CONTAINERIZATION TRAINING

HOW WE TEACH

// We believe in **learning by doing**

// The training is **lab driven**

// Work together!

// Ask questions at any time

SESSION LOGISTICS

// 2 days duration

// Mostly exercises

// Regular breaks

ASSUMED KNOWLEDGE AND REQUIREMENTS

// Familiarity with Bash or Powershell

// Bash Cheat Sheet: http://bit.ly/2mTQr8l

YOUR LAB ENVIRONMENT

// You have been given an instance for use in exercises

// Ask instructor for credentials if you don't have them already

COURSE LEARNING OBJECTIVES

By the end of this course, learners will be able to

// Assess the advantages of a containerized software development & deployment

// Use Docker engine features necessary for running containerized applications

INTRODUCING DOCKER

WHAT WE WANT

Ideal software should

// be modular and flexible (devs)

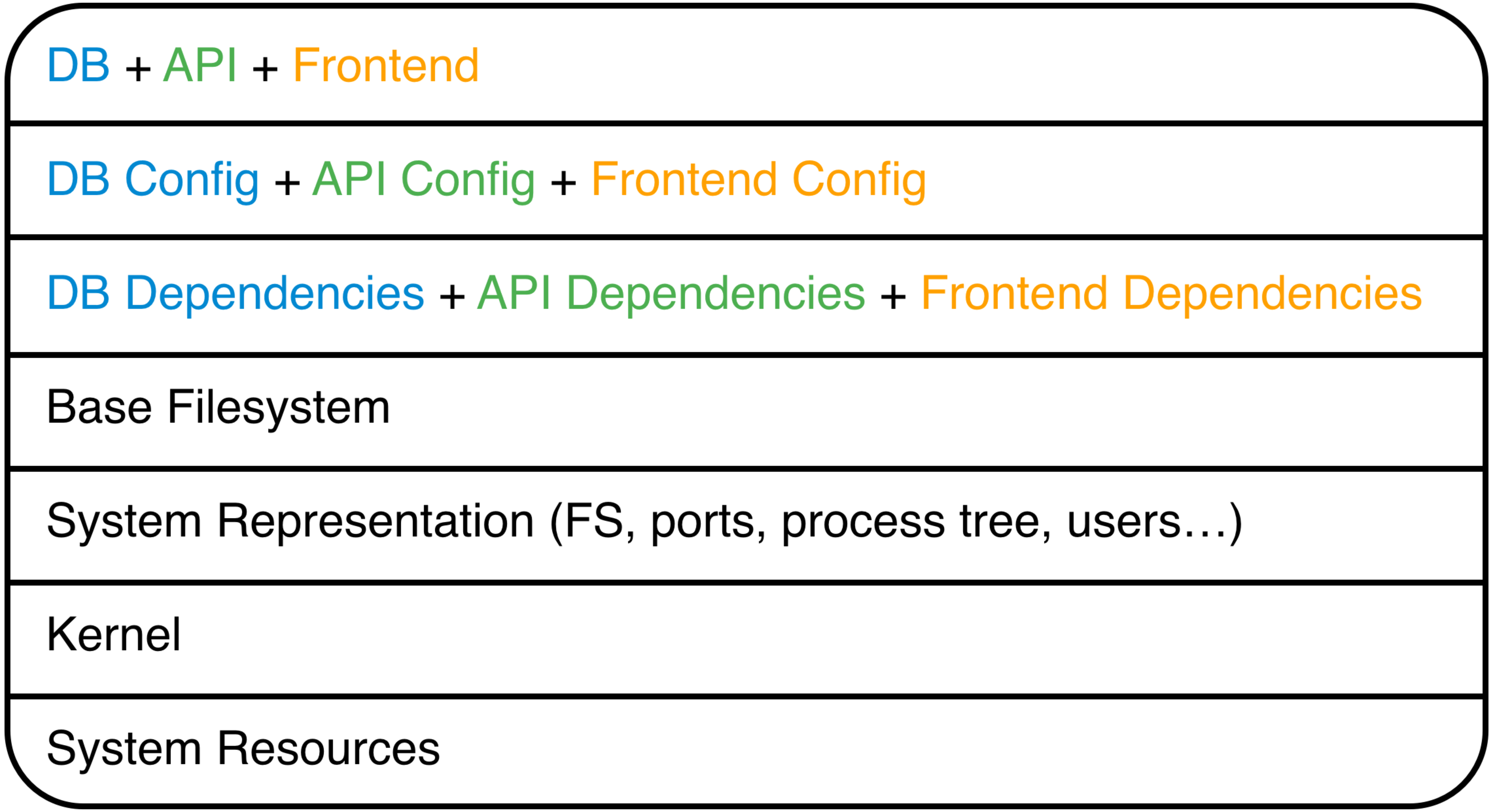
// be easy to migrate (devops)

// be easy to scale, monitor and lifecycle (ops)

// mitigate vulnerabilities (security)

// run cheap (business)

WITHOUT CONTAINERIZATION



DB + API + Frontend

DB Config + API Config + Frontend Config

DB Dependencies + API Dependencies + Frontend Dependencies

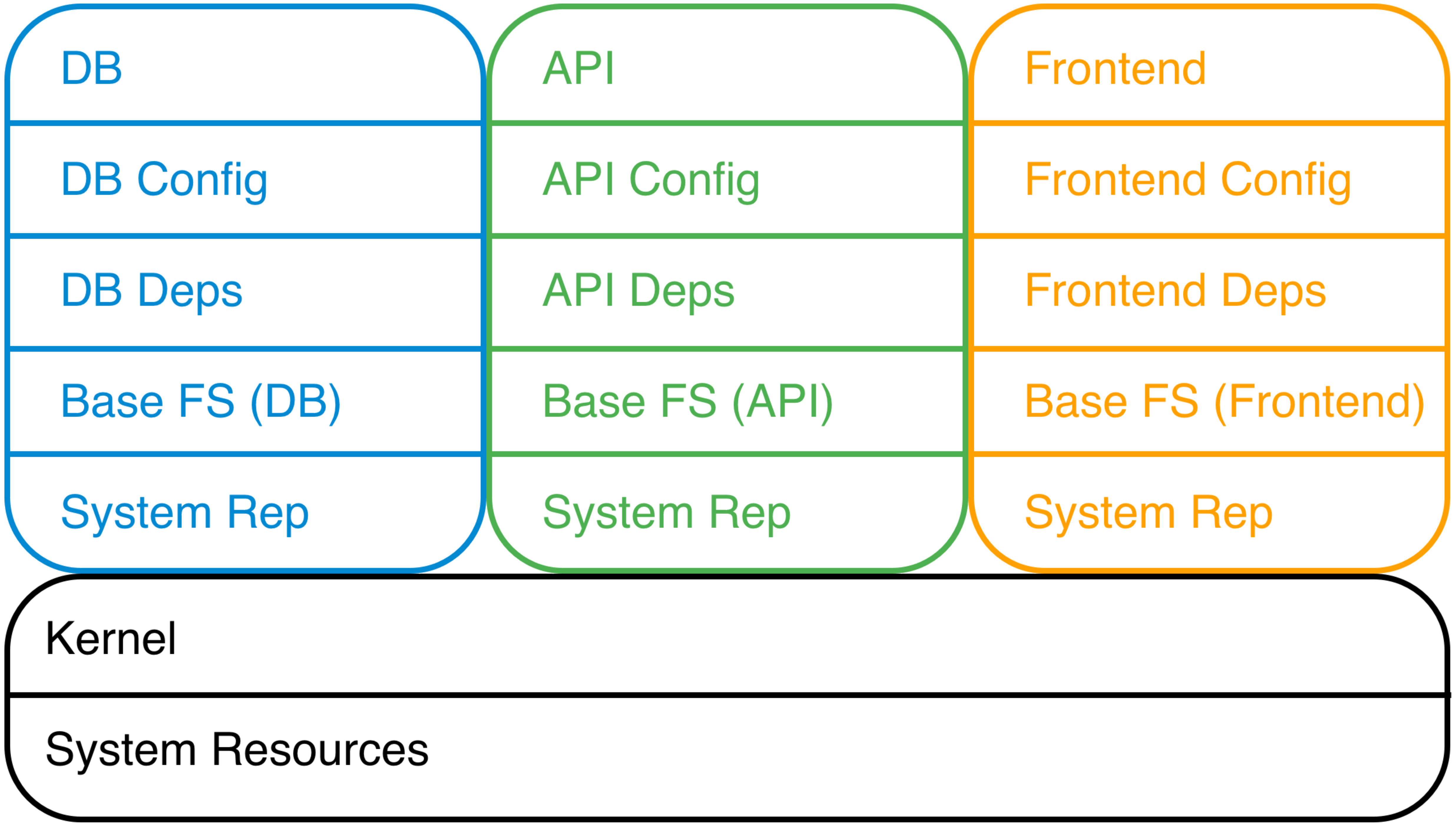
Base Filesystem

System Representation (FS, ports, process tree, users…)

Kernel

System Resources

WITH CONTAINERIZATION



DB

DB Config

DB Deps

Base FS (DB)

System Rep

API

API Config

API Deps

Base FS (API)

System Rep

Frontend

Frontend Config

Frontend Deps

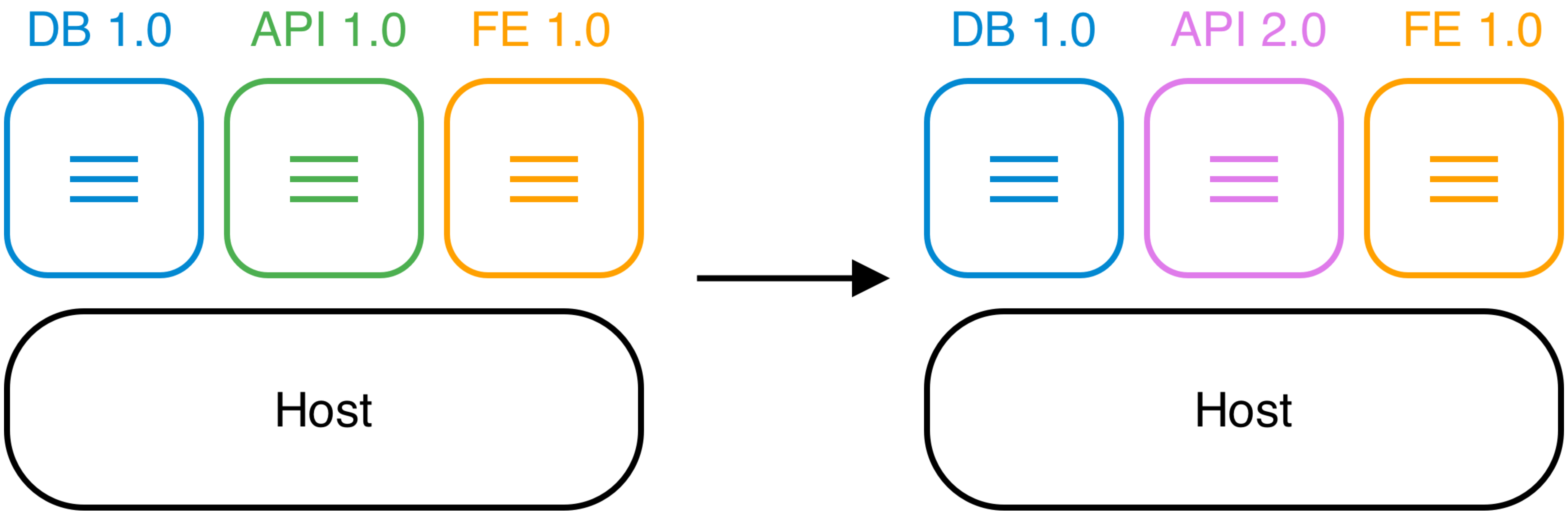
Base FS (Frontend)

