ITI8801 · Cloud Architectures and DevOps

Containers: images, Dockerfiles, registries

Siim Rebane · TalTech · Autumn 2026

Describing an API

The Notes api is used by programs that did not write it: the frontend, a validator, another team's service, a test suite. Each needs to know, for every endpoint:

- the path and the HTTP method
- what goes in: path parameters, query parameters, the request body and its fields
- what comes out: every status code it can answer with, and the body for each
- the types: a number is a number, a date is a date, a field is required or not

The code knows all of this, but reading someone's code is not a way to learn an interface, and a prose description drifts away from the code with every change.

OpenAPI

OpenAPI is a standard format for describing an HTTP API as a YAML or JSON document. Version 3.1 is current.

Part of the document	Holds
`info`	the API's name and version
`servers`	the base URLs it is served from
`paths`	each path, and under it each **operation**: one HTTP method on that path
an operation's `parameters`, `requestBody`	what the request may and must contain
an operation's `responses`	every status code, and the body schema for each
`components` → `schemas`	reusable data types, written in JSON Schema: types, required fields, lengths, formats

People read it rendered as a web page; programs read the same file to generate client code, check requests, or test the API.

The Notes API, described: `/api/notes`

```yaml
openapi: 3.1.0
info: { title: Notes API, version: "1.0" }
servers:
  - url: http://localhost:8090
security: []
paths:
  /api/notes:
    get:
      operationId: listNotes
      responses:
        "200":
          description: All notes, oldest first
          content:
            application/json:
              schema:
                type: array
                items: { $ref: "#/components/schemas/Note" }
    post:
      operationId: createNote
      requestBody:
        required: true
        content:
          application/json:
            schema: { $ref: "#/components/schemas/NewNote" }
      responses:
        "201": …  # a Note
        "400": …  # an Error: the text is missing, empty or too long
```

The Notes API, described: `/api/notes/{id}`

```yaml
/api/notes/{id}:
  parameters:
    - name: id
      in: path
      required: true
      schema: { type: integer, minimum: 1 }
  get:
    operationId: getNote
    responses:
      "200":
        description: The note
        content:
          application/json:
            schema: { $ref: "#/components/schemas/Note" }
      "404":
        description: No note with this id
        content:
          application/json:
            schema: { $ref: "#/components/schemas/Error" }
  delete:
    operationId: deleteNote
    responses:
      "204": { description: Deleted }
      "404": …  # an Error
```

The Notes API, described: schemas

```yaml
components:
  schemas:
    NewNote:
      type: object
      required: [text]
      properties:
        text: { type: string, minLength: 1, maxLength: 200 }
    Note:
      type: object
      required: [id, text, created_at]
      properties:
        id: { type: integer }
        text: { type: string }
        created_at: { type: string, format: date-time }
    Error:
      type: object
      required: [error]
      properties:
        error: { type: string }
```

Reading the schemas

Line	Means
`$ref: "#/components/schemas/Note"`	this body is a `Note`, defined once below
`required: [text]`	a request without `text` is invalid
`minLength: 1, maxLength: 200`	the limits the api enforces, written where every client can read them
`format: date-time`	the string is a timestamp such as `2026-09-25T22:36:14Z`
`parameters:` under `/api/notes/{id}`	applies to every operation on that path: `{id}` is a positive integer
`security: []`	this API has no authentication, said explicitly
`operationId: getNote`	a unique name per operation; generated client code uses it as the function name

The problem

The Notes application needs different things at different times:

When	Needs
building the page	Node.js 22 and npm, and the page's npm packages
building the binary	Go 1.25 and the Go modules the code imports
running	one 11 MB binary

Installing all of that on every machine that runs the application means a different Node and Go version on every machine, minutes of installing on every boot, and machines that differ from each other in ways nobody wrote down.

The boot scripts in weeks 2 to 4 ran one line instead, `docker run … <image>` with the course's image, the beacon and then Notes, and installed nothing else. That line starts a **container** from an **image**.

What a container is

A **container** is an ordinary process on a Linux machine, started from an image, that:

- sees its own filesystem, its own process list, its own network interfaces and its own hostname
- is limited in how much memory and CPU it may use
- shares the machine's kernel with every other process on it

The first two are features of the Linux kernel: **namespaces** decide what a process can see, **cgroups** decide how much it may use. Docker is a program that sets both up and starts the process.

There is no second operating system running. There is one kernel and a process that has been given a restricted view of it.

Container and virtual machine

	Virtual machine	Container
kernel	its own, booted inside the VM	the host's, shared
contains	a whole operating system	the application's files and what they need
starting it means	booting an operating system	starting a process
start time	the time an operating system takes to boot	0.55 s measured for `docker run --rm alpine:3.22 true` on a VM with 1 vCPU
isolation comes from	the hypervisor and the CPU's virtualisation	kernel features: namespaces and cgroups

The two are combined in practice: the containers in weeks 2 to 4 ran inside cloud VMs. The VM separates your workload from other customers'; the container packages the application.

A container is a process on the host

A container running `sleep 3600`, seen from inside and from the host:

```
$ docker exec demo ps
PID   USER     TIME  COMMAND
    1 root      0:00 sleep 3600
    7 root      0:00 ps

$ ps -o pid,user,cmd -C sleep          # on the host
    PID USER     CMD
 175191 root     sleep 3600
```

The same process has two numbers: 1 inside the container, 175191 on the host. Inside, it is the first and only process; on the host, it is one of hundreds, visible to `ps` and `top` like any other.

`docker inspect demo --format '{{.State.Pid}}'` prints the host's number: `175191`.

Namespaces

A **namespace** gives a process its own copy of one kind of system resource. A process in a new namespace sees only what is in that namespace.

Namespace	The container gets its own
`pid`	process numbers; its first process is PID 1
`net`	network interfaces, IP addresses, routing table, ports
`mnt`	filesystem tree, starting from the image's files
`uts`	hostname
`ipc`	shared memory and message queues
`cgroup`	view of the cgroup tree
`user`	user and group numbers (Docker does not use it by default)

Docker creates a new namespace of each kind for every container, except `user`, and `time`, which is not in the table.

One kernel

```
$ uname -r ; docker exec demo uname -r
7.0.0-27-generic
7.0.0-27-generic

$ docker exec demo cat /etc/os-release | head -2
NAME="Alpine Linux"
ID=alpine
```

The host runs Ubuntu 26.04; the container's files are Alpine Linux's. Both run on the same kernel, 7.0.0-27. "An Alpine container" means Alpine's files and programs, on whatever kernel the host has.

Consequences:

- a Linux container needs a Linux kernel. Docker Desktop on macOS and Windows runs a small Linux VM and starts the containers in it
- a container cannot load a different kernel or kernel modules
- a kernel vulnerability is a vulnerability of every container on that host

cgroups: limits

A **cgroup** (control group) is a set of processes with limits the kernel enforces: memory, CPU time, number of processes, disk throughput. Docker puts each container in its own cgroup and writes the limits you give it there.

```
$ docker run -d --name mem --memory 64m --cpus 0.5 alpine:3.22 sleep 3600
$ cat /sys/fs/cgroup/system.slice/docker-<id>.scope/memory.max
67108864
$ cat /sys/fs/cgroup/system.slice/docker-<id>.scope/cpu.max
50000 100000
```

File	Means
`memory.max 67108864`	64 MiB of memory at most
`cpu.max 50000 100000`	50 ms of CPU time in every 100 ms: half a CPU

Without `--memory` and `--cpus` a container has no limit and can take all of the machine's memory.

