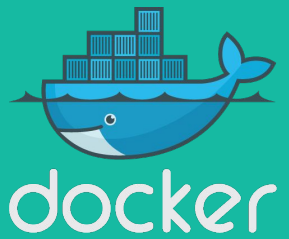
Docker 101 Workshop



About Your Instructors

Agenda

Section 1: **(One Hour)**

What is Docker / What is Docker Not

Basic Docker Commands

Dockerfiles

PWD: Hello World

PWD: First Alpine Image

PWD: Static website

Section 2: (30 minutes)

Anatomy of a Docker image

Docker volumes

Volume use cases

PWD: Docker Volumes

Break (15 minutes)

Section 3: (45 minutes)

Networking

Docker Swarm

PWD: Swarm mode introduction

Section 4: (30 Minutes)

Docker compose / stacks

Secrets

PWD: Swarm stack introduction

PWD: Docker Compose with Secrets

Before we get started

All hands on portions are done via “Play With Docker”

Visit:

http://training.play-with-docker.com

Do the “hello-world” exercise to make sure everything’s copasetic

Section 1:

What is Docker
Basic Docker Commands
Dockerfiles

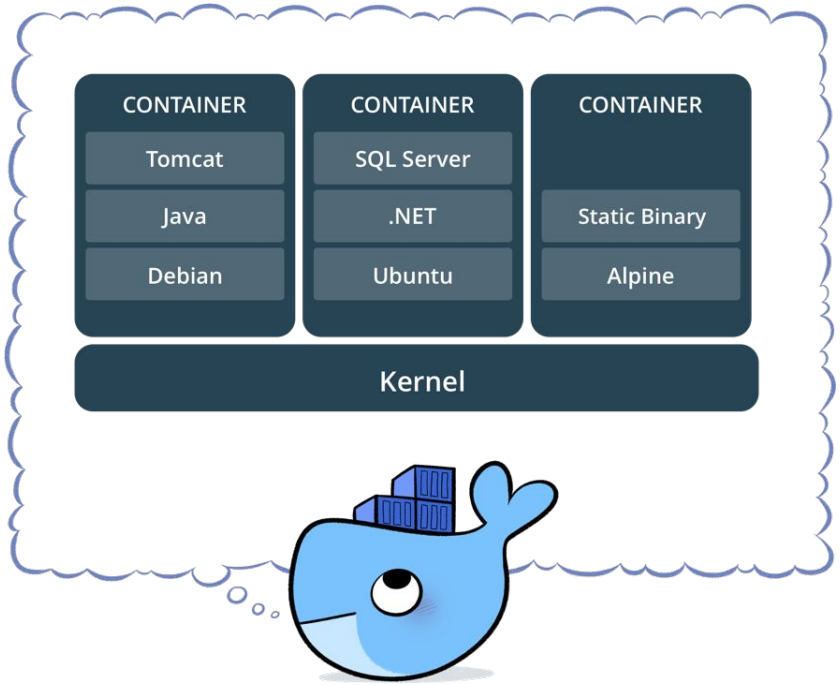
Docker containers are NOT VMs

- Easy connection to make
- Fundamentally different architectures
- Fundamentally different benefits

VMs

Containers

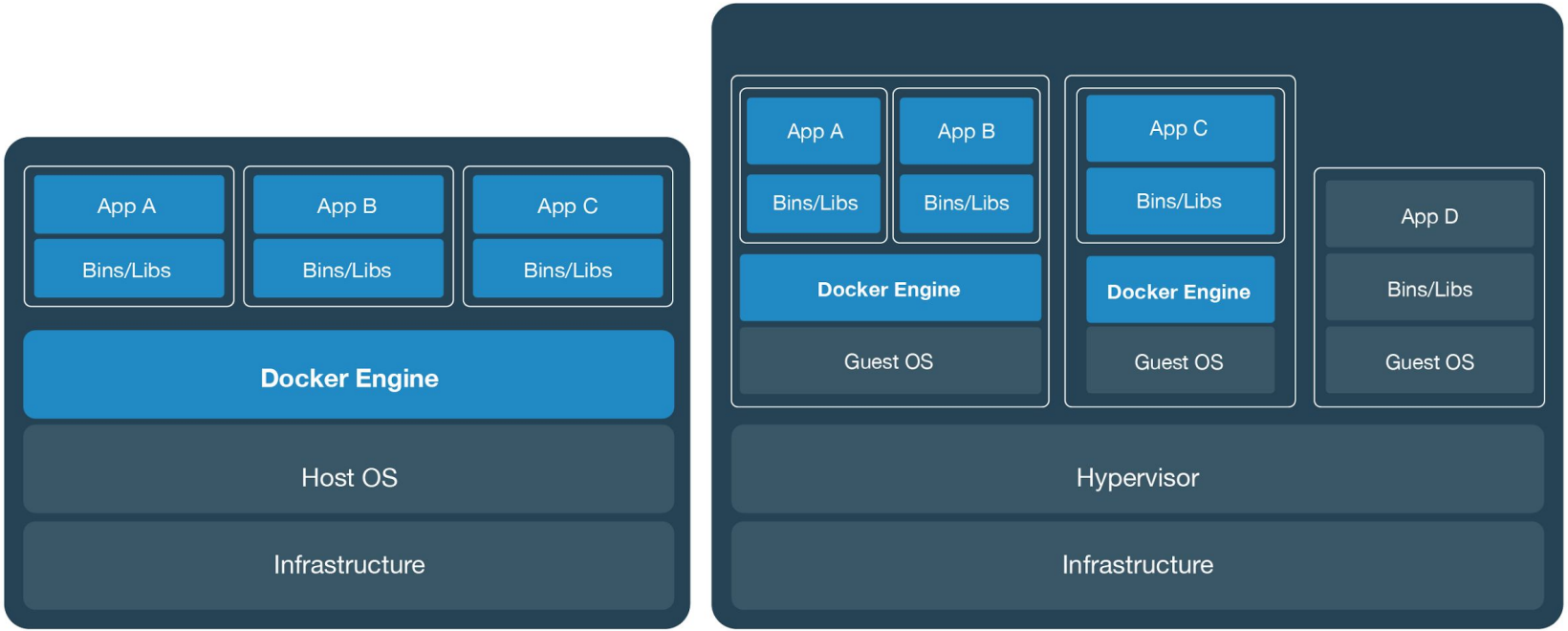
What is a container?



- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works for all major Linux distributions
- Containers native to Windows Server 2016