System Rep

Kernel

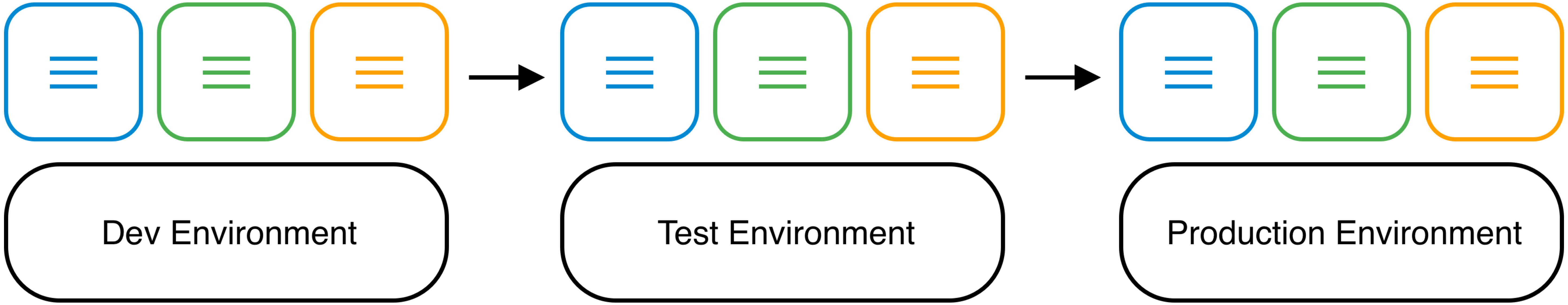
System Resources

RAPID DEVELOPMENT



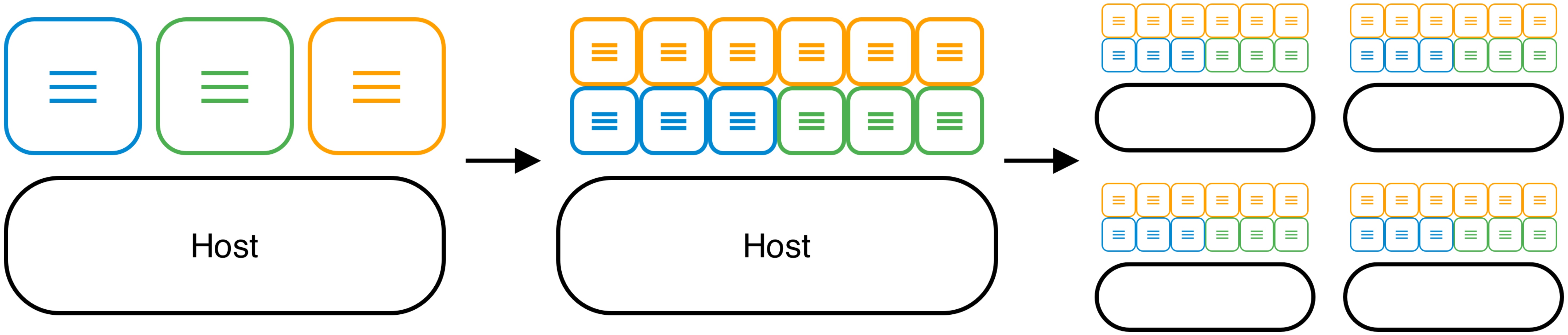
Containers can be removed and replaced with a minimum of impact on their neighbors, increasing developer choice and speed.

SMOOTH MIGRATION



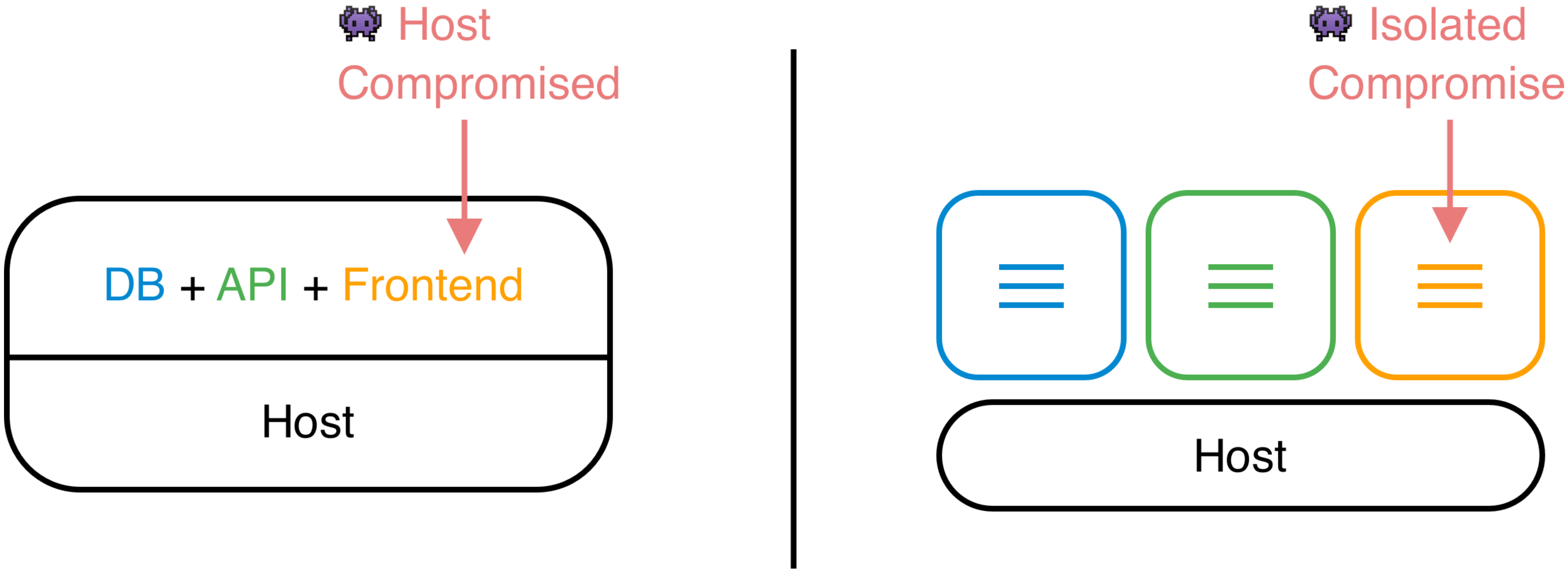
Containers carry their environment and dependencies with them, simplifying and minimizing requirements on the hosts that run them.

SIMPLE SCALE & MAINTENANCE



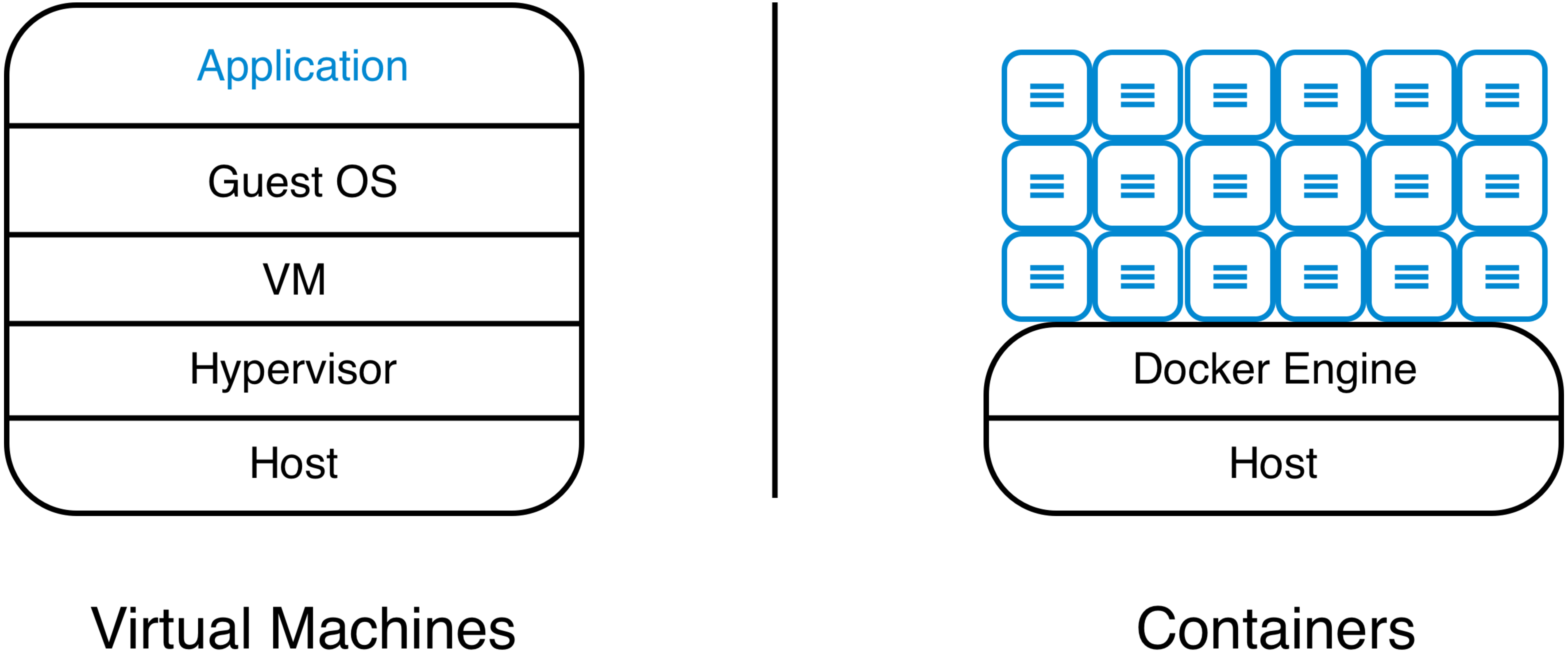
Weak coupling between containers minimizes side effects when scaling and simplifies monitoring.

SECURE BY DEFAULT



Containers have private system resources, so a compromise in one does not affect the rest.

APPLICATION DENSITY



Containers save datacenter costs by running many more application instances than virtual machines can on the same physical hosts.

CONTAINERIZATION BASICS

DISCUSSION: RUNNING CONTAINERS

What assurances would you need to run a process on an arbitrary host? Consider

// Hostile environments

// Required resources

LEARNING OBJECTIVES

By the end of this module, learners will be able to

// Describe what a container is in terms of processes and isolation tools

// Use the key commands for interacting with Docker containers

CONTAINERS ARE PROCESSES

Containers are processes sandboxed by

// Kernel namespaces

// Control Groups

// Root privilege management & syscall restrictions (Linux)

// VM isolation (Windows)

LINUX KERNEL NAMESPACES

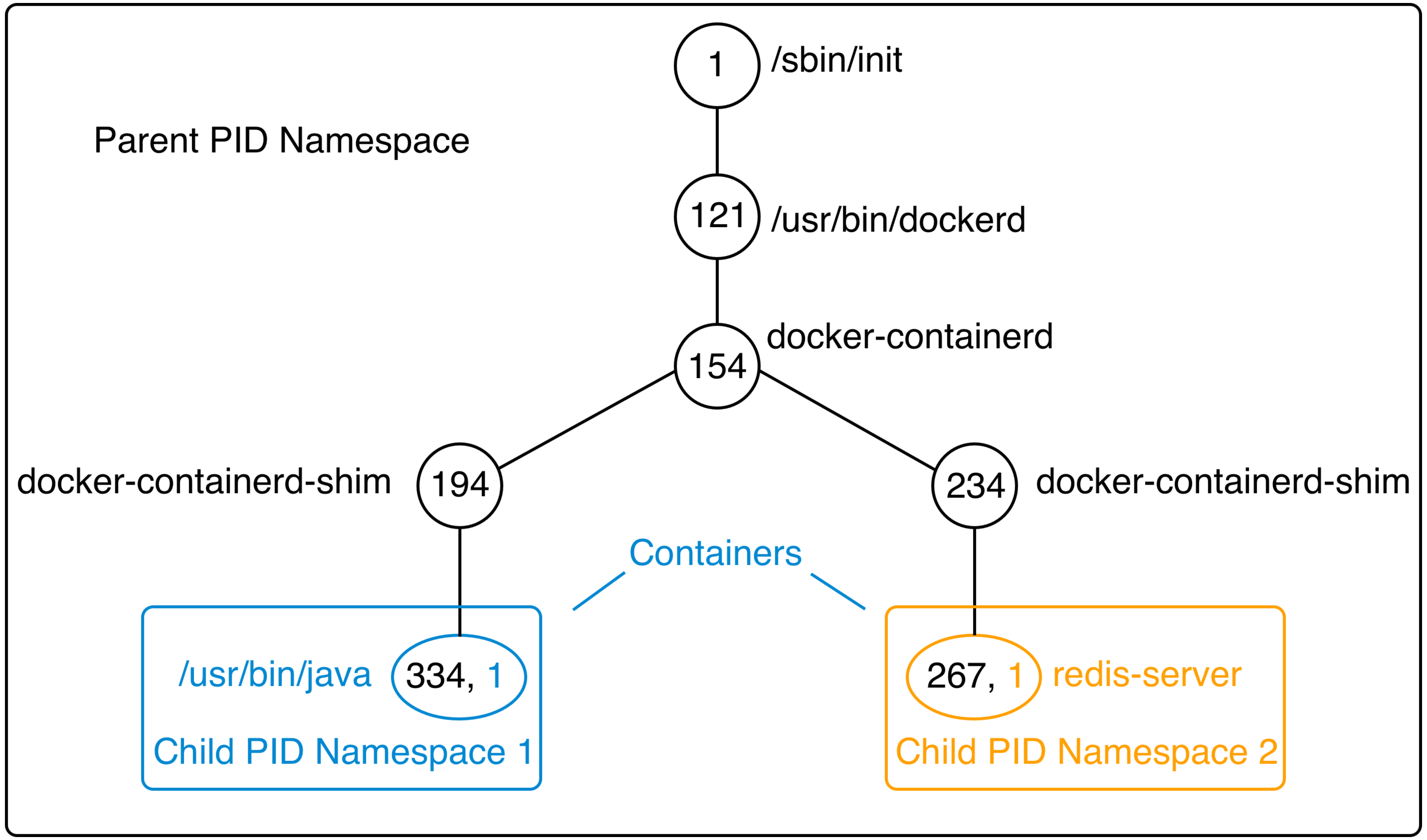
// **DEFAULT**

- Process IDs
- Network stacks
- Inter-process communications
- Mount points
- Hostnames

