

Security SERVICES 3.1

# Document Status

status: Request for Comment (valid values are < Request for Comment, Preliminary Review, Public Review, Architectural Review, Final Review, Published, Deprecated)

This version: **Assembla**.com. Files Tag = CUFX\_3.1\_RFC\_Active

Previous Version: **Assembla**.com. Files Tag = CUFX\_3.0\_RFC\_Archive

# Change Log

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Changes** |
| **2.0.01** |  | * Major draft revision to switch security model to use standards based security mechanisms rather than creating a custom standard. Best practices and examples will be utilized and examples provided on how to use existing industry standards. |
| **2.0.02** |  | * Finished OAuth and added Authorization Use Case Diagram. |
| **2.0.03**  **2.0.04** |  | * Added SAML example * Added OAuth 2.0 and SCIM examples |
| **2.0.05** |  | * Cleaned up formatting and typos * Removed references to session token |
| **2.0.06**  **2.0.07**  **2.0.08**  **2.0.09** |  | * Corrected step reference in use case 2B (Subsequent Interactions…) * Added use case examples with OAuth, and OpenID Connect and SAML * Added symmetric/asymmetric crypto * Removed SAML SSO diagrams. Replaced with OAuth flows * Removed SAML single use case. Replaced AES encryption example with JSON-JS |
| **2.0.10** |  | * Formatting changes, reorganization of the document, miscellaneous updates. |
| **2.0.11** |  | * Removed comment about security token expiration potentially being removed. |
| **2.0.13** |  | * Based on AES common practice, moved the initialization vector from the iv field to the first 16 bytes of the ciphertext field. Added a public certificate for encryption of the AES key on each transmission and response. |
| **2.0.14** |  | * Made both Asymmetric and symmetric AES methods available * Added SAML only SSO example * Removed Error structure and replaced with reference to Error services document. |
| **2.0.15** |  | * Updated the field names and steps for the REST flow and example sections * Added the XML flow sections * Added algorithm used for AES/XML MESSAGE FLOW – ASSYMMETRIC (USED BY SAML) section |
| **2.0.16** |  | * Added AES password iterations password parameter |
| **3.0** | **03/12/2014** | * Versioning and format with release CUFX 3.0 |
| **3.1** | **07/17/2015** | * Updated to release 3.1 |

# Overview of Specification

The CUFX Security Services specification describes the format and methods for secure authentication and general communication security in support of other CUFX specifications such as Personal Financial Management (PFM) and New Member Application (NMA). Version 2.0 represents a major shift from custom security standards to utilizing standards based security models such as SAML, OAuth, and SCIM.

**This document is provided as a starting point/tutorial to reduce the friction of bringing an application online and be secure. It is not meant to be a restrictive document to limit or prescribe specific implementation methods. It is entirely up to the endpoints to determine security requirements.**

# Known Errors in the Document

|  |  |
| --- | --- |
| **Error Description** | Status of Error |
|  |  |

Table of Contents

[Document Status 1](#_Toc369094771)

[Change Log 1](#_Toc369094772)

[Overview of Specification 2](#_Toc369094773)

[Known Errors in the Document 2](#_Toc369094774)

[Document Conventions 4](#_Toc369094775)

[Definitions Related to the Specification 4](#_Toc369094776)

[Security Strategy 6](#_Toc369094777)

[Access and Security Concepts 7](#_Toc369094778)

[Password Encryption 7](#_Toc369094779)

[IP Whitelisting 7](#_Toc369094780)

[Request Signatures 7](#_Toc369094781)

[SSL/TLS 7](#_Toc369094782)

[AES 7](#_Toc369094783)

[Use Standards Based Authorization, Single Sign on and Data Access Prototols 8](#_Toc369094784)

[Other Best Practices 8](#_Toc369094785)

[AES Message Encryption 9](#_Toc369094786)

[AES/REST Message Flow - Assymmetric (JWE Assymetric Method – Example below) 11](#_Toc369094787)

[AES/REST Message Flow – SymMetric (JWE Symmetric Method) 12](#_Toc369094788)

[AES/XML Message Flow – Assymmetric (used By SAML) 13](#_Toc369094789)

[AES/XML Message Flow - symmetric 14](#_Toc369094790)

[Example AES-Asymmetric JSON Request (JWE – JSON Web Encryption) 15](#_Toc369094791)

[SSO Best Practices 16](#_Toc369094792)

[SSO Overview 16](#_Toc369094793)

[OAuth, SAML, OpenID 18](#_Toc369094794)

[OAuth 2.0 Authorization Model Overview 18](#_Toc369094795)

[Openid 2.0 Authentication Overview 19](#_Toc369094796)

[SAML 2.0 SSO Model Overview 20](#_Toc369094797)

[SSO Use Case Overview 20](#_Toc369094798)

[Use case 1: Resource Provider Initiated SSO with OAuth (Authorization Code Grant Type) and OpenID Connect (Authorization Code Grant Type) 21](#_Toc369094799)

[Use case 2: Resource Provider Initiated SSO with OAuth (Authorization Code Grant Type) and SAML 2.0 (Web Browser SSO Profile) 25](#_Toc369094800)

[Use case 3: Client Application Initiated SSO with OAuth (JWT Bearer Token Grant Type) 28](#_Toc369094801)

[Use case 4: Client Application Initiated SSO with OAuth (SAML Bearer Assertion Grant Type) 30](#_Toc369094802)

[Use case 5: OAuth 2.0 Access Control using Client Credentials Grant Type 32](#_Toc369094803)

[Use case 6: SAML: Identity Provider Initiated SSO with REDIRECT 33](#_Toc369094804)

[Manage user identities with SCIM 35](#_Toc369094805)

[SCIM use case 1: First Time Interactions From A CUFX client to Typical SSO App with Client App 35](#_Toc369094806)

[SCIM use case 2: Subsequent Interactions From A CUFX client to SSO App (With Client App Retreiving Security Token) 38](#_Toc369094807)

[SSO User using SCIM idenity mangement 41](#_Toc369094808)

[General Error handling For All Services 44](#_Toc369094809)

[Bibliography 44](#_Toc369094810)

# Document Conventions

“Within this specification, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in W3 Working Group (W3C)]. However, for readability, these words do not appear in all uppercase letters in this specification.

At times, this specification recommends good practice for authors and user agents. These recommendations are not normative and conformance with this specification does not depend on their realization. These recommendations contain the expression "We recommend ...", "This specification recommends ...", or some similar wording.”

All formatting in this document utilize Word Styles.

All Citations must utilize Word Citations to automatically show at the end of the document.

All updates after the initial creation must be performed using Tracking Changes turn on and Accepted by the Architecture committee.

Any Quotes in code examples below should typically be coded as ASCII character decimal 034 or 022 hex.

# Definitions Related to the Specification

SEcurity provider

Software, device or other mechanism that aids a process in providing security validation of message requests.

Message

A request to either create, change, read or delete information. The message may or may not contain confidential information.

ENDPOINT

A system, application or data source that either requests or processes messages.

Identify Provider

An endpoint that acts to validate a user’s credentials.

Service Provider

An endpoint that provides a service to an end user or a business process. Typically a service provider is operated by a 3rd party company that enables additional functionality for users.

data provider

Data provider is any Financial Institution system, Core, or other data source from which data is requested and returned.

Client App

Any end point application (for example Online Banking or Mobile Banking), that end users or business processes utilize to manage or view information. With regards to Single-Sign-On, the client app is a financial institution’s application that allows end users to authenticate and gain access to other service provider endpoint applications to use additional capabilities. Many times the client app acts as an identity provider endpoint.

USER

A consumer that is utilizing a client application to manage or view information.

user agent

Software acting on behalf of the user, e.g., the user’s browser.

Financial institution

Any institution such as a credit union, bank or other entity that provides financial services to consumers.

