OpenID Connect 1.0 Developer Guide

Contents

OpenID Connect Developer Guide

This document provides a developer overview of the OpenID Connect 1.0 protocol (OIDC) and provides instructions for an Application Developer to implement OpenID Connect with PingFederate. Two walkthroughs are provided to demonstrate the OpenID Connect Basic Client Profile and the OpenID Connect Implicit Client Profile.

This is targeted to developers, however the content will be relevant for infrastructure owners to understand the OpenID Connect concepts. Explanations and code examples are provided for "quick win" integration efforts. As such they are incomplete and meant to complement existing documentation and specifications.

This document assumes a basic familiarity with the OpenID Connect 1.0 protocol and the OAuth 2.0 protocol. For more information about OAuth 2.0 and OpenID Connect 1.0, refer to:

- *PingFederate Administrator's Manual*
- *OpenID Connect 1.0 Specifications*
- *OAuth 2.0 developers guide*
- *OAuth 2.0 Specifications*

ⓘ **Note:** This document explains a number of manual processes to request and validate the OAuth and OpenID Connect tokens. While the interactions are simple, PingFederate is compatible with many 3rd party OAuth and OpenID Connect client libraries that may simplify development effort.

What is OpenID Connect

The OpenID Connect protocol extends the OAuth 2.0 protocol to add an authentication and identity layer for application developers. Where OAuth 2.0 provides the application developer with security tokens to be able to call back-end resources on behalf of an end-user; OpenID Connect provides the application with information about the end-user, the context of their authentication, and access to their profile information.

Two new concepts are introduced on top of the OAuth 2.0 authorization framework:

- an OpenID Connect "ID token" which contains information around the user's authenticated session and
- a UserInfo endpoint which provides a means for the client to retrieve additional attributes about the user

OpenID Connect uses the same actors and processes as OAuth 2.0 to get the ID token, and protects the UserInfo endpoint with the OAuth 2.0 framework.

Application Developer Considerations

There are three main actions an application developer needs to handle to implement OpenID Connect:

1. Get an OpenID Connect id_token By leveraging an OAuth2 grant type, an application will request an OpenID Connect id_token by including the "openid" scope in the authorization request.
2. Validate the id_token Validate the id_token to ensure it originated from a trusted issuer and that the contents have not been tampered with during transit.
3. Retrieve profile information from the UserInfo endpoint Using the OAuth2 access token, access the UserInfo endpoint to retrieve profile information about the authenticated user.

The ID Token

The ID token is a token used to identify an end-user to the client application and to provide data around the context of that authentication.

An ID token will be in the JSON Web Token (JWT) format. In most cases the ID token will be signed according to JSON Web Signing (JWS) specifications, however depending on the client profile used the verification of this signature may be optional.

ⓘ **Note:** When the id_token is received from the token endpoint via a secure transport channel (i.e. via the Authorization Code grant type) the verification of the digital signature is optional.

Decoding the ID Token

The id_token JWT consists of three components, a header, a payload and the digital signature. Following the JSON Web Token (JWT) standard, these three sections are Base64url encoded and separated by periods (.).

ⓘ **Note:** JWT and OpenID Connect assume base64url encoding/decoding. This is slightly different than regular base64 encoding. Refer to RFC4648 for specifics regarding Base64 vs Base64 URL safe encoding.

