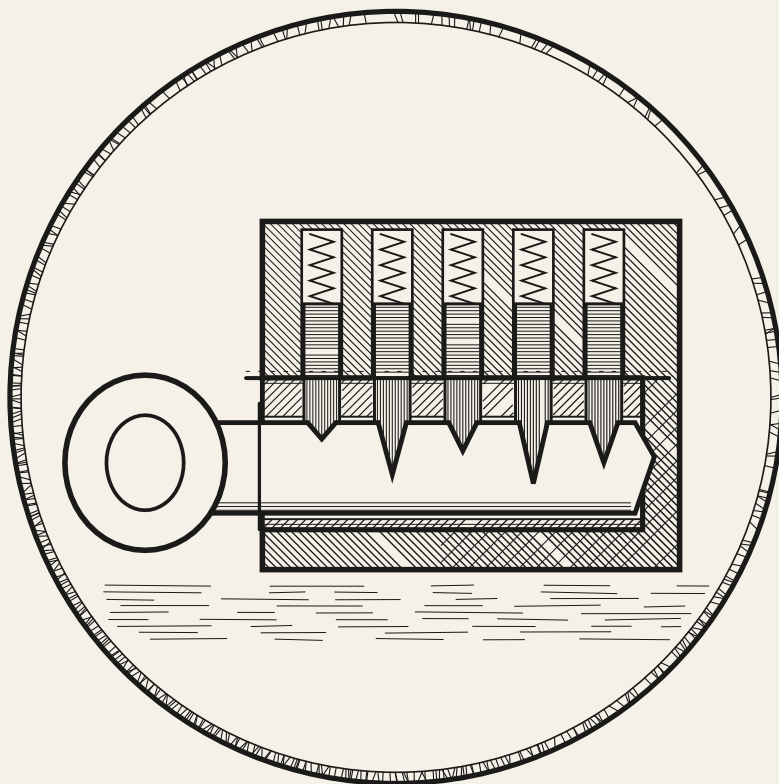




Keycloak in Practice

Hands-on labs for single sign-on you can explain

24 chapters, 24 labs, and two that fail on purpose

Active Directory, Kubernetes, Vault - and what goes over the wire

Thomas Zachmann

BOOK THREE

Keycloak in Practice

A Hands-On Lab Guide to Single Sign-On with Active Directory, Kubernetes, and Vault

Thomas Zachmann

Contents

Keycloak in Practice	1
A Hands-On Lab Guide to Single Sign-On with Active Directory, Kubernetes, and Vault . . .	1
Copyright	1
Acknowledgements	3
About the Author	3
How to Read This Book	4
Chapter 1 — Why Single Sign-On	10
What single sign-on actually is	11
Lab 1 — Three applications, three user databases	12
What the applications get instead	14
What this does not fix	15
Versions	15
Chapter 2 — Building Your Lab	23
If your machine has an Apple silicon processor, read this first	24
Lab 2 — The names, the certificates, the skeleton	25
Why containers and not a virtual machine	30
The names	30
What the issuer has to do with any of this	31
Three ways to be trusted, ranked	31
The four trust stores, and why they are four	32
What reset means	32

Keycloak in Practice

A Hands-On Lab Guide to Single Sign-On with Active Directory, Kubernetes, and Vault

Book Three of the In Practice series

Thomas Zachmann

Copyright

Keycloak in Practice: A Hands-On Lab Guide to Single Sign-On with Active Directory, Kubernetes, and Vault

Copyright © 2026 Thomas Zachmann. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise — without the prior written permission of the author, except for brief quotations in a review.

Trademarks

Keycloak and Red Hat are trademarks of Red Hat, Inc. Kubernetes is a registered trademark of The Linux Foundation. Microsoft, Active Directory, Windows and Entra are trademarks of Microsoft Corporation. Docker is a trademark of Docker, Inc. HashiCorp, Vault and Terraform are trademarks of HashiCorp, Inc. Grafana is a trademark of Grafana Labs. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. All other trademarks are the property of their respective owners.

This book is an independent publication. It is not affiliated with, authorized by, sponsored by, or otherwise approved by Red Hat, Inc., Microsoft Corporation, The Linux Foundation, or any other organisation named in it.

Where trademarks appear in this book, they are used in an editorial fashion to describe the software concerned, with no intention of infringement.

Disclaimer

The information in this book is provided on an “as is” basis, without warranty of any kind, express or implied. Neither the author nor any distributor shall be liable for any damages arising directly or indirectly from the use of the information or the code examples contained herein.

The commands and configurations in this book are written for a **local laboratory environment**. Several of them deliberately weaken security — a private certificate authority is created and trusted by your browser and your operating system, administrator passwords appear in environment files on disk, a directory server is provisioned with passwords chosen for typing rather than for strength, token lifetimes are shortened so that expiry can be observed, and in two chapters you are asked to lock yourself out of your own systems and find out whether you can get back in. **Do not run these commands against a production system.** Every chapter states which settings must change before anything resembling this setup goes near real users, and Appendix I is the checklist for exactly that.

One warning specific to this subject. An identity provider is the system that decides who everyone else is. A mistake in a registry costs you an image; a mistake here costs you every application behind it at once. Two chapters in this book therefore produce a deliberate, total loss of access — Chapter 16 removes your ability to reach a Kubernetes cluster, and Chapter 20 prevents two systems from starting because each waits for the other. Both are recoverable, both are meant to be experienced once, and neither belongs anywhere near a system other people depend on.

Software changes. Every version this book was written against is listed in one file, `VERSIONS.md`, in the companion repository, and the labs are run against those versions weekly. Keycloak releases several times a year and has a history of renaming configuration options between them; where a later release behaves differently, the errata page in that repository says how. If something does not work as printed, check the version first.

Edition

First edition, 2026. Version 1.0, October 2026.

Acknowledgements

To my wife, and to my family, for the evenings this took.

And to the colleagues who have shaped this career — the ones who showed me how things are actually built, and the ones who, just as often and just as usefully, brought me back down to earth.

About the Author

Thomas Zachmann builds Kubernetes platforms for organisations where security and auditability are the requirement rather than the aspiration: on-premises, on bare metal, in air-gapped networks, and in European sovereign clouds. Since 2020 he has worked almost exclusively in regulated environments — federal printing and identity services, public-sector IT providers, cooperative banking, and security technology vendors — under frameworks including BSI IT-Grundschutz, BSI C5, ISO 27001, BAIT and DORA.

Identity is the part of that work this book is about, and it is the part that arrives first. Before anything can be audited, somebody has to be able to say who did it — which means that on every platform he has built, Keycloak went in early and everything else was fitted around it. He has federated it against Active Directory in organisations that had run one for twenty years, brokered it to an upstream identity provider in organisations that had just stopped running one, put it in front of Kubernetes API servers, registries, dashboards and pipelines, and been called in afterwards to explain why a token that looked correct was being refused.

That last one is why this book prints the request and the response on every page. The gap he has watched teams fall into is never conceptual. They can describe an authorization code flow. What they cannot do is open the token, read the six claims that matter, and say which component is lying about which one.

Around identity sits the rest of the platform: policy enforcement with Kyverno, supply-chain evidence with Harbor, Trivy and Dependency Track, and secrets management with HashiCorp Vault, which he has been deploying and operating since 2017. Those two are the subjects of

the first and second books in this series, and this one connects to both — Chapter 17 puts the registry from Book Two behind the login built here, and Chapters 18 to 20 make Vault and Keycloak trust each other, then show what happens when neither will start without the other.

