

**Collaboard Install**

SAML and Storage integration

|  |  |
| --- | --- |
| Title: | Collaboard SAML and storage integration |
| Abstract: | This document describes the integration of SAML into IBV.Auth and it describes the cloud storage integration |
| Status\* | Draft |
| Document Owner: | IBV |
| Document Author: | Gian Paolo Santopaolo |
| \*Draft, In progress, Reviewed, Authorized, Approved, Completed | |

|  |  |  |
| --- | --- | --- |
| **Document Control** | | |
| Document Distribution: | Internal: | All dev Teams |
|  | External: |  |
| Document Authorization: | Authorizer(s): | Gian Paolo Santopaolo |

|  |  |  |  |
| --- | --- | --- | --- |
| **Change History** | | | |
| Date | Ver | Changed By | Details |
| 18/01/2021 | 1.0 | D. Stellakis | Initial Draft |
| 23/01/2021 | 1.1 | D. Stellakis | Added S3 storage chapter |
| 25/01/2021 | 1.2 | GP Santopaolo | Document review |
| 18/05/2021 | 1.3 | D. Stellakis | Added JWT chapter |
|  |  |  |  |
|  |  |  |  |

Index

[Overview 3](#_Toc72254441)

[SAML 4](#_Toc72254442)

[Flows 4](#_Toc72254443)

[Bindings 4](#_Toc72254444)

[Assertions 4](#_Toc72254445)

[JSON Web Tokens 7](#_Toc72254446)

[JSON Web Token structure 7](#_Toc72254447)

[Header 7](#_Toc72254448)

[Payload 7](#_Toc72254449)

[Signature 7](#_Toc72254450)

[Putting it all together 8](#_Toc72254451)

[Storage core concepts 9](#_Toc72254452)

[Amazon S3 9](#_Toc72254453)

[Buckets 9](#_Toc72254454)

[Objects 9](#_Toc72254455)

[Keys 9](#_Toc72254456)

[Shared Access Signature (SAS) 9](#_Toc72254457)

[Storage Integration 12](#_Toc72254458)

[The client access the could storage directly 12](#_Toc72254459)

[The client uses a proxy to access the storage 13](#_Toc72254460)

[The client uses MFT to access the storage 14](#_Toc72254461)

[References 15](#_Toc72254462)

[Microsoft Docs - Shared Access Signatures: Understanding the SAS Model 15](#_Toc72254463)

# Overview

This document describes the integration of SAML into IBV.Auth and it describes the cloud storage integration

Our engineers can answer further questions.

# SAML

SAML stands for Security Assertion Markup Language. It is an XML-based open-standard for transferring identity data between two parties: an identity provider (IdP) and a service provider (SP). We are compatible with identity providers supporting the SAML 2.0 HTTP Redirect / POST Binding

Identity Provider — Performs authentication and passes the user's identity and authorization level to the service provider.

Service Provider — Trusts the identity provider and authorizes the given user to access the requested resource

## Flows

SAML supports two different types of flows: those initiated by the service provider and those initiated by the identity provider. In our case, we cover the SP-initiated flow. In SP-initiated flows, you start out at the service provider (IBV), are redirected to the identity provider to authenticate, and are then redirected back to the service provider. This flow is usually initiated when a user clicks on "Login with SSO" button or something similar.

