# Open Services

## 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.

## 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 1 : 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&client\_secret=YYYY on the Apigee Gateway

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

Apigee Gateway forwards call (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=implicit&client\_id=XXX&client\_secret=YYYY on the Apigee Gateway

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

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

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

When: Request is to /authorize?response\_type=code OR to /token?grant\_type=implicit & Apigee Gateway forwards call to Authorization Service/authorize

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 forwards 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 forwards 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 forwards call to Authorization Service/authorize with response\_type != implicit

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 forwards call to Authorization Service/authorize with response\_type = implicit

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: 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 and provides client\_id & client\_secret as query string params

Then: Apigee Gateway authenticates client\_id & client\_secret before processing the request

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

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 to Inventory or Customer

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=http://oauth.net/grant\_type/saml/2.0/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 and provides client\_id & client\_secret as query string params

Then: Apigee Gateway authenticates client\_id & client\_secret before processing the request

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

Apigee Gateway forwards request to Authorization Service/BatchAuthorize?client\_id=XXXX

Given: Client does not have an access token & client has called Apigee Gateway/token?grant\_type=client\_credentials

When: Apigee Gateway forwards request to Authorization Service

Then: Authorization Service validates that client is enabled for batch access

Authorization Service informs Apigee Gateway of scopes allowed (Apigee Gateway issues access\_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 Flow

### Overview

A sequence diagram at the end of this section is intended to illustrate the steps described below.

### Client Application Requests User Information

When a user first browses to a client application, the application will only be able to invoke calls to OpenAPI that are not scoped to a user. For example, requesting movie descriptions or locating nearby kiosks. Once a call is made that requires user information (or if a user decides to log in directly), the request will be sent to Apigee. (Client application authentication for non-user API operations are authenticated; this document focuses only on the authentication scenario that requires user authentication).

### Apigee Performs Authentication

Apigee will authenticate the client application using their Key Manager using an API key and secret which has been previously provisioned with Apigee and packaged within the HTTP request. If client authentication succeeds, Apigee will determine that a code is not present in the request that represents an authenticated user. As a result, it will forward the request to the passive Federated Authentication endpoint on the Authorization Server.

### Authorization Server Initiates Login Sequence

To initiate user authentication, the Authorization Server will construct an authentication request to the Federation Provider using the passive profile. The authentication request is represented as a URL with a series of query parameters one of which is the return URL. This allows the sequence to eventually return to the Authorization Server since it initiated authentication. This response is returned to Apigee which in turn is returned to the browser in response to the browser’s initial request to log in. The response will be a 302 redirect to the Federation Provider. The request will also include additional information as another query parameter which indicates the realm with which authentication should occur. Since the Federation Provider can delegate authentication to different Identity Providers (which have been previously configured), it needs to know to which Identity Provider it should forward the request. The realm represents the different Identity Providers. For example, different providers could be the digital partner or even Facebook. If this information were not included, ACS would display a page called the home realm discovery page that the user could use to select Redbox; however with this design the request will always be sent to Redbox.

### Federation Provider Performs Authentication

When the request is received at the Federation Provider, it determines that the request does not contain a FedAuth cookie that represents the user’s identity. The request is then sent to the Redbox Identity Provider using another 302 redirect to perform the actual authentication. The Federation Provider also includes another return URL so that control is eventually returned to the Federation Provider at the appropriate point in the sequence.

### Identity Provider Performs Authentication

The Identity Provider receives the request and determines that the authentication request originated with a recognized issuer that has been previously configured. It then looks for another FedAuth cookie to determine if the user has previously logged in. If not, it then looks for a FormsAuth cookie to determine if the user has just returned from the login page. If neither cookie is found, another 302 redirect is used to send the user to the login page. This request stores the entire authentication request within another query parameter.

The login page is shown below. This image was taken directly from the Redbox web site. The register portion of the page is not needed but the page should still maintain a similar look and feel. To render this page the Identity Provider must share assets with Redbox.com such as images, styles and potentially JavaScript. A mechanism for sharing this information hasn’t been determined but it should be generally assumed that the Redbox web site will not be updated to support it with this iteration of development.

