

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.

Revision #5

Created 2026-07-23 15:49:25 UTC by Chris Landis

Updated 2026-07-24 03:06:27 UTC by Chris Landis