Introduction to

kubernetes

Contacts

Péter Megyesi

megyesi@leannet.eu

twitter.com/M3gy0

linkedin.com/in/M3gy0

Dávid Szabó

szabo@leannet.eu

twitter.com/szabvid

linkedin.com/in/szabvid

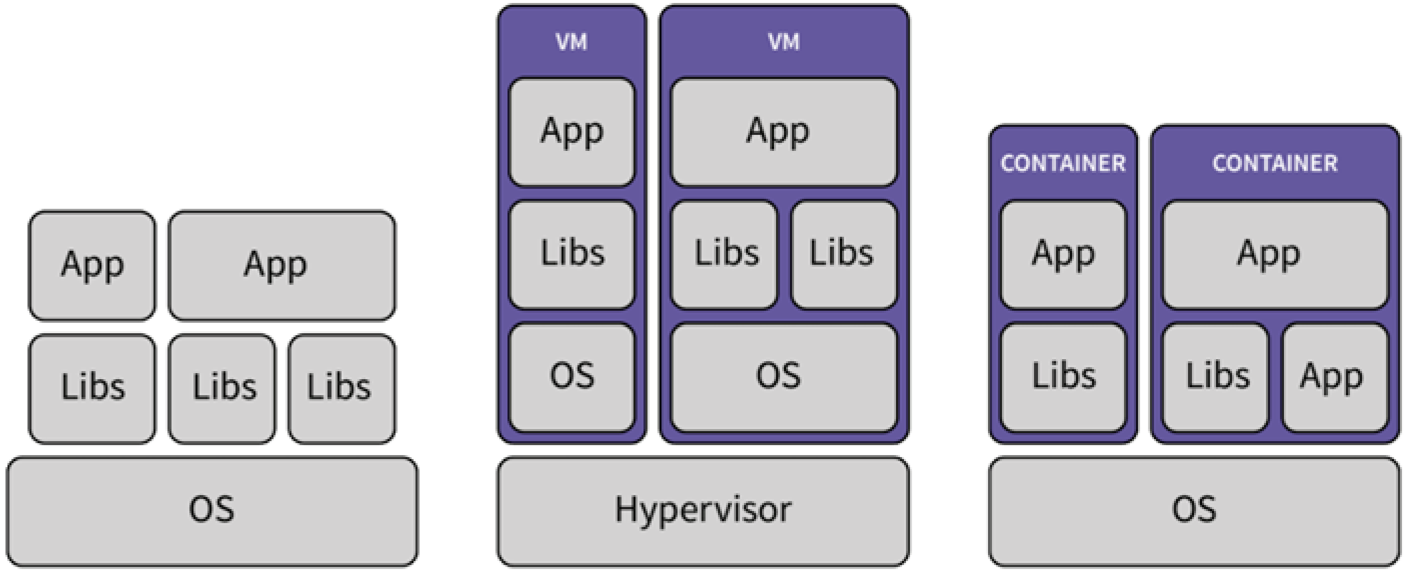
Course Outline

1. What is Kubernetes?
   - Components
   - Installation
2. **Basics of Docker**
   - **Namespaces**
   - **Building and running Docker images**
3. Pods and Deployments
   - Running basic workloads in Kubernetes
   - Scale, Update, Rollback
4. Advanced Pod configuration
   - Args, Envs, ConfigMaps, Secrets
   - Init- and sidecar containers
   - Scheduling and debugging
5. Networking in Kubernetes
   - What are network plugins?
   - Service abstraction and ingress
6. Persistent storage
   - Basics of storage: block vs. object vs. file system
   - StoragesClass, PVC, PV
7. Security
   - RBAC: Roles, ServiceAccounts, RoleBindings
   - Security context and network security policy
8. Advanced topics
   - Helm
   - Custom resources and operators

Quick Recap: What is a Container?

Containers are an application-centric way to deliver high-performing, scalable applications on the infrastructure of your choice

- A **bundle** of the **application** code along with its **runtime** and **dependencies**
- It creates an **immutable isolated executable** environment, also known as container image
- It can be **deployed** on the platform of your choice, such as desktops, servers, VMs or in the cloud



Bare Metal
- Deploy in months
- Live for years

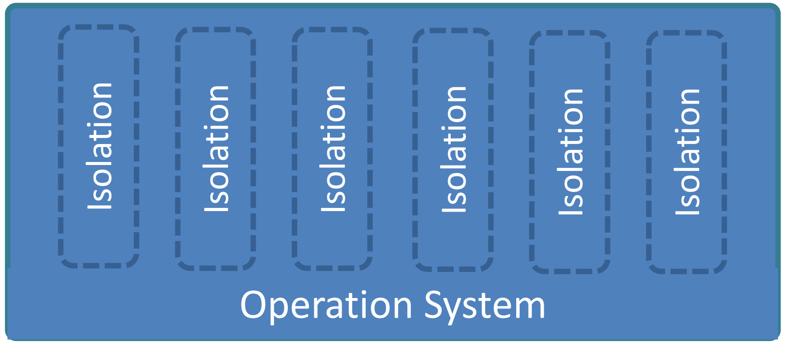
Virtualization
- Deploy in minutes/hours
- Live for weeks

Containers
- Deploy in seconds
- Live for hours/days

- Deploy in milliseconds
- Live for seconds

LXC LXD

Let's Go Deeper!

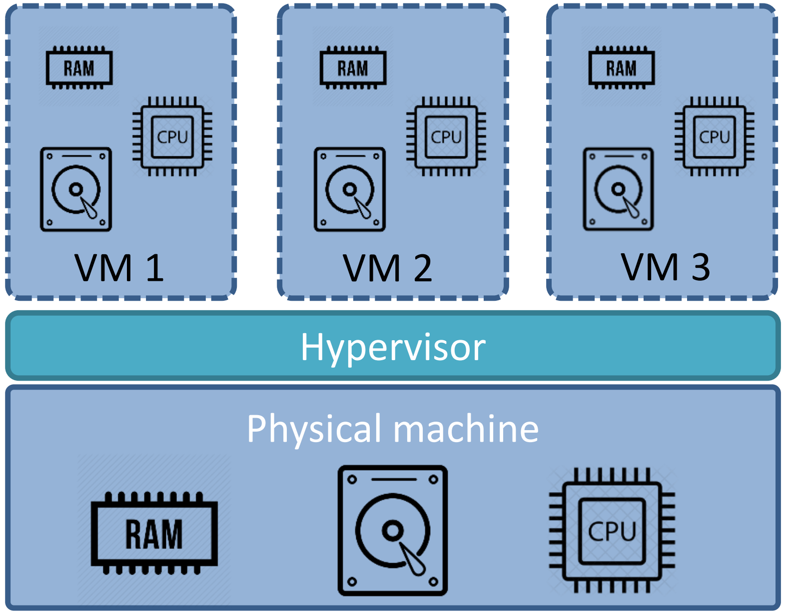


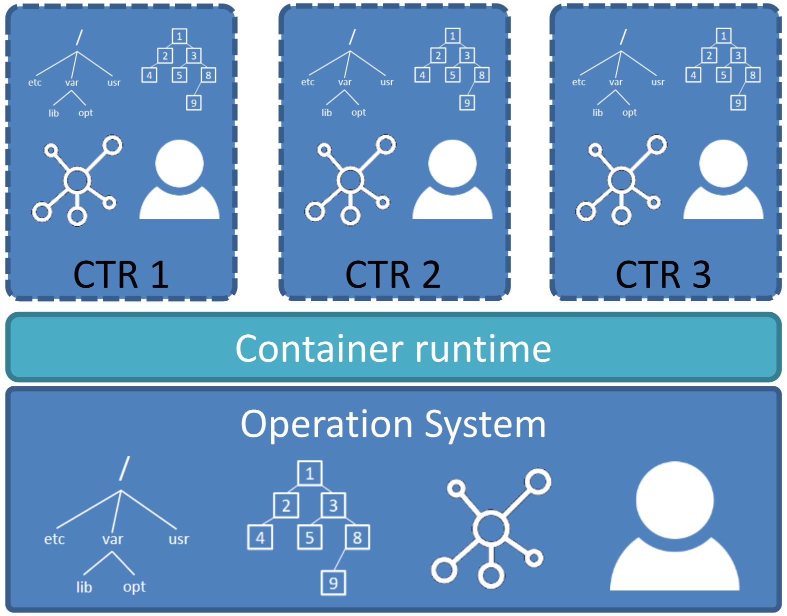
Containers are much older than the ones we know and use nowadays:

- They were created by using Linux kernel primitives:
  - Namespaces: isolation
  - Control Groups: setting the limits for resource usage

Namespaces:

- Makes a global system resource seems as it would belong to one process
- Similarly as the Hypervisor abstracts away hardware for VMs

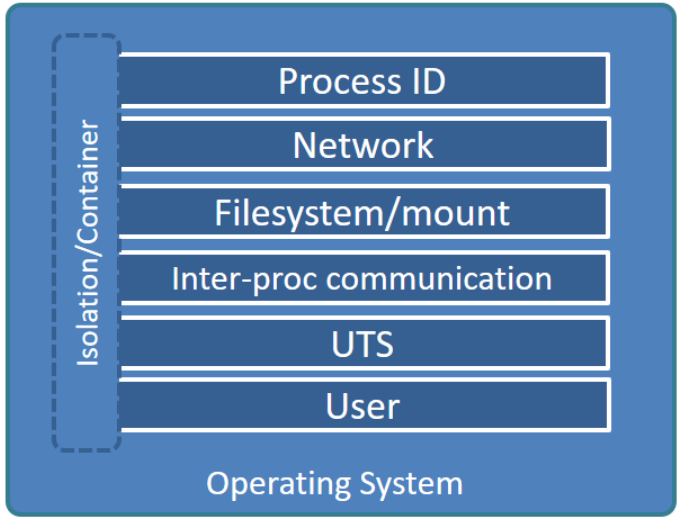




Let’s Go Deeper!

Namespaces:

- In order to create a container we need several global resources isolated:
  - Process ID:
    - inside the every container we have pid = 1
    - however, in the Host OS they can be seen as another pid (one process can have 1+ pid)
  - Network:
    - Allows processes to see entirely different networking interfaces
    - Even the loopback IF is different
    - In order to provide usable network interface in the child namespace:
      - need to set up additional “virtual” NIFs which span multiple NSs
      - need to configure bridging/routing
  - Filesystem/mount:
    - mount and unmount filesystems without it affecting the host filesystem



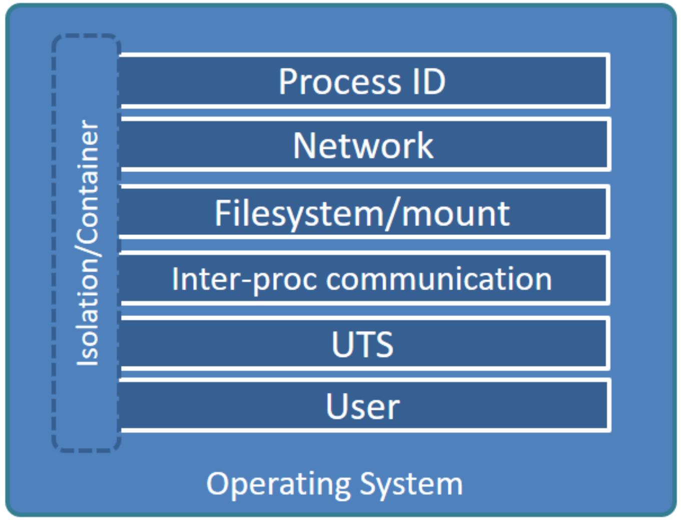
Let’s Go Deeper!

Namespaces:

- In order to create a container we need several global resources isolated:
  - Inter-process communication
    - prevents one process from an ipc namespace accessing the resources of another
  - UTS:
    - isolates the system's host and domain name
  - User:
    - allows a process to use unique user and group IDs within and outside a namespace:
      - a process can use privileged user and group IDs (zero) within a NS
      - and continue with non-zero user and group IDs outside the NS

Control groups:

- kernel mechanisms to restrict and measure resource allocations
- can allocate resources such as CPU time and memory
- prevents one container to starve others



Let's Go Deeper!

- So even before Docker creating containers were possible, but very struggling
- Docker made it very easy, also introducing Docker Layers:
  - read only union file system (UnionFS)
  - to be transparently overlaid, forming a single coherent file system

⇒ Namespaces

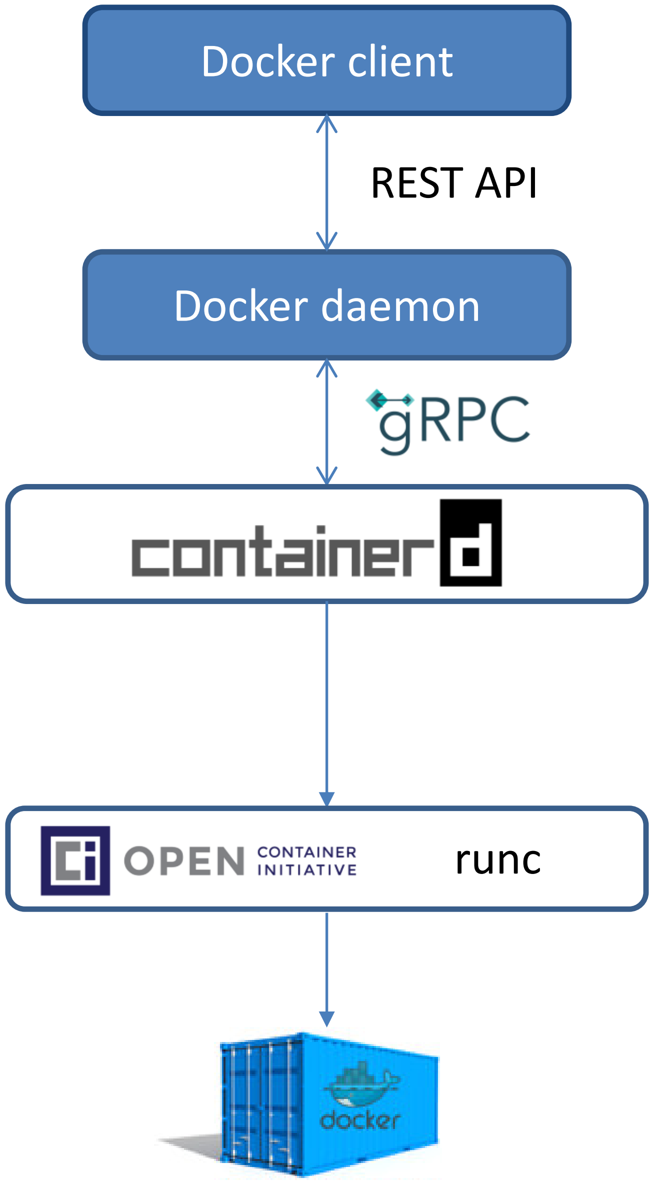
⇒ Control Groups

⇒ Layers

- Small history:
  - Docker originally wrapped the LXC (low level tools, templates, libs for creating containers)
  - However, at one point LXC development broke Docker so they developed their own substitute “libcontainer”
  - Docker became an ecosystem and added many features as a monolith: registry, orchestration, builds, etc.
    - Was bad as it is a monolith
    - Was bad as conflicted with other orchestrators (e.g. Kubernetes)
  - When OCI (Open Container Initiative) came they rewrite their architecture (abandoned libcontainer in 2015)