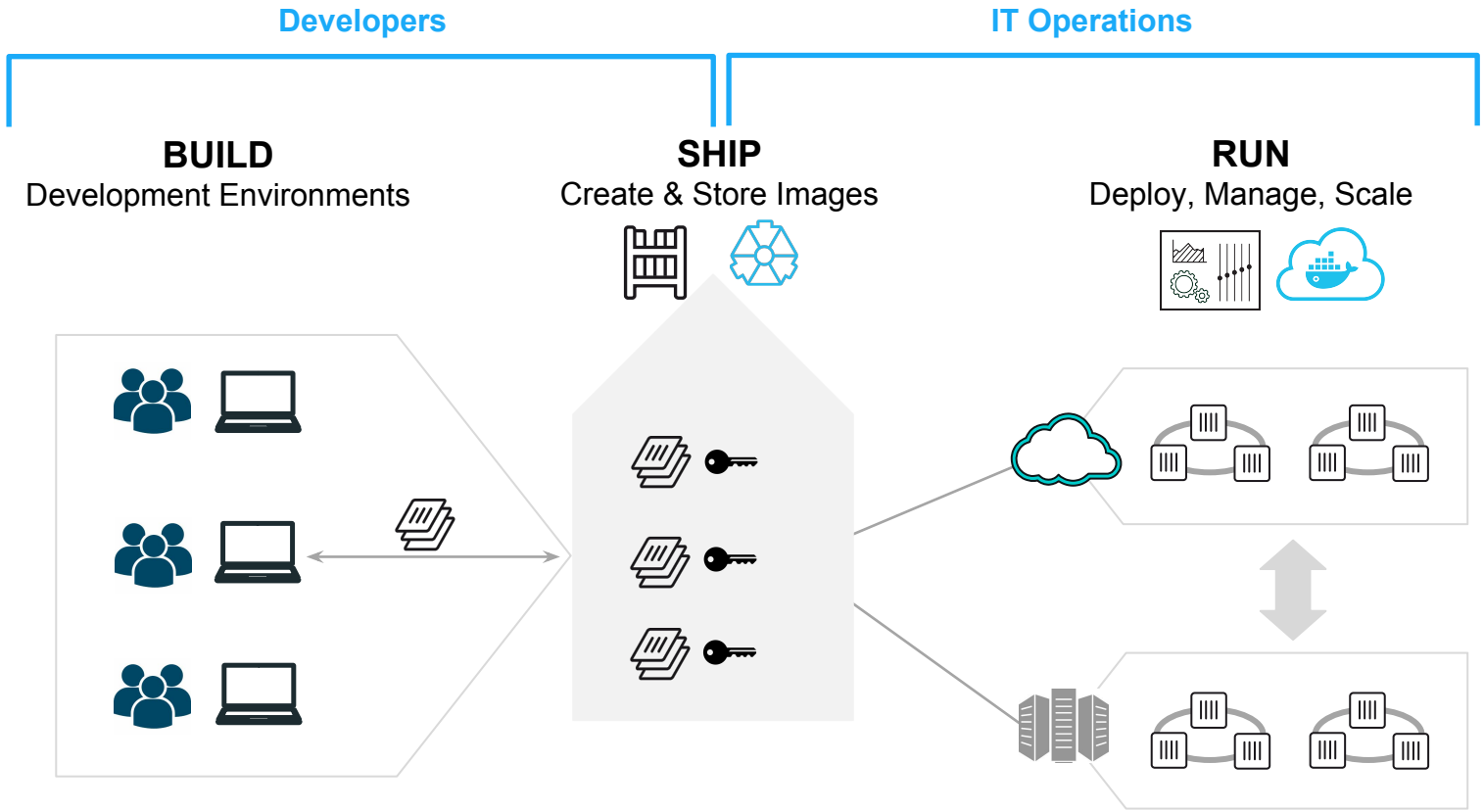
docker

They’re different, not mutually exclusive



docker

Using Docker: Build, Ship, Run Workflow

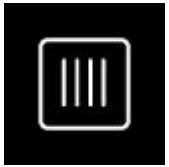


docker

Some Docker vocabulary

Docker Image

The basis of a Docker container. Represents a full application



Docker Container

The standard unit in which the application service resides and executes

Docker Engine

Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

Registry Service (Docker Hub or Docker Trusted Registry)

Cloud or server based storage and distribution service for your images

Basic Docker Commands

```
$ docker image pull mikegcoleman/catweb:latest

$ docker image ls
$ docker container run –d –p 5000:5000 –-name catweb mikegcoleman/catweb:latest

$ docker container ps
$ docker container stop catweb (or <container id>)

$ docker container rm catweb (or <container id>)

$ docker image rm mikegcoleman/catweb:latest (or <image id>)

$ docker build –t mikegcoleman/catweb:2.0 .

$ docker image push mikegcoleman/catweb:2.0
```

Dockerfile – Linux Example

```
1  # our base image
2  FROM alpine:latest
3
4  # Install python and pip
5  RUN apk add --update py-pip
6
7  # upgrade pip
8  RUN pip install --upgrade pip
9
10 # install Python modules needed by the Python app
11 COPY requirements.txt /usr/src/app/
12 RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
13
14 # copy files required for the app to run
15 COPY app.py /usr/src/app/
16 COPY templates/index.html /usr/src/app/templates/
17
18 # tell the port number the container should expose
19 EXPOSE 5000
20
21 # run the application
22 CMD ["python", "/usr/src/app/app.py"]
```

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Dockerfile – Windows Example

19 lines (15 sloc) | 832 Bytes — Raw | Blame | History

```
1  FROM microsoft/windowsservercore
2
3  ENV NPM_CONFIG_LOGLEVEL info
4  ENV NODE_VERSION 6.5.0
5  ENV NODE_SHA256 0c0962800916c7104ce6643302b2592172183d76e34997823be3978b5ee34cf2
6
7  RUN powershell -Command \
8      $ErrorActionPreference = 'Stop' ; \
9      (New-Object System.Net.WebClient).DownloadFile('https://nodejs.org/dist/v%NODE_VERSION%/node-v%NODE_VERSION%-win-x64.zip',
10     if ((Get-FileHash node.zip -Algorithm sha256).Hash -ne $env:NODE_SHA256) {exit 1} ; \
11     Expand-Archive node.zip -DestinationPath C:\ ; \
12     Rename-Item 'C:\node-v%NODE_VERSION%-win-x64' 'C:\nodejs' ; \
13     New-Item '%APPDATA%\npm' ; \
14     $env:PATH = 'C:\nodejs;%APPDATA%\npm;' + $env:PATH ; \
15     [Environment]::SetEnvironmentVariable('PATH', $env:PATH, [EnvironmentVariableTarget]::Machine) ; \
16     Remove-Item -Path node.zip
17
18 CMD [ "node.exe" ]
```

Hands On Exercises

First Alpine Image
Static Website

http://training.play-with-docker.com

Section 2:

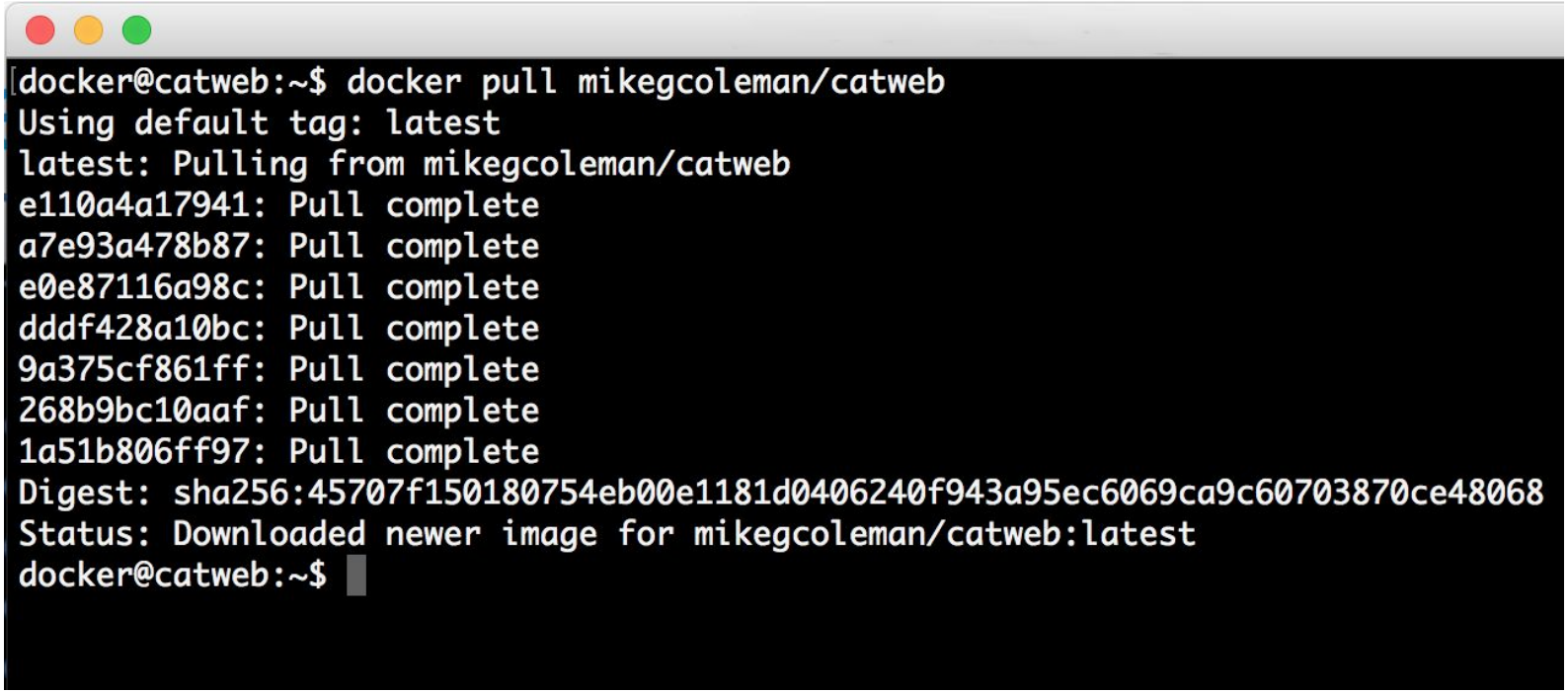
Anatomy of a Docker Container
Docker Volumes
Volume Use Cases

Let’s Go Back to Our Dockerfile

```
1  # our base image
2  FROM alpine:latest
3
4  # Install python and pip
5  RUN apk add --update py-pip
6
7  # upgrade pip
8  RUN pip install --upgrade pip
9
10 # install Python modules needed by the Python app
11 COPY requirements.txt /usr/src/app/
12 RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
13
14 # copy files required for the app to run
15 COPY app.py /usr/src/app/
16 COPY templates/index.html /usr/src/app/templates/
17
18 # tell the port number the container should expose
19 EXPOSE 5000
20
21 # run the application
22 CMD ["python", "/usr/src/app/app.py"]
```

Each Dockerfile Command Creates a Layer

Docker Image Pull: Pulls Layers



Layers on the Physical Disk

- Logical file system by grouping different file system primitives into branches (directories, file systems, subvolumes, snapshots)
- Each branch represents a layer in a Docker image
- Containers will share common layers on the host
- Allows images to be constructed / deconstructed as needed vs. a huge monolithic image (ala traditional virtual machines)
- When a container is started a writeable layer is added to the “top” of the file system

Copy on Write

Super efficient:

- Sub second instantiation times for containers
- New container can take <1 Mb of space

Containers appears to be a copy of the original image

But, it is really just a link to the original shared image

If someone writes a change to the file system, a copy of the affected file/directory is “copied up”

Docker Volumes

- Volumes mount a directory on the host into the container at a specific location

```
$ docker volume create hello
hello
$ docker run -d -v hello:/world busybox ls /world
```

- Can be used to share (and persist) data between containers
  - Directory persists after the container is deleted
    - Unless you explicitly delete it

- Can be created in a Dockerfile or via CLI

Why Use Volumes

- Mount local source code into a running container

```
docker container run -v $(pwd):/usr/src/app/
mikegcoleman/catweb
```

- Improve performance
  - As directory structures get complicated traversing the tree can slow system performance
- Data persistence

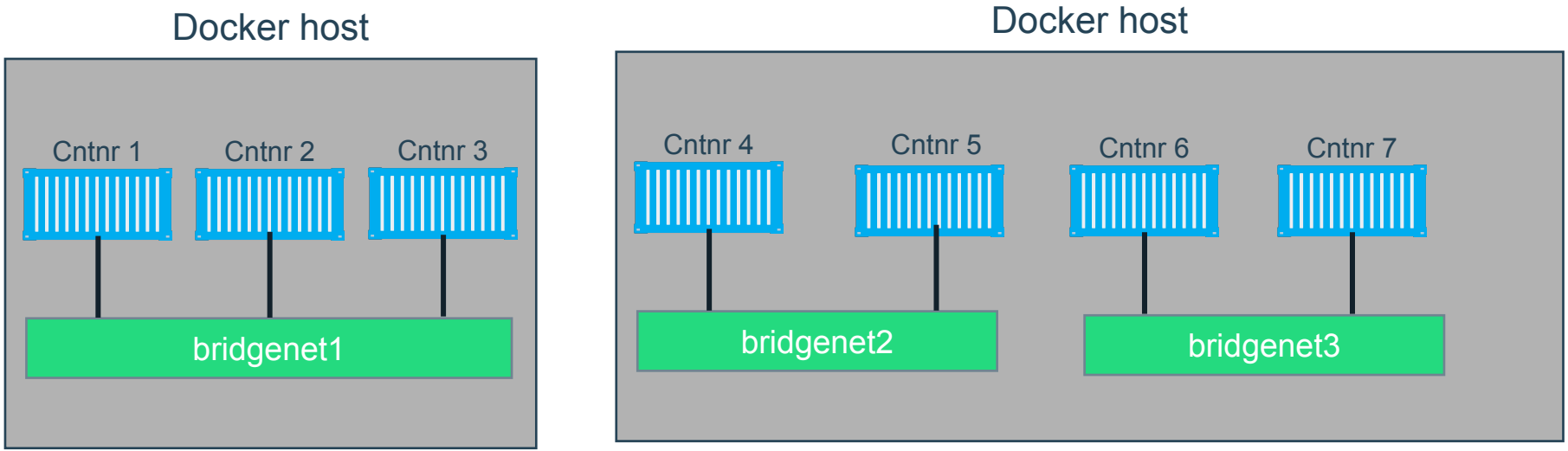
Hands On Exercises (and Break)