A limit being hit

A process that allocates memory without end, in a container limited to 64 MiB:

```
$ docker run --name oom --memory 64m alpine:3.22 sh -c "tail /dev/zero"
$ echo $?
137
$ docker inspect oom --format '{{.State.OOMKilled}} {{.State.ExitCode}}'
true 137
```

- The kernel's **OOM killer** (out of memory) stopped the process when the cgroup reached its limit
- Exit code **137** = 128 + 9: the process was ended by signal 9, `SIGKILL`
- `OOMKilled true` is how you tell it apart from a crash

The rest of the machine was not affected. Without the limit, the same command would have used the host's memory until the host's OOM killer picked a process to end, possibly another one.

Root inside is root on the kernel

```
$ docker run --rm alpine:3.22 id
uid=0(root) gid=0(root) groups=0(root),…
$ docker run --rm --user 65532:65532 alpine:3.22 id
uid=65532 gid=65532 groups=65532
```

- A container runs as root unless told otherwise
- The kernel knows users only by number, and Docker does not change the numbers: uid 0 in the container is uid 0 on the host
- Namespaces limit what the process sees. If it gets out, through a kernel bug or a mistake such as mounting the host's `/` with `-v /:/host`, root in the container is root on the host
- An ordinary user that gets out is only an ordinary user

To run as an ordinary user: `--user 65532:65532` on `docker run`, or `USER 65532` in the Dockerfile. Any number other than 0 works; 65532 is the one distroless images use. Files the program writes must be writable by that user.

Image

An **image** is what a container starts from:

- **layers**: a stack of read-only filesystem changes. Each layer adds, changes or deletes files relative to the layers below it
- **config**: the metadata: which program to start, environment variables, working directory, user, exposed ports

Starting a container stacks the layers into one filesystem, adds one writable layer on top for that container, and starts the configured program in it.

Layers are shared. Ten images built `FROM alpine:3.22` store Alpine's layer once, and a machine that already has it downloads only the layers it lacks.

Pulling an image

```
$ docker pull nginx:1.29-alpine
aee4e54b3865: Pulling fs layer
…
aee4e54b3865: Pull complete
781ff50d2644: Pull complete
…
Digest: sha256:5616878291a2eed594aee8db4dade5878cf7edcb475e59193904b198d9b830de
docker.io/library/nginx:1.29-alpine

$ docker images nginx:1.29-alpine
IMAGE                ID             DISK USAGE   CONTENT SIZE
nginx:1.29-alpine    5616878291a2       93.5MB         26.9MB
```

- Each line with a hexadecimal name is one layer, downloaded in parallel
- **Content size** is what is transferred and stored compressed in the registry; **disk usage** is the unpacked size on this machine
- The **digest** is a hash of the image's contents: it identifies exactly this image

The container's writable layer

Changes a container makes go into its own writable layer, not into the image:

```
$ docker exec demo sh -c 'echo hello > /tmp/note.txt; rm /etc/motd'
$ docker diff demo
C /tmp
A /tmp/note.txt
C /etc
D /etc/motd
```

`A` added, `C` changed, `D` deleted. Remove the container and start a new one from the same image:

```
$ docker exec demo cat /tmp/note.txt
cat: can't open '/tmp/note.txt': No such file or directory
```

The writable layer is removed with the container. Anything that must outlive the container is written to a **volume**, outside the container's layers.

Image names

`ghcr.io/siimrebane/iti8801-notes:a4`, taken apart:

Part	Here	Means
registry	`ghcr.io`	the server that stores the image: GitHub Container Registry
namespace	`siimrebane`	the account or organisation on that registry
repository	`iti8801-notes`	one image's name; its versions are stored together
tag	`a4`	one version, by a name someone chose

Short names are filled in: `alpine:3.22` is `docker.io/library/alpine:3.22`: registry Docker Hub, namespace `library` for official images. Without a tag, the tag is `latest`.

Tags move, digests do not

A **tag** is a name that points at an image. Pushing a new image with the same tag moves the name to the new image; the old one is still there under its digest.

A **digest** is the SHA-256 hash of the image's manifest, the file that lists its layers and config. Change any byte of any layer and the digest changes.

```
docker pull alpine:3.22
docker pull alpine@sha256:<digest>
```

- `alpine:3.22` today and `alpine:3.22` in a month can be different images: 3.22.6 and a later 3.22.x
- `alpine@sha256:…` is the same image every time, or an error if it no longer exists
- `latest` has no special meaning to Docker: it is the tag used when none is given, and points wherever it was last pushed

The Dockerfile

A **Dockerfile** is a text file with the instructions to build an image. Each instruction runs on top of the result of the previous one; most of them add a layer.

Instruction	Does
`FROM image`	start from this image's layers and config
`WORKDIR /path`	set the directory the next instructions and the container run in
`COPY src dest`	copy files from the build context into the image: a layer
`RUN command`	run a command in the image being built; its file changes become a layer
`ENV KEY=value`	set an environment variable in the image's config
`ARG NAME`	a variable that exists only during the build
`EXPOSE 80`	record in the config that the program listens on 80; opens nothing
`USER 65532`	the user the next `RUN` lines and the container run as
`ENTRYPOINT`, `CMD`	the program the container starts, and its default arguments

A first Dockerfile for Notes

Everything in one image: the Go image, Node installed on top, then both builds.

```
FROM golang:1.25
RUN apt-get update && apt-get install -y nodejs npm
WORKDIR /src
COPY . .
RUN cd frontend && npm ci && npm run build
RUN go build -o /notes .
EXPOSE 80 8080
CMD ["/notes"]
```

Built from the Notes source directory: `main.go`, `go.mod`, `go.sum`, and `frontend/` with the page's sources.

The first Dockerfile, line by line

Line	Does
`FROM golang:1.25`	Debian 13 with Go 1.25 installed: 1.26 GB
`RUN apt-get … nodejs npm`	Debian's Node.js and npm, for building the page
`COPY . .`	the whole source directory into `/src`
`RUN cd frontend && npm ci && npm run build`	install the page's packages exactly as in `package-lock.json`, build it into `/src/dist`
`RUN go build -o /notes .`	compile the Go code, with the built page embedded, into `/notes`
`CMD ["/notes"]`	start the binary

Building it

```
docker build -f Dockerfile.naive -t notes:naive .
```

Part	Means
`docker build`	build an image
`-f Dockerfile.naive`	the Dockerfile to use; without `-f`, the file named `Dockerfile`
`-t notes:naive`	the name and tag to give the result
`.`	the **build context**: the directory whose files `COPY` can see

The builder can read only files in the context. A `.dockerignore` file in the context lists files the builder never gets. The Notes repository's lists `node_modules`, `frontend/node_modules`, `dist` and `.git`: the first three are rebuilt inside the image, and `.git` is the repository's history, which the image does not use.

