Developer Resources

June 4, 2025

DEVELOPER RESOURCES

Table of Contents

Application Integration Guide

This guide will walk the developer through the steps to enable standards-based Single Sign-On (SSO) to an application by integrating the application with the Ping platform.

Using the concept of "Integrate, Federate and Operate" we can describe the steps an application owner must complete to provide and manage federated SSO in their application:

- Integrate involves making the application federation-aware so that it can accept a federated security token and use the information in that token to authorize a user and create an application session.
- Federate consists of creating federated connections with partners (primarily exchanging metadata).
- Operate describes the management of these connections, adding new customers, managing connections at scale.

This guide will focus on the integrate step.

Planning

At a high level, there are four items that we will focus on in this guide for the integration stage:

- Authentication - How a user is authenticated and their identity validated
- User Profile - How the user's identity attributes are provided
- Authorization & Access Control - How an application can enforce authorization decisions based on the security token
- Session Management - How to start, end, revoke and refresh a users session

There are a number of additional considerations that should also be taken into account (for example federated user provisioning) however, this guide will focus on the SSO activity.

Application Considerations

Knowing that to provide SSO to my customers, employees and partners I need to integrate my application into the federation infrastructure; the first question that is asked is "How do I integrate my application?". There are a number of mechanisms that can be used for this "last-mile" integrate, so to plan for application integration a number of components should be considered:

Application Format

The format of the application. Whether it is a web application, a mobile application or an API or web service.

Application Platform

The application platform can help determine the simplest integration method. The platform includes the language and framework the application is written in (i.e. Java / Spring or .NET 4.5) as well as the web application server the application is hosted on (i.e. Apache or IIS).

Application Deployment Model

Whether the application is cloud-hosted or deployed inside the firewall can also help determine the simplest integration method. For example PingOne provides a simple REST interface to enable SSO into a cloud-hosted application whereas PingFederate is a a software component but enables more options for complex integrations.

API Requirements

If the application needs to talk to specific API's it may be simpler to define the authentication mechanism around those services (i.e. if the application requires user authentication and accesses OAuth 2.0 protected REST web services, then OpenID Connect protocol is a good choice).

Availability of Source Code

There may be applications where the source code is not available (i.e. COTS application) or that the source code is not supported. Perhaps a mechanism that doesn't require code changes is the most appropriate.

Selecting an Integration Method

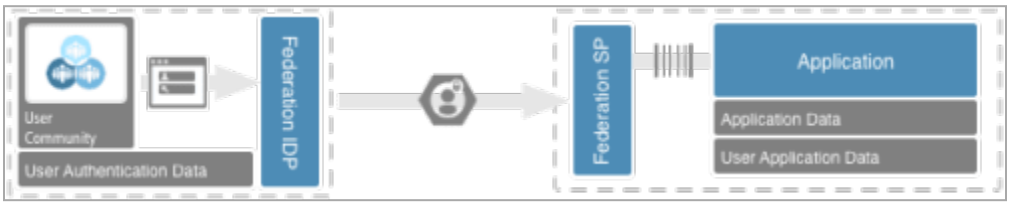
With the information gathered in the previous stage, we can narrow down our integration options to determine if we need to make code changes to support the integration. The next sections will detail the specifics for integrating the different application formats, however the following general steps should be considered for all application formats before opening up the code in the application:

- Leverage SSO support provided by infrastructure or framework - The goal is to leverage an open standard to provide maximum security and interoperability between partners, vendors and customers. Once the application can speak an open standard, then managing connections can be made via configuration rather than code. For example, if a web application can accept a SAML assertion to sign a user in, then using PingOne or PingFederate multiple partners (i.e. enterprises via SAML, social networks via OpenID Connect etc) can be connected and presented to the application through that SAML connection.
- Leverage an access gateway (i.e. PingAccess) to handle the authentication - PingAccess can be used to protect web applications and APIs by handling the protocol work and presenting the user attributes to the application or API via HTTP headers or cookies. If the application or API can be protected behind an access gateway like PingAccess the work of integrating an application can be greatly simplified and the benefits of administration of authentication and authorization policies can be easily implemented.
- Leverage the federation engine (i.e. PingFederate) to handle token translation - PingFederate is highly skilled at exchanging tokens in a standard and interoperable manner. Scenarios such as protocol transition (i.e. providing SAML SSO to an application that is configured to use the WS-Federation protocol) can easily be achieved. Rather than re-writing the authentication logic, the PingFederate protocol engine can perform all the work. The same is true for applications that have API requirements. Although a cleaner solution maybe be to use OpenID Connect and OAuth 2.0 for a web application that calls OAuth protected API's, a simpler and quicker solution may be to retain a SAML authentication model and provide an OAuth access token as an attribute of the assertion or exchange the assertion for an OAuth access token.
- Finally, integration via code. Either implementing open standards support in the application directly or using an integration kit to simplify the integration of standards into the application.

Integrating Web Applications

Our first scenario involves analyzing a web application to determine the changes required to make the application behave in a federated model. We will first identify what a typical web application looks like and how it will look after it has been integrated. Then we will use the information we gained in the previous section and walk through the process needed to integrate a web application.

A user via their web browser access the application, when they visit a protected page the application wil prompt them for authentication. In a non-federated model this is typically an application login form where the user enters their username and password. The application will then validate these credentials against the user authentication store (an internal data store or maybe a networked directory). If the credentials are valid and the authenticated user is authorized to access the protected content, the user is provided an authenticated session and is allowed to continue.



The goal of application integration for a web application is to move the authentication flow to a federated browser SSO model. As we know with the browser SSO model, the authentication event is offloaded to the identity provider and replaced with a mechanism to validate the authentication token returned via the authentication process. When a user accesses protected content, the application will redirect the users browser to the federated sign-in process. The user authenticates at the identity provider, a security token is issued and provided to the service provider who creates a session in the application.

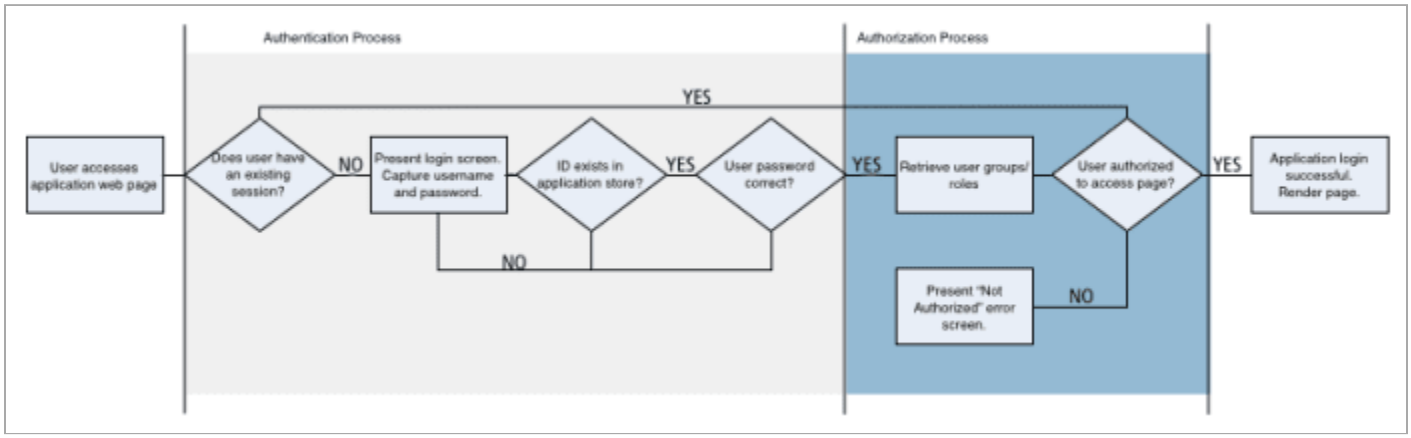
As we learned in the previous section, there are four items we will evaluate:

- Authentication Event(s)
- User Profile
- Authorization Event(s)
- Session Management

Authentication and Single Sign-On

The sign-on process may contain more than one location where a visitor can authenticate; the "front door", at an approval stage, if there are any APIs exposed by the application etc. It is important to identify the events in the application that require authentication and what authentication requirements each event has (ie an approval process may require a step-up to a stronger form of authentication such as a second factor).

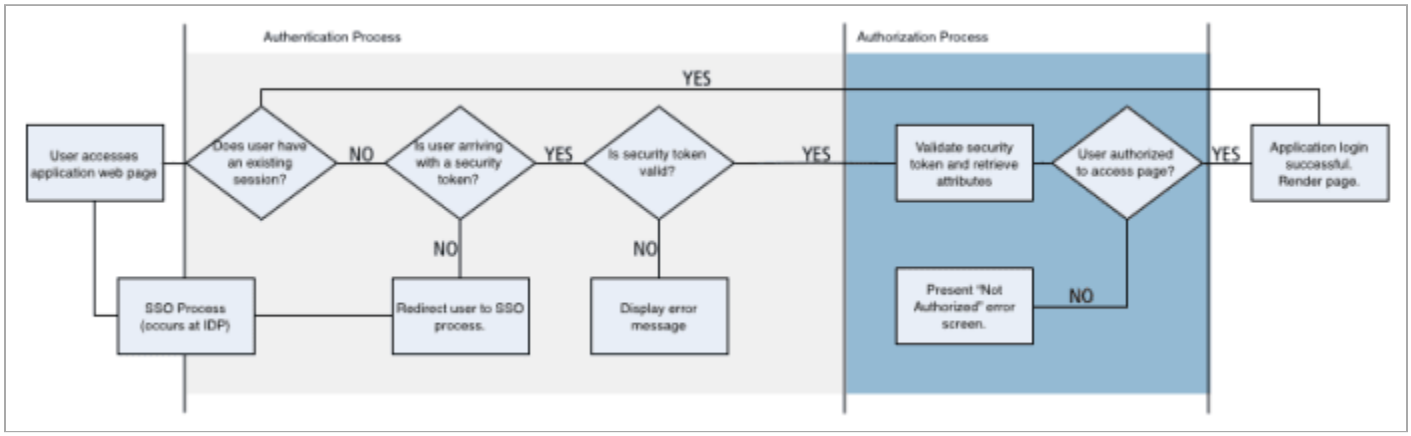
Focusing on the "front-door" authentication, a general web application sign-on process will go through a simple workflow:



To change this workflow to support a federated authentication model, when an unauthenticated user appears the application will redirect the users browser to the SSO process. This will result in the user being challenged for authentication at the identity provider and (after a successful authentication) a token returned to the application sign-on process.

The sign-in process is now expecting three user states:

1. A known user already has an application session (if the user is authorized, the requested page will be rendered)
2. An unknown user is arriving with a federated security token (where the application will validate that token and if valid, create an application session - the user will then be in state #1)
3. An unknown user is arriving without a token (the application will redirect the user to the federated sign-in process - the user will re-appear to the application as state #2)



The results of the authentication request are a token containing proof of authentication and the additional attributes required by the application. The application can use these attributes to create the application session and personalize the application for the user.

User Profile

More than likely, an application requires more information about the user during authentication (i.e. a list of groups to use for authorization decisions or the user's preferred language to personalize the application) these additional attributes can be provided by the authentication provider during the authentication event or through an out-of-band process (i.e. federated provisioning (SCIM) or via an API call like the OpenID Connect UserInfo API).

An application may have an application data store that contains application-specific information about the incoming user. For some applications the only information the application needs from the authentication provider is the username. It can then map that user to their application profile to associate the user with their application profile and authorizations.

Authorization and Access Control

Once we have an authenticated user (and their user profile), the next event is the authorization and access control stage. The application can use the attributes defined in the user profile (whether that information was received during the authentication or was provided in an alternate method) to make authorization decisions and determine theÂ appropriate level of access to the user.

Session Management

The authentication process is generally a one-time event. So when the application receives the authentication token and signs that user in, the application will generally create an application session. Once that session expires it will send the user through the sign-in process again to reauthenticate. This process may be aligned with the authentication provider to provide a seamless session extension - if the application session is shorter than the authentication provider session, then the user will automatically be logged back into the application when the session expires. Products such as PingAccess provide session management out of the box.

Terminating the session can be achieved via single log out or by just killing the application session and redirecting the user / instructing them to close their browser. Single log-out is supported across federation protocols however can be trickly to implement due to differing authentication methods provided by authentication providers. It is best to provide options to the federation partner on how to handle the sign-out event. Generally asking the partner to supply a log-out URL during configuration would be sufficient, then the application can log the user out of the application, then redirect to the URL specified by the partner.

Selecting an Integration Method for Web Apps

The following question can help narrow down the integration options available for web applications. You can learn more about the integration methods in the Explore section of this website.

This flowchart assumes that:

- the application will be integrated directly with the federation infrastructure instead of using an access gateway like PingAccess
- the source code for the web application is available (if not, PingAccess may be able to support this scenario)

Integration Option	Scenario
Integrate directly with application / platform	Application supports a federated security protocol such as SAML or WS-Federation (for example Microsoft SharePoint or Microsoft .NET) PingFederate and PingOne can be used to manage federation connection to partners.
PingFederate Web Server Integration Kits	If the application server is IIS or Apache.
PingFederate Language Integration Kits (Java, .NET, PHP)	Application is either Java, .NET or PHP. Especially if there is no direct communication between PingFederate and the application.
PingFederate Agentless Integration Kit	Any code-based language that can make HTTP calls. *Note: This kit requires direct connectivity between the application and PingFederate.*
PingOne Application Provider Services	Application is cloud-based or there are requirements that no software or libraries are to be used.

Integration Option	Scenario
Native protocol support	Application requires authentication and API authorization. OpenID Connect can be used to authenticate a user to an application client as well as provide an OAuth 2.0 access token use to secure API requests. *Note: PingFederate can perform token translation allowing one token to be exchanged for another (i.e. a SAML assertion to be exchanged for an OAuth access token)*

> **Note**
>
> In all cases, the Agentless Integration Kit and PingOne APS can be used to integrate an application with the Ping infrastructure quickly and simply.

Integrating Mobile Applications

Mobile apps are an increasingly common format for application developers. A mobile app is generally a simple, concise rendering of an application with all the functionality packed into a small easy-to-use application available on a users phone or tablet.

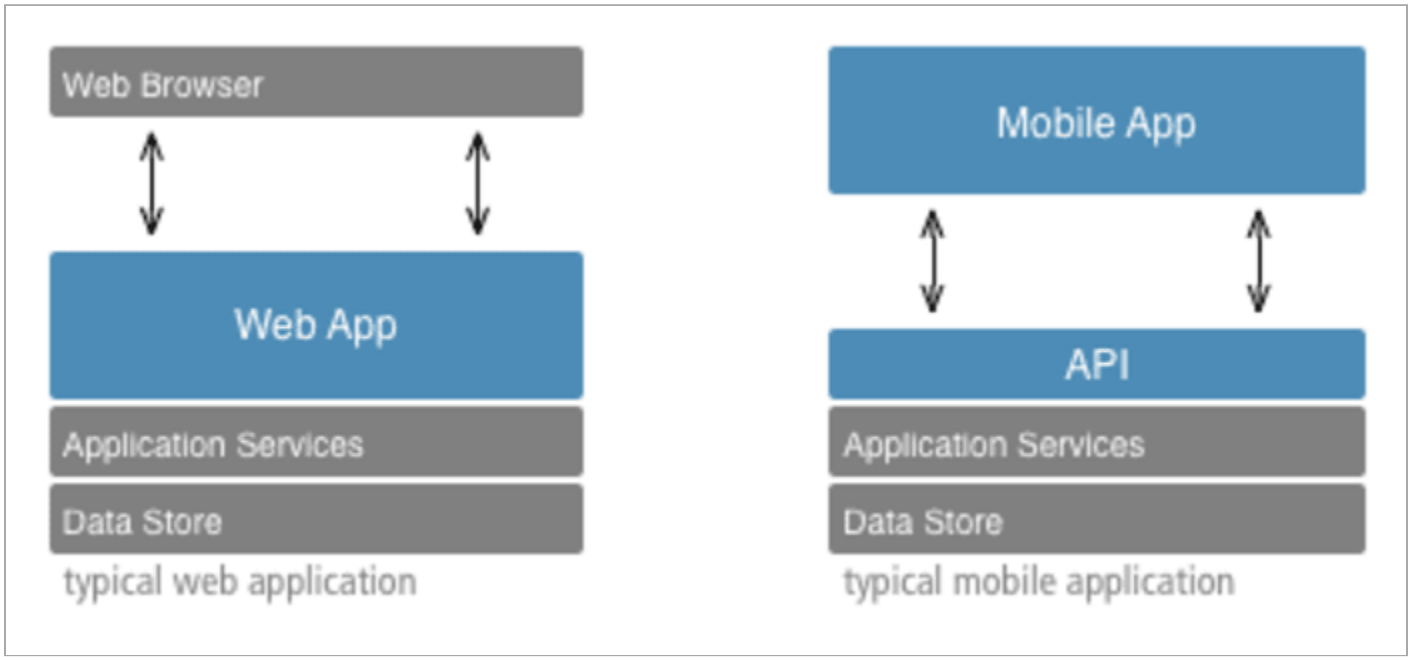
This section will identify a typical mobile application architecture and walk through integrating a mobile app with the federation infrastructure.