Docker Volumes

http://training.play-with-docker.com

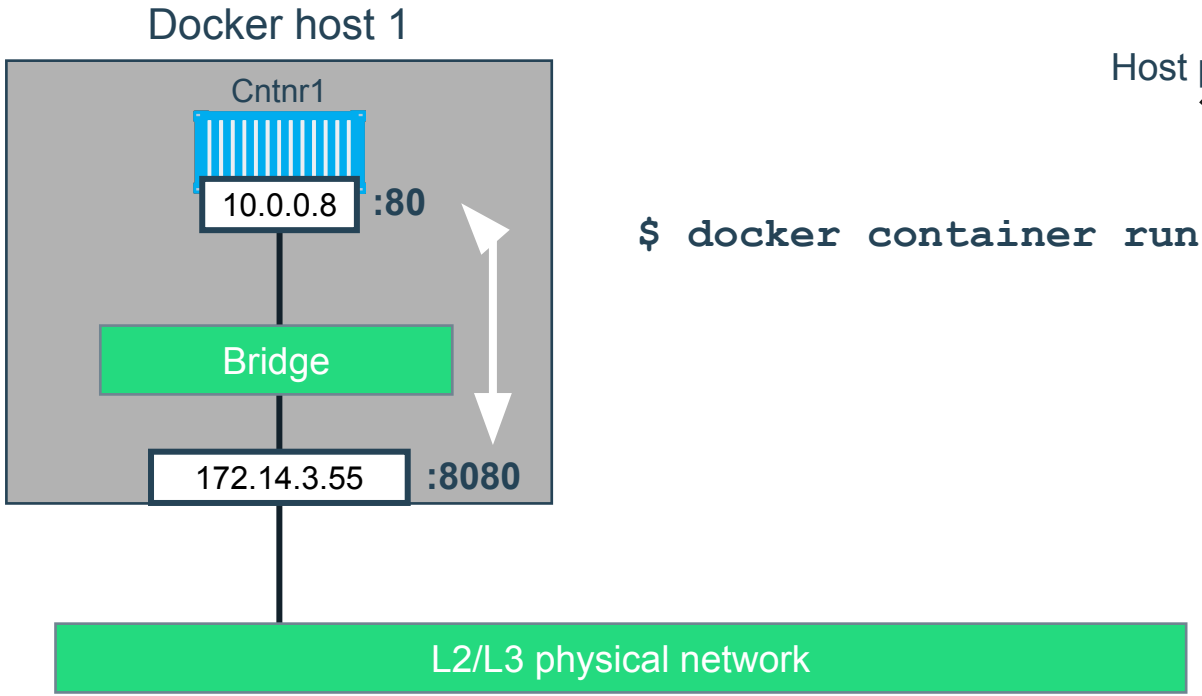
Section 3:
Networking
Docker Swarm

What is Docker Bridge Networking




```
docker network create -d bridge --name bridgenet1
```

Docker Bridge Networking and Port Mapping



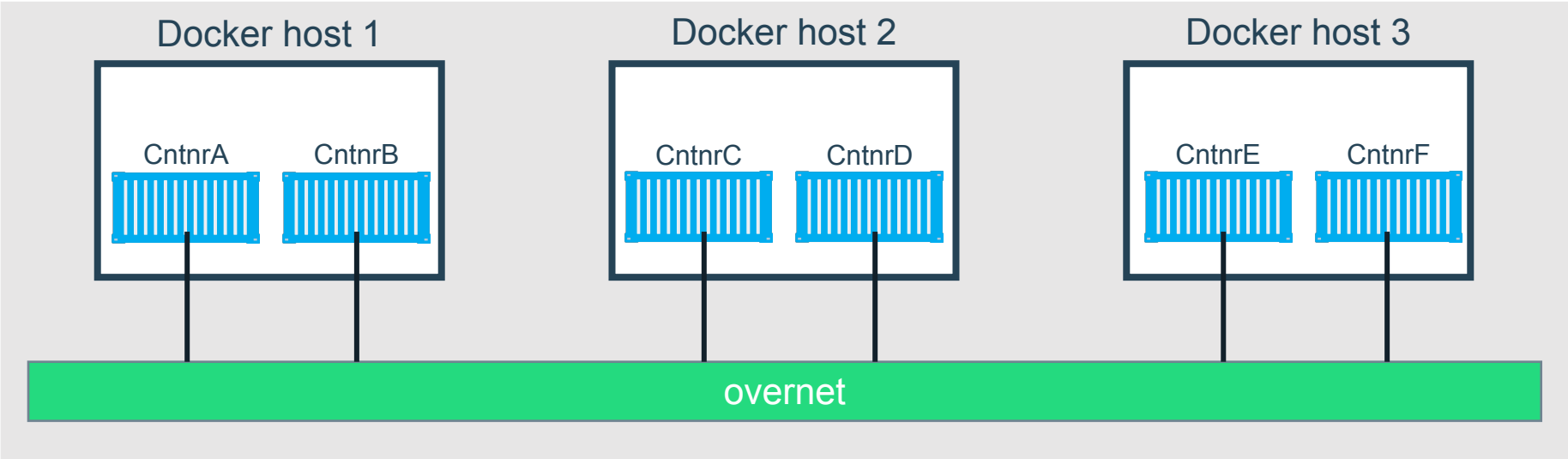
Host port

Container port

```
$ docker container run -p 8080:80 ...
```

What is Docker Overlay Networking

The **overlay** driver enables simple and secure **multi-host** networking



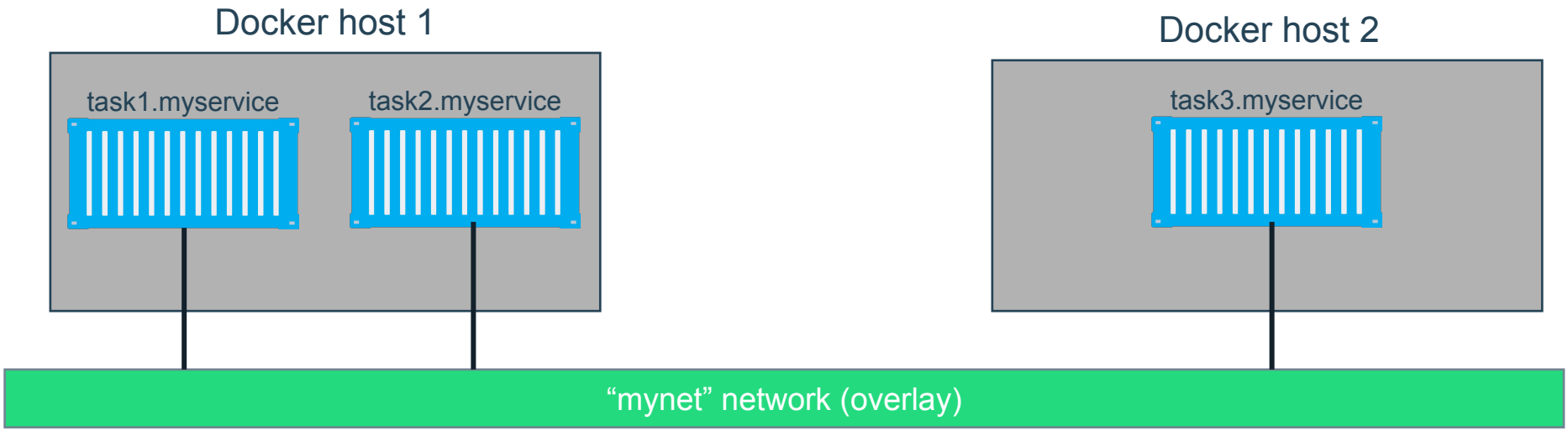

```
docker network create -d overlay --name overnet
```

What is Service Discovery

The ability to discover services within a Swarm

- Every **service** registers its name with the Swarm
- Every **task** registers its name with the Swarm
- Clients can lookup service **names**
- Service discovery uses the DNS resolver embedded inside each container and the DNS server inside of each Docker Engine

Service Discovery Big Picture



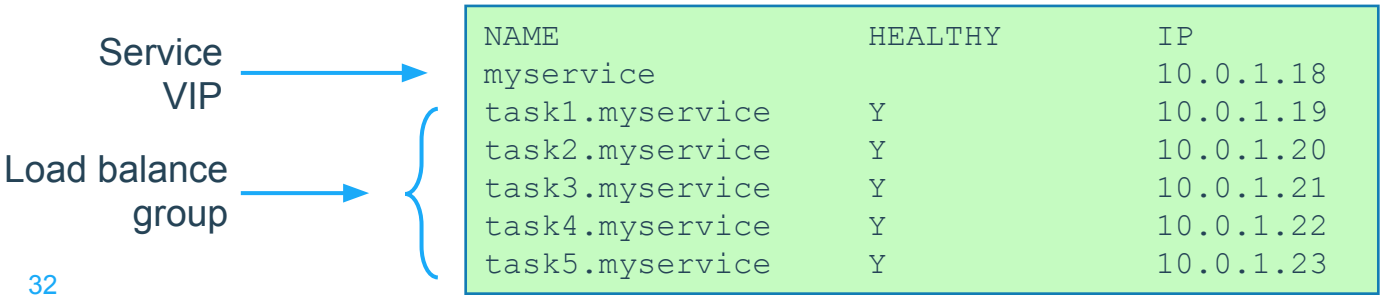

```
task1.myservice      10.0.1.19
task2.myservice      10.0.1.20
task3.myservice      10.0.1.21
myservice            10.0.1.18
```