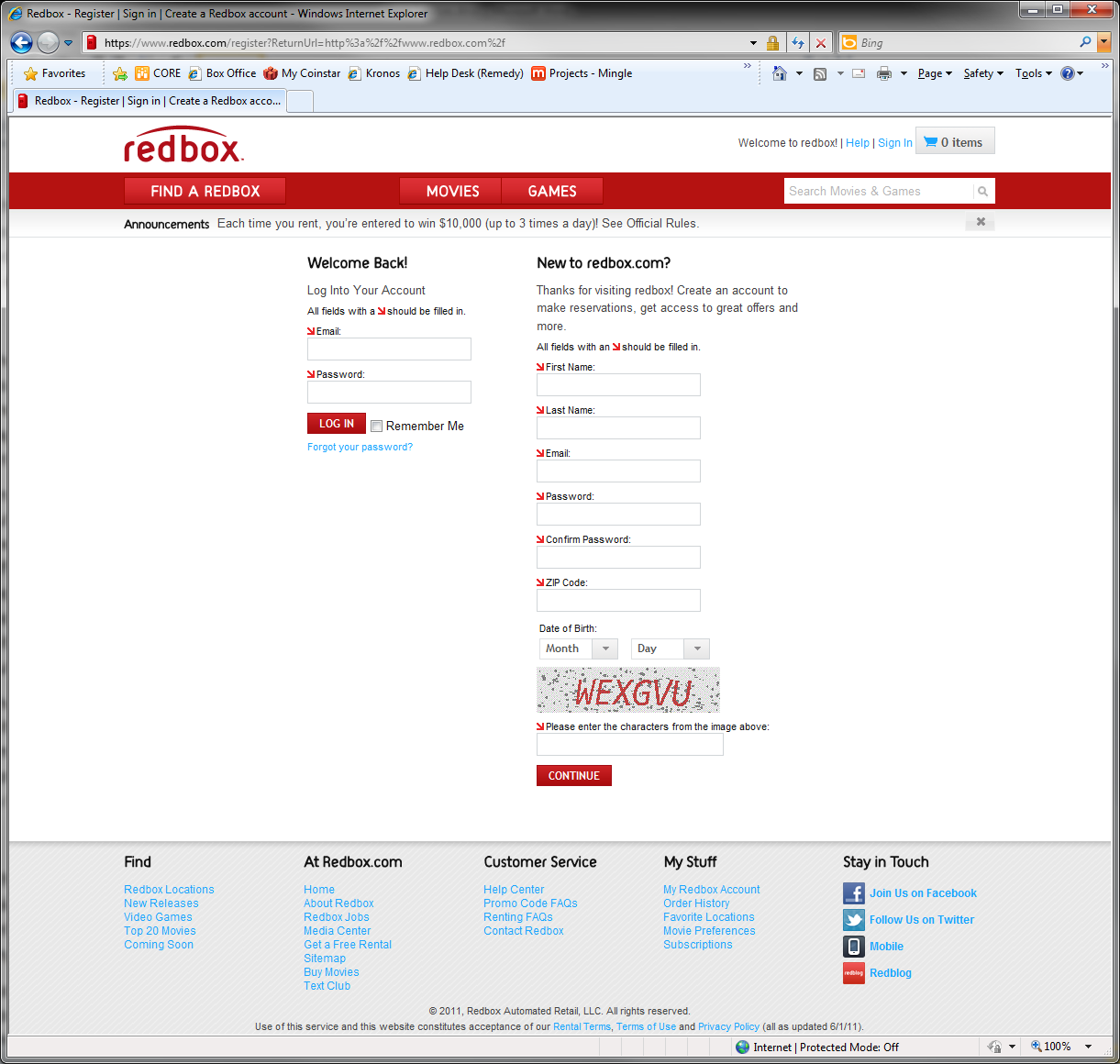


Figure 2: Login Page

Once the user enters their credentials and becomes authenticated, a FormsAuth cookie is created and the entire authentication request is recovered from the query parameter and reissued as a 302 redirect. This request duplicates the authentication request that was originally issued to the Identity Provider from the Federated Provider. The only difference is the referrer which is not observed when the request is processed. When the Identity Provider receives this request again it finds the FormsAuth cookie, generates the FedAuth cookie, deletes the FormsAuth cookie and issues a response to the authentication request. The reason the FormsAuth cookie is deleted is because it may be configured with a different lifetime and session policy; this policy needs to be provided by Federated Authentication not Forms Authentication.

The authentication response is represented as a SAML token which defines the user’s identity and any claims associated with that identity such as a user’s email address, full name, etc. The token is also signed using a certificate’s private key that is managed by the Identity Provider. This time the response is a 200 status that stores the SAML token as a form field (because the allowable length of a URL is not sufficient). The response also includes a JavaScript handler which posts the form data back to the previous provider in the authentication chain. The post mechanism allows a user’s identity to be propagated across domains which is not possible with cookies.

