pykrete

01 Getting started 2

02 Integration 20

03 Administration 79

01

Getting started

Overview 3

Running pykrete 8

Configuration 12

Overview

pykrete is a self-hosted identity provider. Its core is an OpenID Connect provider and OAuth 2.0 authorization server. Around that it adds passkey sign-in, an account dashboard, self-service organizations, applications and APIs, a read-only LDAP server with per-app passwords, end-to-end encryption keys for apps, and HTTP hooks for custom logic.

Generated codes and secrets carry a `DELTA-` prefix (`DELTA-CODE-...`, `DELTA-CLIENT-SECRET-...`, `DELTA-API-KEY-...`), named after the first deployment, delta.rocks.

Features

Area	What you get	Details
OpenID Connect / OAuth 2.0	Discovery, authorization code with PKCE (`S256`, `plain`), refresh tokens, device authorization (RFC 8628, partial), userinfo, JWKS, RP-initiated logout, introspection (RFC 7662), revocation (RFC 7009), dynamic client registration (RFC 7591). The implicit, hybrid and `client_credentials` grants are not supported.	OpenID Connect
Sign-in	Username plus password, or passkeys (WebAuthn with user verification required; discoverable passkeys allow sign-in without typing a username). Every sign-in session starts with a small proof-of-work challenge solved by the browser. There is no public sign-up: an administrator creates accounts. Optional second factor after a password: authenticator app (TOTP), YubiKey (Yubico OTP, checked locally or with YubiCloud) or recovery codes.	Two-factor authentication
Scopes, roles, consent	Built-in OIDC scopes, per-API scopes, roles that grant scopes, stored consent, pairwise or public subject identifiers.	Scopes and claims
Account dashboard	`/account`: profile, emails, sign-in methods, sessions, pending sign-ins, organizations, applications, access tokens, app passwords, API keys, E2EE keys, roles, and the clients and audiences you own.	Account dashboard
Self-service	Users and organizations own clients (applications), audiences (APIs) and roles, and manage them from the dashboard or the `manage` GraphQL API.	Self-service
Organizations	Members, organization roles, invitations, per-application access, an active organization in tokens.	Organizations
HTTP hooks	The backend calls `identity`, `claims` and the email rendering hooks implemented by the frontend.	Hooks
LDAP	Optional LDAPS listener with a read-only directory per application and per-app passwords.	LDAP
App encryption keys	Per-user key material unlocked in the browser by a password or a passkey (PRF); apps receive keys through `app_key:<id>` scopes.	App keys
Backend API	Clerk-style REST subset under `/api/v1`, API keys.	Backend API
Admin API	GraphQL at `/graphql`, an in-browser console at `/graphiql`, "admin mode" in the dashboard.	Administration

SAML	Not supported.	SAML and proxy auth
Forward / proxy auth	Not supported.	SAML and proxy auth
Email delivery	Built-in SMTP for verification and invitation emails. The frontend renders the content, so templates can be customized.	Configuration, Hooks

Components

Component	Runs as	Role
Backend	`pykrete` binary	Listens on `127.0.0.1:3000` by default (plain HTTP). Serves the OIDC/OAuth endpoints, JWKS, `/graphql`, `/api/v1` and the `/control` WebSocket that drives sign-in, consent and logout. Proxies every other path to freshfront. Calls hooks. Runs the optional LDAP listener in the same process.
freshfront	Deno server from `freshfront/` (Preact SSR)	Renders all HTML: `/authorize`, `/code`, `/oauth/logout`, `/v2/logout`, `/invite`, `/add-passkey`, `/account/...`, `/graphiql`, `/docs/...`. Implements the hooks under `/hooks/...`. Calls the backend over HTTP for GraphQL and token exchange. It is an example frontend you are expected to adapt (`freshfront/README.md`).
PostgreSQL	database server	All state: users, credentials, sessions, clients, audiences, tokens, organizations, API keys, images.
PKCS#11 token	SoftHSM2, a hardware HSM or a smartcard	The token signing keys and the refresh-token encryption key are generated and used inside the token; the backend talks to it through a PKCS#11 module. Required.
LDAP listener	part of the `pykrete` process	Optional LDAPS server, enabled by the `ldap` config section.

Who handles which path

The backend is the only entry point. Put the TLS terminator in front of it and nothing else.

Handled by the backend

`/.well-known/openid-configuration`, `/.well-known/jwks.json`, `/jwks`, `/oauth/token`, `/oauth/introspect`, `/oauth/revoke`, `/device-auth`, `/register`, `/register/{client_id}`, `/userinfo`, `/verify`, `/control` (WebSocket), `/graphql`, `/openapi.json`, `/api/v1/...`, `/images/{id}`, `/resources/...`, `/proxy-auth`, `/api/v2/jobs/users-exports` (not available yet).

Proxied to freshfront

Everything else, including `/authorize`, `/oauth/logout`, `/code`, `/account/...`, `/graphiql` and `/docs/...`. Request and response bodies are streamed; WebSocket upgrades are passed through.

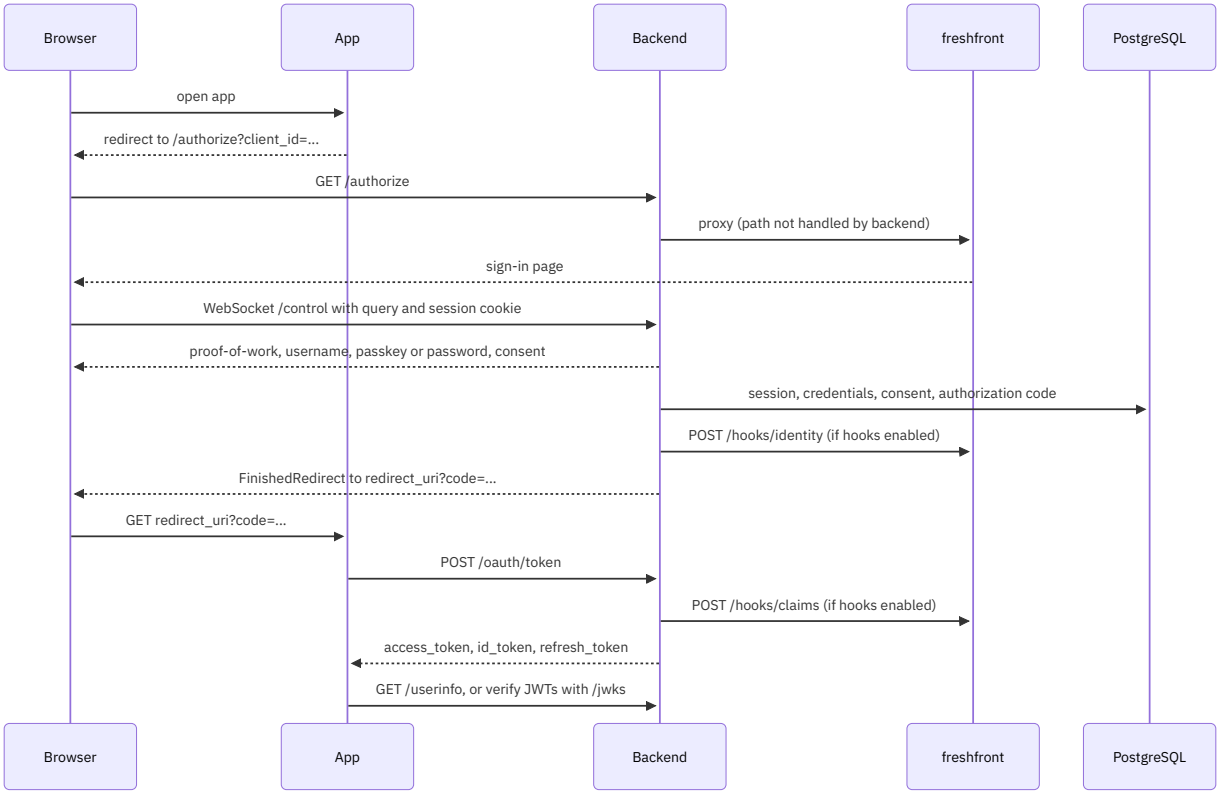
Blocked at the backend

`/hooks` and `/hooks/...` return 404 at the backend (checked after path normalization, so `/./hooks` or `/x/../hooks` do not slip through). The backend itself POSTs hook calls to freshfront over the same upstream connection, with `Authorization: Bearer <hooks_secret>`.

`GET /graphql` redirects to `/graphiql`. Responses generated by the backend get a strict `Content-Security-Policy: default-src 'none'; ...`; proxied freshfront responses keep freshfront's own CSP.

Request flow

A relying party signing a user in with the authorization code flow:



1. The app sends the browser to `/authorize`. The backend has no route for it, so freshfront renders the page.
2. The page opens the `/control` WebSocket on the backend and sends the authorization request plus the session cookie. All sign-in logic runs on that socket: proof-of-work, identification (username or discoverable passkey), credential check, consent, organization choice. The backend sends each step as a message and the page renders the matching form.
3. When done, the backend tells the page to navigate to the app's `redirect_uri` with the code.
4. The app exchanges the code at `/oauth/token`. The `claims` hook may add claims at this point.
5. The app validates the ID token against `/jwks` (or `/.well-known/jwks.json`) and may call `/userinfo`.

Dashboard pages work the same way: `/account` is itself an OIDC client (see startup). freshfront signs the user in with PKCE, keeps the access token in an `HttpOnly` cookie scoped to `/account`, and renders pages by calling the backend's `/graphql` with it.

Glossary

Issuer

`https://<domain>/` (or `http://` without `tls_terminated`), with a trailing slash; on this server `https://auth.example.com/`. All endpoint URLs in discovery are built from the `domain` config key, not from the request's `Host` header.

User

A person. Id `U` plus 32 alphanumerics. Has a username history (sign-in names), a profile (name, picture), emails (verified or not), credentials, public and private metadata, and roles.

Credential

What a user proves at sign-in. Stored as "challenges": `password`, `webauthn` (passkey) and `password_pending` (a one-time registration code that the user turns into a password).

Session

Browser sign-in state. The browser keeps a cookie encrypted with a key derived from `session_secret`; it holds a random session id that the database maps to the user and the credentials already passed. Listed at `/account/sessions`.

Client (application)

An OAuth/OIDC client. Id `C` plus 32 alphanumerics. Public or confidential, with optional secret, PKCE, redirect URLs, post-logout redirect URLs, allowed grant types, allowed and required scopes, subject type and organization usage. Owned by a user, an organization, or nobody (system clients).

Audience (API)

A resource server that access tokens are issued for. Id `A` plus 32 alphanumerics, plus a unique identifier (usually a URL). Defines its own scopes and the access token signature algorithm (RS256 or ES256). The server itself is the "self" audience; its identifier is the issuer and it carries the `admin:graphql`, `account` and `org:*` scopes.

Scope

A permission requested in `scope`. Built-in scopes are `openid`, `offline_access`, `profile`, `email` and `organizations`; audiences add their own; `app_key:<id>` scopes request encryption keys.

Role

A named set of audience scopes. Granted globally (for example `role:admin`) or inside an organization (`role:org-admin`, `role:org-member`, or roles owned by the organization). A token only gets an audience scope if one of the user's roles grants it (the self audience's `account` scope is open to every user) and the client is allowed to request it.

Organization

A group of users. Id `O` plus 32 alphanumerics, plus a unique slug. Members hold organization roles. Clients opt in to organizations, and a token can carry one active organization.

Consent (grant)

A user's stored approval for a client to use a set of scopes. Asked for on first use and listed at `/account/applications`.

Access token

A JWT (RS256 or ES256, chosen by the audience) with `iss`, `sub`, `aud`, `azp`, `scope`, `iat`, `exp`. Valid for one hour. Also recorded in the database, which is what `/userinfo`, `/graphql`, introspection and revocation check.

ID token

A JWT for the client (`aud` is the client id), issued when `openid` is requested. Signed with RS256 or ES256 as configured on the client.

Refresh token

Issued with `offline_access`. An encrypted JWE that rotates on every use; replaying an old one revokes the whole chain.

Hook

An HTTP callback from the backend to freshfront (`POST /hooks/<name>`) for custom claims, email delivery and similar logic.

Running pykrete

A deployment has five parts: PostgreSQL, a PKCS#11 token holding the signing keys, the `pykrete` binary (the backend), the freshfront Deno server, and a TLS reverse proxy in front of the backend. This page goes through them in order. Every flag and key used here is described in Configuration.

Requirements

- Linux.
- PostgreSQL.
- immigrant to apply the database migrations that ship with a release.
- A PKCS#11 module: SoftHSM2 (`softhsm2-util`) for a software token, or the module of your HSM or smartcard.
- Deno 2 to run freshfront.
- A TLS reverse proxy, for example nginx.

Database

Create a database and a role for pykrete:

```
sudo -u postgres createuser pykrete
sudo -u postgres createdb -O pykrete pykrete
export DATABASE_URL=postgres://pykrete@%2Frun%2Fpostgresql/pykrete
# or over TCP: postgres://pykrete:<password>@127.0.0.1/pykrete
```

Applying migrations

The backend does **not** migrate the database on start. Each release ships its migrations in a `migrations/` directory; apply the pending ones before starting a new release (immigrant finds `migrations/` from the current directory):

```
cd /path/to/release
immigrant migrate --dry-run apply   # run everything in a transaction, then roll back
immigrant migrate apply
```

Applied migrations are recorded in the `__immigrant_migrations` table. `apply` stops if an already applied migration differs from the one on disk.

Keystore

The backend refuses to start without a PKCS#11 token. For a software token, initialize SoftHSM2 once and note the slot number it prints; the details and the HSM/smartcard variant are in Keys and secrets.

```
export SOFTHSM2_CONF=/var/lib/pykrete/softhsm2.conf
mkdir -p /var/lib/pykrete/tokens
echo "directories.tokendir = /var/lib/pykrete/tokens" > "$SOFTHSM2_CONF"
softhsm2-util --init-token --free --label pykrete --pin <user-pin> --so-pin <so-pin>
softhsm2-util --show-slots
```

Running

Backend

```
export DATABASE_URL=postgres://pykrete@%2Frun%2Fpostgresql/pykrete
export SOFTHSM2_CONF=/var/lib/pykrete/softhsm2.conf
export PYKRETE_PKCS11_MODULE=/usr/lib/softhsm/libsofthsm2.so
export PYKRETE_PKCS11_SLOT=<slot from softhsm2-util --show-slots>
export PYKRETE_PKCS11_PIN=<user-pin>
export RUST_LOG=info

pykrete --config /etc/pykrete/config.json
```

The module path depends on your distribution. The backend listens on `127.0.0.1:3000` unless `listen` is set in the config file. Startup takes a moment because the session key is derived with 600000 PBKDF2 iterations. It exits if the config file is missing or invalid, the PKCS#11 variables are missing or the slot does not exist, or the database is unreachable.

freshfront

```
cd freshfront
FRESHFRONT_SOCKET=/run/pykrete/freshfront.sock \
AUTH_BACKEND=http://127.0.0.1:3000 \
HOOKS_SECRET=<hooks-secret> \
deno task start
```

Point the backend at it with `"frontend": {"upstream": "unix:/run/pykrete/freshfront.sock", ...}` in the config file. `HOOKS_SECRET` must equal `frontend.hooks_secret`.

> **IMPORTANT**
>
> Use `FRESHFRONT_SOCKET` in production. Without it, freshfront listens on TCP port 8000 on all interfaces; if you run it that way, make sure port 8000 is not reachable from outside, so that all traffic goes through the backend.

Reverse proxy and TLS

The backend speaks plain HTTP and never terminates TLS. Put a TLS terminator in front of it and:

- Pass **every** path to `127.0.0.1:3000` (or the configured `listen` address). The backend decides what it handles and proxies the rest to freshfront; routing pages straight to freshfront bypasses that.
- Set `X-Real-IP` to the client's address, replacing any value the client sent. The backend trusts this header as is and shows the address to users approving backchannel (CIBA) requests. Without it the address shows as unknown.
- Allow WebSocket upgrades. Sign-in, consent and logout run over `/control`.
- Allow request bodies of at least 2 MiB plus 64 KiB on `/graphql` (picture uploads).
- Set `domain` in the config file to the public host name (with the port, if it is not the default) and `tls_terminated` to `true`. Issuer, endpoint URLs, the WebAuthn origin and the internal clients' redirect URIs are built from these two keys, not from the `Host` header. freshfront marks its cookies `Secure` when the issuer is `https://`.
- Expose LDAP (if enabled) as its own TCP port. pykrete terminates TLS for LDAP itself, so don't put it behind the HTTP proxy.

nginx:

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

server {
    listen 443 ssl;
    server_name auth.example.com;
    ssl_certificate     /etc/ssl/auth.example.com/fullchain.pem;
    ssl_certificate_key /etc/ssl/auth.example.com/privkey.pem;
    client_max_body_size 4m;

    location / {
        proxy_pass http://127.0.0.1:3000;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_read_timeout 300s;
    }
}
```

Check it from outside:

```
curl -s https://auth.example.com/.well-known/openid-configuration | jq .issuer
# "https://auth.example.com/"
```

What happens on startup

Each start runs these steps, in this order. All of them are idempotent.

1. Read and parse the config file, derive the session key from `session_secret`.
2. Connect to PostgreSQL, open the PKCS#11 token.
3. Signing keys: delete the public keys in the `jwk_keys` table that have no `authority`, generate `oidc-token-signing-rs256`, `oidc-token-signing-es256` and `oidc-refresh-encrypt-es256` in the token if they don't exist yet, and store their public JWKs in `jwk_keys` again (served at `/jwks`).
4. Self audience `A_____SELF`: identifier set to the issuer (`https://auth.example.com/`), name set to the domain. Scopes `admin:graphql` ("GraphQL admin") and `account` ("Account").
5. Role `role:admin` ("Admin"), which grants `admin:graphql` on the self audience.
6. Organization permissions: scopes `org:manage`, `org:members`, `org:roles`, `org:clients`, `org:audiences`, `org:access` and `org:api-keys` on the self audience; roles `role:org-admin` (all of them) and `role:org-member` (none), both assignable inside organizations.
7. Internal clients, both public with PKCE, grant type `authorization_code`, marked verified. Their redirect URLs are rewritten on every start, so changing `domain` or `tls_terminated` updates them.
    1. `C_____GQL_CLIENT` "GraphiQL": redirect `https://auth.example.com/graphiql`, required scope `admin:graphql`.
    2. `C_____ACCOUNT` "Account": redirect `https://auth.example.com/account/callback`, post-logout redirect to the issuer, required scope `account`, optional scope `admin:graphql`.
8. Import pictures stored in legacy columns into the `images` table (failures are only logged), then start an hourly image garbage collector.
9. Bind the LDAP listener if `ldap` is configured, then listen for HTTP on `listen` (default `127.0.0.1:3000`).

Nothing else is created: there are no users, organizations or third-party clients after the first start.

The first administrator

There is no setup wizard and no CLI command for the first account. Users are created through the admin GraphQL API, which needs an admin, so the first one has to be inserted with SQL. Run this after the backend has started once (it creates `role:admin`):

```
uid="U$(openssl rand -hex 16)"
psql "$DATABASE_URL" <<SQL
BEGIN;
INSERT INTO userdata_users (user_id) VALUES ('$uid');
INSERT INTO userdata_usernames (user_id, username, normalized_username)
  VALUES ('$uid', 'admin', 'admin');
INSERT INTO userdata_profiles (user_id, name) VALUES ('$uid', 'Administrator');
INSERT INTO oauth_user_granted_roles (user_id, user_role_id) VALUES ('$uid', 'role:admin');
COMMIT;
SQL
```

User ids are `U` followed by 32 alphanumerics (a database check enforces the prefix and length). Then:

1. Open https://auth.example.com/account and sign in as `admin`. The new account has no credentials yet, so no password is asked for.
2. Click "Add passkey" in the account header (`/add-passkey`) right away. If you want a password instead, enter admin mode, open `/graphiql` and issue yourself a registration code (below); the next sign-in asks for the code and a new password.
3. Click "Enter admin mode". The dashboard signs you in again with `prompt=login` and requests `admin:graphql`. The admin GraphQL console is at `/graphiql`.

Further users, from `/graphiql` (needs `admin:graphql`). Create the user, then issue a registration code for them right away:

```
mutation {
  userdata {
    createUser(username: "alice", name: "Alice", email: "alice@example.com") { userId }
  }
}
```

```
mutation {
  userdata {
    user(userId: "U...") { issueRegistrationCode }
  }
}
```

Hand the code to the user: on first sign-in they enter it and choose a password, and can add passkeys afterwards. `issueRegistrationCode` refuses users that already have credentials. To make someone an admin:

```
mutation {
  userdata {
    user(userId: "U...") { grantRole(userRoleId: "role:admin") { userRoleId name } }
  }
}
```

More admin tasks are in Administration.

Configuration

The backend takes a few command-line flags, a JSON config file and environment variables; its keys live in a PKCS#11 token. freshfront is configured only through environment variables. For how the pieces fit together see Running pykrete.

Command line

```
pykrete --database-url <URL> --config <PATH>
```

Flag	Type	Default	Required	What it does
`--database-url <URL>`	string	-	yes (or `DATABASE_URL`)	PostgreSQL connection URL, for example `postgres://pykrete:<password>@127.0.0.1/pykrete` or `postgres://pykrete@%2Frun%2Fpostgresql/pykrete` for a unix socket.
`--config <PATH>`	path	-	yes	JSON config file, see Config file.

`--help` prints the same list.

Config file

A JSON object. Unknown keys are ignored silently, so check spelling.

Key	Type	Default	Required	What it does
`domain`	string	-	yes	Public host name, plus `:port` if it is not the default, for example `auth.example.com` or `localhost:3000`. The issuer is `http[s]://<domain>/` and every URL in discovery, the WebAuthn relying party id (the host part) and origin, and the internal clients' redirect URIs are built from it.
`tls_terminated`	boolean	`false`	no	Generate `https://` URLs. Set it to `true` whenever users reach the server over HTTPS; the backend itself never speaks TLS. Also makes freshfront mark its cookies `Secure` (it reads the scheme from the issuer).
`listen`	socket address	`127.0.0.1:3000`	no	Address and port of the HTTP listener, for example `127.0.0.1:3001`. Keep it on a loopback or private address: the backend trusts the `X-Real-IP` header, so only the reverse proxy may reach it. freshfront's `AUTH_BACKEND` must point here.
`session_secret`	string	-	yes	Secret the browser session key is derived from (PBKDF2-HMAC-SHA256, 600000 iterations, fixed salt). The key encrypts the session cookie with AES-256-GCM-SIV. Changing it signs every browser out. Use a long random value, for example `openssl rand -base64 48`.

`otp_secret`	string	absent	for two-factor authentication	Secret that authenticator app and YubiKey secrets are encrypted with (AES-256-GCM-SIV, key derived with HMAC-SHA256). Without it, users can't set up a second factor. Changing or removing it makes existing authenticator apps and YubiKeys unusable; affected users can still sign in with a recovery code or a passkey. Keep it separate from `session_secret` and back it up with the database. Use a long random value.
`yubicloud`	object	absent	no	Lets users add a YubiKey by touching it once, checked with Yubico's YubiCloud service, see YubiCloud. Doesn't need `otp_secret`.
`frontend`	object	absent	in practice, yes	Where freshfront runs and how hooks authenticate, see Frontend. Without it, every path the backend does not handle answers `502 Frontend unavailable`, and hooks are off.
`email`	object	absent	for production	SMTP delivery, see Email. Without it, emails are only written to the log.
`ldap`	object	absent	no	Enables the LDAPS listener, see LDAP.

YubiCloud

Most YubiKeys ship with a Yubico OTP credential in slot 1 whose secret only Yubico knows. With `yubicloud` configured, users register such a key by touching it once, and every code is checked with Yubico's validation service (protocol 2.0). Without it, users can still program their own slot and have the server check codes itself, which needs `otp_secret`.

Key	Type	Default	Required	What it does
`yubicloud.client_id`	string	-	yes	Client ID from https://upgrade.yubico.com/getapikey/.
`yubicloud.api_key`	string	-	yes	API key from the same page (base64). Requests and responses are signed with it (HMAC-SHA1); a response with a wrong signature, nonce or code is treated as the service being unavailable.
`yubicloud.url`	URL	`\https://api.yubico.com/wsapi/2.0/verify`	no	Validation endpoint, for a self-hosted validation server. Must be `https`; private addresses are allowed.

Things to know before turning it on:

- Yubico sees the key's public ID and the time of every sign-in and registration that uses it, and the backend's IP address.
- When the service can't be reached, users with only a YubiCloud key can't pass two-factor authentication; they can use a recovery code or a passkey. The sign-in page tells them the service is unavailable, and the cause is logged.
- The backend needs outbound HTTPS to the validation endpoint. Requests time out after 10 seconds.
- A code accepted by YubiCloud is spent for every site using YubiCloud, so a code typed into another site

can't be replayed here.

Frontend

Key	Type	Default	Required	What it does
`frontend.upstream`	string	-	yes	`http://<host>:<port>` (plain HTTP only, no path or query; the port defaults to 80) or `unix:<path>` for a unix socket, for example `unix:/run/pykrete/freshfront.sock`. Pages and hook calls both go here.
`frontend.hooks_secret`	string	absent	no	Bearer token sent as `Authorization: Bearer <hooks_secret>` with every hook call. When absent the backend calls no hooks at all, and so can't render email (see Email). Must equal freshfront's `HOOKS_SECRET`.

> **⚠ WARNING**
> A hook that fails (wrong secret, non-2xx answer, invalid JSON, more than 5 seconds) fails the operation that called it: sign-in for `identity`, token issuance for `claims`, sending for the email hooks. If you set `hooks_secret`, make sure freshfront has the same `HOOKS_SECRET`. See Hooks.

Email

pykrete sends two emails: email address verification links, and organization invitations addressed to an email address. The backend sends them over SMTP; freshfront renders the content through the email hooks.

Key	Type	Default	Required	What it does
`email.from`	string	-	yes	Sender, a bare address or `Name <address>`, for example `pykrete <noreply@example.com>`. Its domain is also used in `Message-ID`.
`email.smtp`	URL	-	yes	SMTP server and credentials. Always give the port. Percent-encode special characters in the user name and password (`@` becomes `%40`).

`email.smtp` forms:

URL	Connection
`smtps://user:password@mail.example.com:465`	TLS from the first byte
`smtp://user:password@mail.example.com:587?tls=required`	STARTTLS, fails if the server doesn't offer it
`smtp://user:password@mail.example.com:587?tls=opportunistic`	STARTTLS when offered, otherwise plain text
`smtp://127.0.0.1:25`	Plain text, for a relay on the same host only

The plain-text forms send the password unencrypted. Server certificates are checked against a bundled copy of the Mozilla root store, not the system store.

Behavior:

- Without `email`, nothing is sent. The backend logs each email at `info` level ("email delivery is not configured"), including its link: the rendered subject and text when hooks are enabled, the raw data otherwise. Use this only for development.
- With `email`, hooks must be enabled, because the content comes from freshfront. Otherwise every email fails with "sending email requires frontend hooks to render it".
- Email is sent while the request waits. If rendering or SMTP fails, the verification link or invitation is discarded and the user sees an error. There is no queue and no retry.

LDAP

Key	Type	Default	Required	What it does
`ldap.listen`	socket address	-	yes	Address and port to listen on, for example `0.0.0.0:636`.
`ldap.base_dn`	string	-	yes	Directory root, a non-empty DN such as `dc=example,dc=com`.
`ldap.certificate`	path	-	yes	PEM certificate chain for the listener.
`ldap.private_key`	path	-	yes	PEM private key for that certificate.

The listener speaks LDAPS only: TLS starts with the first byte, there is no plain LDAP and no StartTLS. Add, modify, delete, rename and compare requests are refused. Directory layout and binds are in LDAP.

Keys that are ignored

Older config files contain keys that the current backend does not read:

Key	Current equivalent
`port`	`listen`
`ldap_port`	`ldap.listen`
`rs256_key`	the `oidc-token-signing-rs256` key in the PKCS#11 token
`idp_key`, `idp_cert`, `saml_issuer`	none; SAML is not supported

Environment variables

Backend

Variable	Required	What it does
`DATABASE_URL`	yes, unless `--database-url`	PostgreSQL URL.
`PYKRETE_PKCS11_MODULE`	yes	Path to the PKCS#11 module, for example `libsofthsm2.so` or `opensc-pkcs11.so`.
`PYKRETE_PKCS11_SLOT`	yes	Slot id, decimal or `0x`-prefixed hex.

PYKRETE_PKCS11_PIN	for SoftHSM2 and most tokens	User PIN. If unset, the backend calls C_Login without a PIN, which only works with tokens that authenticate by other means.
PYKRETE_PKCS11_KEY_PREFIX	no	Prepended to every key id in the token (CKA_ID and CKA_LABEL), so several deployments can share a token.
RUST_LOG	no	Log filter (tracing EnvFilter syntax), for example info or pykrete=debug,info. Unset means errors only.
SOFTHSM2_CONF	for SoftHSM2	Read by SoftHSM2 itself, not by pykrete: path to its config file.

freshfront

Variable	Default	What it does
FRESHFRONT_SOCKET	unset: TCP port 8000 on all interfaces	Listen on a unix socket at this path instead. A stale socket file is removed at start.
AUTH_BACKEND	http://127.0.0.1:3000	Base URL of the backend for server-side calls: discovery, code exchange for the dashboard, GraphQL.
HOOKS_SECRET	unset: every /hooks/... request gets 404	Must equal the backend's frontend.hooks_secret.
NETBIRD_CLIENT_IDS, NETBIRD_AUTH_AUDIENCE, NETBIRD_DOMAIN_SUFFIX	unset	Used by the bundled claims hook for NetBird, see Organizations.

Fixed values

These are not configurable:

What	Value
Database pool	at least 1 idle connection, 400 ms connection timeout
Access token and ID token lifetime	3600 s (the database keeps access tokens 120 s longer for clock skew)
Authorization code lifetime	600 s
Refresh token chain lifetime	30 days
Device code	1800 s; poll interval 5 s
Backchannel (CIBA) request	600 s by default, requested_expiry 60 to 3600 s; poll interval 5 s; at most 5 pending per client and user; binding_message up to 128 characters
Email verification link	24 h, resend allowed after 1 minute
Organization invitation (default)	7 days
/graphql request body	2 MiB plus 64 KiB
Hook call	5 s timeout, 1 MiB response limit
Email	15 s SMTP timeout, 20 s per message

Second factor	5 wrong codes per sign-in, 1 s delay after each; TOTP accepts the previous and next 30 s step; authenticator app setup valid 15 minutes; 10 recovery codes; YubiCloud requests time out after 10 s
LDAP	1000 entries per search, 128 KiB per message, 10 s TLS handshake, 300 s idle

Example

/etc/pykrete/config.json:

