IN PRACTICE

Harbor in Practice

Hands-on labs for a private container registry

24 chapters, 23 labs, and four that fail on purpose

On a virtual machine, in Kubernetes, and when to use which

Thomas Zachmann

BOOK TWO

Harbor in Practice

A Hands-On Lab Guide to Running a Private Container Registry on Virtual Machines and Kubernetes

Thomas Zachmann

Contents

Harbor in Practice

A Hands-On Lab Guide to Running a Private Container Registry on Virtual Machines and Kubernetes

Book Two of the In Practice series

Thomas Zachmann

Copyright

Trademarks

This book is an independent publication. It is not affiliated with, authorised by, sponsored by, or otherwise approved by The Linux Foundation, the Cloud Native Computing 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 registry runs with no authentication at all in Chapter 1, passwords appear in configuration files on disk, a private certificate authority is created and trusted, and in one chapter you are asked to destroy a registry and discover whether your backup was real. **Do not run these commands against a production system.** Each chapter states which settings must change before anything resembling this setup goes near real data, and Chapter 17 is entirely about the difference.

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. 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.

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: at Bundesdruckerei, Germany's federal security printer and identity provider, for public-sector IT providers, in cooperative banking, and for security technology vendors.

Harbor is the piece of that work this book is about. In an environment where an auditor will eventually ask *what is running, where did it come from, and who could have changed it*, the registry is where the answer either exists or does not. He has built and operated Harbor installations on virtual machines and in clusters, connected them to directories, signed what they hold, and recovered them — which is a shorter list than it sounds and contains everything this book asks you to do.

Around it sits the rest of the platform: policy enforcement with Kyverno, identity and access with Keycloak, vulnerability and dependency evidence with Trivy and Dependency Track, and secrets management with HashiCorp Vault, which he has been deploying and operating since 2017 and which is the subject of the first book in this series.

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. Not a hard question, and not an unfair one: *which build is in production right now?* Everyone assumes it has an answer.

It usually does not, and the reason is almost never carelessness — it is that the tool that could have recorded it was installed as a place to put images and never asked to do anything else.

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 Harbor does. It will not teach you to run Harbor. Those are different skills, and only one of them is worth money.

What you need

A laptop with 16 GB of RAM, Docker, and roughly forty minutes for most chapters. Harbor runs ten containers and one of them is a vulnerability scanner that unpacks image layers, so this book asks for more memory than its predecessor did. No cloud account, no credit card, no spare server. 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/harbor-in-practice
make check
```

Work inside `books/harbor-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 certificate Chapter 2 spends nine steps explaining. `chapters/ch06/expiring-robots.sh` is the check that would have prevented the incident Chapter 6 opens with. 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: Harbor ships a minor release roughly every quarter, and a book with printed output goes stale on somebody else's schedule.

Why this book is shorter than it used to be

An earlier draft explained things. It explained what a registry is, why a digest is better than a tag, and what each field in a replication rule does. Then the world changed underneath it.

You have an AI assistant in the terminal next to this book. It will explain any Harbor setting, any OCI concept 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 number, message and duration printed here was measured against a running Harbor, and the book says which. An assistant will tell you that garbage collection frees space; this book tells you how much it freed, how long it took, and what was unavailable while it ran.

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 `unauthorized: unauthorized to access repository` 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, numbers that are real, and the failures worth causing.

How the chapters work

Every chapter has the same nine parts — with one deliberate exception, named below.

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.

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. The difference between somebody who has read about Harbor and somebody who has run it is almost entirely a matter of having produced the error messages before.

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

What a Harbor expert knows without looking it up. Ten questions with one-line answers. Cover the right column and work down. Any answer that was not immediate is your next five minutes.

Do this without looking. Five tasks with no walkthrough. Worked solutions at the end of the chapter — read them after you have tried, not instead. Chapter 17 is the exception in this too: its tasks are about your own installation, so there is nobody but you who can mark them.

Design Review. Four 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

Harbor has no certification, and this book is not preparation for one. What replaces it is the *Decision and Consequence* boxes and the *Design Review* sections — because the thing that distinguishes people who run registries well is not recall, it is having thought about the trade-off before the incident.

Chapter 17 is the centre of that. It has no lab at all. It answers whether Harbor belongs on a virtual machine or in Kubernetes, what large organisations should do, and why — and it is the one chapter in this book you could usefully hand to somebody who will never type any of the rest.

The chapters that will annoy you

Chapter 1 asks you to push one image over another and watch nothing object. Chapter 7 begins by refusing to let you change a setting, because of something you did two chapters earlier. Chapter 22 asks you to restore a registry that lists images nobody can pull, and then to lose the one file that makes the restore worthless.

These are the most valuable chapters in the book. Nobody who has recovered a registry 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 a5c7197d72
```

From Chapter 2 onwards there are two machines: your laptop and a virtual machine. Every command that could run on either is preceded by **On your laptop** or **On the VM**, and commands typed inside the virtual machine carry its prompt:

```
ubuntu@harbor:~$ docker compose ps
```

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

```
$ docker run -d --rm --name plain-registry \
    -p 5000:5000 \
    registry:2
```

Digests are abbreviated so that lines fit on the page — `sha256:c64c687cbea9...0483b5e` rather than sixty-four characters. Yours will be full length, and on a machine of a different

architecture they will not match the ones printed here at all. Chapter 8 explains why that is the correct behaviour rather than a problem.

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

Chapter 1 — Why a Private Registry

Decision and consequence

Decision	What it costs you later
Pull images straight from public registries	Rate limits, no provenance, tags that move under you, no place to say no
Run a registry of your own	A stateful service everything depends on, with storage, backups and upgrades of its own

Every chapter opens with the decision it forces on you. This one is the decision to have a registry at all, and it is the only decision in this book that most teams make by accident.

The problem

A build stops with this:

```
Step 1/9 : FROM node:20-alpine
toomanyrequests: You have reached your pull rate limit.
```

Waiting and retrying fixes it, which is why most teams never look further. Look further, and there are three problems rather than one.

Nobody can say which build is running. The deployment says `tracking:latest`, the running pods agree, and that is not an answer. The tag has been pushed fifty times and nothing records which push is live, which commit produced it, or who pushed it.

