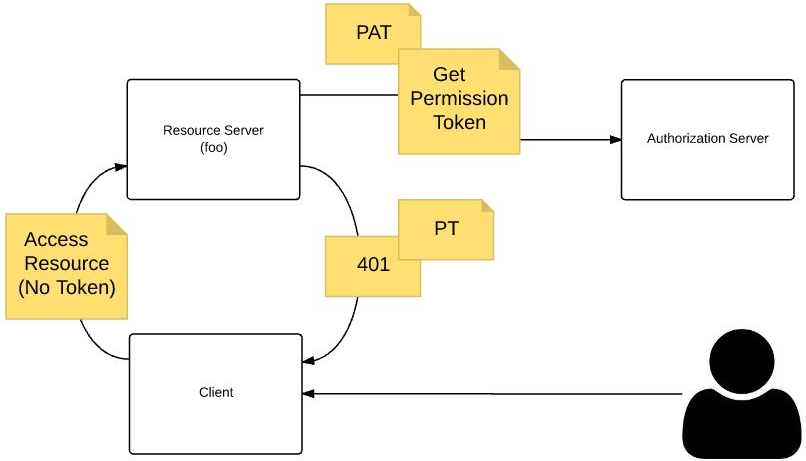
User Managed Access (UMA)

- The resource server registers with the authorization server.
- The communication between the authorization server and the resource server is defined by the OAuth 2.0 Resource Set Registration profile.



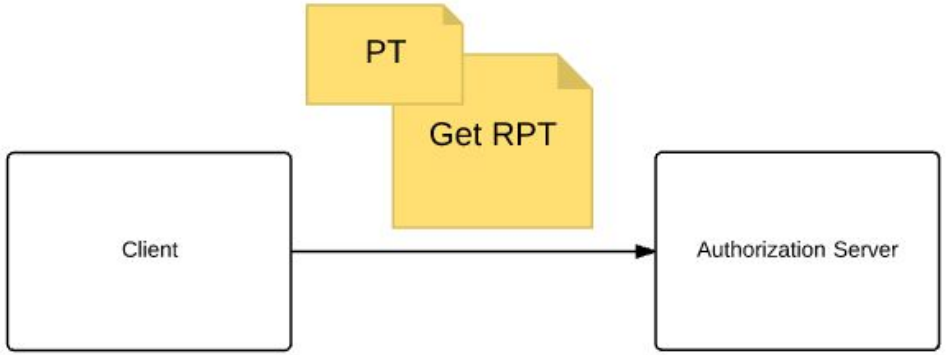
User Managed Access (UMA)

- Client accesses a protected resource with no token.
- The resource server requests one or more permission on the client's behalf from the authorization server, corresponding to the access attempt by the client (with PAT).
- Authorization server responds back with a permission ticket.
- Resource Server responds with 401



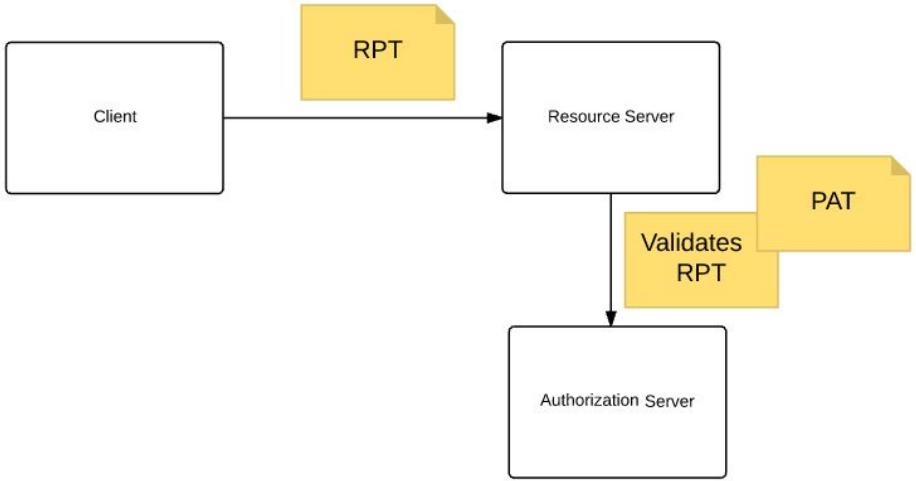
User Managed Access (UMA)