### Federation Provider Processes SAML Token

When the request is received by the Federation Provider it recovers the SAML token from the form and validates the token. The signature is verified using the public key of the certificate provided by the Identity Provider. This validation enforces the trust relationship between the Federation Provider and the Identity Provider. Without this trust it would be impossible to discern whether the request originated with the trusted Identity Provider or a “man in the middle” attack. This also implies that a Security Token Service must make use of a certificate when generating a SAML token. This requirement is enforced by the different SecurityTokenService class constructors available in the WIF API.

Just as the Identity Provider did, the Federation Provider issues a response with a SAML token and a JavaScript induced post to the previous provider in the chain (the Authorization Server). The SAML token is also signed with its own private key that enforces a trust relationship with the Authorization Server. Also, since the Authorization Server is accessed through the Apigee front end layer, the request is actually received by Apigee and forwarded to the Authorization Server.

### Authorization Server Processes SAML Token

The Authorization Server performs a series of steps similar to the Federation Provider. It recovers the SAML token, verifies its signature and generates a FedAuth cookie. Since the Authorization Server will be built using ASP.NET MVC, posts containing XML data are normally rejected. To allow the posting of a SAML token, a custom request validator is needed to verify that the XML actually represents a token. This validator has already been developed for a previous POC and is available for reuse.

### User Grants Access to Client Application

After the Authorization Server processes the SAML token it performs a 302 redirect to the grant page. Just as with the previous step, the request is sent through the Apigee layer and forwarded back to the Authorization Server. The grant page informs the user that the client application they are using requires access to their information and gives the user an opportunity to grant or deny consent.

The grant page is shown below. The page displays the name of the application which is accessible from the Affiliate entity described in the conceptual model section shown later in this document. It also lists the grant scopes along with a description of each; this information is available as a collection of Scopes associated with the Application. The page also has a similar look and feel as the Redbox web site and the login page shown earlier which means that assets need to be shared across three different servers.