What the build printed

With BuildKit, the builder in current Docker, `--progress=plain` shows each step and its time. On a VM with 1 vCPU and 2 GB of memory:

```
#6 [2/6] RUN apt-get update && apt-get install -y nodejs npm
#6 DONE 111.5s
#7 [3/6] WORKDIR /src
#7 DONE 0.6s
#8 [4/6] COPY . .
#8 DONE 0.1s
#9 [5/6] RUN cd frontend && npm ci && npm run build
#9 DONE 8.7s
#10 [6/6] RUN go build -o /notes .
#10 DONE 51.9s
real    4m39s
```

The result:

```
IMAGE          DISK USAGE   CONTENT SIZE
notes:naive       2.54GB          595MB
```

Where the 2.54 GB is

`docker history` lists an image's layers, newest first, with the instruction that made each one:

```
$ docker history notes:naive
CREATED BY                                      SIZE
CMD ["/notes"]                                  0B
RUN /bin/sh -c go build -o /notes .             206MB
RUN /bin/sh -c cd frontend && npm ci && npm …   52.4MB
COPY . .                                        9MB
RUN /bin/sh -c apt-get update && apt-get ins…   734MB
COPY /target/ /                                 255MB
RUN /bin/sh -c set -eux;  apt-get update;  a…   287MB
RUN /bin/sh -c set -eux;  apt-get update;  a…   202MB
RUN /bin/sh -c set -eux;  apt-get update;  a…   65MB
# debian.sh --arch 'amd64' out/ 'trixie' '@1…   134MB
```

- The bottom four lines and `COPY /target/` are the `golang:1.25` image: Debian plus the Go toolchain
- 734 MB is Node.js and npm from Debian
- 206 MB is the Go build: the binary plus Go's build cache, left in the layer; only the 11 MB binary is needed to run

The build cache

BuildKit keeps every layer it builds. Before running an instruction it checks whether it already has a layer for:

- the same previous layer,
- the same instruction,
- for `COPY`, the same contents of the copied files.

If yes, it reuses the layer and prints `CACHED`. If no, it runs the instruction, **and every instruction after it**, because each later layer sits on top of one that changed.

So the order of the Dockerfile decides how much is rebuilt. Things that change rarely go first, things that change on every commit go last.

One changed line, in the first Dockerfile

One comment added to `main.go`, then the same build again:

```
#6 [2/6] RUN apt-get update && apt-get install -y nodejs npm
#6 CACHED
#7 [3/6] WORKDIR /src
#7 CACHED
#8 [4/6] COPY . .
#8 DONE 0.2s
#9 [5/6] RUN cd frontend && npm ci && npm run build
#9 DONE 7.7s
#10 [6/6] RUN go build -o /notes .
#10 DONE 51.2s
```

78 s in total. `COPY . .` copies every file, so a change to `main.go` changes that layer, and the page's `npm ci` and build run again although nothing in `frontend/` changed.

Ordered for the cache

```
FROM golang:1.25
RUN apt-get update && apt-get install -y nodejs npm
WORKDIR /src
COPY frontend/package.json frontend/package-lock.json frontend/
RUN cd frontend && npm ci
COPY frontend/ frontend/
RUN cd frontend && npm run build
COPY go.mod go.sum ./
RUN go mod download
COPY main.go ./
RUN go build -o /notes .
CMD ["/notes"]
```

- The package lists first, then install: `npm ci` reruns only when `package-lock.json` changes
- The page's sources next, then its build: reruns only when a file in `frontend/` changes
- `go.mod` and `go.sum`, then `go mod download`: reruns only when dependencies change
- `main.go` last

The same one-line change, ordered

```
#10 [ 8/11] COPY go.mod go.sum ./
#10 CACHED
#13 [ 9/11] RUN go mod download
#13 CACHED
#14 [10/11] COPY main.go ./
#14 DONE 0.2s
#15 [11/11] RUN go build -o /notes .
#15 DONE 49.2s
```

Everything up to `COPY main.go` came from the cache: 60 s instead of 78 s.

`go build` still took 49 s. Go keeps compiled packages in a build cache in `/root/.cache/go-build`, but that directory was inside the previous `go build` layer, which was thrown away when `main.go` changed. The compiler started from nothing again.

A cache that survives: cache mounts

```
RUN --mount=type=cache,target=/root/.cache/go-build go build -o /notes .
```

`--mount=type=cache` gives the `RUN` a directory that BuildKit keeps between builds and never puts into a layer. The Go compiler's cache lives there.

```
cachemount first build:   51.0 s
cachemount rebuild:        5.6 s
#15 [11/11] RUN --mount=type=cache,target=/root/.cache/go-build go build -o /notes .
#15 DONE 3.0s
```

The same one-line change: 5.6 s instead of 60 s. The cache is on the machine that builds; a fresh machine, such as a CI runner's new container, starts with it empty.

The problem with one stage

The image that builds the application and the image that runs it need different things:

	Building needs	Running needs
Go	the compiler, the modules	nothing: the binary is compiled
the page	Node.js, npm, 84 npm packages	nothing: the built files are inside the binary
operating system	a package manager, a shell, a C library	nothing, if the binary is static

Everything in the build image is also in the shipped image: 2.54 GB to download on every machine that runs it, and every one of those programs is something that can have a vulnerability.

Multi-stage builds

A Dockerfile can have several `FROM` lines. Each one starts a new **stage** with its own layers. A later stage can copy files out of an earlier one; only the last stage becomes the image.

```
ARG BASE=scratch

FROM node:22-alpine AS frontend
WORKDIR /src/frontend
COPY frontend/package.json frontend/package-lock.json ./
RUN npm ci
COPY frontend/ ./
RUN npm run build

FROM golang:1.25-alpine AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY main.go ./
COPY --from=frontend /src/dist ./dist
RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o /notes .

FROM ${BASE}
COPY --from=build /notes /notes
EXPOSE 80 8080
ENTRYPOINT ["/notes"]
```

The multi-stage file, line by line

Line	Does
`FROM node:22-alpine AS frontend`	a stage named `frontend` for the page, on an image that has Node
`FROM golang:1.25-alpine AS build`	a second stage for the binary
`COPY --from=frontend /src/dist ./dist`	copy the built page out of the first stage
`CGO_ENABLED=0`	no C code: a **static** binary that needs no C library at run time
`-ldflags="-s -w"`	leave out the symbol table and debug information: a smaller binary
`ARG BASE=scratch` before the first `FROM`	a build argument that a `FROM` line can use
`FROM ${BASE}`	the final stage, on whatever base `--build-arg BASE=…` names
`COPY --from=build /notes /notes`	the only thing the final image gets from the build

The Node and Go stages stay in the builder's cache and are never shipped.

Four bases for the final stage

Base	What it is
`debian:trixie-slim`	Debian 13 with the package manager, a shell and the C library; nothing else
`alpine:3.22`	Alpine Linux: a small distribution with its own package manager and a shell
`gcr.io/distroless/static-debian12`	Google's "distroless": CA certificates, time zone data, `/etc/passwd`; no shell, no package manager
`scratch`	Docker's name for an empty image: no files at all

`scratch` works here because the binary is static and needs nothing from the filesystem.

