# OpenAPI Authentication & Authorization

## Overview

Open Services encompasses two logical components: an Authorization Server and the OpenAPI services. The Authorization Server will delegate user authentication to an Identity Provider using the Federated Authentication protocol. It will also provide a mechanism for the user to grant consent to the application for access to their information to support the OAuth protocol. The Identity Provider is a Security Token Service that provides authentication using a log in page. The OpenAPI services are the collection of REST services comprising the API. The development of OpenAPI encompasses the development of these three components.

## References

This design is based upon the implementation of several standards:

1. WS-Federation2007: <http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>
2. OAuth2.0 Draft 13: <http://tools.ietf.org/id/draft-ietf-oauth-v2-13.txt>
3. OAuth2.0 Bearer Tokens (Draft 8): <http://www.ietf.org/id/draft-ietf-oauth-v2-bearer-08.pdf>
4. SAML2.0 Bearer Assertion Profiles for OAuth 2.0 (Draft 5): <http://tools.ietf.org/pdf/draft-ietf-oauth-saml2-bearer-05.pdf>

WS-Federation (#1) is used in this design to manage the authentication of browser-based users. OAuth (#2) is used to authenticate and authorize access to the OpenAPI web-services. #3 describes how access-tokens generated via OAuth are presented by clients to the OpenAPI web-service endpoints. SAML tokens can be used in lieu of a user’s username/password as described in #4.

# Design

## Federated Authentication Protocol

Federated Authentication is largely concerned with externalizing user authentication from an application that consumes it. This allows a user to log into multiple different applications with the same credentials as well as allowing a user to log in to an application with multiple different sets of credentials. It also makes possible single sign on behavior with applications if different domains. Federated Authentication makes use of SAML tokens which represent a user’s identity in the form of claims. Clients can authenticate a user in one of two ways. The first is an active mode in which an application invokes calls to a Security Token Service, STS, endpoint directly. The other is a browser-based approach which relies on a series of 302 redirects and JavaScript induced posts to accomplish the same thing. This is the approach that is needed for this design. A full discussion of Federated Authentication is beyond the scope of this document, but it is important to know how it is implemented since it will be necessary to host a security token service endpoint to facilitate user authentication. This is discussed in more depth later. For now be aware that most of this functionality is made possible with the use of Windows Identity Foundation, WIF.

## OAuth Protocol

The OAuth protocol defines four components (called roles) that are coordinated to secure access to a protected resource. A *protected resource* is any resource (such as a user’s profile information) that is provided to a client. A *client* is the role that consumes the protected resource. The client is therefore the application making calls to OpenAPI. A *resource owner* is the role with exclusive ownership of a protected resource. In this design, the resource owner is the user accessing the application. A *resource server* is the role that protects the resource. The resource server is the OpenAPI restful services that provide access to the resource requested by the application (which has been previously granted by the user). Even though the user owns the resource, the resource is stored with the resource server and the user implicitly trusts that the resource server will not divulge this information to any client without their explicit permission. Finally, an *authorization server* is the role that authenticates the user and grants permission to the client for access to the resource on behalf of the user. The granting of permissions is also referred to as providing *user consent*. Consent to resources can be coarse or fine grained which is referred to as *scopes*. For example, a user may grant permission to an application to access profile information but not the right to reserve movies.

## Components

### Overview

OpenAPI must authenticate not only the user accessing the API but also the client application. The application will always be authenticated by the Apigee portion of OpenAPI using a client id and secret. Before a new application is granted access to OpenAPI this information will be provided to Apigee. The authorization server will authenticate the user at the beginning of a user’s session using the Federated Authentication protocol. It will also provide a way for the user to grant the application access to their information. Successful authentication will result in a SAML token sent to the client. The client will then exchange this token for an OAuth access token which is used to authenticate the user for subsequent API calls during the user’s session using the OAuth protocol. Since all calls to OpenAPI are proxied through Apigee, Apigee will be responsible for exchanging the tokens and authenticating the user with subsequent requests. This simplifies the roles of the authorization server and the API server. The authorization server will simply perform Federated Authentication and grant access to applications. The API server will simply host the REST server endpoints.



Figure : Open Services and Dependency Components

### Client

The client is the application that the user interacts with through a browser. The client will issue calls directly to the Apigee layer of OpenAPI hosted in the cloud by Apigee.

### Apigee

Apigee is the layer that faces the internet and exposes the OpenAPI service endpoints. It is responsible for authenticating both the client application and the user with the use of their key manager during a user’s session. The initial user authentication, however, is passed through to the Authorization Server. During the user’s session Apigee will also pass the client id and user id through to Open Services. This simplifies the design by defining distinct responsibilities for the Apigee and Open Services layers of OpenAPI.

### Authorization Server

The Authorization Server is responsible for kicking off the Federated Authentication sequence which involves the use of the Federation Provider and the Identity Provider. Since authentication will occur during a user’s browser session, the authentication will occur using the passive profile which relies on a series of browser redirects and JavaScript induced posts to propagate a user’s identity. The Authorization Server will also host the grant page that allows the user to grant the client application access to their information. The grant page will have the same look and feel as the Redbox.com web site which means it may need to share UI assets. The Authorization Server will be an ASP.NET MVC application that hosts both this service endpoint and the grant page.

### OpenAPI Server

The OpenAPI Server will host all the service endpoints that comprise the OpenAPI services. It will receive calls that are proxied through Apigee. Since Apigee is responsible for authentication during a user’s session, the OpenAPI Server will only be responsible for responding to service calls. Apigee will forward the user id and application id within a set of custom HTTP headers with every request.

### Federation Provider

STS servers are generally categorized as either a Federation Provider or an Identity Provider. The Identity Provider is responsible for user authentication while the Federation Provider is responsible for relaying a user’s identity from an Identity Provider to the Relying Party which requested the user’s identity. The Relying Party in this design is the Authorization Server. Federation Providers are useful for establishing trust relationships with Identity Providers and Relying Parties hosted in different domains. With this design, the Federation Provider will be hosted within the Microsoft Cloud using Azure’s Access Control Service, ACS. No custom development is needed. ACS only needs to be configured with the proper trust relationships with the Identity Provider and Relying Party.

### Identity Provider

The Identity Provider will be responsible for authenticating users at the beginning of a user’s session using Federated Authentication. As a result it will also display the login page. Using passive authentication, the Identity Provider does not initially get accessed directly. The Authorization Server will first direct the browser to the Federation Provider which in turn directs the browser to the Identity Provider. Based on the trust relationships between different Identity Providers, this is necessary to provide cross domain authentication. Each redirect wraps the request with an additional return URL so that the browser can make its way back to the server that initiated the log in sequence. The Identity Provider will also support an active authentication endpoint which allows a non-browser based applications to invoke WS-Federation standard Federated Authentication calls to the Federated Provider and Identity Provider directly without the need to rely on browser redirects and posts. Active authentication flows will be added in a subsequent release.

## Use Cases

### User Begins New Session with First Visit to a Web Client Application

Given: Client does not have an access token and user has no pre-existing consent for the application

When: Client calls \\api.redbox.com\authorize?response\_type=code&client\_id=XXX on the Apigee Gateway

Then: Apigee Gateway redirects client (including all headers & cookies) to \\Authorization Service\authorize?response\_type=code&client\_id=XXX

Given: Client does not have an access token (regardless of existence of pre-existing consents)

When: Client calls \\api.redbox.com\token?grant\_type=token&client\_id=XXX on the Apigee Gateway

Then: Apigee Gateway authenticates client\_id

Apigee Gateway redirects call (including all headers & cookies) to \\Authorization Service\authorize?response\_type=token&client\_id=XXX

Given: Request does not include any pre-existing FedAuth cookies;

When: Apigee gateway redirects request to Authorization Service/authorize (upon either a call to /authorize?response\_type=code OR to /token?grant\_type=token)

Then: WIF FAM module on Authorization Service intercepts request and redirects to ACS to kick-off passive federated authentication

Given: FedAuth cookies are present (user is now authenticated to Authorization Service); application is exempt from consent

When: Apigee Gateway redirects call to Authorization Service/authorize

Then: Authorization Service skips permission page and informs Apigee Gateway of the consent and scopes

Given: FedAuth cookies are present (user is authenticated to Authorization Service); no user consent pre-exists for the application requiring consent

When: Apigee Gateway redirects call to Authorization Service/authorize (regardless of response\_type)

Then: Authorization Service displays permissions page

Given: FedAuth cookies are present (user is authenticated to Authorization Service); user has pre-existing application consent for application requiring consent

When: Apigee Gateway redirects call to Authorization Service/authorize with response\_type != token

Then: Authorization Service skips the permission page and informs Apigee Gateway of the consent & scopes

Given: FedAuth cookies are present (user is authenticated to Authorization Service); user has pre-existing application consent for application requiring consent

When: Apigee Gateway redirects call to Authorization Service/authorize with response\_type == token

Then: Authorization Service displays permissions page   
Note: The above is necessary since the client app has not been fully authenticated, we ALWAYS ask the user to grant permission to the app, regardless of whether they have previously granted access to that same app.