Swarm DNS (service discovery)

docker

Service Virtual IP (VIP) Load Balancing

- Every **service** gets a **VIP** when it's created
  - This stays with the service for its entire life
- Lookups against the VIP get load-balanced across all **healthy tasks** in the service
- Behind the scenes it uses Linux kernel **IPVS** to perform transport layer load balancing
- `docker service inspect <service>` (shows the service VIP)



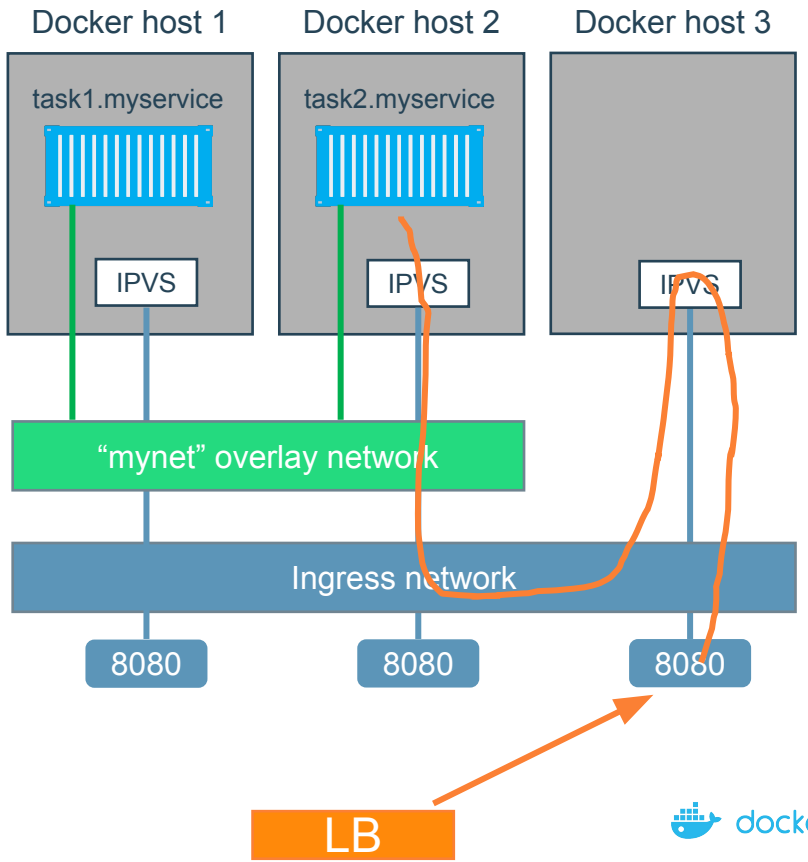
What is the Routing Mesh

Native load balancing of requests coming from an external source

- Services get published on a single port across the entire Swarm
- Incoming traffic to the published port can be handled by all Swarm nodes
- A special overlay network called “**Ingress**” is used to forward the requests to a task in the service
- Traffic is internally load balanced as per normal service VIP load balancing

Routing Mesh Example

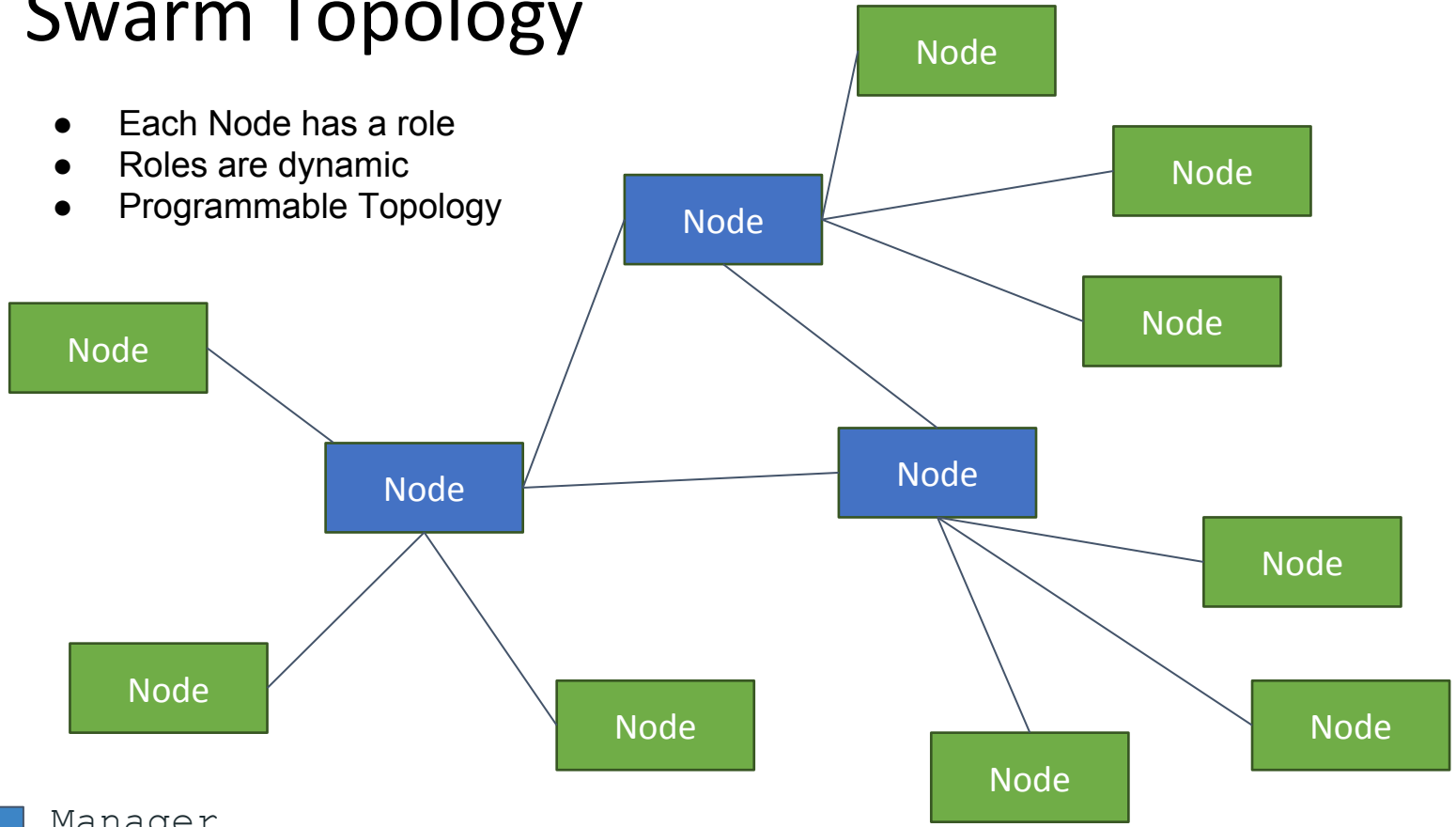
1. Three Docker hosts
2. New service with 2 tasks
3. Connected to the **mynet** overlay network
4. Service published on port 8080 swarm-wide
5. External LB sends request to Docker host 3 on port 8080
6. Routing mesh forwards the request to a healthy task using the ingress network



