OIDC & Authentication-as-a-Service (AaaS)

Implementing OIDC for Login

Contents

Introduction

Authentication-as-a-Service (AaaS) offers OpenID Connect (OIDC) as a login solution. OIDC is a popular identity authentication protocol that standardizes the process for user authentication and authorization for access to digital services.

This document describes how to implement OIDC in your application when using AaaS.

Concept

OIDC is a web browser-based login flow implemented as a series of URL exchanges/redirects, and does not require any local state (e.g., cookies) to be maintained. From the application perspective, there are only two concerns: initiating the login process and receiving the login results. The basic flow entails:

- User navigates to the application and either a) the application detects that the user is not in session and automatically initiates the login sequence or b) the user initiates the login action (e.g., clicking the "login" call to action)
- The login request, known as the OIDC `/authorize` request, is sent from the application to AaaS for user login and other checks to ensure that the user is properly vetted
- AaaS sends information back to the application, which processes the response accordingly
- After the login flow is complete, the application receives tokens:
    - Access Token (`access_token`) – artifact that allows the client application to access the user's resource
    - ID Token (`id_token`) – encoded as a JSON Web Token (JWT) that proves that the user has been authenticated
    - Refresh Token (`refresh_ token`) – credentials that allow retrieval of new access tokens without requiring the user to re-authenticate

(A detailed flow diagram is found in the section OIDC Login Flow)

The remainder of this document describes technical details for implementing OIDC in your application.

Preliminary Requirements

This section describes how to prepare for OIDC prior to starting any coding development.

- Use your resources
    - OIDC specification found at: https://openid.net/specs/openid-connect-core-1_0.html
    - Cencora CIAM team; bring in an SME to answer questions, participate in technical conversation, and provide best practices and guidance
    - Network tracing tool (e.g., SAML Tracer) for inspecting and debugging network requests
    - API platform tool (e.g., Postman) to test and refine network requests
- Know how your software environment supports OIDC
    - Although OIDC is an open protocol, different software stacks provide support in their own distinct ways.
        - Client-side frameworks often rely on add-on components for OIDC, but there are good guides available online, and here are a few:
            - React - https://www.dhiwise.com/post/how-to-use-react-oidc-in-modern-authentication
            - Angular - https://www.npmjs.com/package/angular-auth-oidc-client
        - Server-side infrastructural support and methodologies differ across .NET, Java, node.js, etc., so you need to understand how your environment supports OIDC and/or cryptographic algorithms.
    - If your environment does not support OIDC, no worries; it is fairly straightforward to implement in your code as long as you understand how to handle HTTP request/response and JSON structures.
- Decide what kind of security model you need for your application

    - Traditional implementations rely on backend handling of OIDC responses for greatest security. This is the favored best practice because it prevents any exposure of tokens on the frontend.
    - Where backend connectivity is not possible, such as with Single Page Apps (SPA) and other completely frontend applications, it is recommended to use Proof of Key Code Exchange (PKCE) as detailed later in this document.
- Specify your redirect URI endpoint
    - OIDC requires a URI (identified as the variable `redirect_uri`) to be specified for receiving the final login result; the URI is registered in AaaS to guarantee that login responses are sent to a trusted recipient
    - Registration is case-sensitive, and any changes/updates must also be registered
- Register your application with AaaS
    - AaaS needs to configure settings for your application
    - You will receive a unique application ID (app_id) that needs to be included in your authorize request
- Receive configuration details from AaaS
    - The AaaS team supplies URLs needed to perform OIDC integrations, including sample code
    - The app_id uniquely identifies your application to AaaS

Environment

OIDC can be implemented by hand in homespun code, but most often it is implemented within a software development framework. Many popular frameworks handle OIDC natively, so the application should leverage inbuilt functionality. You will need to become familiar with the expected design pattern for your chosen framework.