```
{
  "domain": "auth.example.com",
  "tls_terminated": true,
  "session_secret": "<session-secret>",
  "otp_secret": "<otp-secret>",
  "yubicloud": {
    "client_id": "<client-id>",
    "api_key": "<api-key>"
  },
  "frontend": {
    "upstream": "unix:/run/pykrete/freshfront.sock",
    "hooks_secret": "<hooks-secret>"
  },
  "email": {
    "from": "pykrete <noreply@example.com>",
    "smtp": "smtps://noreply%40example.com:<smtp-password>@mail.example.com:465"
  },
  "ldap": {
    "listen": "0.0.0.0:636",
    "base_dn": "dc=example,dc=com",
    "certificate": "/etc/pykrete/ldap/fullchain.pem",
    "private_key": "/etc/pykrete/ldap/privkey.pem"
  }
}
```

Backend environment and command:

```
DATABASE_URL=postgres://pykrete@%2Frun%2Fpostgresql/pykrete
SOFTHSM2_CONF=/var/lib/pykrete/softhsm2.conf
PYKRETE_PKCS11_MODULE=/usr/lib/softhsm/libsofthsm2.so
PYKRETE_PKCS11_SLOT=<slot>
PYKRETE_PKCS11_PIN=<user-pin>
RUST_LOG=info

pykrete --config /etc/pykrete/config.json
```

freshfront environment:

```
FRESHFRONT_SOCKET=/run/pykrete/freshfront.sock
AUTH_BACKEND=http://127.0.0.1:3000
HOOKS_SECRET=<hooks-secret>
```

The backend user must be able to connect to the socket that freshfront creates.

Keys and secrets

Keys in the PKCS#11 token

All asymmetric keys are generated and used inside the PKCS#11 token; the backend only reads their public parts. They are created on startup when missing, as persistent token objects whose `CKA_ID` and `CKA_LABEL` are `<PYKRETE_PKCS11_KEY_PREFIX><key id>`:

Key id	Type	Used for
`oidc-token-signing-rs256`	RSA 2048	RS256 signatures on ID tokens and access tokens
`oidc-token-signing-es256`	EC P-256	ES256 signatures, when a client (ID token) or audience (access token) selects ES256
`oidc-refresh-encrypt-es256`	EC P-256	Refresh tokens: JWE `ECDH-ES + A256GCM`. Encryption uses the public key in software; decryption derives the shared secret in the token (`CKM_ECDH1_DERIVE`).

The token has to support `CKM_RSA_PKCS_KEY_PAIR_GEN`, `CKM_EC_KEY_PAIR_GEN` (P-256), `CKM_SHA256_RSA_PKCS`, `CKM_ECDSA` and `CKM_ECDH1_DERIVE`.

On every start the public keys are copied into the `jwk_keys` table (rows without an `authority` are deleted first) and published at `/jwks` and `/.well-known/jwks.json`. The JWKS therefore lists all three keys; the refresh-token key has `"use": "enc"`.

SoftHSM2

```
export SOFTHSM2_CONF=/var/lib/pykrete/softhsm2.conf
mkdir -p /var/lib/pykrete/tokens
echo "directories.tokendir = /var/lib/pykrete/tokens" > "$SOFTHSM2_CONF"
softhsm2-util --init-token --free --label pykrete --pin <user-pin> --so-pin <so-pin>
softhsm2-util --show-slots
```

`--init-token` prints the slot the token was reassigned to; that number is `PYKRETE_PKCS11_SLOT`. The backend needs read and write access to the token directory (it generates keys there on first start).

Hardware tokens

Any PKCS#11 module works if it supports the mechanisms above. For example, an OpenSC-supported smartcard: module `opensc-pkcs11.so` from OpenSC, slot `0`, a key prefix of `pykrete-`, and the card's PIN.

Rotation and backups

Key rotation is not supported. Don't remove the keys from the token: the backend generates new ones at the next start, tokens signed with the old keys stop validating at relying parties, and refresh tokens can no longer be decrypted.

Back up the token storage (for SoftHSM2, the `tokendir`) together with the database.

Other secrets

Secret	Where
Session cookie key	derived from `session_secret` at start, not stored

Hooks secret	config file and freshfront's environment
SMTP password	config file (`email.smtp`)
OTP secret	config file (`otp_secret`)
YubiCloud API key	config file (`yubicloud.api_key`)
Authenticator app and self-programmed YubiKey secrets	`userdata_challenges`, encrypted with `otp_secret`
YubiCloud keys	`userdata_challenges`, public ID only (the secret stays with Yubico)
Recovery codes	`userdata_challenges` (SHA-256)
Client secrets (`DELTA-CLIENT-SECRET-...`)	`oauth_client_secrets`
API keys (`DELTA-API-KEY-...`)	`oauth_api_keys`
App passwords (LDAP)	`userdata_app_passwords`
Passwords	`userdata_challenges`
Registration codes	`userdata_challenges` (`password_pending`)
Email verification and invitation codes	`userdata_email_verifications`, `orgs_invitations`
LDAP TLS key	the file in `ldap.private_key`

Treat database dumps and backups as secret.

02

Integration

OpenID Connect and OAuth	21
Scopes and claims	35
Organizations	42
HTTP hooks	52
LDAP	61
App encryption keys (E2E)	68
SAML and proxy authentication	75
Backend API, API keys, introspection	76

OpenID Connect and OAuth

pykrete is an OpenID Connect provider. This page is for developers of relying parties (apps that sign users in) and of APIs that accept pykrete access tokens. Scopes, audiences, consent and the exact claim set are covered in Scopes and claims.

All examples use `https://auth.example.com` as the server. The issuer is that origin with a trailing slash: `https://auth.example.com/`. It is derived from the configured domain and the https setting, see Configuration.

Discovery and endpoints

Discovery document: `https://auth.example.com/.well-known/openid-configuration`.

```
curl -s https://auth.example.com/.well-known/openid-configuration
```

Purpose	Path	Method	Notes
Discovery	`/.well-known/openid-configuration`	GET	CORS open to any origin
JWKS	`/jwks` (the advertised `jwks_uri`), also `/.well-known/jwks.json`	GET	Public signing keys
Authorization	`/authorize`	GET	Browser page, see Authorization request
Token	`/oauth/token`	POST	Form (`application/x-www-form-urlencoded`) or JSON body
Userinfo	`/userinfo`	GET	`Authorization: Bearer <access token>`
Device authorization	`/device-auth`	POST	RFC 8628, see Device authorization grant
Device verification page	`/code`	GET	Where users type the device code
Backchannel authentication	`/oauth/bc-authorize`	POST	CIBA, poll mode, see Backchannel authentication (CIBA)
Backchannel approval page	`/approve/<id>`	GET	Where users approve a backchannel request
End session	`/oauth/logout` (alias `/v2/logout`)	GET, POST	RP-initiated logout
Introspection	`/oauth/introspect`	POST	Confidential clients only, see Backend API
Revocation	`/oauth/revoke`	POST	Confidential clients only, see Backend API
Dynamic registration	`/register`, `/register/<client_id>`	POST; GET, PUT, DELETE	RFC 7591 / 7592

Supported response type: `code` only, with the response in the query string. Tokens are signed with RS256 or ES256, see Signing and keys.

Registering a client

A client (relying party) has a client id of the form `C` + 32 letters and digits, for example `C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH`. There are three ways to get one:

Self-service

Any signed-in user can create clients under **Account → OIDC → Clients** (`/account/clients`), owned by themselves or by an organization where they hold the `org:clients` permission. See Self-service.

Admin GraphQL

Administrators can create and edit every client through GraphQL. See Admin access.

Dynamic registration

`POST /register` (RFC 7591), see Dynamic client registration.

Clients created on the account pages start with: PKCE required, grant types `authorization_code` and `refresh_token`, public global subjects, RS256 ID tokens, no audience scopes. After creating one:

1. Add the redirect URLs (**Redirect URLs** tab). Matching is exact. Mark one as **Default URL** if you want to be able to omit `redirect_uri` from requests.
2. For a confidential client, **Issue secret** on the **Client secret** tab. The secret is shown once.
3. Optionally add post-logout redirect URLs, grant types, scopes of your APIs (**Scopes**), and subject settings.

Client types

	Public	**Confidential**
Use for	SPAs, mobile and desktop apps, CLIs	Server-side web apps
Client secret	None. Sending any `client_secret` (even empty) is rejected	Required at the token, introspection and revocation endpoints
PKCE	Required (default)	Required by default; can be switched off
Introspection, revocation	Not available	Available

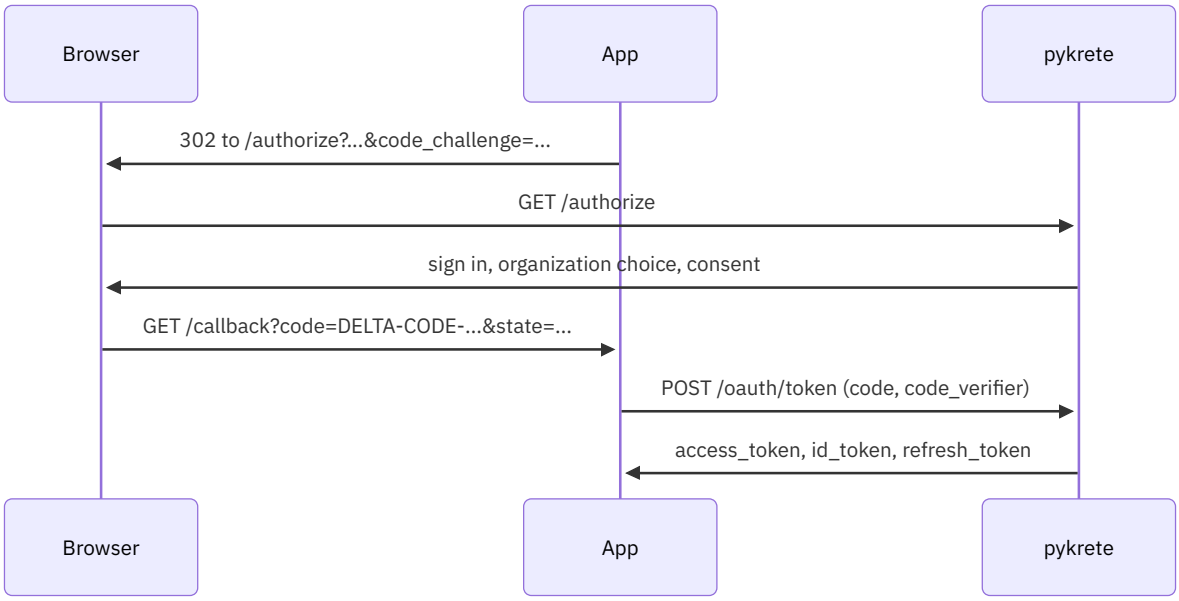
Two independent client flags decide the behavior: **Require client secret** (`usesClientSecret`) and **Require PKCE** (`usesPkce`). The type only sets their defaults.

Client secrets look like `DELTA-CLIENT-SECRET-` followed by 128 letters and digits. A client has one secret at a time: rotating it invalidates the old one immediately. The token endpoint accepts it either as HTTP Basic (`client_secret_basic`; id and secret form-urlencoded as in RFC 6749 section 2.3.1) or in the body (`client_secret_post`).

PKCE is all or nothing per client. If the client requires PKCE, `code_challenge` must be sent to `/authorize` and `code_verifier` to `/oauth/token`; if it does not, sending a `code_verifier` is an error. `S256` and `plain` are accepted; use `S256`.

> **NOTE**
>
> Requests to `/oauth/token` that carry an `Origin` header (that is, made by JavaScript in a browser) are treated as coming from a public browser app:
>
> - `authorization_code` must include `code_verifier`;
> - `refresh_token`, `device_code`, `urn:openid:params:grant-type:ciba` and `client_credentials` are refused with `invalid_request`.
>
> A browser-only app therefore cannot use refresh tokens. Refresh from a backend, or send the user through `/authorize` again (with `prompt=none` it completes without UI while the session and consent are valid).

Authorization code flow with PKCE



1. Prepare state, nonce and PKCE

`state` is mandatory. Generate a fresh `state`, nonce and PKCE verifier per login and keep them in the user's session (for example a short-lived HttpOnly cookie).

```
VERIFIER=$(openssl rand -base64 48 | tr -d '=+/\n')
CHALLENGE=$(printf '%s' "$VERIFIER" | openssl dgst -sha256 -binary | openssl base64 -A | tr '+/' '-_' | tr -d
'=')
STATE=$(openssl rand -hex 16)
NONCE=$(openssl rand -hex 16)
```

2. Send the user to /authorize

```
https://auth.example.com/authorize
  ?response_type=code
  &client_id=C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Fcallback
  &scope=openid%20profile%20email%20offline_access
  &state=6f1c0b6e2a7d4c39a0f5e8b1d2c3a4f5
  &nonce=0d9e8f7a6b5c4d3e2f1a0b9c8d7e6f5a
```

```
  &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
  &code_challenge_method=S256
```

(Line breaks added for readability.) The user signs in, may be asked to choose an organization (see Organizations) and to approve the requested permissions (see Consent).

3. Handle the callback

On success the browser is redirected to:

```
https://app.example.com/callback?code=DELTA-CODE-q3T...&state=6f1c0b6e2a7d4c39a0f5e8b1d2c3a4f5
```

Check that `state` matches what you stored. The code is valid for 10 minutes and can be used once.

4. Exchange the code

Confidential client (HTTP Basic):

```
curl -s https://auth.example.com/oauth/token \
  -u "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH:$CLIENT_SECRET" \
  -d grant_type=authorization_code \
  -d code="$CODE" \
  -d redirect_uri=https://app.example.com/callback \
  -d code_verifier="$VERIFIER"
```

Public client: no secret, `client_id` in the body:

```
curl -s https://auth.example.com/oauth/token \
  -d grant_type=authorization_code \
  -d client_id=C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH \
  -d code="$CODE" \
  -d redirect_uri=https://app.example.com/callback \
  -d code_verifier="$VERIFIER"
```

`redirect_uri` must be the same value as in the authorization request. If omitted, the client's Default URL is assumed. An optional `scope` parameter narrows the token to a subset of the granted scopes. `grant_type` defaults to `authorization_code` when missing.

Response (`Cache-Control: no-store`):

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6Im9pZGMtdG9rZW4tc2lnbmluZy1yczI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer",
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6Im9pZGMtdG9rZW4tc2lnbmluZy1yczI1NiIsInR5cCI6IkpXVCJ9...",
  "expires_in": 3600,
  "scope": "openid profile email offline_access",
  "refresh_token": "eyJ..."
}
```

- `id_token` is present only when `openid` was granted.
- `refresh_token` is present only when `offline_access` was granted (see Refresh tokens).
- `keys_jwe` is added when app keys were requested, see App keys.

5. Validate the ID token

Verify the signature with the JWKS (pick the key by `kid`), then check `iss` is `https://auth.example.com/` (trailing slash), `aud` is your client id, `exp` is in the future and `nonce` equals the value you sent. Decoded payload:

```
{
  "iss": "https://auth.example.com/",
  "sub": "U4Fh8kL2mN6pQ9rS3tV7wX1yZ5bC0dE2g",
  "aud": "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH",
  "exp": 1793530800,
  "iat": 1793527200,
  "email": "ada@example.com",
  "email_verified": true,
  "name": "Ada Lovelace",
  "picture": "https://auth.example.com/images/3b0c...e1",
  "updated_at": 1790000000,
  "nonce": "0d9e8f7a6b5c4d3e2f1a0b9c8d7e6f5a"
}
```

sub depends on the client's subject settings, see Subject identifiers.

Library settings

Any standard OIDC client library works with discovery. Typical settings:

```
issuer: https://auth.example.com/          # trailing slash, must equal "iss"
client_id: C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH
client_secret: DELTA-CLIENT-SECRET-...     # confidential clients only
token_endpoint_auth_method: client_secret_basic   # or client_secret_post; "none" for public clients
redirect_uri: https://app.example.com/callback
response_type: code
scope: openid profile email offline_access
pkce: S256                                 # required unless the client switched PKCE off
id_token_signed_response_alg: RS256        # or ES256, as set on the client
post_logout_redirect_uri: https://app.example.com/
```

> **TIP**
>
> Make sure the library sends `state`; some skip it when PKCE is on, and pykrete rejects the request.

Authorization request

Parameters

Parameter	Status	Notes
`response_type`	Required	Only `code` is supported.
`client_id`	Required	
`state`	Required	Echoed back on success and on errors. A request without it fails.
`scope`	Required	Space-separated. Validation is strict, see Scope validation.
`redirect_uri`	Optional	Must exactly match a registered URL as stored (a bare origin is stored with a trailing slash: `https://app.example.com/`). URLs with a query string never match. When omitted, the client's single Default URL is used; without one the request fails.
`nonce`	Optional	Copied into the ID token.
`code_challenge`, `code_challenge_method`	Per client	Required when the client requires PKCE.

audience	Optional	The Audience ID (A...) of one API. See Audiences.
organization	Optional	Organization id or slug to sign in to. See Organizations.
prompt	Optional	See below.
response_mode	Ignored	The response always uses the query string. form_post is rejected.
keys_jwk	Optional	Ephemeral key for app keys, see App keys.
login_hint, max_age, acr_values, ui_locales, id_token_hint, display	Ignored	Use prompt=login to force re-authentication.
access_type	Ignored	Firefox Accounts compatibility. Request offline_access instead.
action, entrypoint, email, utm_*	Ignored	Firefox Accounts compatibility.

Unknown parameters (for example resource) are ignored.

prompt

Value	Effect
none	No UI is shown. If any would be needed, the user is redirected back with login_required (no session), consent_required (new permissions or an organization decision needed), interaction_required (several organizations to choose from) or access_denied (the client requires an organization and none is available).
login	Ends the current session and asks the user to sign in again.
consent	Shows the consent screen and the organization access screen even if everything was granted before.
select_account	Forces the organization picker for clients that require an organization. Account switching is not supported.

Values can be combined (prompt=login consent), except none. Unknown values or none combined with others make the request fail.

Errors

Problems with client_id, redirect_uri or the request syntax (missing state, unknown prompt) are shown on an error page and never redirect. Everything else is returned to the redirect URI:

```
https://app.example.com/callback?error=invalid_scope&error_description=invalid+scope%3A+user+has+no+permission+to+use+scope+write+at+AudienceId%28%22A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n%22%29&state=6f1c0b6e2a7d4c39a0f5e8b1d2c3a4f5
```

error	Typical cause
invalid_scope	A scope is undefined, not allowed for the client, or not held by the user; also returned when the client is not allowed the authorization_code grant

`invalid_client`	The audience does not exist
`access_denied`	The user cancelled consent or the organization step
`login_required`, `consent_required`, `interaction_required`	`prompt=none` could not complete silently
`invalid_request`	`organization` sent to a client that does not use organizations
`server_error`	Internal error

Values a user could have typed are obfuscated in `error_description` as `<E>...</E>` (Crockford base32), so the description is safe to show but not always readable.

Token endpoint

`POST /oauth/token`, form or JSON body. Errors are JSON:

```
{"error": "invalid_grant", "error_description": "invalid authorization code"}
```

Status is 400, except `invalid_client` (401 with `WWW-Authenticate: Basic`). Common errors:

`error`	**Cause**
`invalid_request`	Missing parameter or `client_id`; browser `Origin` rules above
`invalid_client`	Unknown client, missing or wrong secret, secret sent by a public client
`invalid_grant`	Code unknown, expired, already used, issued to another client or `redirect_uri` mismatch; PKCE failure; invalid refresh token; unknown or already redeemed `device_code` or `auth_req_id`; the active organization is no longer available
`invalid_scope`	Scope validation failed; grant type not allowed for the client
`invalid_target`	The token's audiences use different access token signature kinds
`unsupported_grant_type`	`implicit`
`authorization_pending`	Device or CIBA flow, the user has not decided yet
`slow_down`	Device or CIBA flow, polled again sooner than `interval` seconds
`expired_token`	Device or CIBA flow, the request expired before the user approved it
`access_denied`	Device or CIBA flow, the user denied the request
`server_error`	Internal error, for example a failing claims hook

Refresh tokens

A refresh token is returned by the `authorization_code` grant when the granted scope contains `offline_access`. To use it, the client also needs the `refresh_token` grant type (on by default for clients created on the account pages; add `refresh_token` to `grant_types` for dynamically registered ones).

```
curl -s https://auth.example.com/oauth/token \
  -u "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH:$CLIENT_SECRET" \
  -d grant_type=refresh_token \
  -d refresh_token="$REFRESH_TOKEN"
```

- The response has the same shape as above, including a new `id_token` when `openid` is in scope (without

nonce).

- Rotation is mandatory: every refresh returns a new refresh token and the presented one stops working. Store the new one.
- Reuse detection: presenting an already used refresh token revokes the whole chain; the user has to sign in again. Avoid parallel refreshes with the same token.
- Lifetime: 30 days, absolute, counted from the first issuance. Rotation does not extend it.
- `scope` may narrow the new access token to a subset of the original scope.
- Each refresh re-checks scopes and organization access, so a user who lost a role or removed the app's access to an organization gets `invalid_scope` / `invalid_grant`.
- Refresh tokens are opaque encrypted strings; do not parse them.
- Public clients refresh without a secret, but not from a browser (see the `Origin` note above).

Revoking a refresh token (`POST /oauth/revoke`) revokes its chain, see Backend API.

Device authorization grant

For CLIs and TVs (RFC 8628). The client needs the `urn:ietf:params:oauth:grant-type:device_code` grant type.

```
curl -s https://auth.example.com/device-auth \
  -d client_id=C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH \
  -d scope="openid profile offline_access"
```

```
{
  "device_code": "DELTA-DEVICE-Xk2...",
  "user_code": "HMQT-4WXA",
  "verification_uri": "https://auth.example.com/code",
  "verification_uri_complete": "https://auth.example.com/code?user_code=HMQT-4WXA",
  "expires_in": 1800,
  "interval": 5
}
```

Request parameters (form or JSON body):

- `client_id`: required, in the body or as HTTP Basic user name.
- `client_secret`: required for clients that use a client secret, in the body or as HTTP Basic password, the same way as at the token endpoint. Public clients must not send one.
- `scope`: optional, space-separated. The user sees it on the consent screen; the issued token carries what they approved, plus the client's required scopes (see Client scopes). `openid` adds an ID token, `offline_access` a refresh token. `app_key:` scopes are refused with `invalid_scope`, because the device has no `keys_jwk`; use the authorization code flow or CIBA for those.
- `audience`: optional audience id.

Show the user the `verification_uri` and `user_code`. `verification_uri_complete` opens the page with the code already filled in; the user still submits it. Then poll every `interval` seconds, authenticating the same way:

```
curl -s https://auth.example.com/oauth/token \
  -d grant_type=urn:ietf:params:oauth:grant-type:device_code \
  -d client_id=C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH \
  -d device_code="$DEVICE_CODE"
```

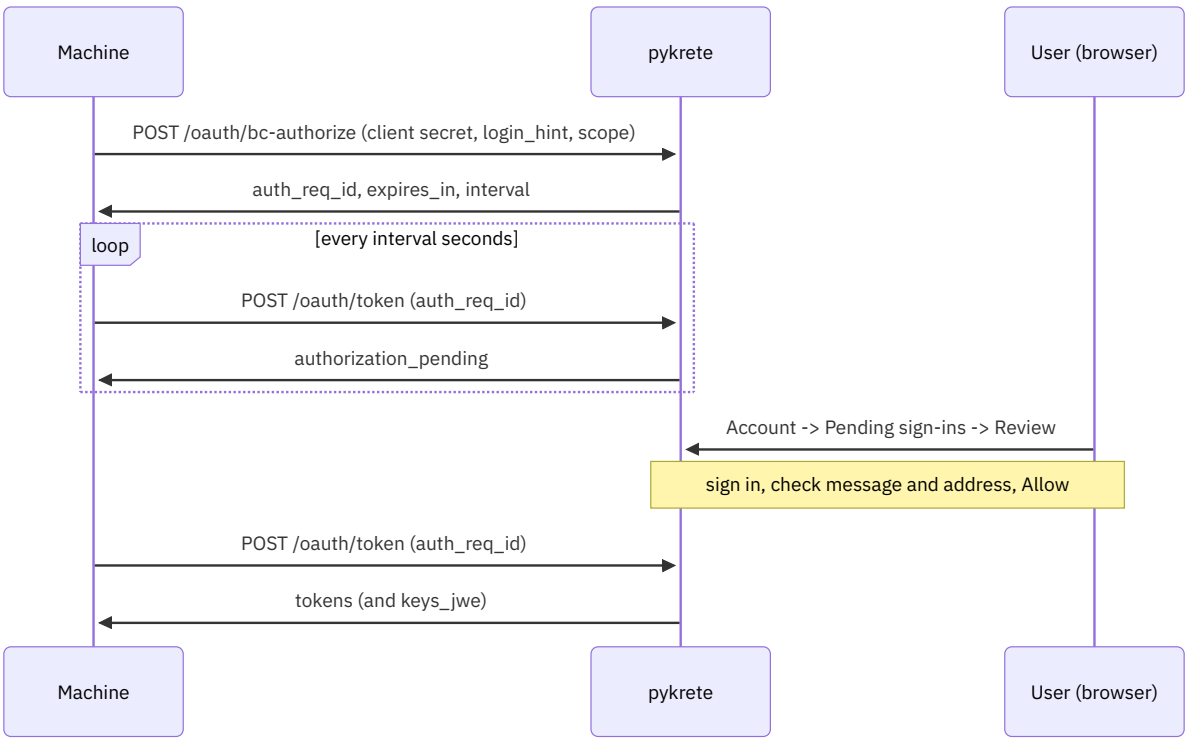
Until the user confirms, the answer is HTTP 400 with `authorization_pending`. Polling again sooner than `interval` seconds returns `slow_down`; wait longer before the next poll. When the user denies, the answer is `access_denied`; after `expires_in` seconds it is `expired_token`. Both end the flow: request a new code to try again. On approval the answer is a normal token response, once; the `device_code` is gone afterwards.

Backchannel authentication (CIBA)

For machines that have no browser and nobody in front of them, such as a server that needs a key at boot (OpenID CIBA Core 1.0, poll mode). The machine asks pykrete for a specific user's approval; the user approves on pykrete and the machine picks up the tokens. With an `app_key:` scope, the approval also delivers the user's end-to-end encryption key, see App encryption keys.

Requirements:

- The client uses a client secret, and has the `urn:openid:params:grant-type:ciba` grant type (**Backchannel (CIBA)** on the **Grant types** tab). Dynamic registration can't enable it.
- The client may only ask for users it belongs to: a personal client only for its owner, an organization's client only for members of that organization. A client without an owner (created by an administrator) may ask for any user. Any other user is answered with `unknown_user_id`, the same as a user that doesn't exist.



Request

```
curl -s https://auth.example.com/oauth/bc-authorize \
  -u "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH:$CLIENT_SECRET" \
  -d scope="openid app_key:luks" \
  -d login_hint=alice \
  -d binding_message="nas1: unlock 2 disks" \
  -d keys_jwk="$KEYS_JWK"
```

```
{"auth_req_id": "DELTA-BC-Xk2...", "expires_in": 600, "interval": 5}
```

Parameter	Required	Meaning
`scope`	yes	Space-separated; must contain `openid`.
`login_hint`	one of the two	User name (current or previous) or user id (`U...`).
`id_token_hint`	one of the two	An ID token pykrete issued to this client earlier; its `sub` selects the user. Expired ID tokens are accepted.
`binding_message`	no	Up to 128 characters, no control characters. Shown to the user on the approval screen; put the machine's name and why it is asking.
`requested_expiry`	no	Seconds the request stays valid, 60 to 3600 (other values are clamped). Default 600.
`keys_jwk`	with `app_key:` scopes	Base64url-encoded public EC P-256 JWK, as for `/authorize`. Must be a valid point on the curve without `d`.
`audience`	no	Audience id.

`login_hint_token` and `user_code` are not supported. Client authentication works as at the token endpoint (`client_secret_basic` or `client_secret_post`).

Errors:

`error`	Cause
`invalid_client` (401)	Missing or wrong client credentials, or the client has no client secret
`unauthorized_client`	The client doesn't have the CIBA grant type
`unknown_user_id`	No such user, or the client may not ask for this user
`invalid_binding_message`	`binding_message` too long or contains control characters
`invalid_request`	No `openid` in `scope`, not exactly one of `login_hint`/`id_token_hint`, `login_hint_token` given, missing or invalid `keys_jwk`, or 5 requests for this user from this client are already pending
`invalid_scope`	Scope validation failed (unknown scope, scope not allowed for the client or the user)

pykrete records the address the request came from (the `X-Real-IP` header set by the reverse proxy, see Reverse proxy and TLS) and shows it to the user.

Approval

The request appears under **Account → Pending sign-ins** (`/account/pending`) of that user, with the client name, the binding message, the address and the requested scopes. **Review** opens `/approve/<id>`, where the user signs in (with a passkey or password when an `app_key:` scope is requested, even with an active session), sees the binding message, address and time next to the requested permissions, and chooses **Allow** or **Deny**. The confirmation is always shown, also for scopes the user consented to before. **Deny** on the list rejects the request without signing in.

pykrete doesn't notify the user by itself: the user has to open the page. Tell the user out of band if needed, for example from the machine's own monitoring.

Polling

```
curl -s https://auth.example.com/oauth/token \
```

```
  -u "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH:$CLIENT_SECRET" \
  -d grant_type=urn:openid:params:grant-type:ciba \
  -d auth_req_id="$AUTH_REQ_ID"
```

The answers are the same as for the device flow: `authorization_pending` while waiting, `slow_down` when polling faster than `interval`, `access_denied` after **Deny**, `expired_token` after `expires_in` seconds, `invalid_grant` for an unknown or already redeemed `auth_req_id`. On approval the response is a normal token response, once, with `keys_jwe` when app keys were requested. `offline_access` adds a refresh token.

Client credentials

Not available yet. Use API keys for machine access.

