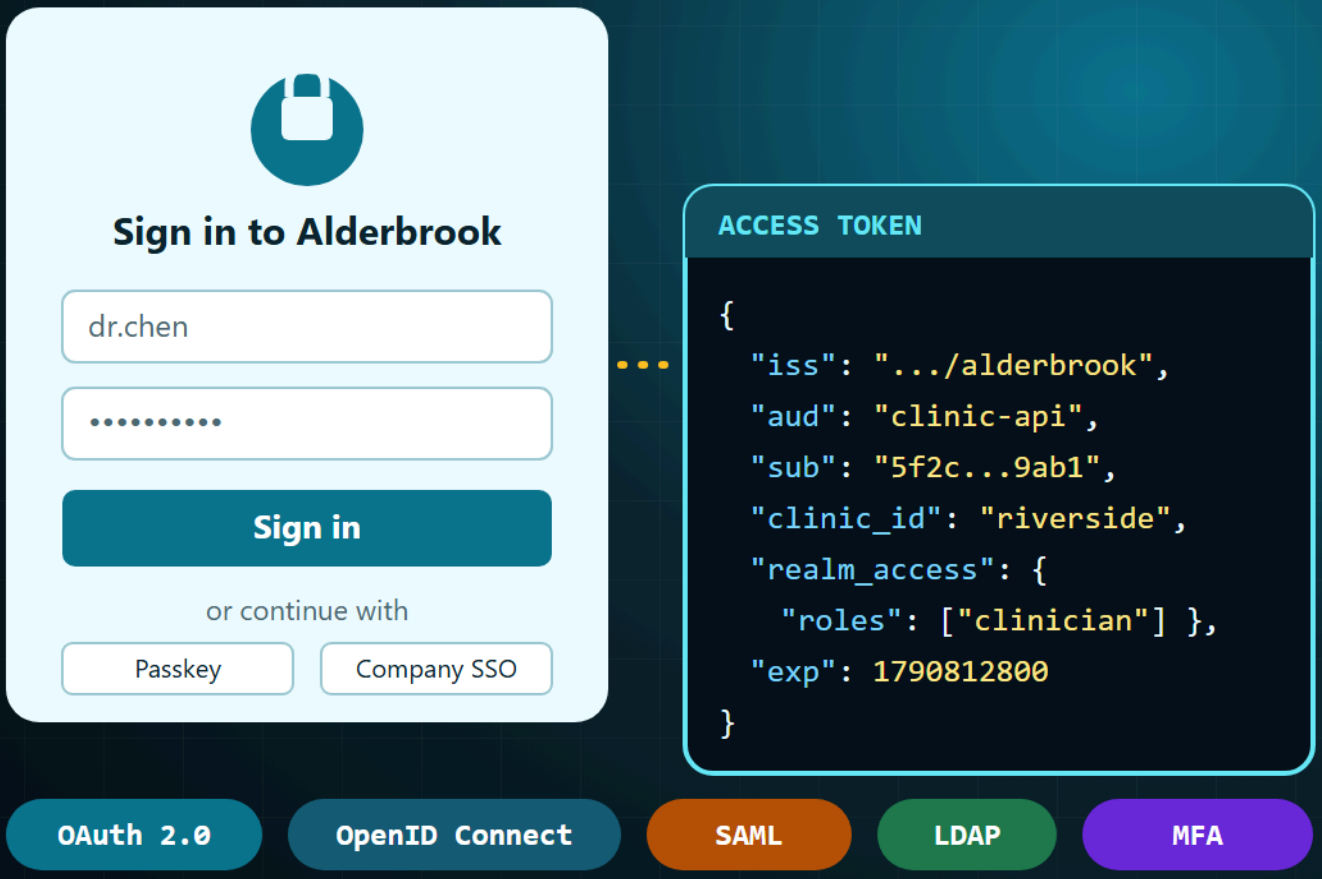


SSO · PKCE · roles · federation · passkeys

THE COMPLETE PRACTICAL GUIDE

Keycloak in Practice

Identity and access management for modern applications, from first login to production

Single sign-on · OAuth 2.0 · OpenID Connect · SAML
MFA and passkeys · Authorization · Kubernetes

SECURE · FEDERATE · AUTHORIZE · OPERATE

THENEXBOOKS

Keycloak in Practice

The Complete Practical Guide to Identity and Access Management
Single sign-on, OAuth 2.0, OpenID Connect, SAML, and production operations

First Edition

THENEXBOOKS

Keycloak in Practice
The Complete Practical Guide to Identity and Access Management

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code and configuration examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Keycloak changes between releases: option names, defaults, console layouts, and feature status may differ from what is described here. Always consult the official documentation and release notes for the version you run. Security configurations must be reviewed for your own environment. Nothing in this book is legal or compliance advice.