// **OPTIONAL**

- User IDs

LINUX PID KERNEL NAMESPACE



OPTIONAL LINUX ISOLATION FEATURES

// Control groups: limit memory & CPU

// Root privilege management: whitelist root powers

// System call management: whitelist available system calls

// Linux Security Modules: mandatory filesystem access control

INSTRUCTOR DEMO: PROCESS ISOLATION

See the demo

// Process Isolation

In the Exercises book.

EXERCISE: CONTAINER BASICS

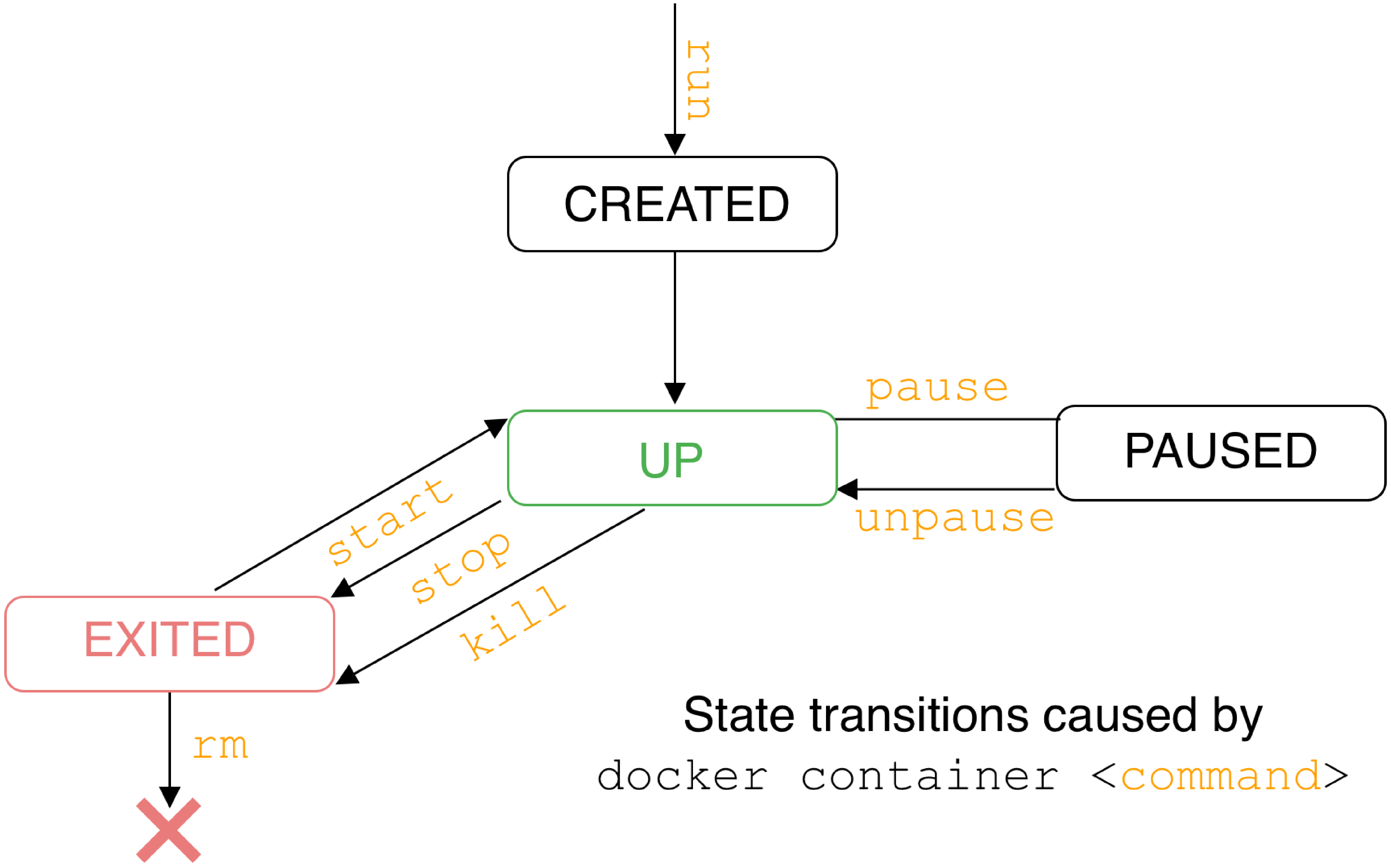
Work through

- // Running and Inspecting a Container
- // Interactive Containers
- // Detached Containers and Logging
- // Starting, Stopping, Inspecting and Deleting Containers

In the Exercises book.

21M 4S

CONTAINER LIFECYCLE



State transitions caused by
`docker container <command>`

CONTAINER LOGS

// STDOUT and STDERR for a containerized process

// `docker container logs <container name>`

CONTAINER BASICS TAKEAWAYS

// Single process constrained by kernel namespaces, control groups and other technologies

// Private & ephemeral filesystem and data

FURTHER READING

- // List of container commands: http://dockr.ly/2iLBV2l
- // Getting started with containers: http://dockr.ly/2gmxKWB
- // Start containers automatically: http://dockr.ly/2xB8sMl
- // Limit a container's resources: http://dockr.ly/2wqN5Nn
- // Keep containers alive during daemon downtime: http://dockr.ly/2emLwb5
- // Isolate containers with a user namespace: http://dockr.ly/2gmyKdf
- // Intro to Windows Containers: https://dockr.ly/2CTYhYb

CREATING IMAGES

DISCUSSION: PROVISIONING FILESYSTEMS

What are some potential difficulties with provisioning entire filesystems for containers? How can we avoid these problems?

LEARNING OBJECTIVES

By the end of this module, learners will be able to

- // Create images via several methods
- // Describe the filesystem structure underlying an image
- // Understand the performance implications of different image design decisions
- // Correctly tag and namespace images for distribution on a registry

WHAT ARE IMAGES?