He works in the public clouds as well — AWS, Azure and Google Cloud — which is what makes the on-premises argument in these books a comparison rather than a preference. Most regulated estates are hybrid, and knowing what a hyperscaler does well is the only honest way to say where it does not fit.

He is certified as a Certified Kubernetes Security Specialist (CKS), Certified Kubernetes Administrator (CKA) and Certified Kubernetes Application Developer (CKAD), and holds cloud certifications from AWS and Google Cloud.

Underneath that is twenty years of software engineering — performance and systems programming at SAP in C and C++, database performance analysis at IBM, and Go since. That background is why the automation in his platforms is written rather than bought, and it is why this book asks you to type rather than to read.

This book exists because of a question he has watched teams fail to answer, usually at the worst moment. Not a hard question: *why was that token rejected?* The login worked. The signature is valid. Something downstream says no, and the room divides into people who will read the token and people who will restart something.

He is based in Hamburg and works across the DACH region.

Contact: thomas@zachmann.work **Web:** thomaszachmann.de · github.com/thomaszachmann

Available for platform work, hardening reviews, and team enablement.

How to Read This Book

This is a book you type into a terminal. Reading it in an armchair will teach you what Keycloak does. It will not teach you to run Keycloak, and it certainly will not teach you to debug it. Those are different skills, and only one of them is worth money.

What you need

A laptop with 16 GB of RAM, Docker or Podman, and roughly forty minutes for most chapters. No cloud account, no credit card, no spare server, and — unusually for a book that covers Active

Directory — no Windows licence: the directory in this book is Samba, running in a container, and Chapter 2 says exactly where it differs from the real thing. Kubernetes arrives in Chapter 15 and runs on the same laptop.

The companion repository

Everything in this book is also in a repository:

```
git clone https://github.com/thomaszachmann/books.git
cd books/keycloak-in-practice
make check
```

Work inside `books/keycloak-in-practice`. Every path printed in this book is relative to that directory, so nothing needs adjusting as you read.

The repository is organised by chapter. `chapters/ch02/make-certs.sh` issues the certificates Chapter 2 makes by hand in steps 4 to 6. `chapters/ch04/decode-token.sh` is thirty lines of shell that does what Chapter 4 teaches you to do by hand. Each chapter directory has a `README.md` stating the state that chapter expects, and `make help` lists every target.

Use it to check your work and to recover when something is broken. Do not use it instead of typing — that is the one way to get no value from this book at all.

Versions, and why they are pinned

`VERSIONS.md` in the repository is the only place a version number appears. The scripts read it; nothing hard-codes one. A pipeline runs the labs every week against those versions and against whatever is current, and the difference between the two becomes an entry in `ERRATA.md`.

That is unusual for a technical book and it is the honest answer to a real problem. Keycloak ships several releases a year, and it has renamed configuration options between them more than once — the move to Quarkus, the build-and-run split, the rewritten hostname options, the separate management port. A book with printed output goes stale on somebody else's schedule, and pretending otherwise would be the first thing you caught it lying about.

Why this book is shorter than it used to be

An earlier draft explained things. It explained what a realm is, why an access token is not an identity token, and what each field in a client configuration does. Then the world changed underneath it.

You have an AI assistant in the terminal next to this book. It will explain any OAuth 2.0 grant, any SAML element and any error message, instantly, at whatever depth you ask for. It is genuinely good at that. A chapter that competes with it is a chapter you will skim.

So this book does not compete with it. It does the two things an assistant cannot:

It knows what actually happened. Every request, response and decoded token printed here was captured on a real machine against a real Keycloak. An assistant will tell you that an access token has an `aud` claim; this book shows you the token your API rejected, with `"aud": "account"` in it, and the mapper that fixed it.

It makes you do it wrong on purpose. Every chapter has failures you cause deliberately, with the command that causes them printed next to the message they produce. You can ask an assistant what `invalid redirect_uri` means. You cannot ask it to give you the experience of having caused it, which is what turns a ten-minute hunt into a one-second recognition.

The practical consequence: **keep an assistant open while you work through this.** Ask it the “why” questions this book no longer answers at length. Have it write your notes and your runbook as you go. This book supplies what the assistant does not have: a sequence that works, traffic that is real, and the failures worth causing.

How the chapters work

Every chapter has the same nine parts.

Decision and Consequence. A short table naming the decision this chapter forces on you and what it costs you later. It is the first thing you read rather than the last.

The problem. What goes wrong without the thing this chapter covers, in a few lines, usually with the error message it produces.

Concepts. What is actually happening, and why it was built that way. Short. You did not buy a reference manual.

Lab. Numbered steps you type. Expected output is printed so you can tell whether it worked.

On the Wire. The request, the response and the token as they actually cross the connection, decoded. This section is in every chapter and it is the reason this book exists. Anyone can make a login page appear. The difference between that and running an identity provider is entirely a matter of having read what it sends.

Break it on purpose. Each entry names a failure, gives you the command that causes it, prints the message it produces, and then the fix. Cause them. Identity failures are unusually good at pointing at the wrong component, so knowing *which system is actually complaining* is most of the work, and you learn that by having been wrong.

Do this without looking. The lab does not end with the last instruction. Five tasks follow with no walkthrough, numbered so later chapters can point at them, and placed **before the summary rather than after it**, because a task after a summary is a task nobody does. Worked solutions are at the end of the chapter — read them after you have tried, not instead.

What you should now be able to do. Six lines. If one of them is not true yet, the lab is still open.

Design Review. Six scenarios with no terminal and no single right answer. Each states a requirement and asks for a decision and, more importantly, a reason. Model answers follow the worked solutions. These are the pages to read on a train.

There is no exam

Keycloak has no certification, and this book is not preparation for one. What replaces it is the *On the Wire* sections and the *Design Review* questions — because the thing that distinguishes people who run identity providers well is not recall. It is being able to open a token in front of an argument and end it.

The rule this book defends

Keycloak never runs in the cluster whose API server it authenticates.

Chapter 16 makes you break that rule on purpose. You will configure a cluster's API server to trust the Keycloak running inside that cluster, verify that logging in works, and then lose that Keycloak — at which point you cannot use `kubectl` to repair it, because the thing that would let you in was inside the thing that is broken. The chapter then shows the way back in, and points the cluster at the Keycloak outside it.

It is stated as a rule rather than a consideration because it is falsifiable and because the failure it prevents is not gradual.

The chapters that will annoy you

Chapter 1 asks you to disable an employee’s account in one system and then log in as them in two others. Chapter 9 asks you to log out and prove that you are still logged in. Chapter 16 locks you out of a cluster. Chapter 20 configures Vault to trust Keycloak and Keycloak to keep its secrets in Vault, then cuts the power and lets you watch Keycloak refuse to start until Vault is unsealed — and then locks you out of Vault through Keycloak.

These are the most valuable chapters in the book. Nobody who has recovered an identity provider at two in the morning learned it from documentation.