Userinfo

```
curl -s https://auth.example.com/userinfo -H "Authorization: Bearer $ACCESS_TOKEN"
```

```
{
  "sub": "U4Fh8kL2mN6pQ9rS3tV7wX1yZ5bC0dE2g",
  "email": "ada@example.com",
  "email_verified": true,
  "name": "Ada Lovelace",
  "picture": "https://auth.example.com/images/3b0c...e1",
  "updated_at": 1790000000
}
```

- GET only, CORS open to any origin.
- The token must include `openid` and must not be expired or revoked; otherwise 401 `invalid_token` (or 400 `invalid_scope`).
- The response is `sub` plus the ID token claims recorded when the access token was issued (including organization and claims-hook claims); it does not reflect later profile changes. Refresh the token to see them.

Tokens

Signing and keys

All tokens issued to apps are JWS compact JWTs:

```
{"alg": "RS256", "kid": "oidc-token-signing-rs256", "typ": "JWT"}
```

kid	Key
`oidc-token-signing-rs256`	RSA 2048, RS256
`oidc-token-signing-es256`	P-256, ES256
`oidc-refresh-encrypt-es256`	`use: enc`, internal (refresh token encryption); ignore it

- ID tokens are signed with the client's **ID token signature** setting: RS256 (default) or ES256.
- Access tokens are signed with the **access token signature** of their audience (API): RS256 (default) or ES256. Tokens without an audience use RS256. If one token would cover audiences with different settings, the request fails with `invalid_target`.

- Lifetimes: access and ID tokens 1 hour (`expires_in: 3600`); the server accepts access tokens for 2 more minutes of clock skew. Authorization codes 10 minutes. Refresh token chains 30 days.

Access token

```
{
  "iss": "https://auth.example.com/",
  "sub": "U4Fh8kL2mN6pQ9rS3tV7wX1yZ5bC0dE2g",
  "aud": "A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n",
  "exp": 1793530800,
  "iat": 1793527200,
  "azp": "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH",
  "scope": "openid email read:projects"
}
```

- `aud` holds Audience IDs: a string for one audience, an array for several, `[]` when the token has none.
- `azp` is the client the token was issued to; `scope` lists every granted scope, builtin ones included.
- Organization claims (`org_id`, `org_slug`, `org_name`, `org_roles`) and claims-hook claims are added when present, see Claims.

Access tokens are also recorded server-side. Revocation and expiry are enforced by `/userinfo`, `/graphql` and introspection, but an API that only checks the JWT signature cannot see a revocation until the token expires.

Validating access tokens in an API

Introspection only reports tokens issued to the calling client, so an API that serves several apps validates the JWT itself. Check signature (JWKS), `iss`, that `aud` contains your Audience ID, `exp`, and the scopes you need. TypeScript with `jose`:

```
import { createRemoteJWKSet, jwtVerify } from "jose";

const jwks = createRemoteJWKSet(new URL("https://auth.example.com/jwks"));

export async function verify(token: string, needed: string) {
  const { payload } = await jwtVerify(token, jwks, {
    issuer: "https://auth.example.com/",
    audience: "A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n",
  });
  if (!String(payload.scope ?? "").split(" ").includes(needed)) {
    throw new Error("insufficient_scope");
  }
  return payload;
}
```

Logout

RP-initiated logout: send the browser to the end session endpoint (GET, or POST as a form).

```
https://auth.example.com/oauth/logout
  ?id_token_hint=eyJhbGciOiJSUzI1NiIs...
  &post_logout_redirect_uri=https%3A%2F%2Fapp.example.com%2F
  &state=4c1d...
```

Parameter	Notes
`id_token_hint`	Recommended. Must be a valid ID token from this server (signature and issuer are checked; expired is fine). Its `aud` identifies the client.

`client_id`	Needed with `post_logout_redirect_uri` when there is no `id_token_hint`. If both are sent, `client_id` must be in the hint's `aud`.
`post_logout_redirect_uri`	Must exactly match one of the client's post-logout redirect URLs. Auth0's `returnTo` is accepted as an alias.
`state`	Appended to the post-logout redirect.
`logout_hint`, `ui_locales`	Ignored.

Clear your app's own session before sending the user here. If the user has a pykrete session, they are asked whether to sign out of pykrete too (and warned if the hint belongs to another account). Then they are redirected to `post_logout_redirect_uri` (with `state`), or shown a message if none was given.

Front-channel and back-channel logout are not supported: other apps are not notified.

Introspection, revocation, API keys

`POST /oauth/introspect` (RFC 7662), `POST /oauth/revoke` (RFC 7009), first-party API keys and the Clerk-compatible REST API are described in Backend API. Both endpoints need a confidential client and only see that client's own tokens.

Dynamic client registration

Open registration (RFC 7591) with management (RFC 7592). An `Authorization: Bearer` access token carrying the `account` scope is optional; with it, the new client is owned by that user and appears under their `/account/clients`. Without it, the client has no owner and the consent screen shows "Unknown publisher".

```
curl -s https://auth.example.com/register \
  -H 'content-type: application/json' \
  -d '{
    "client_name": "Example App",
    "redirect_uris": ["https://app.example.com/callback"],
    "post_logout_redirect_uris": ["https://app.example.com/"],
    "grant_types": ["authorization_code", "refresh_token"],
    "token_endpoint_auth_method": "client_secret_basic",
    "subject_type": "public"
  }'
```

```
{
  "client_id": "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH",
  "client_secret": "DELTA-CLIENT-SECRET-...",
  "client_id_issued_at": 1793527200,
  "client_secret_expires_at": 0,
  "registration_access_token": "DELTA-REGISTRATION-...",
  "registration_client_uri": "https://auth.example.com/register/C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH",
  "client_name": "Example App",
  "redirect_uris": ["https://app.example.com/callback"],
  "post_logout_redirect_uris": ["https://app.example.com/"],
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "client_secret_basic",
  "application_type": "web",
  "subject_type": "public",
  "id_token_signed_response_alg": "RS256"
}
```

Field	Default	Accepted
client_name	Unnamed app	Up to 100 characters
redirect_uris		Required for authorization_code. No fragment, query or credentials. web: https, or http on a loopback host. native: http loopback only, custom schemes are not supported
post_logout_redirect_uris		http or https, no fragment
grant_types	["authorization_code"]	authorization_code, refresh_token, urn:ietf:params:oauth:grant-type:device_code
response_types	["code"] with authorization_code	Only code, exactly when authorization_code is registered
token_endpoint_auth_method	client_secret_basic	none (public client), client_secret_basic, client_secret_post; the token endpoint accepts either secret method
application_type	web	web, native
subject_type	pairwise	pairwise, public
sector_identifier_uri		https URL returning a JSON array that lists every redirect URI
id_token_signed_response_alg	RS256	RS256, ES256

Other fields are ignored. Errors are invalid_redirect_uri or invalid_client_metadata (400); a bad bearer token gives 401 invalid_token, a token without account 403 insufficient_scope.

Registered clients always require PKCE, and their redirect URIs are never Default URLs, so always send redirect_uri. With pairwise subjects the sector is the redirect URIs' host (all non-loopback redirect URIs must share one host) or the host of sector_identifier_uri. A new client can only use builtin scopes until its owner or an admin allows it scopes of an API.

Manage the registration with the registration_access_token:

```
curl -s https://auth.example.com/register/C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH \
  -H "Authorization: Bearer $REGISTRATION_ACCESS_TOKEN"
```

- GET returns the current metadata (including the secret).
- PUT replaces the metadata; the body must contain the same client_id and, if sent, the current client_secret. Omitted fields fall back to the defaults. Changing subject_type or the sector after users have signed in is refused.
- DELETE deletes the client (204).

Scopes and claims

What a token may contain is decided by four things: the audiences (APIs) it is for, the scopes those audiences declare, the roles of the user, and the scopes the client is allowed to ask for. This page explains each, how a request is validated, when users are asked for consent, and which claims end up in ID tokens, access tokens and userinfo. The protocol side is in OpenID Connect and OAuth.

Audiences

An audience is an API (resource server) that accepts pykrete access tokens. It has:

Field	Meaning
Audience ID	`A` + 32 letters and digits, generated. Used in requests and in the `aud` claim
Identifier	Unique free-form name, for example `https://api.example.com`
Name, description, picture	Shown on the consent screen
Access token signature	RS256 (default) or ES256, see Signing and keys
Owner	A user, an organization, or the system
Scopes	Declared by the audience, see below

Audiences are created under **Account → OIDC → Audiences** (`/account/audiences`), owned by yourself or by an organization where you have `org:audiences`, or by admins through GraphQL. See Self-service and Admin access.

Each audience declares its scopes: an id without spaces (for example `read:projects`), a name and a description. Name and description are what users see on the consent screen.

Requesting an audience

Pass the Audience ID as `audience` to `/authorize` (or `/device-auth`) together with the scopes you need:

```
https://auth.example.com/authorize?response_type=code
  &client_id=C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Fcallback
  &audience=A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n
  &scope=openid%20email%20read:projects
  &state=...&code_challenge=...&code_challenge_method=S256
```

- One `audience` per request. More audiences are added automatically by the client's required scopes (see Client allowed and required scopes), so a token can cover several APIs.
- The resulting access token has `"aud": "A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n"` (an array when there are several, `[]` when there is none). APIs check that `aud` contains their Audience ID.
- An unknown `audience` fails the request with `invalid_client`.
- All audiences in one token must use the same access token signature kind, otherwise the token request fails with `invalid_target`.

The self audience

pykrete itself is an audience, with Audience ID `A_____SELF` and the issuer (`https://auth.example.com/`) as Identifier. Its scopes:

Scope	Meaning
`account`	"View and manage your account": the token can call `/graphql` as the user (`me`, `manage`) and register clients owned by the user (`POST /register`). Every user holds it; a client still needs it allowed or required. Intended for first-party apps; the account dashboard requires it.
`admin:graphql`	Full admin GraphQL access. Held through the `role:admin` role. See Admin access.
`org:manage`, `org:members`, `org:roles`, `org:clients`, `org:audiences`, `org:access`, `org:api-keys`	Organization permissions. They are granted by organization roles and checked by the management API from the user's roles; they are not meant to be requested by apps. See Self-service.

Only admins can allow a client to use self audience scopes. Tokens whose only audience is the self audience are signed with RS256.

Builtin scopes

Builtin scopes need no audience, no role and no client allowance: any client may request them for any user. They still need consent.

Scope	Consent label	Effect
`openid`	User id	An ID token is issued; `/userinfo` works
`profile`	Profile	`name`, `picture`, `updated_at` (each only if the user has it)
`email`	Email	`email` and `email_verified: true`, only if the user has a verified address
`offline_access`	Offline access	A refresh token is issued, see Refresh tokens
`organizations`	Organizations	`orgs` claim with the organizations the user shares with the app, see Organizations
`created_at`	Registration time	Not available yet

App key scopes

`app_key:<id>` requests an end-to-end encryption key for the app. The client must be allowed that app key, the user always sees a consent prompt for it (it is never remembered), and the key is delivered in `keys_jwe` of the token response. See App keys.

Roles

A role is a named bundle of scopes, each scope being an (audience, scope) pair. Users get audience scopes only through roles (`account` aside).

Global roles

Granted to a user directly (admin GraphQL: `grantRole` / `revokeRole` on a user). Always count.

Organization roles

Held as a member of an organization. They count only while that organization is the active organization of the request (see Organizations). Unless an admin sets it up, a role owned by an organization may only contain scopes of that organization's audiences plus `org:` permissions.

Organization-assignable global roles

Global roles that organization admins may assign to members, flagged by an admin.

Built-in roles:

Role	Scopes
`role:admin`	`admin:graphql` on the self audience
`role:org-admin`	All `org:` permissions; organization-assignable
`role:org-member`	None; organization-assignable

The scopes a user may receive for audience `A` are therefore:

- all scopes of `A` in the user's global roles, plus
- all scopes of `A` in the user's member roles in the active organization, if any, plus
- `account`, if `A` is the self audience.

Example: audience `A9x...` declares `read:projects` and `write:projects`. Role `role:projects-reader` contains `(A9x..., read:projects)`. A user who holds that role can get `read:projects`; requesting `write:projects` fails for them (see Scope validation).

Client allowed and required scopes

Each client lists audience scopes it may use:

Allowed scopes

The client may request them. They are included only when requested.

Required scopes

Always added to every user token of the client, together with their audience, even when not requested and without an `audience` parameter. They count as allowed.

Both are edited on the client's **Scopes** tab or with the `setAllowedScopes` / `setRequiredScopes` mutations. You can only add scopes of audiences you can manage (your own, your organization's with `org:audiences`, or any as admin).

Uses for required scopes:

- Give a client an API token without passing `audience` (the account dashboard requires `account` this way).
- Restrict who can use an app: if the client requires `(A..., use:app)`, everyone without a role carrying that scope gets `invalid_scope` at sign-in.
- Give device flow tokens access to an API (device tokens carry only required scopes, see Device authorization grant).

Builtin scopes cannot be allowed, required or restricted per client.

Scope validation

Validation is strict: pykrete never silently drops a scope. If anything is wrong, the whole request fails with `invalid_scope`, listing every problem. The rules:

1. Split `scope` on spaces and add the client's required scopes (and their audiences).
2. Builtin scopes always pass.
3. `app_key:<id>` passes if the client is allowed that app key.
4. Every other scope must be declared by at least one of the token's audiences; otherwise it is undefined. A scope of an audience you did not request (and that no required scope brings in) is undefined too.
5. For every token audience that declares the scope, the user must hold it (see Roles) and the client must be allowed it.

Validation runs at `/authorize` (after sign-in and the organization step), again at the token endpoint, and on every refresh, so losing a role makes the next refresh fail.

Error descriptions look like this (the undefined scope value, here `read:all`, is obfuscated because a user could have typed it):

```
invalid scope: undefined scope: <E>E9JP2S1TC5P6R</E>
invalid scope: user has no permission to use scope write:projects at
AudienceId("A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n")
invalid scope: client has no permission to use scope write:projects at
AudienceId("A9xK3mP7qR2sT6vW0yZ4bC8dF1gH5jL3n")
```

At `/authorize` the error goes to the redirect URI (`error=invalid_scope`); at the token endpoint it is a 400 JSON error. If your app has optional features behind scopes, request them in a separate authorization request once the user enables the feature, rather than always asking for everything.

Consent

After sign-in and the organization step, pykrete compares the validated scopes with what the user already approved for this client:

- audience scopes, remembered per user, client, audience and scope;
- builtin scopes, remembered per user, client and scope;
- app key scopes, never remembered.

If anything is new, the consent screen lists the new permissions grouped as "Your identity", "Encryption keys" and one group per API, plus the previously approved ones. The client name is shown on top, followed by a heading that names the targets and the active organization ("Requests access to your identity and Projects API in Acme"). Clients that are not first-party show their publisher (owner) under the name and whether they are verified by an admin.

- Approve: the new permissions are remembered and the flow continues.
- Cancel: the app gets `error=access_denied`.
- `prompt=consent`: the screen is shown even when nothing is new.
- `prompt=none`: if the screen would be needed, the app gets `error=consent_required`.

Required scopes are consented like any other scope. Remembered consent does not expire; withdrawing it is not available yet. **Access → Applications** (`/account/applications`) lists it, see Account dashboard.

Claims

ID token

Claim	When
`iss`	Always, `https://auth.example.com/` (trailing slash)
`sub`	Always, see Subject identifiers
`aud`	Always, the client id (a string)
`exp`, `iat`	Always; valid for 1 hour
`nonce`	If sent to `/authorize` (not on refresh)
`name`, `picture`, `updated_at`	`profile`, if set for the user. `picture` is an absolute `https://auth.example.com/images/<hash>` URL
`email`, `email_verified`	`email`, if the user has a verified address; `email_verified` is always `true`
`orgs`	`organizations`: array of `{"id", "slug", "name", "roles"}`
`org_id`, `org_slug`, `org_name`, `org_roles`	An organization is active
custom	Returned by the claims hook for `id_token`

Example with an active organization and the `organizations` scope:

```
{
  "iss": "https://auth.example.com/",
  "sub": "S8kM2pR6tV0xZ4bD8fH2jL6nP0rT4vX8z",
  "aud": "C7cJ2kQ9xVb4LmN8pR3sT6wY1zA5dF0gH",
  "exp": 1793530800,
  "iat": 1793527200,
  "orgs": [
    {"id": "O2bD6fH0jL4nP8rT2vX6zB0dF4hJ8lN2p", "slug": "acme", "name": "Acme", "roles": ["role:org-admin"]}
  ],
  "org_id": "O2bD6fH0jL4nP8rT2vX6zB0dF4hJ8lN2p",
  "org_slug": "acme",
  "org_name": "Acme",
  "org_roles": ["role:org-admin"]
}
```

Access token

Claim	Value
`iss`, `sub`, `exp`, `iat`	As in the ID token
`aud`	Audience IDs: a string for one, an array for several, `[]` for none
`azp`	The client id
`scope`	Space-separated granted scopes, builtin and app key scopes included
`org_id`, `org_slug`, `org_name`, `org_roles`	An organization is active

custom	Returned by the claims hook for access_token

Profile, email and orgs claims are not put into access tokens. An API that needs them can call /userinfo with the token, or get them from the claims hook.

Userinfo

GET /userinfo returns sub plus the ID token claims other than iss, aud, exp, iat and nonce, as recorded when the access token was issued. It needs openid in the token.

Custom claims

With hooks configured, pykrete calls the claims hook on every token issuance (all grants, refreshes included) with the grant type, client id, audience identifiers, scope, the user (raw U... id plus public and private metadata), the active organization and the listed organization ids. The hook answers with extra claims for id_token and access_token:

```
{
  "id_token": {"groups": ["staff"]},
  "access_token": {"tenant": "acme"}
}
```

Hook claims cannot replace existing claims or use reserved names (iss, sub, aud, exp, iat, nbf, jti, azp, scope, nonce, auth_time, at_hash, c_hash, acr, amr, sid, orgs, or anything starting with org_). Doing so, a hook error or a timeout fails the token request with server_error. See HTTP hooks.

Subject identifiers

sub is chosen per client by two settings, on the client's **Subject identifiers** tab (setSubject), through admin GraphQL (setClientSubject), or with subject_type at dynamic registration:

Subject type	Subject scope	sub
public (default)	global (default)	The user id, U + 32 characters. Same for every client
pairwise	global	S..., one per user and sector
public	organization	S..., one per user and active organization. The user id when no organization is active
pairwise	organization	S..., one per user, active organization and sector. The pairwise global value when no organization is active

- S... values (S + 32 characters) are random, stored and reused, so they are stable but cannot be correlated across sectors or organizations.
- Sector: clients created on the account pages have their own sector (urn:pykrete:client:<client_id>), so pairwise subjects differ per client. **Share subjects with** puts a client into the sector of another client of the same owner, so both see the same sub. Dynamically registered clients use the host of their redirect URIs or of sector_identifier_uri.
- The same sub appears in ID tokens, access tokens, userinfo and introspection. Hooks always get the raw user id.
- Changing these settings changes every sub the client has seen. Once the client issued tokens or got consent, the change needs force: true ("Change even if users already signed in").

- The Backend API requires `public` + `global`. Per-organization subjects for NetBird are described in Organizations.

Organizations

Organizations group users into teams or tenants. They exist once for the whole server and are not owned by a client. An application (OAuth client) never sees an organization automatically: the organization has to be enabled for the client, and every user decides per application which of their organizations it may see, similar to GitHub's organization access for OAuth apps.

This page is for application developers and organization admins. Creating organizations, managing members, roles and the `org:*` permissions are covered in Self-service; server-to-server access to organizations is in Backend API.

Concepts

Organization

Id `0...` (0 plus 32 alphanumerics), a unique slug (1-63 characters of `a-z`, `0-9` and inner dashes), a name, an optional picture, and two JSON metadata objects (see Organization metadata).

Member

A user in the organization, holding zero or more roles. Built-in roles are `role:org-admin` and `role:org-member`; organizations can define their own roles, whose ids look like `0...:<key>`.

Client access

The organization has enabled the client. Only enabled organizations are offered to the client.

Eligible organizations

For a given user and client: the user's memberships whose organization enabled the client. For a first-party client (`allOrganizations`, see below) all memberships are eligible.

Organization access (per user and app)

For each eligible organization the user's decision for this client: granted, hidden, or not decided yet. Only granted organizations reach the client.

Access request

A member asks for an organization that has not enabled the client yet. An organization admin approves or denies it.

Active organization

One granted organization pinned to a sign-in. It is stored on the authorization code or device code and on every token minted from it, including refreshed ones.

Organization usage of a client

Each client has an organization usage setting that controls the active organization:

Value	Behavior
`NONE` (default)	No active organization. Sending `organization=` is an error. The `organizations` scope still works and lists the granted organizations.
`OPTIONAL`	The user picks an organization or "Personal account" (no active organization).

REQUIRED	Every sign-in has an active organization. Users without a granted organization cannot sign in.

Enabling organizations for a client

1. Set the usage: client settings page /account/clients/<client-id>, "Organizations" field, or GraphQL:

```
mutation {
  manage {
    client(clientId: "C0123456789abcdefghijklmnopqrstuv") {
      update(organizationUsage: OPTIONAL) { clientId organizationUsage }
    }
  }
}
```

2. Enable organizations for the client. Any of:
   - An organization admin (permission org:access) opens /account/organizations/<slug>/access and adds the client id under "Allow client", or approves a pending request there. GraphQL:

```
mutation {
  manage {
    organization(organization: "acme") {
      setClientAccess(clientId: "C0123456789abcdefghijklmnopqrstuv", enabled: true)
    }
  }
}
```

   - A server admin (admin GraphQL, see Administration) replaces the full list of organizations of a client: orgs { setClientOrganizations(clientId: "C...", organizations: ["acme", "globex"]) }. Requests are approved with orgs { approveOrganizationAccessRequest(clientId, organization) } and denied with orgs { denyOrganizationAccessRequest(clientId, organization, userId) } (omit userId to deny all requests of that organization for the client).
   - A client created by an organization (owner is the organization) is enabled for it automatically.
   - POST /api/v1/organizations on the Backend API enables the calling client for the new organization.
3. Add the organizations scope to the authorize request if the application wants the list of organizations, and/or let it pin an active organization (next sections).

/account/clients/<client-id>/organizations lists the organizations that enabled a client.

> **TIP**
> First-party applications can skip all of this. A server admin enables "All organizations (first-party)" on the client (manage { client(clientId) { setAllOrganizations(enabled: true) } }). Then all memberships of a user are eligible and granted, nothing is enabled per organization, and the user never sees the organization access checkboxes (the active organization picker still applies). See Backend API.

What the user sees

During /authorize (and device code verification) pykrete runs these steps in order:

1. Sign in.
2. Organization step, only if the client uses organizations or the organizations scope was requested. One screen with up to three parts:
   - "Visible to this app": a checkbox per eligible organization. Organizations the user has not decided on

yet are pre-checked. An organization requested with `organization=` is checked and locked.
   - "Continue as": the active organization picker. "Personal account" is offered unless the usage is `REQUIRED`. Only checked organizations can be picked.
   - "Not enabled for this app", shown together with the checkboxes: the user's other organizations, each with a "Request access" checkbox. Requests go to the organization's admins.
3. Scope consent, if the user has not consented to these scopes yet (or `prompt=consent`). With an active organization the heading reads "Requests access to … in <organization>".
4. Redirect back with the code. Organization decisions and access requests are saved only after the consent step; cancelling the organization or consent screen saves nothing and redirects with `error=access_denied`.

When the screen appears:

Part	Shown when
Organization access checkboxes	Not a first-party client, and any of: an eligible organization has no decision yet, `prompt=consent`, the `organization=` organization is not granted yet, or the usage is `REQUIRED` and nothing is granted yet (but something is eligible). Never with `prompt=none`.
"Continue as" picker	Usage is not `NONE`, no `organization=` parameter, and: for `OPTIONAL` the user has a granted organization (or the access checkboxes are shown); for `REQUIRED` more than one organization is granted, the access checkboxes are shown, or `prompt=select_account` was sent.

With `REQUIRED`, exactly one granted organization and no access checkboxes needed, that organization is used without asking.

To let users change which organizations your app sees, send them through `/authorize` with `prompt=consent`. They can also change it with GraphQL (there is no account dashboard page for this yet):

```
query {
  myOrganizations {
    applications {
      clientId name firstParty
      organizations { organization { id slug name } access }
    }
  }
}

mutation {
  myOrganizations {
    setOrganizationAccess(clientId: "C0123456789abcdefghijklmnopqrstuv", organizationId: "acme", granted: false
)
  }
}
```

`access` is one of `GRANTED`, `HIDDEN`, `UNDECIDED`, `REQUESTABLE`, `REQUESTED`. `myOrganizations { requestOrganizationAccess(clientId, organizationId) }` files an access request outside the sign-in flow.

Listing organizations: the `organizations` scope

Add `organizations` to scope. The user consents to it like any other built-in scope ("Organizations: App will see organizations you share with it, and your roles in them").

```
GET /authorize?response_type=code
  &client_id=C0123456789abcdefghijklmnopqrstuv
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Fcallback
  &scope=openid%20organizations
  &state=af0ifjsldkj
  &nonce=n-0S6_WzA2Mj
  &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
  &code_challenge_method=S256 HTTP/1.1
Host: auth.example.com
```

The ID token (and `/userinfo`) then carries `orgs`, the organizations the user granted to this client, with the user's role ids in each:

```
{
  "iss": "https://auth.example.com/",
  "sub": "U0123456789abcdefghijklmnopqrstuv",
  "aud": "C0123456789abcdefghijklmnopqrstuv",
  "exp": 1793000000,
  "iat": 1792996400,
  "orgs": [
    {"id": "O0123456789abcdefghijklmnopqrstuv", "slug": "acme", "name": "Acme Inc.", "roles": ["role:org-admin"
]},
    {"id": "O9876543210zyxwvutsrqponmlkjihgfe", "slug": "globex", "name": "Globex", "roles": ["role:org-member"
]}
  ],
  "nonce": "n-0S6_WzA2Mj"
}
```

`orgs` is only in the ID token, not in the access token. It works with any organization usage, including `NONE`.

Pinning an active organization

With usage `OPTIONAL` or `REQUIRED` the active organization comes from the picker, or the client names it with `organization=<id or slug>`:

```
GET /authorize?response_type=code
  &client_id=C0123456789abcdefghijklmnopqrstuv
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Fcallback
  &scope=openid%20profile
  &organization=acme
  &state=af0ifjsldkj
  &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
  &code_challenge_method=S256 HTTP/1.1
Host: auth.example.com
```

The organization must be eligible for the user (member, and the organization enabled the client), otherwise the request fails with `access_denied`. If the user has not granted it to the app yet, the access screen is shown with that organization locked on. The picker is skipped.

The device authorization flow (`/device-auth`) has no `organization` parameter; the picker is used there.

Both tokens get the active organization:

```
{
  "iss": "https://auth.example.com/",
  "sub": "U0123456789abcdefghijklmnopqrstuv",
  "aud": "A0123456789abcdefghijklmnopqrstuv",
  "exp": 1793000000,
  "iat": 1792996400,
  "azp": "C0123456789abcdefghijklmnopqrstuv",
```

```
  "scope": "openid profile projects:read",
  "org_id": "O0123456789abcdefghijklmnopqrstuv",
  "org_name": "Acme Inc.",
  "org_roles": ["role:org-admin"],
  "org_slug": "acme"
}
```

`org_id`, `org_slug`, `org_name` and `org_roles` appear in the ID token and the access token. Token introspection reports `org_id` and `org_roles`. Without an active organization none of them are present.

What else the active organization changes:

- Scopes: the user's roles in the active organization grant audience scopes in addition to the user's own roles. Roles the user holds in other organizations do not count. See Scopes and claims.
- Subject: a client with subject scope "Per organization" (`setSubject(subjectScope: ORGANIZATION, ...)`, page `/account/clients/<client-id>/subject`) gets a different sub (`S...`) per active organization. Without an active organization the client's normal subject is used. See Scopes and claims.
- Hooks: the claims hook receives the active organization, its metadata and the user's roles.

Switching organizations

The active organization is fixed for the lifetime of the grant, refresh tokens included. To switch, send the user through `/authorize` again. If the user already granted the target organization, this works silently:

```
https://auth.example.com/authorize?response_type=code&client_id=C0123456789abcdefghijklmnopqrstuv&redirect_uri=
https%3A%2F%2Fapp.example.com%2Fcallback&scope=openid&organization=globex&prompt=none&state=xyz&code_challenge=
...&code_challenge_method=S256
```

Without `organization=`, `prompt=none` never shows the picker: `OPTIONAL` clients get no active organization, `REQUIRED` clients get the only granted organization or an error.

Requiring an organization

Set the usage to `REQUIRED`. Then:

- The picker has no "Personal account" option, and the user cannot continue without picking.
- With one granted organization it is used without asking; `prompt=select_account` forces the picker anyway.
- A user with organizations that are eligible but not granted gets the access screen.
- A user without any eligible organization is redirected back with `access_denied` ("no organization available for this client") and can't request access from the sign-in screen. An organization admin has to enable the client first.

Errors

Authorize errors are returned to `redirect_uri` as `error`, `error_description` and `state` query parameters.

Situation	error	error_description
`organization=` sent, client usage is `NONE`	`invalid_request`	client does not use organizations
`organization=` is not eligible for the user	`access_denied`	user denied the request
User cancelled, or picked an organization that is not checked	`access_denied`	user denied the request

REQUIRED and no organization can be used	access_denied	no organization available for this client
prompt=none, and the access screen would be needed (requested organization not granted yet, or REQUIRED with eligible but no granted organizations)	consent_required	consent required (prompt=none)
prompt=none, REQUIRED, several granted organizations, no organization=	interaction_required	interaction required (prompt=none)
Token endpoint: the active organization is no longer available (see below)	invalid_grant	organization is no longer available for this grant

Refresh, revocation and changes

Organization access is re-checked every time the token endpoint issues tokens (code exchange, device code, refresh):

- The active organization must still be granted: the user is still a member, the organization still enables the client (or the client is first-party), the user has not hidden it from the app, and the client usage is not NONE. Otherwise the request fails with invalid_grant; the application has to send the user through /authorize again.
- orgs and org_roles are recomputed from the current memberships and roles.
- Scopes are re-validated against the user's current roles, including roles in the active organization. A scope the user lost fails the refresh with invalid_scope.

