

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.

Revision #3

Created 2026-07-20 19:33:00 UTC by Chris Landis

Updated 2026-08-29 05:40:42 UTC by Chris Landis