Conventions

Commands you type appear like this, with a `$` prompt:

```
$ docker --version
```

Output appears without a prompt:

```
Docker version 29.2.1, build ce12230
```

Long commands are broken across lines with a backslash. Type them as one line, or paste them including the backslashes — both work.

```
$ openssl x509 -req -sha256 -days 365 -in sso.csr \  
-CA ca.crt -CAkey ca.key -CAcreateserial \  
-extfile sso.ext -out sso.crt
```

Tokens are abbreviated. A real access token is around a thousand characters of Base64URL, and printing one in full would waste a page and teach nothing. Where the raw form appears, it is cut to its header, a couple of lines of payload and a stub of signature, marked with an ellipsis:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXZWQ0aXNldUIiwia2lkIiA6ICJ...
```

The decoded payload is printed in full, because that is the part worth reading; where a chapter leaves lines out, it says so. Appendix E holds complete, unabbreviated examples of every token type in the book, for the times when you need to see one whole.

Where a placeholder must be replaced with a value from your own system, it appears in angle brackets: `<your admin password>`, `<the client secret>`. These are never literal.

Names follow one scheme throughout: `sso.meridian.test` is Keycloak, `dc.meridian.test` is the directory server, `app.meridian.test` is the demo application, and `MERIDIAN.TEST` is the Kerberos realm. Chapter 2 explains why they end in `.test` and why they are not `localhost`.

Chapter 1 — Why Single Sign-On

Decision and consequence

Decision	What it costs you later
Let each application keep its own users	Nothing, until somebody leaves — and then a number of places to remember that nobody wrote down
Move authentication out of the applications	A system everything now depends on, which has to be run properly rather than installed
Give every application the same directory instead	One password, still as many sessions, still as many places a password is typed and could be captured

The third row is the one people mistake for the second, and it is the distinction this chapter exists to draw.

The problem

Meridian Freight runs three internal applications. A wiki, a ticket system, and an operations dashboard. Each was introduced by a different team in a different year, each has a login page, and each has its own table of users.

On a Tuesday, a contractor's engagement ends. Somebody with a keyboard removes the account from the wiki, because the wiki is the one everybody thinks of.

Two weeks later the contractor still has the ticket system and the dashboard.

Nobody was careless. There was no list of the three places, because no system was ever asked to hold one. The wiki does not know the ticket system exists.

This chapter builds that situation, on purpose, and then measures it — because the argument for single sign-on is usually made with adjectives, and it is much stronger made with numbers.

What single sign-on actually is

The phrase is used for two different things, and the difference decides what you get.

One password everywhere. Every application is pointed at the same directory. Users type the same password into three login pages, and each application checks it against the directory. This is what an LDAP integration gives you, and it is a real improvement: one password to change, one place to disable.

One session. Users authenticate *once*, at a single system, and the applications never see the password at all. They receive evidence that authentication happened, issued by something they trust.

Both are called single sign-on. Only the second one is.

The distinction matters because of where the password goes. Under the first arrangement, each application receives the user's password on every login. Which means every application is a place a password can be logged, stored badly, or captured; and it means that adding a second factor requires changing three applications, because each of them is doing the authenticating.

Under the second, the password reaches exactly one system. Adding a second factor, changing the password policy, refusing a login from a new country, requiring re-authentication for an administrative action — all of these become a change in one place, and the three applications are not modified, redeployed, or even informed.

That is the property worth the operational cost of this book. Not convenience. **Authentication becomes a thing you can change.**

Build it first. What each piece is called, and what it costs, is after the lab — where it will mean something.

Lab 1 — Three applications, three user databases

The three applications in this lab are three web servers with a password file each. That is a caricature of a wiki and a ticket system, and it is an exact model of the thing that matters: each one authenticates independently, and none of them knows about the others.

Step 1 — Start them

```
$ cd books/keycloak-in-practice
$ make ch01
$ docker compose ps --format '{{.Name}}\t{{.Status}}'
keycloak-in-practice-dashboard-1    Up 1 second
keycloak-in-practice-tickets-1      Up 1 second
keycloak-in-practice-wiki-1         Up 1 second
```

`make ch01` creates `.env` with the pinned versions if it is not there yet, then runs `docker compose --profile ch01 up -d wiki tickets dashboard`. The three names matter. A profile adds services to the ones Compose starts anyway; it does not select them. Without the names, Compose would also start the database, Keycloak and the client box — and before Chapter 2 has made the certificates they mount, Docker creates the missing files as empty, root-owned directories that Chapter 2 then trips over.

Three services: `wiki`, `tickets`, `dashboard`. Nothing else from this book is running yet, and Keycloak does not exist until Chapter 3.

Step 2 — Give the contractor an account in each

```
$ for a in wiki tickets dashboard; do
    docker compose exec $a htpasswd -bB /etc/httpd-users \
        dmorel 'Autumn-Ledger-77'
done
```

Three commands, because there are three user databases. Note what you had to know to run them: the name of every application, and that there were exactly three.

Step 3 — Confirm access

```
$ for p in 8081 8082 8083; do
  curl -sS -o /dev/null -w "$p %{http_code}\n" \
    -u dmorel:Autumn-Ledger-77 http://localhost:$p/
done
8081 200
8082 200
8083 200
```

Step 4 — Offboard, the way it actually happens

The engagement ends. Remove the account from the wiki:

```
$ docker compose exec wiki htpasswd -D /etc/httpd-users dmorel
```

One command, and no reload: the web server reads the password file on every request, so the account stops working immediately. Nothing about what follows is a caching artefact.

That is a correct, complete, well-executed offboarding — of one application.

Step 5 — Measure what is left

```
$ for p in 8081 8082 8083; do
  curl -sS -o /dev/null -w "$p %{http_code}\n" \
    -u dmorel:Autumn-Ledger-77 http://localhost:$p/
done
8081 401
8082 200
8083 200
```

Two systems still admit a person who no longer works here. Nothing failed. No error was logged anywhere, because from the ticket system's point of view nothing happened at all.

Step 6 — Try to answer the auditor's question

Which systems can dmorel reach?

There is no command for that. There is only this:

```
$ for a in wiki tickets dashboard; do
    printf '%s: ' $a
    docker compose exec -T $a grep -c '^dmorel:' \
        /etc/httpd-users || true
done
wiki: 0
tickets: 1
dashboard: 1
```

You can answer it because you already know the list of applications. The auditor is asking precisely because nobody does. Add a fourth application next quarter and this loop is wrong, silently, and stays wrong until somebody notices.

Step 7 — Count

Write these three numbers down; Chapter 17 asks for them again.

Quantity	Now
Places a password is stored	3
Steps to remove one person	3
Systems that can list a person's access	0

The last row is the one that costs money. The first two get worse linearly with the number of applications. The third is already as bad as it can be.

Step 8 — Stop them

