

SSO For Small Ecosystems

One identity provider, every service, and the token-mapping details that don't show up in the setup guides.

- [Forward](#)
- [Deploying Keycloak](#)
- [Configuring Keycloak](#)
- [Configuring Discourse \(identity-only pattern\)](#)
- [Configuring BookStack \(group-aware pattern\)](#)
- [Configuring a Flask app \(build-it-yourself pattern\)](#)
- [Appendix A - SSO Test App](#)

Forward

One identity provider, every service, and the configuration that makes it work.

This book is a working reference for putting every service in a self-hosted environment behind a single identity provider. It opens with the case for why you should centralize your identities and the reasoning behind the tooling, then gives annotated, reproducible configuration for three representative integration patterns: an identity-only app (Discourse), a group-aware app (BookStack), and an app you write yourself (a Flask service). Each configuration page calls out the pattern it represents so it can be adapted to similar applications.

Why centralize identity

Every application that manages its own logins is a small identity system you now have to run. It stores passwords, it decides how (or whether) to do multi-factor authentication, it has its own idea of what a session is, and it keeps its own list of who is allowed in. Run a handful of apps that way and you have a handful of password stores, a handful of MFA configurations of varying quality, and a handful of places you have to remember to visit when someone should no longer have access.

A single sign-on identity provider (IdP) collapses all of that into one place. Users authenticate once, against one system, and every application trusts that system instead of holding credentials of its own. The practical consequences are the whole point:

- **One place to authenticate.** One login, one session, one password users actually have to remember, which means it can be a strong one, backed by MFA, instead of ten mediocre ones.
- **One place to enforce policy.** MFA, password rules, and session lifetimes are configured once and apply everywhere. No app can quietly be the weak link.
- **Apps stop storing passwords.** Each application holds a client secret and a trust relationship, not user credentials. The blast radius of any single app being compromised shrinks accordingly.
- ***One place to revoke.** Offboarding is a single action. Disable the account at the IdP and every downstream application is closed at once, with no hunting through ten admin panels hoping you didn't miss one.

This matters *more* at small scale, not less. A large organization has an IAM team to manage accounts. One person running an environment for friends and family will find it difficult to keep account state consistent and secure across every service by hand. Centralizing identity is what makes right-sized, genuinely secure operation achievable by one person. It is the single highest-

leverage security decision in the whole environment.

There is a catch: for any of this to hold, *every* service has to sit behind the IdP. A single application with its own local-login escape hatch defeats the revocation guarantee, the MFA guarantee, and the "apps don't store passwords" guarantee all at once. Universality isn't a nice-to-have here. It is the property that makes the model worth anything.

“***SSO centralizes authentication, not sessions.** Each application maintains its own local session once a user has logged in, and disabling an account at the identity provider does not automatically end those sessions. What actually happens on revocation depends entirely on the application:

- Applications that revalidate against the IdP, or that support back-channel logout and have it configured, close promptly.
- Applications with fully self-contained session management keep the user signed in until their local session expires, even with no active IdP sessions.
- Already-issued access tokens remain valid until they expire, because applications validate them by signature rather than by asking the IdP whether they are still good. Shortening token lifetime shrinks that window but does not eliminate it.

There is also a quirk worth knowing: in Keycloak, signing out an individual session fires back-channel logout to clients that support it, while the bulk "sign out all sessions" action does not.

The practical consequence is that full offboarding is a sequence, not a single click.

- Disable the account at the IdP first so nothing can re-authenticate
- Revoke refresh and offline tokens and push a not-before policy
- Then evict each application's local session

In my environment that last step is a script, because the applications expose different eviction surfaces, from clean per-user admin APIs to session tables that have to be cleared directly to filesystem session stores with no user index at all, where the only option is logging everyone out of that app.

None of this undermines the case for centralizing identity. Doing this without an IdP means hunting through every application's admin panel with no single gate to close first, which is undeniably worse. But "one action closes everything" oversells it, and the gap between stopping new access and ending existing access is the kind of detail that matters during an incident.

Why Keycloak

The IdP I chose for my environment is [Keycloak](#). The reasons it won out:

- **Open-source and self-hostable.** It runs on infrastructure I control, with no per-seat SaaS pricing and no user identities living in someone else's cloud. For a privacy-focused, self-hosted environment, that alignment matters; the identity system shouldn't be the one component you rent from a third party.
- **Standards-based.** It speaks OIDC, OAuth 2.0, and SAML, universal protocols of which any application that supports SSO will utilize at least one, and the knowledge transfers directly to enterprise systems. Every integration below is a standard OIDC flow.
- **Realms give real isolation.** Keycloak's realm model lets me run entirely separate identity domains. Production services authenticate against one realm; development instances authenticate against a second. Dev accounts and prod accounts never mingle, and I can experiment with client and mapper configuration in dev without risking the identity system real people depend on.
- **Enough authorization power without the bloat.** Groups, roles, client scopes, and protocol mappers are enough to drive real access control in every downstream app, without dragging in a heavyweight enterprise IAM suite that would be absurd at this scale.