Anatomy of a Mobile Application

The mobile app architecture generally differs from a traditional web app architecture in that the mobile app is more distributed. Where a web app is more monolithic in design with complexity suiting a larger screen format with a user present, a web app is generally a more distributed model with a simple client that performs actions against an API on behalf of a user - whether the user may or may not be present.

A mobile application requires a new model of managing authentication and security due to the app architecture and fundamental differences between it and a traditional web app:

- Traditional web app is monolithic, components can talk to all other components. A mobile app consists of a client communicating to APIs.
- Web application user is generally always present. Mobile app user may be present or may allow the app to function in the background.
- Web application allows for a more complex UX due to screen size and device. Mobile app generally simplified UX targeted to a touch device.

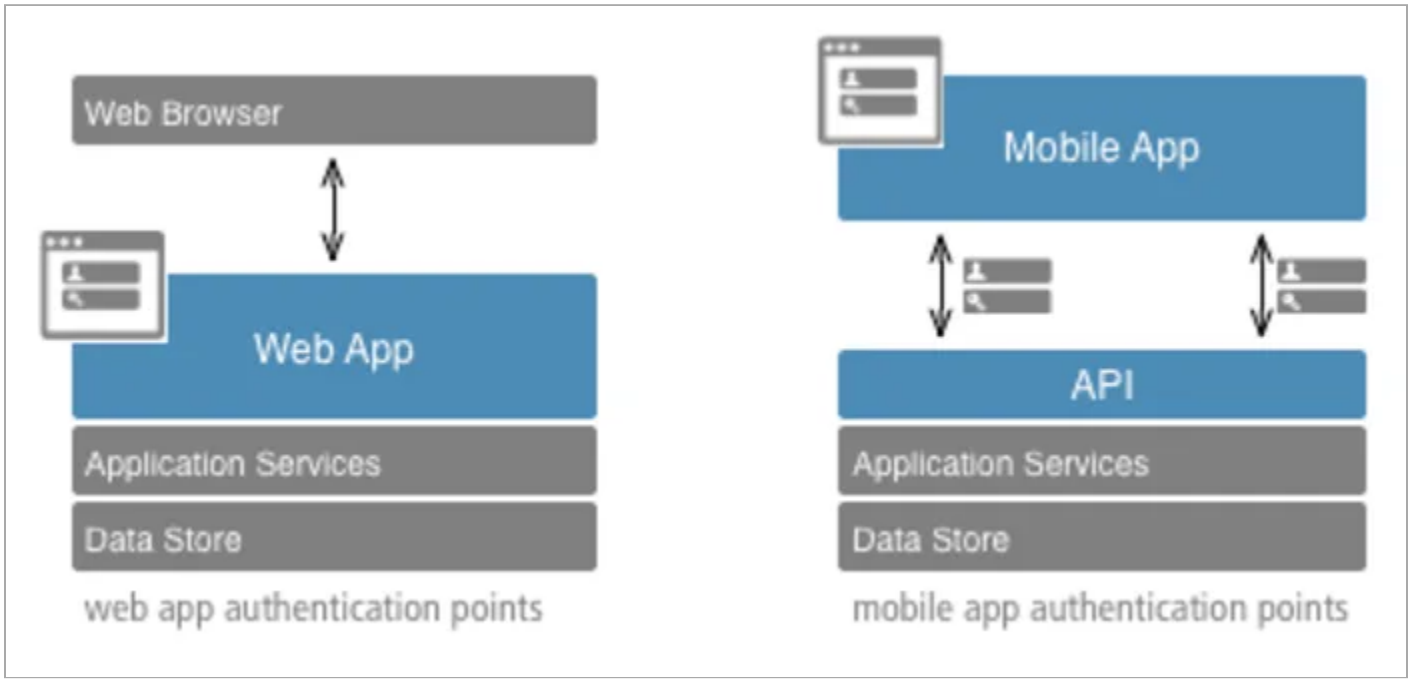


Device capabilities and restrictions also factor into an app design, specifically around security:

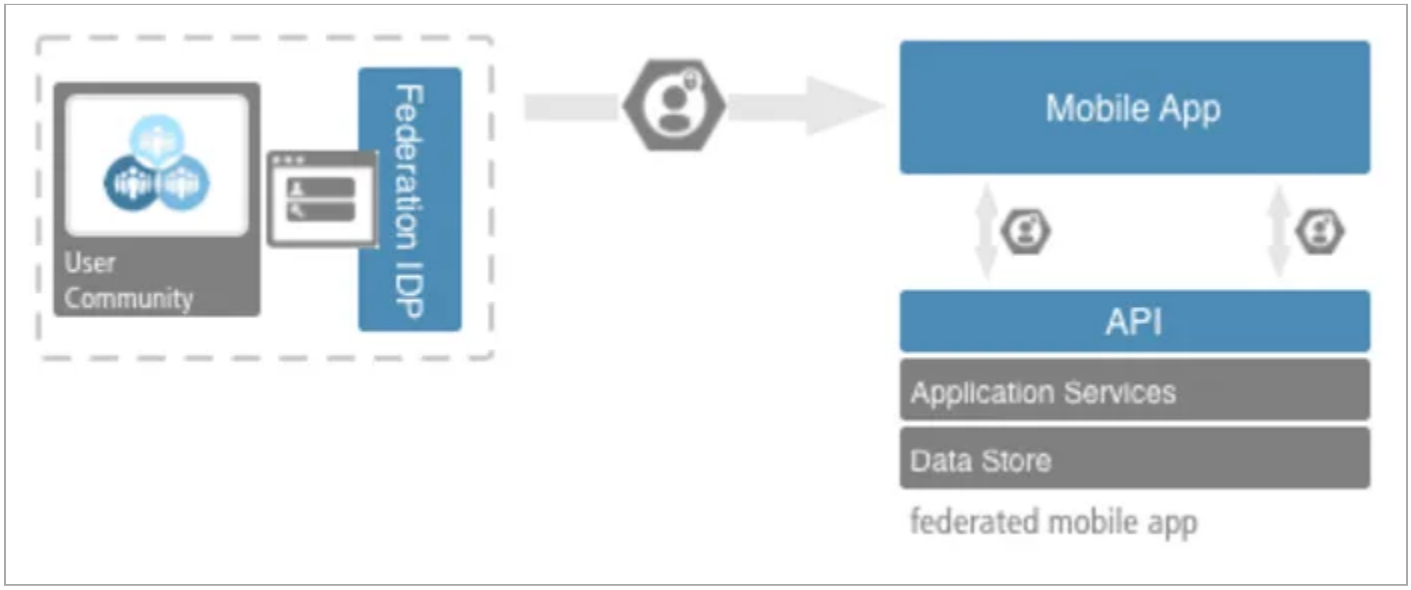
- A mobile device is smaller, therefore easier to lose or be stolen so sensitive data (including stored passwords) needs to be secured on the device or not stored in the first place.
- The prevalence of BYOD sees enterprise applications alongside personal applications, leading to less secure entry mechanisms to the device (ie no unlock PIN on the device).
- Ability to work in a disconnected mode (i.e. airplane mode) and attention to user experience results in cached data and credentials.
- Applications are sandboxed on devices making it difficult to interact with other applications and share credentials and sessions.
- Mobile frameworks generally have limited functionality exposed through SDKs (i.e. lack of SAML framework for mobile developers)

Although a different architecture, the same fundamental challenges remain: to eliminate passwords and reduce the authentication complexity from the app developer.

As the mobile app architecture generally involves both the app client and a resource or API that the app is communicating with, mobile app security involves securing both sides. Authentication to the app client (for user identification, personalisation, access to stored data etc) and securing of the API calls to the backend services.



Like web application SSO, eliminating passwords and replacing with a standards-based security token the application developer can offload the authentication complexity from the application to the authentication provider. The results are a flexible token model that the application can use to be assured of the identity of the user authenticating and can use to authorize access to backend APIs (without storing or transferring passwords).

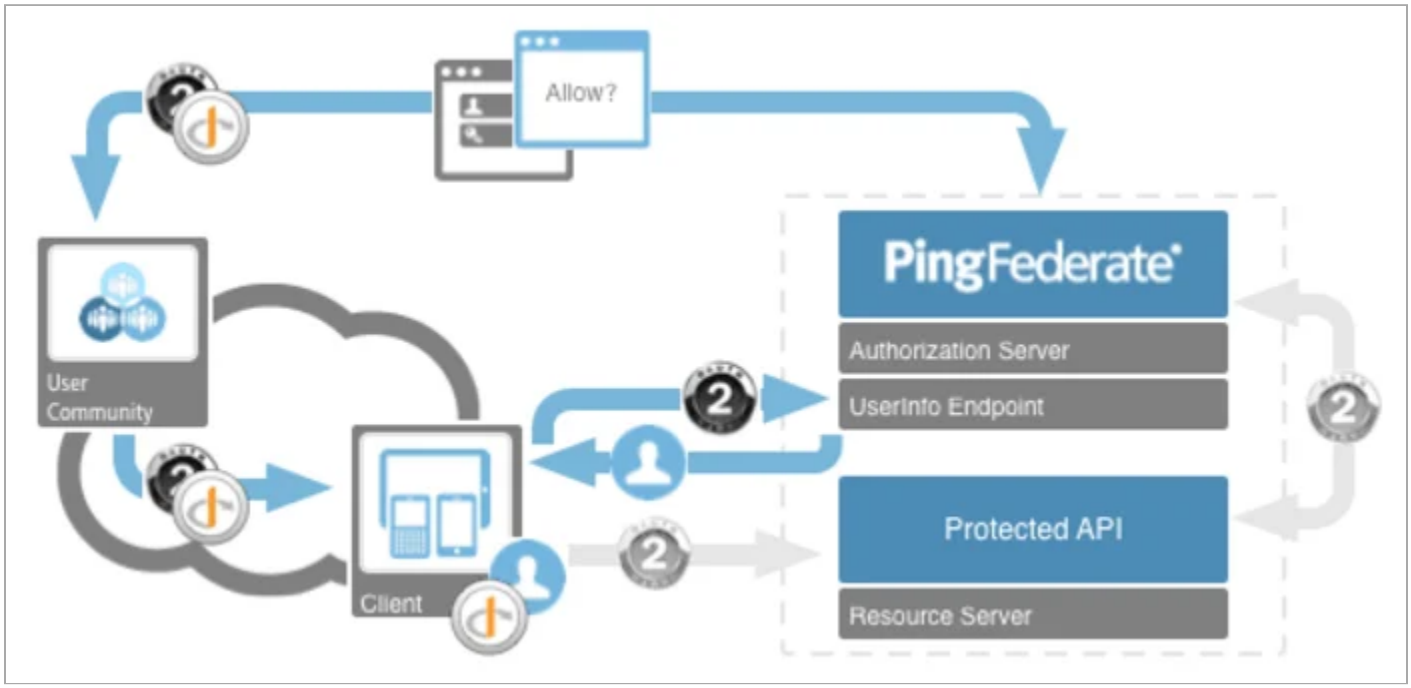


Authentication to mobile applications is also greatly affected by user experience requirements. Better application user experience may be to authenticate a user natively in the application or via a web view in the application, however would limit the authentication flexibility provided when the user is redirected to the mobile browser for authentication (additional authentication functionality such as certificate authentication and a reusable authentication provider session to facilitate SSO across applications).

The prevailing standards for securing mobile applications are OpenID Connect and OAuth 2.0. These complementary protocols allow and application to authenticate a user, retrieve profile information about that user and identify that user to APIs that it calls on the users behalf.

In the OpenID Connect / OAuth 2.0 flow, a mobile app (client) will redirect the user to the Authorization Server for authentication the result of this will be two tokens returned to the client app, an id_token containing information about the user's authentication event and an access_token which the client can use as authorization to protected APIs.

The client will then validate those tokens and, if valid, consider the user authenticated. If additional profile information is required, the client can call the UserInfo endpoint on the AS to retrieve additional information about the authenticated user.



Authentication and Single Sign-on

To leverage OpenID Connect for authentication to a mobile application, the app will need to implement a few simple items:

- An action to redirect the user through the authentication process
- A callback URI that receives the authentication tokens
- A process to validate the received tokens
- A process to retrieve the user profile information (optional)NOTE: Due to challenges with keeping a mobile application's client_secret a secret (i.e. app distribution through the app store, re-issuance of an application if the secret were to be compromised) a mobile application should use the Proof Key for Code Exchange (PKCE) extension with an OpenID Connect basic profile flow.

There are a number of OAuth 2.0 and OpenID Connect libraries and frameworks available via open source which can simplify the implementation even further.

OpenID Connect and OAuth 2.0 can provide secure single sign-on to your mobile applications and the APIs that those applications rely on. By removing passwords from the equation, apps are more secure. By reducing the authentication complexity, developers can focus on building the application rather than the authentication model.

User experience requirements may dictate that the user remain in the application and not redirected to a web flow. In this case, the OAuth 2.0 Resource Owner Password Credentials flows can be used to exchange the user's username / password for OAuth access tokens. As the flow requires a username and password for authentication, this limits the opportunity to handle federated partners.

User Profile

OpenID Connect publishes an endpoint on the authorization server called the UserInfo endpoint. An application can make a REST call to this endpoint authenticating with the OAuth 2.0 access token assigned to the user. This call will return the user's profile information.

Authorization and Access Control

Generally authorization for native mobile apps is enforced at the back-end API. After a user authenticates, the application will call a back-end API with the OAuth 2.0 access token. If the token has expired, is not valid or the authorizations provided in the access token are not sufficient for the request, then the API call will fail. The application should handle these failures gracefully.

The application itself may also enforce access control based on the attributes returned in the user profile. For example and enterprise application may only be available for user's in the finance department. If an enterprise user from marketing launches the mobile app, they may be denied access because of their user profile attributes.

Session Management

A mobile application doesn't have a session as we expect from a web application. Typically the application session is managed by the lifetime of the OAuth 2.0 access token. While the token is valid the user can make API requests and interact with the application. When the access token expires the API calls will fail and the user will need to retrieve a new token to make the API calls.

OAuth 2.0 includes a concept of refresh tokens that can be exchanged for a new access token without requiring the end user to re-authenticate. This can be useful for "offline access" scenarios where the application may want to make a call while the user is not present in the application (i.e. check an order status while the application is running in the background). If a user loses their device, then these refresh tokens should be invalidated so that the application is not available to unauthorized users.

> **Note**
>
> A refresh token is not available in every OAuth 2.0 grant type (i.e. Implicit)

Selecting an Integration Method for Mobile Apps

Mobile application integration is fairly straightforward with the OAuth 2.0 and OpenID Connect 1.0 protocols. Achieving single sign-on across applications has proven to be the challenging piece.

For more information about leveraging the mobile OS to facilitate single sign-on, refer to the Native Application SSO (NAPPS) Developers Guide.

Integrating APIs and Web Services

APIs and Web Services are the heart of applications and system development. They enable us to re-use tried and trusted code across multiple applications and application formats and providing access for partners into internal systems.