Removing access does not affect tokens already issued: access tokens stay valid, also on introspection, until they expire (one hour). Ways access is removed:

- The user hides the organization from the app (prompt=consent, or setOrganizationAccess(granted: false)).
- An organization admin revokes the client ("Revoke access" on /account/organizations/<slug>/access, or setClientAccess(enabled: false)).
- The user leaves or is removed from the organization. Their access decisions and access requests for it are deleted with the membership.

Organization metadata

Each organization has two JSON objects, each at most 8 KiB:

	publicMetadata	**privateMetadata**
Write	Members with org:manage (manage { organization(...) { update(publicMetadata: ...) } }), server admins, Backend API POST /organizations	Server admins only (orgs { updateOrganization(organization, privateMetadata) }), Backend API POST /organizations
Read	Manage API (publicMetadata), admin API, Backend API, claims hook	Admin API, Backend API, claims hook

Neither is copied into tokens. Use the claims hook to turn metadata into claims. An update replaces the whole object:

```
mutation SetNetbirdDomain($m: JSON!) {
  orgs {
    updateOrganization(organization: "acme", privateMetadata: $m) { id privateMetadata }
  }
}
```

```
{"m": {"netbird": {"domain": "acme.netbird.example.com"}}}
```

Users have the same pair (`orgs { userMetadata(userId) }`, `orgs { setUserMetadata(userId, publicMetadata, privateMetadata) }`, admin only), also passed to the claims hook.

Invitations

New members join through invitations. Members with `org:members` create them on `/account/organizations/<slug>/invitations` (link or email) and `/account/organizations/<slug>/members` (existing user); they can only hand out roles they may grant. GraphQL:

```
mutation {
  manage {
    organization(organization: "acme") {
      invite(email: "alice@example.org", roles: ["role:org-member"], expiresIn: 604800) {
        url
        invitation { id email roles expiresAt }
      }
    }
  }
}
```

Server admins use `orgs { createOrganizationInvitation(organization, email, user, roles, expiresIn) }`.

An invitation is a link `https://auth.example.com/invite#<code>`. The code is 40 random alphanumerics, sits in the URL fragment (so it does not reach server logs), and only its SHA-256 is stored. The link is returned once, as `url`.

Kind	Who can accept
Link only (no `email`, no `user`)	Anyone who has the link and signs in.
`email`	An account with that email verified (case-insensitive). The server emails the link (see Email hooks); if rendering or sending fails, the invitation is deleted and creation fails.
`user` (username or user id)	Only that user, through the link or under "Invitations" on their `/account/organizations` page (accept or decline). The user must not be a member already and must not have another pending invitation to the organization. Cannot be combined with `email`.

Rules:

- `expiresIn` is in seconds and must be positive; default 7 days (604800).
- Redeeming adds the membership and the invitation's roles (existing roles are kept) and deletes the invitation, so every link works once.
- Opening the link asks the user to sign in (unless already signed in), then shows "You joined <organization>", or one of: "Invitation is invalid or has expired", "This invitation is for another account", "This invitation was sent to an email address that is not verified on your account".
- Pending invitations are listed on the invitations page and can be revoked there (`revokeInvitation(id)`).

> **NOTE**
> Without email configured (see Email), invitations are not emailed. Copy the link shown after creating the invitation and send it yourself.

NetBird

NetBird can use pykrete as its external OIDC provider, with one NetBird account per pykrete organization. The default claims hook (`freshfront/routes/hooks/claims.ts`) implements the mapping.

What NetBird expects

- A JWT access token whose `aud` is NetBird's configured auth audience, with the user id in `sub`.
- Account grouping claims, namespaced by the auth audience: `<audience>/wt_account_domain` and `<audience>/wt_account_domain_category`. Users with the same domain and category `private` share one NetBird account.
- Optionally a groups claim for NetBird's JWT group sync.
- NetBird ties each `sub` to exactly one account.

> **NOTE**
> This describes NetBird's claim handling as the hook assumes it. Check it against the NetBird version you run.

How pykrete maps it

For tokens issued to one of the NetBird clients with an active organization, the hook adds to both the ID token and the access token:

Claim	Value
`<NETBIRD_AUTH_AUDIENCE>/wt_account_domain`	The organization's `privateMetadata.netbird.domain` if it is a string, else `<organization id in lowercase>.<NETBIRD_DOMAIN_SUFFIX>`
`<NETBIRD_AUTH_AUDIENCE>/wt_account_domain_category`	`private`
`groups`	The user's role ids in the active organization (for example `role:org-admin`, `0...:netbird-admins`)

Trailing slashes of `NETBIRD_AUTH_AUDIENCE` are stripped before the `/` is added. No active organization, client not listed, `NETBIRD_AUTH_AUDIENCE` unset, or no domain (no metadata and no suffix): the hook adds nothing. `wt_account_id` is not used, since it cannot place new users into an account.

Example access token:

```
{
  "iss": "https://auth.example.com/",
  "sub": "S0123456789abcdefghijklmnopqrstuv",
  "aud": "A0123456789abcdefghijklmnopqrstuv",
  "exp": 1793000000,
  "iat": 1792996400,
  "azp": "C0123456789abcdefghijklmnopqrstuv",
  "scope": "openid profile email",
```

```
  "A0123456789abcdefghijklmnopqrstuv/wt_account_domain":
"o0123456789abcdefghijklmnopqrstuv.netbird.example.com",
  "A0123456789abcdefghijklmnopqrstuv/wt_account_domain_category": "private",
  "groups": ["role:org-admin"],
  "org_id": "O0123456789abcdefghijklmnopqrstuv",
  "org_name": "Acme Inc.",
  "org_roles": ["role:org-admin"],
  "org_slug": "acme"
}
```

Configuration

1. Create an audience for NetBird. Its id (`A...`) is what goes into the token's `aud`; use it as NetBird's auth audience and as `NETBIRD_AUTH_AUDIENCE`.
2. Create the NetBird clients (dashboard, and the CLI client if separate) with the grant types, redirect URIs and scopes NetBird needs.
3. On each NetBird client:
   - organization usage `REQUIRED`, so every token has an active organization;
   - subject type public, subject scope "Per organization" (`setSubject(subjectType: PUBLIC, subjectScope: ORGANIZATION)`), so a person in two organizations is two NetBird users. With public subjects the `sub` is the same across all NetBird clients.
4. Enable the NetBird clients for each organization that should get a NetBird account (see Enabling organizations for a client).
5. Set the environment of the freshfront process and restart it (values are read at startup):

Variable	Meaning
`HOOKS_SECRET`	Must equal `frontend.hooks_secret` of the backend; without it no hook runs. See HTTP hooks.
`NETBIRD_CLIENT_IDS`	Comma-separated client ids of the NetBird clients. Other clients get no NetBird claims.
`NETBIRD_AUTH_AUDIENCE`	NetBird's auth audience, the claim prefix.
`NETBIRD_DOMAIN_SUFFIX`	Optional. Domain suffix for organizations without `privateMetadata.netbird.domain`, for example `netbird.example.com`.

6. Optionally pin a domain per organization in `privateMetadata` (example in Organization metadata).
7. In NetBird, point the OIDC configuration endpoint at `https://auth.example.com/.well-known/openid-configuration`. Where NetBird asks for the issuer, use exactly `https://auth.example.com/`, including the trailing slash: it is compared with the tokens' `iss`.
8. In NetBird, enable JWT group sync with claim name `groups` if you want role-based groups.

Caveats

- Treat the domain as permanent. Changing `privateMetadata.netbird.domain` or `NETBIRD_DOMAIN_SUFFIX` later changes the claim for every user of that organization, and NetBird uses it to place users into accounts.
- NetBird assigns its own user roles (for example who owns the account); pykrete roles only arrive as `groups` values, as role ids rather than names.

- "Personal account" sign-ins carry no grouping claims; with `REQUIRED` they cannot happen.
- A user in several organizations picks one at sign-in. Silent re-authentication (`prompt=none`) without `organization=` fails with `interaction_required` for such users; refresh tokens keep the pinned organization.
- Changing a client's subject settings after tokens were issued changes every `sub` (the API asks for `force: true`); set them before the first sign-in.
- NetBird's IdP manager integration is not supported. Run NetBird without an IdP manager.

HTTP hooks

Hooks let you plug custom logic into pykrete without changing the backend: the backend POSTs a JSON document to the frontend (freshfront) at `/hooks/<name>` and uses the JSON answer. The default implementations live in `freshfront/routes/hooks/*.ts`; edit them to customize behavior.

Hooks at a glance

Hook	Fires	On failure	Default implementation
`identity`	A user was identified in an interactive flow: sign-in page, `/authorize` (also silent `prompt=none`), device code verification, adding a passkey, accepting an invitation	The flow is aborted	Does nothing
`claims`	Every successful token request (`authorization_code`, `refresh_token`, device code), before the tokens are signed	The token request fails with `server_error`	Adds NetBird claims, see NetBird
`emails/invitation`	An organization invitation with an `email` was created	The invitation is deleted, creation fails	Renders the bundled invitation email
`emails/email-verification`	A user added an email address or asked to resend the verification link	The link is invalidated, the user sees "Could not send the verification link, try again later"	Renders the bundled verification email

There are no other hook points.

The `emails/*` hooks only render content. The backend sends the result over SMTP, see Email hooks and Email configuration.

Delivery

Configuration

Hooks are configured in the backend's config file (see Configuration):

```
{
  "session_secret": "<session-secret>",
  "frontend": {
    "upstream": "unix:/run/pykrete/freshfront.sock",
    "hooks_secret": "<hooks-secret>"
  }
}
```

`upstream` is either `unix:<path>` or `http://<host>:<port>` (plain HTTP, no path). It is the same upstream the backend proxies all unhandled requests (UI pages, docs) to; hooks reuse it.

Give freshfront the same secret in the `HOOKS_SECRET` environment variable; it is read once at startup. freshfront's `HOOKS_SECRET` must match the backend's `frontend.hooks_secret`; otherwise every hook fails, and with it every sign-in and token request. Set both or neither.

Hooks are **disabled** when `frontend` is missing or `hooks_secret` is missing or `null`. Then:

- `identity` and `claims` are skipped (no custom claims).
- Emails can't be rendered. If `email` is configured, sending fails ("sending email requires frontend hooks to render it"). If it isn't, the backend logs the email's raw data, including the link, at `info` level ("email delivery is not configured") and reports success.

Request

```
POST /hooks/claims HTTP/1.1
Host: frontend
Authorization: Bearer <hooks-secret>
Content-Type: application/json

{"grant_type": "authorization_code", ...}
```

Each call opens a new HTTP/1.1 connection to the upstream.

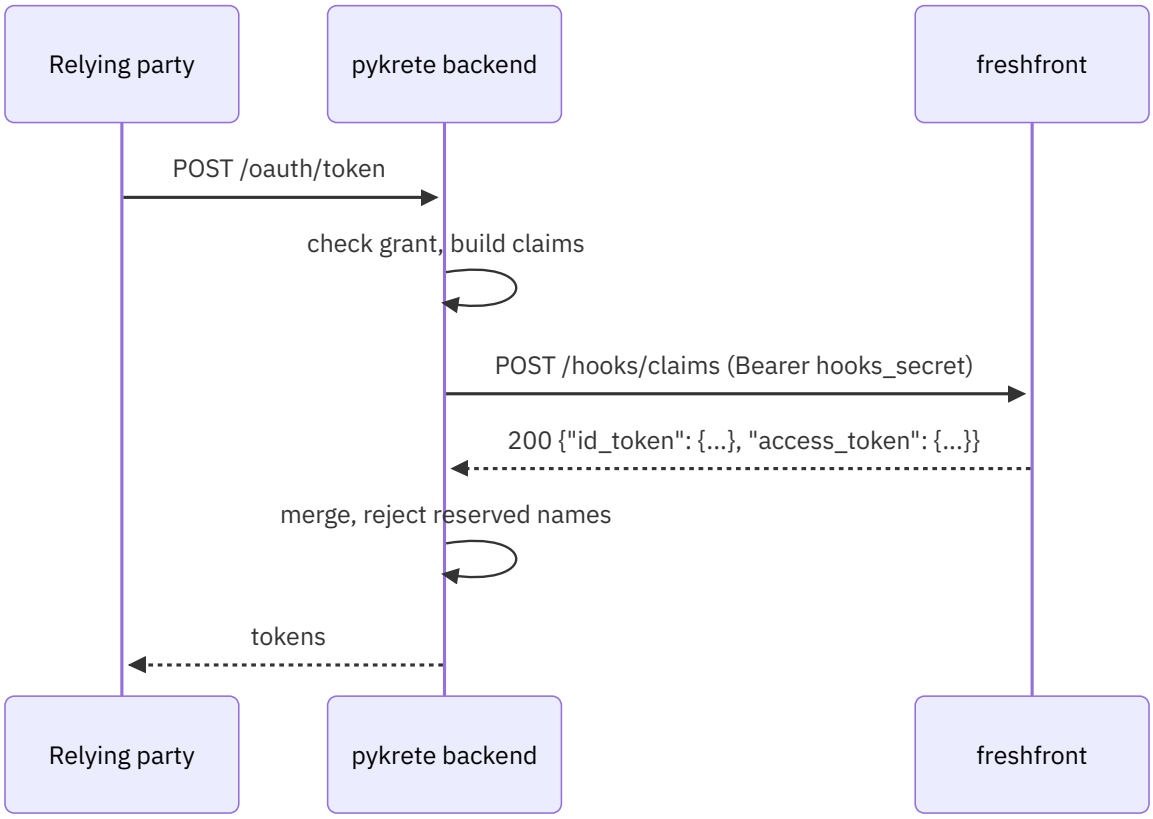
Response and failure

The hook must answer with a `2xx` status and a JSON body within 5 seconds; the body may be at most 1 MiB. `null` is a valid body. Anything else fails the hook:

- a non-`2xx` status (including redirects),
- an empty body (`204 No Content` counts as failure: the body must be JSON),
- invalid JSON or, for `claims` and `emails/*`, JSON of the wrong shape,
- connection errors or the 5 second timeout.

Hooks fail closed: what happens next is listed in the table above. The backend logs the cause, for example `hook: hook returned 404 Not Found`.

The freshfront middleware (`freshfront/routes/hooks/_middleware.ts`) compares the `Authorization` header with `Bearer $HOOKS_SECRET` (constant time) and answers `404` on mismatch or when `HOOKS_SECRET` is unset.



Keeping hooks internal

The backend never proxies `/hooks` or `/hooks/...` from outside: such requests get `404`, also after path normalization (`/x/../hooks/...`, `%2e` segments, backslashes, double slashes).

freshfront's own listener serves `/hooks/*` too. Without `FRESHFRONT_SOCKET` it listens on TCP port 8000 on all interfaces, so run it on a unix socket (`FRESHFRONT_SOCKET=/run/pykrete/freshfront.sock` with `"upstream": "unix:/run/pykrete/freshfront.sock"`) or firewall the port. See Running.

`identity`

Called at the end of every interactive identification, also when the user was already signed in and nothing was shown.

```
{"user_id": "U0123456789abcdefghijklmnopqrstuv"}
```

The response body is ignored (return `null`). A failure aborts the flow: `/authorize` redirects to the client with `error=server_error` ("internal error, try again in a moment"), other flows show "Internal error, try again later". This makes the hook usable as a sign-in gate, for example to block suspended users. It does not run on token refresh; to cut off existing grants as well, fail the `claims` hook for that user.

`claims`

Called by the token endpoint after the grant, scopes and organization were validated and pykrete's own claims were built, and before the tokens are signed. It runs for refresh too, so claims stay fresh.

Request

```
{
  "grant_type": "authorization_code",
  "client_id": "C0123456789abcdefghijklmnopqrstuv",
  "audiences": ["https://api.example.com"],
  "scope": "openid organizations profile",
  "user": {
    "id": "U0123456789abcdefghijklmnopqrstuv",
    "public_metadata": {},
    "private_metadata": {"plan": "enterprise"}
  },
  "organization": {
    "id": "O0123456789abcdefghijklmnopqrstuv",
    "slug": "acme",
    "name": "Acme Inc.",
    "public_metadata": {"tenant": "acme-eu"},
    "private_metadata": {"netbird": {"domain": "acme.netbird.example.com"}},
    "roles": ["role:org-admin"]
  },
  "organizations": ["O0123456789abcdefghijklmnopqrstuv", "O9876543210zyxwvutsrqponmlkjihgfe"]
}
```

Field	Meaning
`grant_type`	`authorization_code`, `refresh_token` or `urn:ietf:params:oauth:grant-type:device_code`
`client_id`	Client the tokens are issued to
`audiences`	Audience **identifiers** (`audienceIdentifier`) of the token's audiences, not the audience ids (`A...`) used in the access token's `aud`
`scope`	Space-separated granted scopes
`user`	The user and their metadata (`{}` when none was set)
`organization`	The active organization (see Organizations) with both metadata objects and the user's role ids in it, or `null`
`organizations`	Ids of the organizations listed in the `orgs` claim. Empty unless the `organizations` scope was granted.

Response

```
{
  "id_token": {"groups": ["role:org-admin"], "tenant": "acme-eu"},
  "access_token": {"tenant": "acme-eu"}
}
```

Both keys are optional; other top-level keys are ignored. `null` or `{}` adds nothing. Values can be any JSON.

Reserved claims

The whole token request fails (fail closed) if the hook returns, for either token:

- a reserved name: `iss`, `sub`, `aud`, `exp`, `iat`, `nbf`, `jti`, `azp`, `scope`, `nonce`, `auth_time`, `at_hash`, `c_hash`, `acr`, `amr`, `sid`, `orgs`;
- any name starting with `org_`;
- a name pykrete already set in that token. In the ID token these are `name`, `picture`, `updated_at` (scope

profile), email, email_verified (scope email), orgs and org_*; in the access token org_*.

The error the client sees is server_error; the backend log says hook: hook tried to set reserved claim <name>.

Where the claims end up

- id_token claims go into the ID token and are returned by /userinfo. They are dropped when the openid scope was not granted (no ID token is issued).
- access_token claims go into the access token JWT. Token introspection does not report them.
- Claims are stored with the issued token; a refresh calls the hook again.

A failed claims hook answers the token request with HTTP 400 and {"error": "server_error", "error_description": "Something is wrong with the server, try again later"}. For the code grant the authorization code is already used up and the user has to sign in again; a refresh token is not rotated, so the client can retry.

Email hooks

pykrete sends email itself, but freshfront writes it. For each email the backend POSTs the data to /hooks/emails/<kind>. freshfront answers with the rendered message, and the backend wraps it into a MIME message and sends it over SMTP. The backend sets From (from email.from), To, Date and Message-ID, so a template can't change the recipient or the sender.

Every request has to, the recipient address, plus the fields of that kind. expires_at is a unix timestamp in seconds. url contains a secret: anyone who gets it can use it, so do not log it.

The response must be:

```
{
  "subject": "Confirm your email address",
  "text": "Open this link to confirm the address: ...",
  "html": "<!DOCTYPE html><html>...</html>"
}
```

- subject: required, non-empty, no control characters (no line breaks).
- text: required, non-empty. Sent as the text/plain part.
- html: optional. When present and non-empty, the message is multipart/alternative with both parts.

A response that breaks these rules fails the hook like any other hook failure.

emails/invitation

Called when an organization invitation with an email is created, by an organization member on /account/organizations/<slug>/invitations, through manage { organization { invite } }, or by an admin via orgs { createOrganizationInvitation }. Link-only and user-bound invitations do not call it. See Invitations.

```
{
  "to": "alice@example.org",
  "organization": {"id": "O0123456789abcdefghijklmnopqrstuv", "slug": "acme", "name": "Acme Inc."},
  "inviter": "Bob",
  "url": "https://auth.example.com/invite#<code>",
  "expires_at": 1793601600
}
```

`inviter` is the display name of whoever created the invitation, or `null`. The invited person has to sign in with an account that has `to` as a verified address. If rendering or sending fails, the invitation is deleted and the creator sees "Could not send the invitation email, try again later"; the cause is only in the backend log.

`emails/email-verification`

Called when a user adds an email address (`me { addEmail }`) or asks for a new link (`me { resendEmailVerification }`), both on the account email page.

```
{
  "to": "alice@example.org",
  "url": "https://auth.example.com/account/emails/verify?code=<code>",
  "expires_at": 1793083200
}
```

The link is valid for 24 hours and only for the signed-in owner. A new link for the same address can be sent at most once a minute, and a user can have at most 5 unverified addresses. If rendering or sending fails, the new link is invalidated.

Customizing hooks

Edit the freshfront handlers

Each hook is a Fresh route in `freshfront/routes/hooks/`: `identity.ts`, `claims.ts`, `emails/invitation.ts`, `emails/email-verification.ts`. The `_middleware.ts` in the same directory already checks the secret, so handlers only deal with the payload.

Example: keep the NetBird mapping and add a `tenant` claim from the active organization's public metadata (`freshfront/routes/hooks/claims.ts`, only the handler changes):

```
export const handler = define.handlers({
   async POST(ctx) {
      const hook: ClaimsHook = await ctx.req.json();
      const netbird = netbirdClaims(hook);
      const tenant = hook.organization?.public_metadata.tenant;
      const extra: Claims = typeof tenant === "string" ? { tenant } : {};
      const output: ClaimsHookOutput = {
         id_token: { ...netbird, ...extra },
         access_token: { ...netbird, ...extra },
      };
      return Response.json(output);
   },
});
```

Keep handlers well under the 5 second budget: the backend gives up after 5 seconds and treats it as a failure.

Change email templates

The email routes only parse the request and call a template from `freshfront/server/emails.tsx`, which exports the request types and one function per email returning `{subject, text, html}`.

The HTML part is baked once when freshfront starts: `Template.bake` from `@delta-docs/build-selfcontained/template` renders the Preact JSX, moves the rules of `freshfront/server/emails.css` into `style` attributes (many mail clients ignore stylesheets), and embeds a Forkex Sans subset holding only the

glyphs the email can show. A request only fills in the placeholders.

- Mark request values with `<Placeholder name="..." expectedCharacters={...} />` in text and `placeholder("...")` in attributes. `expectedCharacters` picks the glyphs that go into the font: the inside of a regex character class (`"a-zA-Z0-9"`), a `RegExp`, or a preset from `chars` (`ascii`, `latin`, `latinExtended`, `cyrillic`, `greek`, `date(format)`). Other characters still show, in the fallback font.
- Values derived from the request (the host of `url`, the formatted expiry) are placeholders of their own: compute them in the email function and pass them to `render`.
- Write colors as `light-dark(<light>, <dark>)`; the palette variables from `@delta-docs/chrome/colors.css` already are. Light values are inlined, dark values go into a `prefers-color-scheme: dark` block that Apple Mail, Outlook for Mac and Thunderbird apply. Gmail and Outlook for Windows ignore it and invert colors on their own.
- Use tables for layout, mail clients handle flexbox and grid poorly.

Every expected character adds a glyph to each weight that can show it. The bundled invitation is about 39 KB with `chars.latin` for names and about 58 KB with `chars.latinExtended`; Gmail clips messages over about 102 KB.

Example: a different invitation subject.

```
export function invitation(email: InvitationEmail): RenderedEmail {
   const org = oneLine(email.organization.name);
   return {
      subject: `Your invitation to ${org}`,
      // ...
   };
}
```

Strip control characters from anything user-controlled that goes into `subject` (the bundled `oneLine` helper does that); a subject with a line break fails the hook. `render` escapes values in the HTML part.

The freshfront dev server lists every email with sample data at `/dev/emails`, with the text part inline and a link to the HTML part. The page answers `404` outside the dev server.

Serve hooks from another service

There is no per-hook URL setting: hooks always go to `frontend.upstream`, which also serves the UI. To run hooks elsewhere, either

- call your service from the freshfront handler, or
- point `upstream` at a reverse proxy that sends `/hooks/*` to your service and everything else to freshfront. Your service then has to check `Authorization: Bearer <hooks-secret>` itself and follow the request and response formats on this page.

A minimal standalone claims hook with Deno:

```
const secret = Deno.env.get("HOOKS_SECRET");

async function digest(value: string) {
   return new Uint8Array(
      await crypto.subtle.digest("SHA-256", new TextEncoder().encode(value)),
   );
}

async function authorized(req: Request) {
   if (!secret) return false;
```

```
  const [a, b] = await Promise.all([
    digest(req.headers.get("authorization") ?? ""),
    digest(`Bearer ${secret}`),
  ]);
  let diff = 0;
  for (let i = 0; i < a.length; i++) diff |= a[i] ^ b[i];
  return diff === 0;
}

Deno.serve({ hostname: "127.0.0.1", port: 9000 }, async (req) => {
  const { pathname } = new URL(req.url);
  if (req.method !== "POST" || !await authorized(req)) {
    return new Response(null, { status: 404 });
  }
  const hook = await req.json();
  switch (pathname) {
    case "/hooks/claims": {
      const plan = hook.user.private_metadata.plan;
      const claims = typeof plan === "string" ? { plan } : {};
      return Response.json({ id_token: claims, access_token: claims });
    }
    case "/hooks/identity":
      return Response.json(null);
    case "/hooks/emails/invitation":
    case "/hooks/emails/email-verification":
      return Response.json({
        subject: "Action needed for your account",
        text: `Open this link: ${hook.url}`,
      });
  }
  return new Response(null, { status: 404 });
});
```

> **NOTE**
>
> A service that replaces freshfront for /hooks/* must answer all four hook names, or sign-in (identity), token issuance (claims) or email fail.

Troubleshooting

Symptom	Likely cause
Every sign-in ends with "Internal error" or error=server_error, backend log shows a hook 404 (hook returned 404 Not Found or Hook(Status(404)))	HOOKS_SECRET in freshfront missing or different from hooks_secret
Token requests fail with server_error, log shows hook tried to set reserved claim <name>	The claims hook returns a reserved name or one pykrete already set
Log shows EOF while parsing a value	A hook answered with an empty body; return null
Log shows hook timed out (or Timeout)	The hook took longer than 5 seconds
Emails never arrive, backend log shows email delivery is not configured	No email in the config file; the backend only logs emails
Sending fails, log shows sending email requires frontend hooks to render it	email is configured but hooks are disabled (no hooks_secret)

Sending fails, log shows `rendered email has an invalid subject`	The template put a line break or other control character into the subject
Sending fails, log shows `smtp: ...` or `smtp timed out`	Wrong `email.smtp` URL or credentials, or the SMTP server is unreachable

LDAP

pykrete ships a small, read-only LDAP server for applications that can't speak OpenID Connect and only offer "log in with LDAP" or "sync users from LDAP". Each application (OAuth client) gets its own directory subtree `o=<client_id>`, sees only the users and groups it has been granted, and users authenticate with a per-app **app password** instead of their main credentials.

If the application supports OpenID Connect, prefer OpenID Connect: LDAP can't do passkeys, consent or app encryption keys.

Transport	LDAPS only (TLS from the first byte) on an address you choose. No plaintext LDAP, no StartTLS.
Operations	Simple bind, search, WhoAmI. Everything that writes is refused.
App account	`cn=service,o=<client_id>,<base_dn>`, password = the client secret
User account	`uid=<username>,ou=people,o=<client_id>,<base_dn>`, password = an app password
Visibility	users and groups of the roles the app was granted with `ldap:search:<role>`

Enabling the server

The LDAP listener is off unless the config file (the JSON file passed with `--config`, see Configuration) has an `ldap` object:

```
{
  "ldap": {
    "listen": "0.0.0.0:636",
    "base_dn": "dc=example,dc=com",
    "certificate": "/var/lib/acme/auth.example.com/fullchain.pem",
    "private_key": "/var/lib/acme/auth.example.com/key.pem"
  }
}
```

Key	Meaning
`listen`	Socket address `ip:port` (`[::]:636` for IPv6). Hostnames are not accepted. There is no default port; 636 is the standard LDAPS port and needs privileges to bind (for example `CAP_NET_BIND_SERVICE`).
`base_dn`	Suffix of every DN, for example `dc=example,dc=com`. Must be a non-empty DN; multi-valued RDNs (+) and #-hex values are rejected.
`certificate`	PEM certificate chain served to LDAP clients (leaf first).
`private_key`	PEM private key for that certificate.

Notes:

- The certificate and key are read once at startup. After renewing the certificate, restart the backend.
- If the `ldap` object is present but the files can't be read, `base_dn` is invalid or `listen` can't be bound, the backend refuses to start.
- TLS 1.2 and 1.3 are offered; client certificates are not requested.
- The listener is separate from the HTTP server, so the HTTP reverse proxy is not involved. If you put a TCP load balancer in front, use TLS passthrough: pykrete terminates TLS itself and doesn't understand the

PROXY protocol.

- Clients that insist on StartTLS (`ldap://` + `-ZZ`, "StartTLS" in app settings) won't work. Configure them for `ldaps://` / "SSL/TLS".

Directory layout

```
dc=example,dc=com                          <base_dn>
`-- o=<client_id>                          one subtree per app; an app only ever sees its own
    |-- cn=service                         bind-only DN of the app itself (no entry)
    |-- ou=people
    |   `-- uid=<username>                 users holding at least one visible role
    `-- ou=groups
        `-- cn=<role_id>                   visible roles
```

An app's view of the root (`<base_dn>` with subtree scope) contains only `<base_dn>` itself and its own `o=<client_id>` subtree. Client ids look like `C` followed by 32 letters and digits and are matched case-sensitively in DNs. The container names `people`, `groups` and `service` and the `base_dn` suffix are matched case-insensitively.

Users

Attribute	Value	Usable in filters
`objectClass`	`top`, `person`, `organizationalPerson`, `inetOrgPerson`	yes
`uid`	current username	yes
`cn`	profile name, or the username if no name is set	yes
`displayName`	same as `cn`	yes
`mail`	every **verified** email address of the user (multi-valued, may be absent)	yes
`memberOf`	DN of every visible group the user is in	equality with a full group DN, presence
`deltaUserId`	pykrete user id (`U...`), stable across username changes	equality, presence

Use `deltaUserId` as the unique/immutable id in application settings: `uid` and the entry DN change when the user renames themselves. There is no `entryUUID`, `sn`, `givenName` or `userPassword`.

Groups

Groups are pykrete roles (see Scopes and claims).

Attribute	Value	Usable in filters
`objectClass`	`top`, `groupOfNames`	yes
`cn`	role id	yes
`displayName`	role name	yes
`description`	role description, omitted when empty	yes

member	DN of every user holding the role	equality with a full user DN, presence

Only roles granted directly to a user count. Roles a user holds only through an organization membership don't appear in LDAP.

