Engineer

OpenID Connect

an identity layer on top of OAuth for signing users in

What it is

OpenID Connect is a standard way for an application to learn who a person is from a service that has just signed them in. Its specification calls it a simple identity layer on top of OAuth 2.0, and that is how it works: the same redirects, endpoints and tokens as OAuth, plus one more token, one more endpoint and the rules for checking them. It is usually shortened to OIDC. The core specification became final in February 2014; the current text, which folds in its second set of corrections, is dated 15 December 2023 and is maintained by a working group of the OpenID Foundation.

OAuth on its own answers the question “what may this application do?” It hands the application an access token meant for an API, and the application is not supposed to read it. OpenID Connect adds the question “who signed in?” and answers it with an ID token: a JSON Web Token, addressed to the application itself, that states who the person is, who vouches for that, when it was issued and which application it is meant for. ID tokens must be signed, except that the specification permits `alg: none` only if the response type returns no ID token from the authorization endpoint and the client explicitly requested `none` at registration.

The specification renames the OAuth roles for this job. The OpenID Provider (OP) is an OAuth authorization server that authenticates people and issues ID tokens. The Relying Party (RP) is the OAuth client that needs to know who the person is. The person is the End-User. A request becomes an OpenID Connect request by putting `openid` in its scope, and that value is required: “If the openid scope value is not present, the behavior is entirely unspecified.”

Further scope values ask for profile data. `profile` covers name, nickname, picture, locale and similar claims, `email` adds `email` and `email_verified`, and `address` and `phone` do what they say. The application can fetch those claims from the provider's UserInfo endpoint, an ordinary OAuth-protected API called with the access token.

There are three ways to run it. The authorization code flow sends only a short-lived code through the browser and collects the tokens with an HTTPS request to the token endpoint; the implicit flow returns tokens straight in the redirect; the hybrid flow mixes the two. A companion specification, OpenID Connect Discovery 1.0, lets the application look up every endpoint and signing key from a single address.

Use the concept in a decision

Put "What is OpenID Connect?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

The reason to adopt it is single sign-on. With one identity provider in front of many applications, people keep one set of credentials, the applications store none, and the policy that decides how someone signs in, from password rules to a second factor, is set once at the provider. Disabling an account at the provider stops it from starting a new session in any application that relies on it, which is what makes offboarding a single step rather than a list.

It also closes a hole that teams fall into when they build login on plain OAuth. RFC 6749 warns that in the implicit flow a public client has no way to tell which client an access token was issued to, so one that assumes only the resource owner can hand it a valid token is open to impersonation: an attacker holding a token issued to another application can present it and be taken for the person behind it. An ID token names the application it is for in its `aud` claim, and the application checks it.

Because it is a standard, the work transfers. The specification promises profile information in an interoperable manner, so the code that signs people in through one provider can sign them in through another, and with Discovery the application learns the provider's endpoints and signing keys from its issuer URL instead of from hand-copied settings.

Finally, OpenID Connect gives each person a stable identifier. The specification is explicit that the pair of `iss` (who issued the token) and `sub` (the subject identifier) is the only claim pair an application can rely on as a stable identifier for a person. The `sub` value is unique within the issuer, never reassigned and at most 255 ASCII characters. A provider may even issue a different `sub` to each application, a pairwise identifier, so that applications cannot correlate a person's activities without permission.

What it looks like in practice

Take a web application registered with an OpenID Provider at `https://server.example.com` under the client ID `s6BhdRkqt3`; the values below come from the specification's own examples. The application starts by reading the provider's configuration from the well-known address under the issuer:

```
GET /.well-known/openid-configuration HTTP/1.1
Host: server.example.com
```

The answer is a JSON document. A shortened version:

```
{
  "issuer": "https://server.example.com",
  "authorization_endpoint": "https://server.example.com/connect/authorize",
  "token_endpoint": "https://server.example.com/connect/token",
  "userinfo_endpoint": "https://server.example.com/connect/userinfo",
  "jwks_uri": "https://server.example.com/jwks.json",
  "scopes_supported": ["openid", "profile", "email", "address", "phone"],
  "id_token_signing_alg_values_supported": ["RS256", "ES256", "HS256"]
}
```

Discovery requires the `issuer` in that document to be identical to the issuer URL the application put in front of the well-known path, and the application stops if it differs. Then it sends the person's browser to the authorization endpoint with

the code flow, the `openid` scope and two random values it has stored against the browser session: `state` for the redirect and `nonce` for the ID token. A public client would add the PKCE parameters as well, which RFC 9700 makes mandatory for it; they are left out here to keep lines short.

```
GET /connect/authorize?response_type=code&client_id=s6BhdRkqt3
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
    &scope=openid%20profile%20email
    &state=af0ifjsldkj&nonce=n-0S6_WzA2Mj HTTP/1.1
Host: server.example.com
```