```
$ docker compose --profile ch01 down
```

What the applications get instead

If an application never sees the password, it needs something else, and the something else is a **token**: a short-lived, signed statement from the identity provider saying who the user is and what they are allowed to ask for.

Signed, so it cannot be edited. Short-lived, so a stolen one is worth little. And verifiable without calling anybody: the application checks the signature using a public key it fetched earlier, which means the identity provider does not have to answer a question on every request.

Chapters 4 and 8 take a real one apart. For now, one consequence is worth carrying forward, because it explains most of the failures in the rest of this book:

The application decides whether to trust a token by checking who issued it, who it was issued for, and whether it has expired. All three are strings and numbers inside the token, and all three can be right in one system and wrong in another.

That is why Chapter 2 spends an entire chapter on a name.

What this does not fix

Two honest limits, stated now rather than discovered in Chapter 16.

Everything now depends on one system. Three applications that each had their own login could each fail alone. After this book they share a dependency, and when it is down, nobody logs in to anything. That is not an argument against doing it — the alternative failed differently and more quietly — but it is the reason Chapters 21 and 22 exist, and the reason this book has a rule about where the identity provider must not run.

Single sign-on is not single sign-out. Logging out of one application does not, by default, end the session at the identity provider, and ending the session at the identity provider does not, by default, tell the applications. Chapter 9 makes you observe this and then fix it, and it surprises people who have been running Keycloak for a year.

Versions

This book is written against Keycloak 26.5.2. Every version the labs use is listed in one file, `VERSIONS.md`, in the companion repository, and the labs are run against those versions weekly. Where a later release differs, the errata page in that repository says how.

Keycloak renames configuration options between releases more often than most products of its age — the move to Quarkus, the build-and-run split, the rewritten hostname options, the

separate management port. If something does not work as printed, check the version before you check yourself.

On the Wire

Before leaving the old world, look at what these three applications actually receive. The `-u` flag has been sending something on every request:

```
$ curl -sS -o /dev/null -v \  
  -u dmorel:Autumn-Ledger-77 http://localhost:8082/ 2>&1 \  
  | grep -i '^> authorization'  
> Authorization: Basic ZG1vcmVsOkF1dHVtbi1MZWRnZXItNzc=
```

That is not encryption. It is Base64, which is an encoding, and it reverses in one command:

```
$ echo ZG1vcmVsOkF1dHVtbi1MZWRnZXItNzc= | base64 -d  
dmorel:Autumn-Ledger-77
```

Three things follow, and they are the whole case for the rest of this book.

The password is in the request. Not only at login — on *every* request, because HTTP basic authentication has no session. Each of the three applications receives the user's actual password, thousands of times a day, and each of them is a place it could be logged.

Every application must be trusted with it. The ticket system that a team installed in 2021 has the same password as everything else that person uses, if they reuse passwords, which they do.

Nothing here can expire. There is no lifetime, no revocation, and no way to end a session that does not involve changing the password everywhere.

Now the preview. By the end of Chapter 4, the equivalent header will look like this:

```
> Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSld...
```

Also Base64, also readable, and deliberately so — there is nothing secret in the payload. What is different is that it was issued by a system these applications trust, it expires in five minutes, it says what it is for, and the applications never saw a password to produce it. Chapter 4 obtains one by hand, without a browser doing anything you cannot see.

Break it on purpose

Cause each of these, read the real message, then fix it. These are the failures you will meet again, and having produced one deliberately is what makes it recognisable in a second rather than in half an hour.

htpasswd: not found

Cause it: run the Step 2 command against a container built from an nginx image instead of the Apache one:

```
$ docker compose exec tickets htpasswd -Bbc /etc/httpd-users a b
```

htpasswd ships inside the Apache image the profile uses. nginx has no equivalent, and the usual answer — `apache2-utils` on the host — writes to a file the container cannot see.

Fix: run it in the Apache container, or write the file on the host into a path that is mounted.

The 401 arrives before you expect it

Cause it: do Step 5 in a browser tab that is already authenticated rather than with `curl`.

Some web servers cache credentials per connection. `curl` opens a new connection each time, so the lab is unaffected; the tab is not, and it keeps working until you close it.

Fix: none needed — it is a preview of Chapter 9. Removing an account does not end a session that already exists.

One application returns 401 after Step 2

Cause it: add a second user to the ticket system with `-c`.

```
$ docker compose exec tickets htpasswd -bc /etc/httpd-users mara pw
$ docker compose exec tickets cat /etc/httpd-users
mara:$apr1$IoTrYk.H$EYI8zBBkCTpmpU2cehxZE/
```

`-c` means *create*, and creating a file that exists replaces it. `mara` can log in to the ticket system; `dmorel` gets 401 there and 200 from the other two, because only that one file was rewritten. The missing `-B` is not the cause: without it `htpasswd` writes an MD5 (`$apr1$`) hash, which Apache accepts as readily as `bcrypt`. The other way to produce the same symptom is an empty file, because `htpasswd` wrote it somewhere else — the `cat` above tells the two apart.

Fix: `-c` only for the first user of a new file, and check the file is where the server reads it.

Port already in use on 8081 to 8083

Cause it: start a second compose project from another chapter without stopping this one.