The registry is not run by anybody. In a great many organisations it is a `registry:2` container somebody started on a virtual machine years ago, with no authentication, because it was meant to be temporary.

The base image can disappear. Upstream tags get removed and retagged. When that happens, rebuilding the release you are running becomes impossible, and you find out at the worst moment.

None of these is the result of carelessness. Every decision behind them was the obvious one at the time. They are the result of a missing tool.

What pulling from anywhere actually costs

It is tempting to treat this as a convenience problem — a rate limit is annoying, a cache would fix it. It is not. Pulling from wherever produces four specific failures, and the design of Harbor is a direct response to each one.

You cannot answer what is running. A tag is a mutable label. Two people can read `tracking:latest` a week apart and get different software, and neither of them will notice, because the string is the same. The honest answer to "what is in production" is "whatever was pushed most recently", and that answer fails every audit ever written.

You cannot answer where it came from. There is no signature, no build record, no link back to a commit. An image that was replaced at 02:00 by someone with a stolen credential looks exactly like an image your pipeline built, because the only thing distinguishing them is a string that anyone with push access can overwrite.

You cannot stop a known-bad image. When a CVE lands in a base image on a Thursday afternoon, there is no point in the path from build to cluster where anything is empowered to say no. Scanning that only reports, and never blocks, is a dashboard, not a control.

You cannot survive the upstream. Rate limits, deleted tags, retagged releases, outages, a licence change, an air-gapped site, a customer who requires that no production dependency leaves the premises. Every one of those is somebody else's decision applied to your release schedule.

Hold those four failures in mind: **identity, provenance, control, availability**. Everything Harbor does is aimed at one of them.

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

Lab 1 — A registry, and everything it cannot do

Harbor is not installed yet. That is on purpose.

This lab runs the minimal thing that satisfies the word “registry”: the reference implementation, in one container, with no authentication, no scanning and no interface. It is roughly what is running in a great many organisations that never decided to run a registry. You will push to it, prove that a tag is a moving target, and then walk through the four questions and watch it answer exactly one of them.

Chapter 3 installs the version that is not a toy.

Step 1 — Check Docker

```
$ docker --version
Docker version 29.2.1, build a5c7197d72
```

Any recent version is fine. Podman and nerdctl work too; where the commands differ, this book says so. If this fails, Appendix A installs it.

Step 2 — Start a registry

```
$ docker run -d --rm --name plain-registry \
    -p 5000:5000 \
    -e REGISTRY_STORAGE_DELETE_ENABLED=true \
    registry:2
```

That environment variable is not the default, and the default is worth a sentence. The reference registry ships a configuration with no `delete` section at all, so deleting anything answers `405` with the error code `UNSUPPORTED` until somebody turns it on. A registry that refuses to delete by default is a reasonable design — and it is also the first hint that “what is stored” and “what can be removed” are separate questions, which is most of Chapter 11.

```
$ docker ps --format '{{.Names}}\t{{.Ports}}'
plain-registry  0.0.0.0:5000->5000/tcp
```

No TLS, no password, no configuration. It works immediately because Docker treats `localhost` and `127.0.0.1` as insecure registries without being asked. Remember that sentence — in Chapter 3 the host stops being `localhost` and the same setup stops working, and the error you get is one you will meet again.

Step 3 — Push something

```
$ docker pull alpine:3.20
$ docker tag alpine:3.20 localhost:5000/meridian/tracking:2.4.1
$ docker push localhost:5000/meridian/tracking:2.4.1
```

```
The push refers to repository
  [localhost:5000/meridian/tracking]
25f1d6b1951a: Pushed
2.4.1: digest: sha256:c64c687cbea9...0483b5e size: 1023

 Info -> Not all multiplatform-content is present and only the
         available single-platform image was pushed
```

Write that digest down, or leave the terminal open. The `digest:` line is the registry telling you the identity of what it just stored.

> **Docker 29 and the containerd image store.** A fresh Docker 29 installation stores images with containerd, and this book was measured against that. Two consequences run through every chapter. First, `docker push` sends an **OCI** manifest (`size: 1023`), not the older Docker one (`size: 527`), and prints the `Info ->` trailer above. Second, when a registry refuses something, Docker 29 reports only the status — `unexpected status from HEAD request to …: 401 Unauthorized` or `412 Precondition Failed` — and not the sentence the registry sent with it. Where this book prints Harbor's own message, that is what Harbor said; if your Docker shows only the number, the message is one `curl` away, and Appendix E shows both forms. If you would rather see the old behaviour, `docker info | grep driver-type` tells you which store you have, and Docker's documentation says how to switch.

Step 4 — Ask the registry what it has

```
$ curl -s http://localhost:5000/v2/_catalog
{"repositories":["meridian/tracking"]}
```

```
$ curl -s \
    http://localhost:5000/v2/meridian/tracking/tags/list
{"name":"meridian/tracking","tags":["2.4.1"]}
```

`/v2/` is the OCI distribution API. Harbor speaks it too, unchanged — that is why `docker push` needs no special client. Everything Harbor adds sits on a second, separate API, and Appendix C is a cookbook for it.

Step 5 — Read the digest from the registry

```
$ OCI='application/vnd.oci.image.manifest.v1+json'
$ V2='application/vnd.docker.distribution.manifest.v2+json'
$ curl -sI -H "Accept: $OCI" -H "Accept: $V2" \
    http://localhost:5000/v2/meridian/tracking/manifests/2.4.1 \
  | grep -i docker-content-digest
```

```
docker-content-digest: sha256:c64c687cbea9...0483b5e
```

Two `Accept` headers, because a registry will hand you an older manifest format if you do not say which ones you understand, and the older format has a different digest. That is not a detail: it is the reason two tools sometimes report two different digests for the same image.

Step 6 — Move the tag

Now push something entirely different under exactly the same tag.

```
$ docker pull busybox:1.36
$ docker tag busybox:1.36 \
    localhost:5000/meridian/tracking:2.4.1
$ docker push localhost:5000/meridian/tracking:2.4.1
```

```
2.4.1: digest: sha256:b7f3d86d6e84...7b9e29f size: 1023
```

```
$ curl -sI -H "Accept: $OCI" -H "Accept: $V2" \
    http://localhost:5000/v2/meridian/tracking/manifests/2.4.1 \
  | grep -i docker-content-digest
```

```
docker-content-digest: sha256:b7f3d86d6e84...7b9e29f
```

The tag did not change. The software did. No warning was printed, no permission was required, and nothing anywhere recorded that it happened.