If your framework does not support OIDC or you are handcrafting code, OIDC is implemented with relatively few lines of code. Basic requirements are discussed later in this document, but it should be straightforward if you understand how to handle and troubleshoot HTTP requests and JSON payloads. You can also rely on the CIAM team for guidance.

Configuration

Most of the necessary configuration for OIDC occurs on AaaS, although some software frameworks require some configuration on the application side (do your own research for your particular environment).

- What you need to provide to AaaS
    - Redirect URI (`redirect_uri`) for receiving login responses from AaaS
        - Example: `https://yourDomain/oidc/login_reply_endpoint`
        - This HTTPS endpoint receives OIDC authorization responses in the form of a URL with parameters that are interpreted by your application for further processing
          Example: `https://yourDomain/oidc/login_reply?code=st2.Atltez...sc3`
        - The URI is recorded as a "trusted" URI in AaaS to help secure the transmission of data between systems; note that this value is **case sensitive**!
    - Any changes to the redirect URI as you build out your application; if changes to your redirect URI are not registered in AaaS, your application will break
- What AaaS provides to you
    - Three environments to work in (requires configuration for each):
        - Test – sandbox for building and experimenting with the integration
        - Stage – stable production-like environment, but not for production releases
        - Production – production environment for production-grade applications
    - Each environment has its own API endpoints, so plan accordingly in your coding practices.
    - Standard "well-known" OIDC discovery endpoints, which may be required by software framework configurations
    - Client ID (`client_id`) to uniquely identify your OIDC configuration within AaaS

    - Application ID (`app_id`) to uniquely identify your application within AaaS
      Note: APIs and `client_id` change as you migrate to higher environments, but your `app_id` always remains the same.
    - API references for user login, user discovery, and token OIDC management
- What you provide for yourself
    - Method(s) for initiating the login process (call the OIDC /authorize API)
        - Application can automatically trigger the login if the user is not yet in session
        - Application should provide visual call to action (e.g., "Click here to login")
    - Redirect URI for receiving and handling the OIDC login response from AaaS
        - If using a software framework, this is likely already prebuilt as a native handler
        - If writing your own code, this will need to be able to consume HTTP responses and determine course of action based on URL variables received

Technical Details

OIDC login begins with an API call to the AaaS `/authorize` endpoint and continues as a series of URL redirects until the flow completes as a reply to the application `redirect_uri` endpoint. In between these endpoints AaaS handles user authentication and validation, so the application need not be concerned with the rigors of enterprise identity management and security. However, once the application receives the response from AaaS, it must determine how to treat the information and, consequently, how to direct the user experience.

Recommendations

To maximize security and minimize dependencies between systems, follow these recommendations:

- Use the appropriate pattern for your application architecture
    - For traditional application architectures with frontend and backend components, `redirect_uri` should be a server-side endpoint
        - Information is not exposed on the client browser
        - Signature verification is performed in a safe manner
        - User discovery and token management is performed in a safe environment
    - For SPA or other applications that run exclusively in the client, use PKCE
        - `redirect_uri` will be a client-side endpoint
        - Allows secure handling of the `/authorize` flow
        - Requires slightly more work in the application prior to calling the /authorize endpoint
        - Preserves the standard flow with simple modification to the `/authorize` request
- When calling `/authorize`, use `response_type=code`
    - On its own, the code cannot be used to call other OIDC methods
    - Retrieval of access token, ID token, and refresh token is deferred to backend processing, not exposed on the client browser
- Design your application with short token lifetimes in mind
    - Access token (default TTL 60 seconds) is used for retrieving user claims from the `/userinfo` endpoint, and the application should persist relevant user details (i.e., create a local application session) for the duration of the user's engagement in the application
    - ID token (default TTL 600 seconds) represents the user session in AaaS
    - Refresh token (default TTL 3600 seconds) can be activated if the application requires new access token and/or ID token
- For applications needing a user session to be established, create a local application user session separate from the OIDC ID token; do not rely on the ID token to control the user session
    - Application session policy is distinct from, and superior to, AaaS session policy
    - ID token only represents the user session on AaaS identity provider service