The four, built and measured

```
docker build --build-arg BASE=<base> -f Dockerfile.multi -t notes:<name> .
```

Final base	Disk usage	Content size
one stage, `golang:1.25`	2.54 GB	595 MB
`debian:trixie-slim`	133 MB	34 MB
`alpine:3.22`	27.9 MB	7.9 MB
`distroless/static-debian12`	21.4 MB	4.8 MB
`scratch`	15.2 MB	4.1 MB

The first multi-stage build took 65 s; the other three took 2 to 3 s each, because only the last stage differed and the builder had the other two cached.

What is in the scratch image

```
$ docker history notes:scratch
CREATED BY                      SIZE
ENTRYPOINT ["/notes"]           0B
EXPOSE [80/tcp 8080/tcp]        0B
COPY /notes /notes              11.1MB
```

One file. What that costs:

```
$ docker exec -it web sh
OCI runtime exec failed: exec failed: unable to start container process:
exec: "sh": executable file not found in $PATH
```

Missing	Consequence
a shell and tools	no `docker exec … sh`, no `ls`, no `curl` inside; debugging happens from outside
CA certificates	the program cannot verify any TLS certificate, unless it carries its own
time zone data	only UTC, unless the program embeds the data
`/etc/passwd`	`USER` works with numbers only

Running a container

```
docker run -d --name web -p 8090:80 -e MODE=frontend -e API_URL=http://127.0.0.1:9 notes:scratch
```

Part	Means
`-d`	detached: start it in the background and print its id
`--name web`	a name for the other commands; without it Docker invents one
`-p 8090:80`	port 8090 on the host leads to port 80 in the container
`-e MODE=frontend`	an environment variable for the process
`notes:scratch`	the image
anything after the image	arguments for the program, replacing the image's `CMD`

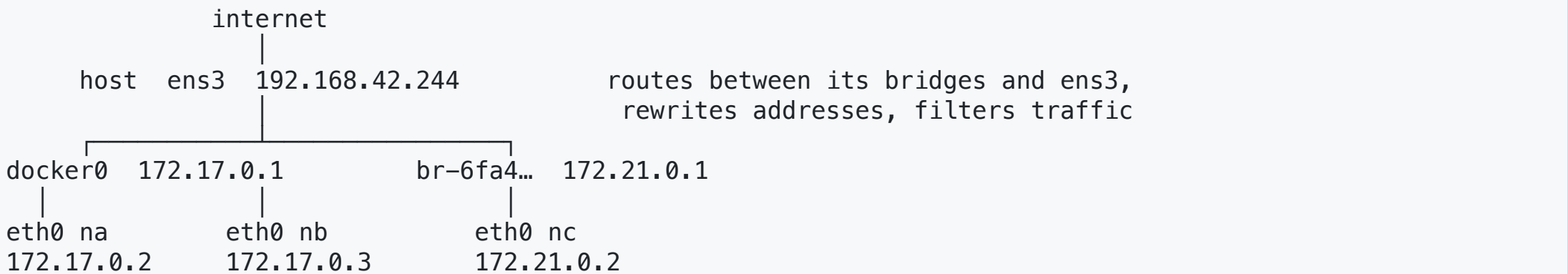
```
$ curl -s localhost:8090/version
{"commit":"local","hostname":"d4080744e4da","mode":"frontend","version":"dev"}
```

The hostname the application reports is the container's id: its own `uts` namespace.

What a Docker network is

A Docker **network** exists on one host and is made of three things:

- a **bridge**: a virtual switch inside the host's kernel. The default one is called `docker0`
- a **subnet** for it: `docker0` has `172.17.0.0/16`; the host itself takes the first address, `172.17.0.1`
- per container, a **veth pair**: two virtual network cards joined like the two ends of a cable. One end is the container's `eth0`, inside its network namespace; the other end is plugged into the bridge



The host is the router between its bridges and the outside, one subnet per bridge.

A network, seen from the host and from a container

```
$ ip -4 addr show docker0                       # on the host
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0

$ docker run -d --name na alpine:3.22 sleep 3600
$ docker exec na ip -4 addr show eth0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
$ docker exec na ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2

$ ip link show master docker0                   # on the host: what is plugged into docker0
299: veth0104b74@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> … master docker0 state UP
```

- the container has an address in the bridge's subnet, and its **default route** is the host's address on the bridge: everything not in `172.17.0.0/16` goes to the host
- `veth0104b74` is the host's end of the pair; the container's `eth0` is the other end

The default network

A container started without `--network` joins the network named `bridge`, which is `docker0`. Two containers on it, `na` at `172.17.0.2` and `nb` at `172.17.0.3`:

```
$ docker exec na ping -c 2 -W 2 172.17.0.3
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.113 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.135 ms
$ docker exec na ping -c 1 -W 2 nb
ping: bad address 'nb'
$ docker exec na wget -q -O /dev/null -T 5 http://example.com ; echo $?
0
```

- containers on the same network reach each other by address
- on the default network, not by name
- every container reaches the internet, through the host
- nothing outside the host reaches a container, except through a published port
- addresses are handed out at start and can change on every restart

Port mapping

The container has its own network namespace, so its port 80 is not the host's port 80. `-p host:container` connects them.

```
$ docker ps
NAMES   IMAGE           PORTS                                                  STATUS
web     notes:scratch   8080/tcp, 0.0.0.0:8090->80/tcp, [::]:8090->80/tcp   Up 1 second

$ sudo ss -tlnp | grep 8090
LISTEN 0  4096   0.0.0.0:8090   0.0.0.0:*   users:(("docker-proxy",pid=175914,fd=7))
```

- `0.0.0.0:8090->80/tcp`: every address of the host, port 8090, forwards to the container's 80
- `8080/tcp` without an arrow: the image says `EXPOSE 8080`, and nothing is mapped to it
- on the host, port 8090 is held by `docker-proxy` and by firewall rules Docker writes; the application itself only listens inside
- `-p 127.0.0.1:8090:80` maps it on the loopback address only: reachable from the host, not from other machines

The rules Docker writes: addresses

The Linux kernel has a packet filter that can drop, accept and rewrite packets by rules. `iptables` is the command that lists and changes those rules; Docker adds rules for every network and every published port. `-t nat` shows the rules that rewrite addresses:

```
$ sudo iptables -t nat -S
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 8097 -j DNAT --to-destination 172.17.0.2:80
```

Part	Means
`-s 172.17.0.0/16 ! -o docker0`	from the default network's subnet, leaving through any interface other than `docker0`
`-j MASQUERADE`	replace the source address with the host's own: **NAT**, so answers from the internet come back to the host, which passes them to the container
`! -i docker0 -p tcp --dport 8097`	TCP to port 8097, arriving from anywhere but the network itself
`-j DNAT --to-destination 172.17.0.2:80`	replace the destination with the container's address and port: this is `-p 8097:80`

The rules Docker writes: what may pass

The same host, the filter rules for the default network:

```
$ sudo iptables -S
-P FORWARD DROP
-A DOCKER-CT -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A DOCKER-FORWARD -i docker0 -j ACCEPT
-A DOCKER -d 172.17.0.2/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 80 -j ACCEPT
-A DOCKER ! -i docker0 -o docker0 -j DROP
```

Rule	Means
`-P FORWARD DROP`	the host passes on nothing that no rule accepts
`-o docker0 … RELATED,ESTABLISHED -j ACCEPT`	answers to connections a container opened may come in
`-i docker0 -j ACCEPT`	anything **leaving** the network is passed on
`-d 172.17.0.2 … --dport 80 -j ACCEPT`	**into** the network: the published port
`! -i docker0 -o docker0 -j DROP`	anything else into the network is dropped

Every network gets the same set with its own bridge name: out of a network, everything; into it, answers and published ports only.