Given: Authorization Service has displayed permission page

When: User grants consent

Then: Authorization Service records permission in the consent data store (UserConsent table and Audit table)

Authorization Service informs Apigee Gateway of the consent & scopes

Given: Authorization Service has displayed permission page

When: User denies consent

Then: Auth Service records denial in Audit

Authorization Service informs Apigee Gateway of the denial

### Client Issues Calls to OpenAPI operations that do not Require User Authentication

Given: Client does not have an access token

When: Client issues a call to Products, Stores, Top20, or Inventory but does not provide client\_id & client\_secret as query string params

Then: Apigee Gateway issues a 401 Unauthorized status code, along with a WWW-Authenticate header with a value of “Bearer”

Given: Client does have an access token

When: Client issues a call to Products, Stores, Top20 or Inventory

Then: Apigee Gateway authenticates access token before processing the request

### CE/Mobile device Client Issues Calls to OpenAPI

Given: Client does not have an access token

When: Client issues a call to OpenAPI

Then: Apigee Gateway issues 401 Unauthorized status code (per Bearer tokens extension to OAuth2.0 standard )

Given: Client has a SAML2.0 token issued by ACS

When: Client calls Apigee Gateway/token?grant\_type="urn:ietf:params:oauth:grant-type:saml2-bearer" and includes the SAML2.0 token in the request (per IETF draft draft-ietf-oauth-saml2-bearer)

Then: Apigee Gateway authenticates client\_id & client\_secret

Apigee Gateway forwards the request to Authorization Service/SAMLAuthorize

Given: Client has a SAML2.0 token issued by ACS and has called Apigee Gateway/token

When: Apigee Gateway forwards a Client submitted SAML2.0 token issued by ACS

Then: Authorization Service confirms the integrity of the SAML2.0 token

Authorization Service looks for pre-existing application consent; constructs one if not found

Authorization Service informs Apigee Gateway of the consent & scopes

### Batch Client Issues Calls to OpenAPI

Given: Client does not have an access token

When: Client issues a call to OpenAPI to Inventory or Customer

Then: Apigee Gateway issues 401 Unauthorized status code with WWW-Authenticate: Bearer header

Given: Client does not have an access token

When: Client issues a call to Products, Stores, Top20, or Inventory but does not provide client\_id & client\_secret as query string params

Then: Apigee Gateway issues 401 Unauthorized status code with WWW-Authenticate: Bearer header

Given: Client does not have an access token

When: Client calls Apigee Gateway/token?grant\_type=client\_credentials

Then: Apigee Gateway authenticates client\_id & client\_secret; validates that client\_id is authorized to use ‘client\_credentials’; issues token

Given: Client does have an access token

When: Client issues a call to Products, Stores, Top20 or Inventory

Then: Apigee Gateway authenticates access token before processing the request

Given: Client does have an access token

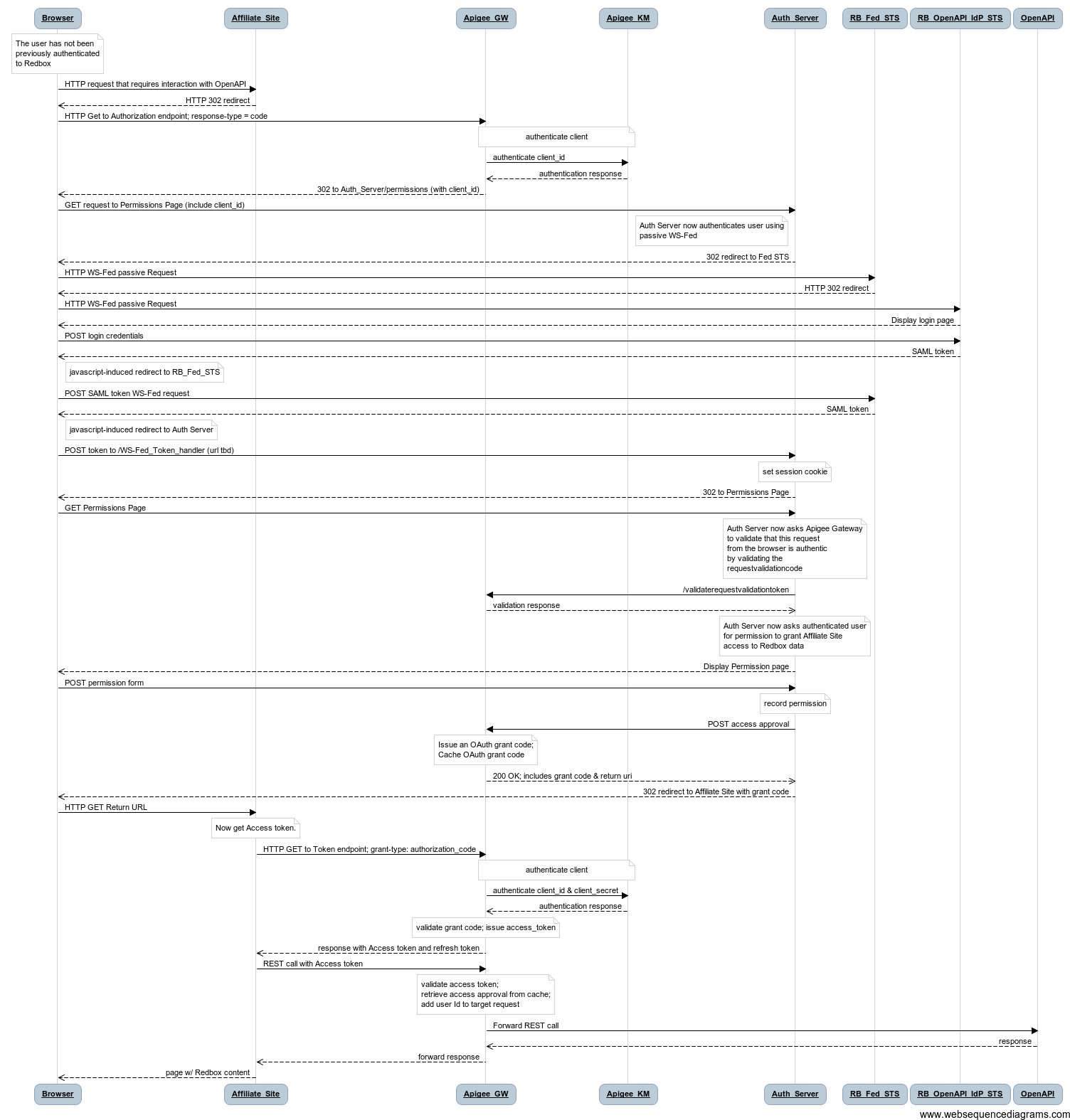
When: Client issues a call to Reservation or Customer or Partner API operations

Then: Apigee Gateway authenticates access token before forwarding the request to OpenServices