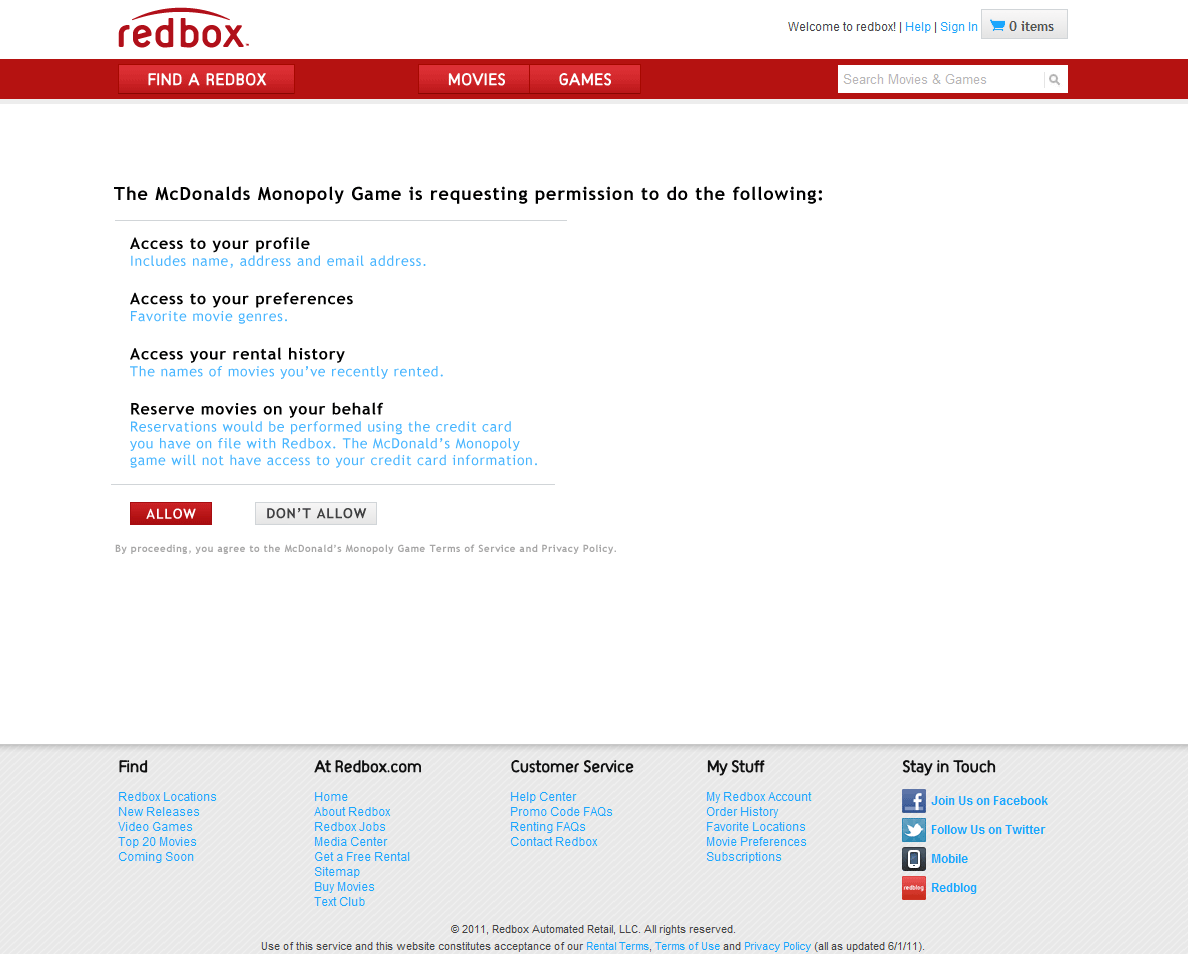


Figure 3: Grant Page

If the user grants consent another series of steps occur. The consents are recorded to a local database and posted to Apigee for storage to support later user authentication. Since both of these operations cross machine boundaries, it is recommended that they be performed in parallel.

The Authorization Server returns control to Apigee by performing a 302 redirect which contains a user id and a grant code which Apigee can use to recover the user’s consents.

### Control Returned to Client Application

Now that Apigee has all the information it needs to authenticate a user it performs a 302 redirect to the Client Application which originally invoked the request to OpenAPI. This request reproduces the original call which now includes a grant code in addition to the client id and user id.

### Apigee Performs Session Based Authentication

Apigee is now able to authenticate the user without the need to forward the request to the Authorization Server. This is true for a user’s entire session (or any session policy alternately defined). It is presumed that a user will fully interact with the Client Application during this session lifetime.

### OpenAPI Processes Request

Once an application and user can be fully authenticated, any OpenAPI requests can be forwarded directly to the OpenAPI server. OpenAPI processes the request and returns the response to the client which is proxied through Apigee.