The following example describes how to manually parse a sample ID token provided below:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6Imkwd25uIn0.eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWNfY2xpZW50Iiwi
dzY0ZoY3RVVDZEdHZiMiIsImlzcyI6Imh0dHBzOlwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6MTM5NDA2MDg1My
MTUzLCJub25jZSI6ImU5NTdmZmJhLTlhNzgtNGVhOS04ZWNhLWFlOGM0ZWY5Yzg1NiIsImF0X2hhc2giOiJ3Zmd2
M2VHFBIn0.lr4L-oT7DJi7Re0eSZDstAdOKHwSvjZfR-
OpdWSOmsrw0QVeI7oaIcehyKUFpPFDXDR0-RsEzqno0yek-_U-Ui5EM-yv0Pia
UOmJK1U-ws_C-fCplUFSE7SK-
TrCwaOow4_7FN5L4i4NAa_WqgOjZPloT8o3kKyTkBL7GdITL8rEe4BDK8L6mLqHJrFX4SsEduPk0CyHJS
ykRqzYS2MEJlncocBBI4up5Y5g2BNEb0aV4VZwYjmrv9oOUC_yC1Fb4Js5Ry1t6P4Q8q_2ka5OcArlo188XH7lMq
N7S46ubGPXRBNsnrPx6RuoR2cI46d9ARQ
```

ⓘ **Note:** It is strongly recommended to make use of common libraries for JWT and JWS processing to avoid introducing implementation specific bugs.

The above JWT token is first split by periods (.) into three components:

JWT Header

Contains the algorithm and a reference to the appropriate public key if applicable:

Component	Value	Value Decoded
JWT Header	eyJhbGciOiJSUzI1NiIsImtpZCI6Imkwd25uIn0	{"alg":"RS256", "kid":"i0wnn" }

JWT Payload

The second component contains the payload which contains claims relating to the authentication and identification of the user. The payload of the above example is decoded as follows:

Component	Value	Value Decoded
JWT Payload	eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWRjX fY2xpZW50IiwianRpIjoidWY5MFNLNHd2Yi ZoY3RVVDZEdHZiMiIsImlzcyI6ImhjdHBzOi lwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6 MTM5NDA2MDg1MywiZXhwIjoxMzk0MDM5NA zLCJub25jZSI6ImU5NTdmZmJhLTlhOTgtNG VhOS04ZWNhLWFlOGM0ZWY5YzgiLCJhdXRo 2hhc2giOiJ3Zmd2bUU5VnhqQXVkc2w5bGM2 VHFBIn0	{"sub":"joe", "aud":"im_oic_client", "jti":"Bf90SK4wscFhctUT6Dtvb2", "iss":"https:\/\/localhost:9031", "iat":1394061153, "exp":1394061853, "nonce":"E957ffba-9a78-4ea9-8eca- 2e854e9f2856", "at_hash":"wfgvmE9VxjAudsl9lc6TqA" }

The following claims you can expect in an id_token and can use to determine if the authentication by the user was sufficient to grant them access to the application. (Refer to the OpenID Connect specifications to additional details on these attributes):

Claim	Description
iss	Issuer of the id_token
sub	Subject of the id_token (ie the end-user's username)
aud	Audience for the id_token (must match the client_id of the application)
exp	Time the id_token is set to expire (UTC, Unix Epoch time)
iat	Timestamp when the id_token was issued (UTC, Unix Epoch time)
auth_time	Time the end-user authenticated (UTC, Unix Epoch time)
nonce	Nonce value supplied during the authentication request (REQUIRED for implicit flow)
acr	Authentication context reference used to authenticate the user
acr	Authentication context reference used to authenticate the user
at_hash	Hash of the OAuth2 access token when used with Implicit profile
c_hash	Hash of the OAuth2 authorization code when used with the hybrid profile

Digital Signature

Base64 URL encoded signature of section 1 and 2 (period concatenated). The algorithm and key reference used to create and verify the signature is defined in the JWT Header.

Component	Value	Value Decoded
JWT Signature	lr4L-oT7DJi7Re0eSZDstAdOKHwSvjZfR-OpdWSOmsrw0QVeI7oaIcehyKUFpPFDXDR0-RsEzqno0yek-_U-Ui5EM-yv0PiaUOmJK1U-ws_C-f CplUFSE7SK-TrCwaOow4_7FN5L4i-4NAa_WqgOjZPloT8o3kKyTkBL7GdITL8rEe4BDK8L6mLqHJrFX4SsEduPk0CyHJSykRqzYS2MEJlncocBBI4up5Y5g2BNEb0aV4VZwYjmrv9oOUC_yC1Fb4Js5Ry1t6P4Q8q_2ka5OcArlo188XH7lMgPA2GnwSFGHBhccjpxhN7S46ubGPXRBNsnrPx6RuoR2cI46d9ARQ	N/A

Validating the ID Token

The validation of the ID token includes evaluating both the payload and the digital signature.

Payload Validation

The ID token represents an authenticated user's session. As such the token must be validate before an application can trust the contents of the ID token. For example, if a malicious attacker replayed a user's id_token that they had captured earlier the application should detect that the token has been replayed or was used after it had expired and deny the authentication.

Refer to the OpenID Connect specifications for more information on security concerns. The specifications also include guidelines for validating an ID token (Core specification section 3.1.3.7). The general process would be as follows:

Step #	Test Summary
1	Decrypt the token (if encrypted)
2	Verify the issuer claim (iss) matches the OP issuer value
3	Verify the audience claim (aud) contains the OAuth2 client_id
4	If the token contain multiple audiences, then verify that an Authorized Party claim (azp) is present
5	If the azp claim is present, verify it matches the OAuth2 client_id
6, 7 & 8	Optionally verify the digital signature (required for implicit client profile) (see section 4.4)
9	Verify the current time is prior to the expiry claim (exp) time value
10	Client specific: Verify the token was issued within an acceptable timeframe (iat)
11	If the nonce claim (nonce) is present, verify that it matches the nonce passed in the authentication request
12	Client specific: Verify the Authn Context Reference claim (acr) value is appropriate

Step #	Test Summary
13	Client specific: If the authentication time claim (auth_time) present, verify it is within an acceptable range
14	If the implicit client profile is used, verify that the access token hash claim (at_hash) matches the hash of the associated access_token

Signature Validation

ⓘ **Note:** Signature validation is only required for tokens not received directly from the token endpoint (i.e. for the Implicit Client Profile). In other cases where the id_token is received directly by the client from the token endpoint over HTTPS, transport layer security should be sufficient to vouch for the integrity of the token.

The ID token is signed according to the JSON Web Signature (JWS) specification; algorithms used for signing are defined in the JSON Web Algorithm (JWA) specification. PingFederate 7.1 can support the following signing algorithms:

"alg" Value	Signature Method	Signing Key
NONE	No Digital Signature	N/A
HS256	HMAC w/ SHA-256 hash	Uses the client secret of the OAuth2 client
HS384	HMAC w/ SHA-384 hash	Uses the client secret of the OAuth2 client
HS512	HMAC w/ SHA-512 hash	Uses the client secret of the OAuth2 client
RS256	RSA PKCS v1.5 w/ SHA-256 hash	Public key available from the JWKS (see below)
RS384	RSA PKCS v1.5 w/ SHA-384 hash	Public key available from the JWKS (see below)
RS512	RSA PKCS v1.5 w/ SHA-512 hash	Public key available from the JWKS (see below)
ES256	ECDSA w/ P-256 curve and SHA-256 hash	Public key available from the JWKS (see below)
ES384	ECDSA w/ P-384 curve and SHA-384 hash	Public key available from the JWKS (see below)
ES512	ECDSA w/ P-521 curve and SHA-512 hash	Public key available from the JWKS (see below)

ⓘ **Note:** RS256 is the default signature algorithm.

The basic steps to verify a digital signature involve retrieving the appropriate key to use for the signature verification and then performing the cryptographic action to verify the signature.

To validate the signature, take the JWT header and the JWT payload and join with a period. Validate that value against the third component of the JWT using the algorithm defined in the JWT header. Using the above ID token as an example:

Signed data (JWT Header + "." + JWT Payload):