Docker Engine Architecture

- Docker is both a platform (ecosystem) and a container manager
- Let’s see the Docker Engine:
  - The monolith is gone
  - Based on open source solutions
  - Docker client:
    - Interface towards the user (here we use “docker run”, etc.)
    - Calls Docker daemon via REST API
  - Docker daemon:
    - Implementing the REST API and translate it to containerD via gRPC
  - Containerd:
    - Execution and lifecycle operations (start, stop, pause)
  - runc:
    - Default OCI implementation
    - OCI: image + runtime specification
    - After starting the container, exits and a shim process stays for the container



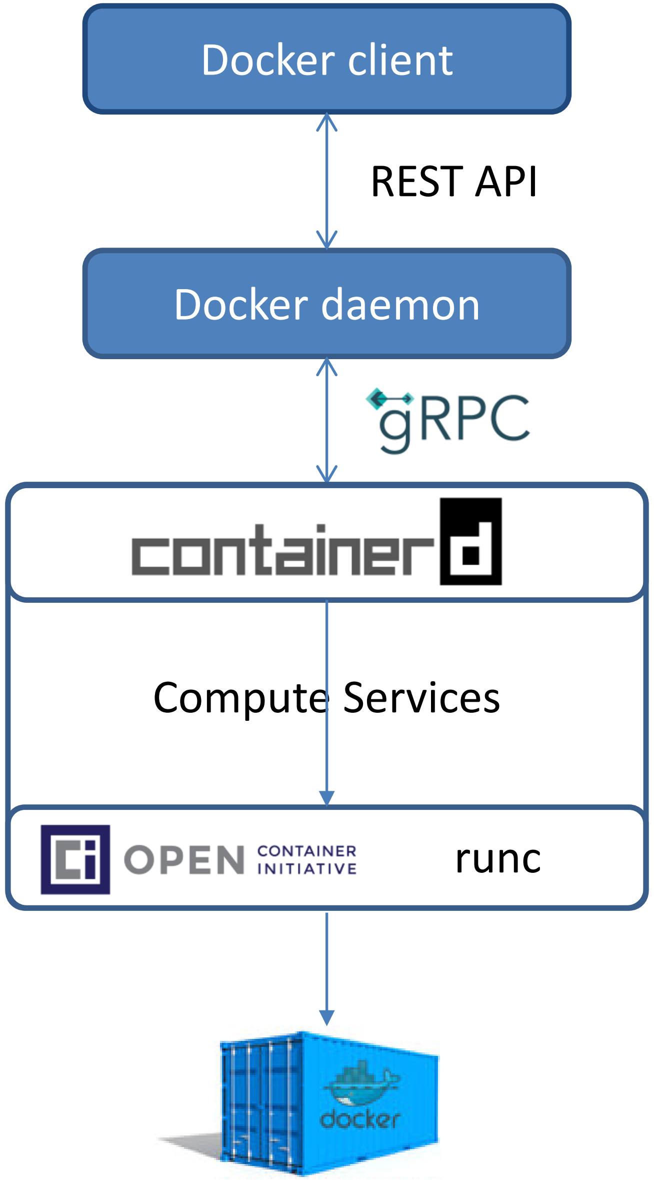
Docker Engine Architecture

The advantage of modularity:

- We can swap out runc without any problem
- We can even swap out containerd + runc and containers do not have to stop!

The difference when we use Docker on Windows:

  - Windows has it own way for lifecycle management



Docker Image vs. Container

Image:

- Read-only template that contains a set of instructions for creating a container that can run
- Can be built from a Dockerfile

Container:

- Running version of an image

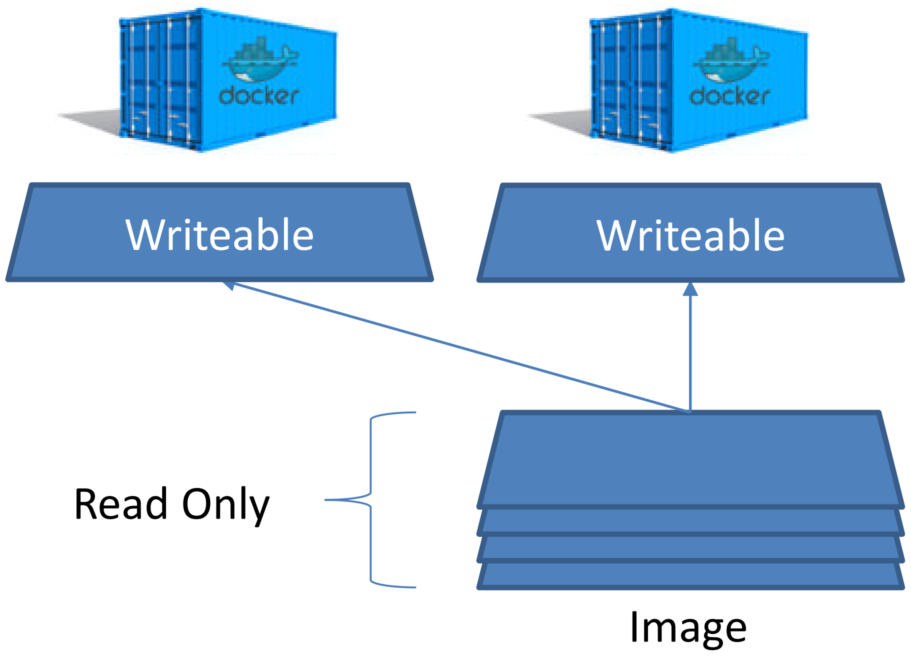
Similar concept as **object** and **class** in programming:

- Class is a blueprint or template from which the object is created
- Object is an instance of a class

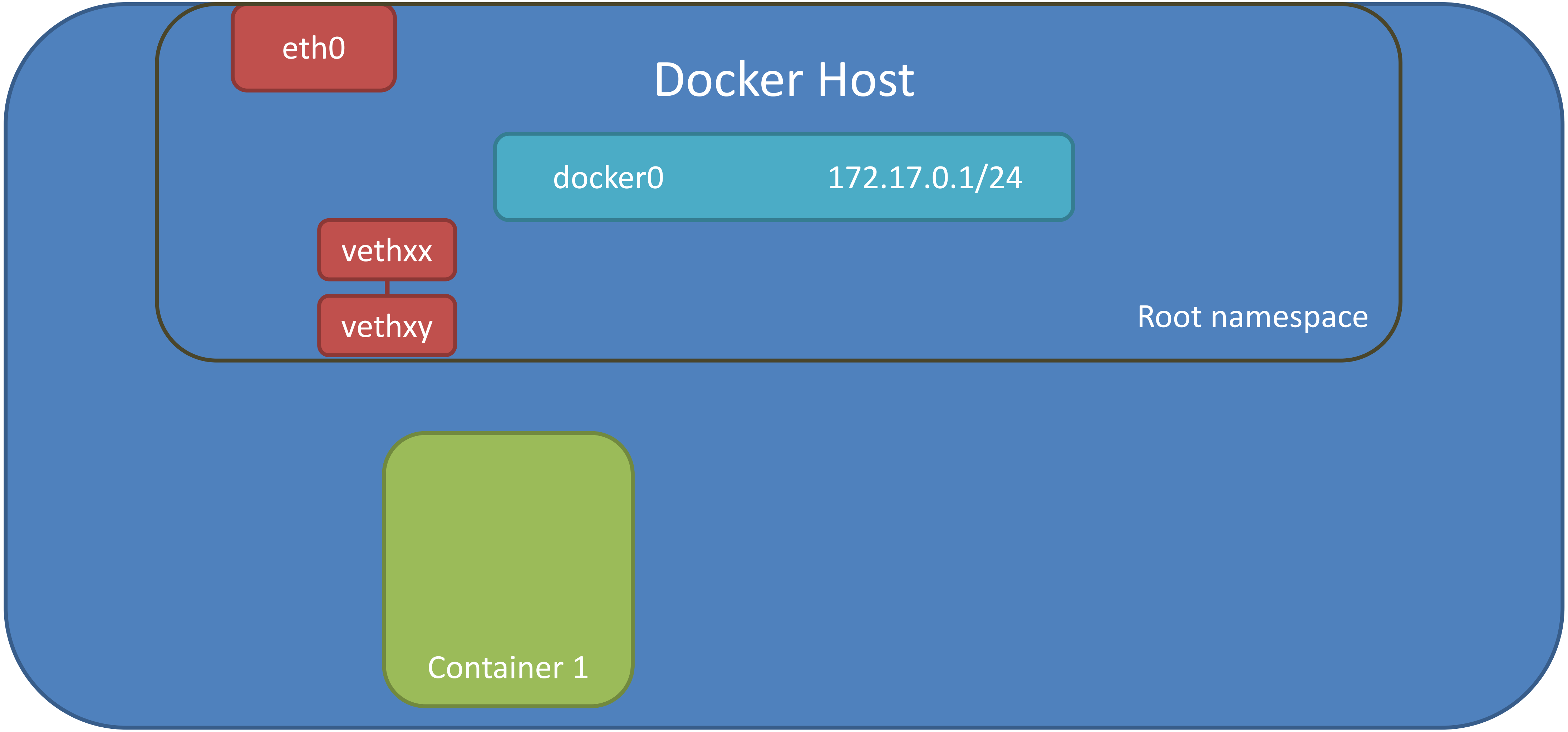
Layers in the image are read only.