    - ID token can serve as a basis for initiating an application user session, but it should not be considered a general-purpose session
- For applications operating under stateless or zero-trust principles, leverage the OIDC refresh token
    - No local user session is established
    - For every action that needs to be verified, calling the OIDC `/token` endpoint with the refresh token ensures that AaaS vouches for the user every time; successful responses indicate the user is known and verified
    - Refresh token lifetime can be set to any length but it is recommended to limit it to a reasonable timeframe such as 8 hours to represent a workday; after expiration of the refresh token, the user must be reauthenticated

For guidance and best practices throughout the implementation, reach out to the AaaS team.

PKCE

PKCE is relevant for client-side apps during the most vulnerable part of the token exchange. Because the authorize request is HTTP GET, the `code` is seen in the response URL query string, so it is exposed on the client device. A man-in-the-middle with stolen credentials could see the vulnerable response, seize the code, and hence, gain control of the user's valid access token.

To protect against code hijacking in client-side applications, a PKCE-enabled application dynamically generates a random secret for the authorization request, so each user has a different secret and there's minimal concern about `client_secret` being compromised. Happily, PKCE does not modify the OIDC flow, it simply introduces additional request parameters to make it work.

Although the authorization `code` (resulting from the initial `/authorize` request) is harmless on its own, keep in mind that the `access_token` (retrieved from the `/token` request) allows anyone possessing it to make OIDC API calls, such as `/userinfo` and `/introspect`, so it is wise to protect the `access_token` on the client.

With PKCE safeguarding the original token transaction, only the initiating application can provide the correct `code_verifier` in the code-for-token transaction, and, because the `/token` request is HTTPS POST, the tokens are safe in the response body rather than visible (and vulnerable) in the query string. Subsequent refresh token requests are likewise performed as POST with everything safe in the response body.

Tokens Lifetimes

The three tokens have different times to live (TTL):

- `access_token` = 60 seconds
- `id_token` = 600 seconds (10 minutes)
- `refresh_token` = 3600 seconds (1 hour) but can be set to a maximum of 9 hours (roughly representing a workday, after which the user must reauthenticate)

Session Management

The reason the Access Token and ID Token are short lived is to impose a security posture where the user cannot maintain access if the user's session is terminated in AaaS. If the user's session needs to be verified after the Access Token and ID Token are expired, the Refresh Token is used to retrieve fresh tokens. To maintain user access independent of these restrictions, the application should establish its own local session.

Session policy depends on the application type. An application that tracks a user's state should establish its own local session, which may be derived from the ID Token but should not be controlled by it. The Access Token is deliberately short lived but is long enough that it can be used for any relevant OIDC endpoint requests, such as the `/user_info` endpoint for retrieving user details. If the app is client-side only (e.g., SPA) without backend session support or operates in a zero-trust environment, the Refresh Token would be necessary.

The Refresh Token is particularly useful in zero-trust environments where every action must be verified. Generally, whenever verification is required but tokens are expired, the Refresh Token is used to generate a new tokens, thereby proving the user is still valid according to the OIDC Provider (OP).

Logout / Session Termination

AaaS operates as an SSO service, so the application never actually logs the user out from AaaS. Applications with local session policy must terminate the local session, which technically logs the user out from the application, but the user remains logged in on AaaS to preserve other SSO sessions the user may have on other applications. To formally terminate AaaS communication with the application, the Access Token and Refresh Token must be invalidated via the `/revoke` endpoint to prevent further usage.

During the logout process, the application is responsible for preventing the user from navigating back to the AaaS login sequence, otherwise the user likely will be automatically logged in again. To prevent this flow, some sort of simple landing page is needed with a "click here to login" message. This page is a safe place for anyone to land without automatically triggering the SSO flow via AaaS.

Implementation Notes

Discovery Endpoints

Each of the three AaaS environments has its own configuration specified in its "well-known" endpoint where you can find important details like the published OIDC endpoints and certificate key URI.