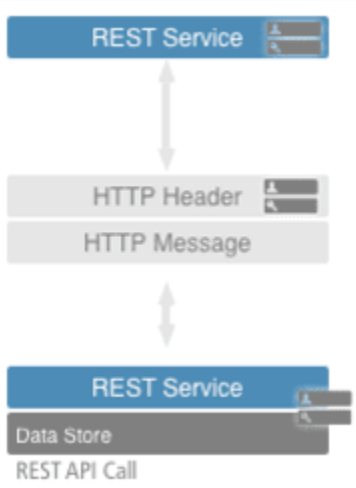
APIs and web services are now a common method of accessing and exposing an application's functionality and therefore a critical interface to secure.

Rest APIs

REST-based services use HTTP verbs and JSON to communicate actions. As an example, an API may represent a "product". The following REST API calls may be performed:

- GET https://api.company.com/product - get all products
- GET https://api.company.com/product/{product_id} - get a specific product
- POST https://api.company.com/product - create a new product

Because they use the HTTP protocol, authentication is usually performed via HTTP headers using the authorization header. The most common protocol used to authorize access to REST APIs is the OAuth 2.0 protocol.



SOAP Web Services

SOAP-based services are XML based and come with a standard security mechanism (WS-Security protocol). This allows for a security element to be presented as part of a SOAP web services call. There are multiple profiles that define these standards (i.e. the username profile which uses a username and password security token or the x509 profile that uses a certificate as a security token) as an authentication token.

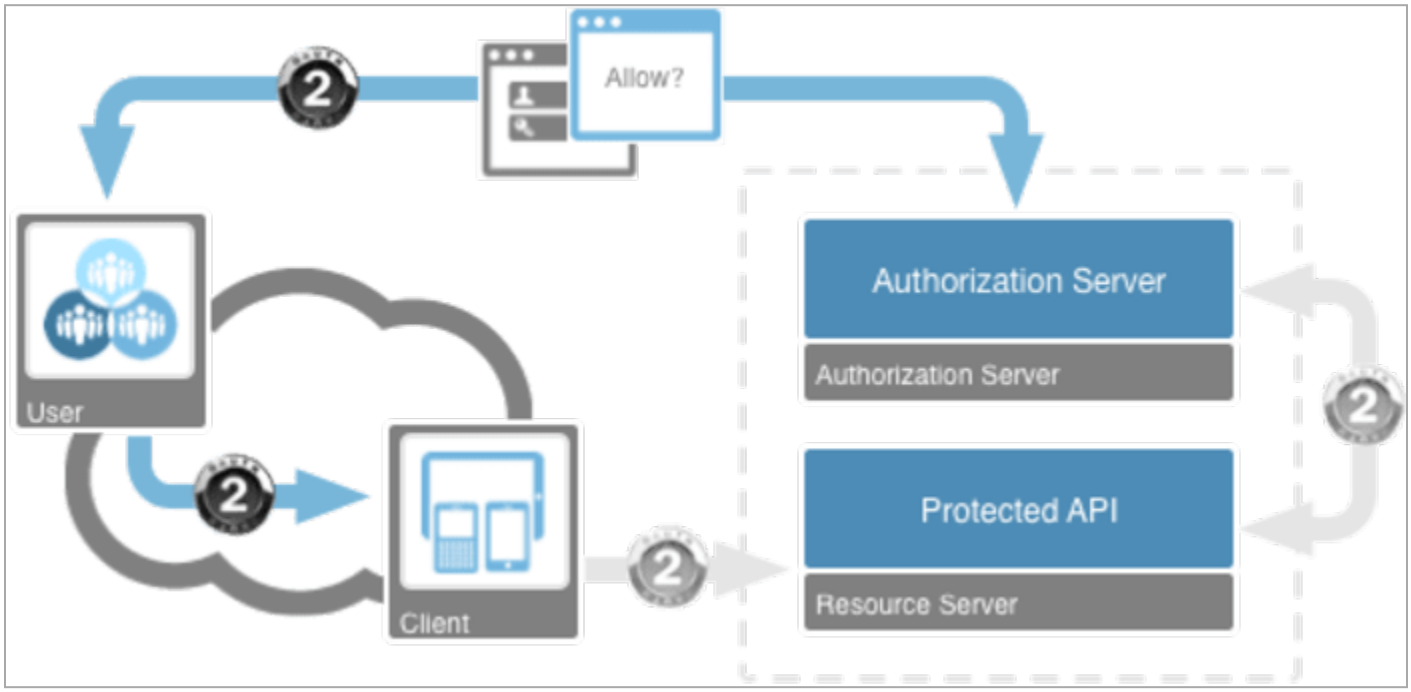
The WS-Trust standard introduces the concept of a Security Token Service (STS) that the web services client and the web services provider can lverage to broker the authentication. In the WS-Trust model, a security token (i.e. a SAML assertion) is issued by the STS for the web service client. This token is passed to the web services provider during the service call. The provider will validate this token against the STS and if valid, allow access to the web services call.

Selecting an Integration Method for APIs and Web Services

As we learned with web and mobile applications, federating API and web services security can greatly increase the flexibility of the APIs and services. By federating, you are replacing username/passwords with tokens allowing an external authentication system to handle the authentication complexity. This in turn allows APIs and services to use that token to authorise access to resources rather than manage the authentication process themselves.

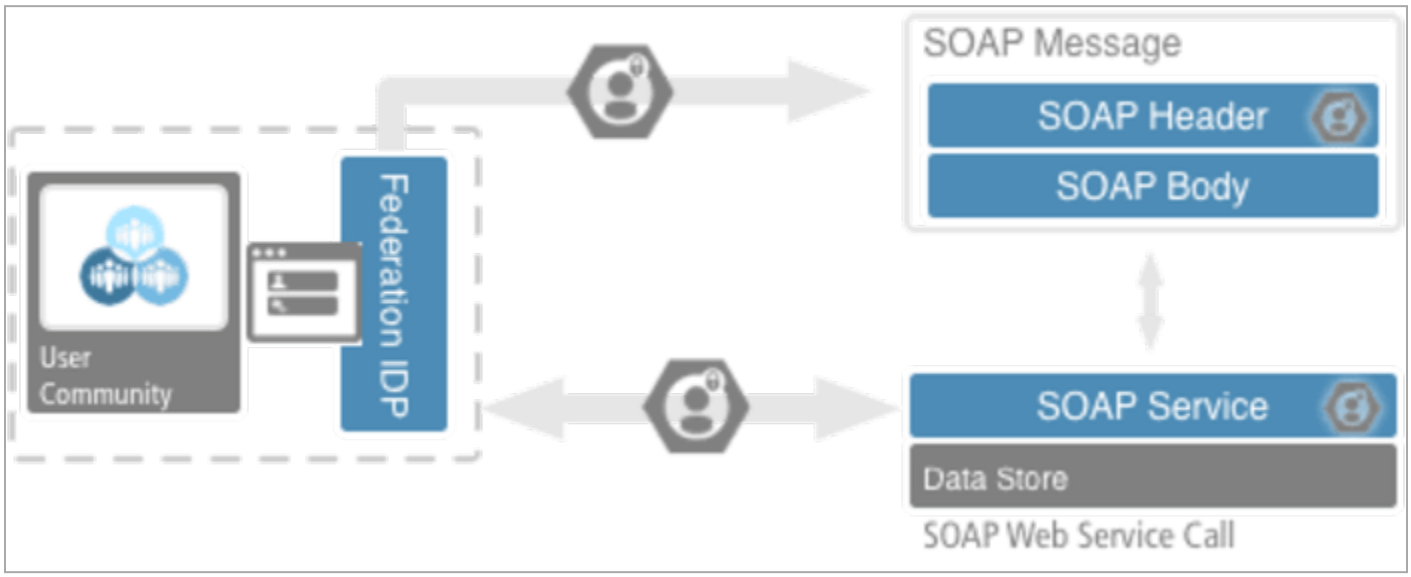
REST APIs

REST-based web services and APIs can leverage OAuth 2.0 for API protection. In the OAuth 2.0 terminology, the API will act as the Resource Server (RS). As a request is made to the API, an OAuth access_token will be presented as a bearer token in the "authorization" HTTP header. The API will validate this token and use the attributes provided in the token to authorize access to the API.



SOAP Services

SOAP services protected by WS-Security / WS-Trust standards leverage a Security Token Service (STS) to broker the federation transaction. The STS can exchange WS-Security tokens for federated security tokens (i.e. a SAML assertion) to provide cross-domain, federated access to your web services.NOTE: A SOAP web service call is essentiall a HTTP call, therefore a SOAP service can still take advantage of OAuth 2.0 to protect the call. The security will be processed and if the call is authorised, the SOAP message can be processed by the web service provider.



Federated SSO Primer

The goal of single sign-on is simple. Remove the need for users to remember a number of passwords to login to their applications.

For an application developer, SSO can enable:

- A web application wanting to log users in across multiple web assets without re-prompting them to login
- A user being able to log in to all their applications (on-prem and SaaS) by only typing their password once (or maybe not at all)
- Removing passwords from applications; reducing risk and aligning SaaS applications with organizational IAM policies

This guide will walk you through the concepts and considerations of authentication, Single Sign-On (SSO) through to federated SSO. Learn what "Federation" means and why open standard federation protocols enable cross-domain identity propagation.

In the following sections, we will provide a background into the protocols, roles and terminology involved in open standard federation protocols and how you as a developer can leverage these protocols to secure and enable identity in your application.

A Single, Sign-on Process

Prior to diving into SSO, lets re-visit the general process a user follows to login to a traditional application:

Authentication

The authentication step is used to determine the identity of the user accessing the application or service. By challenging them with a action (i.e. entering a username and password) that only that person will be able to successfully complete, the application can be reasonably comfortable that the user accessing the system is who they say they are.

In the application world, the most common form of authentication request is the login screen asking for a username and password. These credentials will be validated against a data store and, if valid, the user will be successfully authenticated.

The result of the authentication process is that we now know who is attempting to access our application or service.

User Profile

Prior to the authorization process the application generally wants to know more information about the user logging in to use in authorization decisions (and in some cases, also for personalization of the application).

The attributes required for authorization and application functionality can be handled in different ways:

- Application stores the attributes in its local store
- Application can store the attributes in a remote store
- Attributes can be provided along with the authentication request

After the application has received the information that it needs about the user (i.e. groups / roles) we can continue to the next step.

Authorization and Access Control

Access control rules can be defined by both the organization and the application. As a user accesses an application (or part of an application), the application will check to see if they are authorized to access that page or to perform the action. If the user requesting access meets the requirements of the access rules, they will be authorized access to the page or action. An example of authorization includes validating whether a user exists in a particular group before allowing them access.

Session Management

Once a user has been identified, their attributes retrieved and they are authorized to access the application, the final step the application makes is to create a session inside the application. From there the user can interact with the application without being prompted for authentication again (until the session expires or they log off).

After these steps are complete, the application knows who the user is, has allowed them access and has provided them a session so they may use the application without being prompted for authentication at every request.

Achieving Single Sign-on

For SSO to be achieved, this process needs to be repeated for each application accessed by the user. Various mechanisms can be used to reduce or eliminate the burden of logging in to each application, such as:

- Using a centralised authentication store for all applications (i.e. leveraging LDAP and Active Directory for authentication) - the user will have the same username and password for each application however will be expected to log in to each application.
- Using a domain trust to establish SSO - this could involve using a protocol like Kerberos to achieve single-sign on when the user is on the corporate network.
- Leveraging a Web Access Management (WAM) solution where all applications are protected by an access gateway, the user logs in once to the WAM gateway and gains access to all applications they are entitled to.
- Leveraging a shared cookie or token across multiple applications.
- Vaulting passwords and replaying them as a user access an application.

We see that to achieve the end goal of SSO between applications, we must replace the password with a trusted token (i.e. a kerberos ticket or a cookie) or system (i.e. a WAM).

Extending Single Sign-on to the Cloud

A new challenge arrives when the requirement for cross-domain authentication is introduced "browser SSO" where the user authenticating is no longer in the same domain as the application. A few examples of where this occurs are:

- User wants to access an enterprise application from their home PC or the coffee shop
- Enterprise user accesses a SaaS application and wants to sign on with their enterprise credentials
- Consumer-facing site wants to single sign-on the user to another web asset (either another web application or a 3rd party website)

SSO protocols such as Kerberos rely on the user being located inside a trusted environment and being able to contact authoritative authentication servers (i.e. an Active Directory Domain Controller) so how to support single sign-on across multiple applications located in different security domains. Vaulting passwords is clearly not a secure cross-domain solution as there is still a password stored in the third-party application, meaning when the authentication provider disables a user or changes their password, this is not reflected in the third party application.

Federated sign-on provides a portable trust between an authentication provider and a service provider. When a user from an organization requests authentication to a service provider they will be redirected to their home organization for authentication and, if successful, will be redirected back to the application with a token confirming the authentication.

As the user accesses additional applications, the process is repeated. By retaining a session at the authentication provider an organization can achieve SSO across multiple domains and applications.

For web applications, federated single sign-on uses the web browser to allow the user to interact with both the application and the authentication provider to negotiate authentication. As this "browser SSO" process uses the web browser, all communication is between the end-user and the federation partner (ie between the user and the authentication provider or the user and the application provider) this means that there is no communication direct between the authentication provider and the application provider, no firewall rules, no VPN and reduced risk to an enterprise.

Protocols and Standards

Open standard protocols define how the two parties (application provider and authentication provider) build a trust and communicate to authenticate the identity. Standards are critical as they allow inter-operability between different organizations and vendors - enabling connections to be made to many partners and applications easily and securely. Federation becomes a simple task when the only question that needs to be asked is "Do you speak SAML?".

There are three open standard federation protocols used widely today:

Security Assertion Markup Language (SAML)

SAML is the most common protocol use to provide SSO to SaaS applications today. There are three versions SAML 1.0, 1.1 and 2.0 with version 2.0 being the most common implementation of SAML available.

WS-Federation

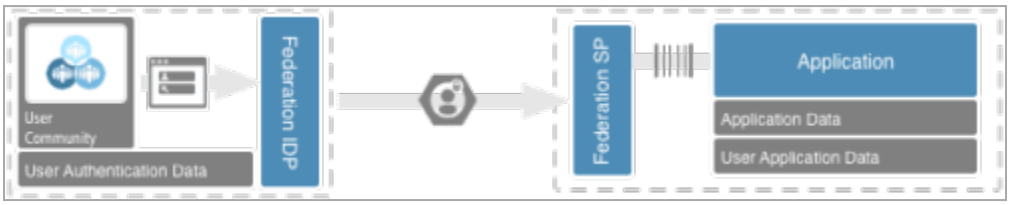
WS-Federation is the passive or browser SSO protocol that is part of the WS-* family of protocols. This protocol is widely used in Microsoft and IBM environments.

OpenID Connect

A relatively new protocol (ratified in February 2014), Connect is designed as an extension to the popular OAuth 2.0 protocol used in web service security. OpenID Connect adds an authentication and identity layer on top of the core OAuth 2.0 protocol allowing an application to authenticate a user and receive a token it can use for API calls via the same process. OpenID Connect is optimal for internal SSO, web application SSO and mobile SSO.