// A **filesystem** for container process

// Made of a stack of **immutable** layers

// Start with a **base image**

// New layer for each change

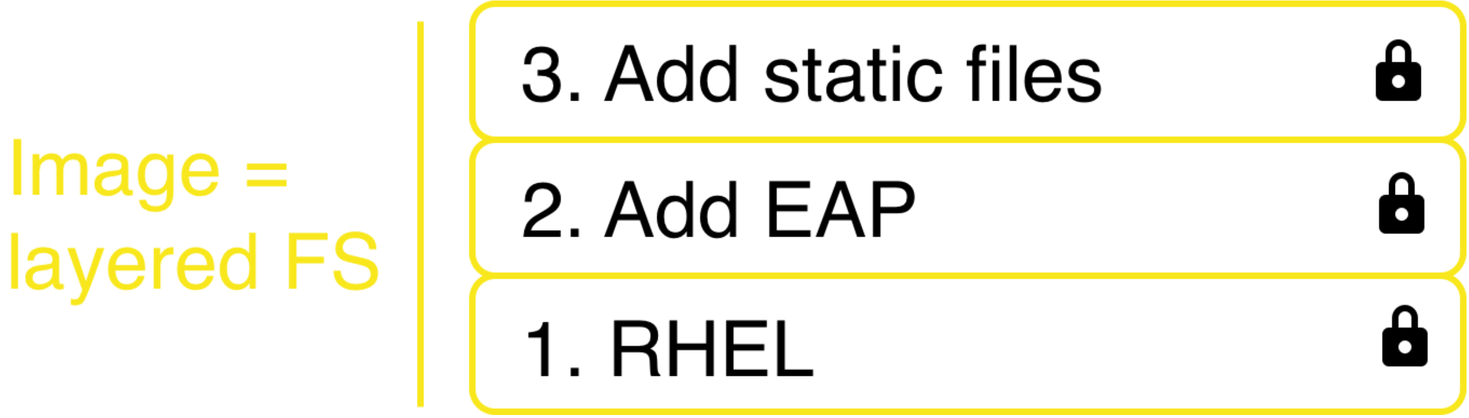


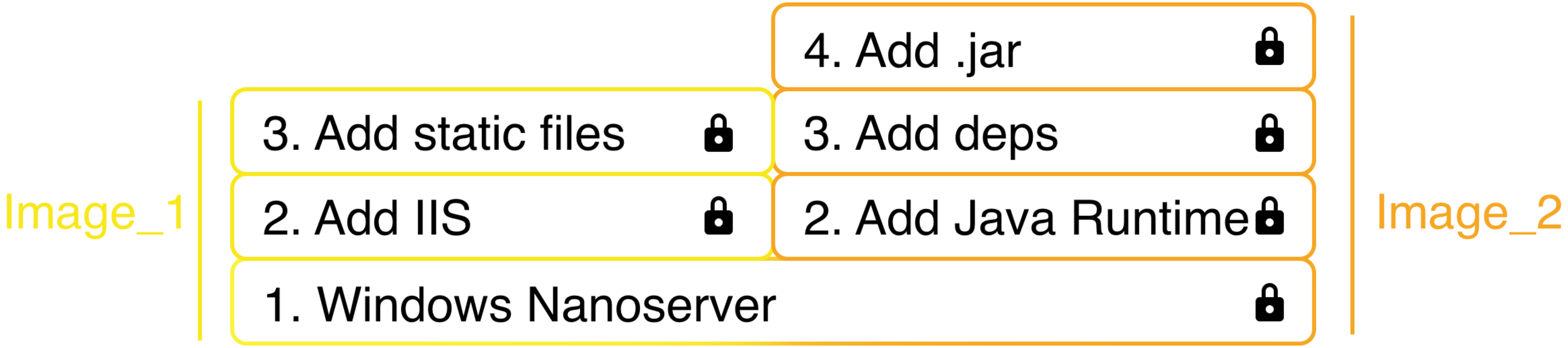
Image = layered FS

3. Add static files

2. Add EAP

1. RHEL

SHARING LAYERS



4. Add .jar 🔒

3. Add static files 🔒

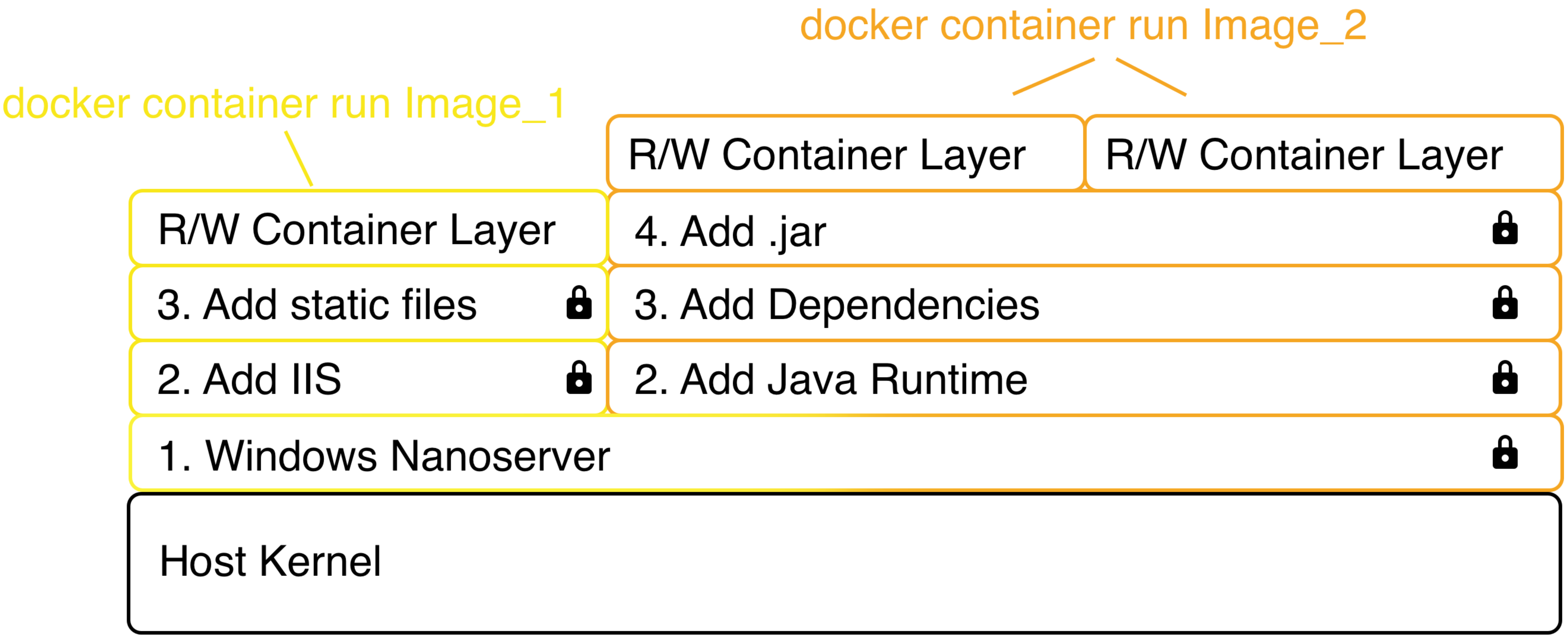
3. Add deps 🔒

2. Add IIS 🔒

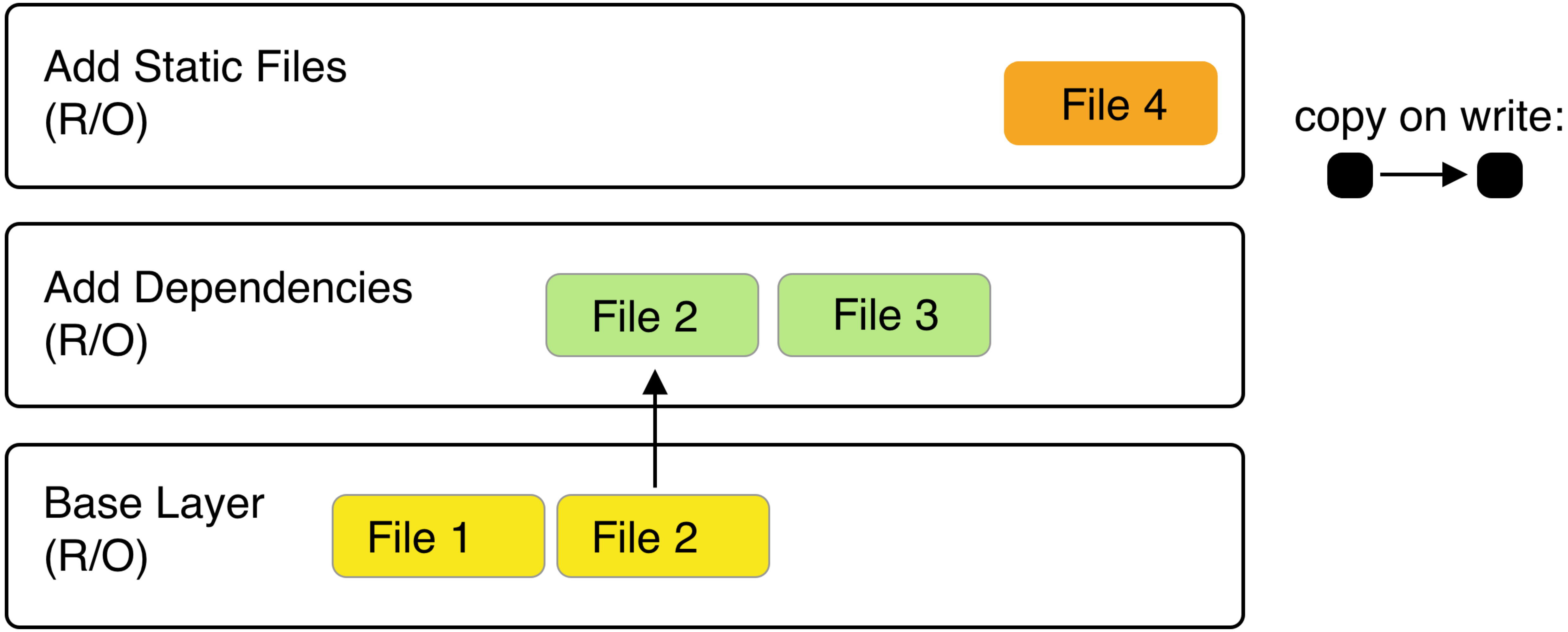
2. Add Java Runtime 🔒

1. Windows Nanoserver 🔒

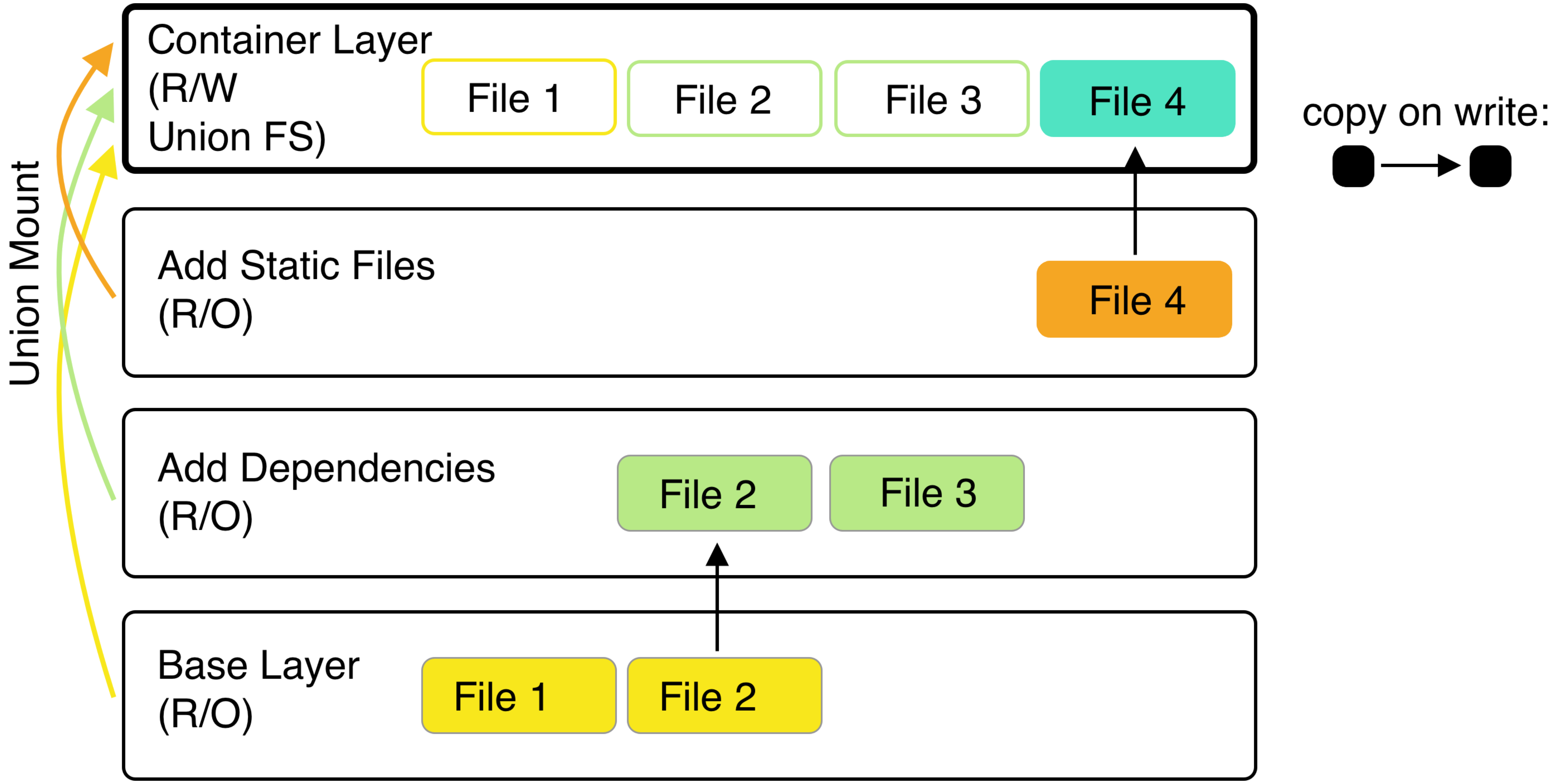
THE WRITABLE CONTAINER LAYER



IMAGES: COPY ON WRITE



LINUX CONTAINERS: UNION FS



CREATING IMAGES

Three methods:

// **Commit** the R/W container layer as a new R/O image layer.

// Define new layers to add to a starting image in a **Dockerfile**.

// **Import** a tarball into Docker as a standalone base layer.

COMMITTING CONTAINER CHANGES

// `docker container commit`
saves container layer as new R/O image layer

// Pro: build images interactively

// Con: hard to reproduce or audit; **avoid this** in practice.

DOCKERFILES

// Content manifest

// Provides image layer documentation

// Enables automation (CI/CD)

DOCKERFILES

// `FROM` command defines base image.

// Each subsequent command adds a layer or metadata

// `docker image build ...` builds image from Dockerfile

```
# Comments begin with the pound sign
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y wget
ADD /data /myapp/data
...
```

EXERCISES: CREATING IMAGES

Work through

- // Interactive Image Creation
- // Creating Images with Dockerfiles (1/2)

in the Exercises book.