Containers and root DSE

- <base_dn>: objectClass: top plus its first RDN as an attribute (for example dc: example).
- o=<client_id>: objectClass: top, organization, o: <client_id>.
- ou=people, ou=groups: objectClass: top, organizationalUnit, ou.
- Root DSE (base "", scope base, readable without binding): namingContexts (the base DN), supportedLDAPVersion: 3, supportedExtension: 1.3.6.1.4.1.4203.1.11.3 (WhoAmI), supportedControl: 1.2.840.113556.1.4.319 (paged results), vendorName: delta.rocks.

Setting up an app

1. **Create a confidential client with a client secret** for the app (see Self-service or Administration). Its client id becomes o=<client_id>, its secret is the service bind password. The client doesn't need any redirect URLs or grant types for LDAP.
2. **Pick the roles the app may see.** Create roles and grant them to users if needed. With the admin GraphQL API (GraphiQL is described in Administration):

```
mutation {
  openid {
    createRole(userRoleId: "staff", name: "Staff", description: "Employees") { userRoleId }
  }
  userdata {
    user(userId: "U...") {
      grantRole(userRoleId: "staff") { userRoleId }
    }
  }
}
```

3. **Make the roles visible to the app.** setClientRoles replaces the app's list of visible roles (unknown role ids are ignored) and returns the resulting list:

```
mutation {
  ldap {
    setClientRoles(clientId: "C...", userRoleIds: ["staff"]) { userRoleId name }
  }
}
```

   Check the current list with query { ldap { clientRoles(clientId: "C...") { userRoleId name } } }.

4. **Issue app passwords** for the users who will log in through the app (next section).

How visibility works: for each role, setClientRoles creates a scope ldap:search:<role_id> on the identity provider's own audience (named "LDAP search: <role name>") and adds it to the client's allowed scopes. The LDAP server treats every ldap:search:* allowed scope of the client as a visible role. The app sees:

- users who hold at least one visible role,
- the visible roles as groups,
- for each user, only the memberOf values of visible roles.

After changing the client's allowed scopes, check `clientRoles` and run `setClientRoles` again if roles are missing.

App passwords

An app password is a random secret for one user in one app. Binding with the main password would mean handing it to every LDAP application, and accounts that sign in with passkeys only have no password at all. App passwords are per-app, revocable, and useless for pykrete's own sign-in.

Format	32 random letters and digits, shown once when issued
Storage	SHA-256 of the password; the plaintext is not kept
Scope	one `(user, client)` pair, identified by a `name` that is unique per user and client
Requirement	the user must hold at least one role visible to the client, otherwise the bind fails

Issue, list and revoke them with the admin GraphQL API:

```
mutation {
  ldap {
    issueAppPassword(userId: "U...", clientId: "C...", name: "laptop")
  }
}
```

```
query { ldap { appPasswords(userId: "U...") { clientId name createdAt } } }

mutation { ldap { revokeAppPassword(userId: "U...", clientId: "C...", name: "laptop") } }
```

`issueAppPassword` returns the password as a string. Hand it to the user over a secure channel.

Users see their app passwords (name, app and creation date) on the **App passwords** page of the account dashboard (`/account/app-passwords`).

> **NOTE**
> Users can't issue or revoke app passwords themselves yet; the dashboard page is read-only.

Binding

Identity	Bind DN	Password
Anonymous	empty	empty
App (service)	`cn=service,o=<client_id>,<base_dn>`	client secret of that client
User	`uid=<username>,ou=people,o=<client_id>,<base_dn>`	an app password issued for that user and client

- Only simple binds are supported. SASL binds get `authMethodNotSupported`.
- A non-empty DN with an empty password is refused with `unwillingToPerform` (no "unauthenticated" binds).
- Wrong DN or password gives `invalidCredentials`. The service bind requires the client to use a client secret.
- The `uid` in a bind DN is matched exactly, including case. Returned entries carry the current username.

- A failed bind leaves the connection anonymous.

What each identity can search:

Anonymous	Root DSE only. Any other search gets `insufficientAccessRights`.
App (service)	Everything in its own `o=<client_id>` subtree: all visible users and groups.
User	The containers and its **own** user entry only (including its `memberOf`); no groups, no other users.

Search bases outside the app's own subtree (another app's `o=...`, or `cn=service`) return `noSuchObject`.

Examples

The examples use these shell variables; replace the client id with yours.

```
CLIENT_ID=Cxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
BASE="o=$CLIENT_ID,dc=example,dc=com"
# -y reads the whole file as the password, so write it without a trailing newline:
printf '%s' 'client-secret-here' > client-secret.txt
```

If the certificate is not from a public CA, point the OpenLDAP tools at your CA with `LDAPTLS_CACERT=/path/to/ca.pem`.

Read the root DSE anonymously:

```
ldapsearch -H ldaps://auth.example.com -x -b "" -s base '(objectClass=*)'
```

List all users the app can see:

```
ldapsearch -H ldaps://auth.example.com -x \
  -D "cn=service,$BASE" -y client-secret.txt \
  -b "ou=people,$BASE" -s one '(objectClass=inetOrgPerson)' \
  uid displayName mail memberOf deltaUserId
```

```
dn: uid=alice,ou=people,o=Cxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,dc=example,dc=com
uid: alice
displayName: Alice Example
mail: alice@example.com
memberOf: cn=staff,ou=groups,o=Cxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,dc=example,dc=com
deltaUserId: Uxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Find one user by login name, the usual "search, then bind as the user" first step:

```
ldapsearch -H ldaps://auth.example.com -x -D "cn=service,$BASE" -y client-secret.txt \
  -b "ou=people,$BASE" '(&(objectClass=person)(uid=alice))' 1.1
```

Users in a group, and members of a group:

```
ldapsearch -H ldaps://auth.example.com -x -D "cn=service,$BASE" -y client-secret.txt \
  -b "ou=people,$BASE" "(memberOf=cn=staff,ou=groups,$BASE)" uid

ldapsearch -H ldaps://auth.example.com -x -D "cn=service,$BASE" -y client-secret.txt \
  -b "cn=staff,ou=groups,$BASE" -s base '(objectClass=*)' member
```

Page through large result sets (simple paged results control):

```
ldapsearch -H ldaps://auth.example.com -x -D "cn=service,$BASE" -y client-secret.txt \
  -E pr=200/noprompt -b "ou=people,$BASE" '(objectClass=person)' uid
```

Check a user's app password:

```
ldapwhoami -H ldaps://auth.example.com -x \
  -D "uid=alice,ou=people,$BASE" -y app-password.txt
# dn:uid=alice,ou=people,o=Cxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,dc=example,dc=com
```

Typical application settings

Most applications ask for the same handful of values:

Setting	Value
Server / port	`auth.example.com`, the port from `listen`
Encryption	LDAPS / "SSL/TLS" (implicit TLS), not StartTLS
Bind DN / manager DN	`cn=service,o=<client_id>,dc=example,dc=com`
Bind password	the client secret
User base DN	`ou=people,o=<client_id>,dc=example,dc=com`
User filter	`(&(objectClass=inetOrgPerson)(uid=%s))` (placeholder syntax depends on the app)
Login / username attribute	`uid`
Unique id attribute	`deltaUserId`
Email / display name	`mail` / `displayName`
Group base DN	`ou=groups,o=<client_id>,dc=example,dc=com`
Group filter / object class	`(objectClass=groupOfNames)` / `groupOfNames`
Group membership	`member` on groups (user DN), or `memberOf` on users

Users then log into the application with their username and an app password issued for that application.

Supported operations

Operation	Behaviour
Bind	simple only, see Binding
Search	scopes base, one, subtree and children; `typesOnly`; attribute selection
Extended: WhoAmI (`1.3.6.1.4.1.4203.1.11.3`)	returns `dn:<bound DN>`, empty when anonymous
Other extended operations (StartTLS, password modify, …)	`protocolError`
Add, Modify, Delete, ModifyDN, Compare	`unwillingToPerform`
Abandon	ignored
Unbind	closes the connection
Controls	simple paged results (`1.2.840.113556.1.4.319`) only

Search details:

- Attribute selection: an empty list, `*` or `+` returns all attributes; `1.1` returns none. Attribute names are case-insensitive and the aliases `userid`, `commonName`, `rfc822mailbox`, `organizationName`, `organizationalUnitName` and `domainComponent` are understood.
- Filters: and, or, not, equality, substring and presence. Approximate match (`~=`) behaves like equality. Matching is case-insensitive.
- Ordering (`>=`, `<=`) and extensible-match filters, and filters on attributes the server doesn't know, evaluate to Undefined, which stays Undefined under `!`, so they never match. A presence test on an unknown attribute is plain false.
- `memberOf` and `member` filters need the full DN of a group or user in the same app subtree.
- At most 1000 entries per search, including when paging; a smaller client size limit is honoured. Larger results are cut off with `sizeLimitExceeded`.
- Connections: 10 s for the TLS handshake, closed after 5 minutes without a request, requests up to 128 KiB.

Security notes

- An app's service account can read the username, display name, verified emails and visible roles of every user holding a visible role. Grant only the roles the app needs.
- App passwords are bound to one app: a password issued for app A doesn't work in app B's subtree. Revoke them when a device or installation is retired.
- Removing a user's last visible role locks them out of the app's LDAP immediately; their app passwords stay stored until revoked.
- The service bind password is the OAuth client secret. Rotating the secret breaks the app's LDAP bind until its configuration is updated.
- There is no bind rate limiting or lockout. Restrict the LDAP port with a firewall to the hosts that need it.

App encryption keys (E2E)

An app can ask pykrete for a per-user, per-app 256-bit symmetric key to encrypt the user's data end-to-end. The key is derived in the user's browser from a secret that only the user's passkey (or password) can unlock, and it reaches the app encrypted to a public key the app supplied. The pykrete server stores only wrapped material and never sees the keys.

The scheme follows Firefox Accounts scoped keys: request a scope, pass `keys_jwk`, receive `keys_jwe` from the token endpoint.

How it works

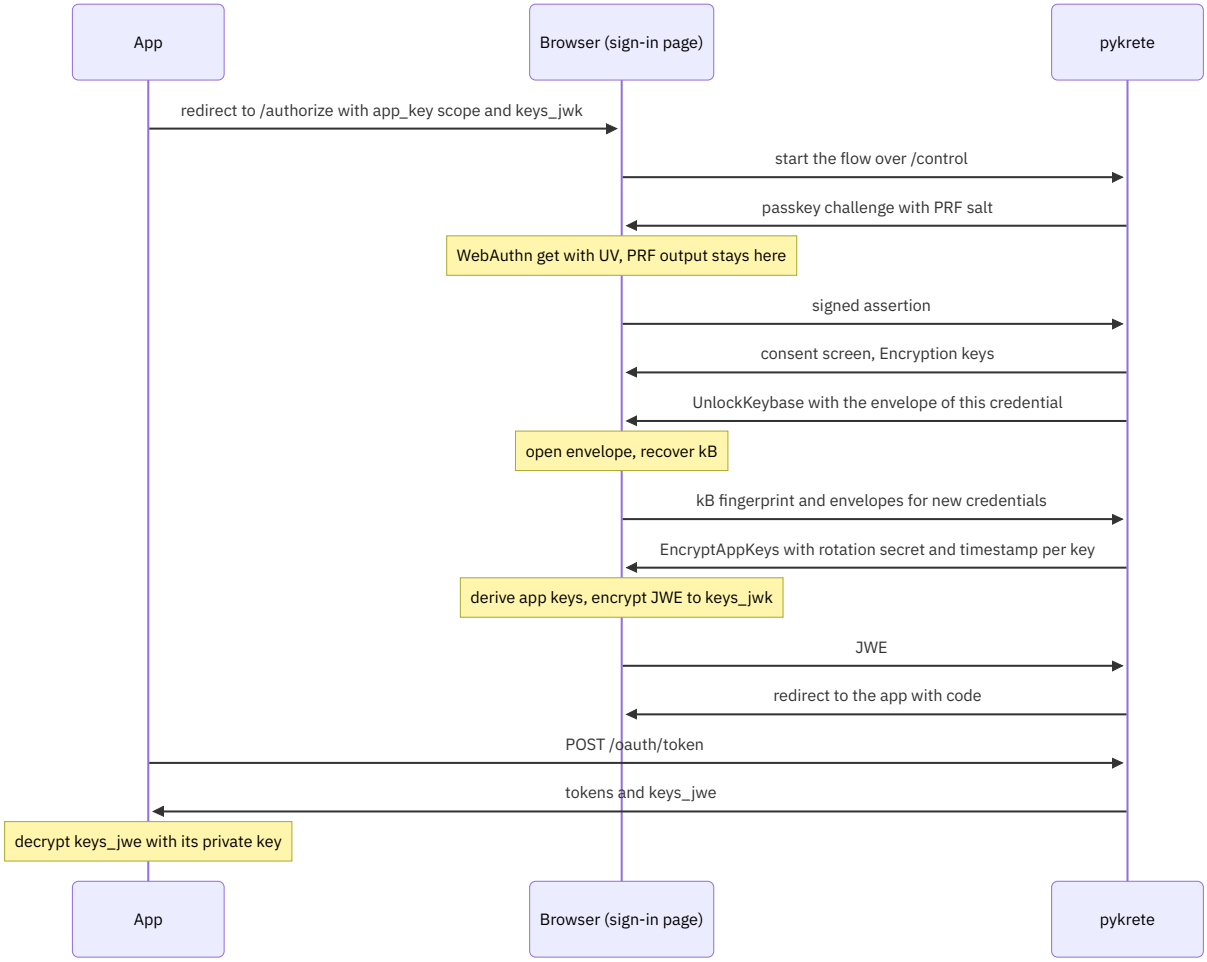
Three pieces of key material are involved:

Material	What it is	Where it lives
Keybase (`kB`)	32 random bytes per user, generated in the browser the first time keys are needed. Never leaves the browser unwrapped.	Server stores one **envelope** per credential (`kB` encrypted with a key derived from that credential) and a fingerprint of `kB`.
Rotation secret	32 random bytes per user and app key scope, generated by the server on first use.	Server (`app_keys` table). Useless without `kB`.
App key	Derived from `kB` and the rotation secret.	Only in the browser (briefly) and in the app.

Unlocking `kB` needs **keybase material** from the credential the user signs in with:

- Passkey: the output of the WebAuthn PRF extension (CTAP2 `hmac-secret` on security keys).
- Password: a value derived from the password in the browser (never sent to the server).

All derivation runs in a WebAssembly worker on pykrete's sign-in page. The server only receives envelopes, the fingerprint and, at the end, a JWE encrypted to the app.



The server checks the fingerprint on every unlock, so a credential can't silently introduce a different `kB`.

Derivation reference

`uid` below is the pykrete user id (`U...`), not necessarily the `sub` the app sees.

```
material (passkey)  = PRF(eval.first = "deltarocks-keybase-prf-v1")                32 bytes
material (password) = derived from the password in the browser                    32 bytes

wrap_key    = HKDF-SHA256(salt = uid, ikm = material, info = "deltarocks-keybase-envelope-v1", 32 bytes)
envelope    = 0x01 || nonce(12) || AES-256-GCM(wrap_key, nonce, kB, aad = 0x01)    1 + 12 + 32 + 16 bytes
fingerprint = HKDF-SHA256(salt = uid, ikm = kB, info = "deltarocks-keybase-fingerprint", 32 bytes)

info = "identity.mozilla.com/picl/v1/scoped_key\n" || "app_key:" || scope_id
info = info || "\n" || client_id                                    (client-bound scopes only)

okm = HKDF-SHA256(salt = uid, ikm = kB || rotation_secret, info = info, 48 bytes)
kid = <rotation timestamp, unix seconds> || "-" || base64url(okm[0..16])
k   = okm[16..48]
```

For an ordinary scope every client allowed to request it gets the same key, as with Firefox Accounts. For a **client-bound** scope the requesting client's id is part of the derivation, so each client gets its own key from the same scope. The rotation secret is shared, so rotating it changes the keys of all clients at once.

Setting up an app key scope

An app key scope has an id, a name and a description (shown on the consent screen and in the dashboard), an owner, and the client-bound flag. Each client that may request it must be allowed explicitly.

1. Create the scope under **Account → OIDC → App key scopes** (`/account/app-key-scopes`): pick the owner (yourself, or an organization where you hold `org:audiences`), the scope id, name and description, and whether it is client-bound.
2. On the client's **App keys** tab (`/account/clients/<client_id>/app-keys`), tick the scope and save.

The same is available through GraphQL (`manage` root): `createAppKeyScope`, `appKeyScopes`, `appKeyScope(scopeId:) { update delete }` and, on a client, `allowedAppKeyScopes` and `setAllowedAppKeyScopes`.

```
mutation {
  manage {
    createAppKeyScope(scopeId: "notes", name: "Notes encryption key",
                      description: "Encrypts your notes end-to-end") { scopeId }
    client(clientId: "C...") { setAllowedAppKeyScopes(scopeIds: ["notes"]) { scopeId } }
  }
}
```

- The scope id must not contain spaces or start with `app_key:`, and is at most 128 characters. The OAuth scope is `app_key:` followed by the id, here `app_key:notes`. URL-style ids such as `https://notes.example.com/` work too. Ids are global: the first owner to create an id holds it.
- The id and the client-bound flag can't be changed after creation; both are part of the key derivation.
- Only the scope's owner (or an administrator) can allow a client to request it. A client's editor can always remove a scope from the client.
- Use a client-bound scope when several clients should each get a separate key, for example one client per server sharing one `luks` scope. Use an ordinary scope when several clients of the same app must read the same data.
- Keys are delivered by the authorization code flow and by CIBA. The device code grant refuses `app_key:` scopes.
- Audience scopes can't start with `app_key:`, so app keys can't be shadowed by an ordinary audience scope.
- Scopes created before ownership existed have no owner; only administrators manage them.

> **⚠ WARNING**
>
> Deleting an app key scope (**Delete app key scope**, or `delete` in GraphQL) deletes every user's rotation secret for it, and the keys can never be derived again. Data encrypted with them is lost. The page shows how many users have a key before you confirm.

Requesting keys

For each authorization:

1. Generate a fresh ECDH key pair on curve P-256 and keep the private key with the PKCE verifier.
2. Put the **public** JWK, JSON-encoded and base64url-encoded without padding, into the `keys_jwk` parameter.
3. Add `app_key:<id>` to `scope` (several app key scopes can be requested at once).

```
import { base64url, exportJWK, generateKeyPair } from "jose";
```

```
const { publicKey, privateKey } = await generateKeyPair("ECDH-ES", { crv: "P-256" });
const keysJwk = base64url.encode(JSON.stringify(await exportJWK(publicKey)));

const url = new URL("https://auth.example.com/authorize");
url.search = new URLSearchParams({
  client_id: "C...",
  response_type: "code",
  redirect_uri: "https://notes.example.com/callback",
  scope: "openid app_key:notes",
  state,
  code_challenge,
  code_challenge_method: "S256",
  keys_jwk: keysJwk,
}).toString();
```

The rest is a normal authorization code exchange, see OpenID Connect.

What to expect:

- The consent screen always shows an **Encryption keys** group for app key scopes, even if the user approved the app before: key access is never remembered as consent.
- The user always has to confirm with a passkey or password, even with an active session, because the server has no way to unlock kB without fresh keybase material. `prompt=none` therefore can't deliver keys (it fails with `login_required` or `consent_required`).
- A client that isn't allowed the scope gets `error=invalid_scope` ("client has no access to app key ...") on its redirect URI, after the user signed in.
- If none of the user's credentials can unlock the keybase, the redirect carries `error=invalid_request` with "none of the passed challenges can unlock the keybase".

> **IMPORTANT**
> `keys_jwk` is required when app key scopes are requested, and must be an EC P-256 public key. Never put the private part (`d`) into the URL.

What the app receives

The token response of the authorization code exchange carries `keys_jwe` next to the usual fields:

```
{
  "access_token": "eyJ...",
  "token_type": "Bearer",
  "id_token": "eyJ...",
  "expires_in": 3600,
  "scope": "app_key:notes openid",
  "keys_jwe": "eyJhbGciOiJFQ0RILUVTIiwiZW5jIjoiQTI1NkdDTSIsImVwayI6ey4uLn19..<iv>.<ciphertext>.<tag>"
}
```

`keys_jwe` is a compact JWE (RFC 7516) encrypted to the `keys_jwk` of that authorization:

- Protected header: `{"alg":"ECDH-ES","enc":"A256GCM","epk":{"crv":"P-256","kty":"EC","x":"...","y":"..."}}`. Direct key agreement, so the encrypted-key part is empty (note the `..`). No `apu`/`apv`.
- Plaintext: a JSON object with one JWK per requested app key scope, keyed by the full scope:

```
{
  "app_key:notes": {
    "kid": "1700000000-LeqpY753aIAKrzda4tSnlg",
```

```
    "kty": "oct",
    "k": "<base64url, 32 bytes>"
  }
}
```

Decrypt with any JOSE library:

```
import { compactDecrypt } from "jose";

const { plaintext } = await compactDecrypt(tokens.keys_jwe, privateKey);
const keys = JSON.parse(new TextDecoder().decode(plaintext));
const notesKey = keys["app_key:notes"];
```

- `k` is the raw 256-bit key, for example for AES-256-GCM. The same user and scope always yield the same key, on any device.
- `kid` is `<timestamp>-<fingerprint>`: the creation time of the rotation secret (unix seconds) and a 16-byte fingerprint of the key. Store it with your ciphertexts to detect key changes.
- `keys_jwe` is only part of the authorization code exchange and of the CIBA token response. Refresh-token responses keep `app_key:...` in `scope` but carry no keys, and the device-code and client-credentials grants never deliver keys. Keep the key in the app, or run a new authorization to fetch it again.
- The JWE is created in the user's browser. The server stores it with the authorization code (valid for 10 minutes, single use) or the CIBA request, and deletes both on exchange. Without the app's private key the JWE is unreadable.

Keys without a browser (CIBA)

A machine with no browser, such as a server that needs a disk key at boot, gets keys through backchannel authentication: it sends `keys_jwk` with the backchannel request, the user approves on pykrete, and the token response carries `keys_jwe`, exactly as above.

- The user unlocks the keybase with a passkey or password on the approval page, every time. pykrete can't release a key without the user.
- `keys_jwk` is checked when the request arrives (EC P-256, valid point, no private part), so a broken key is rejected before the user sees anything.
- Use a fresh key pair per request, generated on the machine, and keep the private key in memory only.
- Combine with a client-bound scope and one client per machine: each machine then gets its own key, and one machine's key reveals nothing about the others.
- Everything the machine stores to authenticate (its client secret) can be read by anyone with access to its disk, who can then send requests in its name. The approval screen shows the binding message, the address the request came from and the time; deny requests you didn't expect.

Rotation

There is no rotation API yet.

- An app key is created on the first authorization that requests it and stays the same afterwards.
- Deleting the user's row in `app_keys` makes the next authorization create a new rotation secret, so a new key with a new `kid`. The app must still hold the old key to re-encrypt its data; pykrete can't recreate it.
- `kB` itself never changes. Adding a credential only adds an envelope.

Passkeys, PRF and credentials without PRF

- User verification is required on every passkey registration and sign-in. PRF (`hmac-secret`) output differs with and without user verification, so allowing both would produce different keys. Security keys therefore need a PIN; without one, registration and sign-in fail with an error ending in "security keys need a PIN set".
- Passkeys are discoverable credentials using ES256 (P-256); other algorithms are rejected.
- The PRF input is the fixed salt `deltarocks-keybase-prf-v1`.

Which credential can unlock the keybase:

Credential	Can unlock
Password	yes
Passkey with PRF and an envelope	yes
Passkey with PRF, no envelope yet	if a keybase already exists, not on its own: after the user also confirms with a credential that can unlock, the browser wraps kB for this passkey too ("Confirm once with another method to let this passkey unlock encryption keys")
Passkey without PRF	never, the passkey only signs in

When the user signs in with a passkey that returns no PRF output, the server asks for another credential that can unlock (password or a PRF passkey) in the same flow. If there is none, the authorization fails with the `invalid_request` error above.

The very first time keys are needed (no keybase yet), a password or any PRF passkey creates kB, and every credential used in that flow gets an envelope. Accounts migrated from the older password-only keybase format are converted to envelopes on their next unlock.

What users see

- The consent screen: the heading mentions "your encryption keys" and an **Encryption keys** group lists the scope names and descriptions.
- **E2EE** page of the account dashboard (`/account/app-keys`): one entry per app key the user has (name, description, creation date).
- **Pending sign-ins** (`/account/pending`): backchannel requests waiting for approval, with **Review** and **Deny**.
- **Sign-in methods** (`/account/sign-in`): an "unlocks encryption keys" badge on each credential that has an envelope, and "End-to-end encryption keys set up" with the kB fingerprint and the number of envelopes.
- **Add passkey** (`/add-passkey`): if the new passkey supports PRF but didn't return output at creation, the page offers "Enable encryption keys" (use the passkey once more) or "Skip, use it for sign-in only". Afterwards the sign-in methods page shows "Passkey added", "Passkey added. Sign in with it and confirm once with another method to let it unlock encryption keys" or "Passkey added. This authenticator can't unlock encryption keys".
- **Access tokens** (`/account/tokens`): tokens list their app key scopes as `app key: <id>`.

Security notes

- The server never receives kB, PRF output, password-derived material or app keys in clear. It stores envelopes, the fingerprint, rotation secrets and, until the code is exchanged, the JWE.
- The derivation code is served by the same pykrete deployment. End-to-end here protects against database leaks and passive server compromise, not against an operator who serves modified sign-in code.
- An envelope made from a password is only as strong as the password: anyone with a database dump can test password guesses against it offline. Passkeys with PRF don't have this problem.
- There is no recovery. If a user loses every credential that has an envelope, kB and all app keys derived from it are gone.
- Use a fresh `keys_jwk` key pair per authorization and drop the private key once `keys_jwe` is decrypted.

SAML and proxy authentication

Neither is available yet. For applications that need single sign-on, use OpenID Connect; for applications that only speak LDAP, use the built-in LDAP server.

SAML

pykrete does not act as a SAML identity provider yet: there are no SAML endpoints and no metadata to import into a service provider.

If an application only supports SAML, put a SAML-to-OIDC bridge in front of it and register the bridge as an ordinary OIDC client. Configure the bridge with the issuer `https://auth.example.com/`.

Proxy / forward authentication

Forward authentication for reverse proxies (nginx `auth_request`, Traefik `forwardAuth`, Envoy `ext_authz`) is not available yet. Don't point a proxy at `/proxy-auth`.

To protect applications without their own login, use an OIDC-aware reverse proxy or authentication middleware, registered in pykrete as a normal confidential client (see OpenID Connect) and configured with the issuer `https://auth.example.com/`. The proxy handles the sign-in redirect and passes the user on from the ID token or userinfo claims.

Backend API, API keys, introspection

Server-to-server surface for first-party relying parties. The REST part is a subset of the Clerk Backend API (2026-08-20); token checks use the OAuth standards. Anything without a Clerk or Auth0 equivalent lives in GraphQL.

Enabling a client

Two GraphQL mutations on `manage { client(clientId: ...) { ... } }`, both on the client settings page:

`setAllOrganizations(enabled: Boolean!)`

Admin only ("All organizations (first-party)" section). Eligible organizations of a user are all of their memberships. No per-organization client access (`oauth_client_organizations`), no organization consent, no access-management screen; `prompt=consent` does not show it either. Covers the authorize flow, the `orgs` claim, refresh re-checks and the account pages. Use `organization_usage=none` plus the `organizations` scope for a silent `orgs` list.

`setBackendApi(enabled: Boolean!)`

Anyone who can manage the client (owner, or `org:clients` for organization clients; "Backend API" section). Enables the REST API below for this client's secret, which then acts for every user who authorized the client. Requires a confidential client with a client secret and `subject_type=public`, `subject_scope=global` (so sub is the `U...` user id). Checked when enabled and on every request.

Backend REST API

Base `<root>/api/v1`, so a Clerk SDK works with `apiUrl = <root>/api`. Auth: `Authorization: Bearer <client secret>`. Pagination `limit`/`offset` → `{"data": [...], "total_count": n}`. Timestamps are unix milliseconds. Errors: `{"errors": [{"message", "long_message", "code"}]}` with `authentication_invalid` (401), `authorization_invalid` (403), `resource_not_found` (404), `bad_request` (400), `form_param_format_invalid` / `form_param_missing` / `form_identifier_exists` (422), `too_many_keys` (409).

Visibility

Invisible users, organizations and keys are 404.

	Regular client	**`allOrganizations` client**
Users	consented to the client (any identity or audience scope)	consented, or a member of any organization
Organizations (/ `organizations`, org key subjects)	enabled the client (`oauth_client_organizations`)	all
Memberships of a user	eligible and granted to the client by that user (normal organization access)	all memberships
`POST /organizations`	`created_by` required and visible; the new organization enables the client and records the creator's grant	`created_by` optional

Organization API keys	organization visible; `created_by` visible, holds `org:api-keys` and granted the organization to the client	same, every membership counts as granted

Endpoint	Notes
`GET /users/{user_id}`	Clerk `User` subset
`GET /users/{user_id}/organization_memberships`	Memberships granted to this client
`GET /organizations`	`limit`, `offset`, `query` (name or slug)
`GET /organizations/{organization_id}`	id or slug
`POST /organizations`	`name`, `slug?`, `created_by` (see visibility), `public_metadata?`, `private_metadata?`; `created_by` gets `role:org-admin`; slug derived from the name when omitted
`POST /api_keys`	`type?`, `name`, `description?`, `subject`, `scopes?`, `claims?`, `created_by?`, `seconds_until_expiration?`; returns `secret` once
`GET /api_keys`	`subject` (required), `type?`, `include_invalid?`, `limit`, `offset`, `query`
`GET, PATCH, DELETE /api_keys/{id}`	`PATCH`: `description`, `claims`, `scopes`, `seconds_until_expiration`
`POST /api_keys/{id}/revoke`	`revocation_reason?`; soft revoke
`POST /api_keys/verify`	`secret`; 404 unless active and issued to this client

Organization API keys need `created_by` with the `org:api-keys` organization permission (granted to `role:org-admin`). User API keys may only be created by their subject (or without `created_by`).

Differences from Clerk

- Ids are pykrete ids: users `U...`, organizations `O...`, API keys `K...` (letter + 32 alphanumerics), not `user_` / `org_` / `ak_`. Clerk's OpenAPI patterns reject them; the SDKs do not check.
- API key secrets are `DELTA-API-KEY-<48 alphanumerics>` and only their SHA-256 is stored, so `GET /api_keys/{id}/secret` is not offered.
- A member can hold several pykrete roles: `role` is the primary one (`role:org-admin`, else `role:org-member`, else the first), `permissions` is the union of `org:*` permissions of all of them.
- `User` lacks phone numbers, wallets, external accounts and sign-in timestamps (empty or null). Email ids (`E...`) and membership ids (`M...`) are derived hashes. `Organization.updated_at` equals `created_at`.
- Keys are bound to the client that created them; other clients see them as nonexistent.

Introspection and revocation

`POST /oauth/introspect` (RFC 7662) and `POST /oauth/revoke` (RFC 7009), advertised in discovery. Client authentication: HTTP Basic (form-urlencoded id and secret, RFC 6749 2.3.1) or `client_id`/`client_secret` in the body; the client must be confidential. Body: `token`, `token_type_hint?`, form or JSON.

Only tokens issued to the calling client are reported; everything else is exactly `{"active": false}`.

API key:

```
{"active": true, "kind": "api_key", "token_type": "Bearer", "iss": "<issuer>",
 "client_id": "C...", "api_key_id": "K...", "name": "...", "scope": "a b",
 "sub": "U..." | "org_id": "O...", "created_by": "U...", "claims": {...},
 "iat": 1700000000, "exp": 1700003600}
```

Access token: `kind: "access_token"`, `iss`, `client_id`, `sub`, `scope`, `aud` (string for one audience, else array), `iat`, `exp`, `org_id` and `org_roles` when an organization was active. Times are unix seconds.

Revocation always answers 200: API keys are soft-revoked, access tokens deleted (with their refresh token), refresh tokens revoke their chain.

03

Administration

Admin access 80

Account dashboard 89

Self-service 95

Admin access

Administrators manage everything that is not owned by a regular user or organization: system clients and audiences, global roles, users, organizations, LDAP visibility. Most of it is done through the admin GraphQL API; the account dashboard covers the parts that overlap with self-service (clients, audiences, organizations).

Who is an admin

An admin is a user holding the built-in role `role:admin` ("Admin"). The server makes sure it exists on every start and gives it one scope: `admin:graphql` on the server's own audience (the audience whose identifier is the issuer URL, `https://auth.example.com/`).

Holding the role is not enough on its own. Admin rights belong to **tokens**: only an access token that carries `admin:graphql` is treated as an admin. Your normal dashboard session is an ordinary user session, even if you hold the role. There are two ways to get an admin token, described below.

- The first admin has to be bootstrapped outside the UI, see Running pykrete.
- Further admins: grant them `role:admin` (see Create a role and grant it to a user). Any role that includes `admin:graphql` on the server's own audience works the same way.

Two ways in

GraphiQL

Open https://auth.example.com/graphiql (a `GET /graphql` redirects there).

1. The page starts a sign-in with the built-in "GraphiQL" client. If you already have a session you are sent right back, otherwise you sign in as usual. The first time, you confirm the "GraphQL admin" permission.
2. The GraphiQL client always requires the `admin:graphql` scope. If your account has no role that grants it, the authorization fails and the toolbar shows the error with a "Sign in again" button.
3. After sign-in the console loads the schema and is ready.

The console has a query editor, a variables editor (JSON) and a response pane. "Run" (or Ctrl+Enter) runs the operation under the cursor, "Schema" shows the full SDL. Query and variables are kept in the browser's local storage; the token is kept in memory only, so a reload signs in again.

> **NOTE**
>
> The GraphiQL token carries only `admin:graphql`, not the `account` scope. The `me` and `myOrganizations` roots answer "account scope required" there. Everything else, including `manage`, works.

Admin mode in the dashboard

The account header shows "Enter admin mode" when you hold a role that grants `admin:graphql` (`me.canEnterAdminMode`). Clicking it:

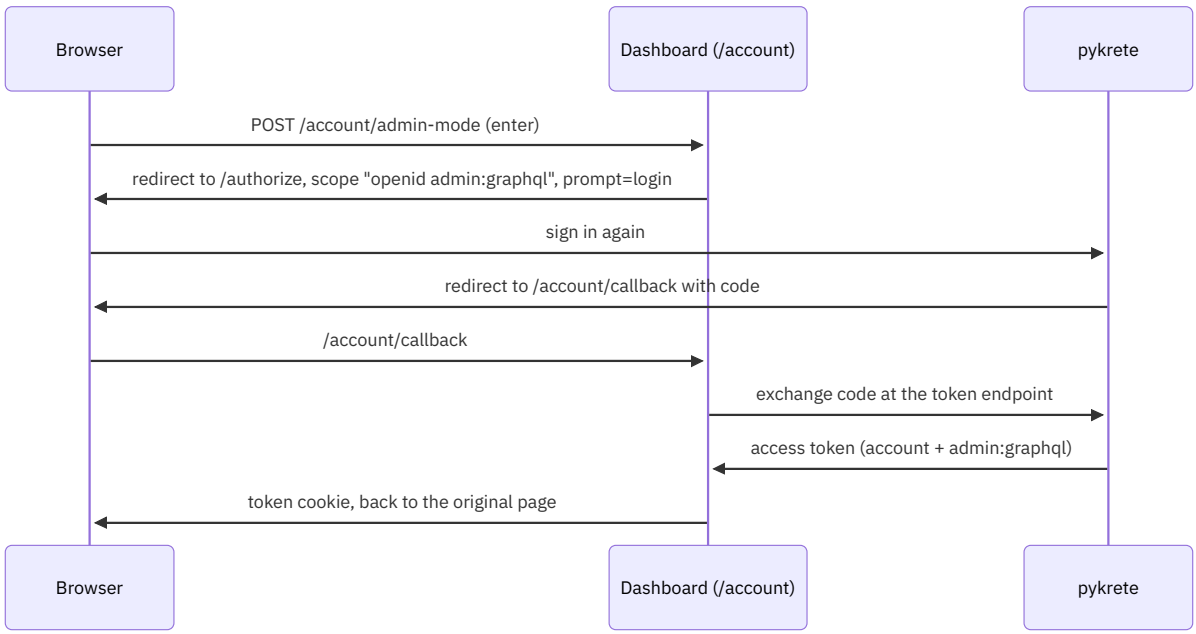
1. ends your current sign-in session and asks you to sign in again (`prompt=login`), so a passkey or password is required even if you were signed in a second ago;
2. requests `openid admin:graphql` for the dashboard's built-in "Account" client, so the new token carries

both `account` and `admin:graphql`;

3. returns you to the page you were on, with an "Admin mode" badge and an "Exit admin mode" button in the header.

If you cancel the sign-in the dashboard shows "Admin mode was cancelled"; if the account can't get the scope it shows "Admin mode is not available for this account". In both cases the dashboard keeps using your normal token.

"Exit admin mode" revokes the admin token (`me.revokeCurrentToken`) and signs you in again without the scope. Admin mode also ends on its own when the token expires (access tokens live one hour); the next page load signs you in normally.



What changes while admin mode is on:

- Every client, audience, organization and organization role is visible and editable, and the escalation guards described in Self-service do not apply.
- Client settings pages get a "Mark as verified" / "Remove verification" button and an "All organizations (first-party)" section.
- Dashboard lists show your own clients, audiences and organizations. Open any other one by its URL, for example `https://auth.example.com/account/clients/<client id>`, `https://auth.example.com/account/audiences/<audience id>` or `https://auth.example.com/account/organizations/<slug>`. The admin API lists all of them: `openid { clients { clientId name } }`, `openid { audiences { audienceId name } }`, `orgs { organizations { id slug name } }`. Organizations you are not a member of show an "admin view" badge.
- Global roles, users and LDAP settings are managed through the admin GraphQL API.

The admin GraphQL API

Endpoint: `POST` https://auth.example.com/graphql with `Authorization: Bearer <access token>`. An unknown or expired token gets HTTP 401; a token with neither `admin:graphql` nor `account` gets 403.

Mutations mirror the query tree: you navigate to an object and call a mutation on it, inside a `mutation` operation. Read fields are available inside mutations too.

```
mutation {
  openid {
    client(clientId: "C...") {
      setAllowedGrantTypes(grantTypes: [AUTHORIZATION_CODE, REFRESH_TOKEN])
    }
  }
}
```

In the schema, object types that contain mutations appear twice: once without mutation fields for queries (`OpenidRoot`, `Client`) and once with a `Mut` suffix for mutations (`OpenidRootMut`, `ClientMut`).

Root	Needs	Covers
`openid`	admin	All clients and audiences (including system-owned ones), their redirect URLs, post-logout URLs, grant types, allowed and required scopes, client secrets and subject settings; audience scopes; global roles and the scopes they grant.
`userdata`	admin	Users: list and look up (by id or username), create, grant and revoke global roles, issue registration codes.
`orgs`	admin	All organizations: create, update (including private metadata), delete, members and their roles, invitations; which organizations a client may see, client organization usage, access requests; user metadata.
`ldap`	admin	Which roles a client can see over LDAP, app passwords. See LDAP.
`manage`	account or admin	The self-service API behind the dashboard (organizations, clients, audiences, roles). With an admin token, ownership checks are bypassed and admin-only mutations (`setVerified`, `setAllOrganizations`, `setOrganizationAssignable`, `organizations(all: true)`) are allowed.
`me`	account	The signed-in user's own account, see Account dashboard.
`myOrganizations`	account	The signed-in user's memberships and per-application organization access.

Calling an admin root without `admin:graphql` fails with the error "admin:graphql scope required".

Recipes

Ids in the examples are placeholders: users look like `U...`, clients `C...`, audiences `A...`, organizations `O...`. Organizations can be referenced by id or slug wherever an argument is called `organization`.

Create a user

There is no self sign-up: accounts are created by admins. Create the user, then issue a one-time registration code:

```
mutation {
  userdata {
    createUser(username: "alice", name: "Alice Example", email: "alice@example.com") {
      userId
    }
  }
```

```
}
```

```
mutation {
  userdata {
    user(userId: "U...") {
      issueRegistrationCode
    }
  }
}
```

Send the code to the user over a trusted channel. On their first sign-in they enter the username, then the "Secret code" and a new password. Afterwards they can add passkeys from the dashboard. The email is stored unverified; the user can verify it under Emails.

`issueRegistrationCode` refuses users that already have any sign-in method; it is meant for new accounts only.

To let a user back in after they lost their authenticator app, YubiKey and recovery codes, remove all their second factors:

```
mutation {
  userdata {
    user(userId: "U...") {
      removeSecondFactors
    }
  }
}
```

It returns how many credentials were removed. The user then signs in with the password alone and can set up a new second factor.

To look users up:

```
query {
  userdata {
    users(usernames: ["alice"]) {
      userId
      username
      emails { email emailVerified }
      roles { userRoleId name }
    }
  }
}
```

Create a client

This creates a system-owned client (no user or organization owner). System-owned clients can only be changed by admins.

```
mutation {
  openid {
    createClient(
      name: "Wiki"
      description: "Team wiki"
      clientKind: CONFIDENTIAL
      usesClientSecret: true
      usesPkce: true
    ) {
      clientId
    }
```

```
  }
}
```

Then configure redirect URLs, grant types and a secret on the new client:

```
mutation {
  openid {
    client(clientId: "C...") {
      setRedirectUrls(urls: [
        { url: "https://wiki.example.com/oauth/callback", public: true }
      ]) {
        redirectUrl
        public
      }
      setPostLogoutRedirectUrls(urls: ["https://wiki.example.com/"]) {
        redirectUrl
      }
      setAllowedGrantTypes(grantTypes: [AUTHORIZATION_CODE, REFRESH_TOKEN])
      resetClientSecret
    }
  }
}
```

- All `set...` mutations replace the whole list.
- Redirect URLs must be `http` or `https`; query and fragment are dropped. `public: true` marks the default URL used when a request has no `redirect_uri`; keep at most one public URL.
- Grant types: `AUTHORIZATION_CODE`, `REFRESH_TOKEN`, `DEVICE_CODE`. `IMPLICIT` is rejected. The client credentials grant (`CLIENT_CREDENTIALS`) is not available yet.
- `resetClientSecret` returns a new `DELTA-CLIENT-SECRET-...` value and invalidates the old one. There is no field to read a secret back, so store it now.
- Public clients (`clientKind: PUBLIC`, `usesClientSecret: false`) use PKCE only.

Finally mark it verified (see Mark an app verified). A system-owned, verified client is "first-party": the consent screen shows no publisher line and it can't be deleted from the dashboard.

> **TIP**
> Creating the client from the dashboard (`/account/clients`, "New client") is quicker, but the client is then owned by your own user account, not by the system. Dashboard-created clients start with the authorization code and refresh token grants and PKCE on.

Endpoint and flow details for the application side are in OpenID Connect.

Create an audience (API) and its scopes

```
mutation {
  openid {
    createAudience(
      name: "Wiki API"
      description: "Wiki REST API"
      audienceIdentifier: "https://wiki.example.com/api"
      signatureType: RS256
    ) {
      audienceId
    }
  }
}
```

```
mutation {
  openid {
    audience(audienceId: "A...") {
      read: declareScope(scopeId: "wiki:read", name: "Read the wiki", description: "Read pages") {
        scopeId
      }
      write: declareScope(scopeId: "wiki:write", name: "Edit the wiki", description: "Create and edit pages") {
        scopeId
      }
    }
  }
}
```

`declareScope` also updates name and description of an existing scope; `removeScope(scopeId:)` deletes one. Clients pass the identifier as the `audience` parameter. Use `RS256` or `ES256` as `signatureType`; other algorithms are not supported.

Scope ids are at most 128 printable ASCII characters without spaces, double quotes or backslashes, not a built-in scope name (`openid`, `profile`, `email`, `offline_access`, `created_at`, `organizations`) and don't start with `app_key:`.

Allow the scopes on a client (this replaces the client's allowed list; `setRequiredScopes` works the same way for scopes that are always added):

```
mutation {
  openid {
    client(clientId: "C...") {
      setAllowedScopes(scopes: [
        { audienceId: "A...", scopeId: "wiki:read" }
        { audienceId: "A...", scopeId: "wiki:write" }
      ]) {
        audienceId
        scopeId
      }
    }
  }
}
```

> **NOTE**
>
> Allowing a scope on the client is only half of it. A user only receives an audience scope if one of their roles grants it; otherwise the authorization fails with `invalid_scope`. Continue with the next recipe. See Scopes and claims for the full rules.

Create a role and grant it to a user

```
mutation {
  openid {
    createRole(
      userRoleId: "role:wiki-editor"
      name: "Wiki editor"
      description: "Can read and edit the wiki"
    ) {
      userRoleId
    }
  }
}
```

The role id is free-form; the built-in roles use a `role:` prefix. Attach scopes (replaces the role's list):

```
mutation {
  openid {
    role(userRoleId: "role:wiki-editor") {
      setScopes(scopes: [
        { audienceId: "A...", scopeId: "wiki:read" }
        { audienceId: "A...", scopeId: "wiki:write" }
      ]) {
        audienceId
        scopeId
      }
    }
  }
}
```

Grant it (use `revokeRole` with the same argument to take it away):

```
mutation {
  userdata {
    user(userId: "U...") {
      grantRole(userRoleId: "role:wiki-editor") {
        userRoleId
        name
      }
    }
  }
}
```

To make someone an admin, grant `role:admin` the same way. Users see their global roles under `/account/roles`.

A global role can also be offered to organizations: once it is organization-assignable, organization admins can give it to their members, and its scopes apply while that organization is the active one.

```
mutation {
  manage {
    role(roleId: "role:wiki-editor") {
      setOrganizationAssignable(assignable: true)
    }
  }
}
```

Enable organizations for a client

Three settings decide what a client sees of organizations; Organizations explains the effect on the sign-in flow and on tokens.

1. Organization usage: `NONE` (default), `OPTIONAL` (users pick an organization or their personal account) or `REQUIRED`.

   ```
   mutation {
     orgs {
       setClientOrganizationUsage(clientId: "C...", usage: OPTIONAL)
     }
   }
   ```

   The same field is the "Organizations" select on the client's Settings tab in the dashboard.

2. Which organizations may see the client. This replaces the full list and returns organization ids:

   ```
   mutation {
     orgs {
       setClientOrganizations(clientId: "C...", organizations: ["acme", "O..."])
   ```

```
  }
}
```

Organization admins can also do this themselves on their organization's "Client access" page.

3. Or, for your own first-party apps, let the client see every organization of every user, with no per-organization access and no organization consent:

```
mutation {
  manage {
    client(clientId: "C...") {
      setAllOrganizations(enabled: true)
    }
  }
}
```

Approve organization access requests

When a client is not enabled for an organization, members can request access on the sign-in screen. List all pending requests:

```
query {
  orgs {
    accessRequests {
      clientId
      organizationId
      userId
      createdAt
    }
  }
}
```

Approve (enables the client for the organization and clears every pending request for that pair):

```
mutation {
  orgs {
    approveOrganizationAccessRequest(clientId: "C...", organization: "acme")
  }
}
```

Deny with `denyOrganizationAccessRequest(clientId:, organization:, userId:)`; leave out `userId` to drop all requests for the pair. Organization admins with the `org:access` permission handle their own requests on `/account/organizations/<slug>/access`.

Mark an app verified

Until a client is verified, its consent screen shows the publisher (the owning user or organization) with an "Unverified app" badge. Verifying needs an admin token:

```
mutation {
  manage {
    client(clientId: "C...") {
      setVerified(verified: true)
    }
  }
}
```

In the dashboard: enter admin mode, open `/account/clients/<client id>`, click "Mark as verified". To find unverified clients:

```
query {
  openid {
    clients {
      clientId
      name
      verified
      ownerUserId
      ownerOrganizationId
    }
  }
}
```

LDAP visibility and app passwords

The built-in LDAP server only shows a client the members of roles you assign to it (this replaces the list):

```
mutation {
  ldap {
    setClientRoles(clientId: "C...", userRoleIds: ["role:wiki-editor"]) {
      userRoleId
      name
    }
  }
}
```

Read the current list with `ldap { clientRoles(clientId: "C...") { userRoleId name } }`. Internally each role becomes an allowed `ldap:search:<role id>` scope of the client on the server's own audience.

Users bind to LDAP with app passwords, not their main password. Users can't create them yet; issue one for them (the password is returned once):

```
mutation {
  ldap {
    issueAppPassword(userId: "U...", clientId: "C...", name: "Mail on laptop")
  }
}
```

List with `appPasswords(userId:)`, revoke with `revokeAppPassword(userId:, clientId:, name:)`. Users see their app passwords (without the secret) under `/account/app-passwords`. Server setup and bind DNs are in LDAP.

Account dashboard

Every user has a self-service area at https://auth.example.com/account. It shows what the server knows about you, which apps you gave access to, and lets you manage organizations, clients and audiences you own.

Signing in

Opening `/account` starts a normal sign-in with the built-in "Account" app. The sign-in screen offers:

- a username field (browsers may suggest saved passkeys right in it), or
- "Sign in with a passkey" without typing a username.

After entering a username you get "Sign in with a passkey" and, if you have a password, "Use password instead".

First sign-in

There is no self sign-up. An administrator creates your account and gives you a username and a one-time secret code (see Admin access). On the first sign-in, enter the username, then the "Secret code" and a new password. Add a passkey afterwards.

Passkeys

Use "Add passkey" in the dashboard header (it opens `/add-passkey`) and click "Create passkey". If the authenticator supports it, the passkey can also unlock your end-to-end encryption keys: you are asked to use it once more ("Enable encryption keys"), or you can choose "Skip, use it for sign-in only". The Sign-in methods page then tells you what happened, for example "Passkey added. Sign in with it and confirm once with another method to let it unlock encryption keys", or "This authenticator can't unlock encryption keys".

Why security keys need a PIN

pykrete requires **user verification** for every passkey: a fingerprint, face, device unlock or, on a hardware security key, a PIN. A key that only checks "user present" (a touch) is rejected with "authenticator did not verify the user, security keys need a PIN set".

This is deliberate. A passkey replaces your password and can unlock your encryption keys through the WebAuthn PRF extension. Authenticators compute a different PRF secret with and without user verification, so allowing both would break the encrypted key envelopes, and touch-only would make unlocking your keys possession-only.

> **TIP**
> Set a FIDO2 PIN on the key with the vendor's tool, or with `fido2-token -S <device>` from libfido2, then add the passkey again.

Layout

The header shows your picture (or initial), name, `@username` and member-since date, and the buttons "Add passkey" and "Sign out". Admins also see "Enter admin mode" there (see Admin mode). The sidebar groups the pages; numbers next to entries are counts (for example active access tokens).

Group	Page	Path
Account	Profile	`/account`
Emails	`/account/emails`	
Sign-in methods	`/account/sign-in`	
Sessions	`/account/sessions`	
Pending sign-ins	`/account/pending`	
Organizations	`/account/organizations`	
Access	Applications	`/account/applications`
Access tokens	`/account/tokens`	
App passwords	`/account/app-passwords`	
API keys	`/account/api-keys`	
E2EE	`/account/app-keys`	
Roles	`/account/roles`	
OIDC	Clients	`/account/clients`
Audiences	`/account/audiences`	
App key scopes	`/account/app-key-scopes`	

Account pages

Profile

Shows your user ID (with a copy button; apps and admins use it to refer to you), name, when the profile was last updated, member-since date, every username you had (current first) and your email addresses.

You can upload or remove your picture (at most 2 MiB). Name and username can't be changed from the dashboard yet.

Emails

Lists your addresses with a "verified" or "unverified" badge and, for unverified ones, whether a verification link is pending.

- "Send verification link" adds an address and emails a link. The link opens `/account/emails/verify?code=...`, where "Confirm email" verifies it. Links are valid for 24 hours and only work for the account that requested them.
- "Resend link" sends a new one, at most once a minute.
- "Remove" deletes the address.
- You can have at most five unverified addresses at a time, and an address verified by another account can't be verified again.

The server emails the link when email is configured (see Email). Without it, no email is sent and the link only appears in the server log.

Verified addresses matter for organization invitations sent by email: only an account with that verified address can accept them.

Sign-in methods

Lists your credentials: "Password", "Passkey" (with a short id), "Password setup code" (an unused registration code), "Authenticator app", "YubiKey OTP" and "YubiKey OTP (YubiCloud)" (both with the key's public ID). A badge marks credentials that can unlock your encryption keys, and a summary line shows the encryption key fingerprint and how many credentials wrap it.

Add passkeys with "Add passkey" in the header. Second factors can be removed with "Remove"; passwords and passkeys can't be removed from the dashboard yet.

Two-factor authentication

With a second factor set up, signing in with a password also asks for a code. Signing in with a passkey doesn't, because passkeys already require a PIN or biometric check. Once a code was accepted, the browser session doesn't ask again.

The code field takes any of:

- the 6-digit code from an authenticator app,
- a code typed by touching a YubiKey,
- one of your recovery codes.

After five wrong codes the sign-in stops and has to be started again. Each code works only once.

"Add authenticator app" shows a QR code and the setup key. Scan it (or tap "open it in the app" on a phone), then enter the code the app shows to finish. The setup page works for 15 minutes.

"Add YubiKey" registers a Yubico OTP slot, in one of two ways depending on the server configuration.

When the server uses YubiCloud, touch the key once to type a code into the field. This works with the credential most YubiKeys ship with in slot 1. Every code is then checked with Yubico's YubiCloud service, so Yubico learns when the key is used here.

When the server has `otp_secret`, you can instead program a slot yourself and have codes checked by the server alone. Program a slot with YubiKey Manager, paste the private ID and secret key it prints, then touch the key to type a code into the last field:

```
ykman otp yubiotp --serial-public-id --generate-private-id --generate-key 2
```

Slot 2 is triggered by a long touch. Slot 1 usually holds Yubico's factory credential, which other sites may rely on.

Adding the first second factor creates ten recovery codes, shown once. Each works once in place of a code. "New recovery codes" replaces them all. Removing the last authenticator app or YubiKey also deletes the recovery codes. If you lose all of them, an administrator can remove your second factors (see Lost second factor).

Authenticator apps and self-programmed YubiKeys need `otp_secret` on the server, YubiCloud keys need `yubicloud` (see Configuration). With neither, two-factor authentication isn't offered.

Sessions

One entry per browser where you are signed in to pykrete, with the time it started and the sign-in methods passed in it (for example "Passkey"). Ending a single session from here is not available yet; "Sign out" ends the session of the current browser (see Signing out).

Pending sign-ins

At the top, requests from machines waiting for your approval (backchannel requests, see CIBA), marked "waiting for you". Each shows the app, its message (for example "nas1: unlock 2 disks"), the address the request came from, when it was made and expires, and the requested scopes.

- "Review" opens the approval page. Sign in (with a passkey or password if the request includes encryption keys), check the message, address and time shown above the permissions, then "Allow" or "Deny". If you are signed in with another account, the page asks you to sign in as the account the request is for.
- "Deny" rejects the request right away.

Approve only requests you expect. A request you didn't start can mean someone has a copy of the machine's credentials.

Below them, sign-ins that are not finished yet: authorization codes issued to an app that the app has not exchanged, and device sign-ins you approved that the device has not picked up. Read-only. Each entry shows when it was created and when it expires; expired entries are marked "expired".

Organizations

- **Invitations** addressed to your account, with "Accept" and "Decline".
- **Your organizations** with the permissions you have in each.
- **New organization**: a slug (lowercase letters, digits and dashes), a name and an optional picture. You become its admin.

Everything inside an organization is described in Self-service.

Access pages

Applications

Every app you granted access to, grouped by what it can see: "Your identity" (built-in permissions such as profile and email) and one group per API (audience) with the scopes you approved.

Revoking a grant from here is not available yet.

Which of your organizations an app may see is chosen on the sign-in screen, when the app uses organizations: "Visible to this app" toggles them, "Continue as" picks the active one, and "Not enabled for this app" lets you request access for organizations that haven't enabled the app. There is no dashboard page for these decisions yet; see Organizations for when the screen appears again.

Access tokens

All access tokens issued to you, newest first; expired ones are folded away. Each entry shows the app, a short id, the grant type, issue and expiry time, scopes, audiences, the refresh token state and, under "Claims", the access and ID token claims. "this page" marks the token the dashboard itself is using. Tokens can't be revoked from here.

App passwords

Passwords for apps that sign you in over LDAP, with the app and creation date. Administrators issue and revoke them; creating your own is not available yet. See LDAP.

API keys

API keys that apps created for you through the Backend API: name, prefix, app, who created it, last use, expiry, scopes and state. "Revoke" disables a key immediately; revoked and expired keys stay listed under a fold. New keys are only created by the apps that use them, see Backend API.

E2EE

App encryption keys: one per key name, created the first time an app is issued an `app_key:...` scope for you. Shows name, description and creation date. Read-only. See App encryption keys.

Roles

Global roles an administrator granted you, with their id and description. Roles you hold inside an organization are shown on that organization's Members page.

OIDC pages

"Clients", "Audiences" and "App key scopes" are for developers: register your own OIDC clients (apps), APIs and end-to-end encryption key scopes, personally or for an organization. See Self-service and App encryption keys.

Admin mode

Users with an admin role see "Enter admin mode" in the header. It asks you to sign in again and then shows an "Admin mode" badge and "Exit admin mode". Details in Admin access.

Signing out

"Sign out" in the header always removes the dashboard's own token. If the browser is signed in to pykrete, it then asks:

Choice	Result
Stay signed in	Only the dashboard is signed out. Your pykrete session stays, so the next visit to `/account` or any app signs you in without asking for a passkey or password.
Sign out of pykrete	Ends the pykrete session in this browser. The next sign-in anywhere asks for a passkey or password again.

Either way you end up on the server's start page. Apps you are signed in to keep their own sessions and tokens; sign out in each of them. When you end your pykrete session from an app's sign-out, the dashboard's token is removed too.

Self-service

Any signed-in user can create organizations, OIDC clients (apps), audiences (APIs) and app key scopes, without an administrator. This page is for developers and organization admins: who owns what, what each permission allows, and which guards stop people from giving themselves more rights than they have.

Everything here is done in the account dashboard (`https://auth.example.com/account/organizations`, `https://auth.example.com/account/clients`, `https://auth.example.com/account/audiences`). The same operations are available in the `manage` GraphQL root, which the dashboard calls with your session token; the examples below show them. Calling `manage` yourself needs a token with the `account` scope of the server's own audience (the dashboard's) or an admin token, see Admin access.

Ownership

Every client, audience and app key scope has exactly one owner. App key scopes follow the audience rules (`org:audiences`).

Owner	Who can see it	Who can change it	Notes
You (personal)	you	you	Pick "Personal" as owner when creating. Deleted together with your account.
Organization	every member, read-only	members with `org:clients` (clients) or `org:audiences` (audiences)	Pick the organization as owner; you need that permission in it. The organization can't be deleted while it owns clients or audiences.
System	admins only	admins only	Created through the admin API. The dashboard shows them as "system" (verified) or "unowned" (unverified).

In GraphQL the owner is `owner { kind id name }` with `kind` one of `USER`, `ORGANIZATION`, `SYSTEM`. Transferring a client or audience to another owner is not available yet.

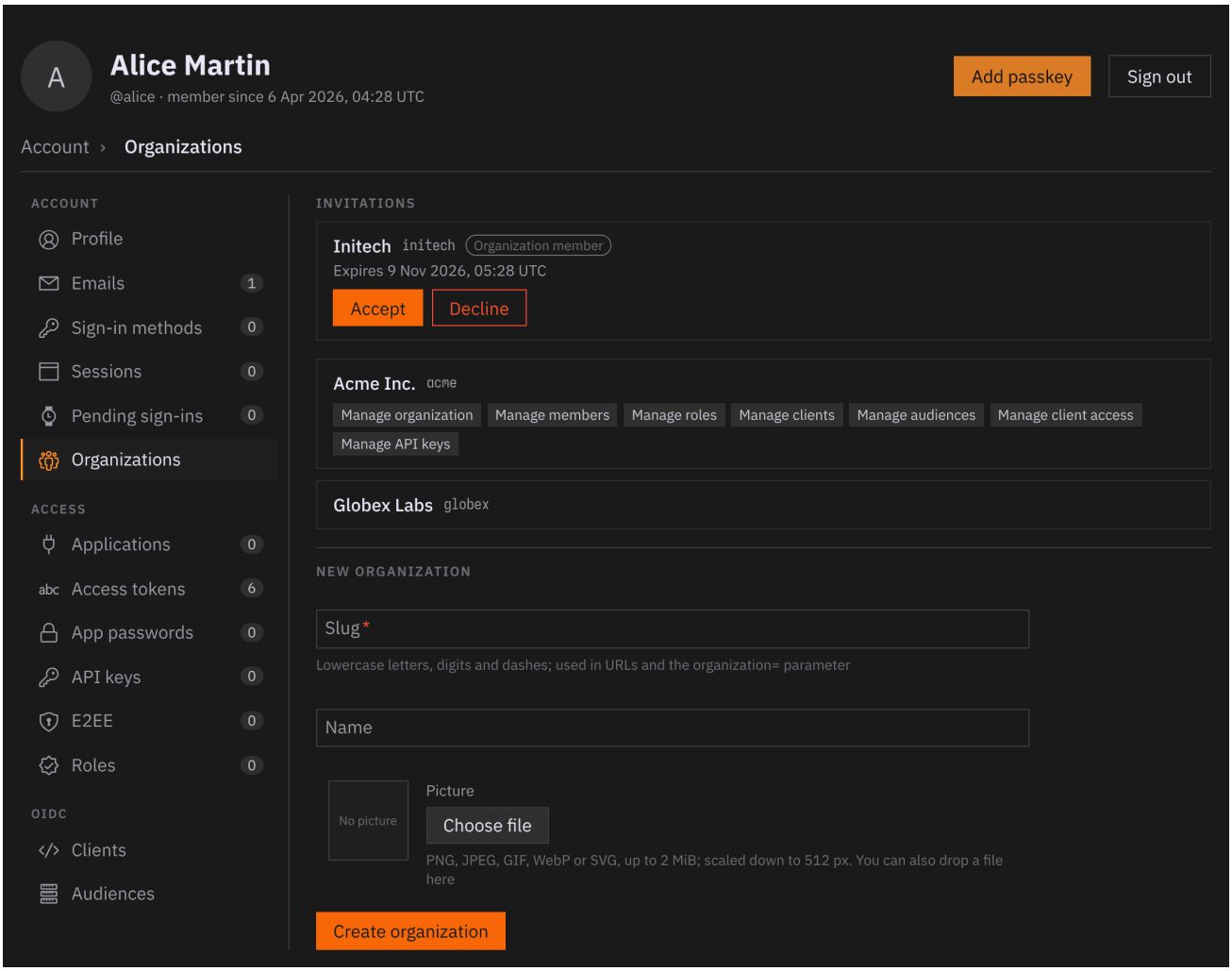
Organizations

Create one

In the dashboard: Organizations, "New organization". In GraphQL:

```
mutation {
  manage {
    createOrganization(slug: "acme", name: "Acme Inc.") {
      id
      slug
      viewerPermissions
    }
  }
}
```

The slug is 1 to 63 lowercase letters, digits and inner dashes. It appears in dashboard URLs and in the `organization=` authorization parameter, so renaming it breaks old links. You become a member with the `role:org-admin` role.

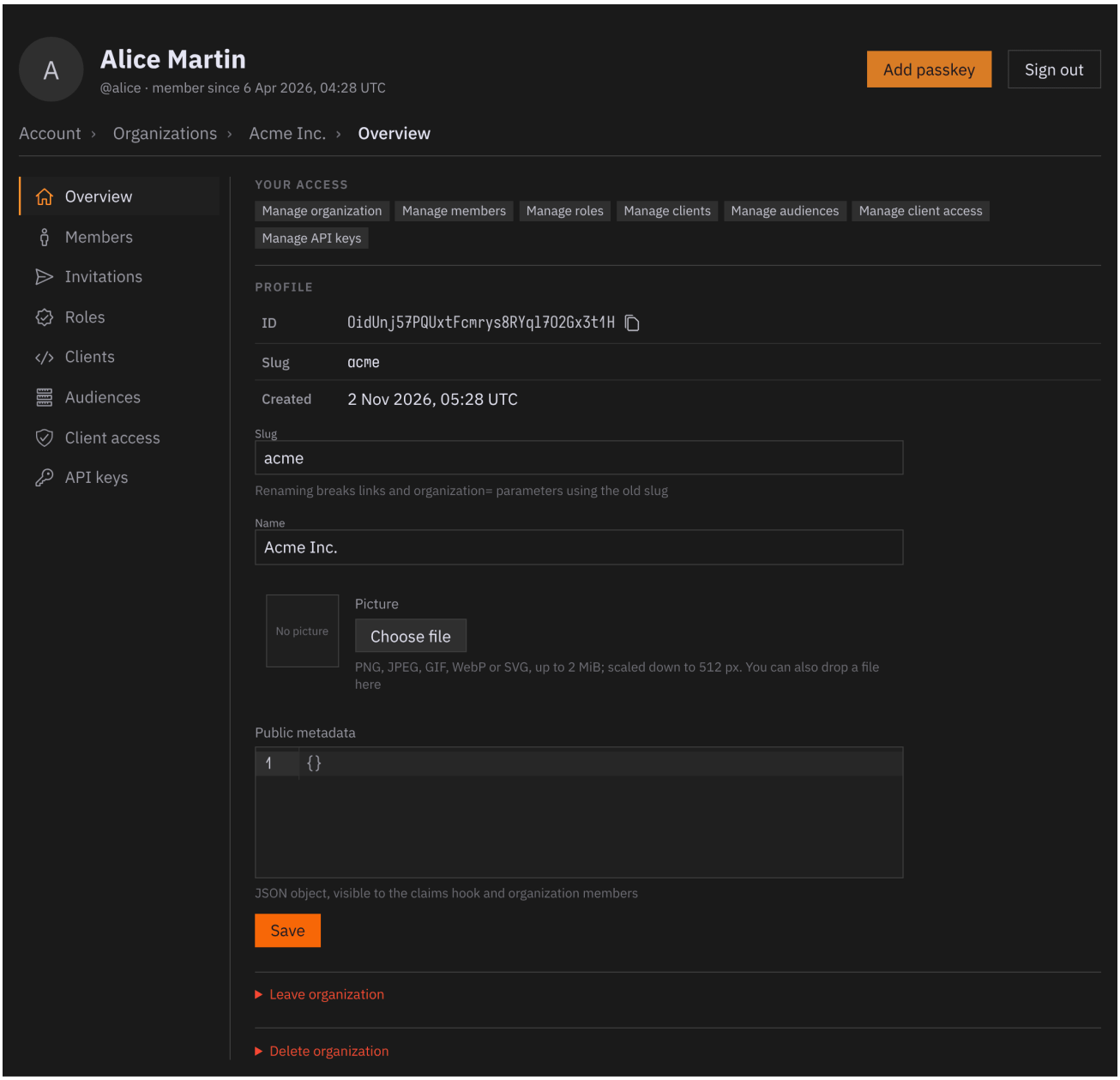


Organizations: a pending invitation, your organizations with your permissions, and the create form

An organization's pages are at `/account/organizations/<slug>`: Overview, Members, Invitations, Roles, Clients, Audiences, Client access and API keys. Some tabs only show up with the matching permission (see What is hidden and what is disabled).

Overview

Shows your permissions, the organization id, slug and creation date. With `org:manage` you can edit slug, name, picture and public metadata (a JSON object of at most 8 KiB, visible to the claims hook and to members), and delete the organization. Deleting fails with "delete or transfer organization clients, audiences and app key scopes first" while it still owns any. The organization's API keys are deleted with it. Members can "Leave organization" here.



An organization's Overview tab

Permissions and roles

Organization permissions are scopes of the server's own audience. Members get them through roles, and the server checks them against the database on every management request.

Scope	GraphQL	Dashboard label	Allows
`org:manage`	`MANAGE`	Manage organization	Edit slug, name, picture and public metadata; delete the organization.
`org:members`	`MEMBERS`	Manage members	Invite (by username, email or link), revoke invitations, change members' roles, remove members.
`org:roles`	`ROLES`	Manage roles	Create organization roles, edit their names and grants, delete them.
`org:clients`	`CLIENTS`	Manage clients	Create and edit clients owned by the organization.

`org:audiences`	`AUDIENCES`	Manage audiences	Create and edit audiences owned by the organization and their scopes; add their scopes to clients.
`org:access`	`ACCESS`	Manage client access	Decide which clients can see the organization; approve or deny access requests.
`org:api-keys`	`API_KEYS`	Manage API keys	See and revoke organization API keys. Also required from the user named as creator when an app creates one.

Two roles exist in every organization:

- `role:org-admin` ("Organization admin"): all seven permissions. The creator gets it.
- `role:org-member` ("Organization member"): no permissions. Members can still see the organization, its members, roles, clients and audiences.

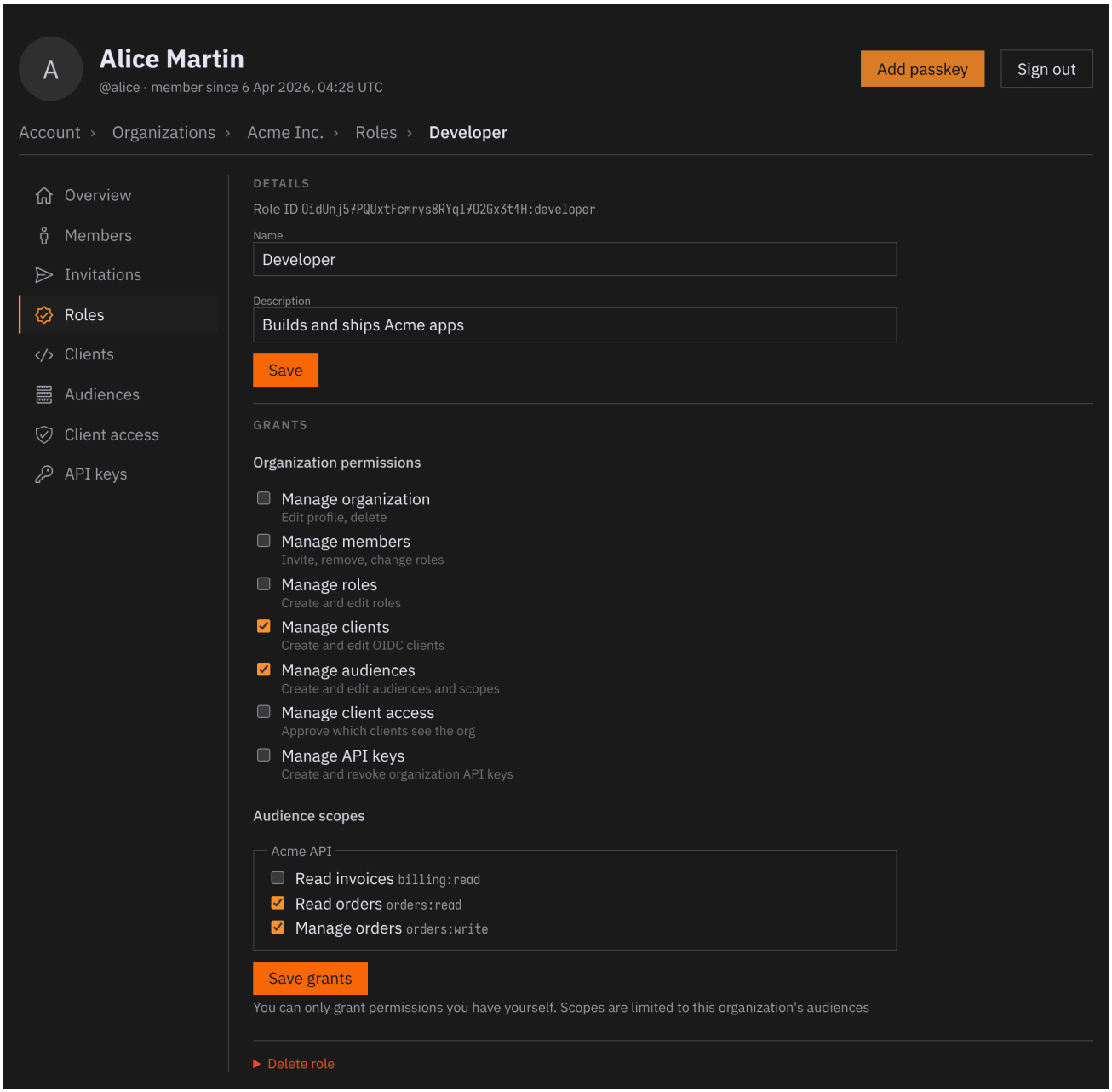
Administrators can make further global roles available to organizations; they show up in the role list next to these two, marked "built-in" like them. Organizations can also create their own roles:

```
mutation {
  manage {
    organization(organization: "acme") {
      createRole(key: "wiki-editors", name: "Wiki editors", description: "Edit the team wiki") {
        id
      }
    }
  }
}
```

The key follows the slug rules; the role id becomes `<organization id>:<key>`. A role's grants are organization permissions plus scopes of audiences owned by the same organization:

```
mutation {
  manage {
    role(roleId: "O...:wiki-editors") {
      setScopes(
        scopes: [{ audienceId: "A...", scopeId: "wiki:write" }]
        permissions: [CLIENTS]
      ) {
        audienceId
        scopeId
      }
    }
  }
}
```

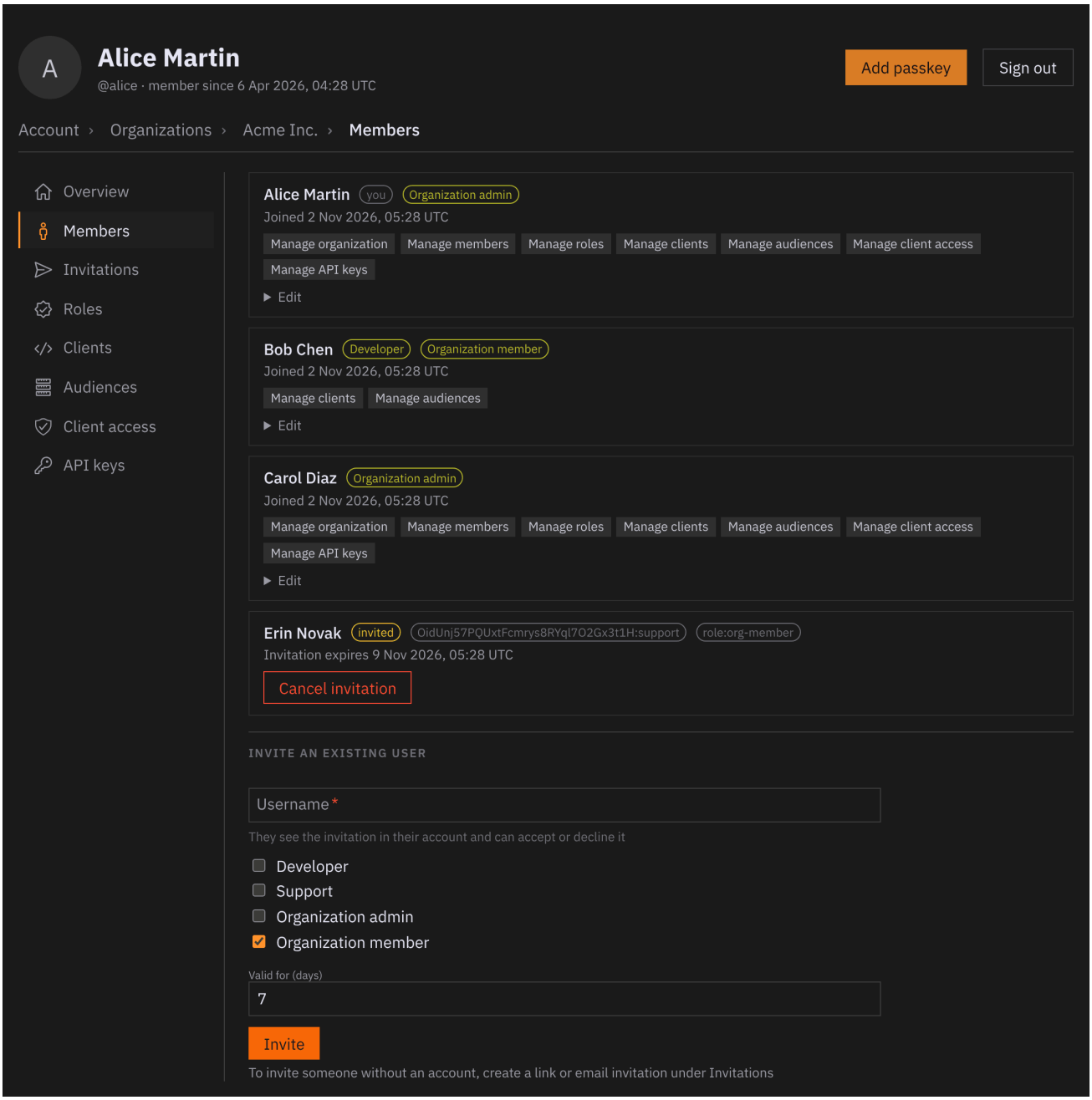
`setScopes` replaces both lists; leaving out `permissions` keeps the current ones. Audience scopes from roles end up in tokens when the user signs in to an app with this organization active, see Organizations and Scopes and claims. In the dashboard: Roles, "New role", then the role's page ("Grants").



A custom role granting two organization permissions and two scopes of the organization's audience

Members

The Members page lists everyone with their roles and resulting permissions. With `org:members`, each entry has an "Edit" fold to change roles ("Save roles") and to remove the member. You can't remove yourself there; use "Leave organization" on the Overview.



Members as seen by an organization admin

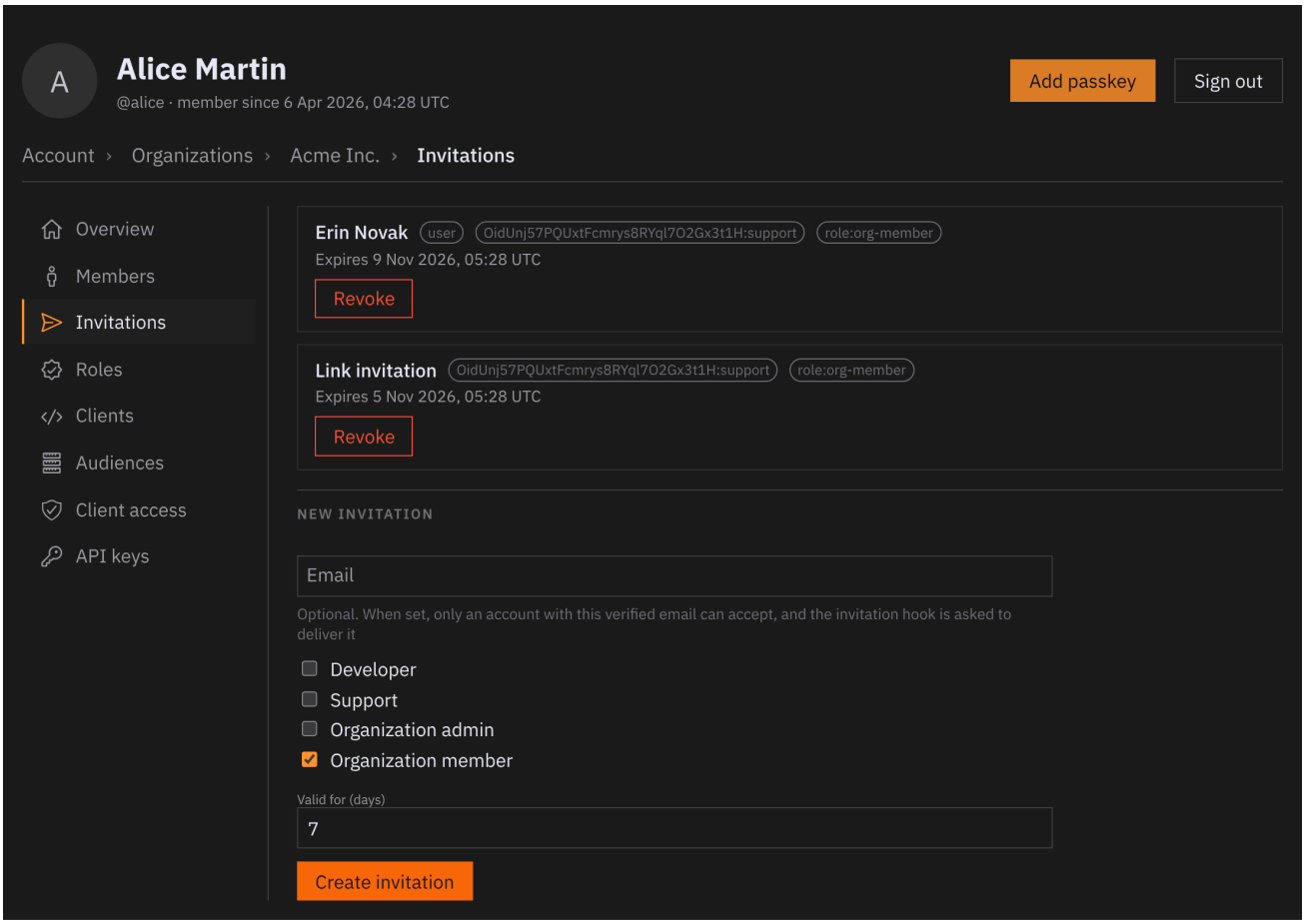
```
mutation {
  manage {
    organization(organization: "acme") {
      setMemberRoles(userId: "U...", roles: ["role:org-member", "O...:wiki-editors"]) {
        id
        name
      }
    }
  }
}
```

`removeMember(userId:)` and `leave` work the same way. All of them are subject to the guards in [Permission rules and escalation guards].

Invitations

Kind	Create	Who can accept	Delivery
Existing user	Members page, "Invite an existing user" (username); `invite(user: ...)`	only that account	Shows up under Organizations, "Invitations", with Accept and Decline.
Email	Invitations page with an email; `invite(email: ...)`	any account with that address verified	Emailed by the server when email is configured (see Email). The link is also shown once.
Link	Invitations page with the email left empty; `invite(roles: ...)` without `email` and `user`	anyone who has the link	You share the link, which is shown once.

Every invitation carries roles (the dashboard preselects `role:org-member`), expires after a number of days (7 by default) and can be revoked while pending. It is used up when someone joins. Links look like `https://auth.example.com/invite#<code>`; the invitee opens it, signs in and joins.



Pending invitations and the form for a new one

```
mutation {
  manage {
    organization(organization: "acme") {
      invite(email: "bob@example.com", roles: ["role:org-member"], expiresIn: 604800) {
        url
        invitation {
          id
          expiresAt
        }
      }
```

```
    }
  }
}
```

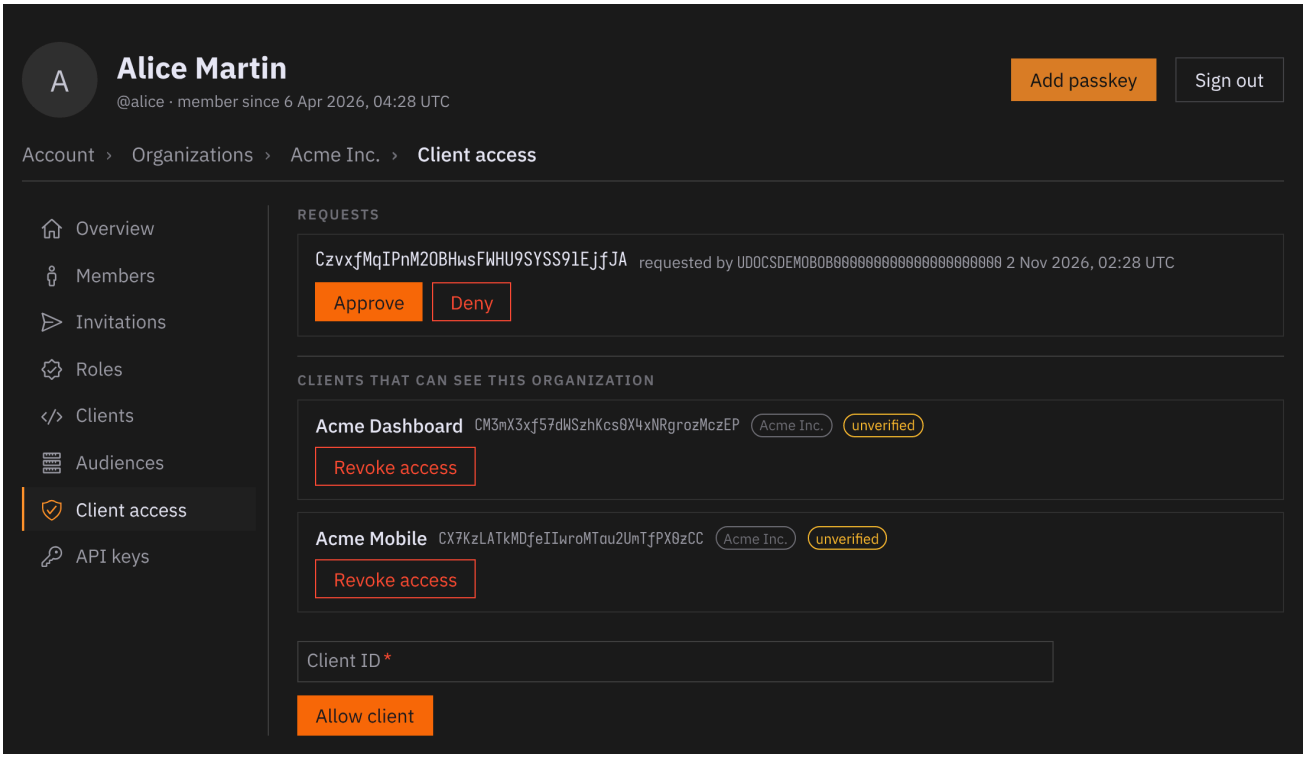
`expiresIn` is in seconds. Errors you may see: "invite either by email or by user, not both", "not a valid email address", "user is already a member", "user already has a pending invitation", "no user with that username", "Could not send the invitation email, try again later". If the server has no email configured, no invitation email is sent; pass on the link yourself.

The invitee's side: `manage { invitations { id organizationName roles expiresAt } }`, then `acceptInvitation(id:)` or `declineInvitation(id:)`.

Client access

A client only sees an organization (in the sign-in flow, in the `orgs` claim, in the Backend API) after the organization enabled it, unless an admin made it first-party with "All organizations". Clients owned by the organization are enabled for it when they are created. Everything else goes through the Client access page (`org:access`):

- **Requests**: members who wanted to use the organization with a client that is not enabled. "Approve" enables the client and clears all requests for it; "Deny" drops that member's request.
- **Clients that can see this organization**, each with "Revoke access".
- **Allow client**: enable a client by its client ID.



Client access with a member's request for a third-party app

Members ask for access on the sign-in screen ("Not enabled for this app", "Request access"). Enabling a client does not expose anyone by itself: each member still decides on the sign-in screen whether that app may see their membership.