## Authentication Flows

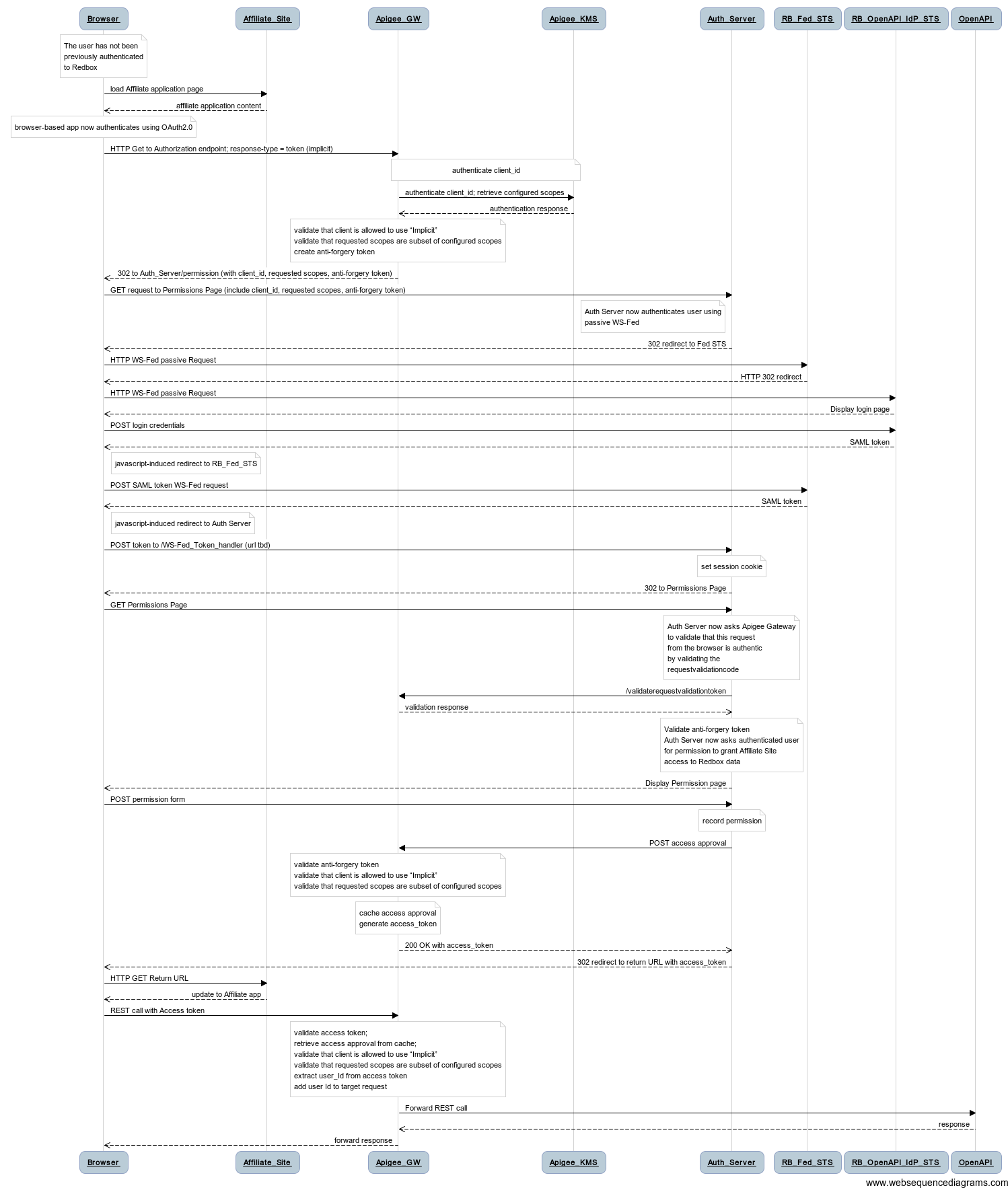
This section provides sequence diagrams that describe the interaction between the components for each of the four profiles supported and the sequence for Token Refresh:

### Auth\_Code

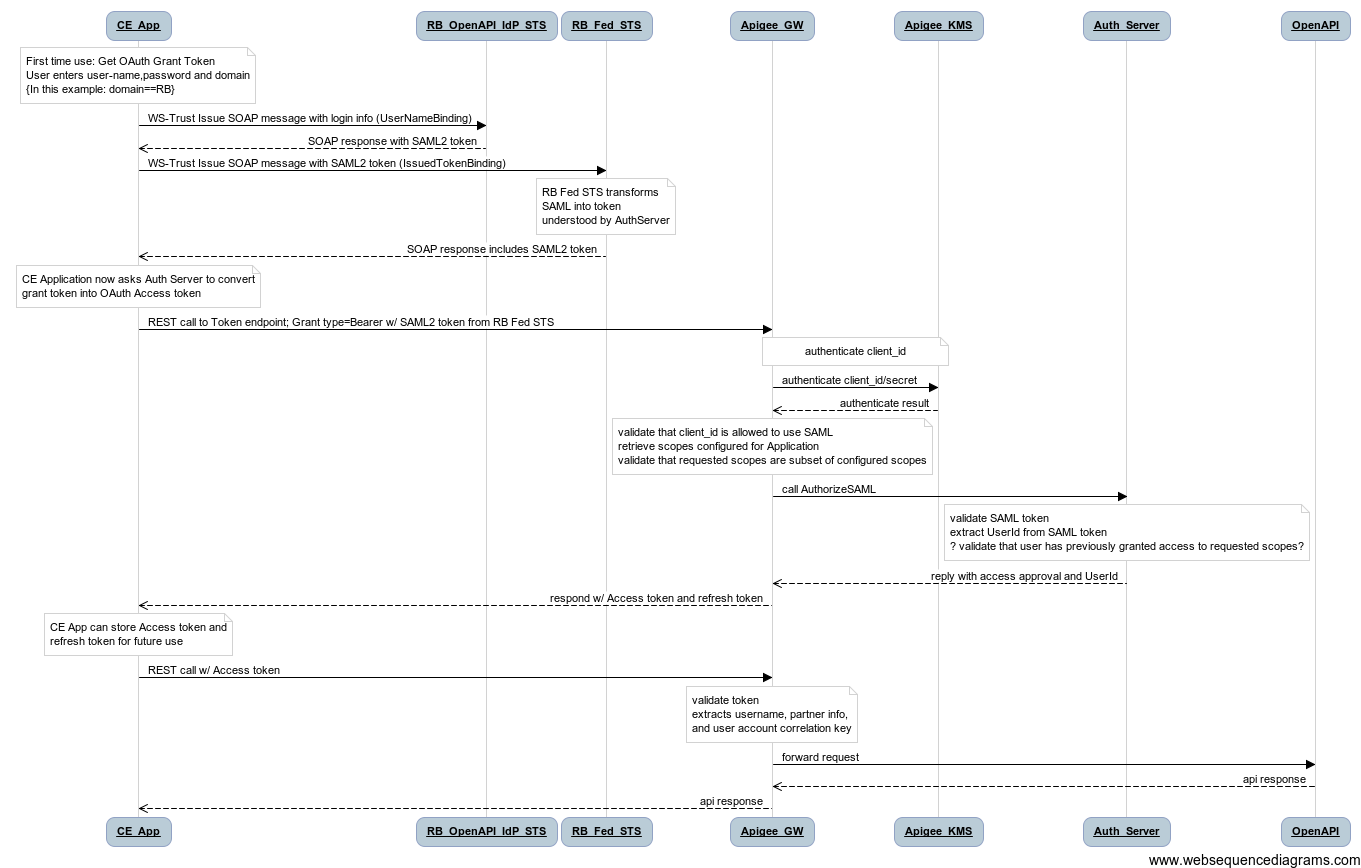


### Implicit

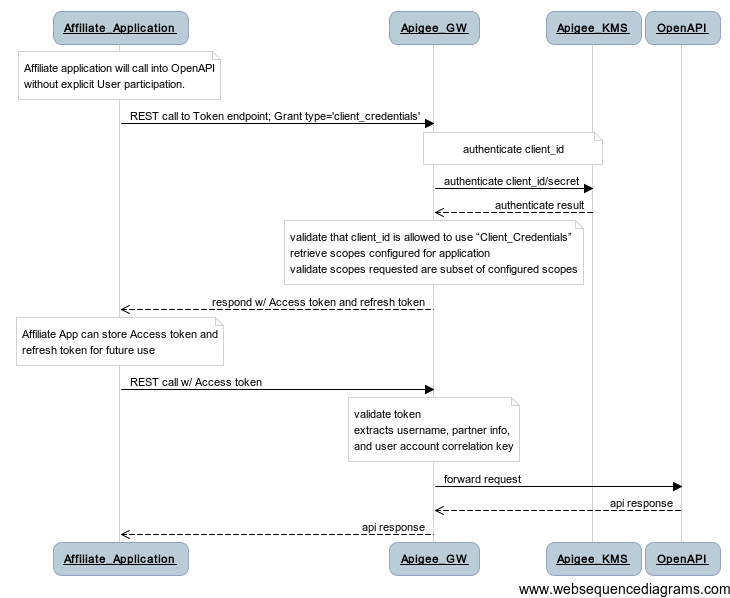
Note: this profile is low-priority and may be dropped from scope.



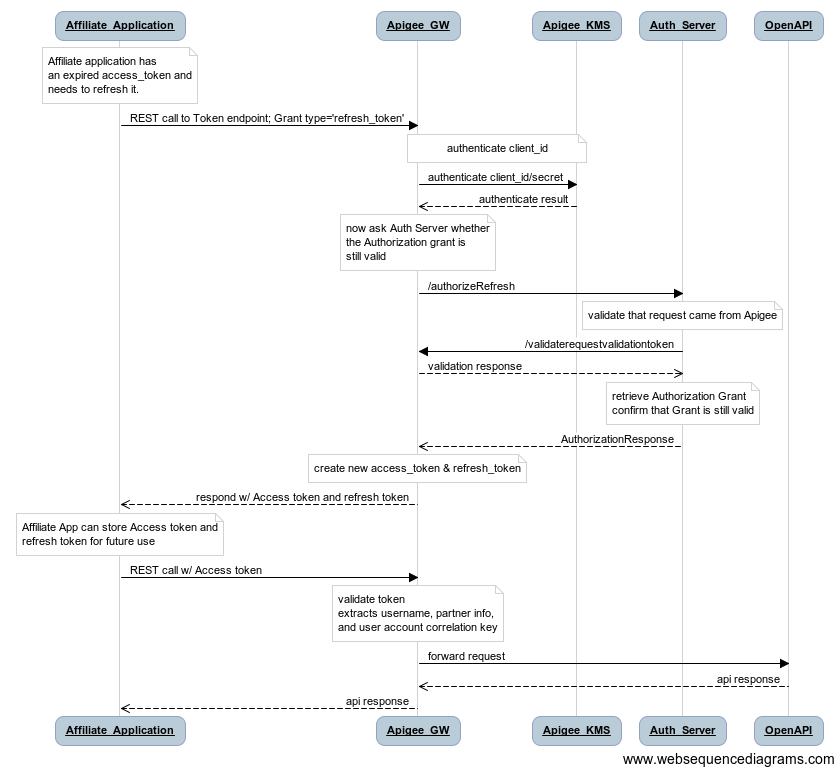
### SAML Bearer



### Client\_Credentials



### Refresh Token



# Operation Descriptions for OAuth Implementation

This section documents the operations that implement the OAuth protocol.