- AaaS TEST:
  https://tst.aaas.cencora.com/oidc/op/v1.0/4_Pv18t6XTOc51PxyYytQzHA/.well-known/openid-configuration
- AaaS STAGE (UAT):
  https://stg.aaas.cencora.com/oidc/op/v1.0/4_pgDlm4GH1hYhGHvA0F0tVw/.well-known/openid-configuration
- AaaS PROD:
  https://aaas.cencora.com/oidc/op/v1.0/4_PGj6CfqhMMUKzG9BHAEdwA/.well-known/openid-configuration

(Note the distinct domain and API key element per environment.)

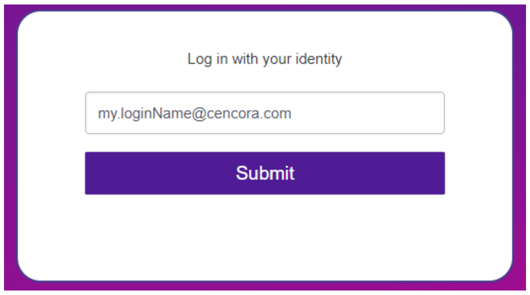
Elements of the /authorize Endpoint

All OIDC flows begin with the `/authorize` endpoint as an HTTP GET request containing these required elements:

Element	Value	Description
`client_id`	Value changes for each of the AaaS environments	Application's unique OIDC context in AaaS
`scope`	`openid email profile uid` [[YOUR_APP_ID]]	Details about the user to be retrieved
`response_type`	code	Artifact returned from the `/authorize` sequence
`redirect_uri`	Any valid URI to receive the `/authorize` response	Endpoint for receiving OIDC responses; must be registered in Aaas for the request to work
`nonce`	Any valid one-time-use value	Random, one-time-use-value to add entropy
`login_hint`	Provided by the AaaS team	Unique application ID in AaaS

Test User Accounts

When the `/authorize` request is submitted, the AaaS login window appears:



To login on any AaaS environment, use your @cencora.com email address or any of these test accounts:

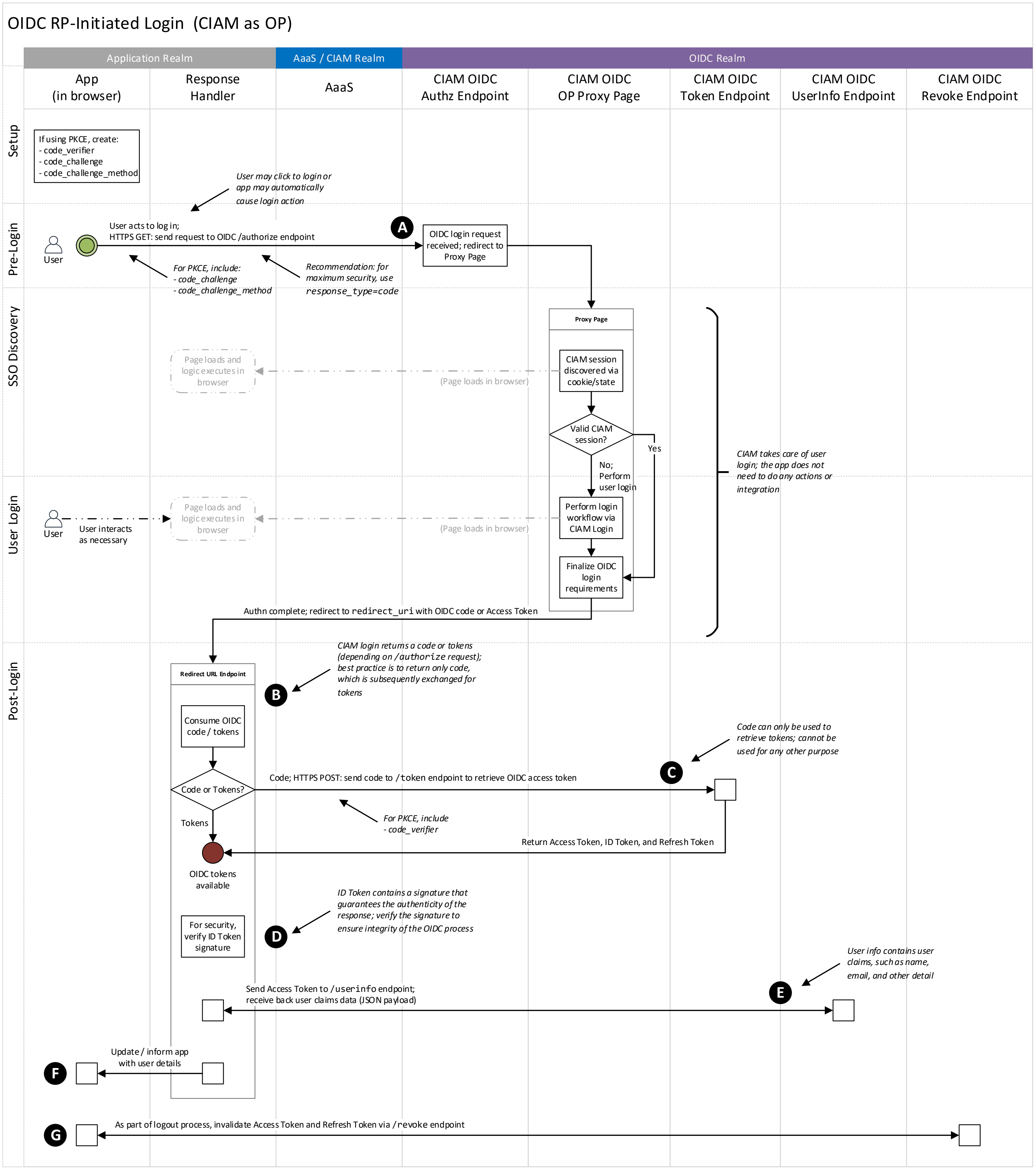
test_001@aaastest.com
test_002@aaastest.com
test_003@aaastest.com
test_004@aaastest.com
test_005@aaastest.com

The password is the same for all test accounts: p@ssw0rd

OIDC Flow and APIs

The following graphic illustrates the flow of control throughout the OIDC process, and sample API requests/responses assist with understanding how to implement the OIDC endpoints within your application. The lettered markers on the flow diagram indicate where significant activities occur, with matching code samples illustrated in the subsequent graphic.

OIDC Login Flow


OIDC RP-Initiated Login (CIAM as OP)
Application Realm
AaaS / CIAM Realm
OIDC Realm
App (in browser)
Response Handler
AaaS
CIAM OIDC Authz Endpoint
CIAM OIDC OP Proxy Page
CIAM OIDC Token Endpoint
CIAM OIDC UserInfo Endpoint
CIAM OIDC Revoke Endpoint
Setup
Pre-Login
SSO Discovery
User Login
Post-Login
If using PKCE, create: - code_verifier - code_challenge - code_challenge_method
User may click to login or app may automatically cause login action
User
User acts to log in; HTTPS GET: send request to OIDC /authorize endpoint
A
OIDC login request received; redirect to Proxy Page
For PKCE, include: - code_challenge - code_challenge_method
Recommendation: for maximum security, use response_type=code
Proxy Page
Page loads and logic executes in browser
(Page loads in browser)
CIAM session discovered via cookie/state
Valid CIAM session?
Yes
No; Perform user login
CIAM takes care of user login; the app does not need to do any actions or integration
User
User interacts as necessary
Page loads and logic executes in browser
(Page loads in browser)
Perform login workflow via CIAM Login
Finalize OIDC login requirements
Authn complete; redirect to redirect_uri with OIDC code or Access Token
Redirect URL Endpoint
CIAM login returns a code or tokens (depending on /authorize request); best practice is to return only code, which is subsequently exchanged for tokens
B
Consume OIDC code / tokens
Code or Tokens?
Code; HTTPS POST: send code to /token endpoint to retrieve OIDC access token
C
Code can only be used to retrieve tokens; cannot be used for any other purpose
For PKCE, include - code_verifier
Tokens
Return Access Token, ID Token, and Refresh Token
OIDC tokens available
ID Token contains a signature that guarantees the authenticity of the response; verify the signature to ensure integrity of the OIDC process
D
For security, verify ID Token signature
Send Access Token to /userinfo endpoint; receive back user claims data (JSON payload)
E
User info contains user claims, such as name, email, and other detail
Update / inform app with user details
F
G
As part of logout process, invalidate Access Token and Refresh Token via /revoke endpoint