20M 28S

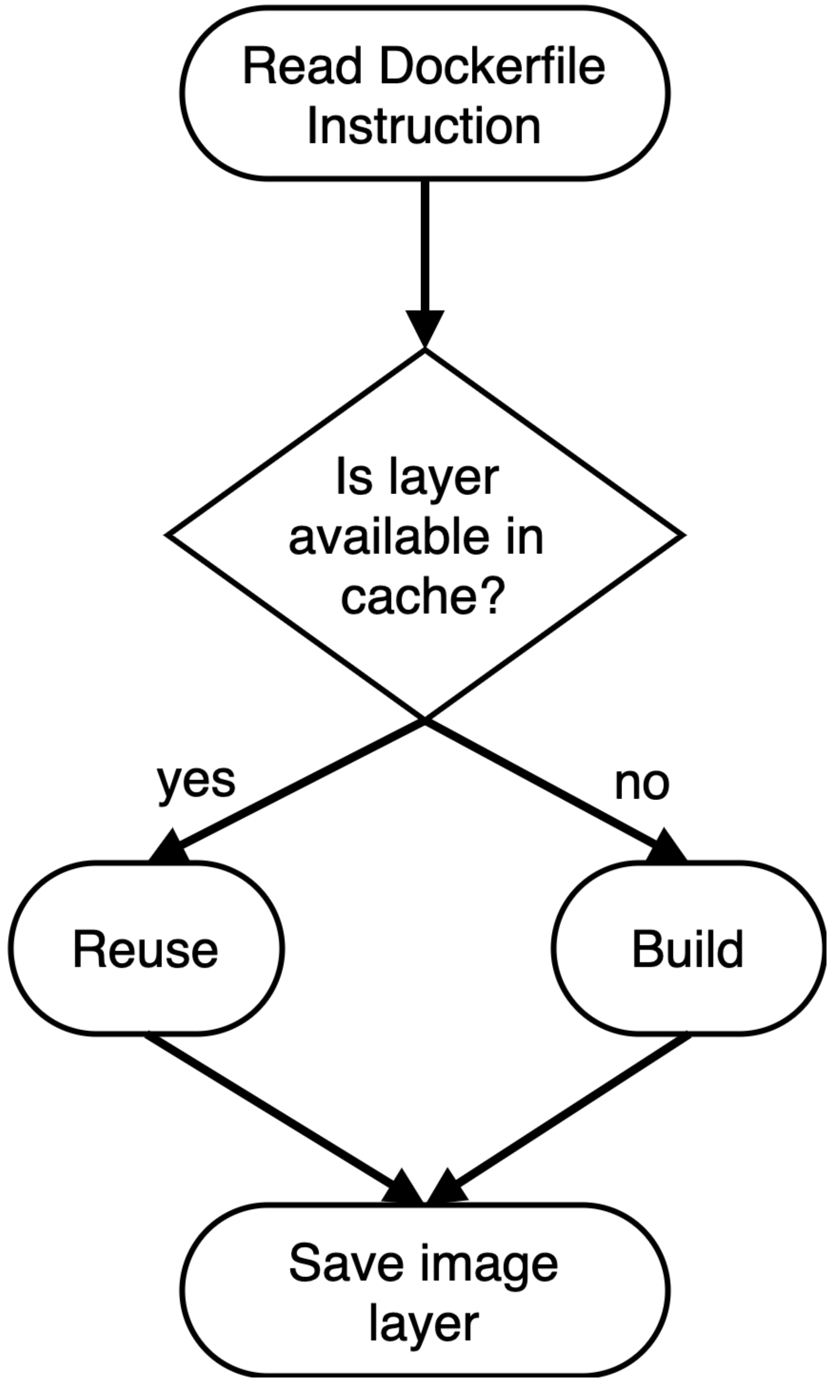
INSTRUCTOR DEMO: CREATING IMAGES

See the demo

// Creating Images

In the Exercises book.

BUILD CACHE



// After completion, the resulting image layer is labeled with a hash of the content of all current image layers in the stack.

CMD AND ENTRYPOINT

// Recall all containers run a process as their PID 1

// `CMD` and `ENTRYPOINT` allow us to specify default processes.

CMD AND ENTRYPOINT

// `CMD` alone: default command and list of parameters.

// `CMD` + `ENTRYPOINT`: `ENTRYPOINT` provides command, `CMD` provides default parameters.

// `CMD` overridden by command arguments to `docker container run`

// `ENTRYPOINT` overridden via `--entrypoint` flag to `docker container run`.

SHELL VS. EXEC FORMAT

```
# Shell form
CMD sudo -u ${USER} java ...

# Exec form
CMD ["sudo", "-u", "jdoe", "java", ...]
```

EXERCISE: DOCKERFILES (2/2)

Work through

// Creating Images with Dockerfiles (2/2)

In the Exercises book.

20M 14S

COPY AND ADD COMMANDS

`COPY` copies files from build context to image

```
COPY <src> <dest>
```

`ADD` can also untar* or fetch URLs.

** Linux containers only!*

In both cases

// create checksum for files added

// log checksum in build cache

// cache invalidated if checksum changed

DOCKERFILE COMMAND ROUNDUP

// FROM: base image to start from (usually OS)

// RUN: run a command in the environment defined so far

// CMD and ENTRYPOINT: define default behavior

// COPY and ADD: copy files into container

Many more Dockerfile commands are available; see the docs at
https://docs.docker.com/engine/reference/builder/

ADVANCED DOCKERFILE CONSTRUCTION

How can we build images that are

// Lightweight

// Secure

// Minimal build times

THE SCRATCH IMAGE

// An "empty" image

// Can't be pulled

// Doesn't create a layer

// Used for building images not based on any pre-existing image

// Linux only

```
FROM scratch

ADD centos-7-docker.tar.xz /

LABEL org.label-schema.schema-version="1.0" \
org.label-schema.name="CentOS Base Image" \
org.label-schema.vendor="CentOS" \
org.label-schema.license="GPLv2" \
org.label-schema.build-date="20181205"

CMD ["/bin/bash"]
```

MULTI-STAGE BUILDS

Hello World, in C:

```
FROM alpine:3.5
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
ADD hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

Builds to:

```
$ docker image ls hwc
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hwc                 latest              142c29686b6a        15 hours ago        184 MB
```

MULTI-STAGE BUILDS

Hello World, lightweight:

```
# Full SDK version (built and discarded)
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
ADD hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello

# Lightweight image returned as final product
FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

Builds to:

```
$ docker image ls hwc
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hwc                 latest              5d925cfc9c96        39 seconds ago      4MB
```

BUILD TARGETS

Dockerfile

```
FROM <base image> as base
...

FROM <foo image> as foo
...

FROM <bar image> as bar
...

FROM alpine:3.4
...
COPY --from foo ...
COPY --from bar ...
...
```

Building the image

```
docker image build --target <name> ...
```

EXERCISE: MULTI-STAGE BUILDS

Work through

// Multi-Stage Builds

In the Exercises book.

19M 57S

IMAGE CONSTRUCTION BEST PRACTICES

// Start with official images

// Use multi-stage builds to drop compilers, SDKs...

// More layers leverage the cache...

// ...but fewer layers perform better.

DEVELOPMENT: MORE LAYERS

Bad caching:

```
FROM python:3.5-alpine
RUN mkdir /app
COPY /mypy /app/
RUN pip install -r app/reqs.txt
...
```

Good caching:

```
FROM python:3.5-alpine
RUN mkdir /app
COPY /mypy/reqs.txt /app/
RUN pip install -r app/reqs.txt
COPY /mypy /app/
...
```

PRODUCTION: LESS LAYERS

// To collapse ALL image layers:

```
docker container run -d --name demo mytallimage:1.0
docker container export demo > image.tar
cat image.tar | docker image import - myflatimage:1.0
```

// Or build with `--squash` flag (experimental): compress all non-base layers

// Combine `container export` with `--squash` for one shareable base layer + one application-specific upper layer

BEST PRACTICE: PATCHING & UPDATES

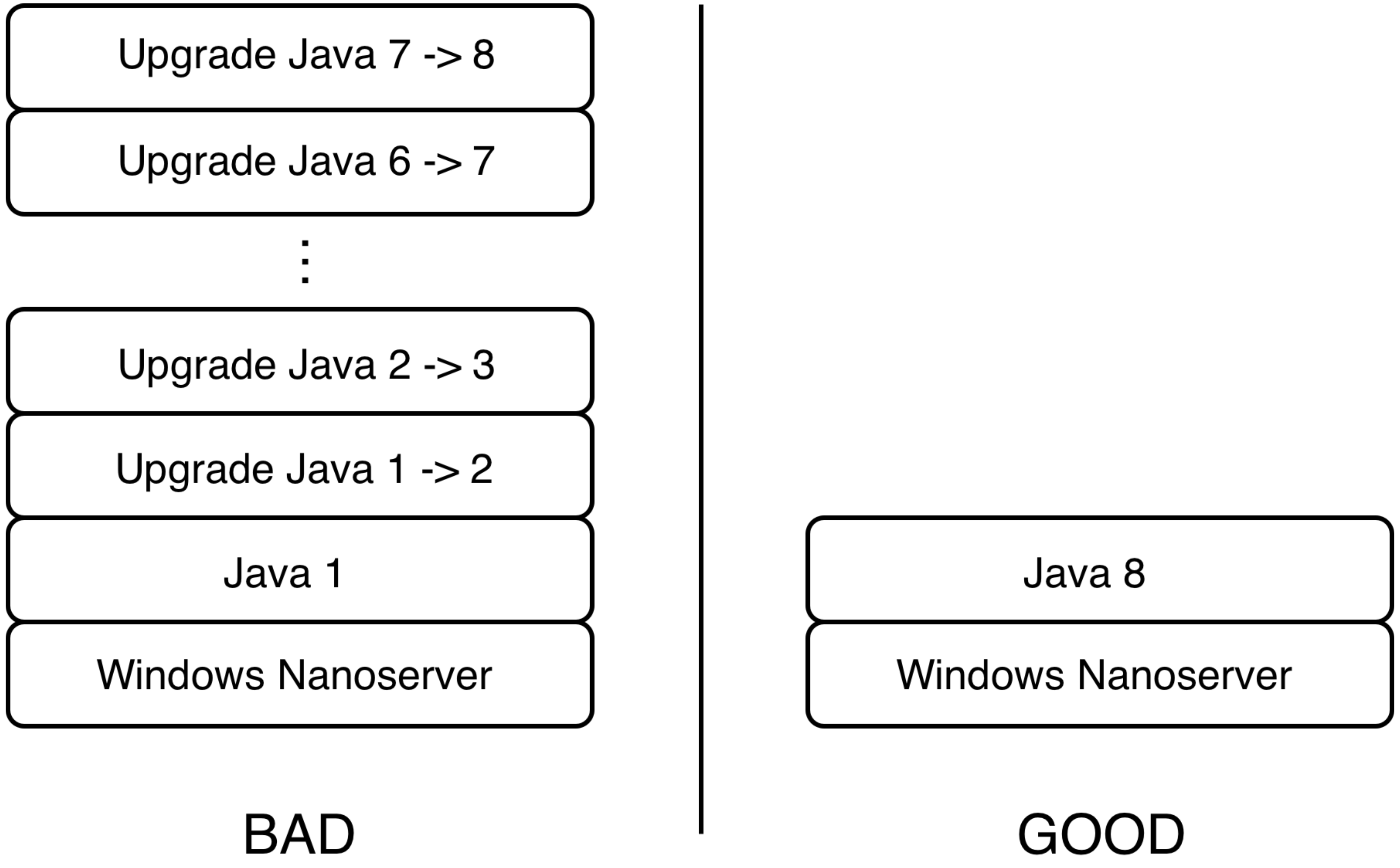


IMAGE TAGS

// Optional string after image name, separated by `:`

// `:latest` by default

// Same image with two tags shares same ID, image layers:

```
$ docker image ls centos*
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              7                   8140d0c64310        7 days ago          193 MB
$ docker image tag centos:7 centos:mytag
$ docker image ls centos*
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              7                   8140d0c64310        7 days ago          193 MB
centos              mytag               8140d0c64310        7 days ago          193 MB
```

IMAGE NAMESPACES

Images exist in one of three namespaces:

// Root (`ubuntu`, `nginx`, `mongo`, `mysql`, ...)

// User / Org (`jdoe/myapp:1.1`, `microsoft/nanoserver:latest`, ...)

// Registry (`FQDN/jdoe/myapp:1.1`, ...)

IMAGE TAGGING & NAMESPACING

// Tag on build:

```
docker image build -t myapp:1.0 .
```

// Retag an existing image:

```
docker image tag myapp:1.0 me/myapp:2.0
```

// Note `docker image tag` can set both tag and namespace.

// Names and tags are just pointers to image ID

// Image ID corresponds to immutable content addressable storage

SHARING IMAGES

// Docker Hub

- Provides certified commercial and free software distributed as Docker Images
- Shares community-generated images and content

EXERCISE: MANAGING IMAGES

Work through

// Managing Images

In the Exercises book.

19M 37S

IMAGE CREATION TAKEAWAYS

// Images are built out of read-only layers.

// Dockerfiles specify image layer contents.

// Key Dockerfile commands: `FROM`, `RUN`, `COPY` and `ENTRYPOINT`

// Images must be namespaced according to where you intend on sharing them.

FURTHER READING

// Best practices for writing Dockerfiles: http://dockr.ly/22WiJiO

// Use multi-stage builds: http://dockr.ly/2ewcUY3

// More about images, containers, and storage drivers: http://dockr.ly/1TuWndC

// Details on image layering: https://bit.ly/2AHX7iW

// Graphdriver plugins: http://dockr.ly/2eIVCab

// Docker Reference Architecture: An Intro to Storage Solutions for Docker CaaS: http://dockr.ly/2x8sBw2

// How to select a storage driver: http://dockr.ly/2eDu8yO

// Use the AUFS storage driver: http://dockr.ly/2jVc1Zz

// User guided caching in Docker: http://dockr.ly/2xKafPf

DOCKER VOLUMES

DISCUSSION: MANAGING DATA

If a container generates a lot of data, where should it be stored? What if you need to provision data to a container?