Keycloak is an open-source project; this book is independent and is not affiliated with, sponsored by, or endorsed by the Keycloak project, the Cloud Native Computing Foundation, Red Hat, or any other organization. All product names, logos, and brands are property of their respective owners and are used for identification only.

Alderbrook Health, the organization used as the running example in this book, is fictional, as are its clinics, partners, people, and data. The domains `alderbrook.example` and `example.com` are used for illustration only.

Preface

Every application eventually needs to answer two questions: *who is this?* and *what are they allowed to do?* The first version of the answer is often a users table and a login form. Then a second application arrives, then a mobile app, then an API that partners call, then a request for single sign-on, multi-factor authentication, and login with the company directory. Suddenly identity is not a feature of one application; it is a platform that every application depends on.

Keycloak is a widely used open-source platform for exactly that job. It implements the standards that modern applications speak — OAuth 2.0, OpenID Connect, and SAML 2.0 — and it provides login pages, user management, social and enterprise federation, multi-factor authentication, fine-grained authorization, and an administration API. Getting a first login working takes minutes. Understanding the protocols well enough to secure real applications, designing a clean authorization model, and running Keycloak reliably in production takes considerably longer.

This book covers that whole path. We start from the problems identity management solves and the protocols in plain language, then secure every kind of application, manage users and authentication, design authorization, customize and automate Keycloak, and finally deploy and operate it in production. Throughout, we explain *why* each setting exists, so that you can make good decisions when your situation differs from ours.

Who This Book Is For

- **Application developers** who need to add login, single sign-on, and API security to web, mobile, and back-end applications.
- **DevOps and platform engineers** who deploy, automate, and operate Keycloak.
- **Architects and tech leads** designing identity and authorization for a portfolio of applications.
- **Security-minded engineers** who want a clear, practical understanding of OAuth 2.0 and OpenID Connect.

You should be comfortable with basic web development and HTTP. No prior knowledge of OAuth, OpenID Connect, SAML, or Keycloak is required.

What You Will Learn

- The concepts behind identity and access management, and OAuth 2.0 and OpenID Connect explained without jargon.
- Running Keycloak locally and configuring realms, clients, users, roles, and groups.
- Securing single-page apps, APIs, server-side web apps, services, and mobile apps with the right flow for each.
- Registration, required actions, password policies, one-time codes, WebAuthn and passkeys, and custom authentication flows.

- Social login, external identity providers, LDAP and Active Directory federation, and SAML applications.
- Designing roles, groups, and scopes; shaping tokens with mappers; and fine-grained authorization services.
- Multi-tenancy, themes, automation with the Admin REST API and Terraform, configuration as code, and custom extensions.
- Production deployment, high availability, Kubernetes, monitoring, hardening, upgrades, and troubleshooting.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains identity management and the protocols, gets Keycloak running, and protects a first application. **Part II — Securing Applications** (Chapters 6–10) covers single-page apps, APIs, server-side apps, service-to-service calls, and mobile apps. **Part III — Users and Authentication** (Chapters 11–15) covers user management, authentication flows, brokering, directory federation, and SAML. **Part IV — Authorization and Tokens** (Chapters 16–19) designs the authorization model, shapes tokens, and covers fine-grained authorization and multi-tenancy. **Part V — Customization and Automation** (Chapters 20–24) covers themes, automation, configuration as code, extensions, and auditing. **Part VI — Production** (Chapters 25–28) deploys and operates Keycloak and closes with a complete case study. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Alderbrook Health

Throughout the book we build the identity platform for **Alderbrook Health**, a fictional network of outpatient clinics. Patients use a web portal and a mobile app to book appointments and see results. Nurses, doctors, and front-desk staff use a staff console and sign in with their corporate directory accounts. A clinic API serves all of these applications, and a partner laboratory sends results through a service integration and signs its staff in through SAML. By the end of the book, all of it runs through a single, well-designed Keycloak realm — with multi-factor authentication, clear roles, audit events, and a highly available production deployment.

Versions and Conventions

Examples target recent Keycloak releases built on Quarkus, running in containers with PostgreSQL. Keycloak evolves quickly: option names, defaults, Admin Console labels, and the status of newer features change between releases. We describe console navigation in general terms, flag features whose status may differ in your version, and point you to the official guides where details matter. Application examples use a React single-page app with `keycloak-js`, a Node.js API with `jose`, and Spring Boot with Spring Security.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or a misconfiguration that can weaken security or cause outages.

Code and configuration appear in `monospace`. Long lines, URLs, and tokens are wrapped or shortened with `...` to fit the page.