When a container is created a new, writable layer is attached to it.

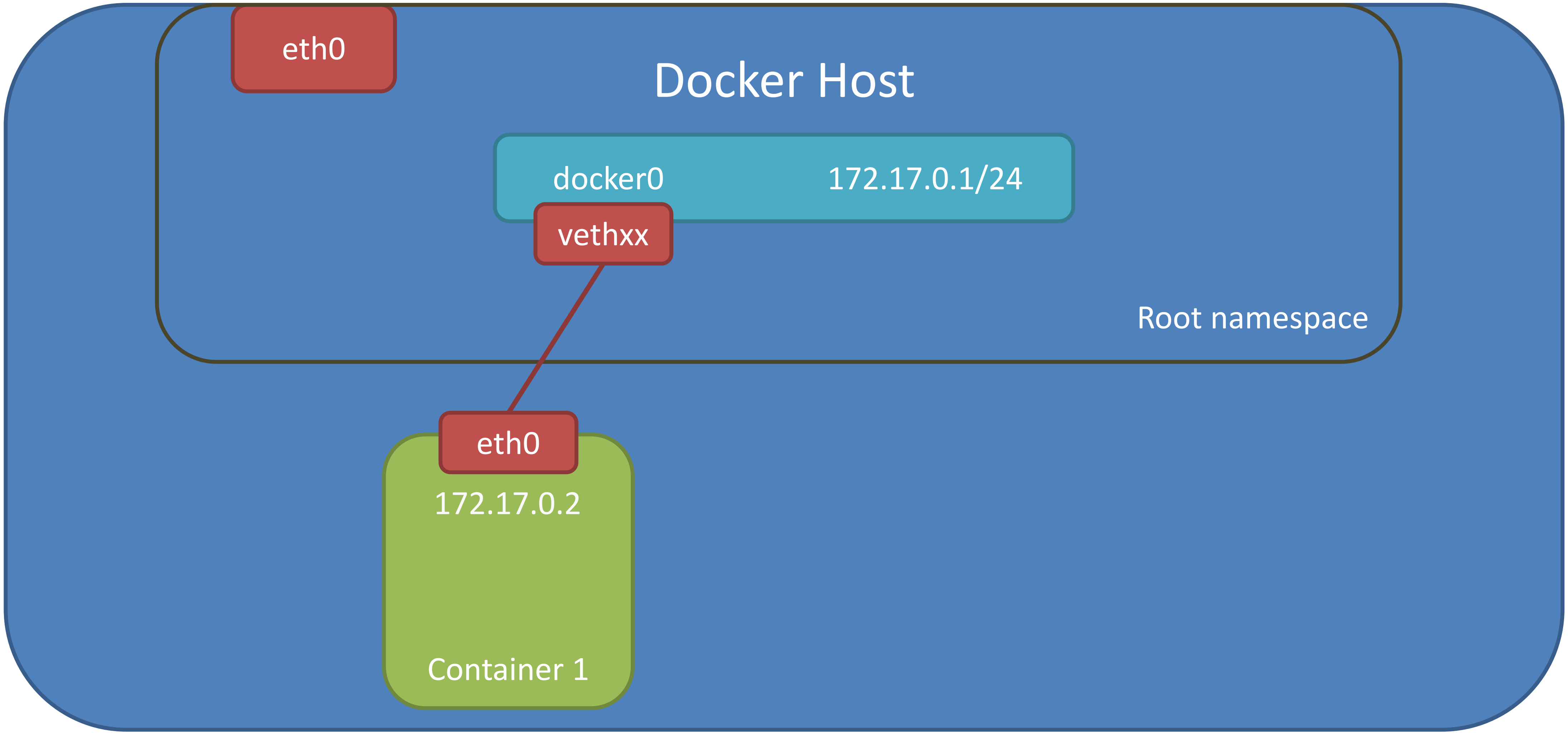
When multiple containers are run they get their own writable layer.