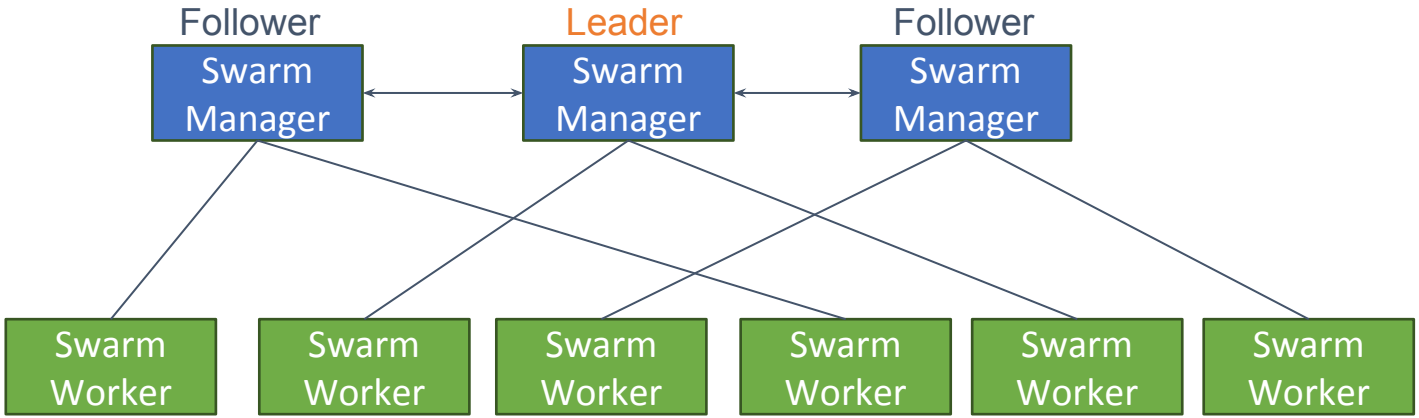
docker

Swarm Topology

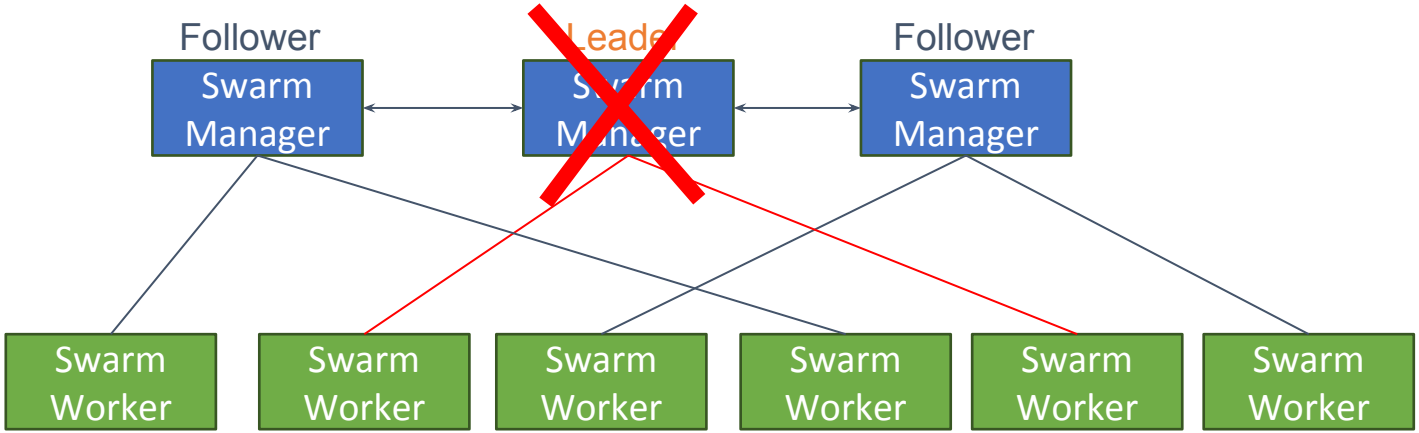
- Each Node has a role
- Roles are dynamic
- Programmable Topology