Roles and Components

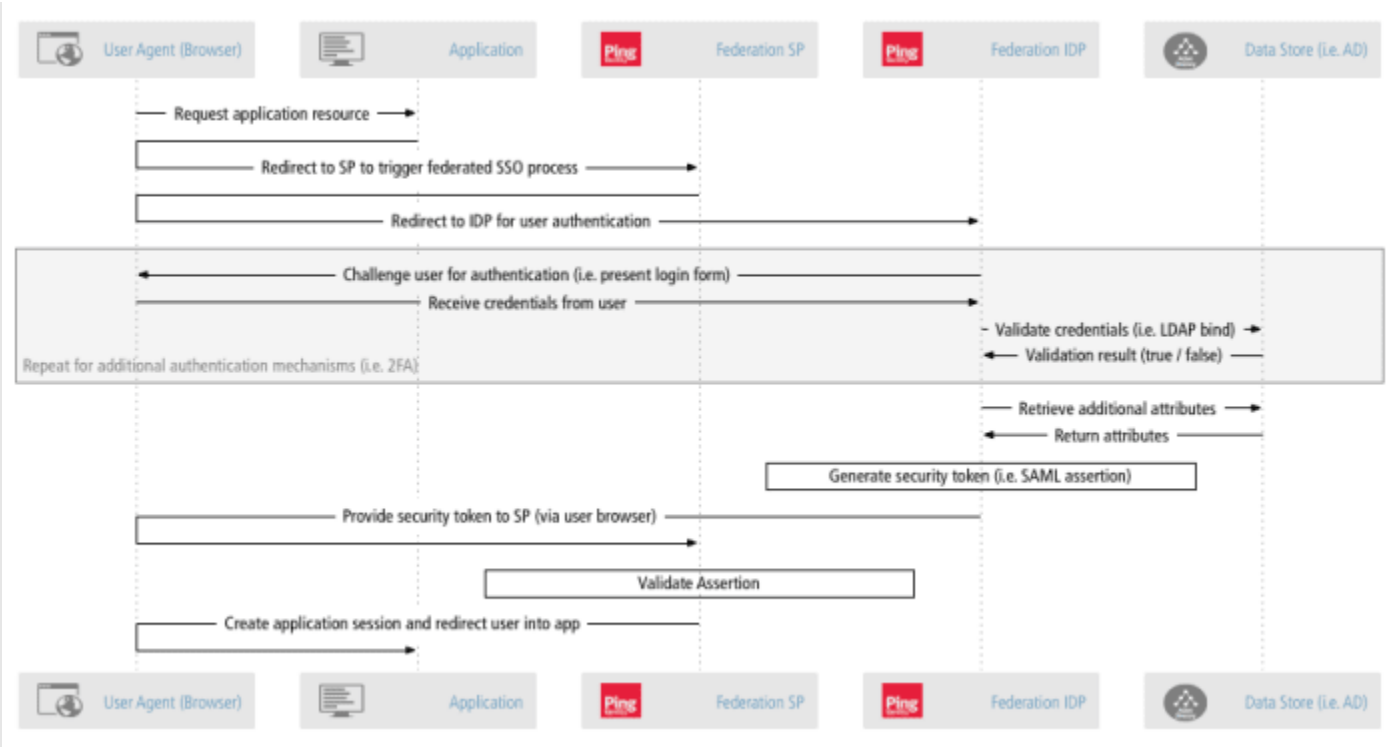
There are two main roles in a browser SSO transaction, the Identity Provider (IDP) who is responsible for authenticating the identity and generating a security token and the Service Provider (SP) who consumes that security token (receives and validates it) and translates the asserted identity into an application session.



Along with the Identity Provider and Service Provider roles, there are a number of components that complete the overall federation scenario:

Component	Description
Security Token	The security token or assertion is used to convey the identity of the authenticated user from the IDP over to the SP in a standard manner. The assertion contains security attributes, a digital signature and identity attributes. A SAML assertion and an OpenID Connect ID token are examples of federated security tokens.
Identity Store	The Identity Store is where the user authentication data is stored. Federation allows for this ID store to be moved from the Service Provider to the Identity Provider reducing the password proliferation challenge. A single identity store can be used to store passwords leading to increased security.
Authentication Integration Point	The interface that the authentication provider uses to authenticate the user. This may be a HTML form, an X509 certificate, Windows Authentication or other custom authentication process such as multi-factor.
Application Integration Point	On the Service Provider side, once a security token has been issued, received and validated at the SP an application session can be generated based on the asserted identity. The process to integrate an application into the federated process occurs at this application integration point.
Federated Trust	For a Service Provider to trust an assertion generated by an Identity Provider a federated trust must be established. Generally this is achieved by swapping a metadata file which contains information about the connection and a digital signature certificate.

Browser SSO Flow



Introduction to JSON Web Tokens (JWT)

A JSON Web Token (JWT) is an open-standard token for securely sharing information between parties as an encoded JavaScript Object Notation (JSON) object.

JWT claims include information about a subject and transfer information about a user. JWTs are trusted because they are signed using either a secret (HMAC) or a public-private key pair (RSA or ECDSA). JWTs are defined in JSON Web Token (JWT) and Introduction to JSON Web Tokens.

JWTs are commonly used for authorization, and particularly single sign-on (SSO). For example, after authenticating to a service, information about the user is encoded and shared between other relevant parties in a JWT. Subsequent domains know who the user is and that they have already been authenticated by a trusted party, so the user can access many different services after signing on once.

JWTs are an increasingly common way to secure application programming interface (API)s. You can share JWTs using Uniform Resource Locator (URL)s, POST parameters, or HTTP headers. JWT claims are embedded as JSON objects, which are used as the payload of a JSON Web Signature (JWS) structure or as text for a JSON Web Encryption (JWE) structure.

JWT Components

A JWT is defined as either a JWS object or a JWE object. When a token is signed it uses JWS, and when encrypted it uses JWE. There are three main parts of a JWS or JWE that include a JWT claim:

Header

The type of encoded object in the payload and information about how it is encoded.

See the following example header:

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

This header shows a JWT that is integrity protected with the HMAC SHA-256 algorithm. The payload with a JWE including this header will be of a JWT signed and encrypted with the HMAC SHA-256 algorithm.

The type property can be left out if the JWSs and JWEs used by the application are JWT types. This property is intended to prevent confusion when different types are being used.

Payload

The JWT object itself, which is a set of claims.

See the following example payload:

```
{
  "aud": "https://api.pingone.com",
  "iss": "https://auth.pingone.com/abcdefg12345/as",
  "exp": "1300819380”
}
```

This payload has an audience, `aud`, of the PingOne API, an issuer, `iss`, of the PingOne Authorization Server, and has a set expiration date, `exp`. The claim names might vary depending on the application and service being used.

Signature

The header and payload encoded using the algorithm specified in the header. In this example, the signature is the encoded header concatenated with the encoded JWT payload encoded with the HMAC SHA-256 algorithm.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Javascript Object Signing and Encryption (JOSE)

A JWT defines the token format and uses supporting specifications to handle signing and encryption. This collection of specifications is known as JOSE and consists of the following components:

JSON Web Signature (JWS)

Defines the process to digitally sign a JWT

JSON Web Encryption (JWE)

Defines the process to encrypt a JWT

JSON Web Algortihm (JWA)

Defines a list of algorithms for digitally signing or encrypting

JSON Web Key (JWK)

Defines how a cryptographic key and sets of keys are represented

Encoded JSON objects within JWTs include a set of claims signed using an algorithm so that they cannot be altered after a token is issued. Claims can be signed digitally or protected using message authentication code (MAC) with encryption. Encryption without MAC is also an option. When a user signs on using credentials, the OpenID Connect (OIDC) specification dictates that they receive an ID token in return, which is always a JWT.

For more information on JOSE, see Javascript Object Signing and Encryption (JOSE).

OAuth 2.0 Developer Guide

This document provides a developer overview of the OAuth 2.0 protocol. It provides an overview of the processes an application developer and an API developer need to consider to implement the OAuth 2.0 protocol.

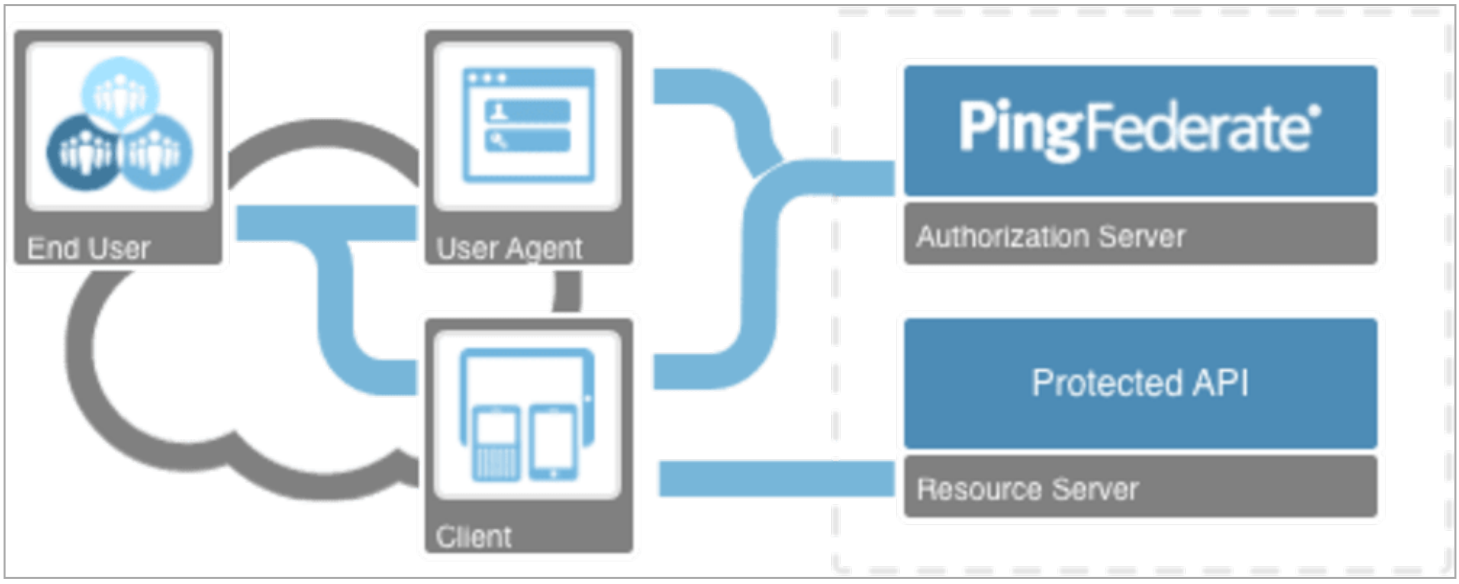
Explanations and code examples are provided for "quick win" integration efforts. As such they are incomplete and meant to complement existing documentation and specifications.

This document assumes familiarity with OAuth 2.0 protocol and PingFederate. For more information about OAuth 2.0, refer to:

- PingFederate Administrator's Manual
- OAuth 2.0 RFC 6749

The samples described in this document use the OAuth2 Playground sample application available for download from the products page on pingidentity.com.

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



The OAuth 2.0 protocol uses a number of actors to achieve the main tasks of getting an access token and using an access token. In addition, optional steps of refreshing this access token and validating the access token are also described.

The main actors involved are:

User or Resource Owner	The actual end user, responsible for authentication and to provide consent to share their resources with the requesting client.
User Agent	The user's browser. Used for redirect-based flows where the user must authenticate and optionally provide consent to share their resources.

Client	The client application that is requesting an access token on behalf of the end user.
Authorization Server (AS)	The PingFederate server that authenticates the user and/or client, issues access tokens and tracks the access tokens throughout their lifetime.
Resource Server (RS)	The target application or API that provides the requested resources. This actor will validate an access token to provide authorization for the action.

Application Developer Considerations

The application developer will be responsible for the user-facing elements of the process. They will need to authenticate the user and interface with the back-end APIs.

There are three main actions an application developer needs to handle to implement OAuth 2.0:

- Get an access token
- Use an access token
- Refresh an access token (optional)

API Developer Considerations

The API developer builds the API that the application talks to. This developer is concerned with the protection of the API calls made and determining whether a user is authorized to make a specific API call.

The OAuth 2.0 process an API developer needs to handle is to:

- Validate a token

> **Note**
>
> In some cases the "API Developer" may be using a service bus or authorization gateway to manage access to APIs and therefore the task of validating the access token would be shifted to this infrastructure.

Get Tokens

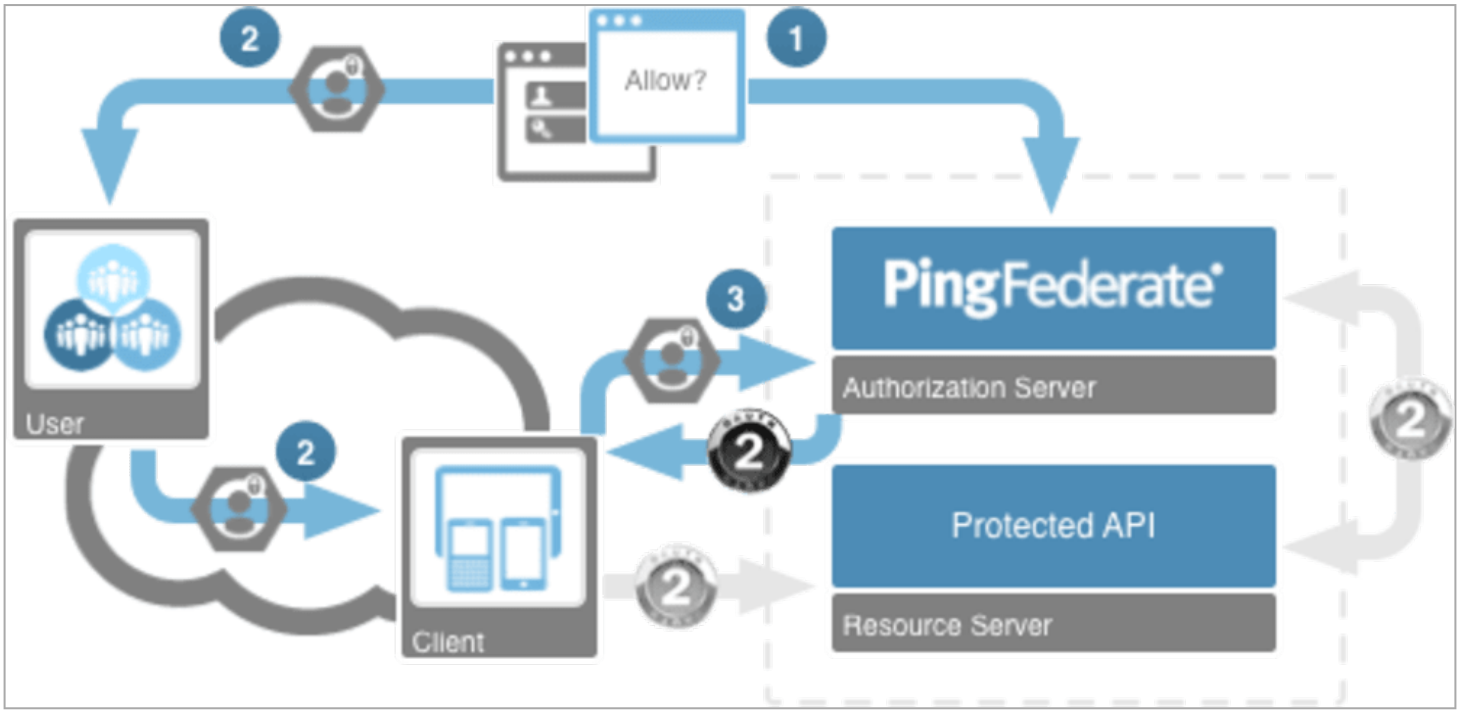
The most critical step for the application in the OAuth flow is how the client will receive an access token (and optionally a refresh token). The mechanism used to retrieve this token is called a grant type. Different grant types are more appropriate for different scenarios as we will discover in the following sections.

OAuth 2.0 provides four standard grant types and an extension grant type that can be used to customize the authentication and authorization process depending on the application requirements. These grant types are described in detail below.

Get Tokens: Authorization Code Grant Type

Authorization grant is a client redirect based flow. In this scenario:

1. The user will be redirected to the PingFederate authorization endpoint via the user agent (i.e. web browser). This user agent will be used to authenticate the end user and allow them to grant access to the client.
2. Once the user has been authorized, and intermediate code will be granted by the authorization server and returned to the client application via the user agent.
3. Lastly, the client will swap this code for an OAuth access token.



Capability	
Browser-based end user interaction	Yes
Can use external IDP for authentication	Yes
Requires client authentication	No*
Requires client to have knowledge of user credentials	No
Refresh token allowed	Yes
Access token is in context of end user	Yes

> **Note**
>
> Although the authorization code grant type does not require a client secret value, there are security implications to exchanging a code for an access token without client authentication.

Sample Client Configuration

For the authorization code grant type example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Client ID	client_id	ac_client
Client Authentication	client_secret	2Federate
Allowed Grant Types	response_type grant_type	• response_type of "code" (code) • grant_type of "authorization_code" (code) • grant_type of "refresh_token" (refresh)

Request authorization from user and retrieve authorization code

To initiate the process, the client application will redirect the user to the authorization endpoint. This redirect will contain the applicable attributes URL encoded and included in the query string component of the URL.

Using the above parameters as an example, the application will redirect the user to the following URL:

```
https://localhost:9031/as/authorization.oauth2?
client_id=ac_client&response_type=code&scope=edit&redirect_uri=sample%3A%2F%2Foauth2%2Fcode%2Fcb
```

This will initiate an authentication process using the browser (user agent). Once the user successfully completes the authorization request, they will be redirected with an authorization code to the redirect_uri value defined in the authorization request (if included) otherwise the user will be returned to the redirect_uri defined when the client was configured.