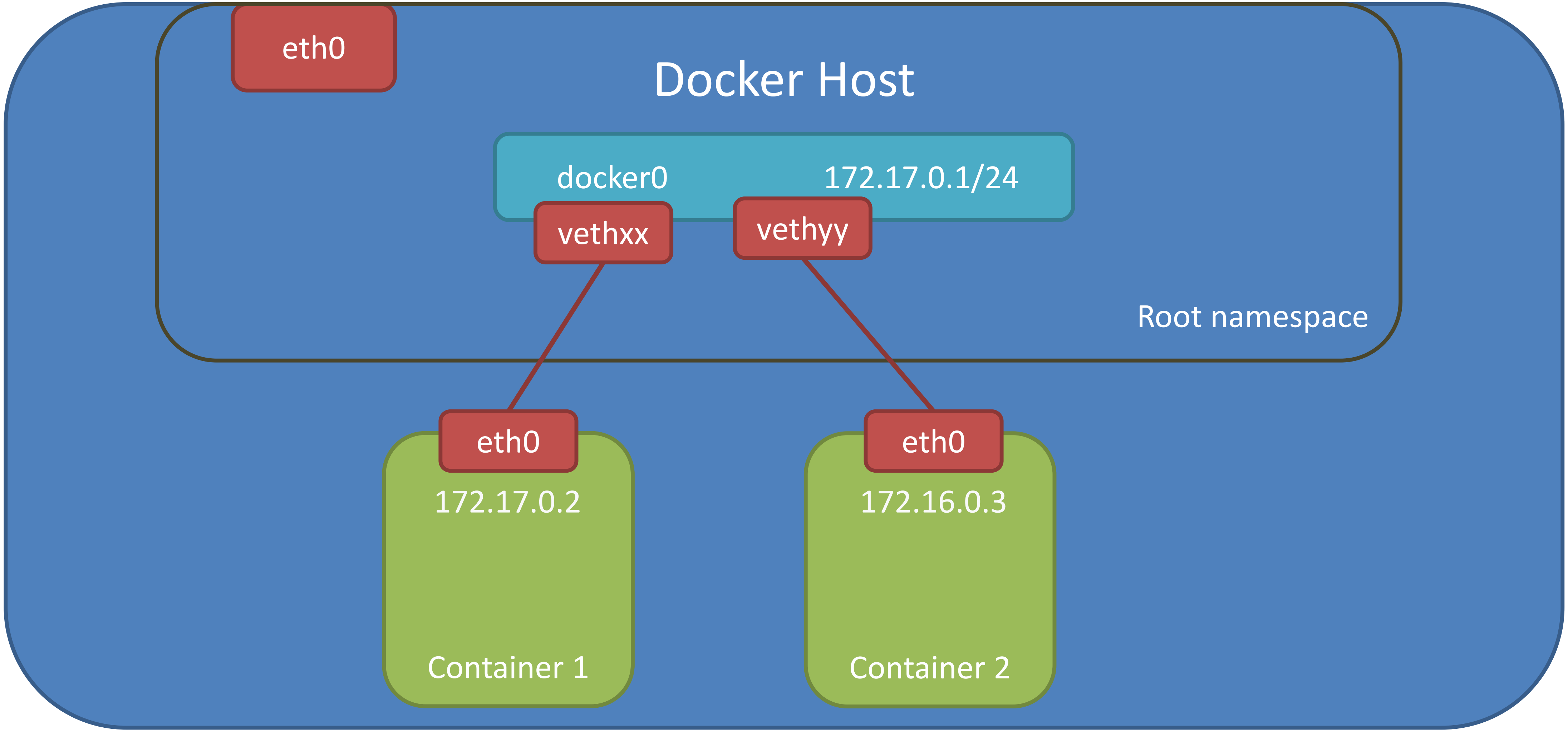
The Docker Networking Model



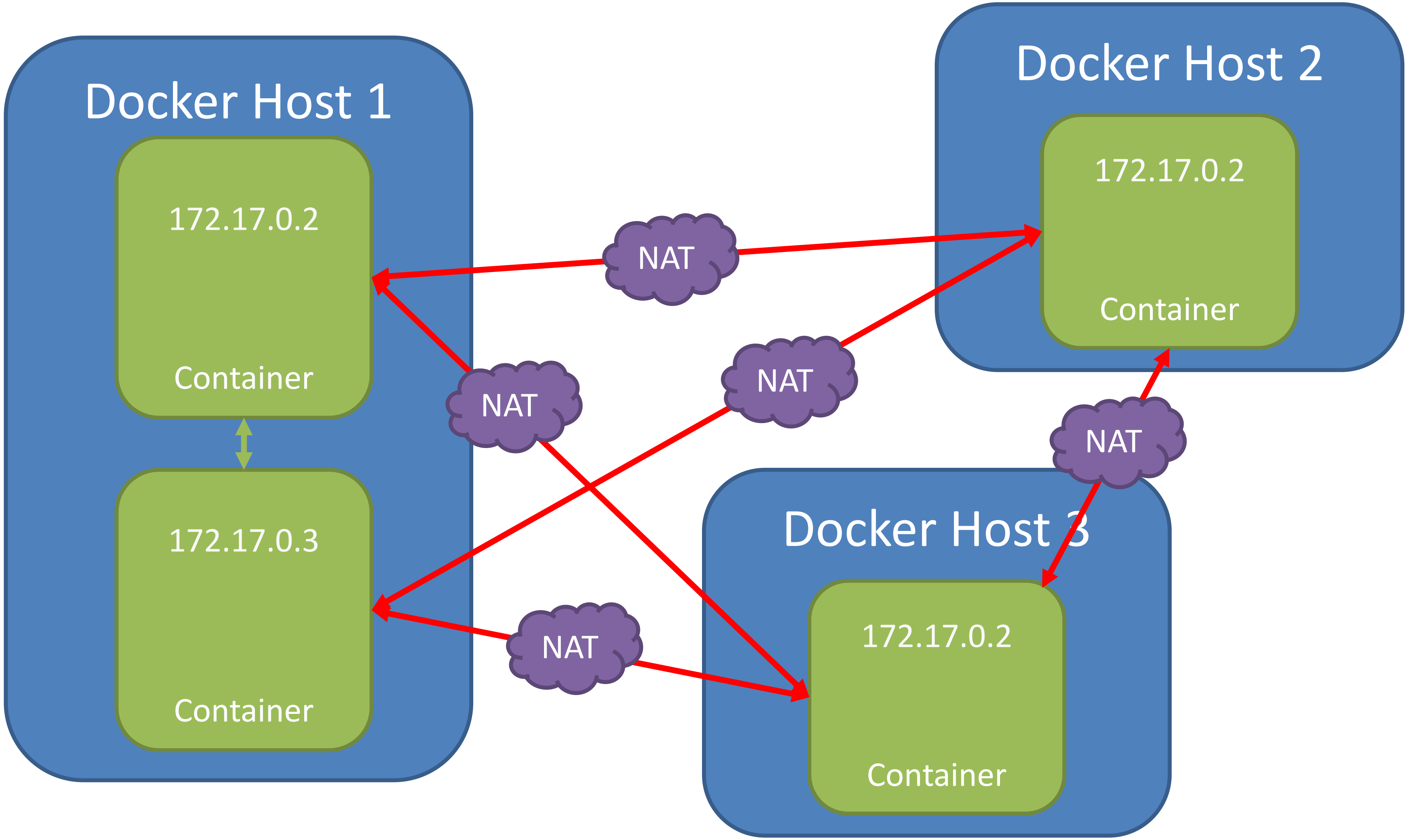
The Docker Networking Model



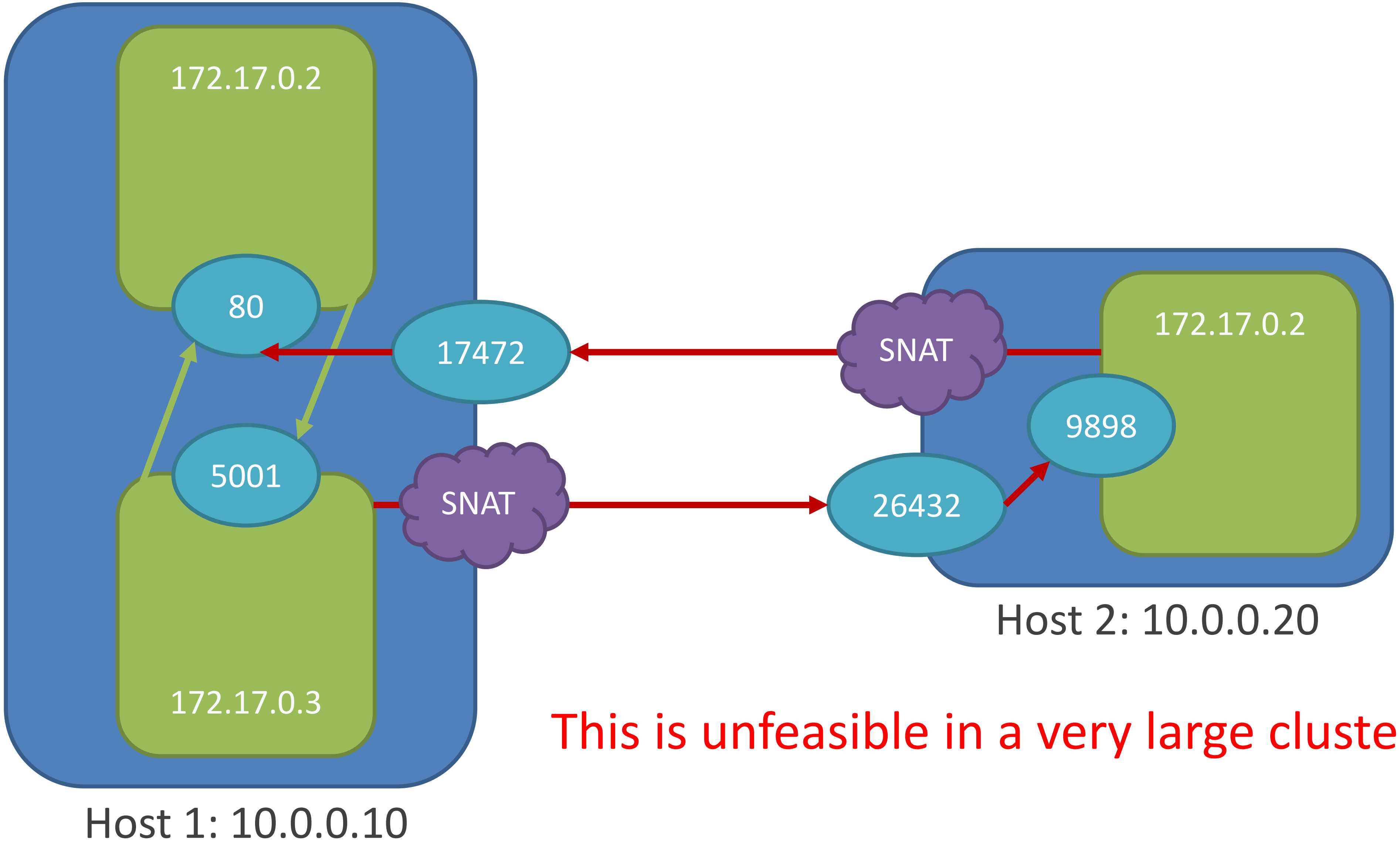
The Docker Networking Model



The Docker Networking Model

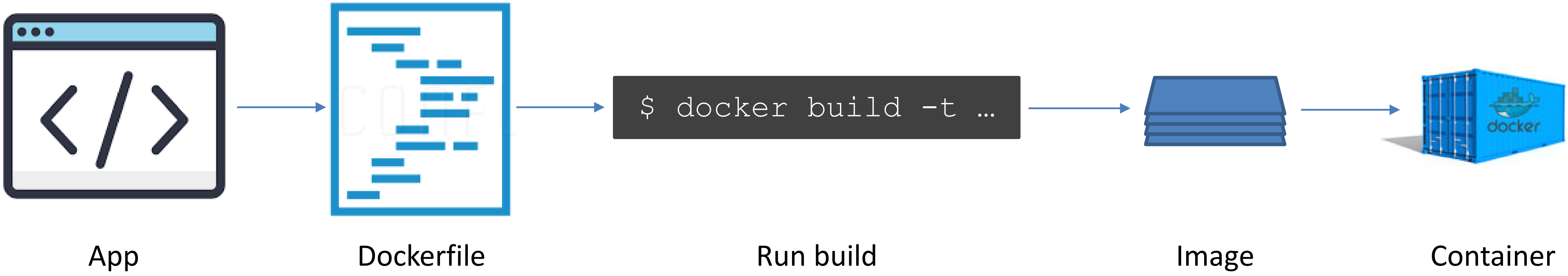


Docker Host Ports



This is unfeasible in a very large cluster!

Containerization workflow



Building Images

- Docker builds images automatically by reading the instructions from a Dockerfile (and using "context" folder)
- Text file in specific format with specific commands defined in Dockerfile reference
- E.g.

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY app .
CMD ["./app"]
```

- Almost each instruction creates one layer:
  - `FROM` creates a layer from the ubuntu:18.04 Docker image
  - `COPY` adds files from your Docker client's current directory
  - `RUN` builds your application with make
  - `CMD` specifies what command to run within the container
- Build:
  - `docker build –t <IMAGE-NAME> .`
  - build context:
    - the directory from which the "build" command is issued
    - the commands relative to this but it can be changed by `CD` command

Building Images – Another Esxample

```
# Use the official image as a parent image.
FROM node:current-slim

# Set the working directory.
WORKDIR /usr/src/app