The honest tradeoff: Keycloak has a real learning curve, and it does not shy away from breaking changes between major versions. Upgrades need planned maintenance windows with database backups taken first, release notes read carefully, and realm and client configuration re-verified afterward, not a casual `docker compose pull`. That is a cost taken on with eyes open. The centralization it buys is worth the operational care it demands.

Keycloak has a good startup guide to get you started here: [Keycloak Startup Guide](#)

How it fits the architecture

My IdP lives in the environment's infrastructure and trust zone, on its own host, and like everything else, the host and config structure is reachable only from within the VPC. Applications never handle credentials themselves. When a user hits an application without a valid session, the app redirects the browser to Keycloak; the user authenticates there (with MFA when required); Keycloak returns an authorization code; and the application exchanges that code for tokens on a back channel it never exposes to the browser. Authorization is then decided by the application from the claims in the token it receives.

Every integration uses the OIDC authorization-code flow with PKCE (S256), confidential clients, and standard flow only. That uniformity is deliberate: a new service isn't a fresh design problem, it is the same pattern applied again, which is what keeps the whole thing maintainable by one person.

Deploying Keycloak

Everything else in this book assumes a working Keycloak that applications can reach over HTTPS and that you can administer safely. This page covers the deployment and the next covers configuring realms inside it.

The deployment below is written as a standalone Keycloak instance. It terminates its own TLS and is reachable directly. If you run it behind a reverse proxy or web application firewall, as my environment does, see the notes in the config file and at the end of the page, because several settings change.

Installing Keycloak

The [startup guide](#) on the Keycloak website will get you started with a dev environment to be able to test drive it. Follow the instructions there to get started. In the appendix I have some base code you can deploy if you want to configure a test application.

When you are ready to move to production, below are the steps you need to follow. Keycloak should be considered critical infrastructure, so the best practice is to run it on its own host alongside no other services.

1. Update DNS to point to your server (e.g., kc.example.com)
2. Validate firewall ports 80 and 443 are open
3. Create directories

```
sudo mkdir -p /opt/apps/keycloak/{nginx,webroot,themes,certbot-etc,certbot-var}
```

4. Write .env file with database credentials, hostname, and bootstrap admin credentials

```
KC_ADMIN=<admin_username>
KC_ADMIN_PASSWORD=<admin_account_password>
POSTGRES_DB=keycloak
POSTGRES_USER=keycloak
POSTGRES_PASSWORD=<postgres_service_account_password>
KEYCLOAK_HOSTNAME=<https://kc.example.com>
```

5. Write keycloak.conf (/opt/apps/keycloak/nginx/keycloak.conf)

```

server {
    listen 80;
    server_name kc.example.com;

    location /.well-known/acme-challenge/ {
        root /webroot;
    }

    location / {
        return 301 https://$host$request_uri;
    }
}

```

6. Write docker-compose.yml (/opt/apps/keycloak/docker-compose.yml)

```

services:
  postgres:
    image: postgres:15 # Pinned major version. Postgres major upgrades require
a # dump/restore, so this is deliberately not floating.
    restart: unless-stopped # Restart on failure and on daemon start, but honor a
manual stop.
    environment:
      POSTGRES_DB: ${POSTGRES_DB} # DB name, from .env
      POSTGRES_USER: ${POSTGRES_USER} # DB user, from .env
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD} # DB password, from .env
    volumes:
      - pgdata:/var/lib/postgresql/data # Named volume: the inside container database files
managed by docker
    networks:
      - keycloak_net # No published ports: reachable only from the
compose network.

  keycloak:
    image: quay.io/keycloak/keycloak:26.6.4 # Pinned. Keycloak has breaking changes between
# minor versions; :latest is an operational
hazard.
    restart: unless-stopped
    command: start # Production mode. (start-dev disables TLS requirements
and