Networks you create: names instead of addresses

```
docker network create l5
docker run -d --name db  --network l5 -e POSTGRES_PASSWORD=pw -e POSTGRES_DB=notes postgres:18
docker run -d --name api --network l5 -e MODE=api \
    -e 'DATABASE_URL=postgres://postgres:pw@db:5432/notes?sslmode=disable' notes:scratch
docker run -d --name web --network l5 -p 8090:80 -e MODE=frontend -e API_URL=http://api:8080 notes:scratch
```

On a network created with `docker network create`, Docker answers DNS queries for container names: `db` and `api` in the URLs are names, not hosts in `/etc/hosts`.

```
$ docker exec db getent hosts api
172.19.0.3      api
```

Only `web` has `-p`: the database and the api are reachable from the other containers, not from outside.

The three containers working

```
$ curl -s -X POST localhost:8090/api/notes -H 'Content-Type: application/json' -d '{"text":"first note"}'
{"id":1,"text":"first note","created_at":"2026-09-25T21:22:07.148746Z"}

$ curl -s localhost:8090/api/notes
[{"id":1,"text":"first note","created_at":"2026-09-25T21:22:07.148746Z"}]

$ docker network inspect l5
api 172.19.0.3/16
web 172.19.0.4/16
db 172.19.0.2/16
```

The same image runs as the frontend and as the api; `MODE` decides. Configuration comes from the environment at `docker run`, not from the image: one image, built once, runs in every environment with different variables.

Two networks: no way across

Two networks created with `docker network create` are separate. A container on one cannot reach a container on the other, neither by name nor by address: Docker's rules drop traffic that enters a network from another bridge.

Each network you create gets its own subnet: `172.18.0.0/16`, `172.19.0.0/16`, and so on.

A container can be on more than one network. `docker network connect <network> <container>` gives it a second network card on the other bridge, and it reaches the containers on both.

Docker networks and a VPC

	VPC / VNet (weeks 2–4)	Docker, on one host
a subnet	a subnet	a network: one bridge, one range
the router	the cloud's, built in	the host
between subnets	routed by default; security groups or NSGs restrict	dropped; a container on both networks is the only way across
from the outside in	a public IP or load balancer, plus an inbound rule	a published port, `-p`: the host forwards it to the container and lets it in
from the inside out	NAT gateway or internet gateway	the host, with its own address as the source: NAT
inbound default	denied	denied, except published ports
outbound default	allowed	allowed; a network created with `--internal` has no way out

`--network host`

`--network host` does not create a network namespace: the container uses the host's.

```
$ docker run --rm --network host alpine:3.22 ip -4 addr | grep inet
    inet 127.0.0.1/8 scope host lo
    inet 192.168.42.244/24 brd 192.168.42.255 scope global dynamic ens3
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
```

- the container sees the host's interfaces and binds directly to the host's ports; `-p` has no effect
- the boot scripts in weeks 2 to 4 started the course images this way (`docker run -d --name beacon --restart=always --network host …`), so that their reports named the VM's address in its subnet, not a 172.17 address that only exists inside the VM
- two containers on host networking cannot both listen on the same port

Published ports and the VM's own firewall

Traffic to a published port is passed on to a container, so it goes through the rules Docker writes, not through the rules for traffic to the host itself. Firewall tools on the host, such as Ubuntu's `ufw`, write theirs for traffic to the host.

- a port published with `-p` is open even where `ufw` on the host denies it
- on a cloud VM, the NSG or security group in front of the VM decides what reaches it from the internet; the rules on the VM come after it
- a port that must not be reachable from other machines is published on the loopback address only: `-p 127.0.0.1:8090:80`

Volumes

A **volume** is storage that Docker keeps outside every container's layers and mounts into a container at a path. Removing the container leaves the volume.

```
docker run -d --name db --network l5 -e POSTGRES_PASSWORD=pw -e POSTGRES_DB=notes \
    -v l5data:/var/lib/postgresql postgres:18
```

`-v l5data:/var/lib/postgresql`: the volume named `l5data`, created if missing, mounted where Postgres 18 keeps its data. Remove the container, start a new one with the same `-v`:

```
$ docker exec db psql -U postgres -d notes -c "select count(*) from notes"
 count
-------
     1
```

The same with a new container and no `-v`:

```
ERROR:  relation "notes" does not exist
```

Anonymous volumes

The `postgres` image's config declares its data directory a volume. Started without `-v`, Docker creates a volume with a random name and does not remove it with the container:

```
$ docker run -d --name pgtmp -e POSTGRES_PASSWORD=pw postgres:18
$ docker rm -f pgtmp
$ docker volume ls
DRIVER    VOLUME NAME
local     0bab34ec28d89b39ea38312b60e54ae00310ede7919b3c1dafdf2d2c34b97be2
```

- the data was not lost, and it is not attached to anything: a new container does not find it by itself
- `docker rm -v` removes a container together with its anonymous volumes
- `docker volume prune` removes volumes that no container uses

Name every volume that holds data you want to keep.

Bind mounts

A **bind mount** mounts a directory of the host into the container: `-v /home/ubuntu/config:/etc/app` or `-v "$PWD":/src`.

	Volume	Bind mount
`-v` form	`name:/path`	`/absolute/host/path:/path`
where the files are	a directory Docker manages	wherever the path says
who creates it	Docker, on first use	you; the path must exist
typical use	database data, anything the application writes	configuration files, source code while developing

A bind mount's files have the host's owners and permissions: a container running as uid 65532 cannot write into a directory owned by uid 1000.

Logs

A containerised program writes to standard output and standard error, not to log files. Docker collects both:

```
$ docker logs web
2026/09/25 21:21:23 iti8801-notes dev (local) mode=frontend id=5b332d964916 listening on :80
```

- `docker logs -f web` follows new lines as they come
- `docker logs --since 10m web` shows the last ten minutes
- the logs are kept with the container, and removed with it

Stopping: PID 1 and signals

`docker stop` sends the container's first process the signal `SIGTERM`, waits 10 seconds, then sends `SIGKILL`, which cannot be caught.

The first process in a PID namespace is **PID 1**, and the kernel treats it specially: a signal it has no handler for is **ignored**, not applied. An ordinary program that relies on the default "SIGTERM ends me" does not end as PID 1.

Container	`docker stop` took
`sleep 3600` as PID 1	10.46 s: `SIGTERM` ignored, ended by `SIGKILL`
the same with `docker run --init`	0.26 s
the Notes binary as PID 1	0.20 s

`--init` puts a small init program, `docker-init`, in as PID 1; it starts your program as PID 7 and passes signals on. The Notes binary stops at once because the Go runtime installs its own handler for `SIGTERM`.