Contents

Preface

PART I — FOUNDATIONS

1 Why Identity Matters: IAM, Single Sign-On, and Where Keycloak Fits

2 OAuth 2.0 and OpenID Connect in Plain Words

3 Getting Started: Running Keycloak Locally with Docker

4 Core Concepts: Realms, Clients, Users, Roles, and Groups

5 Your First Protected App: Logging In with OpenID Connect

PART II — SECURING APPLICATIONS

6 Single-Page Applications: Authorization Code Flow with PKCE

7 Securing APIs: Access Tokens, Audiences, and Token Validation

8 Server-Side Web Apps and the Backend-for-Frontend Pattern

9 Service-to-Service Security: Client Credentials and Service Accounts

10 Mobile and Native Apps

PART III — USERS AND AUTHENTICATION

11 Managing Users: Registration, Profiles, and Required Actions

12 Authentication Flows: Passwords, One-Time Codes, and WebAuthn Passkeys

13 Identity Brokering: Social Login and External Identity Providers

14 User Federation with LDAP and Active Directory

15 SAML 2.0 Applications and Providers

PART IV — AUTHORIZATION AND TOKENS

16 Designing Your Authorization Model: Roles, Groups, and Scopes

17 Tokens in Depth: Claims, Mappers, Client Scopes, Lifetimes, and Logout

18 Fine-Grained Authorization Services: Resources, Policies, and Permissions

19 Multi-Tenancy: Realms, Organizations, and B2B Scenarios

PART V — CUSTOMIZATION AND AUTOMATION

20 Themes and Branding: Login Pages, Emails, and Account Console

21 Automating Keycloak: Admin REST API, kcadm, and Terraform

22 Configuration as Code: Realm Export, Import, and GitOps

23 Extending Keycloak with SPIs: Custom Authenticators and Event Listeners

24 Events, Auditing, and Security Monitoring

PART VI — PRODUCTION

25 Production Deployment: Database, Hostname, TLS, and Reverse Proxies

26 High Availability: Clustering, Caching, and the Kubernetes Operator

27 Operating Keycloak: Hardening, Upgrades, Backups, and Troubleshooting

28 Case Study: Building Alderbrook Health's Identity Platform

APPENDICES

A Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

Why identity deserves its own platform, OAuth 2.0 and OpenID Connect explained without jargon, Keycloak running on your machine, its core building blocks, and a first application protected end to end.

1 · Why Identity Matters
2 · OAuth 2.0 and OpenID Connect
3 · Getting Started
4 · Core Concepts
5 · Your First Protected App

CHAPTER 1 · PART I — FOUNDATIONS

Why Identity Matters: IAM, Single Sign-On, and Where Keycloak Fits

Before we run a single container, we need a clear picture of the problem Keycloak solves. This chapter explains authentication and authorization, why scattered logins hurt users and security teams alike, what single sign-on and federation really mean, and how an open source identity server like Keycloak fits into a modern application landscape, using the fictional Alderbrook Health as our guide.

A Morning at Alderbrook Health

Alderbrook Health is a fictional network of outpatient clinics. It runs two main sites, Riverside and Hillcrest, and it is growing quickly. Over the years, its small IT team has built or bought a handful of applications, and every one of them was written with its own idea of what a "user" is.

Consider an ordinary Monday. Dr. Chen arrives at the Riverside clinic and signs in to her workstation with her corporate directory account. She then opens the staff console, the web application that shows the day's schedule, and signs in again with a different username and a password she last changed eight months ago. At ten o'clock she needs a lab result, so she opens the partner lab's website and signs in a third time, with an account that the lab created for her by email. Meanwhile, Emma Lopez, a patient, tries to book a follow-up appointment on the patient portal, forgets her password, and waits for a reset email that the portal sends from a mail server nobody has checked in weeks.

Down the hall, Nurse Okafor from the Hillcrest clinic is covering a shift at Riverside. The staff console decides what she may see from a spreadsheet imported every Friday. Her record still says Hillcrest, so half of the screens she needs show an access error. And when Frontdesk Kim resigned last year, it took three weeks and four separate tickets before every one of her accounts was disabled.

None of these problems comes from bad intentions or careless people. They come from a missing piece of architecture: a single, trustworthy place that answers two questions for every application: *who is this person?* and *what are they allowed to do?* That missing piece is what the industry calls **Identity and Access Management**, or IAM, and it is what this book is about.

Throughout the book we will follow Alderbrook as it replaces its patchwork of logins with a single identity platform built on Keycloak. The healthcare setting is fictional and simplified. It gives us realistic requirements such as patient self-registration, staff single sign-on, multi-factor authentication, partner federation, and audit trails, without pretending to be legal advice. Real healthcare systems are subject to regulations that may apply to your situation, and meeting them always requires qualified review; no configuration in this book achieves compliance by itself.

Authentication and Authorization

Two words sit at the heart of every identity discussion, and they are easy to mix up because they both start with "auth". Getting the difference right will make every later chapter easier.