```
$ lsof -nP -iTCP:8081 -sTCP:LISTEN
```

Fix: stop the other project. Chapter 2's diagnosis applies.

Do this without looking

1.1 Add a fourth application to the compose profile with its own password file. Then repeat Step 6 without editing the loop, and describe what went wrong and why nothing warned you.

1.2 In Step 4 you removed the account. Instead, change the contractor's password in the wiki only. Which of the three applications now rejects them, and what does that tell you about where the decision is being made?

1.3 Run Step 3 against `https://` instead of `http://`. It will fail; say precisely why, and then say whether TLS would have fixed anything the On the Wire section objected to.

1.4 Write down the offboarding runbook Meridian Freight would need today, as numbered steps. Then say what has to happen to that document every time a new application is introduced, and who you would expect to do it.

1.5 Decode the `Authorization` header from a request to a service you actually use at work, in your browser's developer tools. If it says `Bearer`, you have already met Chapter 4. If it says `Basic`, this chapter is about your employer.

What you should now be able to do

You can now:

- State the difference between one password everywhere and one session, and say which of the two is single sign-on
- Say where the password goes under each arrangement, and why that decides whether a second factor can be added
- Name the three things an application checks in a token

- Give two honest limits of single sign-on
- Demonstrate that an offboarding can be correct and incomplete at the same time
- State the three numbers that measure the problem, and which one does not improve as the estate grows

What a Keycloak expert knows without looking it up

Not exercises — the things that should be automatic. Cover the right column and work down.

Question	Answer
What SSO removes	one password store per application
What it does not remove	the need to end sessions
Protocol for new applications	OIDC
Protocol you will still meet	SAML
What the identity provider issues	tokens, not passwords
Where authorisation usually lives	the application, not the IdP
What breaks when a contractor leaves	whatever has its own account
Cost of one more application	one client, not one more directory
First thing to agree	who owns the user namespace
What the browser actually carries	redirects and a cookie

If any right-hand cell was not immediate, that row is your next five minutes.

Design Review

1. A colleague proposes solving the offboarding problem with a script that removes a user from all three applications. Give the strongest version of that argument, then two failure modes that a directory does not have.
2. After this book, all three applications depend on one identity provider. Name three things that must be true before that is an improvement in availability rather than a reduction.

3. The dashboard is a twelve-year-old application that cannot be modified and understands only HTTP basic authentication. Sketch how it joins the estate anyway, and name the chapter of this book that builds it.
4. A team argues that single sign-on increases risk, because one compromised password now opens three applications instead of one. Give the strongest version of that argument, then the response — and make the response about what becomes possible rather than about what becomes convenient.

Worked solutions

1.1 The loop reports on three applications and there are four, so it returns a clean, confident, wrong answer. Nothing warned you because the list of applications lives in your head and in the loop, not in a system. This is the third row of Step 7 and it is the one that does not improve on its own.

1.2 Only the wiki rejects them. Each application checks the password against its own file, so a change in one is invisible to the others. The decision is being made three times, independently, which is another way of saying it is not being made anywhere.

1.3 Nothing is listening on TLS, so the connection is refused. TLS would encrypt the header in transit and would fix eavesdropping on the network — which was not the objection. The objection is that the password arrives, in full, at the application, on every request. TLS delivers it more safely to a place it should not be going.

1.4 The runbook has one step per application plus a step to check the shared mailbox, the VPN and whatever else is remembered on the day. Every new application must add a line, which means the document is correct only if the person introducing an application knows the document exists and edits it. In practice that document is maintained by whoever was last surprised by it.

1.5 No solution; the point is that `Basic` is still extremely common inside corporate networks, and that finding one is more persuasive than this chapter.

Design Review — model answers

1. The strongest version: it is an afternoon's work, it needs no new infrastructure, it can be run from the existing joiner-mover-leaver process, and it solves the exact problem observed. For three applications that is proportionate.

Two failure modes it does not have an answer for. First, it only removes — it cannot *answer*. The auditor's question in Step 6 stays unanswerable, because the script knows how to delete an account and not how to enumerate one. Second, it is a list that must be edited by whoever adds the fourth application, and the failure mode of such a list is silence: it does not error when it is incomplete, it succeeds. A directory does not have that failure mode because there is no list — the account exists in one place, and an application either checks that place or does not.

2. It must not share a failure domain with what it protects, which is this book's rule and Chapter 16's lab. It must survive the loss of one instance, which is clustering and a database that is not on the same host, in Chapter 21. And there must be a way in when it is down — a break-glass path that does not depend on it, tested rather than documented, which appears in Chapter 16 for the cluster and in Appendix I as a requirement. A fourth, less obvious: the applications must tolerate the provider being briefly unavailable without logging everyone out, which is a consequence of tokens being verified by signature rather than by asking.

3. It sits behind a proxy that performs the authentication on its behalf: the proxy runs the flow with Keycloak, and passes the result to the application in whatever form it understands — a header, or basic authentication with a credential the application already has. The application is not modified and does not know. That is Chapter 17, and the trade-off named there is that anything able to reach the application directly, bypassing the proxy, bypasses authentication entirely.

4. The strongest version is correct as stated: the blast radius of one credential does increase, and a phished password that used to open one application now opens the estate. Anybody who answers this by denying it is not worth listening to.

The response is that the comparison is not between one password and three passwords. It is between an estate where authentication can be strengthened and one where it cannot. With one place to authenticate, you can require a second factor tomorrow, require re-authentication for administrative actions, revoke every session in the estate in one command, and see the login that came from an unexpected place. None of those are available when three applications each

authenticate for themselves — not because they are hard, but because they must be built three times by three teams who will not prioritise it. Consolidation raises the value of one credential and, done properly, lowers the probability that a stolen one is sufficient. Doing only the first half is the failure mode, and it is why Chapter 10 is not optional.

Chapter 2 — Building Your Lab

Decision and consequence

Decision	What it costs you later
Give the identity provider one name, used by everyone	A name that has to resolve the same way from your laptop and from inside every container, which is more work than <code>localhost</code>
Give that name a certificate from a CA you run	A root key you now own, a renewal nobody will remember, and four different trust stores to feed
Use <code>localhost</code> and plain HTTP instead	Two issuers for one server, tokens that validate in the browser and fail in the API, and a rewrite the first time anything else joins the lab

The third row is the one this chapter exists to prevent, and it is the most common way a Keycloak lab wastes a weekend.

The problem

Keycloak starts in one command. That command is in Chapter 3, and it is not the hard part.

The hard part is that an identity provider is not a server you talk to. It is a server that **other** things talk to, about you, when you are not looking. A browser follows a redirect to it. An API asks it for its public keys. A Kubernetes API server, in Chapter 16, fetches a document from it

while nobody is watching. Each of them has to reach it by the same name, and each of them has to believe its certificate.

Get that wrong and nothing fails cleanly. The login page appears, you sign in, you get a token — and the API rejects it for a reason that looks like a bug in the API.

So this chapter builds names, certificates and a skeleton you can throw away, and starts nothing. Keycloak itself is Chapter 3.

If your machine has an Apple silicon processor, read this first

Keycloak's own images are published for both `amd64` and `arm64`, so Keycloak runs natively. Nothing in Parts I to III needs emulation, and neither do the Kubernetes chapters.

The part that would have caught you out is the **Active Directory** from Chapter 12. Samba domain controller images are community-built, and most of them publish `linux/amd64` only. On Apple silicon that means either emulation or a directory server that starts and then dies with a message saying nothing about architecture:

```
exec /usr/sbin/samba: exec format error
```

So this lab does not pull one. It **builds** one, from Ubuntu, with Samba from the distribution's own packages. Ubuntu publishes both architectures, Samba is a package rather than a binary somebody cross-compiled, and the result runs natively wherever you are reading this. It costs about two minutes on first use:

```
$ make ad-up
```

That is the opposite of what this series does elsewhere, deliberately. Harbor and Vault were the thing under study; here the directory is scenery. Chapters 12 to 14 teach LDAP and Kerberos, not Samba's release engineering.

If you substitute a community image later — which means replacing the `build:` section of the `dc` service in `compose.yaml` with an `image:` line — check it before you rely on it:

```
$ docker manifest inspect <the image> \  
  | jq -r '.manifests[].platform | .os + "/" + .architecture'  
linux/amd64  
linux/arm64
```

No `linux/arm64` in that list means the `exec format error` above, and Appendix A has the two ways round it.

Build it first. What each piece is called, and what it costs, is after the lab — where it will mean something.

Lab 2 — The names, the certificates, the skeleton

Nothing in this lab starts Keycloak. At the end of it you will have proved that a name resolves, that a certificate for that name verifies against a root you created, and that your directory server image runs on your machine — and you will have proved the last one with a Kerberos ticket rather than an assumption.

Step 1 — Check what you have

```
$ docker --version
$ docker compose version
$ openssl version
$ jq --version
$ git --version
```

Podman works too, with `podman compose`; Appendix A lists the three differences that matter, one of which is binding port 443.

Step 2 — Get the companion repository

If Chapter 1 did not already have you clone it:

```
$ git clone https://github.com/thomaszachmann/books.git
$ cd books/keycloak-in-practice
$ cat VERSIONS.md
```

`VERSIONS.md` holds every pinned version the labs use, and the compose files read them from `.env`. Chapter 1 named the Keycloak release this book was written against; if `VERSIONS.md` disagrees with your memory of it, `VERSIONS.md` wins, and `ERRATA.md` says what changed.