Shell form and exec form

`CMD` and `ENTRYPOINT` have two forms:

	Written	Config stores	PID 1 is
exec form	`CMD ["sleep", "3600"]`	`["sleep","3600"]`	`sleep`
shell form	`CMD sleep 3600; echo done`	`["/bin/sh","-c","sleep 3600; echo done"]`	the shell

```
$ docker exec s4 ps          # CMD sleep 3600; echo done
PID   USER     COMMAND
    1 root     /bin/sh -c sleep 3600; echo done
    7 root     sleep 3600
```

In shell form the shell is PID 1 and the program is its child; a `SIGTERM` reaches the shell, not the program. The exec form starts the program directly. Use the exec form; images without a shell, such as `scratch`, have only the exec form.

`ENTRYPOINT` and `CMD` together

Image has	`docker run image` runs	`docker run image x y` runs
`CMD ["/notes"]`	`/notes`	`x y`
`ENTRYPOINT ["/notes"]`	`/notes`	`/notes x y`
`ENTRYPOINT ["/notes"]` and `CMD ["--help"]`	`/notes --help`	`/notes x y`

- `ENTRYPOINT` is the program; `CMD` is its default arguments, replaced by anything given after the image name
- `docker run --entrypoint sh image` replaces the entrypoint, for an image that has a shell

Restart policies

`--restart` tells Docker what to do when the container's process ends, and when Docker itself starts after a reboot:

Policy	Restarts the container
`no` (default)	never
`on-failure`	when it exits with a non-zero code
`always`	whenever it ends, and when Docker starts
`unless-stopped`	like `always`, except after a `docker stop`

On a server that reboots for updates, a container without a restart policy is not running afterwards.

Several containers, by hand

The Notes application as three containers took four commands:

```
docker network create l5
docker run -d --name db  --network l5 -e POSTGRES_PASSWORD=pw -e POSTGRES_DB=notes -v l5data:/var/lib/postgresql postgres:18
docker run -d --name api --network l5 -e MODE=api -e 'DATABASE_URL=postgres://postgres:pw@db:5432/notes?sslmode=disable' notes:scratch
docker run -d --name web --network l5 -p 8090:80 -e MODE=frontend -e API_URL=http://api:8080 notes:scratch
```

- they must run in this order: the network before the containers, the database before the api uses it
- the password appears twice and must match
- stopping and removing everything takes as many commands again, and nothing records which containers belong together

Docker Compose

Docker Compose runs an application made of several containers from one file, `compose.yaml`, that describes:

- **services**: one per kind of container, with its image, environment, ports, networks and volumes
- **networks** the services join
- **volumes** they use

`docker compose up -d` creates whatever is missing, in the right order, and starts it; `docker compose down` stops and removes it. The set of containers, networks and volumes from one file is a **project**, named after the file's directory unless `-p` says otherwise.

Compose is part of Docker: the `docker compose` command, not a separate server.

The three services as one file

```yaml
services:
  db:
    image: postgres:18
    environment:
      POSTGRES_DB: notes
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - pgdata:/var/lib/postgresql
    networks: [backend]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d notes"]
      interval: 5s
      retries: 10

  api:
    build: .
    image: notes:compose
    environment:
      MODE: api
      DATABASE_URL: postgres://postgres:${DB_PASSWORD}@db:5432/notes?sslmode=disable
      DB_TARGET: db:5432
    networks: [frontend, backend]
    depends_on:
      db:
        condition: service_healthy
```

The file, continued

```
  web:
    image: notes:compose
    environment:
      MODE: frontend
      API_URL: http://api:8080
      DB_TARGET: db:5432
    ports:
      - "8090:80"
    networks: [frontend]
    depends_on: [api]

networks:
  frontend:
  backend:

volumes:
  pgdata:
```

`DB_TARGET` makes each Notes container try to reach the database and report the result under `/report`.

The file, line by line

Line	Does
`services:` → `db:`, `api:`, `web:`	three services; the name is also the host name the others use
`image: postgres:18`	run this image
`build: .` with `image: notes:compose`	build the image from the `Dockerfile` in `.` and name it `notes:compose`; `web` uses the same image
`environment:`	the environment variables, as with `-e`
`${DB_PASSWORD}`	filled in by Compose before anything runs
`volumes: - pgdata:/var/lib/postgresql`	the named volume `pgdata`, declared at the bottom of the file
`networks: [backend]`	the networks this service joins; `api` joins both
`ports: - "8090:80"`	as `-p 8090:80`; only `web` has one
`healthcheck:`	a command Docker runs every 5 s inside the container; exit code 0 means healthy
`depends_on: db: condition: service_healthy`	start `api` only after `db` reports healthy

Values from `.env`

Compose reads a file named `.env` next to `compose.yaml` and fills every `${NAME}` in the file from it:

```
$ cat .env
DB_PASSWORD=compose-example-pw
```

`docker compose config` prints the file as Compose will use it, with every value filled in:

```
$ docker compose config
      DATABASE_URL: postgres://postgres:compose-example-pw@db:5432/notes?sslmode=disable
      POSTGRES_PASSWORD: compose-example-pw
```

The password is written once and used in two services. `compose.yaml` goes into git; `.env` does not, because it holds the password.

`docker compose up -d`

```
$ docker compose up -d
 Network compose_backend  Created
 Network compose_frontend  Created
 Volume compose_pgdata  Created
 Container compose-db-1  Created
 Container compose-api-1  Created
 Container compose-web-1  Created
 Container compose-db-1  Started
 Container compose-db-1  Waiting
 Container compose-db-1  Healthy
 Container compose-api-1  Started
 Container compose-web-1  Started
```

- the directory is `compose`, so that is the project name, and every name starts with it: `compose_backend`, `compose-db-1`
- networks and the volume first, then the containers in the order `depends_on` gives
- `api` started only after `db` was `Healthy`

Running

```
$ docker compose ps
SERVICE   IMAGE           STATUS                   PORTS
api       notes:compose   Up Less than a second    80/tcp, 8080/tcp
db        postgres:18     Up 6 seconds (healthy)   5432/tcp
web       notes:compose   Up Less than a second    8080/tcp, 0.0.0.0:8090->80/tcp

$ curl -s -X POST localhost:8090/api/notes -H 'Content-Type: application/json' -d '{"text":"from compose"}'
{"id":1,"text":"from compose","created_at":"2026-09-25T22:36:14.598357Z"}
```

Only `web` has a mapping to the host. The request went from the host to `web` on port 8090, from `web` to `api` by name on the `frontend` network, and from `api` to `db` on the `backend` network.

Two networks: who can reach whom

```
compose_frontend: compose-web-1 compose-api-1
compose_backend:  compose-api-1 compose-db-1
```

Each Notes container reports whether it could reach `db:5432`:

```
$ curl -s localhost:8090/report          # web
    "db_check": { "target": "db:5432", "ok": false,
                  "error": "resolve: lookup db on 127.0.0.11:53: server misbehaving" }

$ curl -s localhost:8090/api/report      # api, through web's /api forwarding
    "db_check": { "target": "db:5432", "ok": true, "addr": "172.19.0.2" },
    "db_query": true
```

`web` is not on `backend`, so for `web` the name `db` does not exist: Docker's DNS server, `127.0.0.11`, answers only for containers on the same networks. `api` is on both and reaches the database.