```

```

                                # is for local experimentation only.)

depends_on:
  - postgres                                # Start ordering only. Does NOT wait for Postgres to be
ready,                                     # so Keycloak may need a restart cycle on first boot.

environment:
  KC_DB: postgres                          # Database vendor.
  KC_DB_URL_HOST: postgres                  # Hostname = the compose service name, resolved on
keycloak_net.
  KC_DB_URL_DATABASE: ${POSTGRES_DB}
  KC_DB_USERNAME: ${POSTGRES_USER}
  KC_DB_PASSWORD: ${POSTGRES_PASSWORD}
  KC_HOSTNAME: ${KEYCLOAK_HOSTNAME}        # The PUBLIC URL Keycloak believes it lives at. Used
to build
                                           # the token issuer, the discovery document
endpoints, and
                                           # redirect targets. Wrong value = clients get URLs
they
                                           # can't reach, or issuer validation fails.

  KC_HTTP_ENABLED: "true"                  # Keycloak serves plain HTTP internally; the nginx
sidecar terminates TLS.
  KC_PROXY_HEADERS: xforwarded             # Trust X-Forwarded-* headers.

  KC_BOOTSTRAP_ADMIN_USERNAME: ${KC_ADMIN}                                # Created on FIRST START
only, remove after
  KC_BOOTSTRAP_ADMIN_PASSWORD: ${KC_ADMIN_PASSWORD} # Created on FIRST START only, remove
after

#   Uncomment while developing themes, comment out when finished
#   KC_SPI_THEME_STATIC_MAX_AGE: "-1"      # Disable static asset caching
#   KC_SPI_THEME_CACHE_THEMES: "false"     # Reload themes on each request
#   KC_SPI_THEME_CACHE_TEMPLATES: "false"  # Reload templates on each request

ports:
  - "127.0.0.1:8080:8080"                  # Bound to loopback: not reachable from the network,
only from
                                           # the host itself.

volumes:
  - /opt/apps/keycloak/themes:/opt/keycloak/themes:ro # Custom login themes, read-
only.

```

```

networks:
  - keycloak_net
logging:
  driver: "syslog" # Ship container logs to host syslog...
  options:
    tag: "keycloak" # ...tagged "keycloak", which is what the log-parsing
and
                                # alerting pipeline keys on.

nginx:
  image: nginx:alpine # Unpinned minor version; alpine tag floats.
  restart: unless-stopped
  ports:
    - "80:80" # HTTP redirect to HTTPS
    - "443:443" # Serves HTTPS
  volumes:
    - ./nginx/keycloak.conf:/etc/nginx/conf.d/default.conf # Proxy config for this
container.
    - /opt/apps/keycloak/certbot-etc:/etc/letsencrypt # BIND mount (host path)
    - /opt/apps/keycloak/certbot-var:/var/lib/letsencrypt # BIND mount (host path)
    - ./webroot:/webroot # ACME challenge webroot, shared
w/ certbot
  depends_on:
    - keycloak
  networks:
    - keycloak_net

certbot:
  image: certbot/certbot # Unpinned.
  volumes:
    - /opt/apps/keycloak/certbot-etc:/etc/letsencrypt
    - /opt/apps/keycloak/certbot-var:/var/lib/letsencrypt
    - ./webroot:/webroot # Shared challenge webroot.
  entrypoint: "/bin/sh -c 'trap exit TERM; while ;; do certbot renew --webroot --webroot-
path=/webroot; sleep 12h & wait $$(!); done'"
                                # Attempts renewal every 12h; certbot no-ops unless
within 30 days of expiry.
  networks:
    - keycloak_net

```

```
volumes:
  pgdata:

networks:
  keycloak_net:          # Private bridge network. Service names resolve as
                           hostnames.
```

7. Start everything but certbot

```
docker compose up -d postgres keycloak nginx
```

8. Issue public certificate

```
docker compose run --rm --entrypoint certbot certbot certonly \
  --webroot --webroot-path=/webroot \
  -d kc.example.com \
  --email you@example.com --agree-tos --no-eff-email
```

9. Add 443 serverblock to keycloak.conf

```
server {
    listen 443 ssl;
    server_name kc.example.com;

    ssl_certificate /etc/letsencrypt/live/kc.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/kc.example.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        proxy_pass http://keycloak:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
    }
}
```

10. Restart containers

```
docker compose restart nginx
```

```
docker compose up -d
```

11. Proceed with initial config

Notes

Replace kc.example.com with your actual site in the following places:

- Your DNS site in step 1
- The KEYCLOAK_HOSTNAME variable in step 4
- The server_name in step 5
- The -d switch in step 8
- And three places in your keycloak.conf file in step 9

First login and locking down the admin account

1. Browse to the Keycloak URL and sign in with the bootstrap administrator.
2. Create a real administrative user with a strong, unique password stored in a password manager.
3. I highly recommend you enable MFA on that account immediately. The Keycloak administrator can create clients, alter authentication flows, and grant themselves access to every downstream application. It is the most privileged account in the environment and should never be protected by a password alone.
4. Remove the bootstrap credentials from the .env file and restart the containers.