```
eyJhbGciOiJSUzI1NiIsImtpZCI6Imkwd25uIn0.eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWNfY2xpZW50Iiw
Y0ZoY3RVVDZEdHZiMiIsImlzcyI6Imh0dHBzOlwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6MTM5NDA2MDg1Myw
LCJub25jZSI6ImU5NTdmZmJhLTlhNzgtNGVhOS04ZWNhLWFlOGM0ZWY5Yzg1NiIsImF0X2hhc2giOiJ3Zmd2bUU
In0
```

Signature value to verify:

```
lr4L-oT7DJi7Re0eSZDstAdOKHwSvjZfR-OpdWSOmsrw0QVeI7oaIcehyKUFpPFDXDR0-
RsEzqno0yek-_U-Ui5EM-yv0PiaUOmJK1U-ws_C
-fCplUFSE7SK-
TrCwaOow4_7FN5L4i-4NAa_WqgOjZPloT8o3kKyTkBL7GdITL8rEe4BDK8L6mLqHJrFX4SsEduPk0CyHJSykRqzY
cocBBI4up5Y5g2BNEb0aV4VZwYjmrv9oOUC_yC1Fb4Js5Ry1t6P4Q8q_2ka5OcArlo188XH7lMgPA2GnwSFGHBhc
rPx6RuoR2cI46d9ARQ
```

ⓘ **Note:** The actual implementation of the signing algorithm used to validate the signature will be implementation specific. It is recommended to use a published library to perform the signature verification.

For symmetric key signature methods, the client secret value for the OAuth2 client is used as the shared symmetric key. For this reason the client secret defined for the OAuth2 client must be of a large enough length to accommodate the appropriate algorithm (i.e. for a SHA256 hash, the secret must be at least 256 bits "“ 32 ASCII characters).

Asymmetric signature methods require the application to know the corresponding public key. The public key can be distributed out-of-band or can be retrieved dynamically via the JSON Web Key Set (JWKS) endpoint as explained below:

1. Determine the signing algorithm (alg) and the key identifier (kid) from the JWT header.Â Using the sample JWT token above as an example, the following values are known:

OpenID Connect issuer	https://localhost:9031
Signing algorithm (alg)	RS256
Key reference identifier (kid)	i0wnn

2. Query the OpenID configuration URL for the location of the JWKS:

```
GET https://localhost:9031/.well-known/openid-configuration HTTP/1.1
```

this will result in a HTTP response containing the OpenID Connect configuration for the OpenID Connect Provider (OP) :

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "version":"3.0",

  "issuer":"https:\/\/localhost:9031",

  "authorization_endpoint":"https:\/\/localhost:9031\/as\/
authorization.oauth2",
```

```
  "token_endpoint":"https:\/\/localhost:9031\/as\/token.oauth2",

  "userinfo_endpoint":"https:\/\/localhost:9031\/idp\/userinfo.openid",

  "jwks_uri":"https:\/\/localhost:9031\/pf\/JWKS",

  "scopes_supported":
["phone","address","email","admin","edit","openid","profile"],

  "response_types_supported":["code","token","id_token","code token",
    "code id_token","token id_token","code token id_token"],

  "subject_types_supported":["public"],

  "id_token_signing_alg_values_supported":
["none","HS256","HS384","HS512","RS256",
    "RS384","RS512","ES256","ES384","ES512"],

  "token_endpoint_auth_methods_supported":
["client_secret_basic","client_secret_post"],

  "claim_types_supported":["normal"],

  "claims_parameter_supported":false,

  "request_parameter_supported":false,

  "request_uri_parameter_supported":false
}
```

3. Parse the JSON to retrieve the jwks_uri value (bolded above) and make a request to that endpoint, JSON Web Keystore (JWKS), to retrieve the public key for key identifier "i0wnn" and key type (kty) of RSA as the algorithm is RS256 that was used to sign the JWT:

```
GET https://localhost:9031/pf/JWKS HTTP/1.1
```

Which will return the JWKS for the issuer:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "keys":[
    {
      "kty":"EC",
      "kid":"i0wng",
      "use":"sig",
      "x":"AXYMGFO6K_R2E3RH42_5YTeGYgYTagLM-
v3iaiNlPKFFvTh17CKQL_OKH5pEkj5U8mbel-0R1YrNuraRXtBztcVO",
      "y":"AaYuq27czYSrbFQUMo3jVK2hrW8KZ75KyE8dyYS-
HOB9vUC4nMvoPGbu2hE_yBTLZLpuUvTOSSv150FLaBPhPLA2",
      "crv":"P-521"
    },
    ...
    {
      "kty":"RSA",
      "kid":"i0wnn",
```