Single-Sign-on (sso)

SSO is the ability for an application to open or communicate with another application for the current user.

Vendor

Any provider of services, products, applications, etc. for a financial institution that aids the institution in servings its consumers

Credentials

The credentials are used as a means to identify a user’s or system’s ability to obtain account information. Username and password are typically used for initial credentials but can be any combination of user credentials required for the data source to authenticate the request and acquire a security token and security token expiration.

Login Token

Represents temporary permission by an SSO application that the user may open the SSO application user interface. A temporary token is created by the SSO application, sent to the calling client application and may be passed for example in the URL that kicks off the SSO application user interface.

Security token

Security token is a constant and unique authentication key created by the register resource/service by the back end data source provider that identifies that a user has been authenticated and has access to the data until the token expires or is deleted. The security token typically contains no human readable content. The security token typically cannot be updated by the client but instead may only be changed by the Security provider.

Security Token Expiration

The Date/Time that security token will expire.

# Security Strategy

For version 2.0 of the specifications, the CUFX security committee has agreed upon a course of action regarding security that will be instrumental in carrying the CUFX project forward. After carefully considering the core mission of CUFX—which is chiefly a push to realize the dream of interoperability—the security committee has come up with a way to

1. Satisfy the need for security
2. Provide the structure for nearly all types of security across a wide range of scenarios
3. Provide a conduit for information that might normally reside in the security context to pass into the message body itself so that additional business rules could use the information as input.

After analyzing the security posture put forward in version 1.0 of the Security specification, the security committee realized we may have unnecessarily boxed-in implementations by recommending things such as HMAC encryption of the URL string, IP whitelisting, or even embedding of the vendor id in the URL string. These things, while in the right spirit of providing solid security, presuppose that implementations are always HTTPs oriented and are not otherwise protected in other ways. Even simply requiring SSL assumes that a given implementation COULD NOT use an SSL offload device. It became clear that we needed to avoid dictating the type of security mechanisms a given implementation needed.

In addition, all services provided in the 1.0 Security specifications were custom and did not utilize common security standards that are already on the market. These common security specifications have worked nearly all the complex scenarios that may exist between multiple systems. Also, a) these security standards are rapidly evolving so it remains difficult for the CUFX team to keep up with the ever evolving security threats and b) many of the standards are being commoditized by 3rd party device and software providers making it more cost efficient to adopt rather than create these services.

An implementation might need information from the security context in order to perform a business rule to facilitate this, a messageContext object that could carry not only the security context but other information as well. Currently, this object would have properties such as userID, usertypeID, vendorID, appID, FIID, etc., that would be conditionally required depending on the nature of the method call or the specific implementation (i.e., GetRates might have userID as optional but ApproveLoan would require userID). CUFX will provide a standard error that can be raised in the event that a particular implementation requires a value in a particular field.

This is a fundamental, almost paradigm-changing shift in the specification. As such the inclusion of the messageContext in the method call, and the rewriting of previously published specifications will be carried out as version 2.0 of the CUFX specification. This security document is crafted to explain the details of the messageContext and will provide several examples of how this object could be used in conjunction with security mechanisms such as OAuth , SAML and SCIM.

# Access and Security Concepts

This section provides an **overview** of the options that can be utilized in the CUFX specifications. However, this **does not** limit, preclude or require these services to be utilized to be compliant with the CUFX specifications. It is entirely up to the vendor and financial institution to follow the practices that are required by security review, risk level associated to the data involved and the regulations that are prescribed for the scenario.

There may not be any legal requirement to use specific security concepts in the final solution, but expectations may be high based on the previous experience and the overall environmental risks at the time of the implementation.

## Password Encryption

Client and Security Services can pass encrypted Login and Password credentials using RSA-2048.

## IP Whitelisting

An IP whitelist may be used to prevent unauthorized access to CUFX services by those who’s IP have not been explicitly whitelisted. The IP whitelist is usually implemented using standard web technologies. Its implementation and management is beyond the scope of this document.

## Request Signatures

Request signatures ensure that the message has not been modified in transit. All requests from the client may be signed using an HMAC-SHA1 signature of the salted request body. Depending upon risk level, the salt length can be determined but a string of no less than 24 characters in length that is shared between the client and Security Services is a good starting point but can be shorter or longer if needed. The signature can be passed as the MAC parameter in the request URL or some other mechanism based on agreement by the two endpoints. If provided, the security provider can perform a compare of the provided MAC parameter against the HMAC-SHA1 signature of the salted request body. It is up to the endpoints to agree upon case or case insensitive compares. If the MAC does not match the request body signature, the standard error returned can be HTTP Error Code 400 Bad Request or something else agreed upon by the endpoints.

## SSL/TLS

A minimum encryption can be performed for all CUFX services using such services as Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL) and require valid certificates to operate.

## AES

AES – Advanced Encryption Standard is a best practice for internet facing transmissions to further encrypt the messages within the TLS/SSL encrypted messages. AES is a symmetric-key encryption algorithm allowing decryption by the receiver. Note: This is a federally mandated standard for U.S. companies that have government contracts.

Symmetric keys present challenges with key management, key distribution, and dynamic in-session key changes. A hybrid crypto system of asymmetric key encryption (public key cryptography) in combination with AES solves this problem. The asymmetric key is used to encrypt the AES key which in turn is used to encrypt the body of the message. AES keys used in this manner should be one-time session keys.

See the [**AES Message Encryption**](#_AES_Message_Encryption_1) section for further details.

## Use Standards Based Authorization, Single Sign on and Data Access Prototols

In the SSO Best Practices section, a number of different recommended SSO standards are described. These standards are available and implemented in many device and software systems and can also be done readily by any software provider of a service. These standards typically can enable the following functions of while giving access to the data sources.

* Creating access tokens
* Verifying
* Hashing
* Checking timestamps
* Linking
* Logging
* Auditing
* Revoking access

Examples of standards available that may be implemented to manage SSO and allowing limited access to private data sources are but not limited to the following:

* OAuth 2.0 \*\*
* OpenID Connect 2.0 \*\*
* SAML 2.0 \*\*
* Simple Cloud Identity Management (SCIM) \*\*
* WS-Trust
* WS-Security
* WS-Federation
* OpenID 2.0 (Note: 2.0 only supports two of the four assurance levels within NIST 800-6).

\*\* Detailed discussions provided below.

## Other Best Practices

* The Data Provider should avoid sending any identifiable account information to the client such as complete account and credit card numbers in account names, ids, and account or transaction descriptions unless the service specifically requires this data.

# AES Message Encryption

AES standard includes 128, 192 or 256 bit encryption. (NIST, 2001), (sources, 2012), (IEEE, 2012). The CUFX specification has determined it is up to the two end-point partners to determine what level of AES encryption is required, but *recommend* 192 bit encryption.  **Note that the** data stream may already be encrypted by SSL/TLS. In addition, the AES cipher algorithm selected can also be selected by the two partners communicating. The *recommended* options are CTR, CBC or CFB. It is *recommended* to use a hybrid crypto system of public key cryptography (asymmetric key encryption) in combination with short lived session AES keys to encrypt the body of request/response messages. Optionally you could use shared AES keys. The below AES descriptions and message flow apply regardless of the choice for AES key implementation. Whether one-time AES session keys encrypted under a public key or pre-arranged shared AES keys are in use, the same guidelines apply. The following definitions must be applied for communication to occur. *Recommend* that AES encryption and decryption of message occur behind firewall.