That is the question this chapter opened with — *which build is running in production* — reproduced on your laptop in four commands.

Step 7 — Pull by digest instead

```
$ docker pull \
    localhost:5000/meridian/tracking@sha256:c64c687cbea9...
```

Substitute the full digest from step 3. The old image comes back, intact, even though the tag now points elsewhere. A digest cannot be repointed, because it *is* the content.

Pin by digest in anything that matters. Chapter 11 shows the other half of the answer, which is making the tag itself immutable so that step 6 is refused outright.

Step 8 — Ask the four questions

Work down the list from earlier in the chapter and watch the registry fail all four.

Who are you? Nobody asked. Anyone who can reach port 5000 can push anything to any repository, including over the top of yours.

```
$ curl -s -X DELETE \
    http://localhost:5000/v2/meridian/tracking/manifests/\
sha256:b7f3d86d6e84...7b9e29f -o /dev/null -w '%{http_code}\n'
202
```

```
$ curl -s http://localhost:5000/v2/meridian/tracking/tags/list
{"name":"meridian/tracking","tags":null}
```

That is a delete, accepted, with no credentials of any kind — and the tag it was reached by is gone with it.

What may you do? There are no roles, because there are no users. There are no projects either — `meridian/` is a naming convention that the registry does not understand.

What are you after? This one it answers well. Digests work.

Should you get it? There is nothing here that could form an opinion. No scanner, no signature verification, no policy, no audit log. Nothing to consult and nothing that records the question was asked.

One out of four.

Step 9 — Stop it

```
$ docker stop plain-registry
```

The container was started with `--rm`, so it and everything you pushed are gone. That is the last of the four failures, availability, arriving without being invited.

What Harbor is

Harbor is a registry that stores container images and other OCI artifacts, decides who may push and pull them, inspects what it holds, and copies it to where it is needed.

The word people fixate on is “stores”. The interesting words are “decides” and “inspects”. A plain registry stores. Harbor stores, and then has opinions about what it is storing.

Four capabilities, mapping onto the four failures above:

It gives artifacts a stable identity. Everything is addressed by content digest. Tags are labels on top of that, and Harbor can make a tag immutable so that it stops being a moving target. That is identity.

It records where things came from. Scan results, signatures, build metadata and an audit log of every push and pull, attached to the artifact rather than to a wiki page. That is provenance.

It can refuse. Projects, roles, robot accounts, and policies that block a pull when the artifact is unscanned, vulnerable or unsigned. Not a report. A refusal. That is control.

It holds its own copy. A proxy cache in front of a public registry, and replication between registries, so that an upstream outage or an air gap is an inconvenience rather than an outage. That is availability.

Harbor is a graduated project of the Cloud Native Computing Foundation. It is not a product of a single vendor, and nothing in this book requires a licence.

The four questions

Harbor is easier to hold in your head if you read it as four questions, asked in order, every time anything talks to it.

```
Who are you?         ->  Authentication
     |                   database, LDAP, OIDC, robot account
What may you do?     ->  Project roles
     |                   guest, developer, maintainer, admin
What are you after?  ->  An artifact
     |                   addressed by digest, labelled by tag
Should you get it?   ->  Policies
                         scan gate, signature, immutability
```

A client authenticates — with a password, a CLI secret, an OIDC session, or a robot account token. That identity carries **roles**, and roles are granted per **project**, which is the unit of everything in Harbor. The client then asks for an **artifact**, which is addressed by digest and usually reached through a tag. Before Harbor hands it over, **policies** get a vote.

Chapters 5 through 12 are these four questions, one at a time. Everything else in the book is built on them.

Tags, digests, and the thing everyone gets wrong

A pull reference has four parts, and Harbor uses all of them:

```
harbor.meridian.test/meridian/tracking:2.4.1
|__________________| |______| |______| |___|
       registry      project    repo    tag
```

The second part is the one that surprises people coming from Docker Hub. In Harbor, the first path segment is always a **project**. It is not decoration and it is not a username: it is the boundary that carries members, roles, quotas, retention rules and policies. You cannot push to Harbor without deciding which project you are pushing to, and that is deliberate.

Now the part that matters more than any other sentence in this chapter.

A tag is a label. A digest is the identity.

```
tag  2.4.1  ----->  manifest  sha256:c64c687cbea9...0483b5e
                      |
                      +-- config   sha256:8a1f7d20...
                      +-- layer 1  sha256:3c9e4b81...
                      +-- layer 2  sha256:71b0af6c...
```

The manifest is a small JSON document listing a configuration and a set of layers. Its digest is the SHA-256 of its own bytes, so it cannot lie: change anything at all and the digest changes. The tag is a separate record that points at a digest, and pointing it somewhere else is one API call.

Digests are abbreviated throughout this book so that lines fit on the page. Yours will be sixty-four hexadecimal characters — and if your laptop is an ARM machine they will not match the ones printed here at all, because the manifest for `linux/arm64` is a different document from the one for `linux/amd64`. Different digest, same image name. That is the chapter's point arriving early.

This is why `latest` is not a version, why `docker pull` twice can give you two different things, and why every serious deployment pins by digest. You will prove all three to yourself in the lab, on purpose, in about ten minutes.

Where the data lives

Harbor is not one process. The part of it that matters for backups is that it keeps its state in three different places, and they are not interchangeable.

```
+--------------+   +--------------+   +----------+
| PostgreSQL   |   |   storage    |   |  Redis   |
|              |   |              |   |          |
| projects     |   | blobs        |   | job queue|
| users        |   | manifests    |   | cache    |
| policies     |   |              |   | sessions |
| scan data    |   | disk or S3   |   |          |
+--------------+   +--------------+   +----------+
```

PostgreSQL holds the metadata. Projects, users, roles, robot accounts, retention rules, replication policies, scan results, the audit log. Everything Harbor knows *about* your artifacts.