```
      "use":"sig",

 "n":"mdrLAp5GR8o5d5qbwWTYqNGuSXHTIE6w9HxV445oMACOWRuwlOGVZeKJQXHM9cs5Dm7iUfNVk4pJBttUxz
      9tr20LJB7xAAqnFtzD7jBHARWbgJYR0p0JYVOA5jVzT9Sc-j4Gs5m8b-
am2hKF93kA4fM8oeg18V_xeZf11WWcxnW5YZwX
      9kjGBwbK-1tkapIar8K1WrsAsDDZLS_y7Qp0S83fAPgubFGYdST71s-B4bvsjCgl30a2W-
je9J6jg2bYxZeJf982dzHFqV
      QF7KdF4n5UGFAvNMRZ3xVoV4JzHDg4xe_KJE-gOn-_wlao6R8xWcedZjTmDhqqvUw",
      "e":"AQAB"
    },
    ...
  ]
}
```

We now have the modulus (n) and the exponent (e) of the public key. This can be used to create the public key and validate the signature.

ⓘ **Note:** The public key can be stored in secure storage (i.e. in the keychain) to be used for verification of the id_token when a user is offline.

Validating the token hashes (at_hash, c_hash)

We now have the modulus (n) and the exponent (e) of the public key. This can be used to create the public key and validate the signature.

In specific client profiles, a specific hash is included in the id_token to use to verify that the associated token was issued along with the id_token. For example, when using the implicit client profile, an at_hash value is included in the id_token that provides a means to verify that the access_token was issued along with the id_token.

The following example uses the id_token above and associated access_token to verify the at_hash id_token claim:

Signing algorithm	RS256
at_hash value	wfgvmE9VxjAudsl9lc6TqA
OAuth 2.0 access_token	dNZX1hEZ9wBCzNL40Upu646bdzQA

1. Hash the octets of the ASCII representation of the access token (using the hash algorithm specified in the JWT header (i.e. for this example, RS256 uses a SHA-256 hash)): SHA256HASH("dNZX1hEZ9wBCzNL40Upu646bdzQA") = c1f82f98 4f55c630 2e76c97d 95ce93a8 9a5d61f7 dc99b9ad 37dc12b3 7231ff9d
2. Take the left-most half of the hashed access token and Base64url encode the value. Left-most half: c1f82f98 4f55c630 2e76c97d 95ce93a8 Base64urlencode([0xC1, 0xF8, 0x2F, 0x98, 0x4F, 0x55, 0xC6, 0x30, 0x2E, 0x76, 0xC9, 0x7D, 0x95, 0xCE, 0x93, 0xA8]) = "wfgvmE9VxjAudsl9lc6TqA"
3. Compare the at_hash value to the base64 URL encoded left-most half of the access token hash bytes.

at_hash value	wfgvmE9VxjAudsl9lc6TqA
left-most half value	wfgvmE9VxjAudsl9lc6TqA
Validation result	VALID

The UserInfo Endpoint

The OpenID Connect UserInfo endpoint is used by an application to retrieve profile information about the Identity that authenticated. Applications can use this endpoint to retrieve profile information, preferences and other user-specific information.

The OpenID Connect profile consists of two components:

- Claims describing the end-user
- UserInfo endpoint providing a mechanism to retrieve these claims

ⓘ **Note:** The user claims can also be presented inside the id_token to eliminate a call back during authentication time.

User Profile Claims

The UserInfo endpoint will present a set of claims based on the OAuth2 scopes presented in the authentication request.

OpenID Connect defines five scope values that map to a specific set of default claims. PingFederate allows you to extend the "profile" scope via the "OpenID Connect Policy Management" section of the administration console. Multiple policy sets can be created and associated on a per-client basis.

Connect scope	Returned Claims
openid	None - Indicates this is an OpenID Connect request
profile	name, family_name, given_name, middle_name, nickname, preferred_username, profile, picture, website, gender, birthdate, zoneinfo, locale, updated_at, *custom attributes
address	address
email	email, email_verified
phone	phone_number, phone_number_verified

ⓘ **Note:**

- If a scope is omitted (i.e. the "email" scope is not present), the claim "email" will not be present in the returned claims. For custom profile attributes, prefix the value to avoid clashing with the default claim names.
- If an OpenID Connect id_token is requested without an OAuth2 access token (i.e. when using the implicit "response_type = id_token" request), the claims will be returned in the id_token rather than the UserInfo endpoint.

Sample UserInfo Endpoint Request

Once the client application has authenticated a user and is in possession of an access token, the client can then make a request to the UserInfo endpoint to retrieve the requested attributes about a user. The request will include the access token presented using a method described in RFC6750.

The UserInfo endpoint provided by PingFederate is located at: https://<pingfederate_base_url>/idp/userinfo.openid

ⓘ **Note:** The UserInfo endpoint can also be determined by querying the OpenID Connect configuration information endpoint: https://<pingfederate_base_url>/.well-known/openid-configuration.

An example HTTP client request to the UserInfo endpoint:

```
GET https://pf.company.com:9031/idp/userinfo.openid HTTP/1.1

Authorization: Bearer
```

A successful response will return a HTTP 200 OK response and the users claims in JSON format:

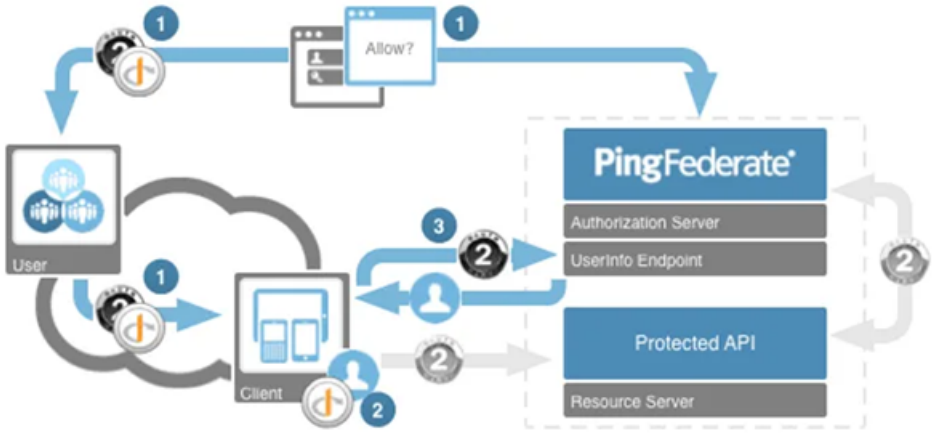
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "sub":"mpavlich",
  "family_name":"Pavlich",
  "given_name":"Matthew",
  "nickname":"Pav",
  ...[additional claims]...
}
```

Before the client application can trust the values returned from the UserInfo endpoint (i.e. as a check for token substitution attack), the client must verify that the "sub" claim returned from the UserInfo endpoint request matches the subject from the id_token.

Implicit Client Profile

The OpenID Connect 1.0 Implicit Client Profile uses the OAuth 2.0 "Implicit" grant type. The flow is almost identical to the OAuth 2.0 implicit flow with the exception of the "openid" scope and the tokens returned.