|  |  |
| --- | --- |
| Public Certificate | **Asymmetric:**  (certificate in the certificate store)  The certificate that is purchase from a certificate authority. Typically, It is stored in the certificate store on the target system. This is used to asymmetrically encrypt the AES Key with the other endpoint’s public key  Since there may be multiple parties involved in CUFX integration projects, AES integration must only be defined between two partners at a time. Each integration pair should use different public certificates. For example, the communication between the CUFX Client App and the CUFX Compliant SSO Application would be one public certificate, and the CUFX Complaint SSO Application would communicate with the CUFX Data Source using a separate public certificate.  **Symmetric:** Not used. |
| AES Key | **Asymmetric:**  (string, required) The AES Key is a value that must be randomly generated per request and first encrypted using the other end-points public certificate. Note: The AES secret key, key length and cipher mode are not shared over the wire during a request/response lifecycle.  **Symmetric:**  (string, shared out-of-band) The AES Key is a value that must be shared out-of-band. Note: The AES secret key, key length and cipher mode are not shared over the wire during a request/response lifecycle. |
| Key Length | 128, 192 or 256 bits. The random keys should be this length. The NIST believes that lengths 192 and 256 will be valid for projects to 2030 and beyond. *Recommend minimum* 192 bit encryption*.* |
| Cipher Mode | Although *recommended,* the CTR algorithm is currently not available on all platforms. The CTR algorithm allows for parallelism and has no feedback so it is less exposed to attack as long as initialization vector is always regenerated. CBC and CFB ciphers are also available. The examples below are based on CBC as the CTR was not yet supported in the libraries used to generate the specification examples. |
| Initialization Vector | Abbreviated as IV. The IV is always 16 bytes, which is encoded as 24 bytes of base64 data. This vector is used in each encrypt/decrypt scenario with AES. It must be shared raw in the request/response lifecycle in order to support AES encryption/decryption. Because it is sharable, it must be regenerated each time an encryption occurs with the AES key.  Sharing the IV during a request cycle occurs in the first 16 bytes of the cipher text. If the IV is not shared when it is expected, an HTTP status code of *400 Bad Request* is returned. |
| Raw REquest/ Response Bodies | The raw request or response body is to be encrypted with random IV. The encrypted data is then encoded via Base64 and placed in the “ciphertext” field of the wrapping body. See below for an example message. **When an error occurs, the body of the response will continue to be encrypted to maintain the highest level of security.** |
| Password iterations | (optional) RFC 2898 –  TREY TO PROVIDE |
| Protected Header | Identifies the algorithms used to encrypt the JSON message. A portion of the protected header is the cipher mode (defined above). The other portion is the algorithm used to encrypt the AES Key in asymmetric. |
| Additional Authenticated Data | “aad” below. Is used to compute an authentication tag that is used for verification (similar to a signature). In the use cases below, it can be the protected header which has been serialized and base64 encoded. |
| Authentication Tag | Is created by concatenating the Additional Authenticated Data, the initialization vector, the encrypted text of the raw request and the length of the additional authenticated data. This is then hashed using HMAC algorithm and that becomes the authentication tag. |
| Encryption Mode | “enc” below. Comprised of two parts. The first part is cipher mode. The second part is the HMAC algorithm used to compute the authentication tag. |

## AES/REST Message Flow - Assymmetric (JWE Assymetric Method – Example below)

–

## AES/REST Message Flow – SymMetric (JWE Symmetric Method)

## AES/XML Message Flow – Assymmetric (used By SAML)

## AES/XML Message Flow - symmetric

## Example AES-Asymmetric JSON Request (JWE – JSON Web Encryption)

The following example is an AES encryption with 192 CBC cipher with the encrypted and Base64 encoded in-session secret key being **AMZ64SYkJ9p10ARl3s3izC3093e6WR9m.** The in-session secret key has been encrypted under a 2048 bit public key of private/public key pair, and the result of that encryption has been Base64 encoded.

The below JSON notation follows the naming convention of the ‘JSON Web Encryption JSON Serialization (JWE-JS)’ working draft document (<http://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-11>). The message syntax is:

{

“protected”: <base64 encoded common header properties>,

"recipients":[ {

"header":"<recipient specific header N contents>",

"encrypted\_key":"<hash and encrypted key N contents>"

}],

"ciphertext":"<ciphertext contents>",

“iv”: “<initialization vector>”,

“aad”: “the additional data used for the hash”,

“tag”: “the hash generated from the add, iv, ciphertext and length of the add.”

}

* A “tag” is generated from the result of concatenating the aad, which is typically base64 encoded protected header object, the unencoded initialization vector, the unencoded encrypted text and the length of the header base64 encoded string. See the link above for the details.

JSON Serialization may be used to send encrypted content to multiple parties. For our purposes, the use case deals with a single recipient. The **header** element describes the encryption algorithm applied to the message content. If all the recipients use the same algorithm, the header would be placed in the **protected** field. The **encrypted\_key** is the AES Key which has been encrypted under the recipient’s public key. The **tag** is the hash generated from the **aad**, **iv**, **ciphertext** and length of the **aad**. The initial encryption random generated input is the **iv**. The **ciphertext** is the encrypted message content.

All of these elements are Base64 encoded. An example is shown below.

In the below example, the header parameter is:

{"alg":"RSA1\_5", "enc":"A256CBC+HS512"}

Where, RSA1\_5 is the algorithm used to encrypt the message encryption key; AES 256 CBC cipher is used to encrypt the message content; and, HMAC calculation is SHA 512. Additional properties can be added to the header dependent upon the agreement of the CUFX end-points.

The recipient of this request would unravel the message by first decrypting the hash key and in-session secret key using their private key of the public/private asymmetric key pair. Once the key has been decrypted, the next step is to follow the specified algorithms as noted in the **header** element to decrypt the message body.

In the below example, the protected header parameter is:

{"alg":"RSA1\_5", "enc":"A256CBC+HS512"}

**POST** [**https://api.vendor.com/user?subMethod=GET&MAC=<MAC**](https://api.vendor.com/user?subMethod=GET&MAC=%3cMAC)**>**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us

X-API-Version: >=1.0.4

{

"protected":"eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMjU2Q0JDLUhTNTEyIn0=",

"recipients":[{

"encrypted\_key":"FKNvhOwIJRg3aUGy29DLVyQ3uyz9ipn2qJjlOOLl6/2fWmJ+fFdVnB9DWWlFGRmiwDX7U+vzgHRJ5CasogD85lwpUCLYWRnd5XL1rNNAguhG0bV210xh3/0+eQP0D0fCNCpdBPiPoXdsSYVpz8Nm/UbvFvjjfrHsTbFX44iFGII="

}],

"ciphertext":" idXs8MBCSbzf7XvxXCF5ZA==S82G/7UzWC7iMKnutoHsIZe/tDXcklVs3ijfa9/Bn+TY3oHJqIqnq+rr",

"iv": "gTuD0KsN/ZMtIBtfj4q7xg==",

"aad": "ZXlKaGJHY2lPaUpTVTBFeFh6VWlMQ0psYm1NaU9pSkJNalUyUTBKRExVaFROVEV5SW4wPQ==",  
 "tag": "Edg/J20WpT+QlXr6S4gjCoN36faDsG54GmmURVIhjho="

}

Note: If a shared secret AES key (one shared by a prior process) is used then communication of that key would obviously not be included. If there is no “**tag**” created, the **aad** would not be required and there would be no element holding the hash key and secret key. In this case, the **encrypted\_key** block would not be included.

Decrypted data element (the body of the POST request) should be:

{"fiUserId":"FI-USER-123","fiId":"FI-123"}

# SSO Best Practices

## SSO Overview

Let’s examine SSO from the perspective of the user. For the user, there are generally three parts to SSO:  
  
 The ability to prove **one identity**...  
 **One time**...  
 To access **multiple systems**.  
  
Let's consider each part in reverse order:

#### Multiple Systems

A CUFX endpoint may be one of the systems that a user might want SSO access to. What follows is a recommendation for how a CUFX endpoint can support SSO from a user's (and user administrator's) perspective based on their preferences and industry standards (namely OAuth with OpenID Connect, SAML and/or WS-Trust).