```
mutation {
  manage {
    organization(organization: "acme") {
```

```
      setClientAccess(clientId: "C...", enabled: true)
    }
  }
}
```

Denying is `denyAccessRequest(clientId:, userId:)`; without `userId` it drops all requests for the client.

API keys

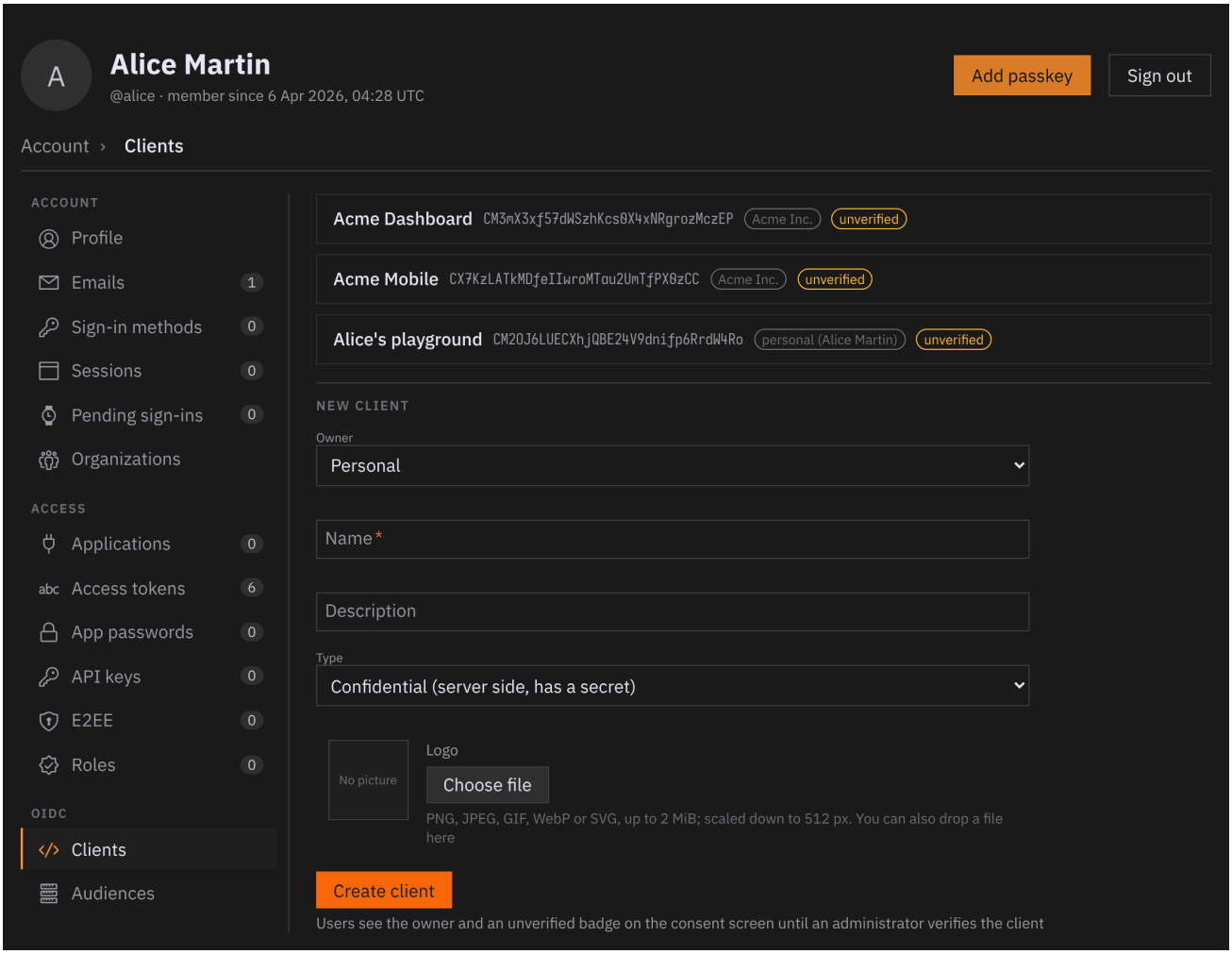
With `org:api-keys` the API keys tab lists the organization's API keys (created by apps through the Backend API) and lets you revoke them (`revokeApiKey(id:, reason:)`). See Backend API.

Clients (apps)

Create a client

Clients, "New client": choose the owner ("Personal" or an organization where you have `org:clients`), name, description, type and logo.

- **Confidential**: runs on a server and authenticates with a client secret.
- **Public**: runs in a browser or as a native app, PKCE only.



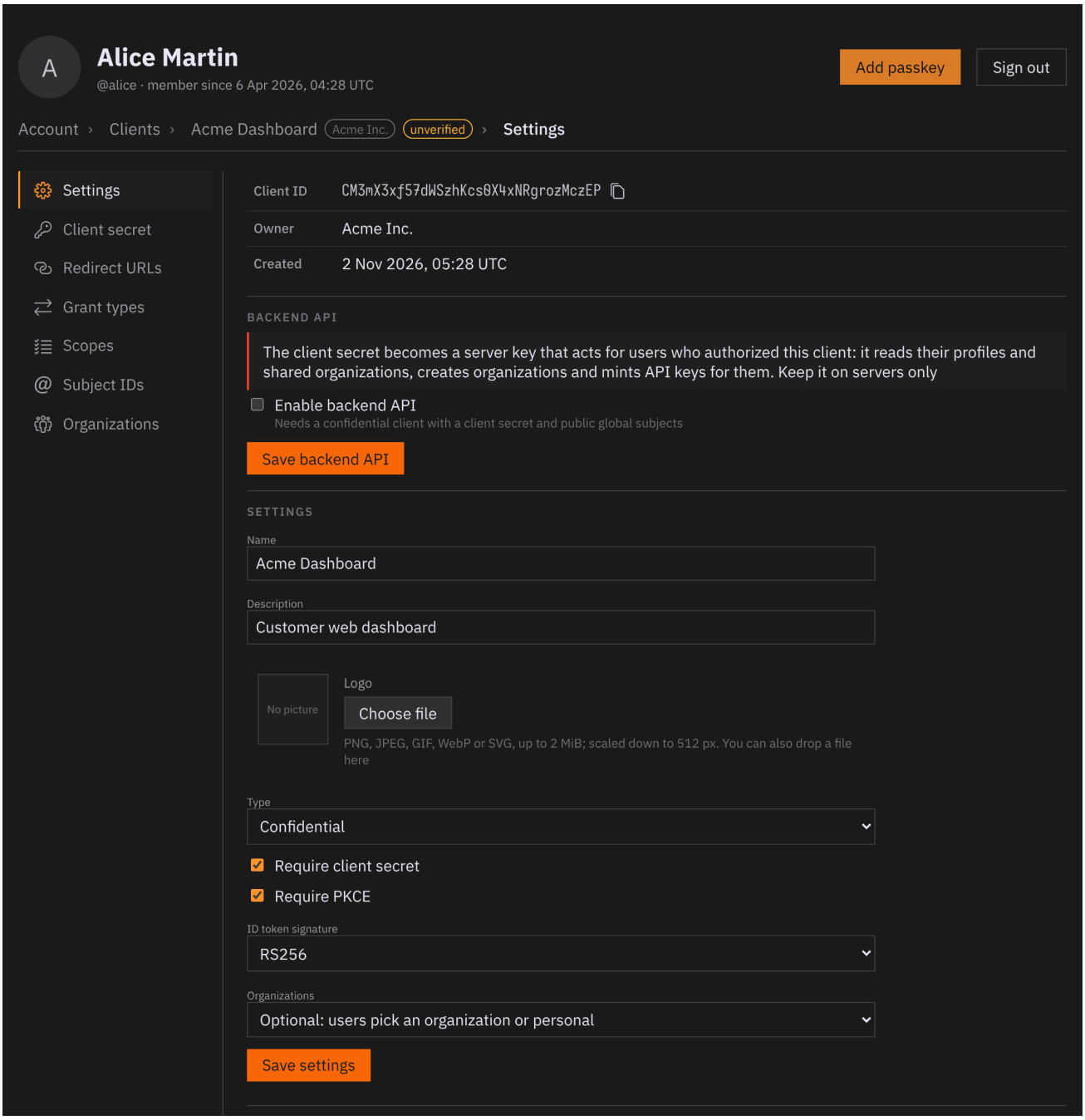
Your clients and the form for a new one

```
mutation {
  manage {
    createClient(organization: "acme", name: "Wiki", description: "Team wiki", clientKind: CONFIDENTIAL) {
      clientId
      allowedGrantTypes
    }
  }
}
```

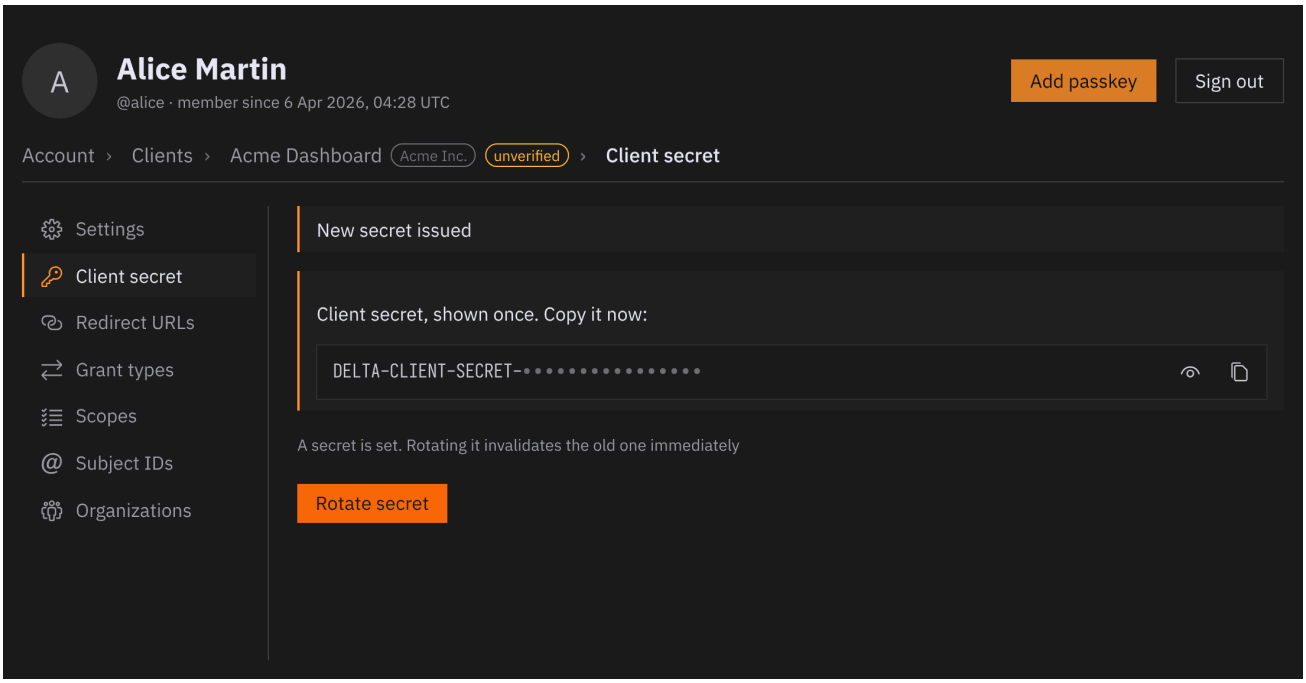
Leave out `organization` for a personal client. A new client allows the authorization code and refresh token grants, requires PKCE, requires a client secret if it is confidential, uses public global subject identifiers, does not use organizations and is unverified.

Client pages

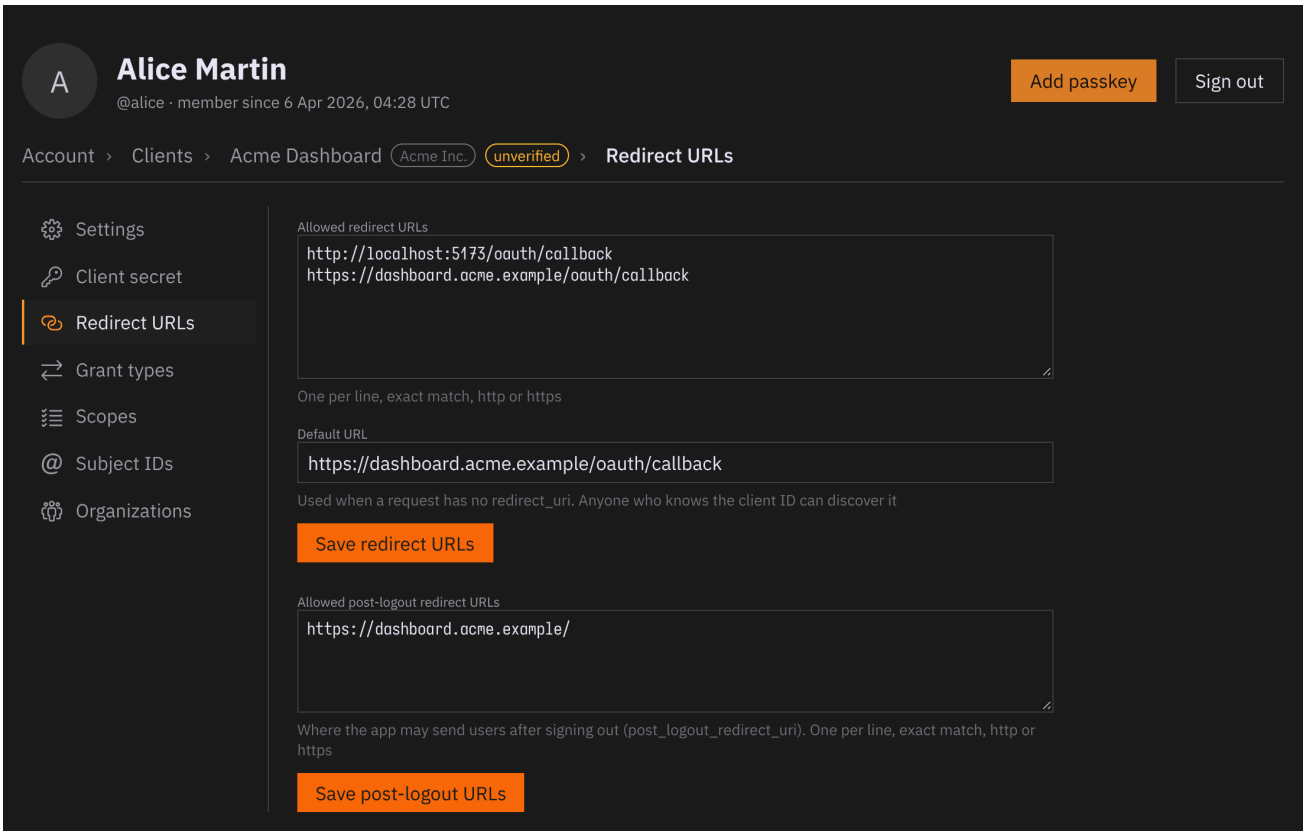
Tab	What you set
Settings	Name, description, logo, type, "Require client secret", "Require PKCE", ID token signature (RS256 or ES256), Organizations ("Not used", "Optional", "Required", see Organizations). The "Backend API" section turns the client secret into a server key, see Backend API. "Delete client" at the bottom.
Client secret	Confidential clients only. "Issue secret" or "Rotate secret". The secret (`DELTA-CLIENT-SECRET-...`) is shown once; rotating invalidates the old one immediately. A confidential client has no secret until you issue one, and token requests fail until then.
Redirect URLs	Allowed redirect URLs, one per line, `http` or `https`, exact match. The optional "Default URL" is used when a request has no `redirect_uri`; anyone who knows the client ID can discover it. Below: allowed post-logout redirect URLs.
Grant types	Authorization code, Refresh token, Device code, Backchannel (CIBA). Implicit is not supported, and Client credentials is not available yet. CIBA needs a client secret, see Backchannel authentication.
Scopes	Allowed scopes the client may request, and required scopes it always gets. The picker lists audiences you can manage. Built-in scopes such as `openid` and `profile` are always available.
App keys	App key scopes the client may request as `app_key:<id>`. The list shows scopes you manage plus the ones already allowed. See App key scopes.
Subject IDs	What `sub` looks like: public (same for every client) or pairwise (per sector), global or per organization, and which sibling clients share a pairwise sector. See Scopes and claims.
Organizations	Read-only list of organizations that enabled this client.



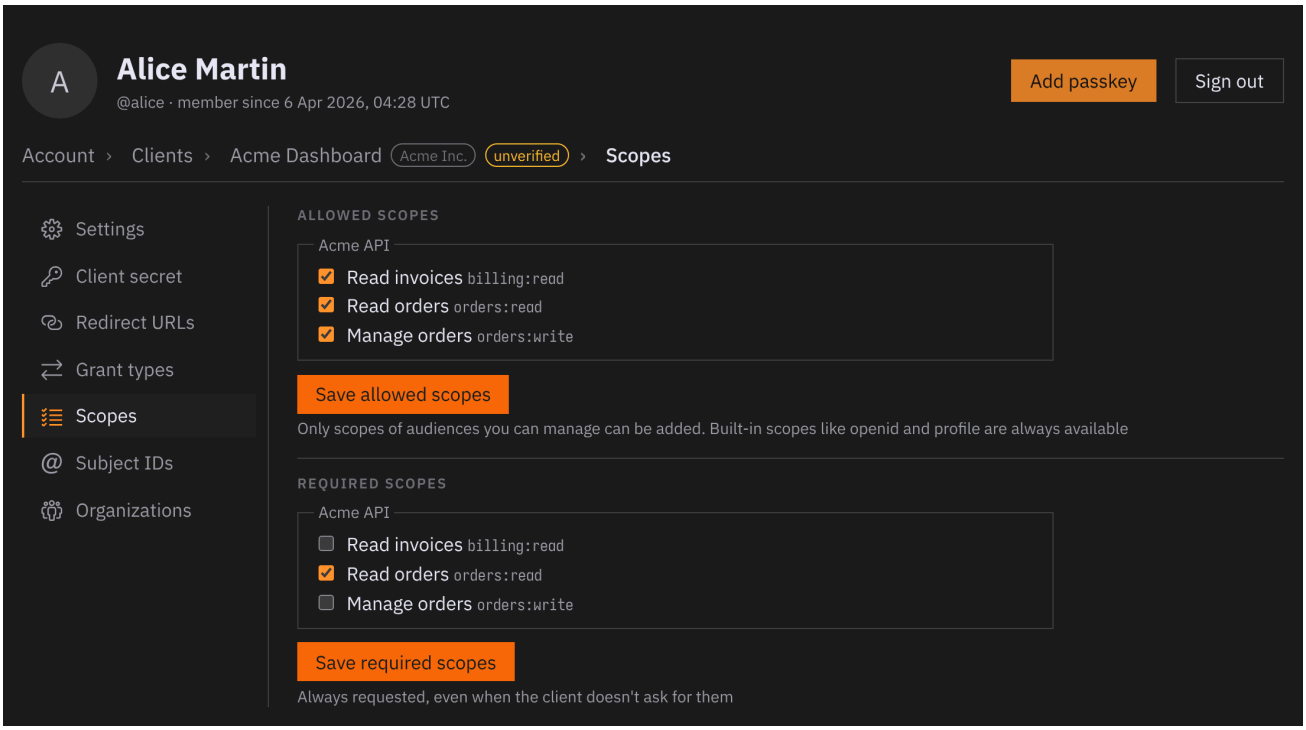
The Settings tab of an organization's client



A new secret right after "Rotate secret": masked, with buttons to show and copy it



Redirect URLs with a default URL, and post-logout redirect URLs



Allowed and required scopes from the organization's audience

Configuring by GraphQL (each `set...` replaces the whole list; `resetClientSecret` returns the new secret):

```
mutation {
  manage {
    client(clientId: "C...") {
      setRedirectUrls(urls: [{ url: "https://wiki.example.com/oauth/callback", public: true }]) {
        redirectUrl
      }
      setAllowedGrantTypes(grantTypes: [AUTHORIZATION_CODE, REFRESH_TOKEN])
      setAllowedScopes(scopes: [{ audienceId: "A...", scopeId: "wiki:read" }]) {
        scopeId
      }
      resetClientSecret
    }
  }
}
```

Other mutations on a client: `update(name:, description:, picture:, clientKind:, usesClientSecret:, usesPkce:, idTokenSignatureKind:, organizationUsage:)`, `setPostLogoutRedirectUrls(urls:)`, `setRequiredScopes(scopes:)`, `setAllowedAppKeyScopes(scopeIds:)`, `setSubject(subjectType:, subjectScope:, shareWith:, force:)`, `setBackendApi(enabled:)`, `delete`.

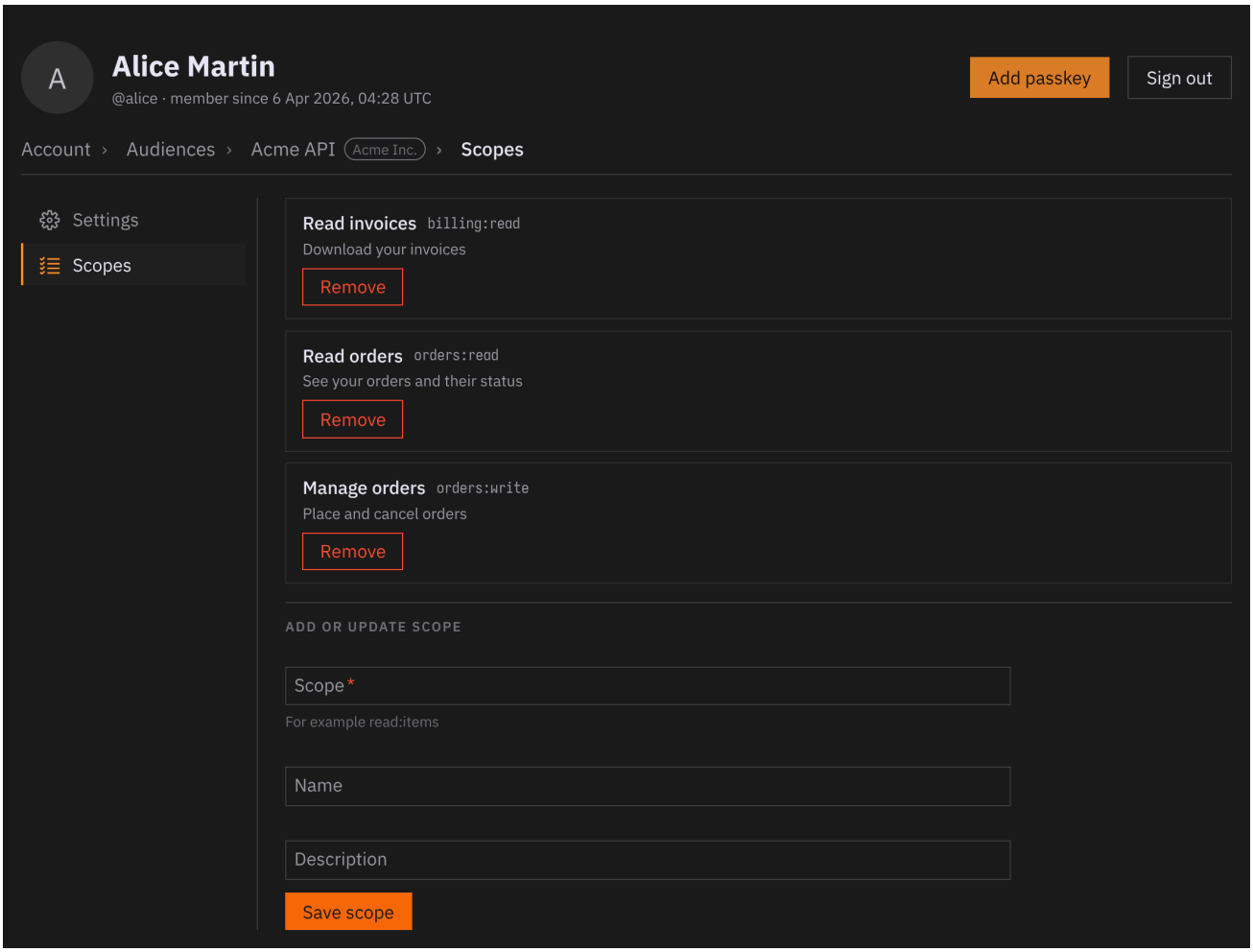
Changing subject settings of a client that already issued tokens or has consents needs `force: true` ("Change even if users already signed in"); every existing user then gets a new `sub`.

> **NOTE**
> Allowing a scope on a client does not give it to users. A user only gets an audience scope through a role: for organization audiences, an organization role assigned to them (active while that organization is selected at sign-in); otherwise a global role, which only admins can set up.

Audiences (APIs)

An audience is an API that clients request tokens for; its scopes are what users consent to.

Audiences, "New audience": owner, name, description, identifier and picture. The identifier must be unique, usually the API's URL (for example `https://api.example.com`); clients pass it as the `audience` parameter. On the audience's Scopes tab, "Add or update scope" takes a scope id (for example `read:items`), name and description; "Remove" deletes one.



An audience's Scopes tab

```
mutation {
  manage {
    createAudience(
      organization: "acme"
      name: "Wiki API"
      description: "Wiki REST API"
      audienceIdentifier: "https://wiki.example.com/api"
    ) {
      audienceId
    }
  }
}
```

```
mutation {
  manage {
    audience(audienceId: "A...") {
      declareScope(scopeId: "wiki:read", name: "Read the wiki", description: "Read pages") {
        scopeId
      }
    }
  }
}
```

- Access tokens are signed with RS256 (default) or ES256 (`signatureKind` on create, `accessTokenSignatureKind` on `update`); other algorithms are rejected.
- Scope ids: up to 128 printable ASCII characters, no spaces, double quotes or backslashes, not a built-in scope name and not starting with `app_key:`.
- `update(...)` changes name, description, picture, identifier and signature; `delete` removes the audience with its scopes (not possible for system audiences).

App key scopes

An app key scope names an end-to-end encryption key that clients can ask users for, see App encryption keys.

App key scopes, "New app key scope": owner, scope id, name, description and "Client-bound". The scope id (clients request it as `app_key:<id>`) and "Client-bound" can't be changed later. Each entry shows how many clients may request the scope and how many users have a key for it; "Edit" changes name and description, "Delete app key scope" deletes it together with every user's key.

```
mutation {
  manage {
    createAppKeyScope(scopeId: "luks", name: "Disk unlock", clientBound: true) {
      scopeId
    }
  }
}
```

- Scope ids: up to 128 printable ASCII characters, no spaces, double quotes or backslashes, not starting with `app_key:`, unique across the server.
- `appKeyScope(scopeId:) { update(name:, description:) delete }`; `delete` is not possible for system scopes.

Verified apps and the consent screen

The consent screen and the organization screen show who published the app:

Client	Publisher line
system-owned and verified ("first-party")	none
system-owned, not verified	"by Unknown publisher" with an "Unverified app" badge
owned by a user	"by <name>", with "Unverified app" until verified
owned by an organization	"by <organization name>", with "Unverified app" until verified

A user's name is their profile name, else their username. Only administrators can verify a client, see Mark an app verified. For personal and organization clients verification only removes the badge; a verified system client becomes first-party. In the dashboard, client lists show a "verified" or "unverified" badge and an owner badge.

Permission rules and escalation guards

All checks run on the server. An admin token (admin mode or GraphiQL) skips them; an admin using a normal session is treated like everyone else.

Visibility

- Personal clients and audiences: only the owner.
- Organization clients, audiences and roles: every member of the organization.
- System clients and audiences: only admins.
- Organizations: only members.

Editing

- A client or audience needs ownership, or `org:clients` / `org:audiences` in the owning organization.
- An organization role needs `org:roles` in that organization. Global roles, including `role:org-admin` and `role:org-member`, can only be edited by admins, and only organization roles can be deleted.
- First-party clients and system audiences can't be deleted through `manage`.

Assigning roles (invitations and member roles)

- The role must belong to this organization or be a global role that admins made organization-assignable, otherwise "role can't be assigned in this organization".
- The role's permissions must all be permissions you hold yourself, otherwise "permission denied".

Changing or removing a member

- The member's current permissions must be a subset of yours: you can't demote or remove someone with more rights.

Role grants

- You can only add organization permissions you hold; permissions a role already has can be kept.
- Audience scopes must come from audiences owned by the same organization ("organization roles may only include scopes of organization audiences"). Other scopes of the server's own audience are rejected.

Keeping a member manager

- Any change that would leave no member with `org:members` fails with "change would leave no member able to manage members": changing a member's roles, removing a member, leaving, changing a role's grants or deleting a role. The last remaining member may still leave.

Client scopes

- Adding allowed or required scopes of an audience needs the right to manage that audience (owner, or `org:audiences` in its organization), otherwise "you can't grant scopes of audience <name>". Removing scopes is always allowed.
- Adding an app key scope to a client needs the right to manage that scope, otherwise "you can't grant app key scope <name>". Removing is always allowed.

Admin only

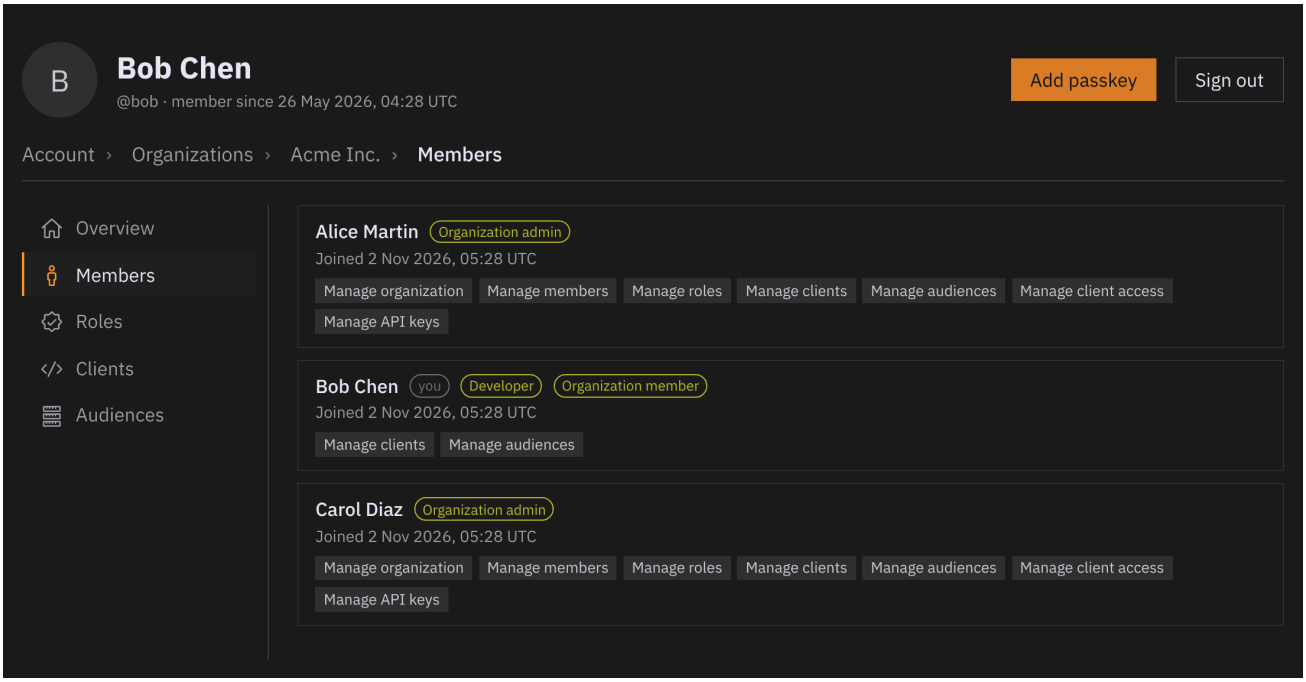
- `setVerified`, `setAllOrganizations`, `setOrganizationAssignable` and `organizations(all: true)`.

Redirect URLs

- Clients created through dynamic registration with a verified pairwise sector must keep their redirect URLs inside that sector.

What is hidden and what is disabled

The API tells the dashboard what you may do, and the dashboard hides actions you can't take and disables fields you can't change.



The same Members page for a member without `org:members`: no Invitations, Client access or API keys tabs and no edit controls

```
query {
  manage {
    viewer {
      userId
      admin
    }
    organization(organization: "acme") {
      viewerIsMember
      viewerPermissions
      invitations { id }
      clients {
        clientId
        viewerCanEdit
        owner { kind name }
      }
    }
  }
}
```

- `viewer.admin` is true only for admin tokens.
- `ManagedOrganization.viewerPermissions` lists your permissions in the organization (all of them for admins); `viewerIsMember` is false when an admin looks at an organization they don't belong to.
- `ManagedClient`, `ManagedAudience` and `ManagedRole` have `viewerCanEdit`.
- Permission-gated organization fields return `null` instead of an error: `invitations` (needs `MEMBERS`), `accessRequests` and `enabledClients` (`ACCESS`), `apiKeys` (`API_KEYS`). The matching dashboard pages answer 404.
- Mutations you are not allowed to run fail with "permission denied".

Dashboard element	Without the right
Invitations, Client access, API keys tabs	hidden without `org:members`, `org:access`, `org:api-keys`
Organization edit form and "Delete organization"	hidden without `org:manage`; "Leave organization" only for members
Member "Edit" fold, "Invite an existing user"	hidden without `org:members`; no remove button on yourself
"New role"	hidden without `org:roles`
Role page fields	disabled unless `viewerCanEdit`; permission checkboxes you don't hold are disabled and keep their state
"New client" / "New audience" links on organization pages	hidden without `org:clients` / `org:audiences`
Owner select on "New client" / "New audience"	lists only organizations where you have `org:clients` / `org:audiences`
Client and audience pages	fields disabled and a "read-only" badge unless `viewerCanEdit`; save buttons, "Backend API", scope add and remove, secret rotation and "Delete" hidden
"Mark as verified", "All organizations (first-party)"	admin mode only