This section provides an example of using OpenID Connect Implicit Client Profile to retrieve an OpenID Connect id_token, validate the contents (steps 1 and 2 in the diagram below) and then query the UserInfo endpoint to retrieve profile information about the user (step 3).



This example assumes PingFederate 7.3 or higher is installed with the OAuth 2.0 Playground developer tool. The following configuration will be used:

PingFederate server base URL	https://localhost:9031
OAuth 2.0 client_id	m_oic_client
OAuth 2.0 client_secret	< none >
Application callback URI	https://localhost:9031/OAuthPlayground/case2A-callback.jsp

ⓘ **Note:** For native mobile applications, the callback URI may be a non-http URI. This is configured in your application settings and will cause the mobile application to be launched to process the callback.

Step 1: Authenticate the End-User and Receive Tokens

The initial user authentication request follows the OAuth2 Implicit Grant Type flow. To initiate the OpenID Connect process, the user will be redirected to the OAuth2 authorization endpoint. The request is made to the authorization endpoint with the following parameters:

client_id	im_oic_client
response_type	token id_token
redirect_uri	https://localhost:9031/OAuthPlayground/case2A-callback.jsp
scope	openid profile
nonce	cba56666-4b12-456a-8407-3d3023fa1002

ⓘ **Note:** As the implicit flow transports the access token and ID token via the user agent (i.e. web browser), this flow requires additional security precautions to mitigate any token modification / substitution.

As for the Basic Client Profile, the client can redirect the user in different ways depending on the client and the desired user experience. For example, a web application can just issue a HTTP 302 redirect to the browser and redirect the user to the authorization URL. A native mobile application may launch the mobile browser and open the authorization URL.

ⓘ **Note:** To mitigate replay attacks, a nonce value must be included to associate a client session with an id_token. The client must generate a random value associated with the current session and pass this along with the request. This nonce value will be returned with the id_token and must be verified to be the same as the value provided in the initial request.

```
https://localhost:9031/as/authorization.oauth2?client_id=im_oic_client
 &response_type=token%20id_token
 &redirect_uri=https://localhost:9031/OAuthPlayground/case2A-callback.jsp
 &scope=openid%20profile
 &nonce=cba56666-4b12-456a-8407-3d3023fa1002
```

Again, like the Basic Client Profile, the user will then be sent through the authentication process (i.e. prompted for their username/password at their IDP, authenticated via Kerberos or x509 certificate etc). Once the user authentication (and optional consent approval) is complete, the tokens will be returned as a fragment parameter to the redirect_uri specified in the authorization request.

```
GET https://localhost:9031/OAuthPlayground/Case2A-
callback.jsp#token_type=Bearer
 &expires_in=7199
 &id_token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0.eyJzdWIiOiJuZnlmZSIsImF1ZCI6Iml

 tX29pY19jbGllbnQiLCJqdGkiOiJUOU4xUklkRkVzUE45enU3ZWw2eng2IiwiaXNzIjoiaHR0cHM6XC9c

 L3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczNzA3MSwiZXhwIjoxMzkzNzM3MzcxLCJub

 25jZSI6ImNiYTU2NjY2LTRiMTItNDU2YS04NDA3LTNkMzAyM2ZhMTAwMiIsImF0X2hhc2giOiJrdHFvZV

 Bhc2praVY5b2Z0X3o5NnJBIn0.g1Jc9DohWFfFG3ppWfvW16ib6YBaONC5VMs8J61i5j5QLieY-
mBEeVi
  1D3vr5IFWCfivY4hZcHtoJHgZk1qCumkAMDymsLGX-
IGA7yFU8LOjUdR4IlCPlZxZ_vhqr_0gQ9pCFKDk

 iOv1LVv5x3YgAdhHhpZhxK6rWxojg2RddzvZ9Xi5u2V1UZ0jukwyG2d4PRzDn7WoRNDGwYOEt4qY7lv_N
```

```
  O2TY2eAklP-xYBWu0b9FBElapnstqbZgAXdndNs-Wqp4gyQG5D0owLzxPErR9MnpQfgNcai-
PlWI_Urvo
  opKNbX0ai2zfkuQ-qh6Xn8zgkiaYDHzq4gzwRfwazaqA
 &access_token=b5bU8whkHeD6k9KQK7X6lMJrdVtV HTTP/1.1
```

ⓘ **Note:** An error condition from the authentication / authorization process will be returned to this callback URI with "error" and "error_description" parameters.

The application now has multiple tokens to use for authentication and authorization decisions:

OAuth 2.0 access_token	b5bU8whkHeD6k9KQK7X6lMJrdVtV
OpenID Connect id_token	eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0.eyJzdWIiOi JuZnlmZSIsImF1ZCI6ImltX29pY19jbGllbnQiLCJqdGkiOiJU OU4xUklkRkVzUE45enU3ZWw2eng2IiwiaXNzIjoiaHR0cHM6XC 9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5Mzcz NzA3MSwiZXhwIjoxMzkzNzM3MzcxLCJub25jZSI6ImNiYTU2Nj Y2LTRiMTItNDU2YS04NDA3LTNkMzAyM2ZhMTAwMiIsImF0X2h c2giOiJrdHFvZVBhc2praVY5b2Z0X3o5NnJBIn0.g1Jc9DohWF fFG3ppWfvW16ib6YBaONC5VMs8J61i5j5QLieY- mBEeVi1D3vr 5IFWCfivY4hZcHtoJHgZk1qCumkAMDymsLGX- IGA7yFU8LOjUd R4IlCPlZxZ_vhqr_0gQ9pCFKDkiOv1LVv5x3YgAdhHhpZhxK6r Wxojg2RddzvZ9Xi5u2V1UZ0jukwyG2d4PRzDn7WoRNDGwYOEt qY7lv_NO2TY2eAklP- xYBWu0b9FBElapnstqbZgAXdndNs-Wqp 4gyQG5D0owLzxPErR9MnpQfgNcai- PlWI_UrvoopKNbX0ai2zf kuQ- qh6Xn8zgkiaYDHzq4gzwRfwazaqA

ⓘ **Note:** Because the implicit grant involves these tokens being transmitted via the user agent, these tokens cannot be kept confidential; therefore a refresh_token cannot be issued using this flow.