Upgrades

Because the version is pinned, upgrades are a deliberate procedure rather than a background event. The sequence that has proven necessary:

1. **Carefully review the [patch notes](#)** - Make note of breaking changes, and it's worth to take a quick look at community pages to see if the patch has introduced any additional issues. There was one such issue involving a [PostgreSQL startup crash loop](#) for an upgrade that, thankfully, I found out about prior to upgrading, so I'm glad I checked.
2. **Back up your database and config** - You can use the following commands to do so:

```
docker exec -t keycloak-postgres-1 pg_dump -U ${POSTGRES_USER} -d ${POSTGRES_DB} >
keycloak_$(date +%F).sql
sudo tar -czvf kc_certbot_$(date +%F).tar.gz /opt/apps/keycloak/certbot-etc
sudo tar -czvf kc_nginx_$(date +%F).tar.gz /opt/apps/keycloak/nginx
sudo tar -czvf kc_themes_$(date +%F).tar.gz /opt/apps/keycloak/themes
sudo tar -czvf kc_config_$(date +%F).tar.gz /opt/apps/keycloak/docker-compose.yml .env
```

3. **Update docker-compose.yml as needed** - Make any changes based on the patch notes; features which may have been added, modified, or deprecated and require new keys in the yml. Then change the image key to reflect the desired updated version.
4. **Pull upgrade and restart** - Run the below, and then test the admin UI and a couple logout / in of your connected services to validate.

```
docker compose up -d --pull-always
```

Running behind a reverse proxy or WAF

In my environment, all my publicly-facing URLs sit behind a Web Application Firewall (WAF). That arrangement changes several things:

- **TLS terminates at the proxy, not at Keycloak.** The public certificate lives on the proxy. Keycloak still needs to know its public URL through `KC_HOSTNAME`, because that is what it publishes in the discovery document and uses to build redirects, but it is no longer the component presenting the public certificate to browsers.
- **Keycloak must be told it is behind a proxy.** Keycloak needs to trust the forwarded headers so it can construct correct URLs and see the real client address rather than the proxy's. Without this, redirects can be built with the internal hostname or the wrong scheme, and login loops or "invalid redirect" errors follow. The proxy must be configured to send the standard forwarded headers, and Keycloak configured to honor them.
- **Certificate management moves.** The public certificate is obtained and renewed on the proxy. Internally, the connection from the proxy to the Keycloak host is authenticated separately, which in this environment means mTLS with certificates from a self-hosted private Certificate Authority.

“ A dedicated book on running a web application firewall in small ecosystems can be found [here](#), covering the proxy configuration, request inspection, and the internal authentication between the proxy and its backends in detail.

What changes

Aligned to the steps above in the `Installing Keycloak` section.

1. DNS should point to the WAF, not your keycloak host
2. Firewall rules depend on how your WAF is configured
3. You can remove nginx, certbot-etc, and certbot-var from the folder list
4. No changes
5. Skip this step
6. Remove the nginx and certbot service blocks
7. Run docker compose up -d Skip steps 8-10, you're done!

What's new

- The WAF must send X-Forwarded-Proto https. Without it, Keycloak builds http URLs and you will get login loops.
- The WAF must pass X-Forwarded-For. Without, Keycloak's brute-force detection sees every attempt as coming from the WAF, and per-address protection becomes meaningless.

Configuring Keycloak

A running Keycloak does nothing useful until you decide how realms are split, how users prove who they are, how long sessions last, and how groups and roles are named. Those decisions are hard to change later, because every downstream application depends on them.

Realms: separating production from development

A realm is an isolated identity domain. Users, groups, roles, and clients belong to exactly one realm, and Keycloak treats users in different realms as entirely separate people even if they share a name or email address.

My environment runs two:

Realm	Purpose
production	Real people, real applications and services.
development	Test accounts and development instances of applications and services.

The separation is worth the small overhead for three reasons:

- **Test accounts are not real people.** Development work needs users to log in as, with various role and group memberships, in states you might not create for a real person. Keeping them in a separate realm means a test account cannot accidentally hold access to a production service, because production clients do not exist in that realm at all. I also want to be able to log in as these test users myself. rather than waiting for a real person to log in and test.
- **Configuration can be broken safely.** Authentication flows, mappers, and client scopes can be experimented with in the development realm without locking real users out of applications and services they depend on. Given how easy it is to break a login flow while tinkering with it, this alone justifies the split.
- **Blast radius.** A misconfigured client or an over-broad group in development has no path to production, because the two realms share nothing.

The one wrinkle worth knowing: because realms are fully isolated, an administrator who wants to test as themselves needs an account in *both* realms. Recreating your own account in the development realm is normal and expected; it is a different user object that happens to share your

name and email address.

Do not use the `master` realm for anything but Keycloak administration. It exists to administer the server itself, and putting application users or clients there conflates "can log into an application" with "can administer the identity provider."

Multi-factor authentication

MFA is the single highest-value setting in the realm, and the argument for it is straightforward. An identity provider concentrates authentication into one place, which means one compromised password is potentially access to every application behind it. Password-only authentication squanders one of the main benefits of centralizing identity.

Where MFA is required in this environment:

- **The Keycloak administrator account, without exception.** This account can create clients, rewrite authentication flows, and grant itself access to every downstream service. It is the most privileged identity in the environment.
- **Anyone with administrative privileges in any application or service.** If a group membership grants someone administrative rights in a wiki, a Git host, or a file service, that person's account needs MFA, because their credential now protects more than their own data.
- **Anyone who can reach sensitive information.** Password vaults, personal documents, financial records: if the account can reach it, the account needs a second factor.