OIDC API Samples

These API samples illustrate the outcomes expected at distinct points in the OIDC login flow. Variables notated in double brackets to indicate where [[YOUR_VARIABLE_VALUE]] should be substituted.

- [[AAAS_DOMAIN_AND_PATH]], [[YOUR_CLIENT_ID]] and [[YOUR_APP_ID]] – provided by the AaaS team during configuration
- [[YOUR_REDIRECT_URI]] – determined by you and shared with the AaaS team for configuration;
  if your `redirect_uri` is not registered with AaaS, the authorization request will fail.

A Application initiates OIDC login with HTTPS GET /authorize request; this example uses `response_type=code`, but other variants are allowed.

```
         curl --location 'https://[[AAAS_DOMAIN_AND_PATH]]/authorize
            ?client_id=[[YOUR_CLIENT_ID]]
            &scope=openid%20email%20profile%20uid
            &response_type=code
            &redirect_uri=https://[[YOUR_REDIRECT_URI]]'
            &nonce=[[UNIQUE_NONCE]]
            &login_hint=[[YOUR_APP_ID]]
If PKCE {   &code_challenge_method=(plain|S256)
            &code_challenge=[[encoded code_verifier]]
```

—— Response ——▶

B Response delivered to redirect_uri; although the code can be exchanged (step C) from the frontend client, ideally it would be done from the backend for maximum security.

```
Https://[[YOUR_REDIRECT_URI]]
   &code=st2.s.AtLtez4-OA.hcmNFt...2JFYqg.sc3

or, if /authorize request includes response_mode=fragment:

Https://[[YOUR_REDIRECT_URI]]
   #code=st2.s.AtLtez4-OA.hcmNFt...2JFYqg.sc3
```

C Send code (result of step B) to HTTPS POST /token endpoint to retrieve access token, ID token, and refresh token

```
         curl --location 'https://[[AAAS_DOMAIN_AND_PATH]]/token
            ?grant_type=authorization_code
            &code=[[CODE_FROM_STEP_B]]
            &redirect_uri=[[YOUR_REDIRECT_URI]]
If PKCE {   &code_verifier=[[code_verifier]]' \
         --header 'Content-Type: application/x-www-form-urlencoded' \
         --header 'Authorization: Basic SmRGYU...JYYjJR' \
         --form 'Content-Type="application/x-www-form-urlencoded"'
```

—— Response ——▶

```
{
    "access_token": "st2.s.AtLtBTyVMQ.3kNsyi...useCEw.sc3",
    "token_type": "Bearer",
    "expires_in": 60,
    "id_token": "eyJ0eX...vsf37Q",
    "refresh_token": "st2.s.AtLtMu...TZyq-Q.sc3"
}
```

D ID token (result of step C) must be verified to ensure the integrity of the OIDC response; see "ID Token Verification" for instructions

E Use access_token to retrieve HTTPS GET /userinfo details

```
curl --location 'https://[[AAAS_DOMAIN_AND_PATH]]/userinfo' \
--header 'Authorization: Bearer [[ACCESS_TOKEN_FROM_STEP_C]]'
```

—— Response ——▶

```
{
    "sub": "-qI2H-stnoT9se60ZqE-0TdepWZ-mPsa-EnOqOXCCT0",
    "name": "John Doe",
    "family_name": "Doe",
    "given_name": "John",
    "email": "John.Doe@cencora.com",
    "uid": "4cb3750cf6d446e1957786099c8fa8a7"
}
```

F Application uses ID token information (step D) to establish local session and state, and user info details (step E) for authorization, personalization, etc.