The storage backend holds the bytes. Layer blobs and manifests, on a filesystem or in an object store. It knows nothing about projects or permissions; it is a content-addressed place to put bytes.

Redis holds work in progress. Job queues, caches and sessions. It is the only one of the three you can lose without losing data, and even then you lose whatever jobs were running.

Two consequences to absorb now, because both of them bite people in production:

A backup of one is not a backup. A database dump without the blobs gives you a registry that knows about ten thousand images and can serve none of them. Blobs without the database give you a disk full of anonymous bytes. Chapter 22 is about doing this correctly, and it begins by breaking it.

Deleting a tag does not free disk space. Deleting a tag removes a pointer. Removing the bytes is a separate, offline-ish operation called garbage collection, and it will not run while you are watching. Chapter 11 is where that surprise gets defused.

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. ### `http: server gave HTTP response to HTTPS client`

Cause it: push to a plain registry by name rather than by `localhost`:

```
$ docker push registry.internal:5000/tracking:2.4.1
```

```
Error response from daemon:
Get "https://registry.internal:5000/v2/": http: server gave
HTTP response to HTTPS client
```

The registry speaks HTTP; the client insisted on HTTPS. Docker makes an exception for `localhost` and `127.0.0.1` only. The moment the host is a name or a routable address, TLS is mandatory unless you explicitly declare the registry insecure.

Do not fix this by adding `insecure-registries` and moving on. That is how a temporary registry becomes a permanent one. Chapter 3 gives Harbor a real certificate, and Chapter 18 issues it from Vault.

`denied: requested access to the resource is denied`

Cause it: push into a first path segment that is not a project you may write to.

```
$ docker push localhost:5000/tracking:2.4.1
denied: requested access to the resource is denied
```

Against Harbor, this almost always means the first path segment is not a project you may write to — or is not a project at all. Harbor does not create projects implicitly. Against a plain registry it means the repository name is malformed.

`manifest unknown`

Cause it: ask for a tag that is not there, or for a manifest format the registry will not convert to.

```
{"errors":[{"code":"MANIFEST_UNKNOWN",
"message":"manifest unknown"}]}
```

Either the tag genuinely does not exist, or you asked for a manifest format the registry does not have and it declined to convert. Check the tag with `/tags/list` first; if the tag is there, send both `Accept` headers from step 5.

`toomanyrequests: You have reached your pull rate limit`

Cause it: pull anonymously from Docker Hub on a shared address, repeatedly.

The error that opened the chapter. Anonymous pulls from Docker Hub are rate limited per IP address, which means a CI runner shared by forty builds spends its budget before breakfast. Authenticating raises the limit. A proxy cache removes the problem, and that is Chapter 14.

`no basic auth credentials`

Cause it: log in to Docker Hub, then push to your own registry.

```
$ docker login
$ docker push harbor.meridian.test/library/tracking:2.4.1
```

```
errors:
denied: requested access to the resource is denied
no basic auth credentials
```

You are not logged in to that registry. Note that `docker login` credentials are stored per registry host, so being logged in to Docker Hub does nothing for `harbor.meridian.test`, which is the name your own registry gets in Chapter 2.

`connection refused` on port 5000

Cause it: on macOS, start the lab registry with AirPlay Receiver enabled.

On macOS, port 5000 is claimed by AirPlay Receiver. Turn it off in System Settings under General, or use `-p 5001:5000` and adjust the commands.

What you should now be able to do

You can now:

- Explain the cost of pulling from wherever, as four failures: identity, provenance, control, availability
- Describe Harbor's flow as four questions: who are you, what may you do, what are you after, should you get it
- Say precisely what a tag is, what a digest is, and why only one is an identity
- Name the three places Harbor keeps state, and why a backup of one is worthless
- Move a tag to different content and prove it happened
- Explain why `localhost` needs no TLS and anything else does

Do this without looking

1.1 Push `alpine:3.20` to the plain registry under two different tags in the same repository. Read the digest of each. Explain the result in one sentence.

1.2 Using `curl` only, list every tag in every repository in the registry. You will need two calls and the output of the first one.

1.3 Push an image, then delete its manifest by digest as shown in step 8. Now ask for the tag again with `/tags/list`. Is the tag still listed? Explain what that tells you about the relationship between a tag and a manifest.

1.4 Start the registry again without `--rm` and with a named volume mounted at `/var/lib/registry`. Push an image, stop and remove the container, start a new one on the

same volume, and confirm the image survived. Which of the four failures did you just fix, and which three did you not?

1.5 Find a public image that publishes both an image index and a single-architecture manifest. Request its manifest with only the `application/vnd.docker.distribution.manifest.v2+json` accept header, then with only the index header. Compare the two digests. Being able to cause this discrepancy on purpose is how you prove you understand step 5.

What a Harbor expert knows without looking it up

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

Question	Answer
Exception Docker makes for plain HTTP	`localhost` and `127.0.0.1`
What `insecure-registries` really costs	a temporary registry becomes permanent
First path segment against Harbor	the project
Whether Harbor creates projects implicitly	no
What rate limits are counted against	the IP address
Where docker login credentials are stored	per registry host
Port AirPlay claims on macOS	5000
Error for a missing tag or format	`manifest unknown`
Two headers a manifest request may need	both index and manifest media types
What removes a rate-limit problem entirely	a proxy cache

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

Design Review

Each scenario states a requirement. Give the decision and, more importantly, the reason. Model answers follow the exercise solutions.

1. A team of four runs one Kubernetes cluster and pulls perhaps sixty images a day, all from Docker Hub. They have never hit a rate limit. They ask whether they need a registry at all. What do you tell them, and what would change your answer?

2. An architect argues that pinning every deployment by digest makes a registry unnecessary, because a digest cannot be forged. Where is the hole in that argument?

3. A regulated customer requires that no production dependency is fetched from outside their premises at deploy time. Rank the four failures by how directly this requirement bears on each.

4. Your organisation already stores build artifacts in a general purpose artifact repository that also speaks the OCI distribution API. What would have to be true for that to be sufficient, and what would have to be true for it not to be?

Worked solutions

1.1 Both tags report the same digest. A digest identifies content, and the content is identical; tags are labels and a manifest may carry any number of them. This is also why deleting one tag does not delete the image.