Authentication answers the question "who are you?" It is the process of verifying that someone, or something, is the identity it claims to be. When Dr. Chen types her password, taps a push notification, or touches a security key, she is authenticating. The output of authentication is a verified identity: the system now believes, with some level of confidence, that this session belongs to `dr.chen`.

Authorization answers the question "what may you do?" Once we know who someone is, we decide whether they may read a patient record, write a prescription, or change a clinic's opening hours. Authorization depends on authentication, because you cannot decide what someone may do until you know who they are, but it is a separate decision made with separate information: roles, group memberships, attributes, the resource being accessed, and sometimes the context of the request.

A useful analogy is a hospital building. The security desk checks your badge photo against your face: that is authentication. The badge reader on the pharmacy door decides whether your badge opens it: that is authorization. The desk and the door can be run by different teams, but they must agree on what a badge means.

Aspect	Authentication	Authorization
Question	Who are you?	What may you do?
Typical inputs	Password, one-time code, passkey, certificate	Roles, groups, attributes, resource, context
Result	A verified identity and a session	Allow or deny a specific action
Failure code in HTTP APIs	401 Unauthorized	403 Forbidden
Where Keycloak helps	Login pages, MFA, federation, sessions	Roles and claims in tokens, optional policy engine

Notice the unfortunate naming of HTTP status 401: it says "Unauthorized" but it really means "unauthenticated", as in "we do not know who you are, please present credentials". Status 403 means "we know who you are, and the answer is still no". When we build the Alderbrook clinic API in Chapter 7, we will return these codes deliberately, and the distinction will matter to the frontend developers who have to react to them.

> **NOTE**
>
> Keycloak is excellent at authentication and at *carrying* authorization data (roles, groups, and claims) to your applications inside tokens. The final authorization decision, such as "may this clinician read this particular record?", is usually enforced in your application or API. Keycloak also offers an optional policy engine called Authorization Services, covered in Chapter 18, but most systems start with roles and claims.

Identity Silos and Their Costs

When each application keeps its own users table, its own password hashing, and its own reset flow, we get **identity silos**. Every silo is a small island with its own rules. Silos are the natural result of building applications one at a time, and almost every organization has them. The trouble is that their costs grow quietly with each new application.

Costs for users

Users face password fatigue. Each silo has its own password rules, so people reuse passwords, write them down, or pick the weakest thing the policy allows. Each silo also has its own "forgot password" process, and each one is a place where users get stuck. For patients, a frustrating login may mean a missed appointment. For staff, it means minutes lost many times a day.

Costs for security

Every silo stores credentials, which means every silo is a target. A single application with outdated password hashing can leak credentials that also work elsewhere because people reuse them. Adding multi-factor authentication means adding it N times, usually inconsistently. Detecting brute-force attempts requires N separate log streams. Most dangerously, offboarding becomes a scavenger hunt: when someone leaves, their access must be removed from every silo, and the one that gets forgotten becomes an open door.

Costs for developers

Developers pay a steady tax. Every new application needs login pages, password storage, email verification, session handling, account lockout, and an admin screen for resetting passwords. None of it is the application's real purpose, and all of it is hard to get right.

Auditors suffer too: "who could access patient records last March?" becomes a research project when access is spread across systems with different log formats.

Alderbrook problem	Root cause	What a central IdP changes
Staff sign in three times a day	Separate accounts per app	One login, then SSO across apps
Slow offboarding	Accounts in many silos	Disable once, access ends everywhere

Inconsistent MFA	Each app implements its own	MFA policy set in one place
Wrong clinic access	Manual spreadsheet imports	Groups and attributes in tokens
Partner lab accounts	Lab creates local accounts	Federation with the partner IdP
No audit answer	Scattered, unaligned logs	Central login and admin events

Centralizing Identity: The Identity Provider

The fix for silos is to move identity out of individual applications and into a dedicated service called an **identity provider**, or IdP. The IdP owns user accounts, credentials, login pages, MFA, and sessions. Applications stop handling passwords entirely. Instead, when a user needs to sign in, the application redirects the browser to the IdP. The IdP authenticates the user and then sends the browser back to the application with a verifiable statement: "this is `dr.chen`, she authenticated at 08:02 with a password and a one-time code, and she holds the role `clinician`".