G Invalidate Access Token and Refresh Token via HTTP POST /revoke endpoint

```
curl --location 'https://[[AAAS_DOMAIN_AND_PATH]]/revoke' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'Authorization: Basic [[HTTP_AUTH_TOKEN]]' \
--data-urlencode 'token=[[access_token or refresh_token]]' \
--data-urlencode 'token_type_hint=(access_token |
   refresh_token)'
```

—— Response ——▶

```
{
    "callId": "445b454b7430dbc1994d5ff318a386bc",
    "errorCode": 0,
    "apiVersion": 2,
    "statusCode": 200,
    "statusReason": "OK",
    "time": "2024-08-21T18:52:33.819Z"
}
```

OIDC ID Token

From the OIDC specification:

> *The ID Token is a security token that contains Claims about the Authentication of an End-User by an Authorization Server when using a Client, and potentially other requested Claims. The ID Token is represented as a JSON Web Token (JWT).*

Basically, this means that the ID token:

- Bundles a lot of information in an **encoded structure** (JWT)
- Contains **authoritative detail** about the user login
- Ensures that data received by the client is trustworthy based on its **verifiable digital signature**

We explore each of these points in this section.

JWT Structure

JSON Web Token (JWT) is an open standard 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.

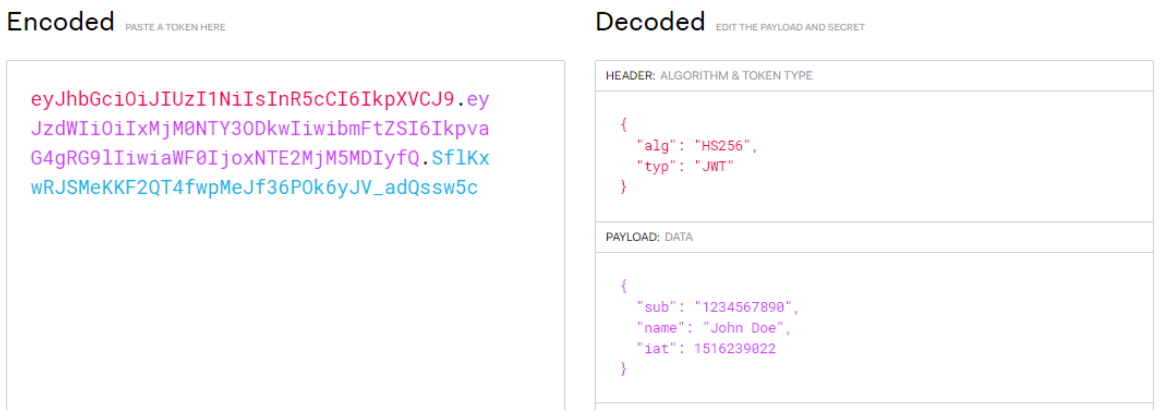
The structure of JWT consist of three JSON parts separated by dots (.): {Header}.{Payload}.{Signature}

- Header – contains hints about how the JWT is secured
- Payload – information and claims about the user login and session on the identity provider (AaaS)
- Signature – verifiable digital value that proves all information is authentic and unchanged from the source

All three segments are Base64-URL encoded, so the result is three strings separated by dots that can be easily passed in HTML and HTTP environments. Here is a color-coded example of what it looks like:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

A useful JWT encoding/decoding tool is found at https://jwt.io/ where you can copy/paste the above value to see the JWT contents, such as this snapshot:


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

For more detailed JWT information, see https://jwt.io/introduction

Authoritative Detail

The JWT header segment contains information about the methods used for creating the JWT. For signature verification we are most interested in the algorithm (`alg`) and key ID (`kid`) values.