#### One Time

Once a user has provided their credentials one time to access a system, they do not want to have to provide their credentials again during that session to access that system, or any other system for which those credentials should have access. A CUFX endpoint might accomplish this with essentially two approaches:

1. Use a credential challenge that's transparent to the user on every request.
2. Or after allowing initial access, establish a secure session between that user and the endpoint using a token or session identifier that can be sent by the user with every subsequent request to the endpoint during the secure session.

Though the former approach may be easier for the endpoint, it's less efficient for all involved and it may lead to performance and scalability problems with high volumes of requests. It is also less secure, because it usually means that the user's credentials need to be sent with every request. If a transaction is compromised along with the credentials, the user’s credentials are compromised whereas if the transaction presents a temporary token, the session can be invalidated and the credentials are not compromised.

OAuth is a rapidly maturing and widely adopted API-focused standard that supports the latter approach. Using OAuth, once access has been allowed, the calling application receives an access token that can be used for the remainder of the secure session. When the valid access token is presented to an endpoint, this endpoint authorizes the API call based on the session properties associated with this token. The session associated with the token can include scope and expiry information, and in some cases, the token can be refreshed without having to repeat the initial handshake. OAuth 2.0 defines multiple types of handshakes called *grant types* to address different situations such as trusted or untrusted client apps. In some cases, the user’s credentials are provided directly to the API provider through redirection in order to avoid the user revealing its credentials to the client application.

While OAuth is an ideal fit for the standardize SSO support by CUFX, OAuth does not provide identity federation on its own. For this reason, we recommend OAuth together with other standards such as OpenID Connect or SAML. OAuth can also be extended to persist additional user attributes provided by the identity providers using these standards and needed during the secure session maintained by OAuth.

Take note, when selecting the above second approach it's the responsibility of each endpoint to establish and maintain its own secure session, and when endpoints rely on an identity provider to federate identity as described below, it's the responsibility of the identity provider to establish and maintain session across endpoints (so that when new endpoints are accessed within an established session and those endpoints redirect the user to their identity provider for authentication, the identity provider does not challenge them again for their credentials).

#### One Identity

Finally, let's consider the idea of one identity. From a user's perspective, that might just mean being able to use the same username and password combination to access multiple systems. The user might not know or care whether each system has its own account for that user, or whether multiple systems have direct or indirect access to a shared account. However, identity administrators care (and in some cases, the user may be their own administrator). User administrators prefer to be able to maintain a user's identity in one account, and they want to have one place to create new accounts, change attributes of existing accounts, manage the state of existing accounts, and manage general account policies. They do not want to have to do all these things separately and repetitively in each different system that the user needs access to, and they especially don't want to do some of these things in third party systems over which they have less control.

CUFX endpoints should respect this preference. They should not require users to maintain a separate account with them. Doing so will place an unwanted burden on the user. That will create a barrier to use of the CUFX endpoint, and that is likely counterproductive to the goals of the CUFX endpoint provider. Also, as it turns out, it's much easier to support the "one time" part of SSO when there is one user account rather than each system maintaining its own account for the user.

An identity should therefore exist in one place and be federated. If a requester identity does not exist within the relying CUFX endpoint (i.e., the relying CUFX endpoint cannot directly authenticate the identity against its containing identity store), the CUFX endpoint should know how to redirect that identity to its identity provider for authentication (i.e., service provider initiated SSO) when access is being requester. Also, the CUFX endpoint should know how to accept a valid assertion of that identity from one of its trusted identity providers to allow access (i.e., identity provider initiated SSO).

Which identity providers a CUFX endpoint should trust is a business decision, and it may involve a business process outside the scope of the CUFX security recommendations. However, verifying identity provider trust is the responsibility of each relying CUFX endpoint. Depending on which standards are being used, this is normally done based on x.509 certificates and/or shared secrets used to digitally sign identity assertions. For example, in the case of SAML, the identity provider digitally signs an assertion containing statements about the federated identity. The CUFX relying party verifies the digital signature and validates that the signing certificate is one of possibly many trusted certificates. Shared secrets are used to verify HMAC signatures of REST messages including JSON Web Tokens (returned by OpenID Connect identity providers).

If the CUFX endpoint needs additional attributes of the user, the CUFX endpoint can either require that those attributes are sent together with the identity assertion, or the CUFX endpoint can directly query the identity provider for those attributes. The CUFX endpoint can then persist those attributes as it sees fit for further and future use. The CUFX endpoint should not allow access based solely on those persisted attributes. The CUFX endpoint should always require an identity assertion from a trusted identity provider to allow access (i.e., a user should not be able to directly access the CUFX endpoint with their federated identity without first having that identity authenticated by the trusted identity provider). The CUFX endpoint should also be responsible for maintaining the persisted attributes by updating their values when new identity assertions are received and/or by making new queries for attributes from identity providers. Users should only have to maintain their attributes with the identity provider. Users should not be responsible for maintaining those attributes with every system they have accessed using their federated identity.

## OAuth, SAML, OpenID

In the following sections several SSO use cases are diagrammed that combine OpenID Connect federated identity with OAuth or SAML 2.0 for authorization. Before doing so, a brief overview of OAuth, and Open ID is presented.

## OAuth 2.0 Authorization Model Overview

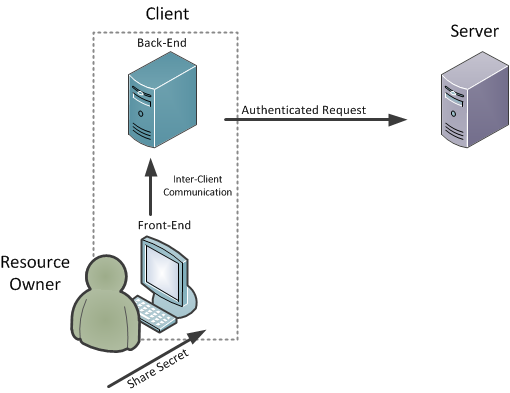
Full OAuth 2.0 Documentation: <http://oauth.net/> and <http://tools.ietf.org/html/draft-ietf-oauth-v2-31>

OAuth 2.0 is a simplification of the OAuth 1.0a specification. The need for complex signatures and multiple tokens is removed in OAuth 2.0. There is only one token and no signatures. OAuth 2.0 requires SSL/TLS across all communication where token generation occurs.

Let’s walk through a fictitious example where Jane (a financial institution account owner) finds a new Personal Finance Manager (PFM) application online and wants to allow it to gain access to her account data so she can better manage her personal budgets. However, she doesn’t feel comfortable providing the PFM application her user credentials to her account information. This is where OAuth steps in because the username and password are never shared with the PFM application. The OAuth process keeps username and password secret by using tokens to represent the user’s agreement to give access to the account information.

The first step is that the PFM application and the online banking application providers must have made a prior agreement to allow the application to gain access to the account data. A shared symmetric secret key that represents this agreement is shared amongst the parties’ out-of-band of this process.

In OAuth terms, the PFM application is referred to as a *client*. The user is referred to as the *resource owner*. The Online Banking application is referred to as the *server or resource provider*. Note that the client can be split into two or more components known as the front-end and back-end, but still acts as a single entity on behalf of the resource owner.



The online banking server contains or has access to *protected resources* such as account information in this case, which requires authentication in order to gain access to it.

## Openid 2.0 Authentication Overview

 OpenID general information: [http://openid.net](http://openid.net/developers/specs/), and <http://openid.net/developers/specs>

OpenID came about in 2005 by an open source community trying to solve a problem that was not easily solved by other existing identity technologies. As such, OpenID is decentralized and not owned by any one group. Today, anyone can choose to use an OpenID or become an OpenID Provider without having to register or be approved by any one organization.