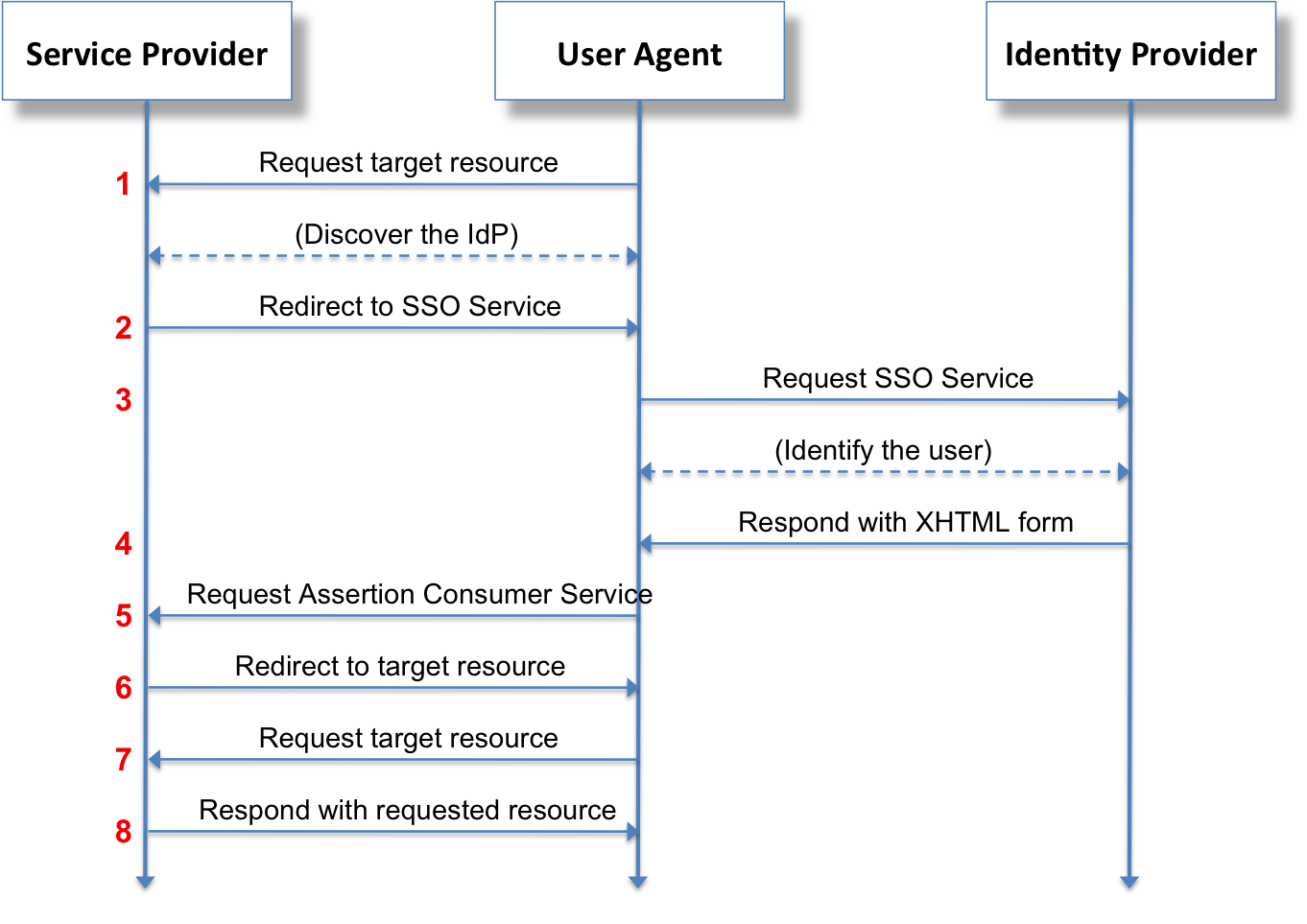
## Bindings

Bindings are the format in which data is transferred between service providers and identity providers. The two most popular are HTTP Redirect Binding and HTTP POST Binding. HTTP Redirect Bindings transfer data using HTTP redirects and query parameters; this type of binding is typically used in authentication requests. HTTP POST Binding transfer data using HTTP POST forms, this type of binding is typically used in authentication responses

## Assertions

Assertions are statements made by the identity provider about the principal. For example, the principal's email address and/or groups/roles the principal may be associated with. Assertions are used by the service provider to create and configure sessions for a principal.