How to enforce it

Keycloak offers two approaches, and the difference matters:

- **A required action** on the user, which prompts them to configure an authenticator the next time they log in. Simple, but it is per-user and easy to forget when adding someone new.
- **A conditional step in the authentication flow**, which requires OTP for users in a particular group or holding a particular role. More work to set up, but it is *declarative*. Anyone who becomes an administrator inherits the MFA requirement automatically, and nobody has to remember to apply it.

The conditional-flow approach is the stronger pattern for the same reason group-based access control is stronger than per-user permissions. The policy follows the role rather than depending on someone remembering to apply it. A realm-wide OTP requirement for every user is stronger still, and worth considering if the user population will tolerate it.

Whichever approach is used, back it up with:

- **Brute-force detection enabled**, so repeated failures lock an account temporarily rather than allowing unlimited attempts.
- **A password policy** that sets a real minimum length. Length matters more than composition rules.

Sessions and token lifetimes

“ **A caveat before the numbers:** the settings below are recommendations with reasoning, not a claim about what any particular environment should use. Applications differ in how they consume tokens, and some maintain their own session lifetimes independently of the identity provider, so realm values are a starting point that individual services may effectively override. Tune with knowledge of how your own applications behave.

Four settings do most of the work:

Setting	What it controls	Reasoning
SSO Session Idle	How long a session survives without activity before the user must log in again.	A few hours is a reasonable balance. Long enough that a user is not re-authenticating repeatedly during a working session; short enough that an abandoned browser does not stay authenticated indefinitely.
SSO Session Max	The absolute cap on a session regardless of activity.	Capping at roughly a day forces a fresh authentication daily, which bounds how long a stolen session can be useful.
Access Token Lifespan	How long an issued access token remains valid.	Short, in minutes. This is the most security-relevant of the four, for the reason below.
Client Session Idle / Max	Per-client session bounds, if you need a specific application to behave differently from the realm default.	Leave at the realm default unless a particular application needs shorter sessions, such as one holding sensitive data.

Why access token lifespan is the one to think hardest about. Applications validate access tokens by checking their signature, not by asking Keycloak whether the token is still good. That means a token that has already been issued stays valid until it expires, even after the account is disabled and its sessions are terminated. The access token lifespan *is* the residual access window during offboarding or incident response, and shortening it is the only thing that shrinks that window. Minutes rather than hours is the right order of magnitude. This connects directly to the revocation caveat elsewhere in this book: disabling an account stops new access immediately, but

already-issued tokens live out their lifespan regardless.

The tradeoff is that shorter access tokens mean more frequent refresh requests to Keycloak. In a small environment this load is negligible, which is another instance of small scale making the secure choice easy.

Groups and roles: decide the strategy before the first client

Getting this wrong is expensive, because changing naming conventions later means revisiting every application if you want to keep consistency.

Keycloak gives you two mechanisms to manage permissions for applications and services:

- **Groups** are collections of users, arranged in a hierarchy with paths like `/appname/groupname`. They are the natural fit when an application wants to know "which bucket is this user in," and they are what most applications consume for authorizations.
- **Realm and client roles** are named permissions that can be assigned to users or to groups. They fit better when an application asks "does this user hold this specific capability."

In practice, groups do most of the work, because most applications map incoming group names onto their own internal roles.

A per-application namespace

The convention that has held up well for me is a top-level group per application, with the application's own role names beneath it:

```
/bookstack/admin
/bookstack/editor
/bookstack/viewer

/gitea/admin
/gitea/developer
/gitea/read

/files/admin
/files/<interestGroup>/<shareName>
```

```
/files/quota/1GB
/files/quota/5GB
/files/quota/unlimited
```

Three properties make this work:

- **The application name is in the path**, so it is immediately obvious what a group grants and which system cares about it. A group called `/bookstack/editor` needs no explanation.
- **The child names match the application's own vocabulary**. BookStack has roles called admin, editor, and viewer, so the groups use those words. This makes the mapping on the application side nearly automatic and removes a translation step where mistakes hide.
- **Adding an application does not disturb existing ones**, because each occupies its own namespace.

Groups can carry more than permissions

Group membership does not have to mean "may perform this action." Applications sometimes consume groups to drive other behavior entirely: storage quotas, feature access, or which shared spaces a user can see. A file-sync service, for instance, can map group membership onto per-user storage limits, so a group named for a quota tier is doing configuration rather than authorization.

This is legitimate and useful, but it is worth naming such groups so their purpose is obvious, and keeping them in their own part of the tree rather than mixed in with permission groups. When a group means "how much space this person gets" rather than "what this person may do," someone reading the group list should be able to tell at a glance.

The identity provider should be the single source of truth for who is in what group, and applications should be downstream consumers that translate group membership into their own permissions. When an application's local permissions drift out of sync with the group that granted them, the model has broken.