OpenID is not OAuth. OpenID is authentication, OAuth is authorization. You use OpenID to associate one login ID with multiple sites. You create an OpenID identity at an OpenID provider (e.g., myOpenID.com), or the login account with some services (e.g., Google, Yahoo, and others) can serve as an OpenID account where this has been enabled. Your OpenID account is used to sign in into sites accepting OpenID authentication. You can also choose to share or keep private additional information (e.g., name, date of birth, residence, etc.) linked with your OpenID account to avoid repetitive entry of common elements requested at new site registrations.

OpenID Connect is a set of standards based on the Oauth 2.0 protocol to provide federated authentication to OAuth enabled systems. In practice, OpenID Connect functions in the same way as OpenID but does so via REST like API’s.

In the following sections some common SSO flows are presented using OAuth in combination with either OpenID Connect or SAML to provide federated identification. It’s important to note that diagrams demonstrate the use of OpenID Connect versus just OpenID. It is further noted that a CUFX endpoint does not need to be an OpenID Connect provider, but may need to know who to redirect to an OpenID Connect provider to support federated SSO as depicted in use case-1.

## SAML 2.0 SSO Model Overview

Full SAML 2.0a Documentation: <http://saml.xml.org/saml-specifications> and here...

<https://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf>

The most well-known use case (Profile in SAML terms) for SAML is SSO. One concrete example is you have a user of online banking (OLB) who wishes to access a PFM solution using the same credentials used to access the OLB application. In SAML SSO profile jargon, the OLB application will be the IdP (Identity Provider) as it’s storing the real identities, and the PFM application will be the SP (Service Provider) as it’s the provider to the user for the PFM resource. Please note the focus is on user authentication. We are not trying to resolve the problem of application authorization, i.e., authorizing the PFM application to call web services provided by the OLB application. Actually in our case, there is a trust established between the PFM application and OLB application which most likely is setup manually out-of-band.

In our use case, the user first visits the OLB (IdP) site. The resource owner/user enters their credentials and click a link on the OLB page which will redirect the user to the PFM website. In the example presented this is a first time user accessing the PFM site. The OLB (IdP) creates an incomplete SAML assertion and artifact to identify the service provider that the user has been authenticated but does not include all the details for security purposes. The service provider then validates the assertion’s signature. If valid, sends the artifact back to the identity provider to pull the original assertion and additional details. The additional details may contain information for the service provider to create a user as well as access the user’s account details. When sending the artifact back to the IdP, a different channel should be used to provide additional security.

The user will be presented with the PFM content without a new credentials challenge by the PFM application.

## SSO Use Case Overview

There are three different security best practices that are reviewed in detail in this document.

1) How does a client like a 3rd party service provider gain access to a member’s account data but not store any critical security information such as the user’s username and password?

2) How does a client like an Online Banking client application work with a 3rd party service provider application to create a user, update, read or delete a user and then link to that application (i.e., Single Sign-on)? For cases where it may be deemed that creating users in 3rd party application is a necessity, one possible flow using SCIM is diagramed to accomplish this. However, this is strongly discouraged as it is not a best practice to spread user profiles around.

3) How does a web client, service provider application or internal application gain access to a data source?

The SSO examples below show how a user can be created in a remote system if the user doesn’t already exist within an application and then linked through a URL.

The SSO examples are broken into two parts

1) USER portion– to manage the user and the

2) REGISTER portion which allows the application to register a session with that remote application and gain access to it.

The 2nd portion of this document (Manage user identities with SCIM) describes how a system can gain access to a data source via the REGISTER resource for the data source itself.

## Use case 1: Resource Provider Initiated SSO with OAuth (Authorization Code Grant Type) and OpenID Connect (Authorization Code Grant Type)

The below sequence diagram combines OAuth with OpenID Connect to support SSO with federated identification. Before beginning this sequence, to establish mutual trust, the client application would have registered with the CUFX endpoint OAuth authorization server, and in turn, the CUFX endpoint OA would have registered with the Resource Owner's OpenID Connect server.

Note that while OAuth supports different three-legged (where the resource owner is directed to the OAuth authorization server for authentication and authorization) and two-legged (where the client application sends the resource owner's credentials to the OAuth authorization server with implied authorization) grant types, OpenID Connect only support three-legged grant types. This sequence diagram could be modified to support any three-legged or two-legged OAuth grant types for the OAuth access control flow, and any three-legged OAuth grant type for the OpenID Connect identity federation. However, different combinations will make more or less sense in the context of SSO with identity federation.



Let’s walk through the details now.

#### Step 1: The Resource Owner (RO) to access a Resource Provider (RP)

The Resource Owner (RO) uses the client application (CA) to access a Resource Provider (RP) of a resource they own. In this case, the CUFX endpoint is the RP (e.g., user account information)

#### Step 2: client application request Oauth access token

Knowing that an OAuth access token is required to access the protected resource, the CA redirects the RO (e.g., end user or other entity) via another user agent (UA; often a browser) to the CUFX endpoint's OAuth Authorization Server (OA).

#### Step 3: Oauth authorization server (oa) redirects to openID connect server (OC).

Normally, this is when the OA would challenge the RO for credentials that it would authenticate against its own instance of that identity. However, in this example, the OA does not have direct access to that identity, and it will instead rely on federated identification through its trust relationship with the RO's OpenID Connect Server (OC). This step begins the OpenID Connect Identity Federation flow. The OA redirects the RO via the UA to the OC.

#### Step 4: Resource owner credential challenge

The OC challenges the RO via the UA for credentials. This can be done however the OC likes. For example, this could be done using HTTP Basic Auth credentials, Windows Integrated Authentication credentials, or HTML form authentication.

#### Step 5: Resource owner credentials collected

RO credentials sent to the OC via the UA.

#### Step 6: OC (openID conenct server) requests RO for authorization decisison

The OC asks the RO for an authorization decision to grant or deny the OA access to the RO's identity and attributes (not the RO's protected resource as is done in step 13 below).

* **This is one step during which the OC can establish a SSO session with the RO by returning a SSO session cookie to the UA (for example, though this is a commonly used mechanism). This SSO session cookie will ensure that the OC will not challenge the RO for their credentials if they're redirected to the OC by another relying party (e.g. another CUFX endpoint) during the SSO session.**

#### Step 7: Resource owner authorization collected

The RO sends their authorization via the UA to the OC.

#### Step 8: OC (openid connect) authorization passed back to OA (oauth authorization)

The OC returns an authorization code via the UA to the OA.

* **Since it is possible to combine steps 4 - 7 into a single challenge and response (e.g. the OC presents the RO with a form in the UA within which they can enter both their credentials and an authorization decision at the same time), this is another step during which the OC can establish a SSO session with the RO by returning a SSO session cookie to the UA (as described above).**

#### Step 9&10: Oa/OC request ID and access token request/return

The OA uses the authorization code to request ID and access tokens from the OC.

#### Step 11&12: oa/oc additional user attribute request/return

It's possible that the ID and access tokens returned by the OC (steps 9/10) are all that the OA and/or RP need to process the original CA request. However, it's also possible that the OA and/or RP need additional attributes of that user's identity. The OA can use the ID and access tokens to optionally get additional attributes from the OC.

#### Step 13: OA requests RO for authorization decisison to protected resource

The OA asks the RO for an authorization decision to grant or deny the CA access to the RO's protected resource (not the RO's identity and attributes as is done in step 6 above).

#### Step 14: RO sends authorization decisison to OA

The RO sends their authorization via the UA to the OA.

#### Step 15: OA sends authorization code to Ca

The OA returns an authorization code via the UA to the CA (client application).

#### Step 16&17: Ca requests access token from OA

The CA requests an access token from the OA, OA returns an access token to the CA.

* **With OAuth, the access token can effectively be the SSO session token that can be used by the CA to avoid additional challenges by the OA when accessing additional protected resources provided by RPs protected by the same OA.**

#### Step 18: ca requests resource from rp (resource provider)

The CA uses the access token to request the protected resource from the RP.