# Copy the file from your host to your current location.
COPY package.json .

# Run the command inside your image filesystem.
RUN npm install

# Add metadata to the image to describe which port the container is listening on at runtime.
EXPOSE 8080

# Run the specified command within the container.
CMD [ "npm", "start" ]

# Copy the rest of your app's source code from your host to your image filesystem.
COPY . .
```

Building Image – General Recommendations

- Create ephemeral containers:
  - Container can be stopped and destroyed, then rebuilt and replaced with minimum set up and configuration
  - This principle refers to VI. Processes in The Twelve-factor App methodology (see later)
- Exclude with .dockerignore:
  - Similar to .gitignore
  - Exclude files not relevant to the build (without restructuring your source repository)
  - Can make the build-context and image size smaller:
    - build-context size is the first line of the build output:

```
Sending build context to Docker daemon 2.048kB
```

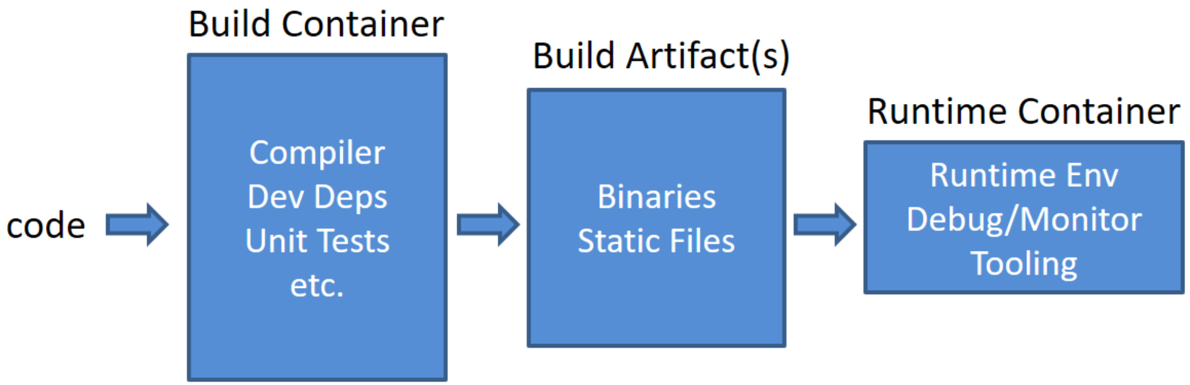
- Minimize the number of steps in the Dockerfile:
  - Decrease the number of steps and layers
  - Only RUN, COPY and ADD instructions create layers
  - Can improve build and pull performance

```
# 4 steps
FROM alpine:3.12
RUN apk update
RUN apk add git
RUN apk add curl
```