Start order and readiness

`depends_on: [db]` without a condition only orders the starts: `api` starts as soon as the `db` container has started, not when Postgres accepts connections. With that file, `api`'s log:

```
api-1  | 22:36:52 database not ready: … dial tcp 172.20.0.2:5432: connect: connection refused (retrying …)
api-1  | 22:36:52 iti8801-notes dev (local) mode=api id=9b4d701f0017 listening on :8080
api-1  | 22:36:57 database connected
```

The Notes api retries until it connects, 5 s later. A program that tries once and exits needs the other form:

```
    depends_on:
      db:
        condition: service_healthy
```

which waits for the `db` service's healthcheck to succeed. `pg_isready` exits 0 once Postgres accepts connections.

`down`, and `down -v`

```
$ docker compose down
 Network compose_frontend  Removed
 Network compose_backend  Removed
$ docker compose up -d
$ curl -s localhost:8090/api/notes
[{"id":1,"text":"from compose","created_at":"2026-09-25T22:36:14.598357Z"}]
```

`down` removes the containers and networks and leaves the volume: after `up` again, the note is there.

```
$ docker compose down -v
 Volume compose_pgdata  Removed
$ docker compose up -d
$ curl -s localhost:8090/api/notes
[]
```

`down -v` removes the volumes too: an empty database.

Compose commands

Command	Does
`docker compose up -d`	create what is missing and start everything in the background
`docker compose up -d --build`	rebuild the images of services with `build:` first
`docker compose ps`	the project's containers, with health and ports
`docker compose logs -f api`	one service's output, followed
`docker compose exec db psql -U postgres`	run a command in a service's container
`docker compose config`	the file with every value filled in
`docker compose pull`	download newer images for `image:` services
`docker compose down`	stop and remove containers and networks; volumes stay
`docker compose down -v`	the same, and remove the volumes

All of them work from the directory with `compose.yaml` and act on that project only.

Checking the document

A **linter** checks that the document is valid OpenAPI and follows common rules. Redocly's, run as a container on the file:

```
docker run --rm -v $PWD:/spec redocly/cli:latest lint /spec/notes-api.yaml
```

Before `security: []` and the `operationId` lines were in the file:

```
error   security-defined: 4
warning operation-operationId: 4
❌ Validation failed with 4 errors and 7 warnings.
```

After adding them:

```
warning no-server-example.com: 1
warning operation-4xx-response: 1
Woohoo! Your API description is valid. 🎉
You have 3 warnings.
```

Testing the running API against the document

Schemathesis reads an OpenAPI document, generates requests from its schemas, including unusual but valid ones, sends them to the running API and checks each answer against the document.

```
docker run --rm --network host -v $PWD:/spec schemathesis/schemathesis:stable \
    run /spec/notes-api.yaml --url http://localhost:8090
```

Part	Means
`--network host`	the container reaches the host's `localhost:8090`, where the Compose project's `web` listens
`-v $PWD:/spec`	the document, from the current directory
`run /spec/notes-api.yaml`	test every operation in it
`--url http://localhost:8090`	the API to test

The test run

Against the Notes application running under Compose, 28 seconds:

```
API Operations: Selected: 4/4  Tested: 4
Failures:
  ❌ Server error: 1
  ❌ Undocumented HTTP status code: 2
  ❌ Unsupported methods: 2
Test cases: 1251 generated, 4 found 5 unique failures
```

For each failure it prints the request that caused it, as a `curl` command that reproduces it.

What it found

A note text containing a zero byte, which is valid JSON and a valid string:

```
$ curl -s -X POST localhost:8090/api/notes -H 'Content-Type: application/json' -d '{"text":"a\u0000b"}'
{"error":"ERROR: invalid byte sequence for encoding \"UTF8\": 0x00 (SQLSTATE 22021)"}   HTTP 500
```

The api passes the text to Postgres, which refuses zero bytes; the document promises 201 or 400, and the api answers 500 with the database's error message.

An id that is a valid integer by the document:

```
$ curl -s localhost:8090/api/notes/12345678901234567890
{"error":"method not allowed"}   HTTP 405
```

The code accepts ids of at most 18 digits; this one has 19, falls through to "method not allowed", and gets 405 where the document says 404.

`TRACE` requests are answered 405 without an `Allow` header naming the methods that do work, which RFC 9110 requires.

Code and document disagree: which one is wrong

Each finding is a place where the document and the code say different things. Either can be changed:

Finding	Change the code	Or the document
zero byte → 500	refuse such text with 400 before it reaches the database	say the text may not contain zero bytes (`pattern`)
19-digit id → 405	answer 404 for any id that is not a note	`maximum: 99999999999999999` on `id`
`TRACE` → 405 without `Allow`	send `Allow: GET, POST` with every 405	nothing: the rule comes from HTTP itself

When the document is written first and this test runs on every change, a disagreement between code and document shows up as a failed test. The test needs only the document and the service's address, so it runs against a deployed service from any machine that can reach it.

Built on one machine, run on another

An image is built for one processor architecture: its binaries contain that processor's instructions. Built on an Apple-silicon Mac, it is `linux/arm64`; the Azure and AWS VMs in the assignments are `linux/amd64`.

```
$ docker image inspect archdemo:mac --format '{{.Os}}/{{.Architecture}}'
linux/arm64
$ docker run --rm archdemo:mac
WARNING: The requested image's platform (linux/arm64/v8) does not match the detected host platform (linux/amd64/v4)
exec /bin/uname: exec format error
$ uname -m
x86_64
```

- `exec format error` is the kernel refusing a binary for another processor
- `docker build --platform linux/amd64 …` builds for the VM's processor, from any machine
- images such as `alpine` and `golang` are **multi-platform**: one name with a variant per architecture, and `docker pull` picks the machine's

Registries

A **registry** stores images and serves them over HTTPS by name. `docker push` uploads, `docker pull` downloads.

Registry	Image names look like
Docker Hub	`docker.io/<account>/<repository>:<tag>`, or just `<account>/<repository>`
GitHub Container Registry	`ghcr.io/<account>/<repository>:<tag>`
Azure Container Registry	`<registry name>.azurecr.io/<repository>:<tag>`
Amazon ECR	`<account id>.dkr.ecr.<region>.amazonaws.com/<repository>:<tag>`

A repository is **public** (anyone may pull) or **private** (only those with credentials). All four require credentials to push; `docker login <registry>` stores them.

Pushing

The image's name decides where it goes. `docker tag` gives an existing image a second name; `docker push` sends it:

```
$ docker tag notes:scratch localhost:5000/notes:l5
$ docker push localhost:5000/notes:l5
The push refers to repository [localhost:5000/notes]
a59d0cfb1dad: Pushed
3a0ad4cdd318: Pushed
l5: digest: sha256:5261304b4d120e0a4b6e36457dc73607e90cedce4dac7dfc5e0303b9527aba80 size: 855
```

Pushed again, nothing changed:

```
a59d0cfb1dad: Already exists
3a0ad4cdd318: Layer already exists
l5: digest: sha256:5261304b4d120e0a4b6e36457dc73607e90cedce4dac7dfc5e0303b9527aba80 size: 855
```

Layers the registry has are not sent again. The same content gives the same digest.

Pulling a private image

A VM that pulls a private image needs credentials too. Without them the registry refuses:

```
$ docker pull ghcr.io/siimrebane/does-not-exist:1
Error response from daemon: error from registry: denied
```

The registry answers the same way for an image that does not exist and for one you may not see; it does not tell outsiders which private images exist.