#### Step 19: rp provides protected reource

Lastly, the RP returns the protected resource to the CA.

## Use case 2: Resource Provider Initiated SSO with OAuth (Authorization Code Grant Type) and SAML 2.0 (Web Browser SSO Profile)

The below sequence diagram combines OAuth with SAML to support SSO with federated identification. Before beginning this sequence, to establish mutual trust, the client application would have registered with the CUFX endpoint OAuth authorization server. The CUFX endpoint OAuth authorization server would store a trusted copy of the SAML identity provider's public certificate used by the SAML identity provider to sign SAML assertions.

The SAML assertion sent to the OAuth authorization server with the SAML response should contain all attributes required by the OAuth authorization server and/or the resource provider.



Let’s walk through the details now.

#### STep 1: The Resource Owner (RO) to access a Resource Provider (RP)

The Resource Owner (RO) uses the client application (CA) to access a Resource Provider (RP) of a resource they own. In this case, the CUFX endpoint is the RP (e.g., PFM provider).

#### Step 2: Client application request Oauth access token

Knowing that an OAuth access token is required to access the protected resource, the CA redirects the RO (e.g., end user or other entity) via another user agent (UA; often a browser) to the CUFX endpoint's OAuth Authorization Server (OA).

#### Step 3: Oauth authorization server (oa) redirects to SAML identity provider

Normally, this is when the OA would challenge the RO for credentials that it would authenticate against its own instance of that identity. However, in this example, the OA does not have direct access to that identity, and it will instead rely on federated identification through its trust relationship with the RO's SAML Identity Provider (IP). This step begins the SAML Identity Federation flow. The OA redirects a SAML request via the UA to the IP.

#### Step 4: resource owner credential challenge

The SAML IP challenges the RO via the UA for credentials. This can be done however the IP likes. For example, this could be done using HTTP Basic Auth credentials, Windows Integrated Authentication credentials, or HTML form authentication.

#### Step 5: resource owner credentials collected

The RO sends their credentials via the UA to the IP.

#### Step 6: SAMl response to oa

The IP redirects a SAML response via the UA to the OA.

* **During this step, the IP can establish a SSO session with the RO by returning a SSO session cookie to the UA (for example, though this is a commonly used mechanism). This SSO session cookie will ensure that the IP will not challenge the RO for their credentials if they're redirected to the IP by another relying party (e.g. another CUFX endpoint) during the SSO session.**

#### Step 7&8: resource owner authorization collected

The OA asks the RO for an authorization decision to grant or deny the CA access to the RO's protected resource. RO sends their authorization via the UA to the OA.

#### Step 9: authorization code sent to ca

The OA returns an authorization code via the UA to the CA.

#### Step 10&11: ca requests access token from OA

The CA requests an access token from the OA, OA returns an access token to the CA.

* **With OAuth, the access token can effectively be the SSO session token that can be used by the CA to avoid additional challenges by the OA when accessing additional protected resources provided by RPs protected by the same OA.**

#### Step 12: ca requests resource from rp (resource provider)

The CA uses the access token to request the protected resource from the RP.

#### Step 13: rp provides protected reource

Lastly, the RP returns the protected resource to the CA.

1. RO uses CA to access RP

## Use case 3: Client Application Initiated SSO with OAuth (JWT Bearer Token Grant Type)

The below sequence diagram shows how the Jason Web Token (JWT) bearer token OAuth extension grant type can be used for client application initiated SSO. Before beginning this sequence, to establish mutual trust, the client application would have registered with the CUFX endpoint OAuth authorization server. The CUFX endpoint OAuth authorization server would share a secret with the token provider to be used with JWT signature algorithms.

Steps 2 and 3 of this diagram have been left initially vague. There are many ways that non-CUFX endpoint components could support these steps, but all that matters from a CUFX endpoint's perspective is that a valid and trusted JWT bearer token is sent by the client application with its request to the OAuth authorization server for an access token (per http://tools.ietf.org/html/draft-jones-oauth-jwt-bearer-04).

The JWT bearer token sent by the client application to the OAuth authorization server should contain all attributes required by the OAuth authorization server and/or the resource provider.



Let’s walk through the details now.

#### Step 1: The Resource Owner (RO) to access a Resource Provider (RP)

The Resource Owner (RO) uses the client application (CA) to access a Resource Provider (RP) of a resource they own. In this case, the CUFX endpoint is the RP (e.g., PFM provider).

#### Step 2: client application requests JWT bearer token

The Client Application (CA) request a JWT bearer token from the JWT Token Provider (TP). This step may or may not involve a user agent. This step includes any authentication and authorization that the TP chooses to support.

#### Step 3: token provider (tp) responds with bearer token

The TP responds with a JWT bearer token.

* **The TP could include a SSO session identifier/token with this response. This identifier/token could be used by the same client application to avoid additional challenges by the same token provider during the SSO session. If a browser like user agent is used as part of step 2, then a cookie could be used. That would support SSO to that TP across multiple client applications that use the same user agent/browser.**

#### Step 4&5: ca access request sent to oa

The CA sends the JWT bearer token with its request for an access token from the OA. OA returns an access token to the CA.

* **With OAuth, the access token can effectively be the SSO session token that can be used by the CA to avoid additional challenges by the OA when accessing additional protected resources provided by RPs protected by the same OA.**

#### Step 6: ca requests resource from rp (resource provider)

The CA uses the access token to request the protected resource from the RP.

#### Step 7: rp provides protected reource

Lastly, the RP returns the protected resource to the CA.

Please note, message examples are coming soon.

## Use case 4: Client Application Initiated SSO with OAuth (SAML Bearer Assertion Grant Type)

The below sequence diagram shows how the SAML bearer assertion OAuth extension grant type can be used for client application initiated SSO. Before beginning this sequence, to establish mutual trust, the client application would have registered with the CUFX endpoint OAuth authorization server. The CUFX endpoint OAuth authorization server would store a trusted copy of the SAML identity provider's public certificate used by the SAML identity provider to sign SAML assertions.

Steps 2 and 3 of this diagram have been left initially vague. There are many ways that non-CUFX endpoint components could support these steps, but all that matters from a CUFX endpoint's perspective is that a valid and trusted SAML bearer assertion is sent by the client application with its request to the OAuth authorization server for an access token (per http://tools.ietf.org/html/draft-ietf-oauth-saml2-bearer-15).

The SAML assertion sent to the OAuth authorization server with the SAML response should contain all attributes required by the OAuth authorization server and/or the resource provider.



Let’s walk through the details now.

#### STep 1: The Resource Owner (RO) to access a Resource Provider (RP)

The Resource Owner (RO) uses the client application (CA) to access a Resource Provider (RP) of a resource they own. In this case, the CUFX endpoint is the RP (e.g., PFM provider).

#### Step 2: client application requests SAML bearer token

The Client Application (CA) requests a SAML bearer assertion from the SAML Identity Provider (IP). This step may or may not involve a user agent. This step includes any authentication and authorization that the IP chooses to support.

#### Step 3: SAML IP responds with a SAML bearer assertion.

The IP responds with a SAML bearer assertion.

* **The IP could include a SSO session identifier/token with this response. This identifier/token could be used by the same client application to avoid additional challenges by the same token provider during the SSO session. If a browser like user agent is used as part of step 2, then a cookie could be used. That would support SSO to that IP across multiple client applications that use the same user agent/browser.**

#### Step 4&5: The CA requests an access token from the OA.

The CA sends the SAML bearer assertion with its request for an access token from the OA. OA returns an access token to the CA

* **With OAuth, the access token can effectively be the SSO session token that can be used by the CA to avoid additional challenges by the OA when accessing additional protected resources provided by RPs protected by the same OA.**

#### Step 6: ca requests resource from rp (resource provider)

The CA uses the access token to request the protected resource from the RP.