1.2 Call `/v2/_catalog` to get the repository list, then call `/v2/<name>/tags/list` for each entry:

```
$ for r in $(curl -s http://localhost:5000/v2/_catalog \
      | tr -d '{}"' | cut -d: -f2 | tr ',' ' '); do
    curl -s http://localhost:5000/v2/$r/tags/list; echo
  done
```

1.3 The tag disappears — `/tags/list` answers `"tags":null` rather than an empty array, which is worth knowing before you write something that iterates over it. Deleting a manifest deletes every tag pointing at it, because a tag is a reference to a manifest and not the other way round. The blobs, however, are still on disk — removing them is garbage collection, and it is Chapter 11.

1.4 The image survives. You fixed part of availability, and only the part that concerns your own registry restarting. Identity is unchanged, because tags still move. Provenance is unchanged, because nothing is recorded. Control is unchanged, because there is still nothing that could refuse. One of four, and the least valuable one.

1.5 The two digests differ. An image index and the manifest it points at are two distinct documents with two distinct digests, and a registry serves whichever one your `Accept` header asks for. If a scanner reports one digest and a deployment pins the other, they are both right and they are talking about different objects.

Design Review — model answers

1. Not yet, and say so plainly — a registry is a stateful service with backups, storage growth and upgrades, and four people running one cluster should not take that on for a problem they do not have. What changes the answer: the first rate limit that stops a release; the first auditor who asks what is in production; the first requirement to run somewhere without internet access; the first time a base image they depend on disappears upstream. Any one of those, and the calculation inverts. Note which failure each of those is.

2. A digest guarantees that what you received is what the digest names. It says nothing about whether that content is still available to be fetched, and nothing about whether it should be trusted. Pinning by digest fixes identity completely and touches neither availability — the upstream can delete it — nor control, because nothing checks the pinned digest for known vulnerabilities before it runs.

3. Availability is the requirement, stated directly: the artifact must be reachable without leaving the premises. Control comes second and is implied, because a site that cannot reach the internet also cannot reach a hosted scanner, so scanning has to run locally. Provenance third: an offline site typically has to prove the chain of custody it can no longer delegate. Identity is unaffected — it is equally broken in both cases.

4. It is sufficient if it does everything in the four questions: per-project roles, robot credentials, a scanner whose result can block a pull rather than merely report, immutable tags, signature verification, and replication or caching. It is not sufficient if scanning is advisory, if the OCI support is a compatibility layer that lags the specification, or if the artifact repository is itself the single point of failure for a cluster that must start without it. The question is never whether it speaks `/v2/`. It is whether it can say no.

Chapter 2 — Building Your Lab

Decision and consequence

Decision	What it costs you later
Put the registry on its own machine	A machine to patch, back up and resize, and a name that has to resolve from somewhere
Give it a certificate from a CA you run	A root key you now own, a renewal nobody will remember, and every client has to be told about it
Declare it insecure instead	A setting on every client that ever talks to it, which outlives the lab and travels to production in a runbook

The third row is the one this chapter exists to prevent.

The problem

Before Harbor can be installed, three things have to be decided, and all three are easier to get right now than later.

Where it runs. The obvious answer is the Kubernetes cluster that runs everything else. Chapter 17 argues against it at length; the short version is that the runbook step which fails first in a cluster outage is *pull the recovery image*.

What it is called. A certificate certifies a name, so the registry needs one before it needs anything else — and an IP address is not one.

Whether it can be rebuilt. A registry that exists because somebody typed commands on a machine in 2023 is a machine nobody dares touch.

The alternative to a certificate is `insecure-registries`, and it is worth naming the cost now: that line is added by hand to every client that ever talks to the registry, it is never removed, and it follows the registry into production in a runbook.

This chapter is the machine, the name and the certificate. The registry itself is Chapter 3.

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

Harbor's released images are built for **amd64 only**. Ask the registry and it answers with a single manifest rather than an index:

```
$ IMG=goharbor/harbor-core:v2.15.2
$ docker manifest inspect "$IMG" | jq -r '.mediaType'
application/vnd.docker.distribution.manifest.v2+json
```

One manifest, one platform — Chapter 8's distinction arriving early. There is an `arm64` build under the unversioned `dev-arm64` tag, which is a development artifact and not something to pin a lab to.

Multipass on Apple silicon creates an **arm64** machine, and Harbor in it does not start:

```
exec /usr/bin/python3: exec format error
```

qemu's user-mode emulation gets further and fails worse — the containers start and Valkey, Harbor's Redis, dies under it:

```
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
```

Core then retries forever and nothing comes up.

So on Apple silicon, use Colima with Rosetta, not Multipass. It runs an arm64 machine and translates the amd64 binaries properly. Appendix A has the four commands; everything from Chapter 3 onwards is unchanged.

Intel Macs, Windows and Linux on amd64: Multipass, as described below.

If you already have an Ubuntu host — a cloud instance, a Proxmox guest, a spare machine — you can skip the Multipass steps entirely and use it. Appendix A says which of the cloud-init steps to run by hand.

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

Build it first. Why it is a machine and not a container, and what that buys you, is after the lab.

Lab 2 — The machine, the name, the certificate

By the end of this lab you will have pushed an image over TLS to a registry with a real name and a properly verified certificate — and Harbor still will not be installed. That is the point. When Chapter 3 installs it, the trust problem is already solved and out of the way.

Step 1 — Check what you have

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

```
Harbor in Practice - prerequisites
Pinned Harbor v2.15.2, chart v1.19.2

  ok      docker      Docker version 29.2.1, build a5c7197d72
  ok      curl        curl 8.7.1
  ok      jq          jq-1.7.1
  ok      multipass   multipass 1.16.3
  MISSING kubectl     needed from Chapter 15
```

Anything needed from Chapter 15 or later can wait. `make install` prints the commands for your platform.

Step 2 — Launch the machine

```
$ make vm-up
```

```
Launching 'harbor' (4 cpu, 8G, 60G)
Launched: harbor

VM is up.

  name  harbor
  ip    192.168.64.7
```

Four processors and eight gigabytes because Harbor runs ten containers and one of them is a vulnerability scanner that unpacks image layers. Two and four gigabytes will start it and will make Chapter 9 miserable.