LEARNING OBJECTIVES

By the end of this module, learners will be able to

// Define a volume and identify its primary use cases

// Describe the advantages and potential security risks of mounting volumes and host directories into containers

VOLUME USECASES

Volumes provide a R/W path separate from the layered filesystem.

// Mount data at container startup

// Persist data when a container is deleted

// Share data between containers

// Speed up I/O by circumventing the union filesystem

BASIC VOLUMES

// **Named**: managed by Docker; filesystem independent; user-specified identifier

// **Anonymous**: managed by Docker; filesystem independent; randomly-generated identifier

// **Host mounted**: mount a specific path on the host; DIY management

INSTRUCTOR DEMO: VOLUMES

See the demo

// Basic Volume Usage

In the Exercises book.

VOLUMES IN DOCKERFILES

// VOLUME instruction creates a mount point

// Can specify arguments in a JSON array or string

// Cannot map volumes to host directories

// Volumes are initialized when the container is executed

```
FROM nginx:latest
...
# string example
VOLUME /myvolume

# string example with multiple volumes
VOLUME /www/website1 /www/website2

# JSON example
VOLUME ["myvol1", "myvol2"]
...
```

VOLUMES AND SECURITY

// Point of ingress to the host and other containers

// Don't mount things unnecessarily

// Use the `:ro` flag

// Linux: in-memory `tmpfs` mounts available

EXERCISE: VOLUMES USECASE

Work through

// Database Volumes

In the Exercises book.

19M 15S

DOCKER VOLUME TAKEAWAYS

// Volumes persist data beyond the container lifecycle

// Volumes bypass the copy on write system (better for write-heavy containers)

FURTHER READING

// How to use volumes: http://dockr.ly/2vRZBDG

// Troubleshoot volume errors: http://dockr.ly/2vyjvbP

// Docker volume reference: http://dockr.ly/2ewrlew

DOCKER SYSTEM COMMANDS

LEARNING OBJECTIVES

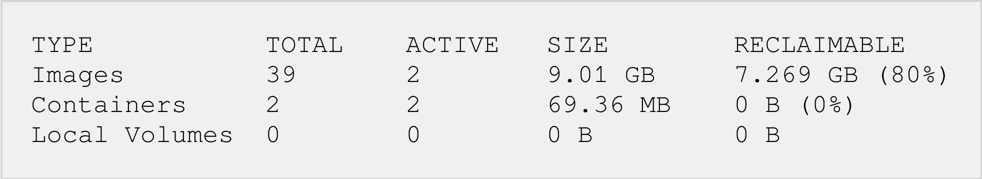
By the end of this module, learners will be able to

// Execute clean-up commands

// Locate Docker system information

CLEAN-UP COMMANDS

```
// docker system df
```



```
TYPE            TOTAL     ACTIVE    SIZE        RECLAIMABLE
Images          39        2         9.01 GB     7.269 GB (80%)
Containers      2         2         69.36 MB    0 B (0%)
Local Volumes   0         0         0 B         0 B
```

```
// docker system prune
```

more limited...

```
// docker image prune [--filter "foo=bar"]

// docker container prune [--filter "foo=bar"]

// docker volume prune [--filter "foo=bar"]

// docker network prune [--filter "foo=bar"]
```

INSPECT THE SYSTEM

`docker system info`

```
Containers: 2
 Running: 2
 Paused: 0
 Stopped: 0
Images: 105
Server Version: 17.03.0-ee
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay
Swarm: active
 NodeID: ybmqksh6fm627armruq0e8id1
 Is Manager: true
 ClusterID: 2rbf1dv6t5ntro2fxbry6ikr3
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  ...
```

SYSTEM EVENTS

Start observing with ...

`docker system events`

Generate events with ...

`docker container run --rm alpine echo 'Hello World!'`

```
2017-01-25T16:57:48.553596179-06:00 container create 30eb630790d44052f26c1081...
2017-01-25T16:57:48.556718161-06:00 container attach 30eb630790d44052f26c1081...
2017-01-25T16:57:48.698190608-06:00 network connect de1b2b40f522e69318847ada3...
2017-01-25T16:57:49.062631155-06:00 container start 30eb630790d44052f26c1081d...
2017-01-25T16:57:49.164526268-06:00 container die 30eb630790d44052f26c1081dbf...
2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b40f522e69318847a...
2017-01-25T16:57:49.815845051-06:00 container destroy 30eb630790d44052f26c108...
```

EXERCISE: SYSTEM COMMANDS

Work through

// Cleaning up Docker Resources

// Inspection Commands

in the Exercises book.

18M 57S

DISCUSSION

// What is the origin of dangling image layers?

// What are some potential pitfalls to automating system cleanup with prune commands, and how to avoid them?

// Questions?

FURTHER READING

// System commands reference: http://dockr.ly/2eMR53i

CONTAINERIZATION FUNDAMENTALS CONCLUSION: ANY APP, ANYWHERE.

// Containers are isolated processes

// Images provide filesystem for containers

// Volumes persist data

WRAP UP – SPRING BOOT

EXERCISE INSTRUCTIONS

// Goal: Build a docker image that runs a Java application

// Clone the git repository at https://github.com/ckaserer/java-helloworld

// Build the application using `gradle bootJar` and find the resulting fat-jar file at `build/libs` (already done for you).

// You can run the java application locally with `java -jar rest-service-0.0.1-SNAPSHOT.jar`, this starts an application server on port 8080. Check it in your browser via http://127.0.0.1:8080/greeting (respectively with your AWS instance public dns).

18M 47S

SOLUTION HINTS

// Build the application outside of the container (already done and checked into git).

// Find a suitable docker base image, that has a JRE installation.

// Use both the `CMD` and the `ENTRYPOINT` instructions.

// Run two instances of the container – what do you have to take care of? Make sure both instances are reachable via browser.

18M 44S

SOLUTION

Sample Dockerfile

```
FROM anapsix/alpine-java
LABEL MAINTAINER=clemens.kaserer@gepardec.com
WORKDIR /data
EXPOSE 8080
COPY build/libs/rest-service-0.0.1-SNAPSHOT.jar \
     rest-service-0.0.1-SNAPSHOT.jar
CMD ["-jar", "rest-service-0.0.1-SNAPSHOT.jar"]
ENTRYPOINT ["java"]
```

Solution Commands

// docker build -t spring_boot_example .

// docker run –d –p 80:8080 spring_boot_example

CONTAINERIZATION TRAINING

Please take our feedback survey

Get in touch: office@gepardec.com

https://www.gepardec.com/trainings

DOCKER NETWORKING BASICS

DISCUSSION: PORTABLE NETWORKS

Network traffic must by definition traverse a network outside its originating container. How can we make inter-container communication as portable and secure as containers themselves?

LEARNING OBJECTIVES

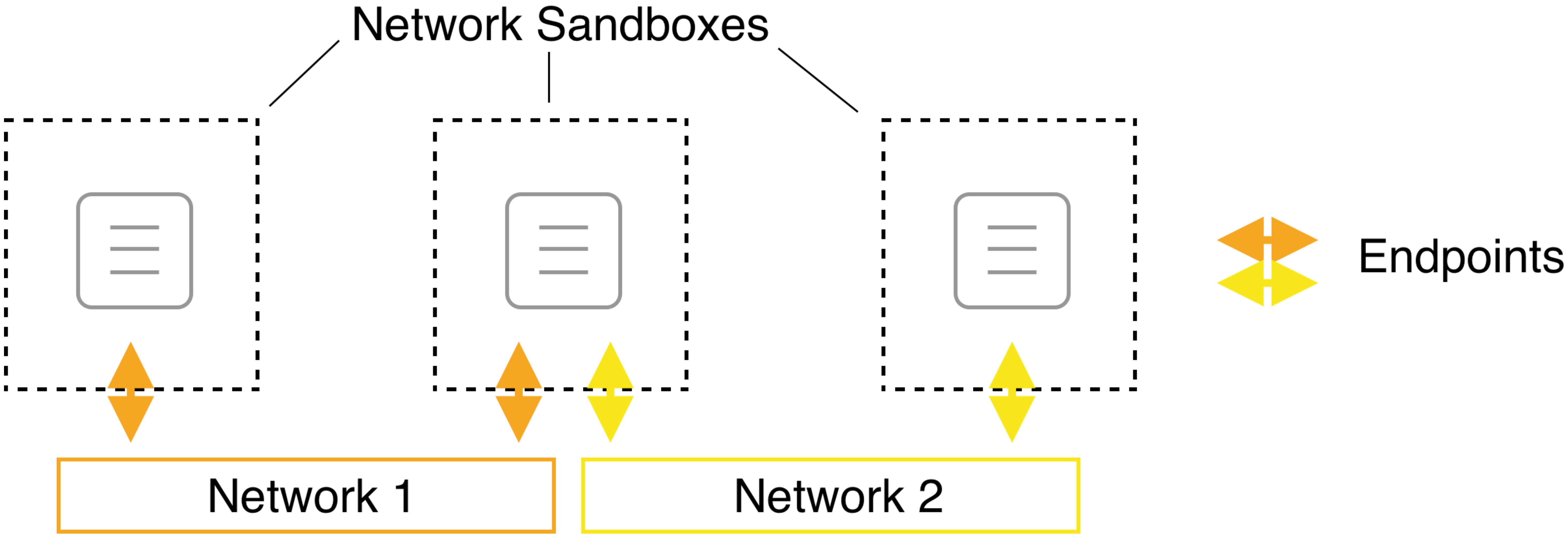
By the end of this module, learners will be able to

// Describe Docker's container network model and its security implications

// Describe the basic technologies that underwrite single host networks

// Understand how Docker manipulates a host's firewall rules to control container traffic