A typical sign-in flow using SAML redirect binding is:



To integrate with an identity provider using the SAML 2.0 protocol, we will need the XML Metadata of the provider. A sample IdP XML metadata file would look like this:

<EntityDescriptor entityID="urn:idp.example.org" xmlns="urn:oasis:names:tc:SAML:2.0:metadata">

<IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">

<KeyDescriptor use="signing">

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<X509Data>

<X509Certificate>MIIDBzCCAe+gAwIBAgI...P3Z3TTTs=</X509Certificate>

</X509Data>

</KeyInfo>

</KeyDescriptor>

<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="https://idp.example.org/saml/logout"/>

<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://idp.example.org/saml/logout"/>

<NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress</NameIDFormat>

<NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:persistent</NameIDFormat>

<NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>

<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="https://idp.example.org/saml"/>

<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://idp.example.org/saml"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="E-Mail Address" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Given Name" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Name" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Surname" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri" FriendlyName="Name ID" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"/>

</IDPSSODescriptor>

</EntityDescriptor>

Alternatively, we would need the following information from the identity provider:

* IdP EntityID
* IdP Redirect URL
* IdP Logout URL
* IdP public certificate
* NameId Format
* User-specific attributes returned by the authentication response

# Identifying a user

Once a user uses an identity provider for signing in to Collaboard, the system will use his user principal name (upn) returned by the SAML identity server, as well as other claims returned (such as first name, last name, email), to create the user’s profile in Collaboard. The user will need to review his profile and accept the terms and conditions of the Collaboard application to proceed to use Collaboard.

If the user’s profile is already present in the system, it will be located and reused, provided that the user has accepted the terms of service.

Since Collaboard does not keep any private user data in its data storage, all that it needs to identify a user is the upn claim returned by the SAML server, matching it to its own username field. Thus, we can create user profiles for external users into the Collaboard data storage having only the username as a required field, and when a user chooses to authenticate externally via SAML, his profile will be completed after a successful authentication.

# JSON Web Tokens

After authenticating with SAML, the server will return to the calling client application a Collaboard-specific authorization token. This token will be used in every communication from the client to the server, being web API calls or SignalR calls, and the server will validate it every time, to allow or disallow access to the called API. The token is a JSON Web Token.

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can also be encrypted to also provide secrecy between parties, we use signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

## 

## JSON Web Token structure

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

### Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

### Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data.

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

### Signature

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

The signature is used to verify the message was not changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

### Putting it all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.



# Storage core concepts

## Amazon S3

### Buckets

A bucket is a container for objects stored in Amazon S3. Every object is contained in a bucket. For example, if the object named *photos/puppy.jpg* is stored in the *awsexamplebucket1* bucket in the US West (Oregon) Region, then it is addressable using the *URL https://awsexamplebucket1.s3.us-west-2.amazonaws.com/photos/puppy.jpg.*

Buckets serve several purposes:

* They organize the Amazon S3 namespace at the highest level.
* They identify the account responsible for storage and data transfer charges.
* They play a role in access control.
* They serve as the unit of aggregation for usage reporting.

### Objects

Objects are the fundamental entities stored in Amazon S3. Objects consist of object data and metadata. The data portion is opaque to Amazon S3. The metadata is a set of name-value pairs that describe the object. These include some default metadata, such as the date last modified, and standard HTTP metadata, such as *Content-Type*. You can also specify custom metadata at the time the object is stored.

An object is uniquely identified within a bucket by a key (name) and a version ID.

### Keys

A key is the unique identifier for an object within a bucket. Every object in a bucket has exactly one key. The combination of a bucket, key, and version ID uniquely identify each object. So you can think of Amazon S3 as a basic data map between "bucket + key + version" and the object itself. Every object in Amazon S3 can be uniquely addressed through the combination of the web service endpoint, bucket name, key, and optionally, a version. For example, in the URL https://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl, doc is the name of the bucket and 2006-03-01/AmazonS3.wsdl is the key.