- Client requests a RPT (Requesting Party Token) from the Authorization Server.
- Uses UMA grant type
- Passes the permission ticket obtained from the resource server.
- This can be a direct call or redirect.
- If it's a direct call client push user claims to the authorization - and if it's a redirect, the authorization server can learn about the requesting party.



User Managed Access (UMA)

- Client accesses the resource with the RPT
- The Resource Server uses the introspection endpoint of the Authorization Server to find the status of the RPT



THREATS / MITIGATIONS / BEST PRACTICES

SESSION INJECTION

THREATS

- CSRF (Cross Site Request Forgery Attack)
  - The **attacker** tries to log into the target website (OAuth 2.0 client) with his account at the corresponding identity provider.
  - The **attacker** blocks the redirection to the target web site, and captures the authorization code. The target web site never sees the code.
  - The **attacker** constructs the callback URL for the target site - and lets the **victim**, clicks on it.
  - The **victim** logs into the target web site, with the account attached to the **attacker** - and adds credit card information.
  - The **attacker** too logs into the target website with his/her valid credentials and uses **victim's** credit card to buy goods.

SESSION INJECTION

MITIGATIONS / BEST PRACTICES

- Short-lived authorization code
- Use the **state** parameter as defined in the OAuth 2.0 specification.
  - Generate a random number and pass it to the authorization server along with the grant request.
  - Before redirecting to the authorization server, add the generated value of the state to the current user session.
  - Authorization server has to return back the same state value with the authorization code to the return_uri.
  - The client has to validate the state value returned from the authorization server with the value stored in the user's current session - if mismatches - reject moving forward.

SESSION INJECTION

MITIGATIONS / BEST PRACTICES

- Use Proof Key for Code Exchange (PKCE)
  - https://tools.ietf.org/html/rfc7636

TOKEN LEAKAGE

THREATS

- Attacker may attempt to eavesdrop authorization code/access token/refresh token in transit from the authorization server to the client.
  - Malware installed in the browser (public clients)
  - Browser history (public clients / URI fragments)
  - Intercept the TLS communication between the client (confidential) and the authorization server (exploiting vulnerabilities at the TLS layer)
    - Heartbleed
    - Logjam
- Authorization Code Flow Open Redirector

TOKEN LEAKAGE

THREATS

- A malicious app can register itself as a handler for the same custom scheme as of a legitimate OAuth 2.0 native app, can get hold of the authorization code.
- Attacker may attempt a brute force attack to crack the authorization code/access token.
- Attacker may attempt to steal the authorization code/access token/refresh token stored in the authorization server.

TOKEN LEAKAGE

MITIGATIONS / BEST PRACTICES

- Always on TLS (use TLS 1.2 or later)
- Address all the TLS level vulnerabilities both at the client, authorization server and the resource server.
- The token value should be >=128 bits long and constructed from a cryptographically strong random or pseudo-random number sequence.
- Never store tokens in clear text - but the salted hash.
- Short-lived tokens.
    - LinkedIn has an expiration of 30 seconds for its authorization codes.

TOKEN LEAKAGE

MITIGATIONS / BEST PRACTICES

- The token expiration time would depend on the following parameters.
  - Risk associated with token leakage
  - Duration of the underlying access grant
  - Time required for an attacker to guess or produce a valid token
- One-time authorization code
- One-time access token (implicit grant type)
- Use PKCE (proof key for code exchange) to avoid authorization code interception attack.
  - Have S256 as the code challenge method
- Enforce standard SQL injection countermeasures

TOKEN LEAKAGE

MITIGATIONS / BEST PRACTICES

- Avoid using the same client_id/client_secret for each instance of a mobile app - rather use the Dynamic Client Registration API to generate a key pair per instance.
  - Most of the time the leakage of authorization code becomes a threat when the attacker is in hold of the client id and client secret.
- Restrict grant types by client.
  - Most of the authorization servers do support all core grant types. If unrestricted, leakage of client id/client secret gives the attacker the opportunity obtain an access token via client credentials grant type.