```
# 2 steps
FROM alpine:3.12
RUN apk update && \
    apk add git && \
    apk add curl
```

Building Images – General Recommendations

- Use-multi stage builds:
  - Minimize size -> inscrease performance
  - Decrease attack surface
- Method:
  1. Small base images:
     - node:8 -> ~670MB
     - node:8-wheezy -> ~520 MB
     - node:8-slim -> ~225MB
     - node:8-alpine -> ~65MB
  2. Builder pattern:



Let’s Get Under the Hood of a Docker Image

Dive:

- A tool for exploring a docker image, layer contents, and discovering ways to shrink the size of your Docker/OCI image.

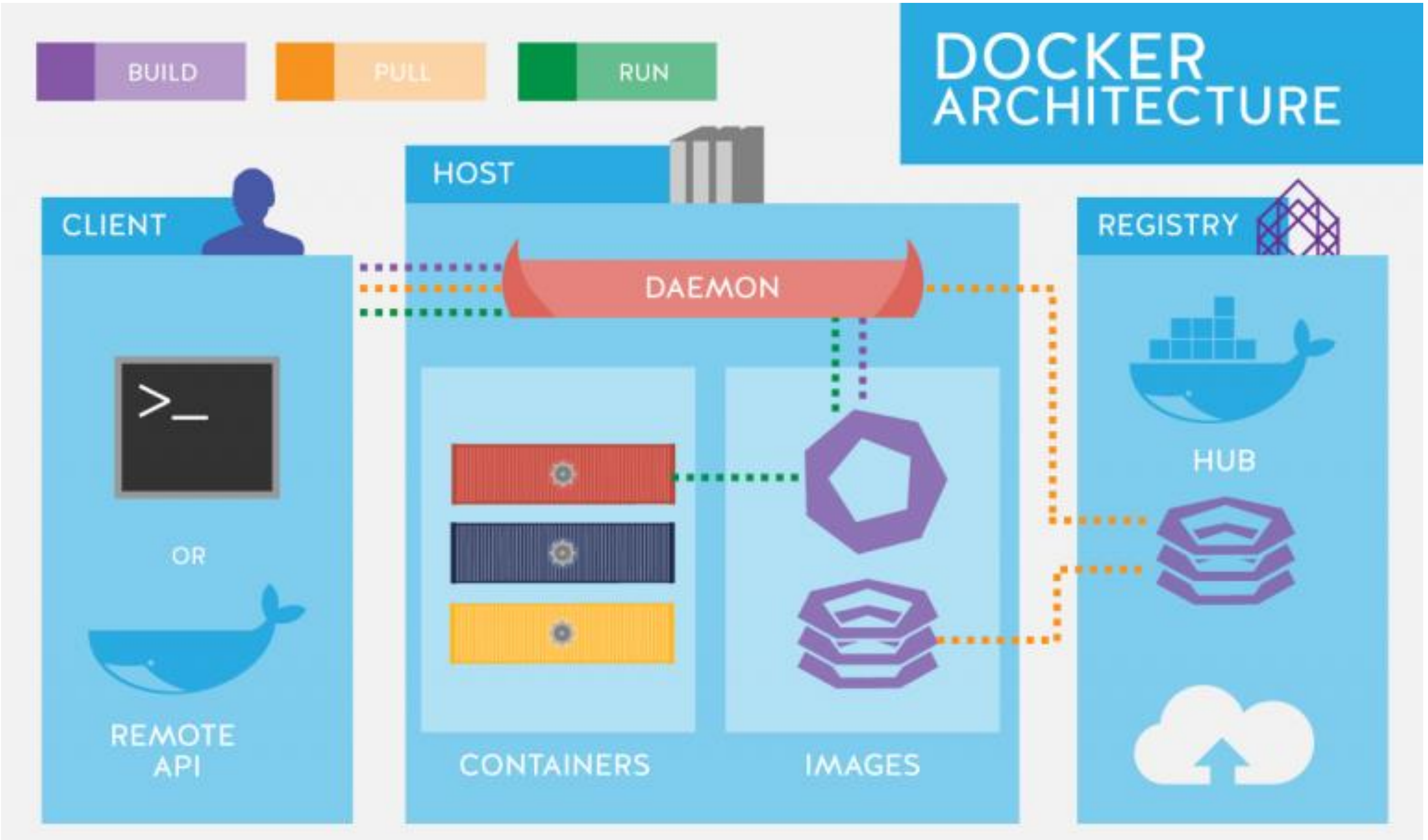
Features:

  - Show Docker image contents broken down by layer
  - Indicate what's changed in each layer
  - Estimate "image efficiency" (based on experimental metrics)
  - Quick build/analysis cycles (docker build -> dive build)
  - CI Integration (get a pass/fail result based on efficiency and wasted space)
  - Multiple container engines are supported