Those statements are usually delivered as **tokens**. In the OpenID Connect world, which we explore in Chapter 2, the statement about the user is an *ID token*, and the credential an application uses to call an API is an *access token*. Both are typically JSON Web Tokens, signed by the IdP so that anyone can verify they have not been tampered with. Here is an abbreviated, decoded example of the kind of access token Alderbrook's staff console will eventually receive:

Decoded access token payload (abbreviated)

```
{
  "iss": "https://auth.alderbrook.example/realms/alderbrook",
  "sub": "5c1f3a0e-8a9d-4a53-9a57-2b1f6f0e7c11",
  "aud": "clinic-api",
  "azp": "staff-console",
  "exp": 1790000300,
  "iat": 1790000000,
  "preferred_username": "dr.chen",
  "email": "dr.chen@alderbrook.example",
  "clinic_id": "riverside",
  "realm_access": { "roles": ["staff", "clinician"] },
  "resource_access": {
    "clinic-api": { "roles": ["appointments:read", "records:read"] }
  },
  "scope": "openid profile email"
}
```

Do not worry about every field yet. The important idea is that the application no longer needs to know Dr. Chen's password. It needs to know only how to validate a token issued by a trusted IdP, and how to read the claims inside it. The `iss` claim says who issued the token, `aud` says which service it is meant for, `exp` says when it expires, and the role claims carry the authorization data that the clinic API will use for its decisions.

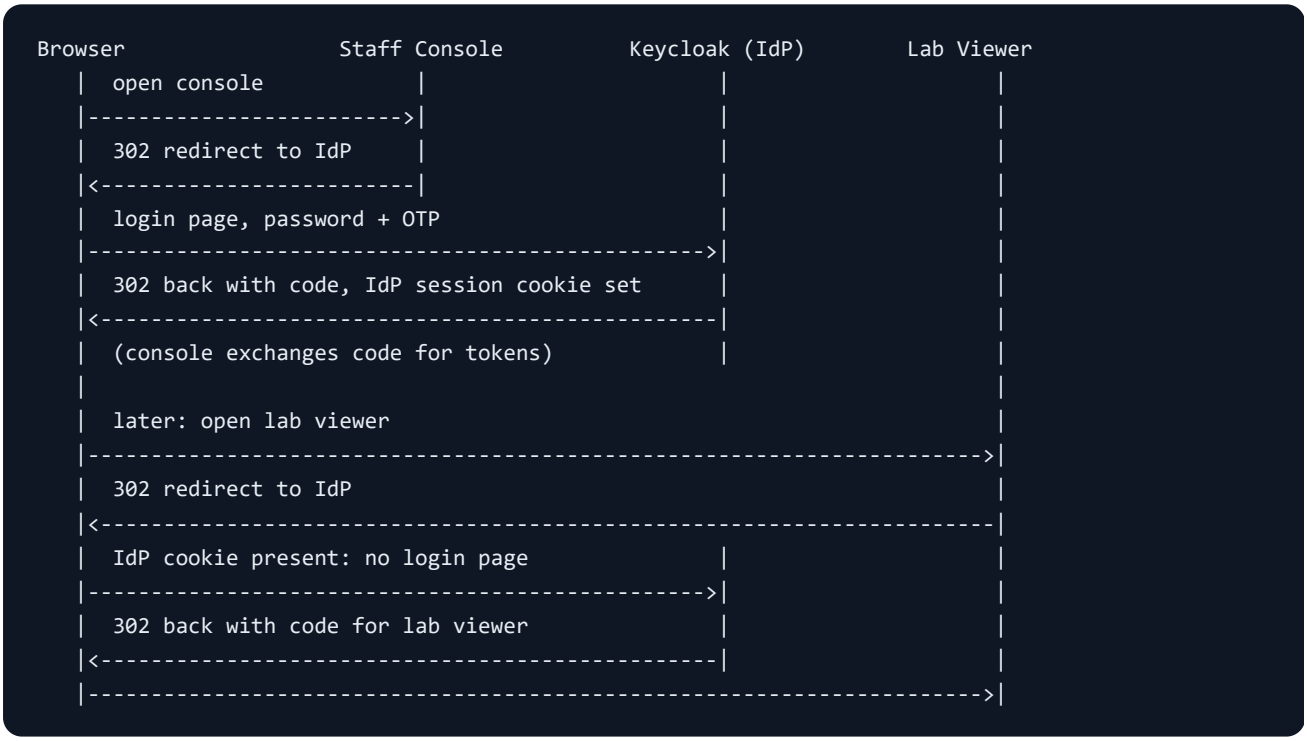
Because the IdP is now the only component that sees passwords, it becomes the one place where we invest in strong credential storage, rate limiting, brute-force protection, MFA, and auditing. Improving it improves every connected application at once.

Single Sign-On

Single sign-on, or SSO, is the most visible benefit of a central IdP. With SSO, a user authenticates once and then moves between applications without being asked for credentials again, for as long as their IdP session lasts.

SSO works because the IdP keeps its own session, usually via a cookie set on the IdP's domain. When Dr. Chen signs in to the staff console in the morning, the browser visits the IdP, she authenticates, and the IdP sets its session cookie. When she later opens a second application that trusts the same IdP, that application also redirects her browser to the IdP. This time the IdP sees its session cookie, recognizes her, and immediately redirects back with fresh tokens for the second application. She may notice a brief flicker in the address bar, but she is never asked for her password.