TOKEN LEAKAGE

MITIGATIONS / BEST PRACTICES

- Enable client authentication via a much secured manner.
  - JWT client assertions
  - TLS mutual authentication
  - Have a key of size 2048 bits or larger if RSA algorithms are used for the client authentication
  - Have a key of size 160 bits or larger if elliptic curve algorithms are used for the client authentication
- White-list callback URLs (redirect_uri)
  - The absolute URL or a regEx pattern

TOKEN REUSE/MISUSE

THREATS

- A malicious resource (an API / Microservice) could reuse an access token used to access itself by a legitimate client to access another resource, impersonating the original client.
- An evil web site gets an access token from a legitimate user, can reuse it at another web site (which trusts the same authorization server) with the implicit grant type
  - https://target-app/callback?access_token=<access_token>
- A legitimate user misuses an access token (issued under implicit grant type/SPA) to access a set of backend APIs in a way, exhausting server resources.

TOKEN REUSE/MISUSE

MITIGATIONS / BEST PRACTICES

- Use scoped access tokens. Qualify the scope name, with a namespace unique to the resource (resource server).
- The client obtains the access token for a given audience - by passing the audience information (representing the resource server) to the token endpoint - as per https://tools.ietf.org/id/draft-tschofenig-oauth-audience-00.html.
- Use OAuth for authorization not for authentication.
    - Use OpenID Connect for authentication

TOKEN REUSE/MISUSE

MITIGATIONS / BEST PRACTICES

- To avoid exhausting resources at the server side, enforce throttle limits by user by application. In case an attacker wants to misuse a token - the worst he/she can do is to eats his/her own quota.

TOKEN EXPORT

THREATS

- An attacker could export an access token from its originating channel and use somewhere else.
- A common attack vector for SPAs (Single Page Applications), where the legitimate user takes out the access token from the web page, and uses it somewhere else to use some functionality not provided by the original client.
- A major concerns with bearer tokens.

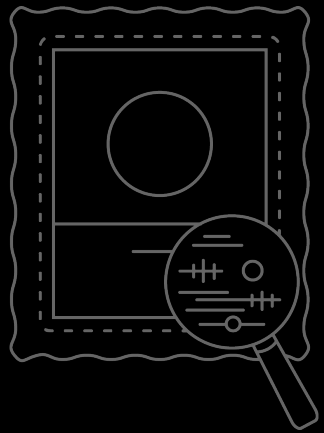
TOKEN EXPORT

MITIGATIONS / BEST PRACTICES

- The use of Token Binding protects access tokens from man-in-the-middle and token export and replay attacks.
- https://tools.ietf.org/html/draft-jones-oauth-token-binding-00

TOKEN BINDING

- Cryptographically binds access/refresh tokens to the TLS connections over which they are intended to be used.
- Protects access/refresh tokens from man-in-the-middle and token export and replay attacks.
- Token Binding
  - https://tools.ietf.org/html/draft-ietf-tokbind-protocol
  - https://tools.ietf.org/html/draft-ietf-tokbind-negotiation
  - https://tools.ietf.org/html/draft-ietf-tokbind-https
- Token Binding Application in OpenID Connect
  - http://openid.net/specs/openid-connect-token-bound-authentication-1_0.html
- Token Binding Application in OAuth
  - https://tools.ietf.org/html/draft-ietf-oauth-token-binding
- Token Binding & Reverse Proxy
  - https://tools.ietf.org/html/draft-campbell-tokbind-tls-term-00



THANK YOU

wso2.com