***These descriptions should be used as a conceptual reference, not a mandatory definition of how the operations should be constructed.***

## Apigee OAuth Operations

These operations are split into two sections. The first describes the OAuth operations that are visible to client applications. The second set describes operations that are visible only to Redbox and are used to coordinate the authentication/authorization process.

### Client-Facing OAuth Operations

These operations are called by client applications and provide the operations that implement OAuth as seen by the client applications.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **HTTP Method** | **URL structure** | **Input** | **Output** |
| OA1 | GET | /oauth2-13/authorize?  response\_type=code  &client\_id={client\_id}  &redirect\_uri={rURL}  &scope={scope}  &state={state} | Response\_type  Client\_id  Redirect\_uri  Scope  State | Authorization Response |
| OA2 | GET | /oauth2-13/authorize?  response\_type=token  &client\_id={client\_id}  &redirect\_uri={rURL}  &scope={scope}  &state={state} | Response\_type  Client\_id  Redirect\_uri  Scope  State | Access Token Response |
| OA3 | POST | /oauth2-13/token | Grant\_type=”authorization\_code”  Code  Redirect\_uri  Client\_id  Client\_secret | AccessToken Response |
| OA4 | POST | /oauth2-13/token | Grant\_type=”urn:ietf:params:oauth:grant-type:saml2-bearer"  Assertion  Scope  Client\_id  Client\_secret | AccessToken Response |
| OA5 | POST | /oauth2-13/token | Grant\_type=”client\_credentials”  Scope  Redirect\_uri  Client\_id  Client\_secret | AccessToken Response |
| OA6 | POST | /oauth2-13/token | grant\_type=’refresh\_token’  client\_id  client\_secret  refresh\_token  scope | AccessToken Response |

Operations:

* OA1: Request an Authorization Code
  + URL: /oauth2-13/authorize?

response\_type=code&client\_id={client\_id}&redirect\_uri={rURL}&scope={scope}&state={state}

* + Method: GET
  + Input:
    - response\_type: mandatory; the type of authorization request (must be ‘code’)
    - client\_id: mandatory; the ApiKey of the calling application
    - redirect\_uri: mandatory; URL to which the authorization code is delivered
    - scope: optional; space separated list of scopes
    - state: optional; provided by the calling application and echoed back to the redirect\_uri
  + Output: This operation redirects the request to the Auth\_Service permissions page
  + Description: This request is issued by an application when it needs to obtain an authorization\_code.
  + Security: The caller will provide a client\_id which will be validated by Apigee.
  + Caching Control: N/A
  + Pseudo-Code of Operation:

Validate client\_id; // call KMS

Validate that redirect\_uri matches that registered in KMS

IF (client\_id is not valid or redirect\_uri does not match registered URL)

THEN

Return 401 Unauthorized

Error message should indicate Invalid client\_id configuration

Validate client\_id is allowed to use ‘Authorization\_Code’ profile

Validate scope is subset of KMS configured scopes

IF (client\_id is not allowed to use ‘Authorization\_Code’)

THEN

Issue 302 redirect, error=unauthorized\_client per Section 4.1.2.1 Error Response of OAuth2.0 Draft 13

IF (Requested scopes not a subset of KMS configured scopes)

THEN

Issue 302 redirect, error=invalid\_scope per Section 4.1.2.1 Error Response of OAuth2.0 Draft 13

// user-privileged scopes might change over time, but for first release will be

// PROFILE and RENTALHISTORY

IF (scope contains one or more of user-privileged scopes)

THEN

// Create one-time use RequestValidationToken based upon client\_id

rvt = new RequestValidationToken(client\_id);

// Cache the token for later retrieval

cache = Get-Cache(RequestValidationTokensCache);

cache.Add(rvt);

// Construct redirection URL to Authorization Service

url = AUTH\_SVC\_BASE\_URI

+ “/permission?”

+ “client\_id=” + client\_id

+ “&redirect\_uri=” + redirect\_uri

+ “&scope=” + scope

+ “&state=” + state

+ “&request\_type=” response\_type

+ “&requestvalidationtoken=” + rvt

Issue 302 Redirect to url

ELSE

// Since the requested scopes don’t require user permission, go ahead and issue an auth\_code

// Create an AuthorizationResult

authResult = new AuthorizationResult();

authResult.Authorization.authorizationEffectiveInstant = Now();

authResult.Authorization.scopesAuthorized = scope;

authResult.Authorization.tokenLifetime = default token lifetime (as configured in KMS);

authResult.Authorization.tokenExpiresOn = Now() + .tokenLifetime;

authResult.Authorization.refreshAllowed = default as configured in KMS;

authResult.Authorization.refreshCount = default as configured in KMS;

authResult.Authorization.refreshTokenLifetime = default as configured in KMS;

authCode = new AuthCode();

//Cache the authCode for later one-time use; cache the authResult using the authCode as the cache-key;

cache = Get-Cache(AUTH\_CODE\_CACHE);

cache.Add(authCode, authResult);

// Construct redirection URL to client application

url = redirect\_uri

+ “?code=” + authCode

+ “&state=” + state

Issue 302 Redirect to url

* OA2: Request an Implicit Token
  + URL: /oauth2-13/authorize?

response\_type=token&client\_id={client\_id}&redirect\_uri={rURL}&scope={scope}&state={state}

* + Method: GET
  + Input:
    - response\_type: mandatory; the type of authorization request (must be ‘token)
    - client\_id: mandatory; the ApiKey of the calling application
    - redirect\_uri: mandatory; URL to which the authorization code is delivered
    - scope: optional; space separated list of scopes
    - state: optional; provided by the calling application and echoed back to the redirect\_uri
  + Output: This operation redirects the request to the Auth\_Service permissions page
  + Description: This request is issued by an application when it needs to obtain an access token using the Implicit profile.
  + Security: The caller will provide a client\_id which will be validated by Apigee.
  + Caching Control: N/A
  + Pseudo-Code of Operation:

Validate client\_id; // call KMS

Validate that redirect\_uri matches that registered in KMS

IF (client\_id is not valid or redirect\_uri does not match registered URL)

THEN

Return 401 Unauthorized

Error message should indicate Invalid client\_id configuration

Validate client\_id is allowed to use ‘token’ profile

Validate scope is subset of KMS configured scopes

IF (client\_id is not allowed to use ‘token’)

THEN

Issue 302 redirect with error=unauthorized\_client per Section 4.2.2.1 Error Response of OAuth2.0 Draft 13

IF (Requested scopes not a subset of KMS configured scopes)

THEN

Issue 302 with error=invalid\_scope //per Sec 4.2.2.1

// Create one-time use RequestValidationToken based upon client\_id

rvt = new RequestValidationToken(client\_id);

// Cache the token for later retrieval

cache = Get-Cache(RequestValidationTokensCache);

cache.Add(rvt);

// Construct redirection URL to Authorization Service

url = AUTH\_SVC\_BASE\_URI

+ “/permission?”