SSO across two applications



SSO has a mirror image: **single logout**. If a user signs out of one application, should they be signed out of all of them? In a clinic with shared workstations, the answer is usually yes. Single logout is harder than single sign-on, because the IdP must notify every application that holds a session. Keycloak supports both front-channel logout, in which the browser visits each application, and back-channel logout, in which Keycloak calls applications directly from the server. Chapter 8 and Chapter 17 cover the trade-offs in detail.

> **TIP**
>
> SSO is not the same thing as "the same password everywhere". Syncing one password to many systems still leaves every system storing and checking credentials. With true SSO, only the IdP ever sees the password, and applications receive signed tokens instead.

Federation: Trusting Identities You Do Not Own

SSO solves logins across applications that trust one IdP. **Federation** extends the idea across organizational boundaries: your IdP agrees to trust statements made by another IdP. This is how "Sign in with Google" works, and it is also how business partners share access without creating accounts for each other.

Alderbrook has three federation needs, and together they illustrate the main patterns:

- **Social login for patients.** Many patients would rather sign in with an existing account, such as a Google account, than create yet another password. Keycloak can act as a broker: the patient chooses Google on Alderbrook's login page, authenticates there, and Keycloak creates or links a local account. Alderbrook's applications still receive tokens from Keycloak, not from Google, so they do not need to know that Google was involved. We configure this identity provider, called `google`, in Chapter 13.
- **The corporate directory for staff.** Staff accounts already live in Alderbrook's LDAP-compatible directory. Rather than copying passwords, Keycloak can connect to the directory as a *user federation* provider called `corp-directory`, look up users, and verify passwords against it. This is technically different from brokering, since the directory is a user store, not an IdP with its own login page, and Chapter 14 explains the details.
- **Partner labs.** Northgate Labs, a fictional partner, runs its own SAML identity provider. Instead of Alderbrook creating accounts for Northgate's technicians, Keycloak trusts Northgate's IdP through an identity provider named `partner-lab-saml`. When Northgate hires or dismisses a technician, access at Alderbrook follows automatically. Chapter 15 covers SAML in depth.

Federation relies on open standards. The main ones you will meet are **OAuth 2.0**, a framework for delegated authorization; **OpenID Connect** (OIDC), an identity layer built on top of OAuth 2.0; and **SAML 2.0**, an older XML-based standard still common in enterprises and healthcare. Keycloak speaks all three, which is one of its main strengths. Chapter 2 explains OAuth 2.0 and OIDC in plain words.

Build or Buy?

Every team that faces identity silos eventually asks whether to build its own login system, buy a commercial identity service, or adopt an open source identity server. There is no universally correct answer, but there are clear trade-offs.

Building your own

Building a login system looks simple: a users table, a hashed password column, and a form. The complexity arrives later: password hashing you can strengthen over time, lockout and rate limiting, email verification, single-use reset tokens, MFA with recovery, multi-device sessions, and eventually SSO, which means implementing OAuth 2.0 and OpenID Connect correctly. Each piece has well-known pitfalls, and the result is only as secure as its least-reviewed part.

Buying a hosted service

Commercial identity-as-a-service platforms handle operations for you and often provide polished developer experiences. The trade-offs are cost that grows with the number of users, less control over where data lives and how features behave, and a dependency on a vendor's roadmap and pricing. For some organizations these are acceptable; for others, especially those with data residency requirements or very large user populations, they are not.

Running an open source identity server

An open source identity server like Keycloak sits in between. You get a mature, standards-based product without license fees, and you keep full control over deployment, data, and customization. In exchange, you take on operations: running the servers and the database, upgrading, backing up, monitoring, and securing the deployment. Part VI of this book is dedicated to exactly that work, because it is the part teams most often underestimate.

Criterion	Build	Hosted service	Keycloak
Upfront effort	High	Low	Medium
Operational effort	High	Low	Medium to high
Standards support	Whatever you implement	Usually broad	OIDC, OAuth 2.0, SAML 2.0
Control over data	Full	Depends on vendor	Full
Customization	Unlimited, at your cost	Within vendor limits	Themes, SPIs, and APIs
License cost	None	Subscription	None (Apache 2.0)

Alderbrook chose Keycloak for three reasons. First, it needs SAML federation with partner labs and LDAP integration with its directory, both of which Keycloak supports out of the box. Second, it wants to keep identity data in infrastructure it controls. Third, its platform team already runs containers and PostgreSQL, so the operational burden is manageable. Your own reasons may differ, and that is fine: the point is to decide deliberately, with operations in the budget.