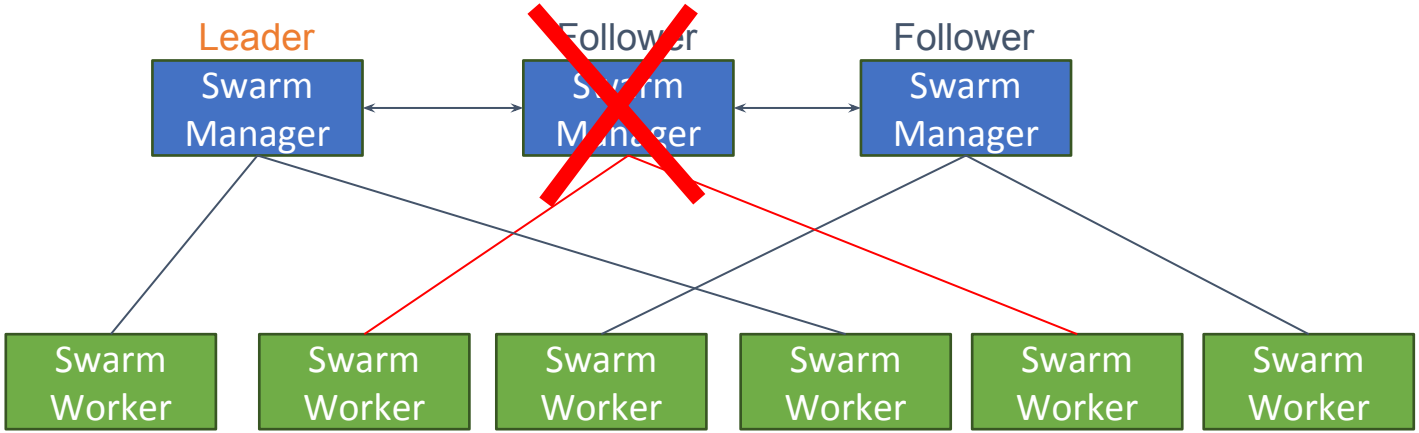
Swarm Topology: High Availability



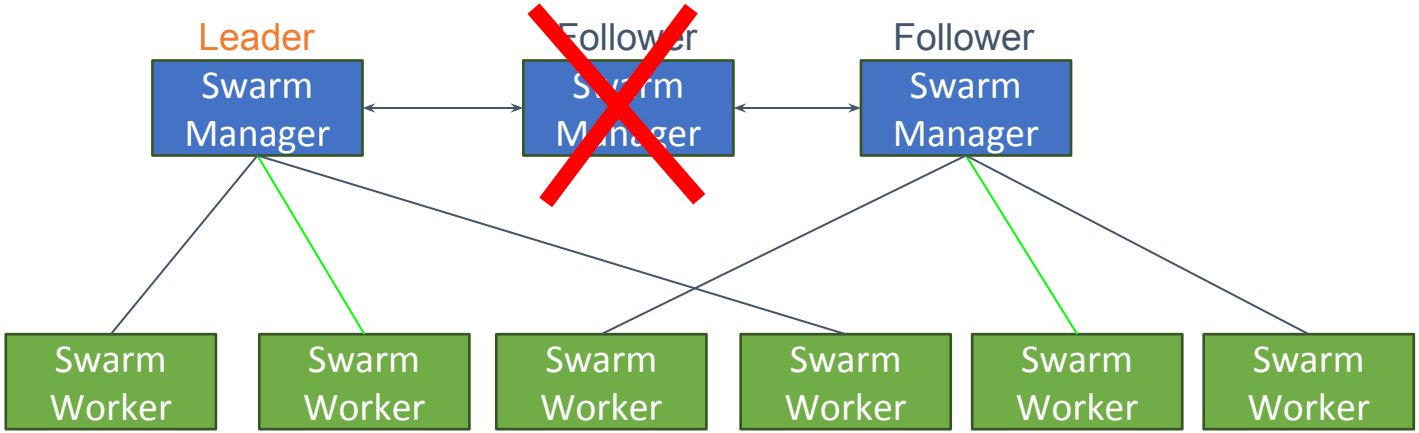
Swarm Topology: High Availability



Swarm Topology: High Availability



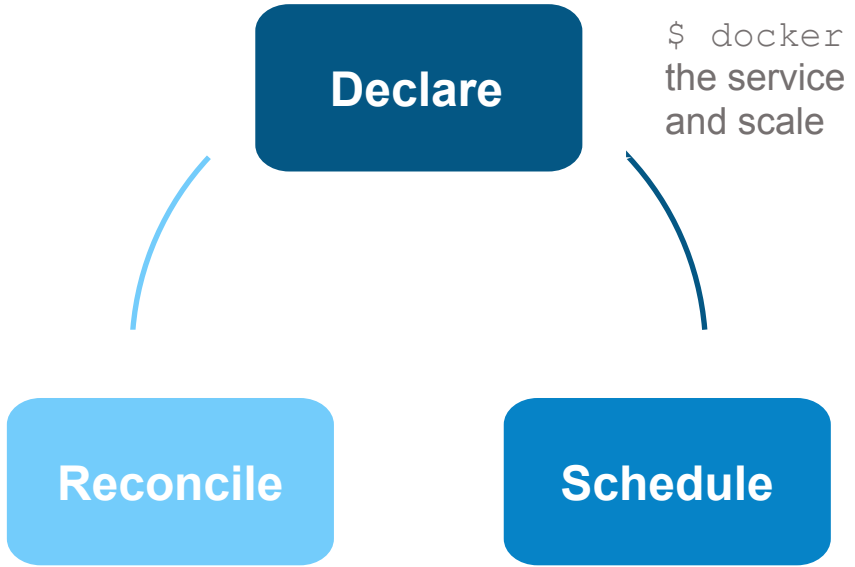
Swarm Topology: High Availability



Services \ Tasks

- Services provide a piece of functionality
  - Based on a Docker image
- Replicated Services and Global Services
- Tasks are the containers that actually do the work
  - A service has 1-n tasks

How service deployment works

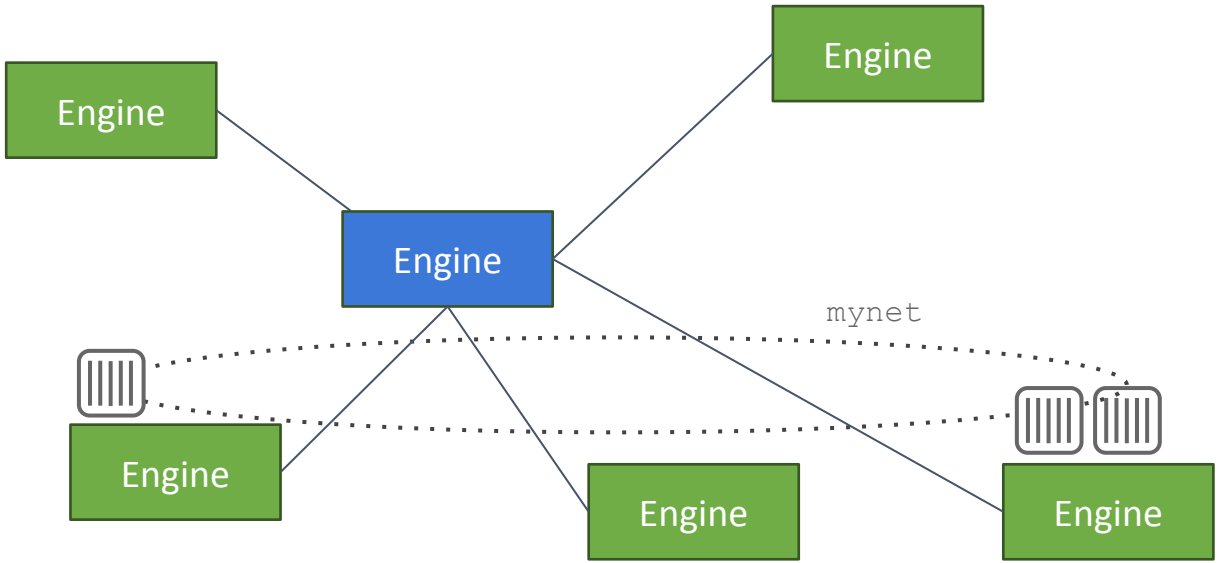


`$ docker service create` declares the service name, network, image:tag and scale

Managers break down service into tasks, schedules them and workers execute tasks

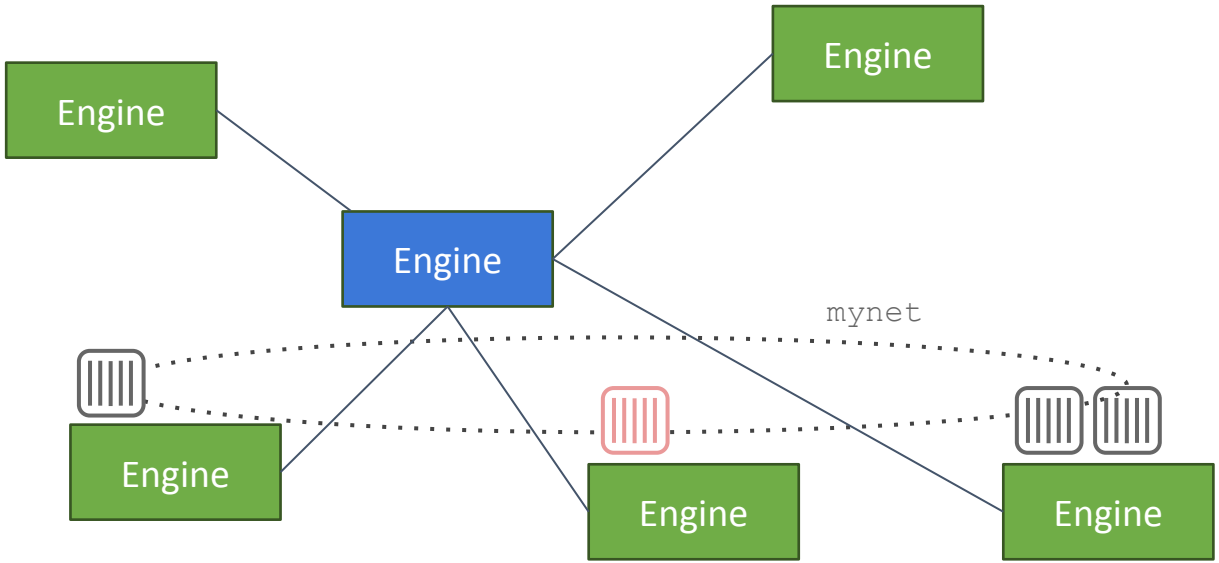
Engines check to see what is running and compared to what was declared to “true up” the environment

Services




```
$ docker service create --replicas 3 --name frontend --network mynet
  --publish 80:80/tcp frontend_image:latest
```