THE CONTAINER NETWORK MODEL



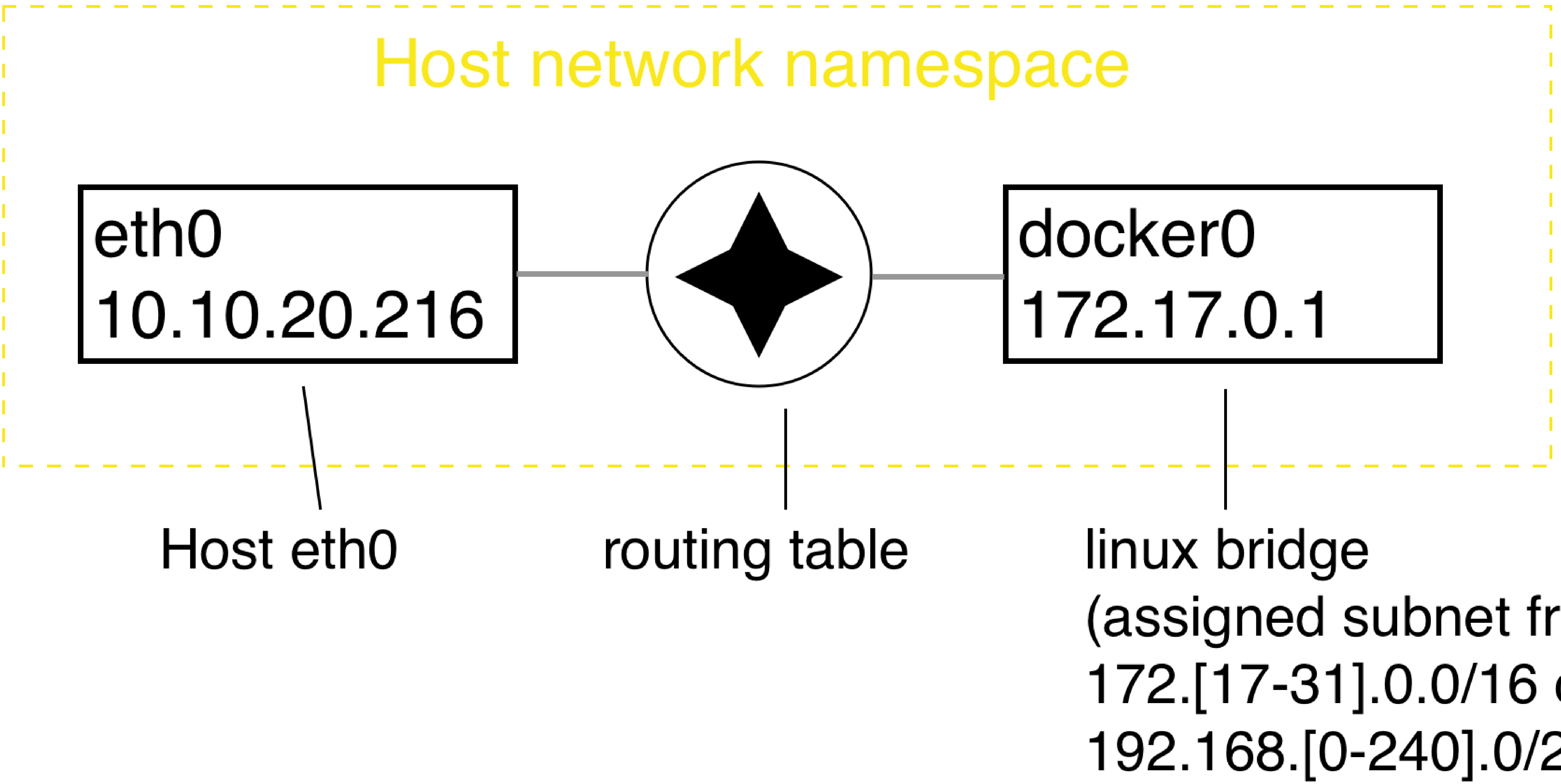
Network Sandboxes

Endpoints

Network 1

Network 2

LINUX: DEFAULT SINGLE-HOST NETWORK

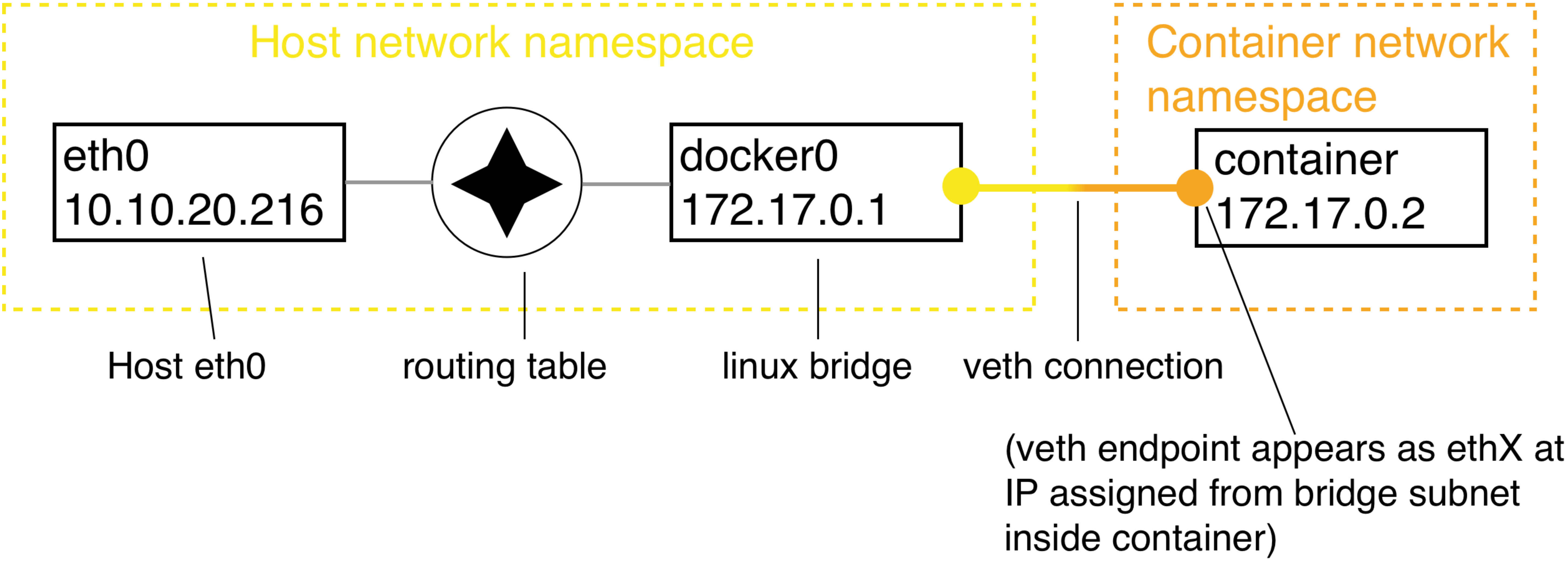


Host eth0

routing table

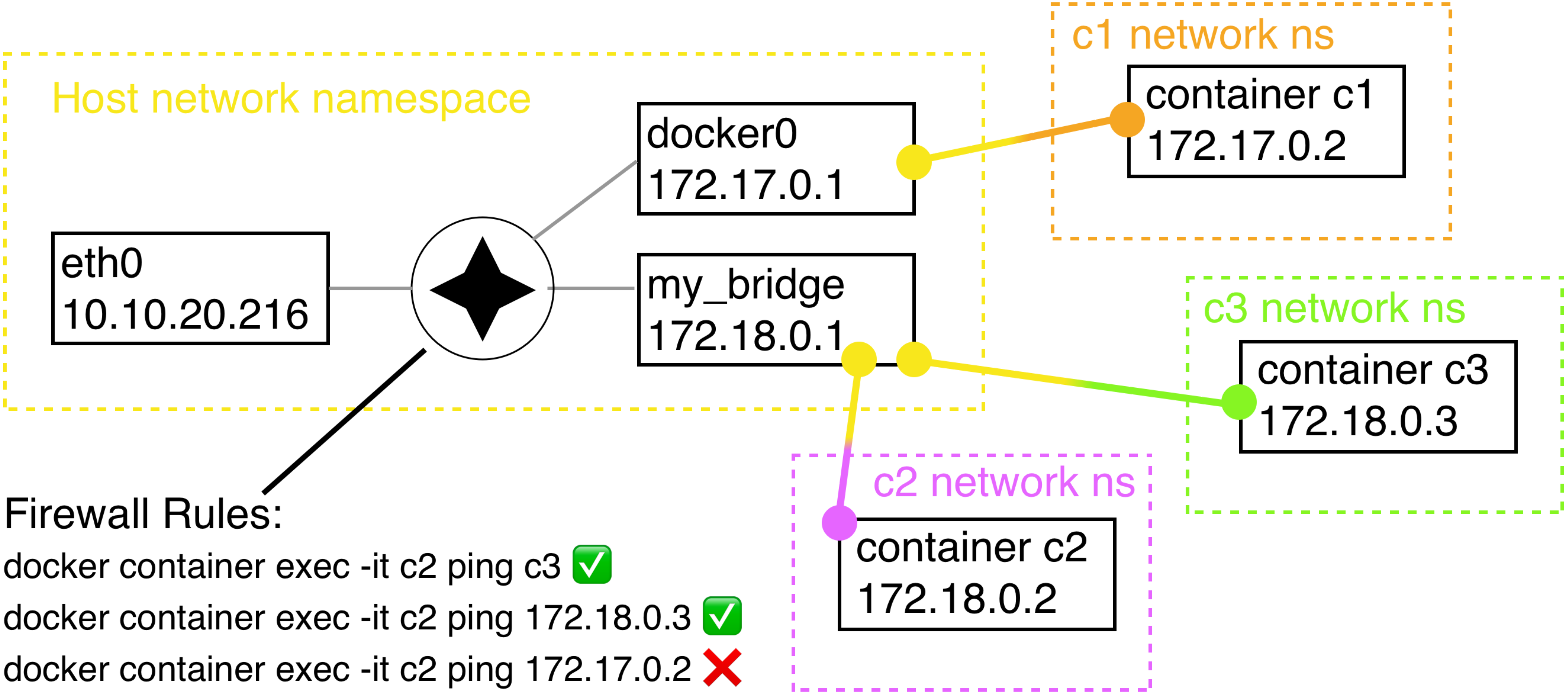
linux bridge
(assigned subnet from
172.[17-31].0.0/16 or
192.168.[0-240].0/20)

LINUX: DEFAULT CONTAINER NETWORKING



Quiz: identify the sandbox, endpoint and network corresponding to the container networking model objects in this diagram.

LINUX: USER-DEFINED BRIDGES & FIREWALLS



Firewall Rules:

docker container exec -it c2 ping c3 ✅

docker container exec -it c2 ping 172.18.0.3 ✅

docker container exec -it c2 ping 172.17.0.2 ❌

EXPOSING CONTAINER PORTS

// Containers have no public IP address by default.

// Can forward host port -> container port

// Mapping created manually or automatically.

// Port mappings visible via
`docker container ls` or
`docker container port`

INSTRUCTOR DEMO: SINGLE HOST NETWORKS

See the demo

// Single Host Networks

In the Exercises book.

EXERCISE: SINGLE HOST NETWORKS

Work through

// Introduction to Container Networking

// Container Port Mapping

in the Exercises book.

18M 17S

DOCKER NETWORKING TAKEAWAYS

// Single host networks follow the container networking model:

- Sandbox: Network namespaces
- Endpoint: veth (linux)
- Network: bridge (linux)

// Containers resolve each other by DNS lookup when explicitly named and attached to custom networks

// Docker software defined networks are firewalled from each other by default

FURTHER READING

// Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks: https://dockr.ly/2q3O8jq

// Network containers: http://dockr.ly/2x1BYgW

// Docker container networking: http://dockr.ly/1QnT6y8

// Understand container communication: http://dockr.ly/2iSrHOO

INTRODUCTION TO DOCKER COMPOSE

DISCUSSION: PROCESSES VS. APPLICATIONS

Containers and images describe individual processes. What will we need to describe entire applications?

LEARNING OBJECTIVES

By the end of this module, learners will be able to

// Design scalable Docker services

// Leverage Docker's built in service discovery mechanism

// Write a compose file describing an application

DISTRIBUTED APPLICATION ARCHITECTURE

// Applications consisting of one or more containers across one or more nodes

// Docker Compose facilitates multi-container design **on a single node**

DOCKER SERVICES

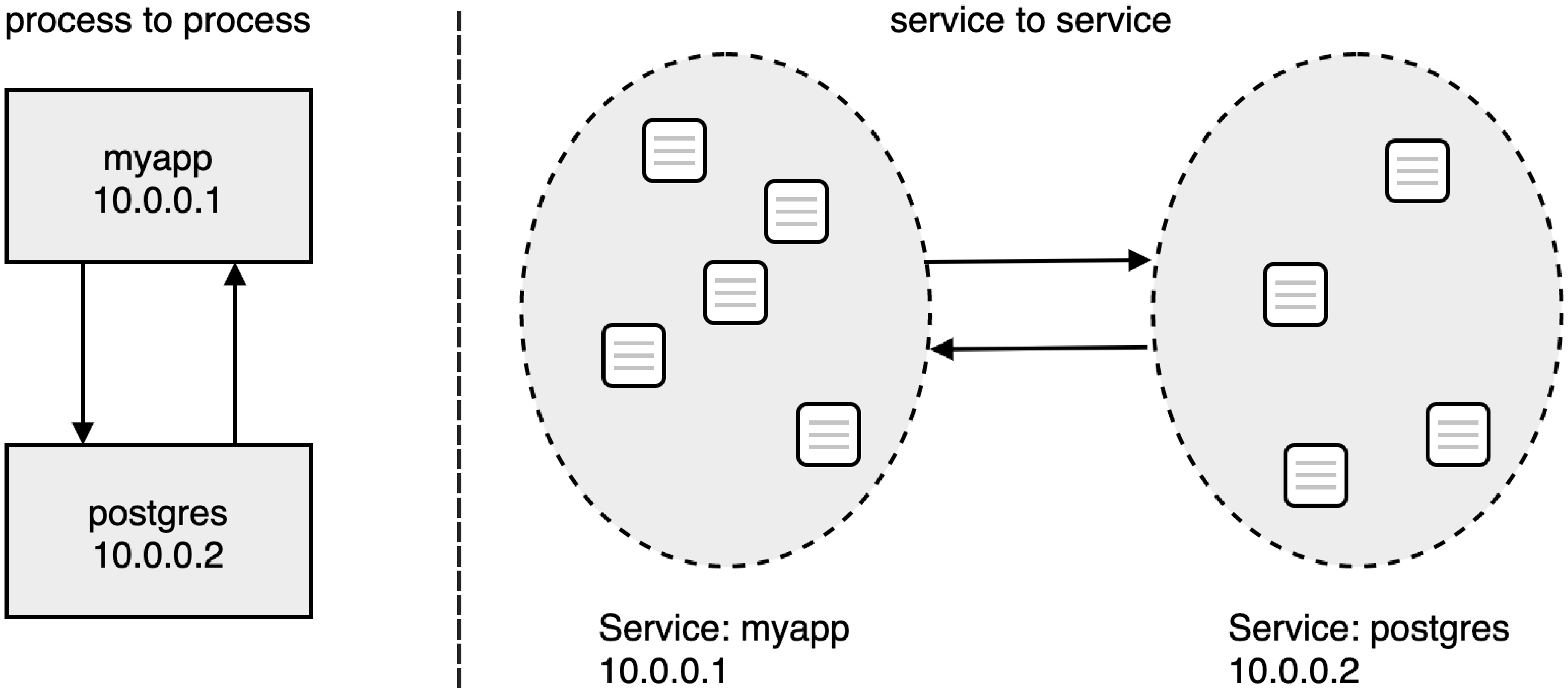
// Goal: declare and (re)configure many similar containers all at once