> **WARNING**
>
> "Open source" does not mean "free to operate". An identity server is critical infrastructure: if it is down, nobody can sign in to anything. Plan for high availability, backups, upgrades, and security monitoring from the start, and budget people's time for them. Treat the identity platform with the same care as your production database.

What Keycloak Is

Keycloak is an open source identity and access management server. It is licensed under the Apache License 2.0 and is a Cloud Native Computing Foundation (CNCF) project, with an active community and regular releases. In practical terms, Keycloak is a Java application that you run as a server, typically in a container, backed by a relational database such as PostgreSQL.

The current distribution is built on Quarkus, a modern Java framework, and has been since Keycloak 17. If you find older tutorials that talk about WildFly, `standalone.xml`, or URLs that start with `/auth`, they describe the legacy distribution, which is gone. In this book we target recent Keycloak releases in the 26.x era. Keycloak evolves quickly, so wherever a feature's details depend on the version, we say so and suggest checking the release notes and documentation for the version you run.

Main capabilities

- **Standard protocols.** OpenID Connect, OAuth 2.0, and SAML 2.0, so standards-compliant libraries and products can integrate with it.
- **Login and account pages.** Hosted login, registration, password reset, and a self-service Account Console.
- **Strong authentication.** One-time passwords, WebAuthn security keys and passkeys, and configurable authentication flows.
- **Brokering and federation.** Social providers, other OIDC or SAML IdPs, and LDAP or Active Directory user stores.
- **Authorization data.** Realm, client, and composite roles, groups, attributes, and protocol mappers that shape token claims, plus optional fine-grained Authorization Services.
- **Multi-tenancy.** Isolated realms and, in recent versions, Organizations for business-to-business scenarios.
- **Administration and extensibility.** The Admin Console, the Admin REST API, `kcadm.sh`, themes, and Service Provider Interfaces (SPIs) for custom code.
- **Operations.** Clustering, health checks, metrics, and a Kubernetes Operator.

How applications talk to Keycloak

Keycloak publishes everything an application needs through standard, well-known endpoints. For example, an OpenID Connect discovery document describes the realm's endpoints and capabilities. Once Keycloak is running locally in Chapter 3, you will be able to fetch it like this:

Terminal

```
$ curl -s http://localhost:8080/realms/alderbrook/.well-known/openid-configuration \
    | jq '{issuer, authorization_endpoint, token_endpoint, jwks_uri}'
{
  "issuer": "http://localhost:8080/realms/alderbrook",
  "authorization_endpoint":
    "http://localhost:8080/realms/alderbrook/protocol/openid-connect/auth",
  "token_endpoint":
    "http://localhost:8080/realms/alderbrook/protocol/openid-connect/token",
  "jwks_uri":
    "http://localhost:8080/realms/alderbrook/protocol/openid-connect/certs"
}
```

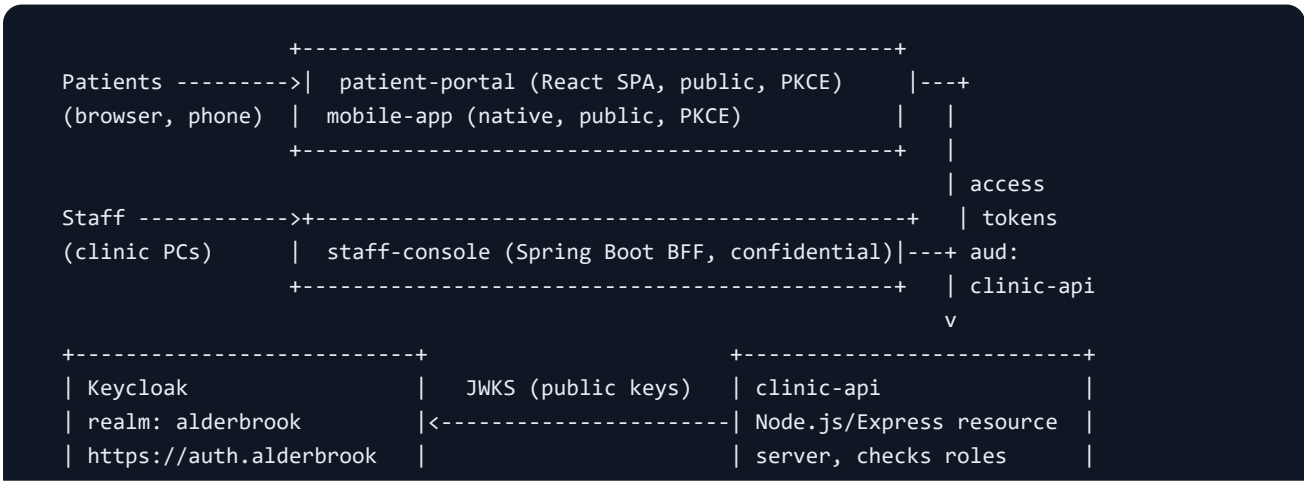
We wrapped long values for readability. Standard OIDC libraries read this document automatically, so integrating one usually takes only an issuer URL, a client ID, and, for confidential clients, a secret. Keycloak no longer ships Java servlet adapters; use standard libraries such as Spring Security. The JavaScript adapter `keycloak-js` is still provided as an npm package, and we use it in Chapter 6.