Configuring Discourse (identity-only pattern)

Pattern: identity only. Discourse authenticates users through Keycloak but does no group or role mapping. The IdP answers one question, "who is this user," and Discourse handles authorization internally with its own trust levels and groups. Use this pattern for any application that needs single sign-on but manages permissions on its own.

Note: At the time of this writing, Discourse does have the capability for groups to be managed by an IdP, but it uses a Discourse-specific HMAC-signed payload format, not OIDC.

Keycloak side

Create a client in the realm:

Setting	Value	Why
Client ID	discourse	Matches the client ID configured in Discourse.
Client authentication	On (confidential)	Discourse holds a client secret and exchanges the auth code on a back channel.
Standard flow	Enabled	Authorization-code flow.
PKCE method	S256	Proof key for code exchange; hardens the code flow against interception.
Valid redirect URIs	https://forum.example.org/auth/oidc/callback	Where Keycloak returns the user after login.

No client scopes beyond the defaults are needed. Because Discourse does not consume groups, there is no group-membership mapper and no custom scope to create. This is the whole reason the identity-only pattern is the simplest one: you stop after the client exists.

Discourse side

Discourse's OpenID Connect support is provided by its official connector, configured entirely from the admin settings UI (**Admin → Settings → search "openid"**). The relevant settings:

Setting	Value	Notes
openid connect enabled	checked	Turns on the connector.
openid connect discovery document	<code>https://auth.example.org/realms/landisfam/.well-known/openid-configuration</code>	Discourse resolves all endpoints from this; you never hand-configure token or authorize URLs.
openid connect client id	<code>discourse</code>	Must match the Keycloak client ID.
openid connect client secret	<i>(from Keycloak, stored in the password manager)</i>	The confidential client's secret.
openid connect authorize scope	<code>openid email profile</code>	The three standard scopes. No <code>groups</code> here, by design.
openid connect match by email	checked	Links a Keycloak identity to an existing Discourse account by email address.
openid connect use pkce	checked	Must match the S256 method set on the Keycloak client.

Gotchas

- **Match-by-email is a decision, not a default.** Enabling "match by email" means a user who already had a local Discourse account gets linked to their Keycloak identity on first SSO login, rather than getting a second, empty account. Leave it off and you can end up with duplicate accounts. Turn it on only if you trust that email addresses in Keycloak are verified, because email-matching is exactly as trustworthy as your IdP's email verification.
- **PKCE has to agree on both ends.** If Keycloak requires S256 and Discourse doesn't send a PKCE challenge (or vice versa), login fails. Set it in both places.

Disabling local login

Once you have confirmed the configuration is working, you can disable local logins and force SSO only.

- Go to admin -> all site settings -> search for "enable local login."
- Uncheck "enable local logins" and "enable local logins via email."
- Then search all site settings for "allow new registrations" and uncheck that as well.

Make sure you have saved your local login credentials somewhere safe, like in a password vault, in case you ever get locked out.

To enable if locked out

If you ever get locked out and need to enable local logins to fix, navigate to the directory where your discourse launcher file sits.

- Type `./launcher enter app`
- Then `rails c`
- Then `SiteSetting.enable_local_logins = true`

Adapt this for...

Any application that offers "OIDC / OpenID Connect login" as a plugin or built-in feature and handles its own authorization, e.g., forums, wikis without group needs, dashboards, status pages. The recipe is always the same three moves: confidential client in Keycloak, point the app at the discovery document, request `openid email profile`. If the app never needs to know a user's *role* from the IdP, you are done here.

If you want to install your own discourse

Discourse has this pretty well [documented](#).

Configuring BookStack (group-aware pattern)

Pattern: group-aware. BookStack authenticates users through Keycloak and reads their group membership from the token, mapping Keycloak groups onto BookStack roles so that access is driven entirely by the IdP. This is the pattern for any app that should grant permissions based on what groups a user belongs to.

Keycloak side

The client

Setting	Value	Why
Client ID	bookstack	Matches <code>OIDC_CLIENT_ID</code> in the app.
Client authentication	On (confidential)	Back-channel code exchange with a client secret.
Standard flow	Enabled	Authorization-code flow.
PKCE method	S256	Matches the app.
Valid redirect URI	<code>https://bookstack.example.org/oidc/callback</code>	App callback.
Valid post-logout redirect URI	<code>https://bookstack.example.org/</code>	Where the user lands after single logout.

BookStack accepts **only RS256** for token signing. Keycloak signs with RS256 by default, so this usually needs no change, but it is worth confirming: if the realm or client is ever set to a different algorithm, BookStack rejects the tokens as invalid even though everything else is correct.

The group scope and mapper

Step 1: Create the client scope

In your Keycloak realm, go to **Client scopes** and click **Create client scope**.