```
$ make env
$ make check
```

`make env` copies `.env.example` to `.env` if there is none — `make ch01` in Chapter 1 may already have done so — and appends the pinned versions. A plain `cp .env.example .env` is not enough: Compose does not read `VERSIONS.md`, so the image tags would be empty and the failure would arrive in Chapter 3 as an image with no tag. `make check` reports anything missing; `pki/ca.crt` is expected to be missing until Step 4.

Then open `.env` and change `KC_BOOTSTRAP_ADMIN_PASSWORD`. The example value says to, and Chapter 3 asks you for it.

Step 3 — Make the names resolve

On macOS and Linux:

```
$ sudo tee -a /etc/hosts <<'EOF'
127.0.0.1  sso.meridian.test
127.0.0.1  dc.meridian.test
127.0.0.1  app.meridian.test
EOF
```

On Windows the file is `C:\Windows\System32\drivers\etc\hosts` and the editor needs to be running as administrator.

Prove it before going on:

```
$ getent hosts sso.meridian.test || \
  dscacheutil -q host -a name sso.meridian.test
```

`getent` is the Linux answer; `dscacheutil` is the macOS one. If neither prints an address, nothing later in this chapter can work.

Step 4 — Create a certificate authority

Skip the `openssl` commands if you still have `ca.crt` and `ca.key` from book two; copy them into `pki/` instead. Steps 4 to 7 run inside `pki/`.

```
$ mkdir -p pki && cd pki
$ openssl genrsa -out ca.key 4096
$ openssl req -x509 -new -nodes -sha256 -days 1825 \
  -key ca.key -out ca.crt \
  -subj "/O=Meridian Freight/CN=Meridian Lab Root CA"
```

Five years on the root, one year on the leaf. That asymmetry is deliberate and Exercise 2.4 makes you feel it.

Step 5 — Sign a certificate for the identity provider

A certificate needs a Subject Alternative Name. Modern clients ignore the Common Name entirely, and a certificate without a SAN fails in a way that reads like a name mismatch rather than a missing field.

```
$ cat > sso.ext <<'EOF'
subjectAltName=DNS:sso.meridian.test
extendedKeyUsage=serverAuth
EOF
$ openssl genrsa -out sso.key 4096
$ openssl req -new -sha256 -key sso.key -out sso.csr \
    -subj "/O=Meridian Freight/CN=sso.meridian.test"
$ openssl x509 -req -sha256 -days 365 -in sso.csr \
    -CA ca.crt -CAkey ca.key -CAcreateserial \
    -extfile sso.ext -out sso.crt
```

Read back what you made, rather than trusting that it worked:

```
$ openssl x509 -in sso.crt -noout -text \
    | grep -A1 'Subject Alternative Name'
      X509v3 Subject Alternative Name:
        DNS:sso.meridian.test
```

Step 6 — Install the root into the trust stores

Two stores now, two later.

Operating system. On macOS:

```
$ sudo security add-trusted-cert -d -r trustRoot \
    -k /Library/Keychains/System.keychain ca.crt
```

On Debian and Ubuntu:

```
$ sudo cp ca.crt \
    /usr/local/share/ca-certificates/meridian-root.crt
$ sudo update-ca-certificates
```

It may print a line beginning `rehash: warning: skipping ca-certificates.crt`. That is the bundle file it has just written, and the warning is harmless.

On Fedora and RHEL:

```
$ sudo cp ca.crt /etc/pki/ca-trust/source/anchors/  
$ sudo update-ca-trust
```

Firefox. Preferences, Privacy and Security, Certificates, View Certificates, Authorities, Import, select `ca.crt`, tick *Trust this CA to identify websites*. There is no command for this, which is precisely why it gets forgotten.

Chrome, Edge and Safari use the operating system store, so they are already done.

Step 7 — Prove it, before Keycloak exists

Serve the certificate with `openssl` and nothing else. If this works, every failure from Chapter 3 onwards is Keycloak's, not the certificate's — and that division is worth an hour later.

In one terminal:

```
$ openssl s_server -accept 8443 -www \  
-cert sso.crt -key sso.key
```

In another:

```
$ curl -sS https://sso.meridian.test:8443/ -o /dev/null \  
-w '%{http_code}\n'  
200
```

No `-k`. If `curl` prints a certificate error, the trust store step did not take effect, and Chapter 3 will not fix it.

Now the same thing in a browser: open `https://sso.meridian.test:8443/`. A padlock, no warning. If Firefox complains and Chrome does not, re-read Step 6.

Stop the server with Ctrl-C, and go back to the repository root:

```
$ cd ..
```

Step 8 — Bring the directory up once, then put it away

The directory serves LDAPS, and Chapter 12 will insist that its certificate is signed by your root. Issue it now, before the first start, so that the domain controller never serves anything else. Exercise 2.2 does it by hand; `make pki` does it for you, and reuses what Steps 4 and 5 made:

```
$ make pki
root CA: reusing pki/ca.crt
sso: reusing pki/sso.crt
dc: issuing
...
$ make ad-up
$ docker compose exec client kinit administrator
Password for administrator@MERIDIAN.TEST:
Warning: Your password will expire in 41 days on
  Mon Nov  9 16:27:27 2026
$ docker compose exec client klist
Ticket cache: FILE:/tmp/krb5cc_0
Default principal: administrator@MERIDIAN.TEST
...
```

The password is `AD_ADMIN_PASSWORD` from `.env`. `make ad-up` starts two containers, the domain controller and the client box, and nothing else — Keycloak still does not exist. The first start provisions the domain and takes a minute or two; if `kinit` cannot reach the KDC, wait and try again. The expiry warning is Samba’s default password policy and does not matter here. A ticket means the domain controller runs on your machine, its KDC answers, and DNS inside the compose network works. That is everything Chapter 12 needs, established ten chapters early.

```
$ make ad-down
```

The directory stays down until Chapter 12. It is the heaviest thing in the lab and nothing before Chapter 12 uses it.

Step 9 — Tidy up

```
$ make down
$ git status
```

`pki/` is in `.gitignore`. A private key in a public repository is a bad habit that starts in a lab.

Why containers and not a virtual machine

Harbor needed a virtual machine because its installer is a tarball that expects a Linux host. Keycloak is a container image configured through environment variables, and its own documentation treats it that way, so there is no machine to administer and inventing one would teach a shape nobody uses.

A compose file in a repository gives what the virtual machine gave: Linux inside, a lab described by files rather than by what somebody typed, and the same result on every host — with one exception, in the next section, and it only concerns the directory server.

The names

A certificate certifies a name. An OIDC token, as the next section shows, also certifies a name. So the names come first.

Name	What it is
sso.meridian.test	Keycloak
dc.meridian.test	The Samba domain controller, from Chapter 12
app.meridian.test	The demo application, from Chapter 4

The Kerberos realm is `MERIDIAN.TEST` and the NetBIOS domain is `MERIDIAN`. Chapter 13 explains why a Kerberos realm is conventionally the DNS domain in capitals, and what breaks when it is not.

The `.test` top-level domain is reserved by RFC 6761 for exactly this: it will never be delegated, it cannot collide with a real domain, and no resolver will go looking for it on the internet. Using `.local` instead would collide with multicast DNS on macOS, and using a name you do not own is how a lab ends up sending credentials to somebody else's server.

In the lab, these names resolve two ways at once, and that is the point of the whole arrangement: **From your laptop**, through your hosts file, to `127.0.0.1`. The browser and every command you type reach the published ports.

From inside the containers, through the compose network, to the container. No hosts file is involved, because a container does not read yours.

Same name, two paths, one certificate. Chapter 15 adds a third path, from inside a Kubernetes cluster, and makes the same name work there too.

What the issuer has to do with any of this

Every OpenID Connect provider publishes a document at a fixed path:

```
/realms/<realm>/.well-known/openid-configuration
```

Inside it is a field called `issuer`, built from the URL Keycloak believes it is reached at. Every token carries that string in its `iss` claim, and every correct validator compares the two. If they differ the token is rejected — the validator doing the one job it has.

With `localhost` there are two. The browser reaches `http://localhost:8080` and gets a token whose `iss` says so. The demo application, in a container, cannot reach `localhost:8080` — inside a container that is the container — so it is pointed at `http://keycloak:8080`, discovers that issuer, compares, and rejects your perfectly valid token.

The login worked. The token is signed. The API says no.

The fix is one name, reachable from both sides. Chapter 3 tells Keycloak what it is; Chapter 4 puts an API in front of it that compares `iss` before anything else. Two consequences arrive later:

The issuer includes the scheme. `https://sso.meridian.test` and `http://sso.meridian.test` are different issuers. Adding TLS later is a migration, not a setting.

The issuer includes the port, unless it is the default. Running on `8443` and moving to `443` changes every token the lab has ever issued. That is why this chapter puts the certificate on `443` now.

Three ways to be trusted, ranked

Browsers, Go clients, JVMs and Kubernetes all refuse to talk to a server whose certificate they cannot verify. There are exactly three ways out, and the order matters.

A certificate from a public CA. Correct, and unavailable here — no public CA will issue for `.test`, and none should. Chapter 3 says how the steps change when you have one.

A private CA that you run. What this chapter does: a root, a server certificate signed by it, and the root installed wherever something talks to Keycloak. Real chain, verified properly. The cost is a root key that is now yours to look after.

Turning verification off. `curl -k`, `NODE_TLS_REJECT_UNAUTHORIZED=0`, a Java property. It works and teaches nothing, because from Chapter 12 onwards the things doing the verifying are not yours to add flags to. A Kubernetes API server has no `-k`.

If you worked through book two, reuse its root for `meridian.test`. The registry there and the identity provider here are meant to share a trust anchor; Chapter 17 hangs one on the other.

The four trust stores, and why they are four

This is where Keycloak differs from a registry, and it catches people out, so here is the map before the lab rather than after it.

Store	Who reads it	Chapter
Operating system	<code>curl</code> , <code>kubectl</code> , most command-line tools	2
Firefox's own store	Firefox, on every platform	2
Keycloak's truststore	Keycloak, when it dials out to LDAPS	12
The cluster nodes	The Kubernetes API server	16

The second row is the one that produces the most confused hours. Firefox does **not** use the operating system's trust store by default on macOS or Linux. Installing your root into the keychain and then testing in Firefox proves nothing about either.

The third row is the direction people forget entirely. Everything up to Chapter 11 has clients connecting to Keycloak. From Chapter 12, Keycloak is a client itself — it opens an LDAPS connection to the domain controller — and for that it needs a truststore of its own.

What reset means

Every chapter from here names the state it expects, in the `README.md` of its directory in the repository. There are three levels of getting back to a known state, and knowing which to reach for saves a lot of rebuilding.

Realm reset. `make realm-reset` deletes the realm and imports `realms/meridian.json` again. Seconds. That file is the realm as Chapter 3 creates it — empty. Users, clients and scopes that later chapters added are gone afterwards, and you rebuild them by running those labs again. Use it when an exercise has damaged the realm beyond repair, not as an undo.

Container reset. `make down` && `make up` removes the containers and recreates them. Volumes survive, so the database and its users survive. Use it when a container is in a state you cannot explain.

Full reset. `make clean` removes the volumes as well and returns you to the end of this chapter. Certificates are **not** removed — they live in `pki/` on disk, not in a volume — because reissuing them would mean reinstalling the root into four trust stores. `make clean-pki` exists for when you actually want that, and Exercise 2.4 uses it.

On the Wire

The chapter has no tokens yet, but it does have a handshake, and the handshake already carries a name. Start the `openssl s_server` from Step 7 again, in `pki/`, and in the other terminal:

```
$ openssl s_client -connect sso.meridian.test:8443 \
  -servername sso.meridian.test </dev/null 2>&1 \
  | head -14
depth=1 0 = Meridian Freight, CN = Meridian Lab Root CA
verify return:1
depth=0 0 = Meridian Freight, CN = sso.meridian.test
verify return:1
CONNECTED(00000003)
---
Certificate chain
 0 s:0 = Meridian Freight, CN = sso.meridian.test
 1 i:0 = Meridian Freight, CN = Meridian Lab Root CA
...
```

Three things in that output are worth reading properly.

The chain. The `depth=` lines are the path the *client* built: `depth=0` is the server certificate, `depth=1` your root, found in your store. Under `Certificate chain` is what the *server* sent. `openssl s_server` sends the root along as entry `1`, because it finds it in the same store; Keycloak in Chapter 3 sends only its own certificate. It makes no difference to trust, and that is the point: a root that arrives over the connection proves nothing. Trust comes from your store.

The verification result. `Verify return code: 0 (ok)`, further down the same output. Any other number is the entire finding; the rest of the output is decoration. `19` is a self-signed root that is not in your store, and `10` is expired.

The name. `-servername` sets the Server Name Indication field in the ClientHello, and the ClientHello is sent **before** encryption begins. The hostname you are visiting travels in clear text, every time, on every TLS connection. That is not a lab artefact and there is nothing to fix — it is how a server with several certificates knows which to present.

It matters twice more in this book. In Chapter 15 an ingress controller routes on SNI alone, before it can see any of the request. And in Chapter 16 the Kubernetes API server opens exactly this connection to fetch the discovery document, from inside the cluster, with a trust store you have to have thought about in advance.

Break it on purpose

Cause each of these, read the real message, then fix it. These are the failures you will meet again, and having produced one deliberately is what makes it recognisable in a second rather than in half an hour.

x509: certificate signed by unknown authority

Cause it: call the endpoint without telling the client about your root:

```
$ curl https://sso.meridian.test:8443/realms/master
```

Note that it says *unknown authority*, not *bad certificate*: the chain is fine, the anchor is missing.

Fix: `--cacert pki/ca.crt`, or install the root into the store that client actually reads. A Go program on Linux reads `/etc/ssl/certs`; Firefox reads neither that nor the macOS keychain.

certificate is valid for X, not sso.meridian.test

Cause it: reissue the certificate with `-extfile` left off Step 5, then look at what you made:

```
$ openssl x509 -in sso.crt -noout -ext subjectAltName
```

No output means no SAN, and modern clients do not consult the Common Name — so the certificate is valid for nothing at all.

Fix: reissue with the extension file. You do not need to touch the root.

Firefox says `SEC_ERROR_UNKNOWN_ISSUER` , Chrome is happy

Cause it: import the root into the system store only, then open the same URL in both browsers. Firefox has its own trust store on every platform. This is the second half of Step 6 and the single most common way to lose thirty minutes in this chapter.

Fix: import it into Firefox as well.

`dial tcp: lookup sso.meridian.test: no such host from a container`

Cause it: run the client outside compose:

```
$ docker run --rm curlimages/curl \  
  curl -s https://sso.meridian.test:8443/
```

Your hosts file is your laptop's, and a container does not read it.

Fix: run it inside the compose network, where the name resolves through the alias.

`bind: address already in use`

Cause it:

```
$ lsof -nP -iTCP:8443 -sTCP:LISTEN
```

On macOS this is often another compose project from an earlier book.

Fix: stop the other listener rather than moving this one — the hostname and the certificate both name this port.

`kinit: Cannot contact any KDC for realm 'MERIDIAN.TEST'`

Cause it: run `kinit` before the domain controller has finished provisioning, which on a first start takes appreciably longer than on later ones because it builds the directory.