+ “client\_id=” + client\_id

+ “&redirect\_uri=” + redirect\_uri

+ “&scope=” + scope

+ “&state=” + state

+ “&request\_type=” response\_type

+ “&requestvalidationtoken=” + rvt

* OA3: Obtain a token from an authorization code
  + URL: /oauth2-13/token
  + Method: POST
  + Input:
    - grant\_type: mandatory; the type of grant included in the request. Must be ”authorization\_code”
    - code: mandatory; string; must match a previously issued but not yet claimed authorization code
    - redirect\_uri: mandatory; URL to which the access token is delivered
    - client\_id: mandatory; the ApiKey of the calling application
    - client\_secret : mandatory; the password for the client\_id
  + Output: An access token (per Section 4.1.4 of OAuth2.0 specification)
  + Description: This request is issued by an application to exchange a SAML assertion for an access token.
  + Security: The caller will provide a client\_id and secret which will be validated by Apigee.
  + Caching Control: no-store
  + Pseudo-Code of Operation:

Validate client\_id and client\_secret; // call KMS

Validate that redirect\_uri matches that registered in KMS

IF (client\_id is not valid or redirect\_uri does not match registered URL)

THEN

Return 400, with error=invalid\_client //per Sec 5.2

Error message should indicate Invalid client\_id configuration

Validate client\_id is allowed to use ‘code’ profile // from configuration in KMS

IF (client\_id is not allowed to use ‘code’)

THEN

Return 400, error=unauthorized\_client // per Sec 5.2

// Confirm that code exists as an unclaimed authorization\_code

authCodecache = Get-Cache(AUTH\_CODE\_CACHE);

IF (authCodecache.Contains(code) )

THEN

// retrieve AuthorizationResult from cache

authResult = authCodecache.Get(code);

authCodecache.Remove(code); // remove the code so that it cannot be used again

IF (authResult.authorized)

THEN // the user authorized this token

// Create a new AccessToken

accessToken = new AccessToken();

// Update the Authorization Result by computing the Token Expire Instant

expiresOn = authResult.TokenLifetime + Now();

authResult.Authorization.tokenExpiresOn = expiresOn;

// Cache the Authorization Result using the Access Token as the cache-key

accessTokenCache = Get-Cache(ACCESSTOKENCACHE);

accessTokenCache.Add(accessToken, authResult);

//if refresh allowed, create refresh token; cache it

IF (authResult.refreshAllowed)

THEN

refreshToken = new RefreshToken();

// Cache a copy of the AuthResult with the refreshToken as the cache-key

refreshTokenCache = Get-Cache(REFRESHTOKENCACHE);

refreshTokenCache.Add(refreshToken, authResult);

ELSE

refreshToken = NULL;

// Now return the result as form-encoded payload

access\_token = accessToken;

token\_type = ‘bearer’;

expires\_in = authResult.TokenLifetime.AsSeconds();

refresh\_token = refreshToken;

Return HTTP Status Code 200 with access\_token, token\_type, expires\_in, scope; optionally include refresh\_token if it is NOT NULL; //as json per Sec 5.2

ELSE // user did NOT authorize this token

Return HTTP Status Code 400 with error message per Section 5.2 of OAuth 2.0 spec

ELSE // authCode not found in cache

Return HTTP Status Code 400 with error=invalid\_grant message per Section 5.2 of OAuth 2.0 spec

* OA4: Obtain a token from SAML assertion
  + URL: /oauth2-13/token
  + Method: POST
  + Input:
    - grant\_type: mandatory; the type of grant included in the request. Must be "urn:ietf:params:oauth:grant-type:saml2-bearer"
    - assertion: mandatory; string; an XML SAML assertion
    - scope: optional; space separated list of scopes
    - redirect\_uri: mandatory; URL to which the access token is delivered
    - client\_id: mandatory; the ApiKey of the calling application
    - client\_secret : mandatory; the password for the client\_id
  + Output: An access token (per Section 4.1.4 of OAuth2.0 specification)
  + Description: This request is issued by an application to exchange an authorization\_code for an access token.
  + Security: The caller will provide a client\_id and secret which will be validated by Apigee.
  + Caching Control: no-store
  + Pseudo-Code of Operation:

Validate client\_id and client\_secret; // call KMS

Validate that redirect\_uri matches that registered in KMS

IF (client\_id is not valid or redirect\_uri does not match registered URL)

THEN

Return 401 Unauthorized

Error message should indicate Invalid client\_id configuration

Validate client\_id is allowed to use ‘SAML’ profile // from configuration in KMS

IF (client\_id is not allowed to use ‘SAML’ OR

Requested scopes not a subset of KMS configured scopes)

THEN

Issue 302 redirect per Section 4.2.2.1 Error Response of OAuth2.0 Draft 13

rvt = new RequestValidationToken();

// cache the rvt for future retrieval

rvtCache = Get-Cache(REQUESTVALIDATIONTOKENCACHE);

rvtCache.Add(rvt, client\_id);

// Call out to Redbox AuthService to validate the SAML token; returns an AuthResult

authResult = Call AuthService/AuthorizeSAML(assertion, client\_id, scope, rvt);

IF (authResult.authorized )

THEN

// Create a new AccessToken

accessToken = new AccessToken();

// Update the Authorization Result by computing the Token Expire Instant

expiresOn = authResult.Authorization.TokenLifetime + Now();

authResult.Authorization.tokenExpiresOn = expiresOn;

// Cache the Authorization Result using the Access Token as the cache-key

accessTokenCache = Get-Cache(ACCESSTOKENCACHE);

accessTokenCache.Add(accessToken, authResult);

//if refresh allowed, create refresh token; cache it

IF (authResult.Authorization.refreshAllowed)

THEN

refreshToken = new RefreshToken();

// Cache a copy of the AuthResult with the refreshToken as the cache-key

refreshTokenCache = Get-Cache(REFRESHTOKENCACHE);

refreshTokenCache.Add(refreshToken, authResult);

ELSE

refreshToken = NULL;

// Now return the result as form-encoded payload

access\_token = accessToken;

token\_type = ‘bearer’;

expires\_in = authResult.Authorization.tokenLifetime.AsSeconds();

refresh\_token = refreshToken;

RETURN HTTP Status Code 200 with access\_token, token\_type, expires\_in; optionally include refresh\_token if it is NOT NULL;

ELSE // AuthService indicated that the SAML assertion is not authorized

RETURN HTTP Status Code 400 with error message per Section 5.2 of OAuth 2.0 spec

* OA5: Obtain an access token from client credentials only
  + URL: /oauth2-13/token
  + Method: POST
  + Input:
    - grant\_type: mandatory; the type of grant included in the request. Must be ”client\_credentials"
    - scope: optional; space separated list of scopes
    - redirect\_uri: mandatory; URL to which the access token is delivered
    - client\_id: mandatory; the ApiKey of the calling application
    - client\_secret : mandatory; the password for the client\_id
  + Output: An access token (per Section 4.1.4 of OAuth2.0 specification)
  + Description: This request is issued by an application to obtain an access token based solely on client credentails.
  + Security: The caller will provide a client\_id and secret which will be validated by Apigee.
  + Caching Control: no-store
  + Pseudo-Code of Operation:

Validate client\_id and client\_secret; // call KMS

Validate that redirect\_uri matches that registered in KMS

IF (client\_id is not valid or redirect\_uri does not match registered URL)

THEN

LOG WARN

Return 400

Error=invalid\_client //per Section 5.2 Error Response portion of OAuth spec

Validate client\_id is allowed to use ‘client\_credentials’ profile // from configuration in KMS

IF (client\_id is not allowed to use ‘client\_credentials’)

THEN

LOG WARN

Return 400, error=unauthorized\_client //per Sec 5.2

Validate that scope is a subset of KMS configured scopes // from configuration in KMS

IF (Requested scopes not a subset of KMS configured scopes)

THEN

LOG WARN

Return 400 – error=invalid\_scope // per Sec 5.2

// Create a new Authorization Result

authResult = new AuthResult();

authResult.Authorization.authorized = true;

authResult.Authorization.authorizedBy = “policy”;

authResult.Authorization.authorizationEffectiveInstant = Now();

authResult.Authorization.scopesAuthorized = scope;

authResult.Authorization.tokenLifetime = (token duration as configured in KMS);

authResult.Authorization.tokenExpiresOn = Now() + .tokenLifetime;

authResult.Authorization.refreshAllowed = false;

authResult.AuthorizationContext.requestingClient\_id = client\_id;

authResult.AuthorizationContext.redirect\_uri = redirect\_uri;

authResult.AuthorizationContext.requestedTokenType = ‘client\_credentials’;

// Create a new AccessToken

accessToken = new AccessToken();

// Update the Authorization Result by computing the Token Expire Instant

expiresOn = authResult.Authorization.tokenLifetime + Now();

authResult.Authorization.tokenExpiresOn = expiresOn;

// Cache the Authorization Result using the Access Token as the cache-key

accessTokenCache = Get-Cache(ACCESSTOKENCACHE);

accessTokenCache.Add(accessToken, authResult);

//if refresh allowed, create refresh token; cache it

IF (authResult.refreshAllowed)

THEN

refreshToken = new RefreshToken();

// Cache a copy of the AuthResult with the refreshToken as the cache-key

refreshTokenCache = Get-Cache(REFRESHTOKENCACHE);

refreshTokenCache.Add(refreshToken, authResult);

ELSE

refreshToken = NULL;

access\_token = accessToken;

token\_type = ‘bearer’;

expires\_in = authResult.Authorization.tokenLifetime.AsSeconds();

refresh\_token = refreshToken;

// per Sec 5.1, return the token information as json payload

RETURN HTTP Status Code 200 with access\_token, token\_type, expires\_in; scope, optionally include refresh\_token if it is NOT NULL;

* OA6: Obtain an access token from a refresh token
  + URL: /oauth2-13/token
  + Method: POST
  + Input:
    - grant\_type: mandatory; the type of grant included in the request. Must be ”refresh\_token"
    - refresh\_token: mandatory; string; the refresh token
    - scope: optional; space separated list of scopes
    - client\_id: mandatory; the ApiKey of the calling application
    - client\_secret : mandatory; the password for the client\_id
  + Output: An access token (per Section 4.1.4 of OAuth2.0 specification)
  + Description: This request is issued by an application to obtain an access token by providing a refresh token.
  + Security: The caller will provide a client\_id and secret which will be validated by Apigee.
  + Caching Control: no-store
  + Pseudo-Code of Operation:

Validate client\_id and client\_secret; // call KMS

IF (client\_id/secret are invalid)

THEN

Return HTTP 400 error=invalid\_client per Section 5.2 of OAuth2.0 spec

// Retrieve the AuthorizationResult from the RefreshToken cache….