> **Note**
>
> For mobile scenarios, the redirect_uri may be a custom URL scheme that will cause the code to be returned to the native application.

Using the example above, a successful authorization request will result in the resource owner redirected to the following URL with the authorization code included as a code query string parameter:

```
sample://oauth2/code/cb?code=XYZ...123
```

> **Note**
>
> - If the authorization request also included a state value, this will also be included on this callback.
> - An error condition from the authentication / authorization process will be returned to this callback URI with error and error_description parameters.

The client will then extract the code value from the response and, optionally, verify that the state value matches the value provided in the authorization request

Swap the authorization code for an access token

The final step for the client is to swap the authorization code received in the previous step for an access token that can be used to authorize access to resources. By limiting the exposure of the access token to a direct HTTPS connection between the client application and the authorization endpoint, the risk of exposing this access token to an unauthorized party is reduced.

For this to occur, the client makes a HTTP POST request to the token endpoint on the AS. This request will use the following parameters sent in the body of the request:

Item	Description
grant_type	Required to be "authorization_code"
code	The authorization code received in the previous step
redirect_uri	If this was included in the authorization request, it MUST also be included

This request should also authenticate as the pre-configured client using either HTTP BASIC authentication or by including the client_id and client_secret values in the request.

To retrieve the access token in the example, the following request will be made:

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: BasicYWNfY2xpZW50OjJGZWRlcmF0ZQ==

grant_type=authorization_code
  &code=XYZ...123
```

A successful response to this message will result in a 200 OK HTTP response and the access token (and optional refresh token) returned in a JSON structure in the body of the response.

```
HTTP/1.1 200 OK

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

{
   "access_token":"zzz...yyy",
   "token_type":"Bearer",
   "expires_in":14400,
   "refresh_token":"123...789"
}
```

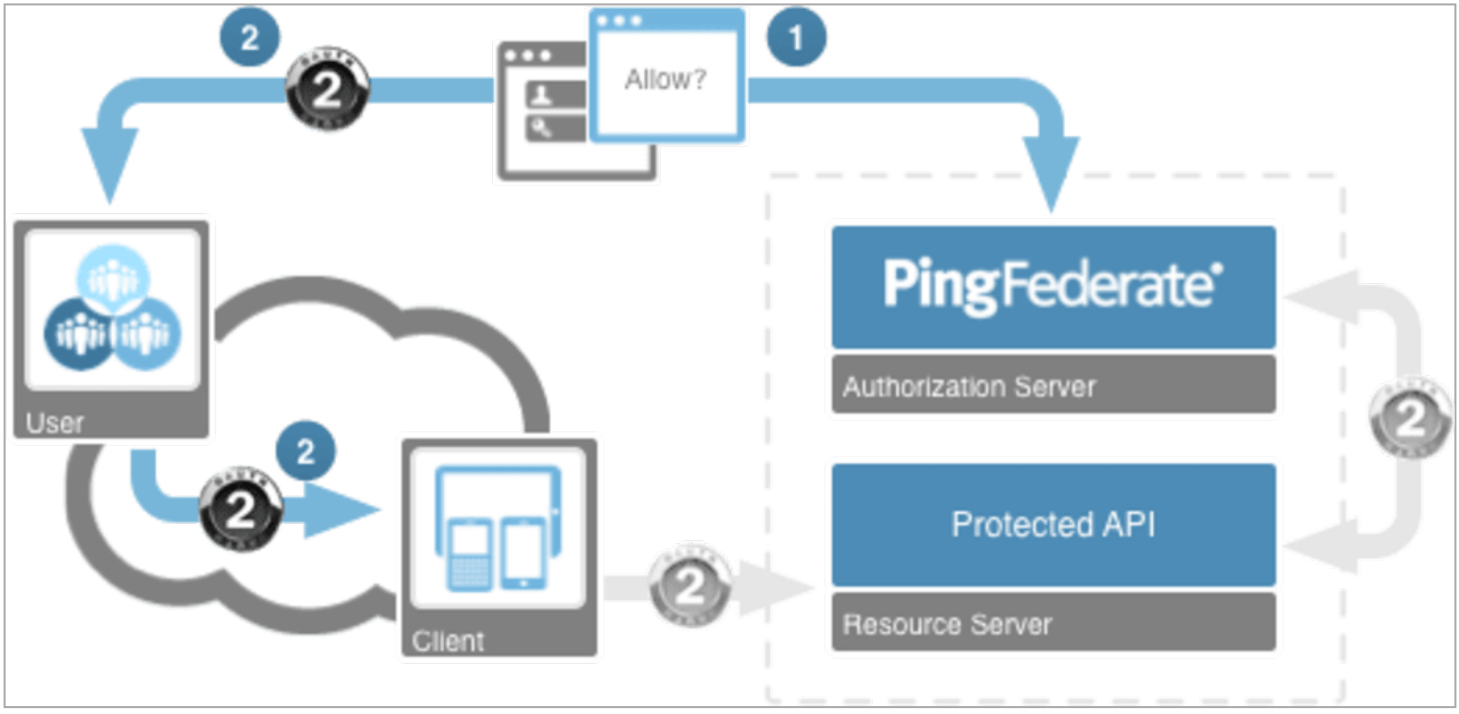
The application can now parse the access token and, if present, the refresh token to use for authorization to resources. If a refresh token was returned, it can be used to refresh access token once it expires.

Get Tokens: Implicit Grant Type

The implicit grant is similar to an authorization code grant, however the user agent will receive an access token directly from an authorization request (rather than swapping an intermediate authorization code).

In this flow,

- the user requests authentication and authorization via the user agent (step 1 below). If authorized, the authorization server will redirect the user to a URL containing the access token in a URL fragment.
- The client can then parse this from the URL (step 2) to use for requests to protected resources.



This grant type is suitable for clients that are unable to keep a secret (i.e. client-side applications like JavaScript). The client is mapped to the authorization server via the redirect_uri, as there is no client secret to authenticate the client, the access token will be sent to a specific URL pre-negotiated between the client and the authorization server.

As the access token is provided to the client in the request URI, it is inherently less secure than the authorization code grant type. For this reason, an implicit grant type cannot take advantage of refresh tokens. Only access tokens can be provided via this grant type.

Capability	
Browser-based end user interaction	Yes
Can use external IDP for authentication	Yes
Requires client authentication	No
Requires client to have knowledge of user credentials	No
Refresh token allowed	No
Access token is in context of end user	Yes

Sample Client Configuration

For the implicit grant type example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Client ID	client_id	im_client
Allowed Grant Types	response_type	response_type of "token"
Redirect URIs	redirect_uri	sample://oauth2/code/cb
Scope Settings	scope	edit

Getting the Token

To initiate the process, the client application will redirect the user to the authorization endpoint. This redirect will contain the applicable attributes URL encoded and included in the query string component of the URL.

Using the above parameters as an example, the application will redirect the user to the following URL:

```
https://localhost:9031/as/authorization.oauth2?
client_id=im_client&response_type=token&scope=edit&redirect_uri=sample%3A%2F%2Foauth2%2Fimplicit%2Fcb
```

This will initiate an authentication process using the browser (user agent). Once the user has authenticated and approved the authorization request they will be redirected to the configured URI with the access token included as a fragment of the URL. A refresh token will NOT be returned to the client:

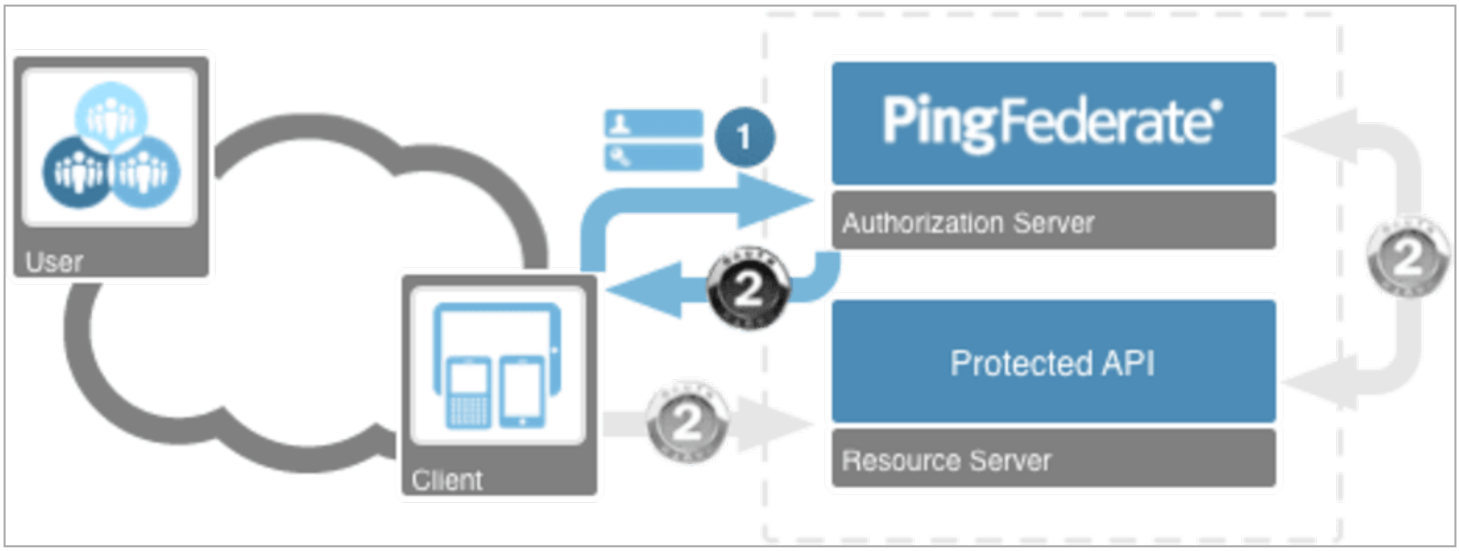
```
sample://oauth2/implicit/cb#access_token=zzz...yyy&token_type=bearer&expires_in=14400
```

Note

- For mobile scenarios, the redirect_uri may be a custom URL scheme which will cause the access token to be returned to the native application.
- The implicit response is returned via a URL fragment. The fragment is only visible from client-side code. Therefore if you need to parse the values from server-side code, you must post the values to the server for parsing.

Get Tokens: Client Credentials Grant Type

The client credentials type works in a similar way to the ROPC grant type and is used to provide an access token to a client based on the credentials or the client, not the resource owner. In this grant type, the client credentials are swapped for an access token (step 1 below).



Capability	
Browser-based end user interaction	No
Can use external IDP for authentication	No
Requires client authentication	Yes
Requires client to have knowledge of user credentials	No
Refresh token allowed	No
Access token is in context of end user	No

Sample Client Configuration

For the client credentials example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Client ID	client_id	cc_client
Client Authentication	client_secret	2Federate
Allowed Grant Types	grant_type	grant_type of "client_credentials"
Scope Settings	scope	edit

Getting the Token

The client makes a request (HTTP POST) to the token endpoint with the client credentials presented as HTTP Basic authentication:

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: Basic Y2NfY2xpZW50OjJGZWRlcmF0ZQ==

grant_type=client_credentials
  &scope=edit
```

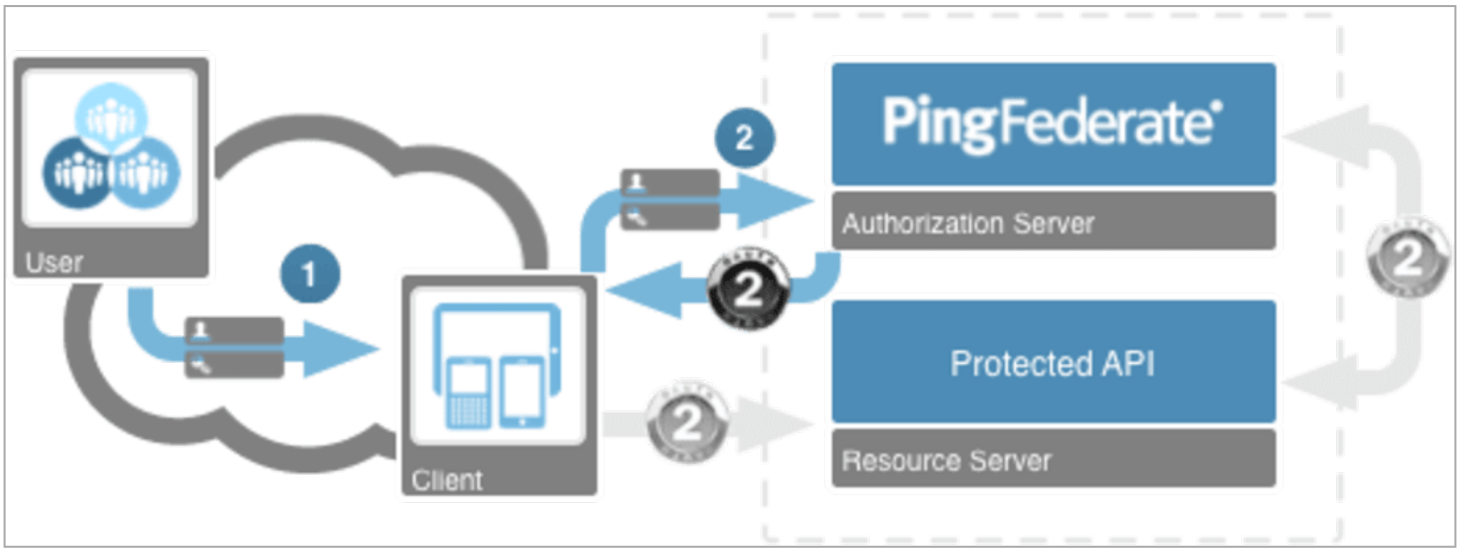
> **Note**
>
> The client credentials can also be provided using the client_id and client_secret parameters in the contents of the POST.

The client will receive a response to this request. If successful, a 200 OK response will be received and the access token will be returned in a JSON structure. A refresh token will NOT be returned to the client.

```
HTTP/1.1 200 OK

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

{
  "access_token":"zzz...yyy",
  "token_type":"Bearer",
  "expires_in":14400,
}
```

Get Tokens: Resource Owner Password Credentials

The ROPC grant type can be used in scenarios where an interactive user agent is not available, where specific design requirements warrant the use of a native application login interface, or for legacy reasons (i.e. retro-fitting a login form for OAuth2). In the ROPC grant type, the client captures the user credentials (step 1 below) and uses those credentials to swap for an access token (step 2).



Capability	
Browser-based end user interaction	No
Can use external IDP for authentication	No
Requires client authentication	No
Requires client to have knowledge of user credentials	Yes
Refresh token allowed	Yes
Access token is in context of end user	Yes

Sample Client Configuration

For the resource owner password credentials grant type example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Client ID	client_id	ro_client
Client Authentication	client_secret	2Federate

Admin Label	OAuth2 Parameter	Example Value
Allowed Grant Types	response_type grant_type	• grant_type of "password" (ropc) • grant_type of "refresh_token" (refresh)

Getting the Tokens

At this stage, the client displays a login form to the user and collects the credentials (i.e. username/password) and defined scope if required from the resource owner (user) and makes a HTTP POST to the token endpoint.

For the example below, the following credentials were received by the client and are used to request an access token:

> **Note**
>
> The credentials passed via the Resource Owner Password Credential flow are processed through a PingFederate Password Credential Validator. These credentials do not have to be a username and password, they could be for example a username / PIN combination or another credential that is validated by a PCV.