What `make vm-up` actually runs is one Multipass command with a cloud-init file. Read it — it is thirty lines and it is the entire definition of the machine:

```
$ cat vm/cloud-init.yaml
```

The only part worth defending here is that it installs Docker from Docker's own repository rather than Ubuntu's package. Harbor's installer checks the version of the compose plugin, and the distribution package lags far enough behind to fail that check.

Step 3 — Make the name resolve

Take the IP address from step 2 and add it to your hosts file. On macOS and Linux that is `/etc/hosts`; on Windows it is `C:\Windows\System32\drivers\etc\hosts`, edited as Administrator.

```
192.168.64.7  harbor.meridian.test
```

```
$ ping -c1 harbor.meridian.test | head -1
PING harbor.meridian.test (192.168.64.7): 56 data bytes
```

Multipass assigns addresses by DHCP, so this can change when you recreate the machine. `make diagnose` checks this and tells you when the entry has gone stale, which it will, at the least convenient moment.

Step 4 — Look inside

```
$ multipass shell harbor
```

```
ubuntu@harbor:~$ docker --version
Docker version 29.2.1, build a5c7197d72
ubuntu@harbor:~$ docker compose version
Docker Compose version v2.40.2
ubuntu@harbor:~$ df -h /data | tail -1
/dev/sda1        59G  2.1G   54G   4% /data
```

Type `exit` to come back. From here on, this book says **on the VM** or **on your laptop** before every command that could be either.

Step 5 — Create a certificate authority

On your laptop, in the repository:

```
$ mkdir -p certs && cd certs
$ openssl genrsa -out ca.key 4096
$ openssl req -x509 -new -nodes -sha256 -days 3650 \
    -key ca.key -out ca.crt \
    -subj "/O=Meridian Freight/CN=Meridian Lab Root CA"
```

Ten years on the root, because renewing a root is a project and not a task. Chapter 13 of Book One is about why that number is what it is; if you have not read it, the short version is that a root certificate is renewed by visiting every machine that trusts it.

Step 6 — Sign a certificate for the registry

```
$ openssl genrsa -out harbor.key 4096
$ openssl req -new -sha256 -key harbor.key -out harbor.csr \
    -subj "/O=Meridian Freight/CN=harbor.meridian.test"
```

Now the extension file. This is the step people skip, and skipping it produces an error message that blames the wrong thing.

```
$ cat > v3.ext <<'EOF'
basicConstraints=CA:FALSE
keyUsage=digitalSignature,keyEncipherment
extendedKeyUsage=serverAuth
subjectAltName=@alt

[alt]
DNS.1=harbor.meridian.test
IP.1=192.168.64.7
EOF
```

```
$ openssl x509 -req -sha256 -days 365 \
    -in harbor.csr -CA ca.crt -CAkey ca.key \
    -CAcreateserial -extfile v3.ext -out harbor.crt
```

Confirm the name landed where it has to be:

```
$ openssl x509 -noout -text -in harbor.crt \
  | grep -A1 'Subject Alternative Name'
```

```
X509v3 Subject Alternative Name:
    DNS:harbor.meridian.test, IP Address:192.168.64.7
```

If that section is missing, nothing else in this chapter will work. Go clients — which means Docker, containerd, Harbor and Kubernetes — have ignored the Common Name field since Go 1.15. A certificate whose name is only in the Subject is a certificate with no name at all.

The repository does all of step 5 and 6 in one script if you would rather read it than type it:

```
$ ./chapters/ch02/make-certs.sh 192.168.64.7
```

Step 7 — Install the root into the trust stores

Two places: the machine that serves, and the machine that connects.

On your laptop, targeting the VM:

```
$ multipass transfer certs/ca.crt harbor:/tmp/ca.crt
$ multipass exec harbor -- sudo cp /tmp/ca.crt \
    /usr/local/share/ca-certificates/meridian-lab.crt
$ multipass exec harbor -- sudo update-ca-certificates
```

```
1 added, 0 removed; done.
```

On your laptop, and here the platforms genuinely differ. This is not Docker being inconsistent; it is three operating systems with three different ideas of what a trust store is.

Platform	Where the root goes
Linux	`/etc/docker/certs.d/harbor.meridian.test/ca.crt` for Docker, **and** the system store for `curl`
macOS	System keychain, marked trusted, then restart Docker Desktop
Windows	Local Machine, Trusted Root Certification Authorities

On macOS:

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

Then restart Docker Desktop. It reads the keychain at start and not after.

On Linux, note that the directory is named after the registry host and that Docker does not need restarting:

```
$ sudo mkdir -p /etc/docker/certs.d/harbor.meridian.test
$ sudo cp certs/ca.crt \
    /etc/docker/certs.d/harbor.meridian.test/ca.crt
```

That directory is Docker's alone. `curl` — and step 8 uses it — reads the system store, which on Debian and Ubuntu is a second copy:

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

Docker reads the system store only when it starts, so if you rely on that copy alone, restart Docker. The `certs.d` directory is read on every connection, which is why it is the one this book uses.

Step 8 — Prove it, before Harbor exists

Run the same registry from Chapter 1 on the VM, this time with the certificate in front of it.

On the VM:

```
ubuntu@harbor:~$ mkdir -p ~/certs
```

On your laptop:

```
$ multipass transfer certs/harbor.crt harbor:/home/ubuntu/certs/
$ multipass transfer certs/harbor.key harbor:/home/ubuntu/certs/
```

On the VM:

```
ubuntu@harbor:~$ docker run -d --rm --name tls-registry \
    -p 443:5000 \
    -v /home/ubuntu/certs:/certs \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/harbor.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/harbor.key \
    registry:2
```

On your laptop:

```
$ curl -s https://harbor.meridian.test/v2/ && echo OK
{}OK
```

No `-k`. That is the whole point of the chapter: `curl` verified the chain, matched the name, and said nothing, because there was nothing to complain about.

```
$ docker pull -q alpine:3.20
$ docker tag alpine:3.20 \
    harbor.meridian.test/meridian/tracking:2.4.1
$ docker push harbor.meridian.test/meridian/tracking:2.4.1
```

```
The push refers to repository
  [harbor.meridian.test/meridian/tracking]
2.4.1: digest: sha256:c64c687cbea9...0483b5e size: 1023
```

A name, a verified certificate, and a push. Everything Chapter 1's registry could not do about identity is still missing — the tag will still move under you — but the transport is now correct, and Chapter 3 can install Harbor into a lab where trust is already solved.