Step 2: Validate the ID Token

The next step is to parse the id_token, and validate the contents. Note, that as the id_token was received via the user agent, rather than directly from the token endpoint, the verification of the digital signature

is required to detect any tampering with the id_token. Firstly, decode both the header and payload components of the JWT:

Component	Value	Value Decoded

Header	eyJhbGciOiJSUzI1NiIsImtpZCI6IjRv	<pre>{
		  "alg":"RS256",
		  "kid":"4oiu8"
		}</pre>

Payload	eyJzdWIiOiJuZnlmZSIsImF1ZCI6Im
	jbGllbnQiLCJqdGkiOiJUOU4xUklkR
	U3ZWw2eng2IiwiaXNzIjoiaHR0cHM
	y5tZXljbG91ZC5uZXQ6OTAzMSIsI
	MzczNzA3MSwiZXhwIjoxMzkzNzM
	jZSI6ImNiYTU2NjY2LTRiMTItNDU2
	NkMzAyM2ZhMTAwMiIsImF0X2hh
	VBhc2praVY5b2Z0X3o5NnJBIn0
		<pre>{
		  "sub":"nfyfe",
		  "aud":"im_oic_client",
		  "jti":"T9N1RIdFEsPN9zu7el6zx6",
		  "iss":"https:\/\/
		localhost:9031",
		  "iat":1393737071,
		  "exp":1393737371,
		  "nonce":"cba56666-4b12-456a-8407-
		  "at_hash":"ktqoePasjkiV9oft_z96rA
		}</pre>

Now we follow the guidelines in the OpenID Connect specifications (Core specification section 3.1.3.7 also taking into consideration section 3.2.2.11) for ID Token Validation:

Step #	Test Summary	Result
1	Decrypt the token (if encrypted)	Token not encrypted, skip test
2	Verify the issuer claim (iss) matches the OP issuer value	Valid
3	Verify the audience claim (aud) contains the OAuth2 client_id	Valid
4	If the token contain multiple audiences, then verify that an Authorized Party claim (azp) is present	Only one audience, skip test
5	If the azp claim is present, verify it matches the OAuth2 client_id	Not present, skip test
6,7,8	Optionally verify the digital signature (required for implicit client profile) (see section 4.4)	Verify signature as per "ID Token" section
9	Verify the current time is prior to the expiry claim (exp) time value	Valid
10	Client specific: Verify the token was issued within an acceptable timeframe (iat)	Valid

Step #	Test Summary	Result
11	If the nonce claim (nonce) is present, verify that it matches the nonce passed in the authentication request	Nonce matches, Valid
12	Client specific: Verify the Authn Context Reference claim (acr) value is appropriate	No acr value present, skip test
13	Client specific: If the authentication time claim (auth_time) present, verify it is within an acceptable range	No auth_time present, skip test
14	If the implicit client profile is used, verify that the access token hash claim (at_hash) matches the hash of the associated access_token	Validate at_hash as per "ID_Token" section

The results of the ID token validation are sufficient to trust the id_token and the user can be considered "authenticated".

Step 3: Retrieve the User Profile

We now have an authenticated user, the next step is to request the user profile attributes so that we can personalize their app experience and render the appropriate content to the user. This is achieved by requesting the contents of the UserInfo endpoint.

Accessing the UserInfo endpoint requires that we use the access token issued along with the authorization request. As the implicit flow transports the access token using the user agent, there is the threat of tokens being substituted during the authorization process. Before using the access token, the client should validate the at_hash value in the id_token to ensure the received access token was issued alongside the id_token.

To validate the at_hash value, see section 4.5. Once the at_hash is verified, the client can then use the access token to request the user profile:

```
GET https://localhost:9031/idp/userinfo.openid HTTP/1.1

Authorization: Bearer b5bU8whkHeD6k9KQK7X6lMJrdVtV
```

The response from the UserInfo endpoint will be a JSON structure with the requested OpenID Connect profile claims:

```
{
 "sub":"nfyfe",
 "family_name":"Fyfe",
 "given_name":"Nathan",
 "nickname":"Nat",
 ...[additional claims]...
}
```

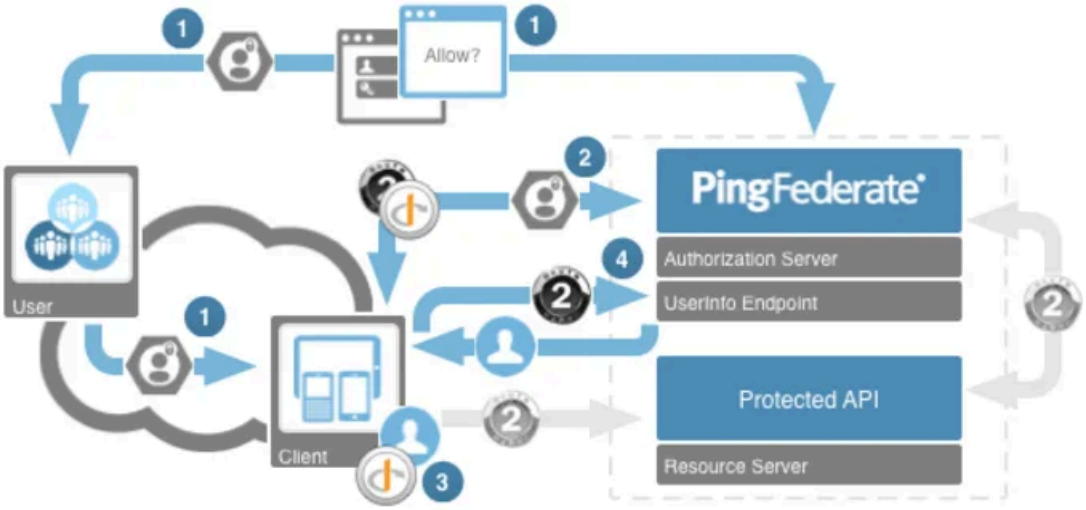
Before we can be confident the response to the UserInfo reflects the authenticated user, we must also check that the subject ("sub" claim) returned from the UserInfo endpoint matches the authenticated user

we received in the id_token. In this case, the "sub" claim in both the UserInfo response and the id_token match so we can use the values in the UserInfo response for our application needs.