The provider signs the person in, sends the browser back with a code, and the application redeems the code at the token endpoint exactly as in plain OAuth. The difference is in the answer, which now carries an `id_token` next to the access token (shortened here):

```
{
  "access_token": "SlAV32hkKG",
  "token_type": "Bearer",
  "refresh_token": "8xLOxBtZp8",
  "expires_in": 3600,
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc..."
}
```

The ID token is three base64url parts separated by dots. The first decodes to `{"alg":"RS256","kid":"1e9gdk7"}` : the signing algorithm and the ID of the key in the provider’s published key set. The middle part holds the claims, and the specification’s example reads:

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "auth_time": 1311280969,
  "acr": "urn:mace:incommon:iap:silver"
}
```

Before the application trusts a word of it, it validates the token, and the specification lists the steps. The `iss` must match the provider's issuer exactly. The `aud` must contain the application's own client ID and no audience it does not trust. The signature must verify with the provider's keys from `jwks_uri`, using the algorithm in the header; when the token arrived straight from the token endpoint over TLS, as here, checking the server's TLS certificate may stand in for the signature check. The current time must be before `exp`. The `nonce` must equal the value the application sent. Only then does it sign the person in, keyed on `https://server.example.com` plus `24400320`, and use the access token to fetch email and profile claims from the UserInfo endpoint if it needs them.

A single provider for many systems is as much an organisational change as a technical one. One of this site's projects, access management across M365 and SSO, consolidated identities and access across M365, 1Password, Red Hat SSO and OKTA SSO, enforced least privilege, and standardised onboarding and offboarding so that access was revoked promptly when someone moved on.

Where it goes wrong

The email address is used as the account key. It is the obvious choice and the wrong one. The specification says a provider may reuse an email address for a different person over time, and that `email`, `phone_number`, `preferred_username` and `name` must not be used as unique identifiers. An application that looks people up by email can hand one person's account to the next owner of that address. Store `iss` and `sub` together as the key, keep the email as an attribute, and before relying on an address for anything, check that `email_verified` is true.

An access token is taken as proof of who signed in. Some applications receive an access token, call an API with it, and treat the answer as a login. The token may have been issued to a different application altogether, which is the impersonation RFC 6749 warns about. Sign people in on the ID token, and only after confirming that its `aud` names your own client ID.

The ID token is decoded but never checked. Reading the claims out of a token takes a base64url decode and no key at all, and that ease is the trap. A token whose signature, `iss`, `aud` or `exp` goes unverified can be forged, belong to another application or have expired. Use a maintained library, give it the issuer and client ID, and let it run every check in the specification's list rather than picking the convenient ones.

The nonce is missing, fixed, or never compared. Without it, an ID token captured from one sign-in can be replayed into another. RFC 9700 adds that the nonce must be specific to one sign-in and bound to the browser that started it. The specification suggests one way to do that: keep a cryptographically random value in an HttpOnly session cookie, send its hash as the `nonce`, and compare the two when the ID token comes back.

Signing keys are cached for good. Providers rotate keys, and an application holding yesterday's key set rejects every token signed with the new one. The specification expects the provider to publish keys at `jwks_uri` and mark each token with a `kid`, and the application to fetch the key set again when it meets a `kid` it does not know. Cache the keys, but treat an unfamiliar `kid` as a signal to refresh rather than a reason to fail the login.

Two tenants on one host are treated as one issuer. A provider can run several issuers on one host, and the path is part of the issuer identifier: the subject `1234` at `https://example.com` and the subject `1234` at `https://example.com/sales` are different people. Store the full issuer string, compare it exactly, and never trim it to a host name.

Further reading

- OpenID Connect Core 1.0 — the ID token, the three flows, the standard claims and the validation steps, in the edition incorporating errata set 2 of December 2023.
- OpenID Connect Discovery 1.0 — the well-known configuration document and the rule that its issuer must match the issuer URL it was fetched under.
- RFC 6749: The OAuth 2.0 Authorization Framework — the framework OpenID Connect builds on, with the warning in section 10.16 about using access tokens to identify people.
- RFC 7519: JSON Web Token (JWT) — the token format every ID token uses.
- RFC 9700: Best Current Practice for OAuth 2.0 Security — current guidance on PKCE, the nonce and the redirect, which applies to OpenID Connect as much as to OAuth.



Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is OpenID Connect?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/openid-connect/