```
$ docker compose logs dc | tail -20
```

Fix: look for the provisioning line before assuming a network fault.

`exec /usr/sbin/samba: exec format error`

Cause it: run an image with no build for your platform.

Architecture, not configuration — and it should not happen with the image this lab builds.

Fix: re-read the Apple silicon section, and check what you substituted.

Do this without looking

2.1 Delete the lab with `make clean` and rebuild it. Time it. Then say which files made that possible, and which one you would have lost if `pki/` had been a volume.

2.2 Sign a second certificate for `dc.meridian.test` from the same root, without recreating the root. Chapter 12 needs it for LDAPS.

2.3 Deliberately sign a certificate with no `-extfile`. Serve it with `openssl s_server`, reach it with `curl`, and read the error. Then fix it. Causing an error on purpose is how you prove you understand it.

2.4 Reissue the server certificate with a lifetime of one day. Confirm it works. Then set your machine's clock forward two days and try again. Describe the error, and then describe what an unattended renewal would have to do that you just did by hand.

2.5 Without editing your hosts file, make `curl` reach the test server by name using only a `curl` flag. Then explain why that flag would not help a Kubernetes API server in Chapter 16.

What you should now be able to do

You can now:

- Say why an identity provider needs one name rather than `localhost`, in terms of the `iss` claim
- Predict what happens when two consumers discover two different issuer values
- Say why adding TLS later is a migration rather than a setting
- Create a root CA and sign a server certificate with a SAN, and recognise on sight what a certificate without one does
- Name the four trust stores, say who reads each, and which one Keycloak reads as a client
- Verify a chain with `openssl s_client` independently of any application

What a Keycloak expert knows without looking it up

Not exercises — the things that should be automatic. Cover the right column and work down.

Question	Answer
Top-level domain reserved for labs	<code>.test</code>
Why not <code>.local</code>	it collides with mDNS
What a certificate certifies	a name
Field modern clients read	the SAN
Field they ignore	the Common Name
Number of trust stores in this lab	four
Store Firefox uses	its own, on every platform
What the issuer includes	the scheme
Effect of adding TLS later	every issuer string changes
Why the hosts file is a lab arrangement	it is per machine

If any right-hand cell was not immediate, that row is your next five minutes.

Design Review

1. A colleague proposes running the lab on `localhost:8080` with no TLS, on the grounds that it is a laptop and the certificate work is ceremony. Give the strongest version of their argument, then the reason you still would not do it — and make the reason about tokens, not about security.
2. The lab resolves three names through a hosts file on one laptop. Name two ways this stops working as soon as a second person is involved, and say what replaces it.
3. A requirement arrives that the identity provider's certificate must be rotated every ninety days without downtime. Sketch what has to exist for that to be routine, and name the chapter of this book where each piece appears.
4. Someone proposes putting Keycloak, the domain controller and the demo application behind a single wildcard certificate for `*.meridian.test`. Give one good reason to do it and one good reason not to, and say which you would choose for a production identity provider.

Worked solutions

2.1 Under a minute, most of it pulling images you already have. It was possible because `compose.yaml`, `.env` and the `Makefile` describe the lab; the certificates would have been lost, and with them the root you installed into four trust stores, which is the expensive half.

2.2 Only the CSR and the extension file change; the root is reused:

```
$ cat > dc.ext <<'EOF'
subjectAltName=DNS:dc.meridian.test
extendedKeyUsage=serverAuth
EOF
$ openssl genrsa -out dc.key 4096
$ openssl req -new -sha256 -key dc.key -out dc.csr \
    -subj "/O=Meridian Freight/CN=dc.meridian.test"
$ openssl x509 -req -sha256 -days 365 -in dc.csr \
    -CA ca.crt -CAkey ca.key -CAcreateserial \
    -extfile dc.ext -out dc.crt
```

2.3 The certificate has no SAN, and the error names the host rather than the missing field:

```
curl: (60) SSL: no alternative certificate subject name
matches target host name 'sso.meridian.test'
```

The Common Name says `sso.meridian.test` and the client does not care. That behaviour has been the rule since roughly 2017 and the fallback is gone, not deprecated.

2.4 With the clock moved forward the certificate is expired, and the verification code is `10`:

```
$ openssl s_client -connect sso.meridian.test:8443 \
    -servername sso.meridian.test </dev/null 2>/dev/null \
    | grep 'Verify return'
Verify return code: 10 (certificate has expired)
```

An unattended renewal has to issue a new certificate before the old one expires, place it where the server reads it, and cause the server to pick it up without dropping connections. The third part is the one that is usually forgotten, and Chapter 19 does all three from Vault's PKI engine.

2.5 `curl --resolve sso.meridian.test:8443:127.0.0.1 ...`. It sets the resolution for that one invocation. It does not help a Kubernetes API server because you do not invoke that; it resolves names through the cluster's DNS or its node's hosts file, on a schedule of its own.

Chapter 15 solves it properly rather than per command — a DNS entry for the pods, a hosts entry on the node for the API server — and Chapter 16 depends on it.

Design Review — model answers

1. The strongest version: the lab is disposable, `localhost` needs no DNS and no certificate, and every minute spent on a private CA is a minute not spent on Keycloak. It is a real argument and for a single-container demo it wins.

It stops winning the moment a second party is involved. The `iss` claim is a string that must match on both sides, so `localhost` produces one issuer for the browser and a different one for anything in a container. The failure is not a warning; it is a rejected token with an error message that points at the wrong component. And because the issuer includes the scheme, adding TLS later reissues every token in the lab. The certificate work is not about the lab being secure. It is about the lab being able to have a second participant.

2. First, the second person's laptop does not have your hosts file, so nothing resolves for them and the fix is manual on every machine. Second, the names point at `127.0.0.1`, which means *their* machine, so even a correct copy of the file sends them to their own empty ports. What replaces it is DNS that both machines query — in a cluster, CoreDNS; in an office, whatever already answers for the internal zone. Chapter 15 makes this switch inside the cluster, and it is the reason the hosts file is described here as a lab arrangement rather than a technique.

3. Something has to issue certificates on demand rather than by hand, which is Vault's PKI engine in Chapter 19. Something has to place the new certificate where Keycloak reads it, which is the Vault Agent or the Secrets Operator in the same chapter. Something has to make Keycloak serve the new certificate without a restart that drops sessions, which in practice means terminating TLS at an ingress rather than at Keycloak, and that is Chapter 15. And something has to notice when it did not happen, which is Chapter 24. Four chapters, and the ninety-day requirement is the reason all four exist.

4. For: one certificate, one renewal, one place to get it wrong, and a lab where adding a fourth name costs nothing. Against: a wildcard private key that is present on three machines is a key whose compromise is three times as likely and whose blast radius is the whole domain, and a wildcard certificate cannot be pinned to a single service.

For a production identity provider, a dedicated certificate. It is the one service whose private key compromise means every token in the estate can be forged, and sharing that key with a demo application to save a renewal is a poor trade. In the lab the wildcard is defensible, which is exactly why the book does not use one — the habit is worth more than the five minutes.