Step 9 — Tidy up

On the VM:

```
ubuntu@harbor:~$ docker stop tls-registry
```

Leave the machine and the certificates. Chapter 3 uses both.

Why a virtual machine and not just Docker

Harbor's installer is not a container you run. It is a tarball that expects a Linux host: it writes to `/data`, generates configuration into `/opt/harbor`, binds ports 80 and 443, and drives `docker compose` itself. Pointing it at Docker Desktop's hidden virtual machine works until you need to reach it by name, mount a volume, look at a log file, or reboot it — at which point you are administering a machine you cannot log into.

So the lab uses a real one. Three properties matter, and only three:

It is Linux. The installer, the file paths and every command in Chapters 3 to 14 are what you would type on a server.

It is reproducible. The machine is described by a file, not by the sequence of things somebody typed in 2023. Delete it and rebuild it in four minutes.

It is the same on every host — with one exception, and it is a big one if it is yours.

Multipass gives all three in one command. Chapter 17 returns to the question of where a real Harbor belongs, with an answer that is not "a laptop".

The name

A certificate certifies a name. So the registry needs one, before it needs anything else.

This book uses `harbor.meridian.test`. 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 ever go looking for it on the internet. Using `harbor.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, the name resolves through your hosts file. That is a manual step, it is honest about being one, and Chapter 15 replaces it when the cluster brings its own DNS.

Three ways to be trusted, ranked

Docker refuses to talk to a registry whose certificate it cannot verify. There are exactly three ways out of that, and the order matters.

A certificate from a public CA. Correct, and unavailable here — a public CA will not issue for `.test`, and it should not. This is what you use in production, and Chapter 3 says how the same steps change when you have one.

A private CA that you run. What this chapter does. You create a root, you sign a server certificate with it, and you install the root into the trust store of everything that talks to the registry. Real certificates with a real chain, verified properly. The cost is that the root is now yours to protect and the certificate is yours to renew — and in Chapter 18 both of those problems move to Vault, which is what Vault is for.

Declaring the registry insecure. `insecure-registries` in the Docker configuration turns verification off for that host. It takes one line and it is the answer given by most of the internet.

Do not use the third one. Not because it is insecure in a lab — it is a lab — but because of what it does afterwards. The line has to be added to every client: every developer laptop, every CI runner, every Kubernetes node. It is added by hand, it is never removed, and it follows the registry into production in the form of a runbook step that says “add this to daemon.json”. You are not saving a step. You are distributing a setting to machines you will lose track of.

The private CA takes eleven minutes. Do that.

```
ca.key  ---signs--->  harbor.crt   (harbor.meridian.test)
  |                       |
  |                       +-- presented by the registry
  |
ca.crt  ---installed into--->  the VM's trust store
                               Docker on your laptop
                               every Kubernetes node, Chapter 15
```

What reset means

Every chapter in this book begins from a defined state and can be returned to it. That is not a courtesy. It is the difference between a lab you experiment in and a lab you are afraid of.

Three levels, and they are not the same:

Level	Command	What survives
Restart	`multipass stop` and `start`	Everything
Reset the lab	`make reset`	The VM; Harbor's data does not
Teardown	`make teardown`	Nothing

You will use the middle one constantly from Chapter 3 onwards, and you will use the last one twice: once at the end of the book, and once in Chapter 22, when you delete something on purpose and find out whether your backup was real.

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: push before installing the root, or install it into a directory whose name does not match the host exactly.

```
Error response from daemon: Get
"https://harbor.meridian.test/v2/": x509: certificate signed by
unknown authority
```

The client does not have your root. On Linux, check the directory name in `/etc/docker/certs.d/` character for character — it must match the registry host including the port if you use one. On macOS, you almost certainly did not restart Docker Desktop.

Test the chain independently of Docker, which narrows it in one command:

```
$ openssl s_client -connect harbor.meridian.test:443 \
    -CAfile certs/ca.crt < /dev/null 2>&1 | grep Verify
Verify return code: 0 (ok)
```

If `openssl` says 0 and Docker still complains, the problem is Docker's trust store, not your certificate.

`x509: certificate relies on legacy Common Name field`

Cause it: sign the certificate without `-extfile` in step 6.

```
x509: certificate relies on legacy Common Name field, use SANs
instead
```

You skipped the extension file in step 6, or `-extfile` was not passed to `openssl x509`. Re-sign it. Nothing else fixes this, and no amount of reinstalling the root will.

`certificate is valid for X, not harbor.meridian.test`

Cause it: recreate the machine so the IP changes, then connect by the old address.

The Subject Alternative Name does not include the name you connected with. Common cause: you recreated the machine, the IP changed, and you are now reaching it by an address that is in the certificate no longer. Re-sign with the new address, or stop using the address and use the name.

`dial tcp: lookup harbor.meridian.test: no such host`

Cause it: remove the hosts file entry, or edit the file without administrator rights so the save goes elsewhere.

The hosts file entry is missing, or you edited the wrong one. On Windows the file must be edited as Administrator, and an editor that silently saves to a user directory instead is a genuinely common trap.

```
$ make diagnose
```

`bind: address already in use` on the VM

Cause it: start a second listener on 443 inside the machine.

Something already holds port 443 inside the machine. In Chapter 3 that something will be Harbor itself, which is the usual reason this appears later in the book:

```
ubuntu@harbor:~$ sudo ss -lntp | grep ':443'
```

`launch failed: instance "harbor" already exists`

Cause it: run `make vm-up` twice.

`make vm-up` is safe to run twice; it says so and does nothing. If you want the machine rebuilt from scratch:

```
$ multipass delete harbor --purge
$ make vm-up
```

Your hosts file entry will be wrong afterwards. It always is.