## Shared Access Signature (SAS)

A shared access signature (SAS) (or signed URL) provides secure delegated access to resources in your storage account. With a SAS, you have granular control over how a client can access your data. For example:

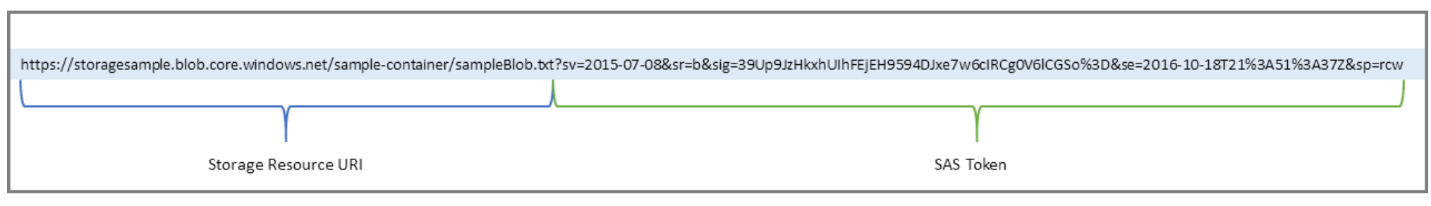
* What resources the client may access.
* What permissions they have to those resources.
* How long the SAS is valid.

A shared access signature is a signed URI that points to one or more storage resources. The URI includes a token that contains a special set of query parameters. The token indicates how the resources may be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the key that was used to create the SAS. This signature is used by Cloud Storage to authorize access to the storage resource.

The SAS token is a string that you generate on the client side, for example by using one of the Azure Storage client libraries. The SAS token is not tracked by Azure Storage in any way. You can create an unlimited number of SAS tokens on the client side. After you create a SAS, you can distribute it to client applications that require access to resources in your storage account.

Client applications provide the SAS URI to Azure Storage as part of a request. Then, the service checks the SAS parameters and the signature to verify that it is valid. If the service verifies that the signature is valid, then the request is authorized. Otherwise, the request is declined with error code 403 (Forbidden).

Here's an example of a service SAS URI, showing the resource URI and the SAS token. Because the SAS token comprises the URI query string, the resource URI must be followed first by a question mark, and then by the SAS token:



*Example os SAS token*

Use a SAS to give secure access to resources in your storage account to any client who does not otherwise have permissions to those resources.

A common scenario where a SAS is useful is a service where users read and write their own data to your storage account. In a scenario where a storage account stores user data, there are two typical design patterns:

1. Clients upload and download data via a front-end proxy service, which performs authentication. This front-end proxy service allows the validation of business rules. But for large amounts of data, or high-volume transactions, creating a service that can scale to match demand may be expensive or difficult.



1. A lightweight service authenticates the client as needed and then generates a SAS. Once the client application receives the SAS, it can access storage account resources directly. Access permissions are defined by the SAS and for the interval allowed by the SAS. The SAS mitigates the need for routing all data through the front-end proxy service.



Many real-world services may use a hybrid of these two approaches. For example, some data might be processed and validated via the front-end proxy. Other data is saved and/or read directly using SAS.

Additionally, a SAS is required to authorize access to the source object in a copy operation in certain scenarios:

* When you copy a blob to another blob that resides in a different storage account.
* You can optionally use a SAS to authorize access to the destination blob as well.
* When you copy a file to another file that resides in a different storage account.
* You can optionally use a SAS to authorize access to the destination file as well.
* When you copy a blob to a file, or a file to a blob.

You must use a SAS even if the source and destination objects reside within the same storage account.

Now that we defined what a shared access signature is we can move forward and explain the two ways the client can access the storage in Collaboard

The difference between the Amazon and Azure cloud for SAS is that on Azure is possible to create

# Storage Integration

Currently, Collaboard supports two types of persistent storage for it's assets:

* Azure Storage
* Local (on-premise) storage

There are three ways that the client can access the stored assets.