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: Basic cm9fY2xpZW50OjJGZWRlcmF0ZQ==

grant_type=password&username=joe&password=2Federate&scope=edit
```

If successful, the client will receive a 200 OK response to this request and the access token (and optional refresh token) will be returned in a JSON structure:

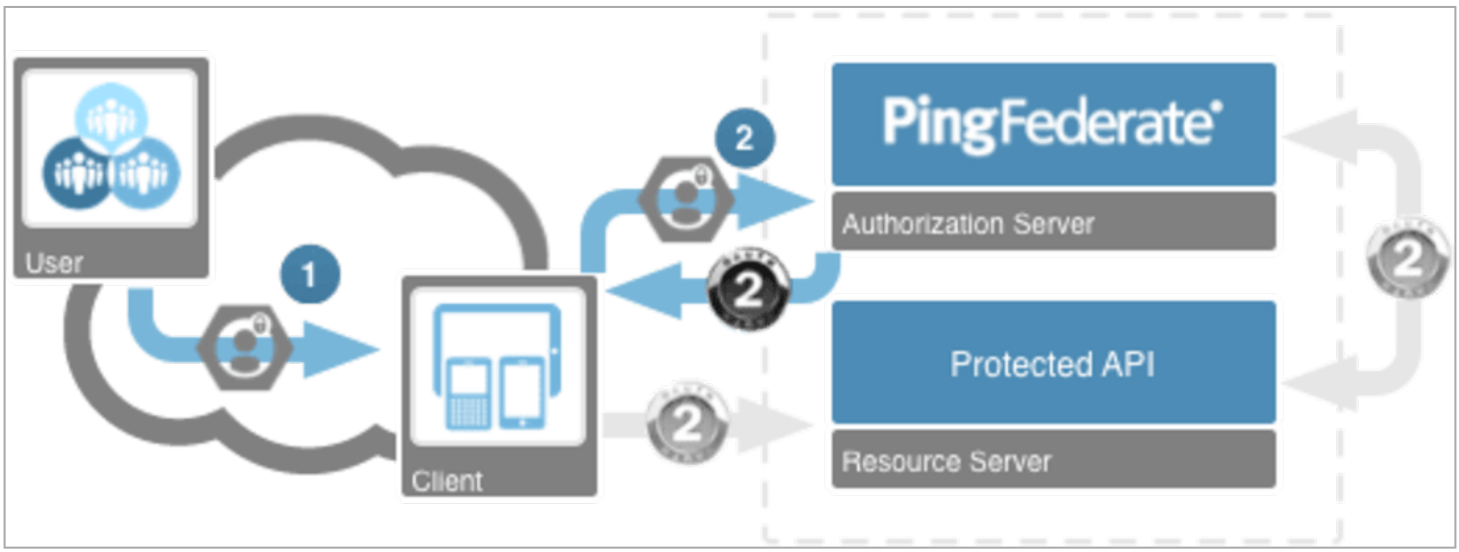
```
HTTP/1.1 200 OK

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

{
  "access_token":"zzz...yyy",
  "token_type":"Bearer",
  "expires_in":14400,
  "refresh_token":"123...789"
}
```

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

The application can now parse the access token and, if present, the refresh token to use for authorization to resources. If a refresh token was returned, it can be used to refresh access token once it expires.

Get Tokens: Extension Grants

The extension grant type provides support for additional grant types extending the OAuth2.0 specifications. An example is the use of the SAML 2.0 Bearer extension grant. In this grant type, a SAML assertion (indicated by step 1 below, however the process used to acquire this SAML assertion is out of scope of this document) can be exchanged for an OAuth 2.0 access token (step 2).



Capability	
Browser-based end user interaction	No[1]
Can use external IDP for authentication	Yes[2]
Requires client authentication	No
Requires client to have knowledge of user credentials	No
Refresh token allowed	No
Access token is in context of end user	Maybe[3]

[1] Although the grant type does not allow for user interaction, the process to generate the SAML assertion used in this flow can involve user interaction.

[2] As long as the PingFederate AS is able to verify the SAML assertion, this assertion can be generated from a foreign STS.

[3] Access token will be in the context of the subject of the SAML assertion, which may be an end-user a service or the client itself.

Sample Client Configuration

For the SAML bearer extension grant type example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Client ID	client_id	saml_client
Client Authentication	client_secret	2Federate
Allowed Grant Types	grant_type	grant_type of "urn:ietf:params:oauth:grant-type:saml2-bearer"

Getting a Token

At this stage, the client has a SAML assertion that it needs to exchange for an OAuth 2.0 access token. The process in which the client received the assertion is out of scope (i.e. bootstrap assertion, STS token exchange), however the client would Base64 URL encode the assertion and include it in a HTTP POST to the token endpoint.

For the example below, the following SAML assertion (abbreviated for readability) was received by the client and is are used to request an access token:

```
PHNhbWw6QXNzZXJ0aW9uIElEPSJTdXdCSDdiQjM3cWVmT0tycmlaZkc3Y09H
...
Pjwvc2FtbDpBc3NlcnRpb24-
```

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: Basic c2FtbF9jbGllbnQ6MkZlZGVyYXRl

grant_type= urn:ietf:params:oauth:grant-type:saml2-bearer&assertion=
PHNhbWw6QXNzZXJ0aW9uIElEPSJTdXdCSDdiQjM3cWVmT0tycmlaZkc3Y09H...Pjwvc2FtbDpBc3NlcnRpb24-&scope=edit
```

> **Note**
>
> The client credentials can also be provided using the client_id and client_secret parameters in the contents of the POST.

The client will receive a response to this request. If successful, a 200 OK response will be received and the access token will be returned in a JSON structure. A refresh token will NOT be returned to the client.

```
HTTP/1.1 200 OK

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

{
  "access_token":"zzz...yyy",
  "token_type":"Bearer",
  "expires_in":14400
}
```

Refresh the Token

If a refresh token was requested along with the access token, then the refresh token can be used to request a new access token without having to ask the user to re-authenticate. If the refresh token is still valid, then a new access token and refresh token will be returned to the client.

If the refresh token has been invalidated for any reason, then the client must require the user to re-authenticate to retrieve a new access token. The reasons for refresh tokens becoming invalid are:

- Refresh token has expired;
- Refresh token has been administratively revoked (separation / security reasons);
- User has explicitly revoked the refresh token

To refresh a token, the access token must have been requested with a grant type that supports refresh tokens (authorization code or resource owner password credentials). A request will then be made to the token endpoint with the grant_type parameter set to "refresh_token".

> **Note**
>
> A new access token can be requested with a scope of equal or lesser value than the original access token request. Refreshing an access token with additional scopes will return an error. If the scope parameter is omitted, then access token will be valid for the original request scope.

Sample Client Configuration

For the refresh token example below, the client configuration for the authorization code grant type from above will be used to refresh the token:

Admin Label	OAuth2 Parameter	Example Value
Scope Settings	scope	edit
Client ID	client_id	ac_client
Client Authentication	client_secret	2Federate

Admin Label	OAuth2 Parameter	Example Value
Allowed Grant Types	response_type grant_type	• response_type of "code" (code) • grant_type of "authorization_code" (code) • grant_type of "refresh_token" (refresh)

Refreshing the Token

The following request is made by the client:

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: Basic YWNfY2xpZW50OjJGZWRlcmF0ZQ==

grant_type=refresh_token&refresh_token=123...789
```

Note

A token can only be refreshed with the same or a lesser scope than the original token issued. If the token is being refreshed with the same scope as the original request, the scope parameter can be omitted. If a greater scope is required, the client must re-authenticate the user.

A successful response to this message will result in a 200 OK HTTP response and the following JSON structure in the body of the response:

```
HTTP/1.1 200 OK

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

{
  "access_token":"aaa...ccc",
  "token_type":"Bearer",
  "expires_in":14400,
  "refresh_token":"456...321"
}
```

Note

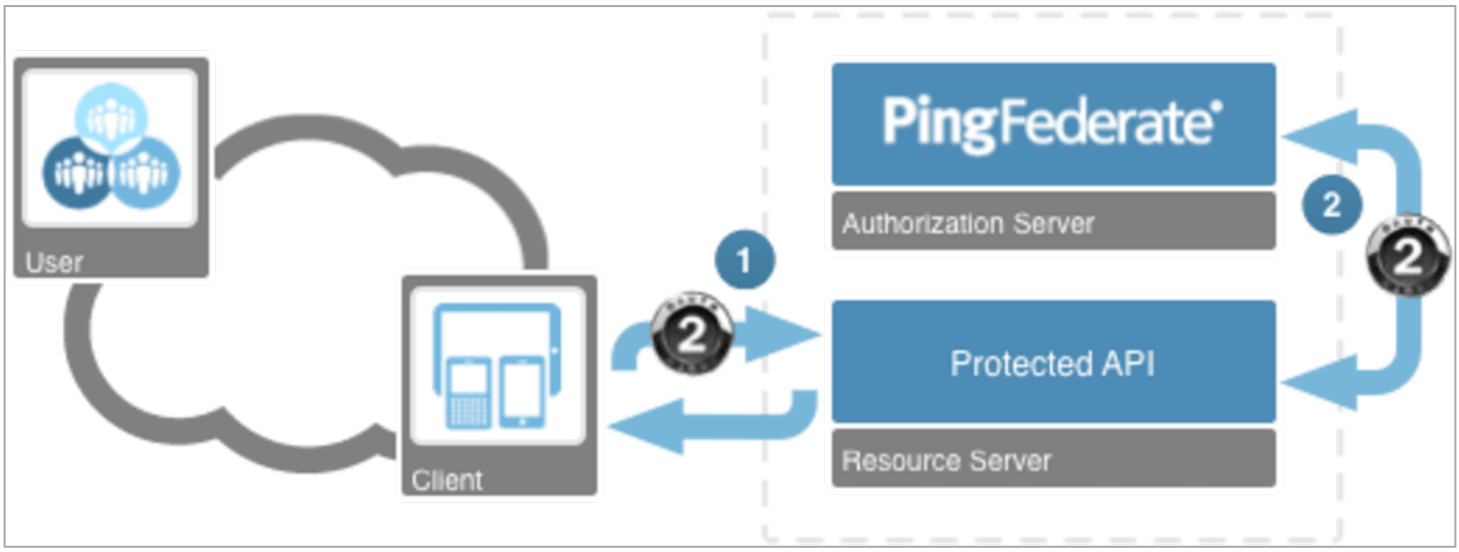
Depending on the PingFederate configuration, the client could be configured to roll the refresh token returned from a refresh token request. i.e. a new refresh token is returned and the original refresh token is invalidated.

Use the Token

An access token can then be used as an authorization token to configured web services. To use an access token to access a protected resource, the access token must be passed to the resource server.

The client should use a bearer authorization method as defined in RFC 6750 to present the access token to the resource. The most common approach is to use the HTTP Authorization header and include the access token as a Bearer authorization credential, however RFC 6750 also defines mechanisms for presenting an access token via query string and in a post body.

In the diagram below, the client presents the OAuth 2.0 access token to the protected resource (step 1). The resource then validates the access token before returning the requested resource (if authorized).



Using a Token

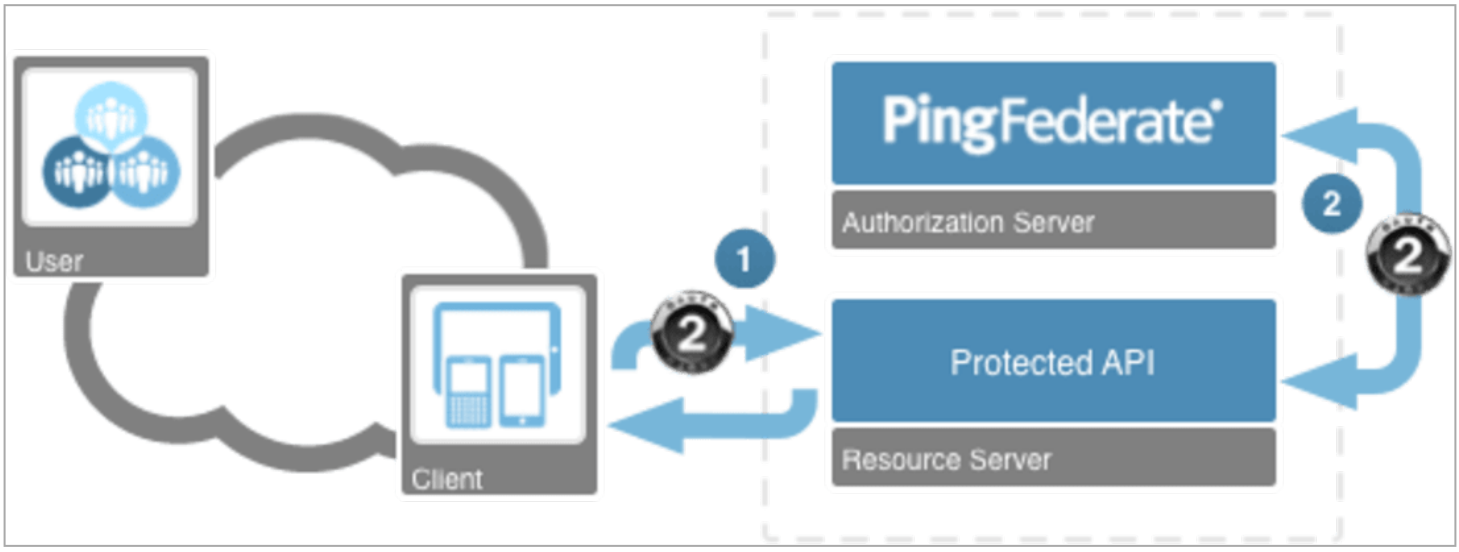
For example, to enact a GET request on a REST web service, given an access token AAA...ZZZ, the client makes the following HTTP request:

```
GET https://api.company.com/user HTTP/1.1
Authorization: Bearer AAA...ZZZ
```

This will provide the access token to the resource server, which can then validate the token, verify the scope or the request, the identity of the resource owner and the client and perform the appropriate action if authorized.

Validate the Token

For an API developer to integrate with OAuth 2.0, the resource must accept and validate the OAuth 2.0 access token (step 1 below). Once the token has been received, the resource can then validate the access token against the PingFederate authorization server (step 2). The response from the access token validation will include attributes that the resource can use for authorization decisions.



Note

- This section will demonstrate the manual method of validating an access token through code. This effort could also be handled by an API gateway / service bus architecture or by the API validating a JWT formatted token internally.
- The OAuth 2.0 specifications do not define a standard mechanism for access token validation. The process described in this section is specific to a PingFederate implementation.

Sample Client Configuration

For the access token validation example below, the following client information will be used:

Admin Label	OAuth2 Parameter	Example Value
Redirect URIs	redirect_uri	sample://oauth2/code/cb
Scope Settings	scope	edit
Refresh Token	refresh_token	123...789
Client ID	client_id	rs_client
Client Authentication	client_secret	2Federate
Allowed Grant Types	grant_type	Access Token Validation (Client is a Resource Server) • grant_type of "urn:pingidentity.com:oauth2:grant_type:validate_bearer"

Validating the Token

The API first needs to receive the access token from the client as it was provided per the "Use a Token" section of this guide.

A request from a client would look similar to the following:

```
GET https://api.company.com/user HTTP/1.1
Authorization: Bearer AAA...ZZZ
```

In order to fulfill the request, the API first extracts the access token from the authorization header, then queries the token endpoint of the PingFederate AS to validate the token:

```
POST https://localhost:9031/as/token.oauth2 HTTP/1.1

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

Authorization: Basic cnNfY2xpZW50OjJGZWRlcmF0ZQ==

grant_type=urn:pingidentity.com:oauth2:grant_type:validate_bearer&token=AAA...ZZZ
```

A successful response to this message will result in a 200 OK HTTP response and a JSON structure in the body of the response similar to the following:

```
HTTP/1.1 200 OK

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

{
  "access_token": { "role":"all_access" },
  "token_type":"Bearer",
  "expires_in":14400,
  "scope":"edit",
  "client_id":"ac_client"
}
```

The resource server can then use this information to make an authorization decision and allow or deny the web request.

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 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 Developer 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.eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWNfY2xpZW50IiwianRpIjoidWY5MFNLNH
dzY0ZoY3RVVDZEdHZiMiIsImlzcyI6Imh0dHBzOlwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6MTM5NDA2MDg1MywiZXhwIjoxMzk0MDYx
MTUzLCJub25jZSI6ImU5NTdmZmJhLTlhNzgtNGVhOS04ZWNhLWFlOGM0ZWY5Yzg1NiIsImF0X2hhc2giOiJ3Zmd2bUU5VnhqQXVkc2w5bG
M2VHFBIn0.lr4L-oT7DJi7Re0eSZDstAdOKHwSvjZfR-OpdWSOmsrw0QVeI7oaIcehyKUFpPFDXDR0-RsEzqno0yek-_U-Ui5EM-yv0Pia
UOmJK1U-ws_C-fCplUFSE7SK-TrCwaOow4_7FN5L4i4NAa_WqgOjZPloT8o3kKyTkBL7GdITL8rEe4BDK8L6mLqHJrFX4SsEduPk0CyHJS
ykRqzYS2MEJlncocBBI4up5Y5g2BNEb0aV4VZwYjmrv9oOUC_yC1Fb4Js5Ry1t6P4Q8q_2ka5OcArlo188XH7lMgPA2GnwSFGHBhccjpxh
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	eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWNfY2xpZW50IiwianRpIjoidWY5MFNLNHdzY0ZoY3RVVDZEdHZiMiIsImlzcyI6Imh0dHBzOlwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6MTM5NDA2MDg1MywiZXhwIjoxMzk0MDYxMTUzLCJub25jZSI6ImU5NTdmZmJhLTlhNzgtNGVhOS04ZWNhLWFlOGM0ZWY5Yzg1NiIsImF0X2hhc2giOiJ3Zmd2bUU5VnhqQXVkc2w5bGM2VHFBIn0	\{ "sub":"joe", "aud":"im_oic_client", "jti":"uf90SK4wscFhctUT6Dtvb2", "iss":"https:\/\/localhost:9031", "iat":1394060853, "exp":1394061153, "nonce":"e957ffba-9a78-4ea9-8eca-ae8c4ef9c856", "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

Claim	Description
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-fCplUFSE7SK-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

Step #	Test Summary
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
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

"alg" Value	Signature Method	Signing Key
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.eyJzdWIiOiJqb2UiLCJhdWQiOiJpbV9vaWNfY2xpZW50IiwianRpIjoidWY5MFNLNHdz
Y0ZoY3RVVDZEdHZiMiIsImlzcyI6Imh0dHBzOlwvXC9sb2NhbGhvc3Q6OTAzMSIsImlhdCI6MTM5NDA2MDg1MywiZXhwIjoxMzk0MDYxMTUz
LCJub25jZSI6ImU5NTdmZmJhLTlhNzgtNGVhOS04ZWNhLWFlOGM0ZWY5Yzg1NiIsImF0X2hhc2giOiJ3Zmd2bUU5VnhqQXVkc2w5bGM2VHFB
In0
```

Signature value to verify:

```
lr4L-oT7DJi7Re0eSZDstAdOKHwSvjZfR-OpdWSOmsrw0QVeI7oaIcehyKUFpPFDXDR0-RsEzqno0yek-_U-Ui5EM-yv0PiaUOmJK1U-
ws_C
-fCplUFSE7SK-
TrCwaOow4_7FN5L4i-4NAa_WqgOjZPloT8o3kKyTkBL7GdITL8rEe4BDK8L6mLqHJrFX4SsEduPk0CyHJSykRqzYS2MEJln
cocBBI4up5Y5g2BNEb0aV4VZwYjmrv9oOUC_yC1Fb4Js5Ry1t6P4Q8q_2ka5OcArlo188XH7lMgPA2GnwSFGHBhccjpxhN7S46ubGPXRBNsn
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:

Step 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

Step 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
}
```

Step 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":"mdrLAp5GR8o5d5qbwWTYqNGuSXHTIE6w9HxV445oMACOWRuwlOGVZeKJQXHM9cs5Dm7iUfNVk4pJBttUxzcnhVCRf
      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

Connect scope	Returned Claims
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.openidNOTE: 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:

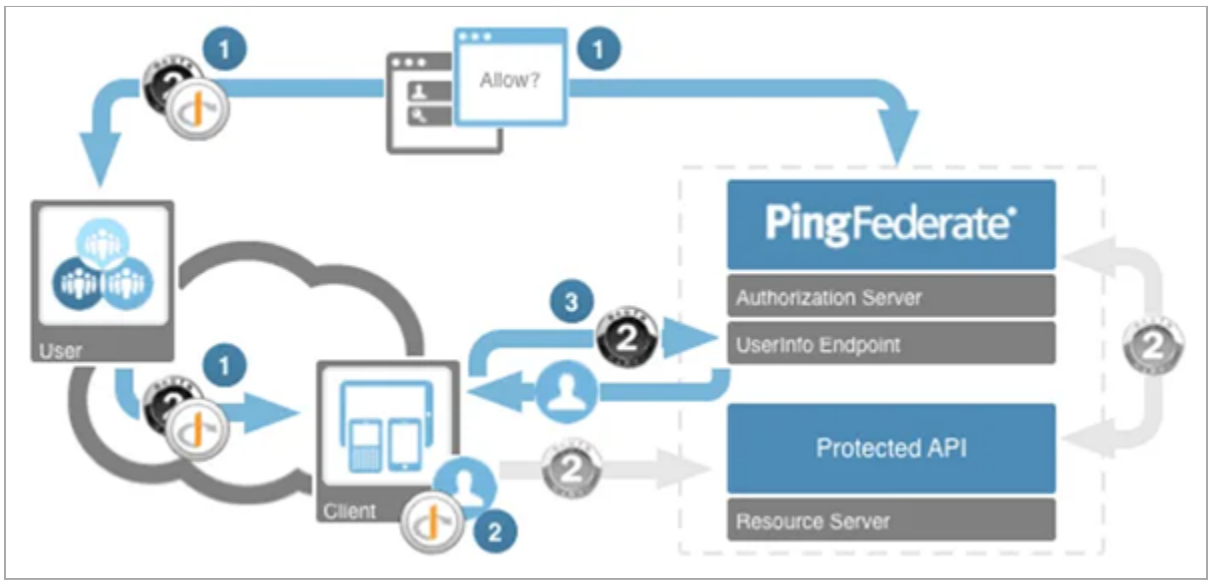
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "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.eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImltX29pY19jbGllbnQiLCJqdGkiOiJUOU4xUklkRkVzUE45enU3ZWw2eng2IiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczNzA3MSwiZXhwIjoxMzkzNzM3MzcxLCJub25jZSI6ImNiYTU2NjY2LTRiMTItNDU2YS04NDA3LTNkMzAyM2ZhMTAwMiIsImF0X2hhc2giOiJrdHFvZVBhc2praVY5b2Z0X3o5NnJBIn0.g1Jc9DohWFfFG3ppWfvW16ib6YBaONC5VMs8J61i5j5QLieY-mBEeVi1D3vr5IFWCfivY4hZcHtoJHgZk1qCumkAMDymsLGX-IGA7yFU8LOjUdR4IlCPlZxZ_vhqr_0gQ9pCFKDkiOv1LVv5x3YgAdhHhpZhxK6rWxojg2RddzvZ9Xi5u2V1UZ0jukwyG2d4PRzDn7WoRNDGwYOEt4qY7lv_NO2TY2eAklP-xYBWu0b9FBElapnstqbZgAXdndNs-Wqp4gyQG5D0owLzxPErR9MnpQfgNcai-PlWI_UrvoopKNbX0ai2zfkuQ-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	eyJhbGciOiJSUzI1NiIsImtpZCI6IjRvaXU4In0	<pre>{
		  "alg":"RS256",
		  "kid":"4oiu8"
		}</pre>

Payload	eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImltX29pY19jbGllbnQiLCJqdGkiOiJUOU4xUklkRkVzUE45enU3ZWw2eng2IiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczNzA3MSwiZXhwIjoxMzkzNzM3MzcxLCJub25jZSI6ImNiYTU2NjY2LTRiMTItNDU2YS04NDA3LTNkMzAyM2ZhMTAwMiIsImF0X2hhc2giOiJrdHFvZVBhc2praVY5b2Z0X3o5NnJBIn0	<pre>{
		  "sub":"nfyfe",
		  "aud":"im_oic_client",
		
		"jti":"T9N1RIdFEsPN9zu7el6zx6",
		  "iss":"https:\/\/localhost:9031",
		  "iat":1393737071,
		  "exp":1393737371,
		
		"nonce":"cba56666-4b12-456a-8407-3d3023fa1002",
		
		"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

Step #	Test Summary	Result
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 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:

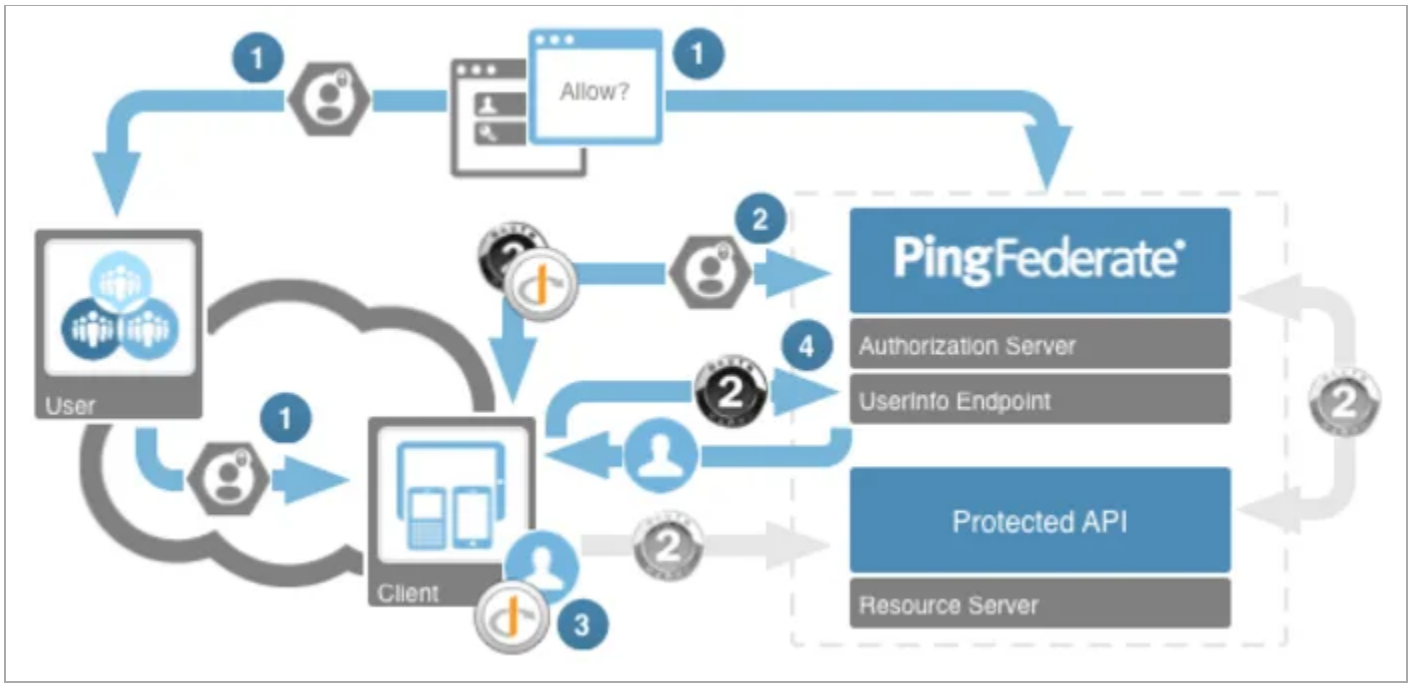
```
{
        "sub":"nfyfe",
        "family_name":"Fyfe",
        "given_name":"Nathan",
        "nickname":"Nat",
        ...[additional claims]...
}
```

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	abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910SRQPOnmlijhoauthplaygroundapplication
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=abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910SRQPOnmlijhoauthplaygroundapplication&
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=abc123DEFghijklmnop4567rstuvwxyzZYXWUT8910SRQPOnmlijhoauthplaygroundapplication&
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.eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImFjX29pY19jbGllbnQiLCJqdGki

JIR1AwdnlxbmgwOVBjQ3MzenBHbUVsIiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczMDM4MCwi

ZXhwIjoxMzkzNzMwNjgwfQ.EQeAm84Xj2lekxUMSK9H3BvoCl511JV1TWHCyQQ7vTnXcuvZYdBHE9_OpIr9gD5OHjoDrOhwVEjKUqvwwGhzBPN

EueeY8bUgkTfIBKzUUJETSeaO1U8uH9Td0QYv7q3rRfurLhrpzubFbAIfjPOiv8jxgBjMyGEdPJ7aXtBwP_cr2RxMUzg_iBRA4cD8c4PwEORO
                0T-
xKnwZcocDZs_rYAOHFljLPgO2tX8BBePJfqUUUG46U1K4hSqo7LP3zru4BDE2wNbZyOhb2keeLjetNq2ES33YthNU9dkmHUgbtoD-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.eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImFjX29pY19jbGllbnQiLCJqdGkiOiJIR1AwdnlxbmgwOVBjQ3MzenBHbUVsIiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczMDM4MCwiZXhwIjoxMzkzNzMwNjgwfQ.EQeAm84Xj2lekxUMSK9H3BvoCl511JV1TWHCyQQ7vTnXcuvZYdBHE9_OpIr9gD5OHjoDrOhwVEjKUqvwwGhzBPNEueeY8bUgkTfIBKzUUJETSeaO1U8uH9Td0QYv7q3rRfurLhrpzubFbAIfjPOiv8jxgBjMyGEdPJ7aXtBwP_cr2RxMUzg_iBRA4cD8c4PwEOROr0T-xKnwZcocDZs_rYAOHFljLPgO2tX8BBePJfqUUUG46U1K4hSqo7LP3zru4BDE2wNbZyOhb2keeLjetNq2ES33YthNU9dkmHUgbtoD-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	eyJzdWIiOiJuZnlmZSIsImF1ZCI6ImFjX29pY19jbGllbnQiLCJqdGkiOiJIR1AwdnlxbmgwOVBjQ3MzenBHbUVsIiwiaXNzIjoiaHR0cHM6XC9cL3Nzby5tZXljbG91ZC5uZXQ6OTAzMSIsImlhdCI6MTM5MzczMDM4MCwiZXhwIjoxMzkzNzMwNjgwfQ	\{ "sub":"nfyfe", "aud":"ac_oic_client", "jti":"HGP0vyqnh09PcCs3zpGmEl", "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
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

Step #	Test Summary	Result
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.

SCIM 1.1 Developer Guide

SCIM or System for Cross-Domain Identity Management is a federated provisioning protocol. Providing a consistent API for user and group CRUD (Create, Read, Update and Delete) actions.

SCIM can be used by developers to standardize the way user profile information is retrieved from a data source (i.e. instead of having to manage connections to SQL tables, LDAP datastores and other data stores SCIM can provide a single interface to this data).

SCIM can also be used to provision user and group information from an enterprise to a partner (i.e. SaaS application) either as an out-of-band process or as part of an authentication action.

This developers guide provides a reference for developers to build against a SCIM data store interface to help standardize the method used to access identity data in a federated manner. It uses information from the SCIM v1.1 specifications (specifically the core schema and the SCIM API).

This developer guide references the SCIM v1.1 specifications:

- SCIM Core Schema 1.1
- SCIM Protocol API 1.1

> **Note**
>
> Although this guide provides raw protocol calls, it is highly recommended a developer utilize existing libraries to avoid implementation specific errors.

SCIM Actions

The SCIM protocol uses the REST concept to define the actions a SCIM Consumer can perform on a resource managed by a SCIM Service Provider. These actions are:

GET	Reads a resource (or resources) from the Service Provider
POST	Creates a new resource at the Service Provider
PUT	Updates an existing resource, as this action requires the entire resource provided in the body, it is more like a replace than an update
PATCH	Updates an existing resource with changes (where supported). **Note** The PATCH action is optional and not supported in some implementations
DELETE	Deletes a resource at the Service Provider

SCIM Components and Roles

SCIM contains three main components, as with a number of the federation protocols, the terminology can be slightly confusing so we describe the components below:

- **Service Provider** The provider of the identity information (in a traditional enterprise scenario, the SCIM Service Provider is most likely the same as the SAML Identity Provider). For a majority of this guide we will use the PingOne Directory as an example of a SCIM Service Provider.
- **SCIM Consumer** The application or service that will consume the SCIM data. For example in a federated provisioning scenario, the SCIM Consumer will be the 3rd party receiving the identity information.
- **Resource** The object (i.e. a User or a Group) that the SCIM request is being performed on.

SCIM Schema

SCIM provides a standard schema that can be used to represent a user or a group. This schema is extensible so additional schema objects can be added to provide custom schema support.