What you should now be able to do

You can now:

- Say why the lab uses a virtual machine, in terms of what Harbor's installer expects
- Rank the three ways to make a registry trusted, and give the argument against `insecure-registries` that is about distribution
- Create a root CA and sign a certificate with a SAN, and recognise on sight what a certificate without one does to a Go client
- Install a root into a Linux store, a Docker trust directory and the macOS keychain, and say why the three differ
- Verify a chain with `openssl s_client` independently of Docker
- Describe the three levels of reset and what each keeps

Do this without looking

2.1 Delete the virtual machine and rebuild it with `make vm-up`. Time it. Then explain, in one sentence, what makes this different from the contractor's machine in Chapter 1.

2.2 Sign a second certificate for `harbor-2.meridian.test` from the same root, without recreating the root. You will need this in Chapter 13, when replication needs somewhere to replicate to.

2.3 Deliberately sign a certificate with no `-extfile`. Serve it, push to it, and read the error. Then fix it. Being able to cause an error on purpose is how you prove you understand it.

2.4 Set the server certificate's lifetime to one day instead of 365. Confirm it works. Then set the VM's clock forward two days with `multipass exec harbor -- sudo date -s '+2 days'` and try again. Describe both the error and what an unattended renewal would have to do.

2.5 Without editing your hosts file, make `curl` reach the registry by name using only a `curl` flag. Then explain why that flag does not help `docker push`.

What a Harbor expert knows without looking it up

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

Question	Answer
Directory Docker reads certificates from	`/etc/docker/certs.d/<host>`
Whether the port belongs in that name	yes, if you use one
Command that tests a chain without Docker	`openssl s_client -CAfile`
Return code that means the chain is good	0
Field modern clients require	the SAN
What `-extfile` supplies	that field
What macOS needs after installing a root	a Docker Desktop restart
Why a rebuilt VM breaks the certificate	the address changed
Where a Windows hosts file must be edited	as Administrator
What `make vm-up` does twice	nothing

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

Design Review

1. A colleague proposes skipping the private CA and adding `insecure-registries` to the four machines that need it, arguing that the lab is not production and the CA is ceremony. Give the strongest version of their argument, then the reason you still would not do it.

2. The lab resolves the registry name through a hosts file on every client. Name two ways this stops working as soon as more than one person is involved, and say what replaces it.

3. A requirement arrives that the registry certificate must be rotated every ninety days. Sketch what has to exist for that to be routine rather than an outage, and name the chapter of this book where each piece appears.

4. The VM has 60 GB. A colleague asks whether that is enough. What do you need to know before answering, and which of Harbor's three storage locations does the question actually concern?

Worked solutions

2.1 Around four minutes, most of it package installation. The difference is that the machine is defined by `vm/cloud-init.yaml` rather than by the sequence of commands somebody once typed, so rebuilding it is a routine operation rather than a risk.

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

```
$ openssl genrsa -out harbor-2.key 4096
$ openssl req -new -sha256 -key harbor-2.key \
    -out harbor-2.csr \
    -subj "/O=Meridian Freight/CN=harbor-2.meridian.test"
$ sed 's/^DNS.1=.*/DNS.1=harbor-2.meridian.test/' v3.ext \
    > v3-2.ext
$ openssl x509 -req -sha256 -days 365 -in harbor-2.csr \
    -CA ca.crt -CAkey ca.key -CAcreateserial \
    -extfile v3-2.ext -out harbor-2.crt
```

The point is that clients already trust the root, so the second certificate needs no work on any client at all. That property is the whole reason to run a CA rather than to distribute self-signed certificates.

2.3 `x509: certificate relies on legacy Common Name field, use SANs instead`. The certificate is otherwise valid and correctly signed, which is what makes the error confusing the first time: nothing is wrong with the chain, and the name simply is not in a field anyone reads.

2.4 The first attempt works. After the clock moves, every client fails with `x509: certificate has expired or is not yet valid`. An unattended renewal has to issue a new certificate, place it where the server reads it, and cause the server to reload — and

the third step is the one that gets forgotten, because a renewed certificate on disk that nobody reloaded looks exactly like a renewal that worked. Chapter 18 does all three from Vault.

2.5

```
$ curl --resolve harbor.meridian.test:443:192.168.64.7 \
    https://harbor.meridian.test/v2/
```

`--resolve` is a flag of the client process. `docker push` is not performed by your shell: it is performed by the Docker daemon, which is a different process with its own name resolution and no such flag. That distinction explains a great deal of confusing Docker behaviour and it returns in Chapter 15, where the daemon is on a Kubernetes node and resolves nothing you configured.

Design Review — model answers

1. The strongest version: a lab exists to be thrown away, the CA adds four steps and a private key to protect, and TLS verification teaches you nothing about Harbor. That is all true. The reason not to do it is that the setting does not stay in the lab. It is applied per client, by hand, and it has no expiry — so it accumulates on laptops and runners, and when the same registry appears in production the runbook that gets copied is the lab's. You are not choosing between secure and insecure. You are choosing whether a workaround gets distributed to machines you will lose track of.

2. It stops working when a second person joins, because each of them must edit a file by hand and nobody will notice when one of them has it wrong; and it stops working when a machine you do not control has to pull — a CI runner, a Kubernetes node — because there is no hosts file you can reach. Both are replaced by a DNS record. In the lab, Chapter 15 substitutes the cluster's own DNS; in production it is a real record, and it should exist before the certificate does.

3. Something must issue certificates on demand rather than by hand (Chapter 18, Vault PKI); something must fetch and place the new certificate before the old one expires (Chapter 18, Vault Agent); something must make the server load it without an outage (Chapter 18, the agent's restart command, and Chapter 21 if there is more than one replica); and something must tell you when it did not happen (Chapter 24, monitoring the expiry rather than the renewal job). Ninety days is not the hard part. Reload and alerting are.

4. You need the average compressed size of the images, how many tags you will keep, and the retention and garbage collection policy — because without retention the answer is always no, eventually. The question concerns the storage backend, which is one of Harbor's three storage locations and the only one that grows without bound. PostgreSQL grows slowly with metadata and Redis is a cache. Chapters 11 and 12 are retention and quotas; Chapter 21 moves the backend to object storage, at which point the question stops being about the virtual machine at all.