Keycloak organizes everything into *realms*, isolated spaces of users, applications (called *clients*), and settings; Chapter 4 covers the full vocabulary. Every installation starts with a realm called `master`. It is meant only for administering Keycloak itself. Alderbrook's applications and users live in their own realm, `alderbrook`, and you should follow the same practice: never put application users in `master`.

Alderbrook's Target Picture

With the vocabulary in place, we can sketch where Alderbrook is headed. The following diagram shows the target architecture that we build, piece by piece, over the course of the book. Chapter 28 revisits it as a complete case study.

Alderbrook Health: target identity architecture





Let us walk through the pieces and the requirements they satisfy.

Patients

Patients use the `patient-portal`, a React single-page application, and the `mobile-app`. Both are **public clients**: code that runs on a device the user controls cannot keep a secret, so they use the Authorization Code flow with PKCE, which Chapter 2 explains. Patients can register themselves, verify their email, and optionally sign in with Google. They receive the realm role `patient`. Sample patients in our examples are `emma.lopez` and `raj.patel`, with fictional addresses at `example.com`.

Staff

Staff use the `staff-console`, a Spring Boot web application that keeps tokens on the server and gives the browser only a session cookie. That makes it a **confidential client** with its own secret. Staff accounts come from the corporate directory through `corp-directory`, and staff must use multi-factor authentication. Their permissions come from realm roles such as `staff`, `clinician`, and `admin`, and from groups: clinic membership through `/clinics/riverside` and `/clinics/hillcrest`, and job function through `/staff/nurses`, `/staff/doctors`, and `/staff/front-desk`. Our sample staff are `dr.chen`, `nurse.okafor`, `frontdesk.kim`, and the administrator `it.admin`.

The clinic API

The `clinic-api` is a Node.js/Express service that holds appointments and records. It never shows a login page. It only validates access tokens issued by Keycloak, checks that each token was issued for it (the `aud` claim must contain `clinic-api`), and enforces permissions using client roles such as `appointments:read`, `appointments:write`, `records:read`, `records:write`, and `labs:write`. A user attribute `clinic_id` travels in tokens as a claim of the same name, so the API can restrict data to the right clinic.

Machines and partners

The `lab-results-service` pushes results into the clinic API without any human involved. It authenticates as itself with the client credentials grant and a service account, and it holds only the `labs:write` role. A second machine client, `ops-automation`, lets scripts and infrastructure tools

manage Keycloak's configuration with deliberately limited administrative roles. Technicians at Northgate Labs sign in through their own SAML IdP.

Across all of this, Alderbrook wants audit trails from login and admin events, multiple Keycloak instances behind a reverse proxy at `https://auth.alderbrook.example` with PostgreSQL, and realm configuration kept as versioned code.

What Changes for Each Application

Moving to a central IdP changes every application's job. Applications **stop** storing passwords, rendering login forms, sending reset emails, and implementing MFA. They **start** redirecting to Keycloak, handling a callback, validating tokens, and mapping claims to their own authorization decisions. The new duties are defined by standards and handled mostly by mature libraries. Here is a preview of token validation in the clinic API with the `jose` library; Chapter 7 explains every line:

clinic-api/src/auth.js (preview)

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

const issuer = 'http://localhost:8080/realms/alderbrook';
const jwks = createRemoteJWKSet(
  new URL(`${issuer}/protocol/openid-connect/certs`)
);

export async function verifyAccessToken(token) {
  const { payload } = await jwtVerify(token, jwks, {
    issuer,
    audience: 'clinic-api',
  });
  return payload; // claims such as sub, realm_access, clinic_id
}
```

Notice what is missing: no password, no database lookup, no call to Keycloak on every request. The API downloads Keycloak's public signing keys once, caches them, and verifies each token's signature locally. This is one of the reasons token-based architectures scale well.

Common Misconceptions

Teams new to IAM tend to carry a few assumptions that cause trouble later. Clearing them up now will save time.

"OAuth is a login protocol"

OAuth 2.0 is a framework for *delegated authorization*: it lets a client obtain an access token to call an API. It says nothing about who the user is. OpenID Connect adds the identity layer on top, with the ID token and standardized user claims. When people say "log in with OAuth", they almost always mean OpenID Connect. Chapter 2 makes this distinction precise.