Ways a VM gets pull rights:

- `docker login` in its boot script: the password or token is then on the VM, readable by anyone who can read the boot script
- the cloud's identity: a VM's managed identity (Azure) or instance role (AWS) given pull rights on the registry; no secret on the VM
- a public repository: no credentials at all

Vulnerability scanning

A **CVE** (Common Vulnerabilities and Exposures) entry is a published, numbered description of a security flaw in a specific piece of software and the versions it affects: `CVE-2026-56852`.

A **scanner** lists what software is in an image and looks each version up in databases of CVEs:

- operating-system packages, from the package manager's own records in the image (`dpkg` on Debian, `apk` on Alpine)
- language packages, from lock files and from build information inside binaries: Go binaries record every module and version they were built from

Each finding has a **severity**, from the CVE's rating: `LOW`, `MEDIUM`, `HIGH`, `CRITICAL`, or `UNKNOWN` when not yet rated, and a **fixed version** if one exists.

Trivy

Trivy is an open-source scanner. It runs as a container itself:

```
docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v trivycache:/root/.cache \
    aquasec/trivy:latest image --severity HIGH,CRITICAL notes:scratch
```

Part	Means
`-v /var/run/docker.sock:/var/run/docker.sock`	lets Trivy ask the host's Docker for the image; a bind mount of Docker's control socket
`-v trivycache:/root/.cache`	keeps the CVE database between runs; the first run downloads it
`aquasec/trivy:latest`	the scanner's image
`image notes:scratch`	scan this image
`--severity HIGH,CRITICAL`	list only these

Giving a container Docker's socket gives it control of Docker on that host: do it for tools you trust.

The five images, scanned

Image	Critical	High	All findings
one stage, `golang:1.25`	24	405	4,063
`debian:trixie-slim`	0	44	155
`alpine:3.22`	0	1	1
`distroless/static-debian12`	0	1	2
`scratch`	0	1	1

Counted with `trivy image --format json` for each image, on the same day.

Where the one-stage image's findings are

Found in	Critical	High
Debian packages	23	383
`esbuild` binary in `frontend/node_modules`	1	21
the `notes` binary	0	1

Every program in the image is scanned, including the build tools that the running application never uses.

The finding in the empty image

```
notes (gobinary)
================
Total: 1 (HIGH: 1, CRITICAL: 0)

┌──────────────────┬─────────────────┬──────────┬────────┬───────────────────┬───────────────┐
│     Library      │ Vulnerability   │ Severity │ Status │ Installed Version │ Fixed Version │
├──────────────────┼─────────────────┼──────────┼────────┼───────────────────┼───────────────┤
│ golang.org/x/text │ CVE-2026-56852 │ HIGH     │ fixed  │ v0.29.0           │ 0.39.0        │
└──────────────────┴─────────────────┴──────────┴────────┴───────────────────┴───────────────┘
```

`scratch` has no packages, but the application has dependencies. The fix is in the application, not in the image:

```
go get golang.org/x/text@v0.39.0 && go mod tidy
```

Rebuilt and scanned again:

```
┌────────┬──────────┬─────────────────┐
│ Target │   Type   │ Vulnerabilities │
├────────┼──────────┼─────────────────┤
│ notes  │ gobinary │        0        │
└────────┴──────────┴─────────────────┘
```

Findings with no fix

From the `debian:trixie-slim` image:

```
CVE-2026-76642 bsdutils 1:2.41.5-0+deb13u1 -> None HIGH util-linux: failed external mount helper still runs privil…
```

- **Fixed version: none**: the flaw is known and no version without it has been released
- rebuilding does not help until Debian publishes a fixed package
- the program is `mount`, from util-linux. The Notes application never runs it; the scanner reports it because it is in the image

The two ways to reduce this: an image without the package, which is what the smaller bases are; and rebuilding regularly so that fixes arrive when they are published. A scan result is true for the day it ran: the same image gets new findings as new CVEs are published.

Secrets in build arguments

A build argument is visible in the image's history:

```
FROM alpine:3.22
ARG API_TOKEN
RUN echo "token is set" && test -n "$API_TOKEN"
```

```
$ docker build --build-arg API_TOKEN=sk-example-123 -t leak .
$ docker history --no-trunc leak
RUN |1 API_TOKEN=sk-example-123 /bin/sh -c echo "token is set" && test -n "$API_TOKEN"
```

Every `RUN` after an `ARG` records the argument's value in the image's metadata. Anyone who can pull the image can read it.

A deleted file is still in the image

```
COPY credentials.txt /tmp/credentials.txt
RUN rm /tmp/credentials.txt
```

The filesystem a container gets does not have the file. `docker export` writes a container's filesystem as one archive; unpacked:

```
$ ls /tmp/leak/tmp/credentials.txt
ls: cannot access '/tmp/leak/tmp/credentials.txt': No such file or directory
```

The image does. `docker save leak` writes the image's layers as files; the `COPY` layer still holds it:

```
found in layer 0c8f0702b561:
db_password=hunter2-example
```

A layer never changes. `rm` in a later layer only records "hidden from here on"; the earlier layer is shipped with every pull.

Where secrets go instead

Need	Way
a secret while building, such as a token for a private package server	`RUN --mount=type=secret,id=token …` with `docker build --secret id=token,src=token.txt`: the file exists during that `RUN` only and is in no layer
a secret while running, such as a database password	an environment variable or a file given at `docker run`, from the platform's secret store; never in the image
a value baked into a binary	anyone with the image can read it: copy the binary out and search it

Images are copied, cached and pushed to places you do not control. Treat everything in an image as readable by anyone who can pull it.

Running as a non-root user

```
FROM gcr.io/distroless/static-debian12
COPY --from=build /notes /notes
USER 65532:65532
ENTRYPOINT ["/notes"]
```

```
$ docker image inspect notes:scratch --format 'User={{.Config.User}}'
User=
$ docker image inspect notes:static-debian12 --format 'User={{.Config.User}}'
User=0
```

Both images built so far run as root: empty and `0` both mean uid 0. `USER` in the final stage changes that for every container started from the image.

A non-root process and port 80

Ports below 1024 are normally reserved for root. Inside a container's own network namespace Docker lowers that limit to 0, so the Notes frontend runs as uid 65532 and still listens on 80:

```
$ docker run --rm --user 65532 alpine:3.22 cat /proc/sys/net/ipv4/ip_unprivileged_port_start
0
```

With `--network host` the host's limit applies, usually 1024.

The commands for images

Command	Does
`docker build -t name:tag .`	build an image from the Dockerfile in `.`
`docker images`	list local images, with disk usage and content size
`docker history image`	its layers and the instructions that made them
`docker image inspect image`	its config: entrypoint, environment, user, platform
`docker tag image registry/repo:tag`	give it a name in a registry
`docker push registry/repo:tag`, `docker pull` …	upload it; download it

The commands for containers

Command	Does
`docker run -d --name n -p h:c -e K=V -v vol:/path image`	start a container
`docker ps`, `docker ps -a`	running containers; all of them
`docker logs -f n`	its output
`docker exec -it n sh`	a shell inside it, if the image has one
`docker stop n`, `docker rm n`	stop it; remove it
`docker network create`, `docker volume ls`	networks; volumes



Questions