On the **Settings** tab:

Field	Value	Why
Name	groups	This is the name the application requests. It must match <code>OIDC_ADDITIONAL_SCOPES</code> in BookStack exactly.
Type	Default	Applied automatically to clients it is assigned to, without the client having to ask for it per request.
Include in token scope	On	Without this the scope is not honored when requested.
Include in OpenID Provider Metadata	On	Advertises the scope in the discovery document.

Then go to the **Mappers** tab, click **Add mapper -> By configuration**, and select **Group Membership**:

Field	Value	Why
Name	kc_groups	A label for the mapper itself.
Token Claim Name	kc_groups	The key the group list arrives under in the token. This must match <code>OIDC_GROUPS_CLAIM</code> in BookStack. Note it differs from the scope name above.
Full group path	On	Sends <code>/bookstack/admin</code> rather than <code>admin</code> . See the note on group matching below.
Add to ID token	On	The setting that matters most. BookStack reads groups from the ID token; if this is off, everything else is correct and BookStack still sees nothing.
Add to access token	On	
Add to userinfo	On	

Step 2: Assign the scope to the client

Go to **Clients**, click your `bookstack` client, and open the **Client scopes** tab.

1. Click **Add client scope** and select the `groups` scope you just created.
2. In the scope list, click the dropdown beside `groups` and choose **Default**.

Assigning it as *Default* rather than *Optional* means it is included on every authentication without the client needing to request it explicitly.

Step 3: Create the groups

Under **Groups**, create a parent group `bookstack` with children `admin`, `editor`, and `viewer`. With full group path enabled, membership arrives in the token as `/bookstack/admin` and so on. A user's membership in these groups is what drives their BookStack role.

BookStack side

BookStack is configured through environment variables (here, in its `docker-compose.yml`). The OIDC-relevant ones, secrets redacted:

```
environment:
  - AUTH_METHOD=oidc
  - AUTH_AUTO_INITIATE=false          # show the login page; don't force-redirect to
Keycloak
  - OIDC_NAME=LandisFam SSO
  - OIDC_DISPLAY_NAME_CLAIMS=name
  - OIDC_CLIENT_ID=bookstack
  - OIDC_CLIENT_SECRET=REDACTED
  - OIDC_ISSUER=https://auth.example.org/realms/landisfam
  - OIDC_ISSUER_DISCOVER=true        # resolve endpoints from the discovery document
  - OIDC_ADDITIONAL_SCOPES=groups    # request the 'groups' scope (must exist in Keycloak)
  - OIDC_GROUPS_CLAIM=kc_groups      # read groups from the 'groups' claim
  - OIDC_USER_TO_GROUPS=true         # map incoming groups onto BookStack roles
  - OIDC_END_SESSION_ENDPOINT=true   # enable single logout
```

For group mapping to take effect, BookStack roles must exist whose names match the Keycloak group names it receives (`admin`, `editor`, `viewer`). BookStack matches the incoming group to a role of the same name.

“ A note on `OIDC_REMOVE_FROM_GROUPS`: this option makes Keycloak the sole authority on group membership, removing a user from BookStack roles they are no longer in on the IdP. It is powerful and correct in principle, but enable it only *after* group mapping is confirmed working, or a misconfigured claim can strip everyone's access on their next login.

Map bookstack roles to Keycloak groups

BookStack matches incoming groups against each role's External Authentication ID, not against the role name. In BookStack, go to Settings -> Roles, and edit each role mapping the External Authentication ID to the full Keycloak group path:

Bookstack Role	External Authentication ID
Admin	/bookstack/admin
Editor	/bookstack/editor
Viewer	/bookstack/viewer

Adapt this for...

Any application that maps IdP groups to internal roles: wikis, dashboards, file platforms, ticketing systems. The shape is always: create a real client *scope* (not just a mapper) whose name matches what the app requests, put the group-membership mapper on it, and confirm the claim lands in the specific token the app reads.

Configuring a Flask app (build-it-yourself pattern)

Pattern: build it yourself. Discourse and BookStack are someone else's applications where you configure OIDC. When the application is your own code, *you* are the OIDC client: you write the login redirect, the callback, the token validation, and the claim reading. This is more work, but it is also the most control, and the pattern generalizes to any framework, not just Flask.

“ The snippets below are a representative, working Authlib integration that matches the Keycloak client and group/role structure documented for this environment. Treat them as a reference skeleton to reconcile against your own app's code, not a verbatim dump of a specific file.

Keycloak side

Same as any confidential client:

Setting	Value
Client ID	my - app
Client authentication	On (confidential)
Standard flow	Enabled
PKCE method	S256
Valid redirect URI	https://app.example.com/auth/callback

For an app that makes authorization decisions from groups and roles, add the same `groups` **client scope + Group Membership mapper** described on the BookStack page (Token Claim Name `groups`, Full group path On, Add to ID token: On). Since you control the code, you get to decide which token that is, and the ID token is the natural choice, since Authlib parses and validates it for you.