// using the refresh\_token as the cache key

refreshCache = Get-Cache(REFRESHTOKENCACHE);

IF (refreshCache.Contains(refresh\_token) )

THEN

authResult = refreshCache.Get(refresh\_token);

refreshCache.Remove(refresh\_token); // remove it so no one else may use it

// Confirm that refresh\_allowed is TRUE && refreshCount > 0

IF (authResult.Authorization.refreshAllowed = false OR authResult.Authorization.refreshCount = 0)

THEN

Return HTTP 400 error=invalid\_grant per Section 5.2 of OAuth2.0 spec

rvt = new RequestValidationToken();

// cache the rvt for future retrieval

rvtCache = Get-Cache(REQUESTVALIDATIONTOKENCACHE);

rvtCache.Add(rvt, client\_id);

// CALL AuthServer to confirm that authorization is still valid

CALL newResult = AuthServer/authorizeRefresh (AS4)

IF (newResult.Authorization.authorized = false)

THEN

Return HTTP 400 error=invalid\_grant per Section 5.2 of OAuth2.0 spec

// Decrement refreshCount

authResult.Authorization.refreshCount -= 1

// Create a new Access Token

accessToken = new AccessToken();

// Update the Authorization Result by computing the Token Expire Instant

expiresOn = authResult.Authorization.tokenLifetime + Now();

authResult.Authorization.tokenExpiresOn = expiresOn;

// Cache the Authorization Result using the Access Token as the cache-key

accessTokenCache = Get-Cache(ACCESSTOKENCACHE);

accessTokenCache.Add(accessToken, authResult);

// Create a new refresh Token

IF (authResult.Authorization.refreshCount > 0)

THEN

refreshToken = new RefreshToken();

// Cache a copy of the AuthResult with the refreshToken as the cache-key

refreshTokenCache = Get-Cache(REFRESHTOKENCACHE);

refreshTokenCache.Add(refreshToken, authResult);

ELSE

refreshToken = NULL;

access\_token = accessToken;

token\_type = ‘bearer’;

expires\_in = authResult.Authorization.tokenLifetime.AsSeconds();

refresh\_token = refreshToken;

// per Sec 5.1, return the token information as json payload

RETURN HTTP Status Code 200 with access\_token, token\_type, expires\_in; scope, optionally include refresh\_token if it is NOT NULL;

ELSE

// refresh token provided is not found

Return HTTP 400 error=invalid\_grant per Section 5.2 of OAuth2.0 spec

### Auth\_Server Facing Operations

These Apigee operations are called only by the Redbox Auth\_Service:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **HTTP Method** | **URL structure** | **Input** | **Output** |
| AV1 | GET | /oauth2-13/validaterequestvalidationtoken?  requestvalidationtoken={rvt}.  &requestingclient\_id={rcid}  &client\_id={client\_id}  &client\_secret={client\_secret} | RequestValidationToken  RequestingClientId  Client\_id  Client\_secret | ValidationTokenResponse |
| AV2 | POST | /oauth2-13/accessapproval | client\_id  client\_secret  authorizationResult | AccessToken Response  OR  AuthCodeReesponse |

Operations:

.

* AV1: Validate a RequestValidationToken
  + URL: /oauth2-13/validaterequestvalidationtoken?

requestvalidationtoken={rvt}&requestingclient\_id={rcid}&client\_id={client\_id}&client\_secret={client\_secret}

* + Method: GET
  + Input:
    - requestvalidationtoken(rvt): Mandatory; string; the code to be validated
    - requestingclient\_id: Mandatory; string; ID of the client that initiated the request
    - client\_id: Mandatory; string; the ID of the Redbox AuthService
    - client\_secret: Mandatory; string; the secret of the Redbox AuthService
  + Output: HTTP status code indicating the result of the validation operation
  + Description: This operation will be called will be called by the Redbox AuthService to confirm that a request it has received from a browser is valid and has been previously processed by Apigee.
  + Security: The caller (Redbox AuthService) will provide its client\_id & secret which will be validated by Apigee. Only Redbox will have this id & secret.
  + Pseudo-Code of Operation:

Validate client\_id & secret;

Validate that caller has authorization to call this operation (only redbox);

IF client\_id/secret NOT valid; return HTTP 401 UnAuthorized

rvtCache = Get-Cache(REQUESTVALIDATIONTOKENCACHE);

IF (rvtCache.Contains(rvt))

THEN

approved\_client = rvtCache.Get(rvt);

IF (approved\_client = requestingclient\_id)

THEN

rvtCache.Remove(rvt); // remove this entry from the cache so that it cannot be used again

Return HTTP Status Code 200

ELSE

Return HTTP Status Code 400 with Error code indicating that requesting client\_id doesn’t match configured client\_id (leave the item in the cache)

ELSE

Return HTTP Status Code 400 with Error code indicating that the RVT is not valid

* AV2: Access Approval
  + URL: /oauth2-13/accessapproval
  + Method: POST
  + Input:
    - client\_id: Mandatory; string; client Id of the caller (should be Redbox AuthService)
    - client\_secret: Mandatory; string; secret of the caller (should be Redbox)
    - authresult: Mandatory; AuthorizationResult
  + Output:
    - AuthCodeResponse OR
    - AccessTokenResponse
  + Description: Redbox will call this operation to inform the gateway the result of the User authorization. If the request was for an auth\_code, then the response will be an AuthCodeResponse. If the request was made using the ‘Implicit’ profile, the response will be an AccessTokenResponse.
  + Security: The caller will provide an ApiKey which will be validated by Apigee.
  + Caching Control: N/A
  + Pseudo-Code of Operation:

Validate client\_id and client\_secret;

Validate that caller has authorization to call this operation (via KMS);

IF (client\_id is not valid OR client\_id is not allowed to call this operation)

THEN

Return HTTP Status Code 400 UnAuthorized

IF (authresult.AuthorizationContext.requestedTokenType = ‘code’)

THEN

authCode = new AuthCode();

//Cache the authCode for later one-time use; cache the authResult using the authCode as the cache-key;

cache = Get-Cache(AUTH\_CODE\_CACHE);

cache.Add(authCode, authResult);

Return HTTP Status Code 200 with:

authCode as payload in form-encoded format

ELSE IF (authresult.AuthorizationContext.requestedTokenType = ‘token’)

THEN

// Create a new AccessToken

accessToken = new AccessToken();

// Update the Authorization Result by computing the Token Expire Instant

expiresOn = authResult.Authorization.tokenLifetime + Now();

authResult.ExpiresOn = expiresOn;

// Cache the Authorization Result using the Access Token as the cache-key

accessTokenCache = Get-Cache(ACCESSTOKENCACHE);

accessTokenCache.Add(accessToken, authResult);

//if refresh allowed, create refresh token; cache it

IF (authResult.refreshAllowed)

THEN

refreshToken = new RefreshToken();

// Cache a copy of the AuthResult with the refreshToken as the cache-key

refreshTokenCache = Get-Cache(REFRESHTOKENCACHE);

refreshTokenCache.Add(refreshToken, authResult);

ELSE

refreshToken = NULL;

// Now return the result as form-encoded payload

access\_token = accessToken;

token\_type = ‘bearer’;

expires\_in = authResult.Authorization.tokenLifetime.AsSeconds();

refresh\_token = refreshToken;

RETURN HTTP Status Code 200 with access\_token, token\_type, expires\_in; optionally include refresh\_token if it is NOT NULL; in form-encoded format

// payload should be formatted in json per Sec 5.1 of OAuth 2.0 spec

ELSE

Return 400 – Unrecognized token type

## Redbox Auth\_Server Operations

These operations will be built by and execute on the Redbox Authentication servers.