#### Step 7: rp provides protected reource

Lastly, the RP returns the protected resource to the CA.

## Use case 5: OAuth 2.0 Access Control using Client Credentials Grant Type

This is an example of using OAuth 2.0 client credentials grant type for access control if federated SSO and resource owner participation is not required. Before beginning this sequence, the client application would have registered with the CUFX endpoint OAuth authorization server.



Let’s walk through the details now.

#### STep 1: The CA requests an access token from the OA

The CA sends its client credentials with its request for an access token from the OA

#### STep 2: OA returns access token to CA

The OA returns an access token to the CA.

* **With OAuth, the access token can effectively be the SSO session token that can be used by the CA to avoid additional challenges by the OA when accessing additional protected resources provided by RPs protected by the same OA.**

#### Step 3: ca requests resource from rp (resource provider)

The CA uses the access token to request the protected resource from the RP.

#### Step 4: rp provides protected reource

## Use case 6: SAML: Identity Provider Initiated SSO with REDIRECT

Here is an example of Identity Provider initiated SSO with a redirect using an artifact resolve/response methodology. <http://documentation.pingidentity.com/display/PF66/IdP-Initiated+SSO--Artifact>



#### STep 1:

Resource owner logs into the client application using the user credentials which validates them against the identity provider. Many times the identity provider will be a core system that holds the user’s credentials.

#### STep 2:

The user will select the resource within the client application by clicking or some other action in the user agent (typically a browser).

#### Step 3:

The client application will request a SAML assertion from the Identity Provider. The SAML assertion is incomplete in that it doesn’t include all the details necessary to gain access to the resource owner’s data.

#### Step 4:

The client application will request a SAML assertion from the Identity Provider. The SAML assertion is incomplete in that it doesn’t include all the details necessary to gain access to the resource owner’s data. The client application will encrypt the message using a symmetric AES 256 bit key with the CBC algorithm. The Identity Provider generates a hash of the encrypted message using the private key of the client application. (The public key was shared out-of-band when setting up the process). The user agent then will forward the incomplete SAML assertion including the hash to the resource provider.

#### Step 5:

The user agent will then forward the encrypted incomplete SAML assertion and artifact to the resource provider. This is done by placing the encrypted SAML assertion and artifact in hidden form fields and then posting to the Resource Providers endpoint which results in the end user being redirected to the resource provider.

#### Step 6:

The resource provider will validate the signature of the message using the provided hash and the client application’s public key. In addition, the resource provider will decrypt the SAML assertion.

#### Step 7:

The resource provider packages up a SAML resolve request including the artifact. The resource provider will encrypt the message using a symmetric AES 256 bit key with the CBC algorithm. A new signature is created representing the signature for this new message using the resource provider’s private key. The resolve is requested of the Identity Provider.

#### Step 8:

The Identity Provider decrypts the AES encrypted SAML resolve message and verifies the signature of the resolve message using the resource provider’s public key (provided out-of-band when setting up the process).

#### Step 9:

The Identity Provider sends the AES encrypted complete SAML assertion with resolved artifact which includes the claims to completely identity the resource owner and access rights of the resource owner.

#### Step 10/11:

The Resource Provider can then decrypt the message and use the claims (typically a token) to access information from the client application’s resources. CUFX messages are provided by the client application to gain access to those resources. The messageContext in the CUFX messages contains the claims in the user:userId field. (See the discussion on AES, SSL, IP, etc. to determine what level of security is needed for the implementation for the CUFX messages).

## Manage user identities with SCIM

SCIM (System for Cross-Domain Identity Management). Documentation: <http://www.simplecloud.info/> and <http://tools.ietf.org/html/draft-ietf-scim-core-schema-00>

The prior SSO examples used a federated model where user credentials were centrally located at one resource. This is the recommend approach to handle user authentication and authorization. If it is deemed an absolute necessity to break from this model and create user profiles in remote system, SCIM is one possible model to accomplish this. SCIM is based on an object model, and user data is represented as a SCIM object in JSON.   
  
*Note: CUFX does not recommend creation of users across multiple systems. The federated identity for authentication and authorization is the preferred approach.*

Two use cases are presented to illustrate the above. In use case 1, the user first visits the OLB (IdP) site. They enter their credentials and click a link on the OLB page which will redirect the user to the PFM website. In the example presented this is a first time user accessing the PFM site. The user is first created at the PFM provider using SCIM (System for Cross-domain Identity Management) specification. Next, a background verification of the user will occur between PFM and OLB applications (SAML exchanges). If the verification succeeds, the user will be presented with the PFM content without a new credentials challenge by the PFM application.

In second use case, the user first visits the PFM site and the PFM site will then redirect the user to the OLB site (the IdP). The OLB application will then challenge the user if there is no stored authentication in a session. Once the user is authenticated at the OLB site, the user is redirected back to the PFM site without another credentials challenge. Most of this occurs in background with the help of HTTP redirection.

## SCIM use case 1: First Time Interactions From A CUFX client to Typical SSO App with Client App

The following process flows shows the typical flow for an SSO application for the first time that a user is using the SSO application when the Client App is retrieving the Security Token from the data source.



Let’s walk through the details now.

#### STep 1: USER selects Service provider url on the identity provider’s web page

User clicks on the service provider URL from identity provider’s web page. SSO process is initiated using SAML.

#### Step 2&3: CREAT SSO USER AT SERVICE PROVIDER

As this is the first time the user is using this service it will create a SSO user at the service provider using SCIM.  
*(Refer to SSO user section for sample SCIM messages)*

#### Step 4&5: Identity provider responds with samlart which IS posted to service provider

As the IdP has already authenticated the member, it now can issue a SAMLart (artifact) for this authentication which is a POST to the SP (service provider) through the user’s browser with HTTP redirection.

#### Step 6&7: service provider verify artifact against identity provider

Service provider then issues an artifact resolution to the identity provider to confirm if this user has been authenticated. And, if the response is yes, the service provider considers the user authenticated and grants access to the resources provided by the service provider.

## SCIM use case 2: Subsequent Interactions From A CUFX client to SSO App (With Client App Retreiving Security Token)



Let’s walk through the details now.

#### STep 1: USER selects Service provider url on the identity provider’s web page

User clicks on the service provider URL from identity provider’s web page. SSO process is initiated.

#### Step 2&3: GET SSO USER FROM SERVICE PROVIDER

Will get the SSO user profile from the service provider using SCIM specification.

#### Step 4&5: UPDATE SSO USER AT SERVICE PROVIDER

Will update the SSO user profile at the service provider if it has changed using SCIM specification.

#### Step 6&7: Identity provider responds with samlart which IS posted to service provider

As the IdP has already authenticated the member, it now can issue a SAMLart (artifact) for this authentication which is a POST to the SP (service provider) through the user’s browser with HTTP redirection.

#### Step 8&9: service provider VERIFY artifact against identity provider

Service provider then issues an artifact resolution to the identity provider to confirm if this user has been authenticated. And, if the response is yes, the service provider considers the user authenticated and grants access to the resources provided by the service provider.