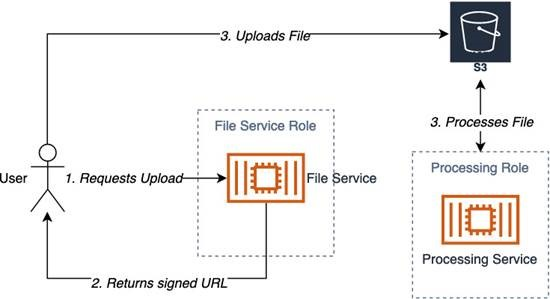
Before describing the two scenarios, we need to define what is it a SAS (Shared access signature).

## The client access the could storage directly

This is one of the scenarios we currently support in Collaboard. The client has the necessary libraries and code to call the cloud Storage directly to access the resources.

If we decide to go for this approach, to integrate Amazon Simple Storage Service (Amazon S3) we would have to

* Add code to the client to be able to access Amazon S3 directly (possibly using Amazon S3 javascript library)
* Add code to specific parts of the server that access the storage, to be able to also access Amazon S3, along with Azure Storage and on-premise storage (possibly using Amazon S3 .NET library)
* Provide a way in the server to create signed URLs to upload/download data to the S3 buckets, which then the client will use to do the operation directly to S3. This is similar to the Azure Storage SAS token, that the client uses to upload/download data from Azure Storage.



*Flow: Client accessing cloud storage directly*

Currently in Collaboard, we generate a SAS token per-project, and this token is stored in the database and used for every storage action initiated by the client. Another approach would be that the clients calls a server API, provides information about the asset, and the server responds with the SAS token / Signed URL used to perform the necessary action.

This approach may be needed in the case of Amazon S3 because each request action needs to have its own signed URL, and the signed URLs are short-lived. Creating a signed URL does not communicate with Amazon S3, it is a completely offline process, and the signing credentials used are stored in the server and not visible to the client.

User authentication would happen as it normally happens, where the client authenticates with the identity provider, gets an access token, and uses this token to generate a signed URL to access a storage asset. The client does not need to authenticate with the actual storage used.

The disadvantage of this approach is that code changes are required in the client to be able to use S3 API

## The client uses a proxy to access the storage

The clients call our server API, which acts as a proxy to the underlying storage used

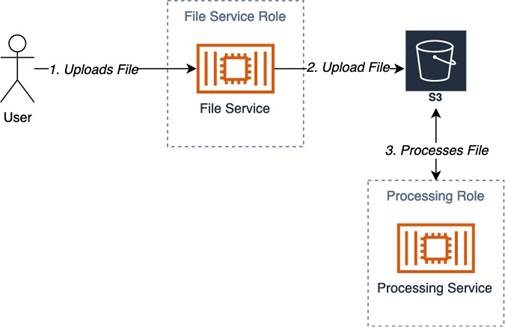
This solution removes the need for the client to be aware of the storage providers supported by Collaboard, and instead introduces a proxy API that the client can call to get access to the storage resource. This proxy API then has all the implementation details needed to access the storage provider and get or put the asset.

Also in this scenario, user authentication would happen as it normally happens, where the client authenticates with the identity provider, gets an access token, and uses this token to invoke our proxy storage API. The client does not need to know any storage-specific URLs or keys, or to in any way authenticate with the actual storage used.

To upload an asset, the client would call our upload API and the API would then save/stream the asset to the storage used.

To download an asset, the client would call our download API and the API would then download / stream the asset from the storage used and deliver it to the client. Alternatively (in the Amazon S3 case), the API can return a redirect with the signed url of the asset, so that the client can download the asset directly via HTTP.

The disadvantages of this approach are that it limits upload performance and introduces extra cost via intermediate service's CPU time (proxy API). Also we would need to actually create this server proxy storage API, and code changes would be needed for the client to use the proxy API (although these changes would be done once, the next time we introduce a new storage provider, no client changes would be necessary).



## The client uses MFT to access the storage

For this use case, please refer to the Managed File Transfer FDS

# References

## Microsoft Docs - Shared Access Signatures: Understanding the SAS Model

<https://docs.microsoft.com/en-us/azure/storage/common/storage-sas-overview>