The Authorization Service will expose the following operations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **HTTP Method** | **URL structure** | **Input** | **Output** |
| AS1 | GET | /v1/permission?  client\_id={client\_id}  &redirect\_uri={rurl}  &scope={scope}  &state={state)  &request\_type={type}  &requestvalidationtoken={rvt} | Client\_id  Redirect\_URI  Scope  State  Request\_Type  RequestValidationToken | Permission Form |
| AS2 | POST | /v1/permission | Permission Form:   * Client\_id * Request\_Type * Permission\_Result * Redirect\_URI * Scope * State * Timestamp * Anti-request\_forgery\_token * Form\_Signature | AuthCode OR  AccessToken |
| AS3 | POST | /v1/authorizeSAML | Client\_id  Scope  Assertion  RequestValidationToken | AuthorizationResult |
| AS4 | GET | /v1/authorizeRefresh?  client\_id={client\_id}  &customer\_number={customernum}  &scope={scope}  &requestvalidationtoken={rvt} | Client\_id  Customer\_Number  Scope  ReqeustValidationToken | AuthorizationResult |

Operations:

* AS1: Get the Permissions Page
  + URL: /v1/permission?

client\_id={client\_id}&redirect\_uri={rurl}&scope={scope}&state={state) &request\_type={type}&requestvalidationtoken={rvt}

* + Method: GET
  + Input:
    - client\_id: Mandatory; string; the id of the client application
    - redirect\_uri: Mandatory; string; the URL of the client application
    - scope: Optional; string; space separated list of scopes
    - state: Optional; string; opaque string from client app to be echoed back
    - request\_type: Mandatory; string (either ‘code’ or ‘token’)
    - requestvalidationtoken: Mandatory; string; used to verify that the request is coming from Apigee
  + Output: Permissions Web Form
  + Description: This operation will be called by the browser when the user needs to be authenticated and have the user authorize the client application.
  + Security: This URL will be protected by WIF, causing WS-Fed to authenticate the user. The User’s identity (CustomerNumber) will be available in the IPrincipal when this page is called.
  + Pseudo-Code of Operation:

Validaition = Make REST WS call to Apigee AV1 (requestvalidationtoken, requestingclient\_id=client\_id, and the client\_id & secret of the Permissions Page)

IF (validation = 400)

THEN

Return 400 – Unrecognized request

UserId = Page.User.Name;

App = db.FindApplicaionByClientId(client\_id);

UserGrant = db.FindGrant(UserId, App);

userGrantExists = false;

IF (UserGrant != NULL && UserGrant.Approved == true && UserGrant.ExpiresOn > Now())

THEN

userGrantExists = true;

IF (App.IsExemptFromGrant OR userGrantExists)

THEN

IssueToken(); // Common routine, see below

ELSE

Timestamp = Now();

FormSignature = SHA256 Hash of (Client\_id, Request\_Type, Redirect\_URI, Scope, State, Timestamp);

Anti-RequestForgeryToken = provided by ASP.NET MVC

Render Permission View();

**Common Routine: IssueToken()**

public void IssueToken(App, userId, request\_type, scope, state, authorization)

{

// Create New AuthorizationResult

authResult = new AuthorizationResult();

authResult.Authorization.authorized = authorization;

etc…

// Send result to Apigee operation: AV2

result = Call REST WS to AV2

// If request\_type = code, render result as JSON result

// Per Section 4.1.4 of OAuth2.0 spec Draft 13

// IF request\_type = token, render result as HTTP302 to redirect\_uri

// Per Section 4.2.2

* AS2: Process the Permission Form
  + URL: /v1/permission
  + Method: POST
  + Input: The Permission Form includes the following fields
    - client\_id: Mandatory; string; Hidden; the id of the client application
    - redirect\_uri: Mandatory; string; Hidden; the URL of the client application
    - scope: Optional; string; Hidden; space separated list of scopes
    - state: Optional; string; Hidden; opaque string from client app to be echoed back
    - request\_type: Mandatory; string (either ‘code’ or ‘token’); Hidden
    - timestamp: Mandatory; DateTime; Hidden
    - permissionResult: Mandatory; Boolean; the result of the user choice to approve the application;
    - formSignature: Mandatory; string; Hidden
    - Anti-RequestForgeryToken: Mandatory; string; Hidden
  + Output: An Access Token or AuthCode depending upon Request\_Type
  + Description: This operation processes the result of the User’s choice whether to approve the application’s access to his/her Redbox account.
  + Security: The Anti-RequestForgeryToken is used by MVC to prevent a form being cross-posted from a malicious site. The FormSignature will detect whether the original request parameters have been changed.
  + Caching Control: no-store
  + Pseudo-Code of Operation:

// ASP.NET MVC will validate the Anti-RequestForgeryToken before the page loads

// Validate that hidden parameters have not been tampered with by recomputing the form signature.

ThisFormSignature = SHA256 Hash of (Client\_id, Request\_Type, Redirect\_URI, Scope, State, Timestamp);

IF (ThisFormSignature <> formsignature)

THEN

LOG WARN

IF (request\_type = ‘token’)

THEN

Return error code “invalid\_request” in HTTP302

// Per section 4.2.2.1 of OAuth2.0 spec Draft 13

ELSE // request\_type is Code

Return errorcode = “invalid\_request” in HTTP302

// Per section 4.1.2.1

// Validate that the form was rendered recently – to protect against replay

IF (Now() – timestamp > timeLimit) // time limit from configuration

THEN

LOG WARN

IF (request\_type = ‘token’)

THEN

Return error code “invalid\_request” in HTTP302

// Per section 4.2.2.1 of OAuth2.0 spec Draft 13

ELSE // request\_type is Code

Return errorcode = “invalid\_request” in HTTP302

// Per section 4.1.2.1

IF (permissionResult = false)

THEN

LOG INFO

IF (request\_type = ‘token’)

THEN

Return error code “access\_denied” in HTTP302

// Per section 4.2.2.1 of OAuth2.0 spec Draft 13

ELSE // request\_type is Code

Return errorcode = “access\_denied” in HTTP302

// Per section 4.1.2.1

App = db.FindApplicaionByClientId(client\_id);

UserId = Page.User.Name;

grant = new UserGrant(userId, App, permissionResult);

db.SaveGrant(grant);

Audit Grant;

IssueToken();

* AS3: Authorize SAML token
  + URL: /v1/authorizeSAML
  + Method: POST
  + Input :
    - client\_id: Mandatory; string
    - scope: optional;string
    - assertion: Mandatory; string of XML
    - requestValidationToken: Mandatory: string
  + Output: AuthorizationResult
  + Description: Authorization Service will validate the SAML token and return an AuthorizationResult built from it
  + Security: The RequestValidationToken is used to confirm that this request comes from Apigee.
  + Pseudo-Code of Operation:

Validaition = Make REST WS call to Apigee AV1 (requestvalidationtoken, requestingclient\_id=client\_id, and the client\_id & secret of the Permissions Page)

IF (validation = 400)

THEN

Return 400 – Unrecognized request

// Validate SAML request per OAuth SAML Bearer Tokens spec

IF (SAML is INVALID)

THEN

LOG WARN

Return 400

// Create New AuthorizationResult

authResult = new AuthorizationResult();

authResult.Authorization.authorized = true;

authResult.Authorization.authorizedBy = ‘user’;

authResult.Authorization.refreshAllowed = false;

etc….

authResult.AuthorizationContext.UserIdentity.CustomerNumber = subject from SAML token;

// map each attribute assertion in SAML into Claims

etc…

Return HTTP 200 with authResult as payload

* AS4: Authorize refresh
  + URL: /v1/authorizeRefresh
  + Method: GET
  + Input :
    - client\_id: Mandatory; string
    - customer\_number: Mandatory; string
    - scope: optional;string
    - requestValidationToken: Mandatory: string
  + Output: AuthorizationResult
  + Description: Authorization Service will validate that an existing authorization is still valid
  + Security: The RequestValidationToken is used to confirm that this request comes from Apigee.
  + Pseudo-Code of Operation:

Validaition = Make REST WS call to Apigee AV1 (requestvalidationtoken, requestingclient\_id=client\_id, and the client\_id & secret of the Permissions Page)

IF (validation = 400)

THEN

Return 400 – Unrecognized request

// Retrieve User-Grant from repository

Grant = db.FindGrantByUserAndApp(customer\_number, client\_id)

IF (Grant == NULL)

THEN

Bool authorized = false;

ELSE

authorized = Grant.Authorized;

// Create New AuthorizationResult

authResult = new AuthorizationResult();

authResult.Authorization.authorized = authorized;

authResult.Authorization.authorizedBy = ‘user’;

etc….