```
Analyzing image...
  efficiency: 98.4421 %
  wastedBytes: 32025 bytes (32 kB)
  userWastedPercent: 2.6376 %
Run CI Validations...
  Using CI config: .dive-ci
  PASS: highestUserWastedPercent
  SKIP: highestWastedBytes: rule disabled
  FAIL: lowestEfficiency: image efficiency is too low
        (efficiency=0.9844212134184309 < threshold=0.99)
Result:FAIL [Total:3] [Passed:1] [Failed:1] [Warn:0]
```

Docker Architecture

1. Build image from Dockerfile (with App)
2. Push image to the registry:
   - Private or public (DockerHub)
3. Pull image from the registry
4. Run the image, e.g. create a container



Docker Ecosystem

- Docker Hub:
  - Official source of pre-written Dockerfiles, providing public (for free) and private (paid) repositories for images
- Docker Desktop:
  - Application for MacOS and Windows machines for the building and sharing of containerized applications
- Docker Machine and Swarm:
  - Simple set of tools for moving and scaling your local projects to a variety of virtualization and cloud providers
  - Kubernetes is the direct competitor
- Docker Compose:
  - Makes assembling applications consisting of multiple components (and thus containers) simpler
  - You can declare all of them in a single configuration file started with one command
  - Great tool for development, but for production is not enough

Docker Container Alternatives

Kata Container:

- Open source project governed by the OpenStack Foundation (OSF)
- Address security concerns within containers through Intel® Virtualization Technology (Intel® VT)
- Launches containers as lightweight virtual machines (VMs)
- Intel Clear Containers project is merged into Kata Containers
- Security:
  - Runs in a dedicated kernel,
  - providing isolation of network, I/O and memory and
  - can utilize hardware-enforced isolation with virtualization VT extensions.
- Compatibility:
  - Supports OCI container format, Kubernetes CRI as well as legacy virtualization techniques
- Performance:
  - Delivers consistent performance as standard Linux containers
- Simplicity:
  - Eliminates the requirement for nesting containers inside full blown virtual machines

LXE:

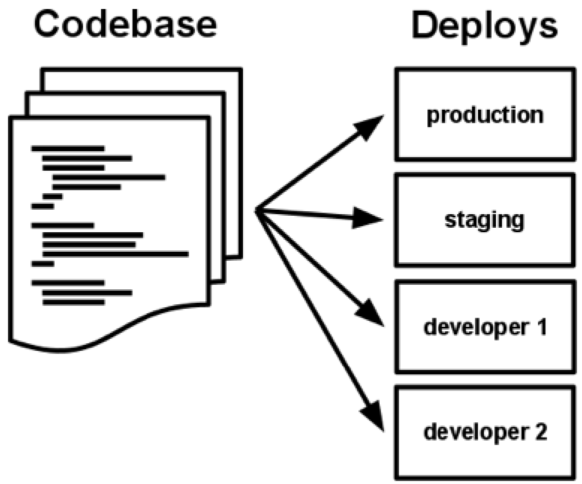
- shim of the Kubernetes CRI for LXD -> under heavy development

Modern Application Design: The Twelve-Factor Application

- Was drafted by developers at Heroku, a platform-as-a-service company
- Was first presented by Adam Wiggins circa 2011
- Best practices on how to build applications that are:
    - portable
    - scalable
    - resilient
    - in cloud environments / for software-as-a-service applications
- Useful for developers and ops engineers as well
- https://12factor.net/

Modern Application Design: The Twelve-Factor Application

- **I. Codebase**
  - One codebase tracked in revision control, many deploys
  - Always a one-to-one correlation between the codebase and the app:
    - If there are multiple codebases, it’s not an app – it’s a distributed system
    - Multiple apps sharing the same code is a violation of twelve-factor app
    - Factor shared code into libraries which can be included through the dependency manager



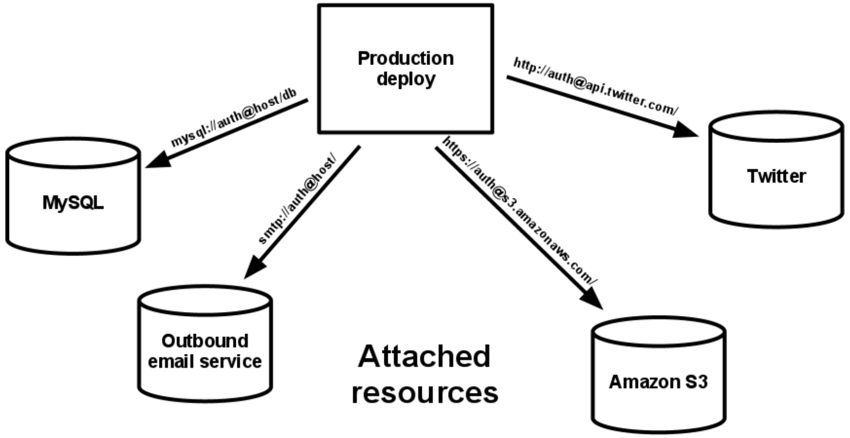
Modern Application Design: The Twelve-Factor Application

- **II. Dependencies**
  - Explicitly declare and isolate dependencies
  - Never rely on implicit existence of system-wide packages
  - Declare all dependencies, completely and exactly, via a *dependency declaration* manifest
  - Uses a *dependency isolation* tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system
  - E.g. In Python:
    - Pip -> for declaration
    - Virtualenv -> isolation

- **III. Store config in the environment**
  - Store config in the environment
  - Here *"config"* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:
    - Credentials to external services such as Amazon S3 or Twitter
    - Per-deploy values such as the canonical hostname for the deploy
    - Apps sometimes store config as constants in the code
  - So "config" does not include internal application config, only that changes between environments!
  - Keep the configuration in environments:
    - The twelve-factor app stores config in environment variables:
      - Little change of being checked into repo
      - Language- and OS-agnostic

Modern Application Design: The Twelve-Factor Application

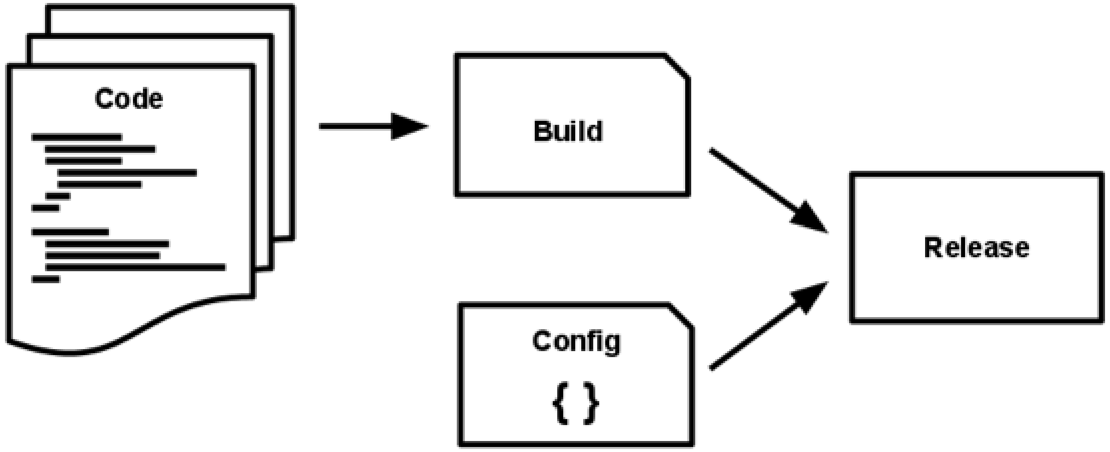
- **IV. Backing services**
  - Treat backing services as attached resources -> loose coupling
  - *Backing service* is any service the app consumes over the network as part of its normal operation, e.g.:
    - datastores (MySQL)
    - messaging/queueing systems (RabbitMQ)
    - metrics-gathering services (New Relic)
    - and even API-accessible consumer services (Twitter, Google Maps)



  - Services should be easily interchangeable: referencing them as simple URLs with login crendentials
  - This will ensure good portability and helps maintain your system:
    - E.g. swap out a local MySQL database with Amazon RDS without any changes to the app’s code

Modern Application Design: The Twelve-Factor Application

- **V. Build, release, run**
  - Strictly separate build and run stages
  - Build - converting code repo into an executable bundle known as the build.
  - Release - getting the build and combining it with a config on a certain environment- ready to run.
  - Run - starting the app in the deployment
  - Separation is important to make sure that automation and maintaining the system will be as easy as possible



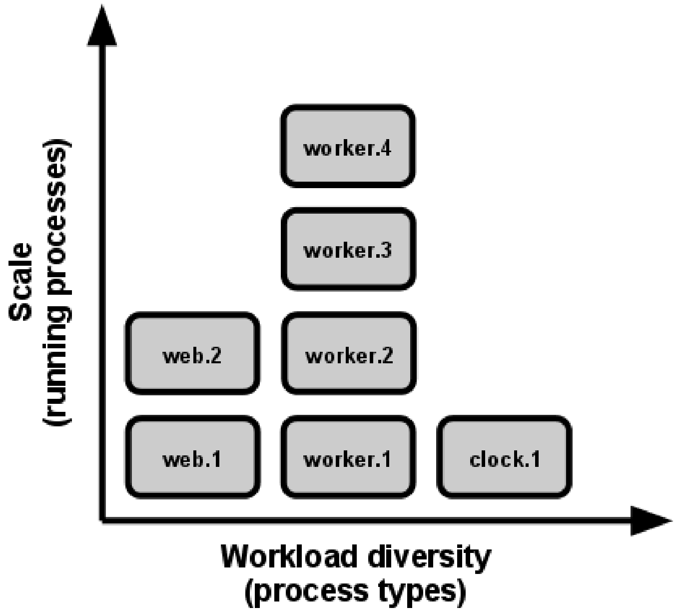
Modern Application Design: The Twelve-Factor Application

- **VI. Processes**
  - Execute the app as one or more stateless processes
  - Processes are stateless and share-nothing.
  - Any data that needs to persist must be stored in a stateful backing service, typically a database.
  - The memory space or filesystem of the process can be used as a brief, single-transaction cache.
    - E.g. downloading a large file, operating on it, and storing the results of the operation in the database.
  - Never assume that anything cached in memory or on disk will be available on a future request or job
  - With many processes: there is a high chance that a future request will be served by another process
  - With one process: a restart will usually wipe out all local state.

- **VII. Port binding**
  - Export services via port binding
  - Web apps are sometimes executed inside a webserver container:
    - E.g. PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat
  - The app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service
  - Exports HTTP as a service by binding to a port, and listening to requests coming in on that port:
    - E.g. use an URL like “http://localhost:5000/”
  - Let other services to treat you service as a resource (swappable, not local, etc.)

Modern Application Design: The Twelve-Factor Application

- **VIII. Concurrency**
  - Scale out via the process model
  - When the app needs to scale:
    - Scale by be deploying more copies of the application (process)
    - Rather than trying to make the application larger (run on more powerful machine)
  - Developer can architect their app:
    - To handle diverse workloads
    - By assigning each type of work to a *process type*
    - Scale by process type if it is necessary



Modern Application Design: The Twelve-Factor Application

- **IX. Disposability**
  - Maximize robustness with fast startup and graceful shutdown
  - The app can be started or stopped at a moment’s notice:
    - Processes should strive to minimize startup time
    - Processes shut down gracefully when they receive a SIGTERM signal:
      - E.g. for a web process:
        - refusing any new requests
        - allowing any current requests to finish, and then exiting
  - Crashes also need to be handled (however, this will be the responsibility of the whole system, not just the service)

Modern Application Design: The Twelve-Factor Application

- **X. Dev/prod parity**
  - Keep development, staging, and production as similar as possible
  - Historically, there have been substantial gaps between development
  - The time gap: A developer may work on code that takes days, weeks, or even months to go into production.
  - The personnel gap: Developers write code, ops engineers deploy it.
  - The tools gap: Dev stack is like Nginx, SQLite, and OS X, while the production stack is Apache, MySQL, and Linux.
  - The app is designed for continuous deployment

	Traditional app	**Twelve-factor app**
Time between deploys	Weeks	Hours
Code authors vs code deployers	Different people	Same people
Dev vs production environments	Divergent	As similar as possible

Modern Application Design: The Twelve-Factor Application

- **XI. Logs**
  - Treat logs as event streams
  - Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes
  - Logs in their raw form are typically a text format with one event per line
  - Logs have no fixed beginning or end, but flow continuously as long as the app is operating
  - The app never concerns itself with routing or storage of its output stream.
  - It should not attempt to write to or manage logfiles.
  - Instead, each running process writes its event stream, unbuffered, to stdout.
  - Log collectors can be used, e.g. ELK/EFK stack

Modern Application Design: The Twelve-Factor Application

- **XII. Admin processes**
  - Run admin/management tasks as one-off processes
  - Most of the applications require a few one-off tasks to be executed before the actual flow of the application starts
  - These tasks (like DB migration or executing one-off scripts in the environment) are not required very often:
    - So we generally create a script for it which we run from some other environment
  - However, what if these should be run periodically?
    - Handle it with schedulers and perform automatically
    - It can be a manual process as well
    - But in both cases admin code must ship with application code to avoid synchronization issues
    - So it must be made as part of our codebase itself managed in the version control system.