Basic Client Profile

The OpenID Connect 1.0 Basic Client Profile uses the OAuth 2.0 "Authorization Code" grant type. You will notice the flow is almost identical to the OAuth 2.0 authorization code flow with the exception of the "openid" scope and the tokens returned.

This section walks through an example authentication using the OpenID Connect Basic Client Profile. This will step through requesting the authentication of a user, receiving and validating the OpenID Connect id_token (step 1 through 3 below) and then query the UserInfo endpoint to retrieve profile information about the user (step 4).



This example assumes PingFederate 7.3 or higher is installed with the OAuth 2.0 Playground developer tool. The following configuration will be used:

PingFederate server base URL	https://sso.pingdeveloper.com
OAuth 2.0 client_id	ac_oic_client
OAuth 2.0 client_secret	abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910$RQPOnmlijh
Application callback URI	https://sso.pingdeveloper.com/OAuthPlayground/case1A-callback.jsp

ⓘ **Note:**

- For native mobile applications, the callback URI may be a non-http URI. This is configured in your application settings and will cause the mobile application to be launched to process the callback.
- Also with mobile applications, the client secret is guaranteed to be secret and therefore can be omitted. The Proof Key for Code Exchange (PKCE) specification is used to mitigate this scenario.

Step 1: Authenticate the End-User and Receive Code

The initial user authentication request follows the OAuth2 Authorization Grant Type flow. To initiate the OpenID Connect process, the user will be redirected to the OAuth2 authorization endpoint with the "openid

profile" scope value. Additional scope values can be included to return specific profile scopes. The request is made to the authorization endpoint with the following parameters:

client_id	ac_oic_client
response_type	code
redirect_uri	https://sso.pingdeveloper.com/OAuthPlayground/case1A-callback.jsp
scope	openid profile

The client will then form the authorization URL and redirect the user to this URL via their user agent (i.e. browser). This can be performed in different ways depending on the client and the desired user experience. For example, a web application can just issue a HTTP 302 redirect to the browser and redirect the user to the authorization URL. A native mobile application may launch the mobile browser and open the authorization URL. The authorization URL using the values above would be:

```
https://sso.pingdeveloper.com/as/authorization.oauth
 ?client_id=ac_oic_client
 &response_type=code
 &redirect_uri=https://sso.pingdeveloper.com/OAuthPlayground/case1A-
callback.jsp
 &scope=openid%20profile
```

For mobile application scenarios where it is not guaranteed that the app at the end of the redirect_uri is the intended application, the Proof Key for Code Exchange (PKCE) specification should be used to mitigate tokens being issued to an incorrect client. The "plain" variant of PKCE involves including a code_challenge parameter at this stage to link this authorization request with the subsequent token request (step 2 below). Therefore an example of a mobile authorization request (using com.pingidentity.developer.oauthplayground://oidc_callback as the redirect_uri) will be:

```
https://sso.pingdeveloper.com/as/authorization.oauth2
 ?client_id=ac_oic_client
 &response_type=code
 &redirect_uri=com.pingidentity.developer.oauthplayground://oidc_callback
 &scope=openid%20profile
 &code_challenge=abcd-this-is-a-unique-per-request-value
```

The user will then be sent through the authentication process (i.e. prompted for their username/password at their IDP, authenticated via Kerberos or x509 certificate etc). Once the user authentication (and optional consent approval) is complete, the authorization code will be returned as a query string parameter to the redirect_uri specified in the authorization request.

```
GET https://sso.pingdeveloper.com/OAuthPlayground/Case1A callback.jsp?
code=ABCâ€¦XYZ HTTP/1.1
```

(or for a mobile application, this URL will be handled in according to the mobile OS - for example in iOS in the AppDelegate class using the application:handleOpenUrl:function)

ⓘ **Note:** An error condition from the authentication / authorization process will be returned to this callback URI with "error" and "error_description" parameters.

Step 2: Exchange the Authorization Code for the Tokens

Following the Authorization Code grant type defined in the OAuth 2.0 protocol, the application will then swap this authorization code at the token endpoint for the OAuth2 token(s) and the OpenID Connect ID Token as follows:

```
POST https://sso.pingdeveloper.com/as/token.oauth2 HTTP/1.1

Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
client_id=ac_oic_client&
client_secret=abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910SRQPOnmlijhoauthplaygroundapplic
code=ABC...XYZ&
redirect_uri=https://sso.pingdeveloper.com/OAuthPlayground/case1A-
callback.jsp
```

ⓘ **Note:** As the redirect_uri was specified in the original authorization request. It is required to be sent in the token request.

In the mobile scenario, as we are using PKCE to prove to the Authorization Server that we are the same application that initiated the authorization request, we also need to include the PKCE code_verifier parameter and use our application's redirect_uri:

```
POST https://sso.pingdeveloper.com/as/token.oauth2 HTTP/1.1

Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
client_id=ac_oic_client&
client_secret=abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910SRQPOnmlijhoauthplaygroundapplic
code=ABC...XYZ&
redirect_uri=com.pingidentity.developer.oauthplayground://oidc_callback&
code_verifier=abcd-this-is-a-unique-per-request-value
```

ⓘ **Note:** An OAuth client used for mobile authentication is not likely to have a client_secret. In this scenario, the client_secret parameter int he request can be omitted.