authResult.AuthorizationContext.UserIdentity.CustomerNumber = customer\_number;

Return HTTP 200 with authResult as payload

## XML Payload Protection

Operations: OA4 & AV2

|  |  |
| --- | --- |
| **XML Payload Protection Settings** |  |
| Max. Node Depth | 15 |
| Max. Attributes Per Element | 25 |
| Max. Namespace Per Element | 10 |
| Max. Sibling Nodes | 50 |
| **Limit Payload Size (Max Characters)** |  |
| Max. Content Text | 150 |
| Max Attribute Value | 150 |
| Max Comment Size | 1 |
| Max CData Size | 1 |
| Max PI (ProcessingInstructions) Size | 50 |
| **Limit on Length of Tag Names (Max Characters)** |  |
| Max Element Name | 50 |
| Max Attribute Name | 50 |
| Max. Namespace Prefix | 50 |
| Max. PI Name | 5 |
| **On Error** |  |
| Send Error Response | Custom |
| Error Message Format |  |
| Error Message | Bad Request |
| Response Code | 400 |

## Audit Policy Configuration

Operations: OA2, OA3, OA4, OA5, OA6

The following default information will be audited for each operation:

* 1. Date time
  2. User identity
  3. Client ip
  4. Request uri.
  5. Application details
  6. Message
     1. Partial – Customer\_number only for OA4
     2. Should never log client\_secret

## Log Policy Configuration

Operations: OA1-OA6, AV1-AV2

All logs should have client\_secret removed.

All operations will have default logging enabled which will log the requests coming into the operations as-is.

The logs will also include the HTTP method used for the operation as well as the HTTP status code returned by the operation.

## Replay Threat Configuration

None.

# Schemas



## AuthorizationResult

Sample:

<?xml version="1.0" encoding="utf-8"?>

<AuthorizationResult xmlns="http://oauth2.redbox.com/v1/Authorization">

  <Authorization authorized="true" authorizedBy="User" authorizationEffectiveInstant="1900-01-01T01:01:01-06:00" scopesAuthorized="scopesAuthorized1" tokenLifetime="P396DT1H1M1S" tokenExpiresOn="1900-01-01T01:01:01-06:00" refreshAllowed="true" refreshCount="0" refreshTokenLifetime="P396DT1H1M1S">

    <Extensions />

  </Authorization>

  <AuthorizationContext scopesRequested="scopesRequested1" requestedTokenType="code" requestingClient\_id="requestingClient\_id1" state="state1" redirect\_uri="redirect\_uri1">

    <UserIdentity>

      <CustomerNumber>CustomerNumber1</CustomerNumber>

      <Claims>

        <Claim name="name1" value="value1" />

        <Claim name="name2" value="value2" />

        <Claim name="name3" value="value3" />

      </Claims>

      <Extensions />

    </UserIdentity>

    <Extensions />

  </AuthorizationContext>

</AuthorizationResult>

Schema:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema id="AuthorizationResult"

targetNamespace="http://oauth2.redbox.com/v1/Authorization"

elementFormDefault="qualified"

xmlns="http://oauth2.redbox.com/v1/Authorization"

xmlns:tns="http://oauth2.redbox.com/v1/Authorization"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

>

<xs:element name="AuthorizationResult" type="tns:AuthorizationResult" />

<xs:complexType name="AuthorizationResult" mixed="false" >

<xs:sequence >

<xs:element name="Authorization" minOccurs="1" maxOccurs="1" nillable="false" type="tns:AuthorizationType" />

<xs:element name="AuthorizationContext" minOccurs="0" maxOccurs="1" nillable="false" type="tns:AuthorizationContextType" />

</xs:sequence>

</xs:complexType>

<!-- This data structure documents the outcome of an authorization decision -->

<xs:complexType name="AuthorizationType" >

<xs:sequence>

<xs:element name="Extensions" type="tns:ExtensionData" minOccurs="0" maxOccurs="1"/>

</xs:sequence>

<!-- This attribute indicates the result of the user's authorization of the application -->

<xs:attribute name="authorized" type="xs:boolean" use="required" />

<!-- This attribute indicates whether the authorization was generated by the user or by configured policy -->

<xs:attribute name="authorizedBy" type="tns:AuthorizedByEnum" />

<!-- The date/time when the authorization was originally granted. -->

<xs:attribute name="authorizationEffectiveInstant" type="xs:dateTime" />

<!-- The set of scopes authorized by the user. -->

<xs:attribute name="scopesAuthorized" type="xs:string" />

<!-- The amount of time a token will be valid. When not provided, Apigee may use the Lifetime configured in KMS. -->

<xs:attribute name="tokenLifetime" type="xs:duration" use="optional" />

<!-- Used by Apigee, this attribute records when the current token expires-->

<xs:attribute name="tokenExpiresOn" type="xs:dateTime" use="optional"  />

<!-- Indicates whether a refresh token is allowed to be granted -->

<xs:attribute name="refreshAllowed" type="xs:boolean" />

<!-- The number of times a refresh token may be granted -->

<xs:attribute name="refreshCount" type="xs:unsignedInt" default="0" />

<!-- The amount of time each token granted via Refresh is allowed to be valid.  When not provided, Apigee may use the token lifetime configured in KMS. -->

<xs:attribute name="refreshTokenLifetime" type="xs:duration" use="optional" />

</xs:complexType>

<!-- This data structure records the input that was used when the Authorization decision was made.  -->

<xs:complexType name="AuthorizationContextType" >

<xs:sequence>

<xs:element name="UserIdentity" minOccurs="0" maxOccurs="1" nillable="false" type="tns:UserIdentityType" />

<xs:element name="Extensions" type="tns:ExtensionData" minOccurs="0" maxOccurs="1"/>

</xs:sequence>

<!-- space delimited list of scopes -->

<xs:attribute name="scopesRequested" type="xs:string" />

<xs:attribute name="requestedTokenType" type="tns:OAuthTokenRequestMethods" />

<!-- The api Key of the requesting application -->

<xs:attribute name="requestingClient\_id" type="xs:string" />

<!-- The application state as provided by the application.  Not used in the authorization process.-->

<xs:attribute name="state" type="xs:string" use="optional"/>

<!-- The redirection URL provided by the application at the time of the request -->

<xs:attribute name="redirect\_uri" type="xs:string" />

</xs:complexType>

<xs:simpleType name="AuthorizedByEnum" >

<xs:restriction base="xs:string" >

<xs:enumeration id="User" value="User" />

<xs:enumeration id="Policy" value="Policy" />

</xs:restriction>

</xs:simpleType>

<xs:simpleType name="OAuthTokenRequestMethods" >

<xs:restriction base="xs:string" >

<xs:enumeration id="code" value="code" />

<xs:enumeration id="SAMLbearer" value="SAMLbearer" />

<xs:enumeration id="token" value="implicit" />

<xs:enumeration id="client\_credentials" value="client\_credentials" />

<xs:enumeration id="password" value="usernamePassword" />

</xs:restriction>

</xs:simpleType>

<!-- This type provides information about the Customer-->

<xs:complexType name="UserIdentityType" >

<xs:sequence >

<!-- The identity of the customer-->

<xs:element name="CustomerNumber" type="xs:string" minOccurs="1" maxOccurs="1" nillable="false" />

<!-- A collection of name/value pairs that describe the customer.-->

<xs:element name="Claims" minOccurs="0" maxOccurs="1">

<xs:complexType >

<xs:sequence minOccurs="0" maxOccurs="unbounded">

<xs:element name="Claim" type="tns:ClaimType" />

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="Extensions" type="tns:ExtensionData" minOccurs="0" maxOccurs="1"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="ClaimType">

<xs:attribute name="name" type="xs:string" />

<xs:attribute name="value" type="xs:string" />

</xs:complexType>

<xs:complexType name="ExtensionData">

<xs:sequence>

<xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any" />

</xs:sequence>

</xs:complexType>

</xs:schema>

## AuthCodeResponse

Sample:

<?xml version="1.0" encoding="utf-8"?>

<AuthCode xmlns="http://oauth2.redbox.com/v1/Authorization">AuthCode1</AuthCode>

Schema:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema id="AuthCodeResponse"

    targetNamespace="http://oauth2.redbox.com/v1/Authorization"

    elementFormDefault="qualified"

    xmlns="http://oauth2.redbox.com/v1/Authorization"

    xmlns:tns="http://oauth2.redbox.com/v1/Authorization"

    xmlns:xs="http://www.w3.org/2001/XMLSchema"

>

<xs:element name="AuthCode" type="tns:AuthCodeResponseType" />

<xs:complexType name="AuthCodeResponseType"  >

<xs:simpleContent >

<xs:extension base="xs:string" >

</xs:extension>

</xs:simpleContent>

</xs:complexType>

</xs:schema>