// Goal: scale apps by adding containers seamlessly

// A **service** defines the **desired state** of a group of identically configured containers

// Docker provides **transparent service discovery** for Services

SERVICE DISCOVERY



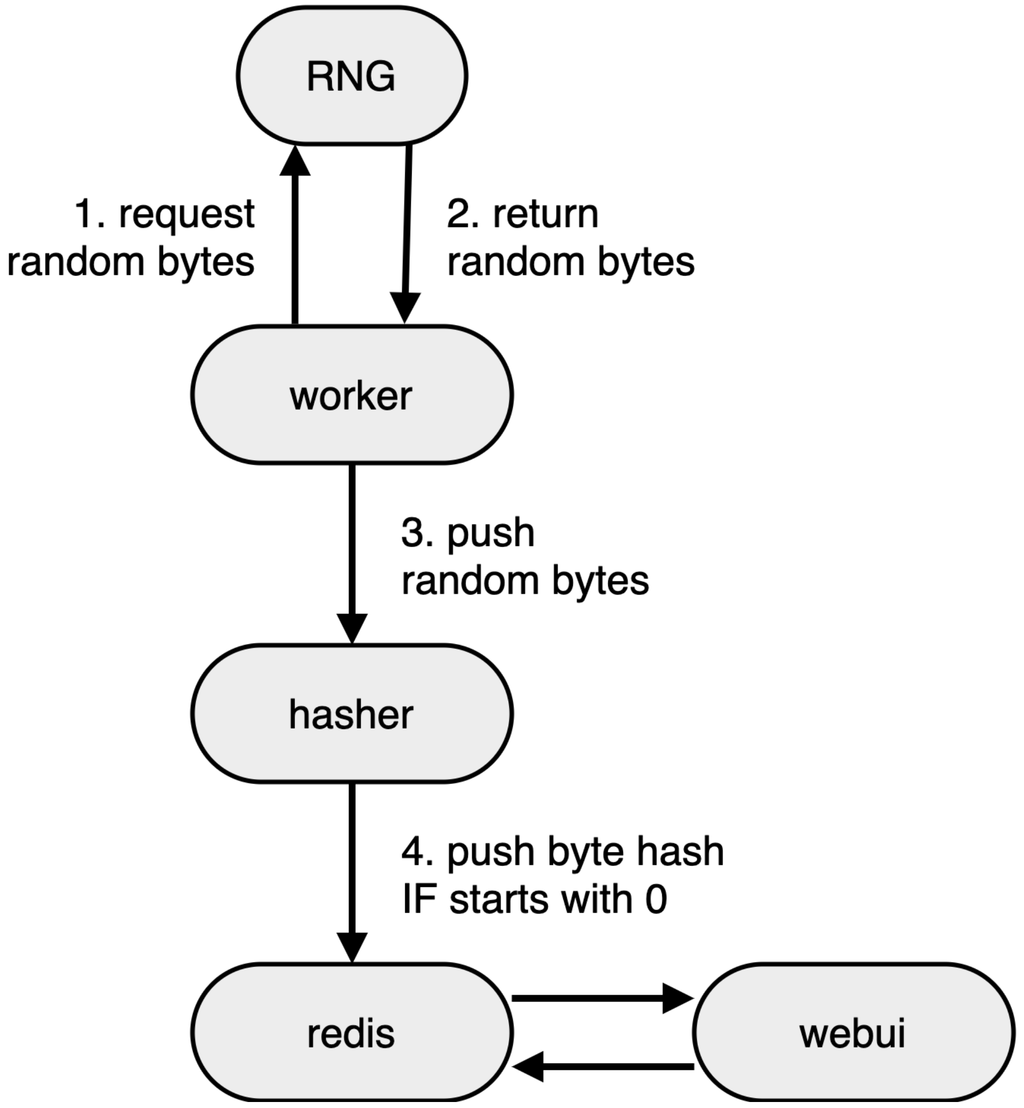
Services are assigned a Virtual IP which spreads traffic out across the underlying containers automatically.

OUR APPLICATION: DOCKERCOINS

(DockerCoins 2016 logo courtesy of @XtlCnslt and @ndeloof. Thanks!)

// It is a DockerCoin miner! 💰🐳📦🚢

// Dockercoins consists of 5 services working together:



RNG

1. request random bytes

2. return random bytes

worker

3. push random bytes

hasher

4. push byte hash IF starts with 0

redis

webui

INSTRUCTOR DEMO: DOCKER COMPOSE

See the demo

// Docker Compose

In the Exercises book.

EXERCISE: COMPOSE APPS

Work through

- // Starting a Compose App
- // Scaling a Compose App

in the Exercises book.

17M 54S

DOCKER COMPOSE TAKEAWAYS

// Docker Compose makes single node orchestration easy

// Compose services makes scaling applications easy

// Bottleneck identification important

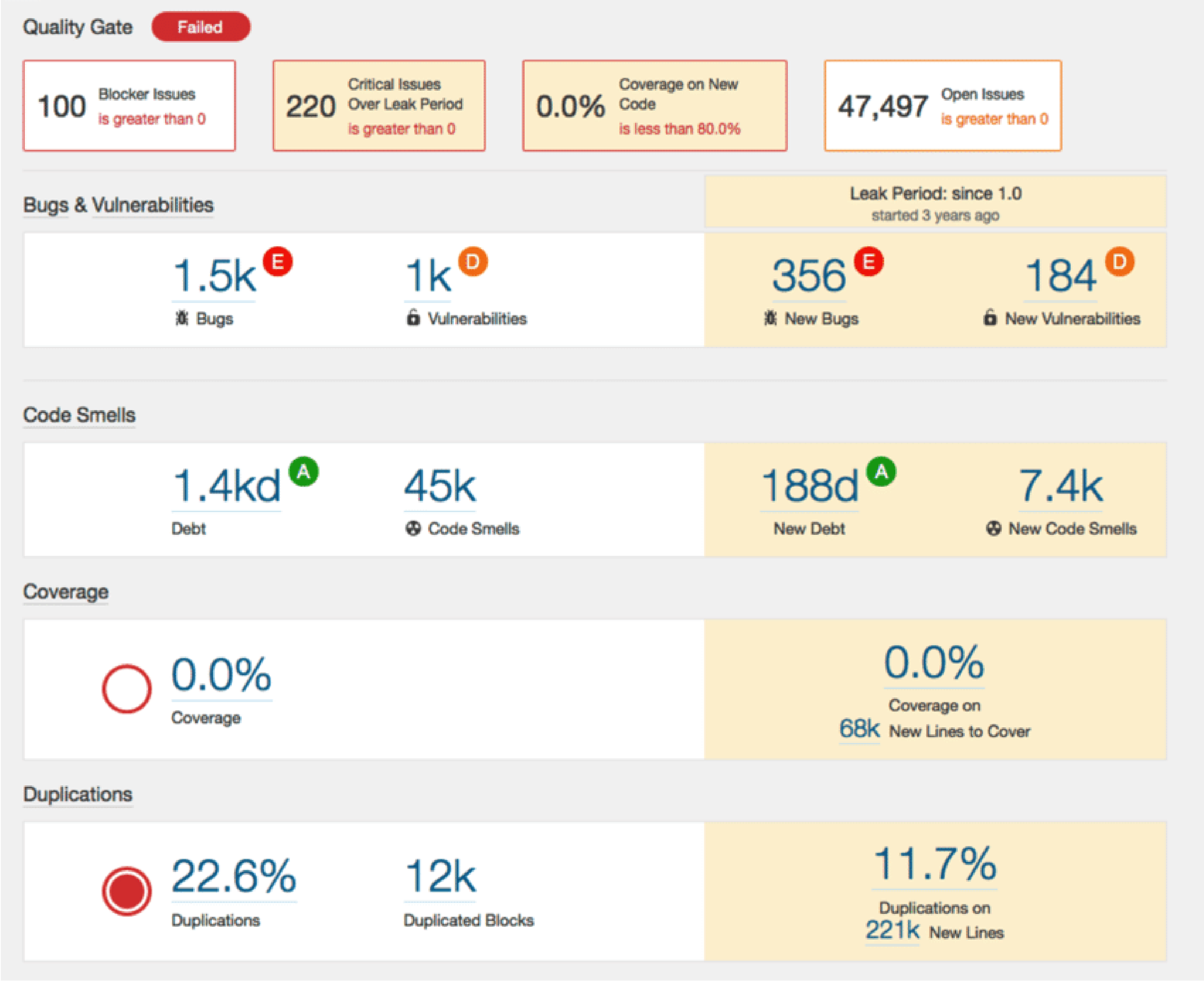
// Syntactically: `docker-compose.yml` + API

FURTHER READING

- // Docker compose examples: http://dockr.ly/1FL2VQ6
- // Overview of docker-compose CLI: http://dockr.ly/2wtQlZT
- // `docker-compose.yaml` reference: http://dockr.ly/2iHUpeX
- // Docker Compose and Windows: http://bit.ly/2watrqk

WRAP UP DOCKER-COMPOSE - SONARQUBE

SONARQUBE



Quality Gate Failed

100 Blocker Issues is greater than 0

220 Critical Issues Over Leak Period is greater than 0

0.0% Coverage on New Code is less than 80.0%

47,497 Open Issues is greater than 0

Bugs & Vulnerabilities

Leak Period: since 1.0
started 3 years ago

1.5k E Bugs

1k D Vulnerabilities

356 E New Bugs

184 D New Vulnerabilities

Code Smells

1.4kd A Debt

45k Code Smells

188d A New Debt

7.4k New Code Smells

Coverage

0.0% Coverage

0.0% Coverage on 68k New Lines to Cover

Duplications

22.6% Duplications

12k Duplicated Blocks

11.7% Duplications on 221k New Lines

EXERCISE INSTRUCTIONS

// Setup a Sonarqube server that listens on port 9000

// Connect it to a persistent database
i.e. if you , docker rm –f' your Sonarqube container and run a new one, no data is lost

// Use postgresql and persist it's data on the host filesystem using volumes

// Verify e.g. by creating a user via Sonarqube UI, remove the container and run a new one – is the user still present?

// Check that Sonarqube is really using your postgresql database

// **Hint**: use docker-compose

17M 43S

SOLUTION

```
version: "2"
services:
  sonarqube:
    image: sonarqube
    ports:
      - "9000:9000"
    networks:
      - sonarnet
    environment:
      - SONARQUBE_JDBC_URL=jdbc:postgresql://db:5432/sonar
    volumes:
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_bundled-plugins:/opt/sonarqube/lib/bundled-plugins

  db:
    image: postgres
    networks:
      - sonarnet
    environment:
      - POSTGRES_USER=sonar
      - POSTGRES_PASSWORD=sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/datanetworks:
    sonarnet:
      driver: bridge

volumes:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_bundled-plugins:
  postgresql:
  postgresql_data:
```