The token endpoint will respond with a JSON structure containing the OAuth2 access token, refresh token (if enabled in the OAuth client configuration) and the OpenID Connect ID token:

```
HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

{
 "token_type":"Bearer",
 "expires_in":7199,
 "refresh_token":"BBB...YYY",
 "id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0.eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImFjX2
 JIR1Awdnlxbmgwo VBjQ3MzenBHbUVsIiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIs
 ZXhwIjoxMzkzNzMwNjgwfQ.EQeAm84Xj2lekxUMSK9H3BvoCl511JV1TWHCyQQ7vTnXcuvZYdBHE9_OpIr9gD5
```

```
 EueeY8bUgkTfIBKzUUJETSeaO1U8uH9Td0QYv7q3rRfurLhrpzubFbAIfjPOiv8jxgBjMyGEdPJ7aXtBwP_cr2F
  0T-
xKnwZcocDZs_rYAOHFljLPgO2tX8BBePJfqUUUG46U1K4hSqo7LP3zru4BDE2wNbZyOhb2keeLjetNq2ES33YthN
Ji7kYn
  Maij3ta1OyLSB_HB-NbhQCKvjm4GT9ocm0w",
 "access_token":"AAA...ZZZ"
}
```

The application now has multiple tokens to use for authentication and authorization decisions:

OAuth 2.0 access_token	AAA...ZZZ
rOAuth 2.0 refresh_token	BBB...YYY
OpenID Connect id_token	eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0.eyJzdWIiOiJuZnlmZ llbnQiLCJqdGkiOiJIR1AwdnlxbmgwOVBjQ3MzenBHbUVsIiwiaXN bG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczMDM4MCwiZXhw MSK9H3BvoCl511JV1TWHCyQQ7vTnXcuvZYdBHE9_OpIr9gD5 TfIBKzUUJETSeaO1U8uH9Td0QYv7q3rRfurLhrpzubFbAIfjPOiv8j _iBRA4cD8c4PwEOROr0T- xKnwZcocDZs_rYAOHFljLPgO2tX8BBePJfqUUUG46U1K4hSqo7 wNbZyOhb2keeLjetNq2ES33YthNU9dkmHUgbtoD- Ji7kYnMaij3ta1OyLSB_HB- NbhQCKvjm4GT9ocm0w

Step 3: Validate the ID Token

The next step is to parse the id_token, and validate the contents. Note, that as the id_token was received via a direct call to the token endpoint, the verification of the digital signature is optional.

Firstly, decode both the header and payload components of the JWT:

Component	Value	Value Decoded
Header	eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0	{ "alg":"RS256", "kid":"4oiu8" }
Payload	eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImFjX29pY19f jbGllbnQiLCJqdGkiOiJIR1AwdnlxbmgwOVBjQ3 MzenBHbUVsIiwiaXNzIjoiaHR0cHM6Ly y5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MT MzczMDM4MCwiZXhwIjoxMzkzNzMwNjgwfQ	{ "sub":"nfyle", "aud":"ac_client", "jti":"IR1AwdnlxbmgwOVBjQ3 "iss":"https://localhost:9031", "iat":1393730380, "exp":1393730680 }

Now we follow the guidelines in the OpenID Connect specifications (Core specification section 3.1.3.7) for ID Token Validation (see 4.3 for details on validating the id_token)

Step #	Test Summary	Result
1	Decrypt the token (if encrypted)	Token not encrypted, skip test
2	Verify the issuer claim (iss) matches the OP issuer value	Valid
3	Verify the audience claim (aud) contains the OAuth2 client_id	Valid

Step #	Test Summary	Result
4	If the token contain multiple audiences, then verify that an Authorized Party claim (azp) is present	Only one audience, skip test
5	If the azp claim is present, verify it matches the OAuth2 client_id	Not present, skip test
6,7,8	Optionally verify the digital signature (required for implicit client profile) (see section 4.4)	TLS security sufficient, skip test
9	Verify the current time is prior to the expiry claim (exp) time value	Valid
10	Client specific: Verify the token was issued within an acceptable timeframe (iat)	Valid
11	If the nonce claim (nonce) is present, verify that it matches the nonce passed in the authentication request	Nonce was not sent in initial request, skip test
12	Client specific: Verify the Authn Context Reference claim (acr) value is appropriate	No acr value present, skip test
13	Client specific: If the authentication time claim (auth_time) present, verify it is within an acceptable range	No auth_time present, skip test
14	If the implicit client profile is used, verify that the access token hash claim (at_hash) matches the hash of the associated access_token	Not an implicit profile, skip test

The results of the ID token validation are sufficient to trust the id_token and the user can be considered "authenticated".

Step 4: Retrieve the User Profile

We now have an authenticated user, the next step is to request the user profile attributes so that we can personalize their application experience and render the appropriate content to the user. This is achieved by requesting the contents of the UserInfo endpoint:

```
GET https://sso.pingdeveloper.com/idp/userinfo.openid HTTP/1.1

Authorization: Bearer AAA...ZZZ
```

The response from the UserInfo endpoint will be a JSON structure with the requested OpenID Connect profile claims:

```
HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8
```

```
{
 "sub":"nfyfe",
 "family_name":"Fyfe",
 "given_name":"Nathan",
 "nickname":"Nat",
 ...[additional claims]...
}
```

Before we can be confident the response to the UserInfo reflects the authenticated user, we must also check that the subject ("sub" claim) returned from the UserInfo endpoint matches the authenticated user we received in the id_token.

In this case, the "sub" claim in both the UserInfo response and the id_token match so we can use the values in the UserInfo response for our application needs.