<samlp:ArtifactResolve

xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"  
 xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"  
 ID="pfm\_SAML\_Id"  
 Version="2.0"  
 IssueInstant="2012-08-28T09:21:58"  
 Destination="https://olb/SAML2/ArtifactResolution"  
 <saml:Issuer>pfm.com</saml:Issuer>  
 <samlp:Artifact>”olb\_issued\_artifact\_for\_olb\_user\_xxx”</samlp:Artifact>  
 </samlp:ArtifactResolve>

<samlp:ArtifactResponse

      xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"

      ID="olb\_SAML\_id" Version="2.0"

      InResponseTo="pfm\_SAML\_id"

      IssueInstant="2012-08-09T08:30:32Z">

    <Issuer>olb.com</Issuer>

    <samlp:Status>

      <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>

    </samlp:Status>

<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" ID="olb\_SAML\_Id”  Version="2.0" IssueInstant="2012-08-28T09:22:23Z">  
 <saml:Issuer xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">olb.com</saml:Issuer>  
 <samlp:Status xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">

<samlp:StatusCode xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>

</samlp:Status>

   <saml:Subject>

  <saml:NameID   Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">OLB/PFM UserId</saml:NameID>

  <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">

  <saml:SubjectConfirmationData NotOnOrAfter="2012-09-26T02:44:24.173Z" Recipient="http://localhost:9000"/>

  </saml:SubjectConfirmation>

   </saml:Subject>

</samlp:Response>

</samlp:ArtifactResponse>

## SSO User using SCIM idenity mangement

Before a user can register with a remote app, the user must first be created within the remote application. To manage a user in a remote third party system, a client like online banking would create the user inside the remote system if the user doesn’t already exist. This user would correspond to a single logged in entity on the system. The “fiUserId” allows the financial institution to uniquely identify a user for later update or delete requests in a format of their choosing. This value must be unique per institution.

The response would be the successful message for the newly created user or an error message explaining the failure. During testing, an appVendorToken will be provided for the test services.

|  |  |
| --- | --- |
| Definition | Single-sign-on user resource |
| Overview of Capabilities | Ability to Create, Read, Update, Delete users from 3rd party system. Ability to get redirect information for 3rd party application. |
| Dependencies | Security Services for authentication and security. |
| CUFX TEST REST-JSON LINK | <https://services.cufx.org/vendor/appVendorToken/user> |
| CUFX TEST REST-XML LINK | None |
| CUFX WSDL LINK | None |

#### REST-JSON User Create format:

**REQUEST:**

**Headers:**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us *(IANA – language codes)(W3C, HTTP Protocols)*

Content-type: application/json; charset=utf-8

X-API-Version: >=1.0.5

**POST** https://api.serviceprovider.com/v1/FIs/<fiId>/Users/

{

"schemas":["urn:scim:schemas:core:1.0"],

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara"

}

}

RESPONSE:

**Headers:**

Status Code: 200 Ok

Content-type: application/json; charset=utf-8

Content-Language: en-us

{

"schemas":["urn:scim:schemas:core:1.0"],

"id":"2819c223-7f76-453a-919d-413861904646",

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara"

}

}

#### REST-JSON User Read Request format:

**REQUEST:**

**Headers:**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us

Content-type: application/json; charset=utf-8

X-API-Version: >=1.0.5

**GET** https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/

**RESPONSE:**

**Headers:**

Status Code: 200 Ok

Content-type: application/json; charset=utf-8

Content-Language: en-us

{

"schemas":["urn:scim:schemas:core:1.0"],

"id":"2819c223-7f76-453a-919d-413861904646",

"meta":{

"created":"2011-08-01T18:29:49.793Z",

"lastModified":"2011-08-01T18:29:49.793Z",

"location":"https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/",

"version":"W\/\"f250dd84f0671c3\""

},

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara"

}

}

**REQUEST:**

**Headers:**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us

Content-type: application/json; charset=utf-8

X-API-Version: >=1.0.5

**GET** https://api.serviceprovider.com/v1/FIs/<fiId>/Users/?userName=bjensen

**RESPONSE:**

**Headers:**

Status Code: 200 Ok

Content-type: application/json; charset=utf-8

Content-Language: en-us

{

"schemas":["urn:scim:schemas:core:1.0"],

"id":"2819c223-7f76-453a-919d-413861904646",

"meta":{

"created":"2011-08-01T18:29:49.793Z",

"lastModified":"2011-08-01T18:29:49.793Z",

"location":"https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/",

"version":"W\/\"f250dd84f0671c3\""

},

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara"

}

}

#### REST-JSON User Update format:

**Headers:**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us

Content-type: application/json; charset=utf-8

X-API-Version: >=1.0.4

**PUT** https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/

{

"schemas":["urn:scim:schemas:core:1.0"],

"id":"2819c223-7f76-453a-919d-413861904646",

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara",

"middleName":"Jane"

},

"emails":[

{

"value":"bjensen@example.com"

},

{

"value":"babs@jensen.org"

}

]

}

RESPONSE:

Headers

Status Code: 200 Ok

Content-type: application/json; charset=utf-8

Content-Language: en-us

{

"schemas":["urn:scim:schemas:core:1.0"],

"id":"2819c223-7f76-453a-919d-413861904646",

"userName":"bjensen",

"externalId":"bjensen",

"name":{

"formatted":"Ms. Barbara J Jensen III",

"familyName":"Jensen",

"givenName":"Barbara",

"middleName":"Jane"

},

"emails":[

{

"value":"bjensen@example.com"

},

{

"value":"babs@jensen.org"

}

],

"meta": {

"created":"2011-08-08T04:56:22Z",

"lastModified":"2011-08-08T08:00:12Z",

"location":" https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/",

"version":"W\/\"b431af54f0671a2\""

}

}

#### REST-JSON User Delete format:

**REQUEST:**

**Headers:**

Accept: application/json

Accept-Charset: utf-8

Accept-Language: en-us

Content-type: application/json; charset=utf-8

X-API-Version: >=1.0.5

**DELETE** https://api.serviceprovider.com/v1/FIs/<fiId>/Users/2819c223-7f76-453a-919d-413861904646/

**RESPONSE:**

Headers

Status Code: 200 Ok

# General Error handling For All Services

Refer to latest CUFX documentation *Error Mapping*.

# Bibliography

Bazzari, F. (2009, August 14). *The Definitive Guide to GET vs POST.* Retrieved April 12, 2012, from http://thinkvitamin.com/code/the-definitive-guide-to-get-vs-post/

Diffen.com. (n.d.). Retrieved from http://www.diffen.com/difference/Get\_vs\_Post

IEEE. (2012, January 05). *AES: 512 bit Abstract.* Retrieved April 25, 2012, from IEEE: http://ieeexplore.ieee.org/xpl/login.jsp?reload=true&tp=&arnumber=6122835&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs\_all.jsp%3Farnumber%3D6122835

IETF. (2012, December). *JSON Web Encryption (JWE) draf-ietf-jose-jwe-web-encryption-08 .* Retrieved February 21, 2013, from IETF.org: http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-08

IETF. (2012, December). *JSON Web Encryption JSON Serialization (JWE-JS) draft-jones-jose-jwe-json-serialization-04 .* Retrieved February 21, 2013, from IETF.org: tools.ietf.org/html/draft-jones-jose-jwe-json-serialization-04

IETF. (2010, April). *RFC 5849 - The Oauth 1.0 Protocol.* Retrieved July 27, 2012, from IETF.org: http://tools.ietf.org/html/rfc5849

Introduction to Public-Key-Cryptograpy. (n.d.). Retrieved January 5, 2013, from Mozilla.org: https://developer.mozilla.org/en-US/docs/Introduction\_to\_Public-Key\_Cryptography

NIST. (2001, November 26). *Announcing the Advanced Encryption Standard (AES).* Retrieved April 25, 2012, from NIST: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

OpenID Connect. (n.d.). Retrieved January 15, 2013, from http://openid.net/connect/

*Simple Cloud Identity Management.* (n.d.). Retrieved July 27, 2012, from Simple Cloud Identity Management: http://www.simplecloud.info/

sources, W. M. (2012, April 25). *Wikipedia : AES.* Retrieved April 25, 2012, from Wikipedia: http://en.wikipedia.org/wiki/Advanced\_Encryption\_Standard

W3C. (n.d.). *Date Time Formats.* Retrieved from World Wide Web Consortium: http://www.w3.org/TR/NOTE-datetime

W3C. (n.d.). *XML Encryption Syntax and Processing.* Retrieved from World Wide Web Consortium: http://www.w3.org/TR/xmlenc-core