### Authentication Sequence

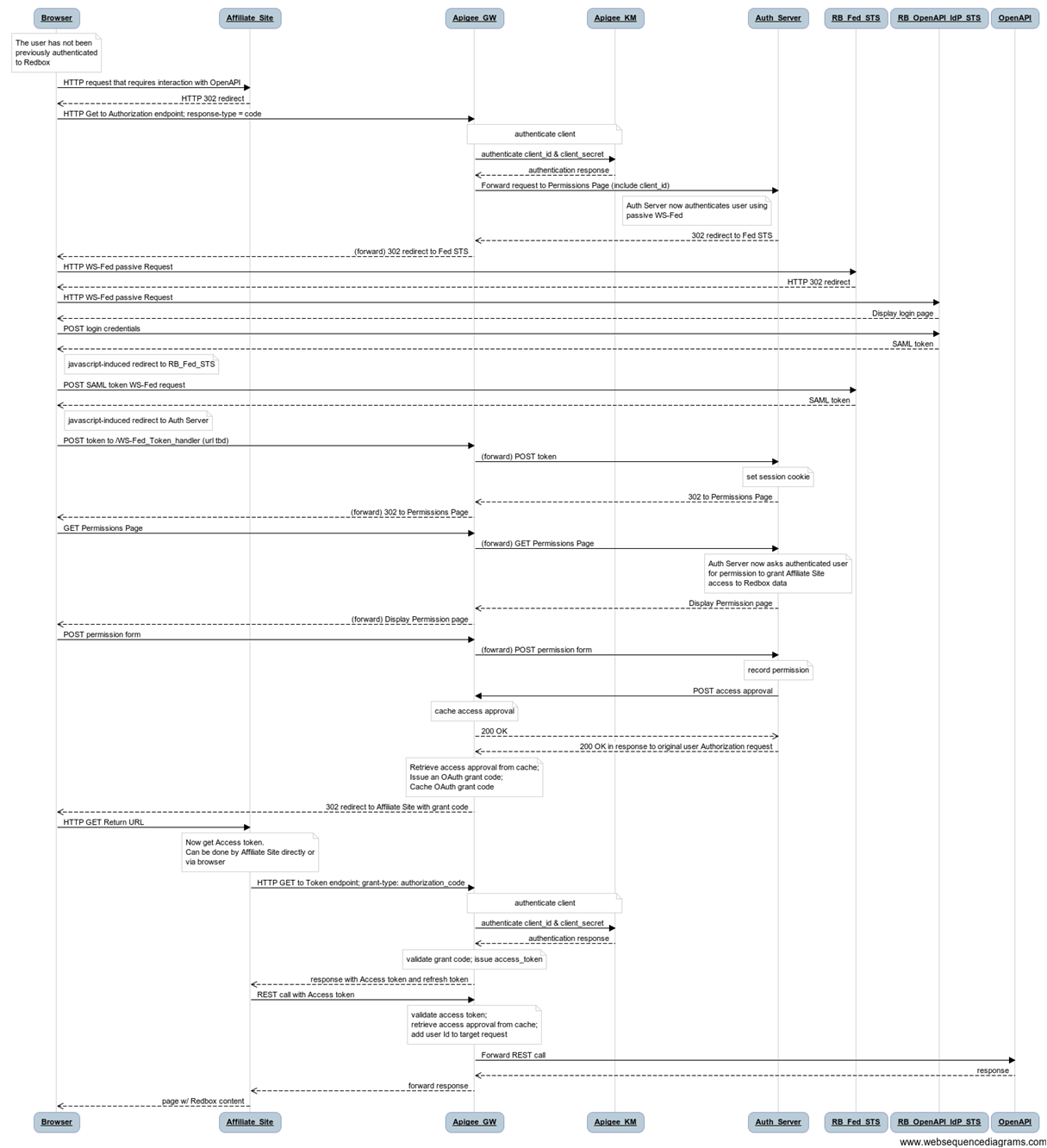


Figure 4: Authentication Sequence

## Data Model

### Conceptual Model



Figure 5: Data Model

### Description

The data model shown above illustrates how the database data will be represented within the Authorization Server’s domain. The model is self-explanatory.

1. An affiliate represents a digital partner along with several attributes.
2. An affiliate may have several applications which are all identified by an application key.
3. The different applications are also scoped to a device type which could be a desktop browser, mobile device, CE device, etc.
4. Applications are associated with a set of consents that they require.
5. Users have the ability to grant or deny the applications access to specific scopes which are all or a subset of the scopes associated with the application.
6. The scopes are attributed with additional information that is used for display on the grant page.

## Database Model

### Description

# The Affiliate table contains all the information that is directly mapped to the Affiliate entity.

# With the exception of collection type properties, the AffiliateApplication contains all the information mapped to an Application entity.

# The Application entity will have two collection properties

# Application Keys are mapped directly the AffiliateApplicationKey table. Each Application Key is mapped to a single Device Type which is stored in the DeviceType table.

# Application Scopes are a collection of Scopes defined by the Scope table using a many to many relationship provided by the AffiliateApplicationScopeMapping table to the Scope table.

# A user (not defined in this data model) will be identified using a customer number indicated in the schema as the UserId. The Identity Provider will issue a WebAccountId claim which will be used to resolve the CustomerNumber.

# A user will be associated with a collection of consents that are scoped to each application which are either granted or denied. The actual consents will be derived through the AffiliateApplicationScopeMapping table to the Scope table. This relationship insures that a user can only grant or deny consent to the scopes that are associated with specific applications. Some users, however, will be explicitly opted in for all scopes for certain applications if the ConsentPreApprovalFlag in the Affiliate table is set to true. In this case these consents will not be available through the UserConsent table, but instead through the AffiliateApplication to Scope mapping.

# The UserConsentAudit table will be maintained by a trigger as changes are made to the UserConsent table. This table is not represented in the data model because the Authorization Server will never need access. It is purely intended for reporting purposes.

# Finally, the ApplicationUserConsentOverrideMapping table insures that users cannot grant consent to scopes for which applications explicitly opt in consents for all users. This will have the side effect of producing an empty collection of user consents for applications that forcibly opt in all users. This behavior needs further consideration. It may be desired in the sense that a user cannot later change their consents to applications using a user profile page. The consents would essentially be displayed (via the Application entity) as read-only consents.

### Schema



Figure 6: Database Schema