Along with the SCIM schema, specific data types are defined to simplify interoperability between partners.

SCIM Data Types

The SCIM core schema has a support for common data types to provide maximum interoperability between SCIM Service Providers and SCIM Consumers. The following data types are available in the SCIM specification and examples are provided in JSON representation:

String	"familyName" : "Archer"
Boolean	"active" : true
Decimal	"weight" : 173.2
Integer	"age" : 36
DateTime (xml date/time format)	"created" : "2015-05-18T15:00:00Z"
Binary (base64 encoded string)	"photo" : "U2F5IENoZWVzZSE="

SCIM Attribute Types

Attributes in SCIM can be either single-valued or multi-valued and SCIM can support complex attributes where an attribute can be comprised of multiple single or multi-valued sub attributes, for example:

Simple Attribute (single-valued)	An attribute that contains a single value	`{ "displayName": "Archer, Meredith A" }`
Simple Attribute (multi-valued)	An attribute that contains multiple values. Multiple values can include a "type" attribute to define the type of value specified (i.e. work vs home address).	`{ "emails": [ { "type" : "other", "value" : "marcher@pingdevelopers.com" }, { "type": "work", "value" : "meredith.archer@pingdevelopers.com" } ] }`
Complex Attribute	An attribute that contains one or more simple attributes	`{ "name": { "familyName": "Archer", "givenName": "Meredith", "displayName" : "Archer, Meredith A" } }`
Sub-Attribute	An attribute that is a member of a complex attribute.	using the previous example, "familyName" is a sub-attribute of "name"

Common Schema Attributes

Common schema elements must be included on all resources and are used to provide a reference identifier for the resource as well as information about the resource:

id		String	Unique identifier for the resource as defined by the Service Provider [REQUIRED]

externalId		String	Identifier for the resource as defined by the SCIM Consumer (i.e. a local identifier or customerId in an application) [REQUIRED]
meta		Complex Attribute	The resources metadata, the "meta" complex attribute may consist of the following attributes: [REQUIRED]
created	meta	DateTime	When the resource was created
lastModified	meta	DateTime	When the resource was last modified (if the resource has not been modified since creation, this value will be the same as the created attribute)
location	meta	String	The direct URI of the resource. You can use this URI to directly manage a resource rather than searching for it and then modifying.
version	meta	String	(if supported). The version of the resource being returned.
attributes	meta	String (multi-valued)	(if supported). Contains the list of attributes to remove during a PATCH operation.

SCIM User Attributes

A SCIM User consists of one required attribute (userName) and additional descriptive attributes:

userName		String	Unique identifier for the User as described by the SCIM Consumer (typically the user name used to login) [REQUIRED]
name		Complex Attribute	Components of the user's real name:

formatted	name	String	The formatted representation of the user (i.e. "Ms Meredith Anne Archer, II")
familyName	name	String	Family or last name of the user (i.e. Archer)
givenName	name	String	Given or first name of the user (i.e. Meredith)
middleName	name	String	The middle name(s) or initial(s) of the user (i.e. Anne)
honorificPrefix	name	String	Honorific or personal title of the user (i.e. Mr, Ms)
honorificSuffix	name	String	Honorific or generational suffix of the user (i.e. Jr, II)
displayName		String	How the user name should be presented in an application, this is not necessarily tied to the formatted name attribute (i.e. Archer, Meredith A)
nickName		String	Casual or preferred representation of the user's name (i.e. Bob rather than Robert)
profileUrl		String	A fully qualified URL of the users profile (i.e. https://profiles.pingdevelopers.com/marcher)
title		String	Work title of the user (i.e. "Software Developer")
userType		String	Defines the relationship of the user to the SCIM Service Provider organization (i.e. Employee)
preferredLanguage		String	User's preferred language and dialect (i.e. en_US)

locale		String	User's locale for localization purposes (currency, date time format etc) (i.e. en_US)
timezone		String	User's timezone in the "Olson" timezone database format (i.e. America/Denver)
active		Boolean	Whether the user is active at the Service Provider
password		String	WRITE-ONLY. The user's clear-text password. Can only be provided in a POST operation (for a create) or a PUT operation for a password change.
emails		String Multi-valued	Email address(es) for the user. Common "type" values are work, home, other.
phoneNumbers		String Multi-valued	Telephone number(s) for the user. Common "type" values are work, home, fax, pager, mobile, other.
ims		String Multi-valued	Instant messaging address(es) for the user. Common "type" values are gtalk, icq, aim, skype
photos		String Multi-valued	URL of a profile photo for the user. Common "type" values are thumbnail, photo
addresses		Complex Attribute Multi-Valued	Physical mailing address for the user. Common "type" values are home, work, other
formatted	addresses	String	Full mailing address formatted for display (i.e. 1001 17th Street\nSuite 100\nDenver CO 80202)
streetAddress	addresses	String	Full street address component (i.e. 1001 17th Street\nSuite 100)

locality	addresses	String	The city or locality (i.e. Denver)
region	addresses	String	The region / state / province of the address (i.e. CO)
postalCode	addresses	String	The postal code or zipcode of the address (i.e. 80202)
country	addresses	String	ISO3166-1 alpha 2 "short" format of the country (i.e. US)
groups		String Multi-Valued	List of groups the user belongs to
entitlements		Undefined Multi-Valued	List of entitlements for the user (SCIM doesn't specify a format for these entitlements)
roles		String Multi-Valued	List of roles the user has (although SCIM doesn't specify a format for roles, its expected that they are a list of String values)
x509Certificates		Binary	List of x509 certificates for the user. Value is a binary (base64 encoded) DER encoded x509 certificate.

Group Schema Attributes

A group resource can be used to define roles or groups a user is a member of.NOTE: Groups can be nested inside other groups to provide users with indirect membership of a group.

displayName		String	Human readable name for the group [REQUIRED]
members		String Multi-Valued	List of members of the group, the value will be the "id" value of the resource (user or group) and the multi-valued attribute type may be "user" or "group"

SCIM Action: Create a Resource

SCIM resources are created using HTTP POST verb. The body of the call contains a JSON formatted SCIM schema resource.

Creating a User

A user create is performed by a POST operation. The JSON representation of the user (according to the SCIM schema) is POSTed to the User endpoint.

```
curl -v -X POST --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '{
"schemas":["urn:scim:schemas:core:1.0"],
"userName":"marcher",
"password":"2Federate",
"active":true,
"name":{ "familyName":"Archer", "givenName":"Meredith" },
"emails": [ { "type": "work", "value": "meredith.archer@pingdevelopers.com }]
}' \
https://directory-api.pingone.com:443/api/directory/user
```

A successful request will result in a HTTP 200 OK response and the JSON representation of the user that was just created:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
  "id":"a7d67610-ceb5-4350-ba5a-746472c4f1f7",
  "schemas": [
    "urn:scim:schemas:core:1.0",
    "urn:scim:schemas:com_pingone:1.0"
  ],
  "urn:scim:schemas:com_pingone:1.0": {
     "createTimeStamp":1429123454227,
     "accountId":"a6538050-412a-4bca-a44d-07deb4b073a8",
     "lastModifiedTimeStamp":1429123454227,
     "directoryId":"90b3dfe3-f8d0-45ad-8c04-047c88b03137",
     "state":"ACTIVE"
  },
  "name": {
     "familyName":"Archer",
     "givenName":"Meredith"
  },
  "userName":"marcher",
  "active":true,
  "emails":[
    {
      "value":"meredith.archer@pingdevelopers.com",
      "type":"work"
    }
  ],
  "meta": {
    "lastModified":"2015-04-15T12:44:14.227-06:00",
    "location":"https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7",
    "created":"2015-04-15T12:44:14.227-06:00"
  },
  "groups":[
    {
      "display":"Users",
      "value":"0b854f8d-a291-4e95-ad4b-68474a666e55"
    }
  ]
}
```

In the response, the id attribute contains the unique identifier for this user in the PingOne Directory. To modify this user, add them to groups etc you will reference them via this attribute value. The location attribute contains the full resource URL that you will use to manipulate the resource in the next sections.

Creating a Group

Creating a group is a similar process, performing a POST operation against the Group endpoint:

```
curl -v -X POST --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
-d ' {
"schemas":["urn:scim:schemas:core:1.0"],
"displayName":"Software Developers" }' \
https://directory-api.pingone.com:443/api/directory/group
```

A successful request will result in a HTTP 200 OK response and the JSON representation of the group that was just created:

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

{
  "id":"7c513a7e-55d4-441c-858c-7329e6268084",
  "displayName":"Software Developers",
  "schemas": [
    "urn:scim:schemas:core:1.0",
    "urn:scim:schemas:com_pingone:1.0"
  ],
  "meta": {
    "lastModified":"2015-04-16T10:08:22.324-06:00",
    "created":"2015-04-16T10:08:22.324-06:00",
    "location":"https://directory-api.pingone.com/v1/group/7c513a7e-55d4-441c-858c-7329e6268084"
  },
  "urn:scim:schemas:com_pingone:1.0": {
    "createTimeStamp":1429200502324,
    "lastModifiedTimeStamp":1429200502324,
    "accountId":"a6538050-412a-4bca-a44d-07deb4b073a8",
    "directoryId":"90b3dfe3-f8d0-45ad-8c04-047c88b03137"
  }
}
```

SCIM Action: Read a Resource

To retrieve or read a resource from the SCIM Service Provider, the HTTP GET verb is used. There are two mechanisms that can be used to retrieve a user - via a filter or directly. These are both described below.

The SCIM query defines filters that can be used to constrain the returned list of users as well as sorting and pagination mechanisms.NOTE: SCIM query, sort and pagination functionality is optional in the SCIM specifications so varying degrees of support may be found in SCIM Service Providers.

Retrieving Users via Filter

To query for a user you will use the SCIM filter language to form your query. The filter language is fairly straightforward and the operators are defined below:

eq	equal	The attribute and operator values must be identical for a match.

co	contains	The entire operator value must be a substring of the attribute value for a match.
sw	starts with	The entire operator value must be a substring of the attribute value, starting at the beginning of the attribute value. This criterion is satisfied if the two strings are identical.
pr	present (has value)	If the attribute has a non-empty value, or if it contains a non-empty node for complex attributes there is a match.
gt	greater than	If the attribute value is greater than operator value, there is a match. The actual comparison is dependent on the attribute type. For string attribute types, this is a lexicographical comparison and for DateTime types, it is a chronological comparison.
ge	greater than or equal	If the attribute value is greater than or equal to the operator value, there is a match. The actual comparison is dependent on the attribute type. For string attribute types, this is a lexicographical comparison and for DateTime types, it is a chronological comparison.
lt	less than	If the attribute value is less than operator value, there is a match. The actual comparison is dependent on the attribute type. For string attribute types, this is a lexicographical comparison and for DateTime types, it is a chronological comparison.
le	less than or equal	If the attribute value is less than or equal to the operator value, there is a match. The actual comparison is dependent on the attribute type. For string attribute types, this is a lexicographical comparison and for DateTime types, it is a chronological comparison.

A simple example of a SCIM filter is to find a specific record by username:

```
username eq "marcher"
```

The filter expressions can be joined using the "and" and "or" logical operators and grouped via parentheses, for example to find all users who have a familyName that starts with "A" and have been modified since the 1st January 2015 we can use the filter:

```
(name.familyName sw "A") and (urn:scim:schemas:com_pingone:1.0:createTimeStamp gt 1420084800000)
```

> **Note**
>
> When submitting the GET request with a filter, be sure to urlencode the filter value.

To retrieve our recently created user we will use the filter: username eq marcher which will be urlencoded and appended to the User endpoint as the filter parameter. Both requests below will result in the same response:

```
curl -v --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
https://directory-api.pingone.com/api/directory/user?filter=userName%20eq%20%22marcher%22
```

Retrieving Users Directly

If we know the resources location value, we can perform a GET directly on the user resource (which is defined in the "location" attribute in the "meta" section of the user record). So to retrieve Meredith's profile directly I can also perform the following command:

```
curl -v --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7
```

A successful request will result in a HTTP 200 OK response and the JSON representation of the user:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
  "id":"a7d67610-ceb5-4350-ba5a-746472c4f1f7",
  "schemas": [
    "urn:scim:schemas:core:1.0",
    "urn:scim:schemas:com_pingone:1.0"
  ],
  "urn:scim:schemas:com_pingone:1.0": {
    "createTimeStamp":1429123454227,
    "accountId":"a6538050-412a-4bca-a44d-07deb4b073a8",
    "lastModifiedTimeStamp":1429123454227,
    "directoryId":"90b3dfe3-f8d0-45ad-8c04-047c88b03137",
    "state":"ACTIVE"
  },
  "name": {
    "familyName":"Archer",
    "givenName":"Meredith"
  },
  "userName":"marcher",
  "active":true,
  "emails": [
    {
      "value":"meredith.archer@pingdevelopers.com",
      "type":"work"
    }
  ],
  "meta": {
    "lastModified":"2015-04-15T12:44:14.227-06:00",
    "location":"https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7",
    "created":"2015-04-15T12:44:14.227-06:00"
  },
  "groups": [
    {
      "display":"Users",
      "value":"0b854f8d-a291-4e95-ad4b-68474a666e55"
    }
  ]
}
```

SCIM Action: Update a Resource

SCIM resource modifications are performed using a PUT operation returning the entire profile with the changes applied. For example, to add a "title" to Meredith's record I will take the contents of a GET request, add in the title attribute and PUT it back to Meredith's User resource location.

```
curl -v “X PUT --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '{
"id":"a7d67610-ceb5-4350-ba5a-746472c4f1f7",
"schemas":[ "urn:scim:schemas:core:1.0", "urn:scim:schemas:com_pingone:1.0" ],
"urn:scim:schemas:com_pingone:1.0":{
  "createTimeStamp":1429123454227,
  "accountId":"a6538050-412a-4bca-a44d-07deb4b073a8",
  "lastModifiedTimeStamp":1429123454227,
  "directoryId":"90b3dfe3-f8d0-45ad-8c04-047c88b03137",
  "state":"ACTIVE" },
"name":{ "familyName":"Archer", "givenName":"Meredith" },
"userName":"marcher",
"title":"Software Developer",
"active":true,
"emails":[{"value":"meredith.archer@pingdevelopers.com","type":"work" }],
"meta":{
  "lastModified":"2015-04-15T12:44:14.227-06:00",
  "location":"https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-
746472c4f1f7",
  "created":"2015-04-15T12:44:14.227-06:00" },
"groups":[{ "display":"Users", "value":"0b854f8d-a291-4e95-ad4b-68474a666e55" }]
}' \
https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7
```

A successful response from a modify operation is the user profile returned in full:

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

{
  "id":"a7d67610-ceb5-4350-ba5a-746472c4f1f7",
  "schemas": [
    "urn:scim:schemas:core:1.0",
    "urn:scim:schemas:com_pingone:1.0"
  ],
  "urn:scim:schemas:com_pingone:1.0": {
    "createTimeStamp":1429123454227,
    "accountId":"a6538050-412a-4bca-a44d-07deb4b073a8",
    "lastModifiedTimeStamp":1429123456527,
    "directoryId":"90b3dfe3-f8d0-45ad-8c04-047c88b03137",
    "state":"ACTIVE"
  },
  "name": {
    "familyName":"Archer",
    "givenName":"Meredith"
  },
  "userName":"marcher",
  "title":"Software Developer",
  "active":true,
  "emails": [
    {
      "value":"meredith.archer@pingdevelopers.com",
      "type":"work"
    }
  ],
  "meta": {
    "lastModified":"2015-04-15T12:46:18.224-06:00",
    "location":"https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7",
    "created":"2015-04-15T12:44:14.227-06:00"
  },
  "groups": [
    {
      "display":"Users",
      "value":"0b854f8d-a291-4e95-ad4b-68474a666e55"
    }
  ]
}
```

SCIM Action: Delete a Resource

To delete a resource (user or group) you will use the DELETE operation against the resource's endpoint. The following command will delete Meredith from the directory and return a 200 OK message if successful.

```
curl -v -X DELETE --user 1234-aaaa-bbbb-5678:eXJzbmVha3kh \
-H "Content-Type: application/json" \
-H "Accept: application/json" \
https://directory-api.pingone.com/v1/user/a7d67610-ceb5-4350-ba5a-746472c4f1f7
```