The JWT payload segment contains useful information about the AaaS user session and, potentially, user details/claims. You can leverage the session details in determining the application user session, but remember the AaaS session is separate and independent of the application session.

Verifying the Signature

The JWT signature segment is an encrypted value, so it is not human readable (even after unencoding it from Base64). To ensure the JWT is altogether trustworthy, verify the signature by recreating the exact algorithmic outcome using information that has been provided in the JWT itself. Generally, the process involves applying an encryption algorithm to the combination of the Base64 encoded header and payload along with an AaaS public key, but the specific process will depend on a) the algorithm to be applied and b) your software framework.

The algorithm to use is denoted in the "`alg`" value found in the JWT header segment. RS256 (RSA using SHA256; asymmetric algorithm) and HS256 (HMAC with SHA-256; symmetric algorithm) are the most common algorithms used for JWT signing.

The AaaS public key is retrieved from within the JWKS URI found in the publicly available AaaS OIDC configuration:

https://[[AAAS_DOMAIN_AND_PATH]]/.well-known/openid-configuration

The configuration defines a value for `jwks_uri` (JWKS endpoint) [1]:

https://[[AAAS_DOMAIN_AND_PATH]]/.well-known/jwks

The response from the JWKS endpoint contains an array of all valid AaaS public keys (defined in a JSON structure), but we must only use the key identified with the matching `kid` from the JWT header.

Example JWT header:

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "RjBEQ0FDN0U0QzgwN0FGNjM2MTQ5QzIwQkYwQUNBRDlDRTA2MUY3MQ"
}
```

Example JWKS response:

```
{
  "keys": [
    {
      "alg": "RS256",
      "e": "AQAB",
      "kid": "RjBEQ0FDN0U0QzgwN0FGNjM2MTQ5QzIwQkYwQUNBRDlDRTA2MUY3MQ",
      "kty": "RSA",
      "n": "sgzA0R...qV4Jjw",
      "use": "sig"
    },
    {
      "alg": "RS256",
      "kid": "GmnDGH...MUY3MQ",
      // other details removed
    }
  ]
}
```

Use the JWT header `kid` to find the matching public key for signature verification

This JSON structure defines the key

Extract the complete JSON structure of the key that matches the `kid` from the JWT header. As seen in the above example, the `alg` value also must match, specifying the algorithm to use for verification. Applying the algorithm to the inputs must produce the exact same value as the JWT signature value.

The signature verification process depends on the algorithm and functional support within your specific environment. If you use a software development framework, cryptographic support is probably available within the framework; if you are writing your own code, refer to the cryptographic capabilities of your software platform/language.

[1] Example of JWKS for AaaS TEST environment: https://tst.aaas.cencora.com/oidc/op/v1.0/4_Pv18t6XTOc51PxyYytQzHA/.well-known/jwks

(To find information about verifying cryptographic signatures in your specific environment, start with a web search for "[[YOUR_ENVIRONMENT]] cryptography" where you can find articles and postings with sample code.)

AaaS currently uses RSASHA256 as the default algorithm, which is denoted in both the JWT header and public key definition: `"alg":"RS256"`. In pseudocode, the signature verification looks as follows:

```
RSASHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  [[public key formatted according to algorithm requirements]]
)

if (RSASHA256 result exactly matches the JWT signature segment) { do stuff } else { ignore the request }
```

When the result of the signing algorithm exactly matches the JWT signature, you know the JWT is trustworthy; if the result does not exactly match, the JWT is not trustworthy, and the request must be ignored.

Conclusion

Authentication-as-a-Service simplifies the integration of enterprise user management and login within your application, allowing you to focus on the business purpose rather than the intricacies of identity management. From concept to deployment, the AaaS team is available to provide guidance and best practice along the way, just let us know how we can help.