Groups and roles for this client:

```
Groups: families/<familyID>/{administrator,member}
Roles:  family_administrator, family_member, site_admin
```

Flask side

Using [Authlib](#), which handles discovery, PKCE, the code exchange, and ID-token validation so you are not hand-rolling crypto.

1. Register the provider

Point Authlib at the discovery document and let it configure endpoints, keys, and PKCE automatically.

```
from authlib.integrations.flask_client import OAuth

oauth = OAuth(app)
oauth.register(
    name="keycloak",
    client_id="my-app",
    client_secret=app.config["OIDC_CLIENT_SECRET"],  # from env / secrets manager, never
    hard-coded
    server_metadata_url=(
        "https://kc.example.com/realms/<realm_name>/",
        ".well-known/openid-configuration"
    ),
    client_kwargs={
        "scope": "openid email profile groups",  # request the groups scope
        "code_challenge_method": "S256",        # PKCE, matches the Keycloak client
    },
)
```

2. The login route

Kick off the authorization-code flow. Authlib builds the redirect, the state, and the PKCE challenge.

```
from flask import url_for

@app.route("/login")
def login():
    redirect_uri = url_for("auth_callback", _external=True)
    return oauth.keycloak.authorize_redirect(redirect_uri)
```

3. The callback route

Exchange the code for tokens (back channel), then read the validated ID-token claims. This is where identity and authorization are established.

```
from flask import session, redirect, url_for

@app.route("/auth/callback")
def auth_callback():
    token = oauth.keycloak.authorize_access_token() # code -> tokens, ID token validated
    claims = token["userinfo"] # parsed, verified ID-token claims

    session["user"] = {
        "sub": claims["sub"], # stable unique ID; use this as the
key
        "username": claims.get("preferred_username"),
        "email": claims.get("email"),
        "groups": claims.get("groups", []), # the group claim you configured
    }
    return redirect(url_for("dashboard"))
```

A detail that matters: key the user record on `sub`, not on email or username. `sub` is Keycloak's stable, immutable identifier; emails and usernames can change, and keying on them means a user who updates their email becomes a stranger to your app.

4. Turning groups into authorization

The claim gives you a list of group paths. Map them to whatever your app's access model is. Keep the mapping in one place so authorization logic isn't scattered:

```
from functools import wraps

from flask import session, abort
```

```

def require_group(*allowed):
    def decorator(view):
        @wraps(view)
        def wrapped(*args, **kwargs):
            user = session.get("user")
            if not user:
                abort(401)
            user_groups = set(user.get("groups", []))
            if not user_groups.intersection(allowed):
                abort(403)
            return view(*args, **kwargs)
        return wrapped
    return decorator

# usage
@app.route("/admin")
@require_group("/site_admin")
def admin_panel():
    ...

```

5. Logout

Clear the local session and, ideally, hit Keycloak's end-session endpoint so single logout actually ends the IdP session too, not just the local one.

```

@app.route("/logout")
def logout():
    session.clear()
    end_session = (
        "https://kc.example.com/realms/<realm_name>/protocol/"
        "openid-connect/logout?post_logout_redirect_uri="
        "https://home.example.org/"
    )
    return redirect(end_session)

```

Gotchas

- **Which token, again.** The same lesson as BookStack: your code reads groups from wherever you look. Authlib gives you the validated ID-token claims via `token["userinfo"]`, so configuring the mapper with **Add to ID token: On** is what makes `claims["groups"]` populated. If it's empty, the claim is in the wrong token.
- **Validate, don't trust.** Let Authlib verify the ID token's signature and issuer. Do not decode the token yourself and read claims from an unverified payload; that turns a signed assertion into an attacker-controllable input.
- **sub is the identity.** Keying on email or username is not the best practice, and it surfaces as "a user changed their email and lost all their data."
- **Full group path on/off changes the string.** If Keycloak sends `/site_admin` (full path on) and your code checks for `site_admin` (no slash), the comparison silently fails. Pick one convention and make the app match it.

Adapt this for...

Any application you write, in any framework. The five routes are fairly universal: register the provider from its discovery document, redirect to authorize, exchange the code in a callback, read validated claims, and enforce authorization from those claims. The library changes (Authlib for Flask/Python, equivalent OIDC libraries elsewhere) but the flow and the gotchas do not. The single most important rule when you own the code: never read claims from an unvalidated token, and always let a real OIDC library do the validation.

Appendix A - SSO Test App

Clone the repo into a location capable of serving a python flask application.

```
git clone https://git.landisfam.org/landisfam/ssotest
```

Create a virtual environment and install the requirements.

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Create a .env file with information for your instance.

```
FLASK_SECRET = "your_secret_key"           # Just make something up, this is
only for test
KEYCLOAK_URL = "https://kc.example.com"     # Update to your Keycloak site
CLIENT_ID = "test-app"
CLIENT_SECRET = "from_keycloak"            # Copy / paste from your Keycloak
client
REDIRECT_URI = "https://test-app.example.com/callback" # Update to the page where you're
hosting
```