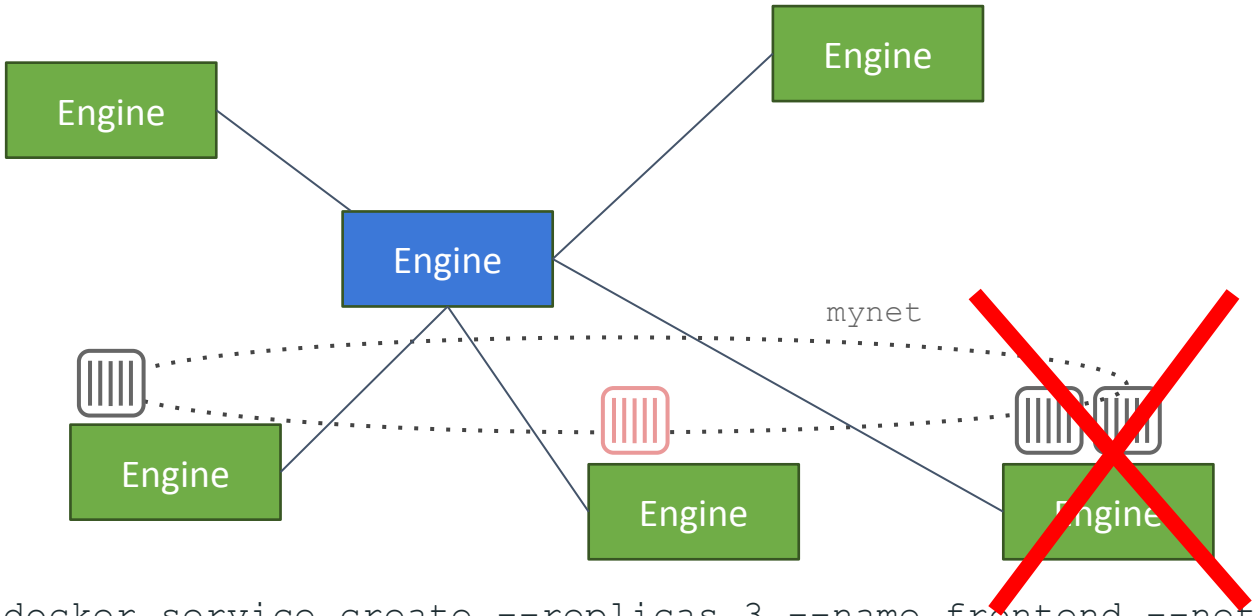
Services




```
$ docker service create --replicas 3 --name frontend --network mynet
 --publish 80:80/tcp frontend_image:latest
```

```
$ docker service create --name redis --network mynet redis:latest
```

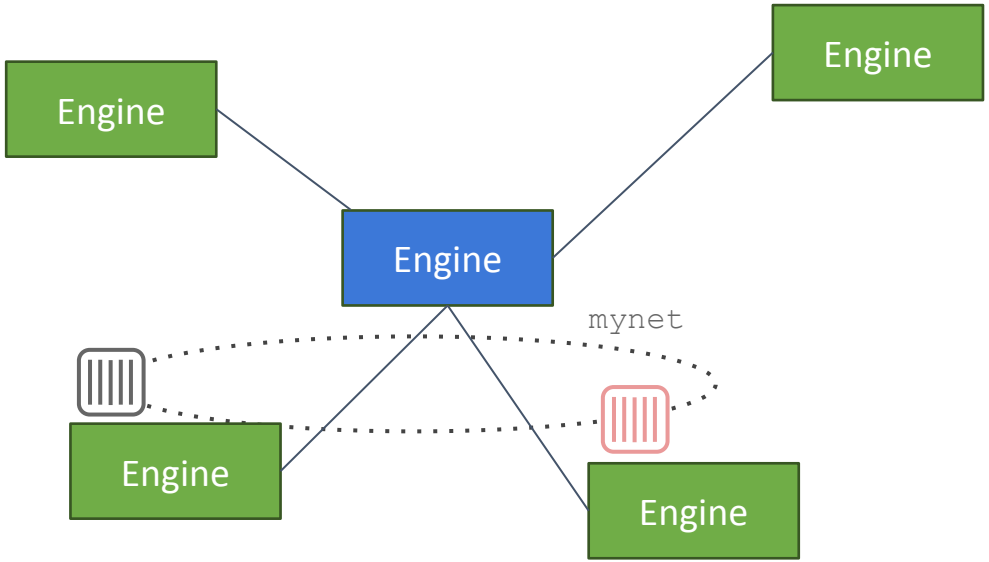
Node Failure




```
$ docker service create --replicas 3 --name frontend --network mynet
  --publish 80:80/tcp frontend_image:latest
```

```
$ docker service create --name redis --network mynet redis:latest
```

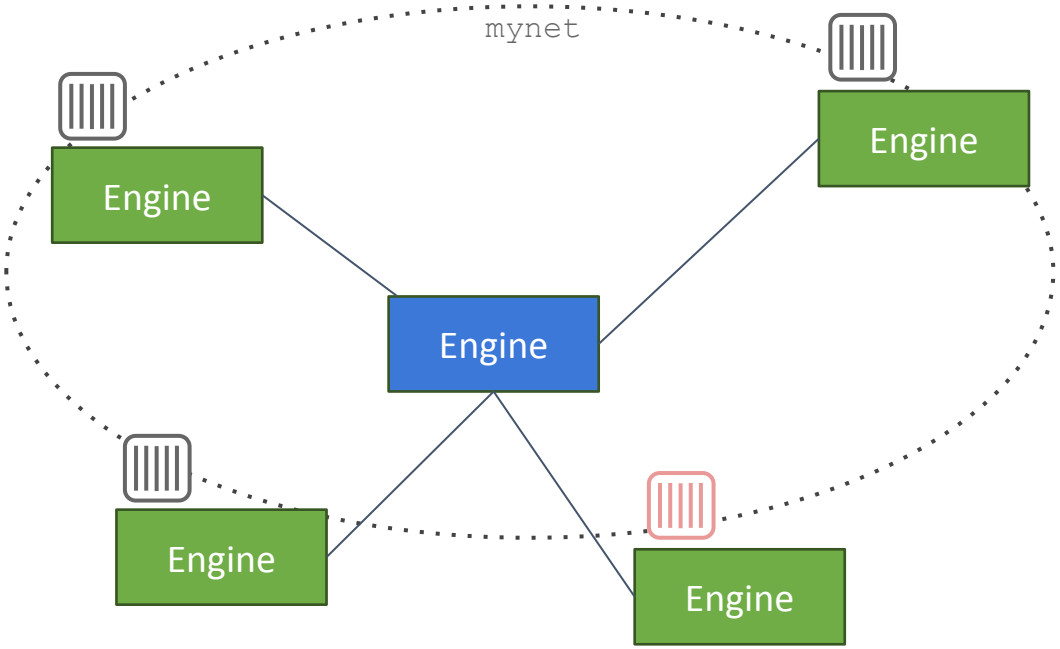
Desired State ≠ Actual State



```
$ docker service create --replicas 3 --name frontend --network mynet
  --publish 80:80/tcp frontend_image:latest
```

```
$ docker service create --name redis --network mynet redis:latest
```

Converge Back to Desired State



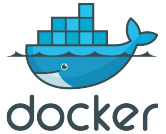
```
$ docker service create --replicas 3 --name frontend --network mynet
 --publish 80:80/tcp frontend_image:latest
```

```
$ docker service create --name redis --network mynet redis:latest
```

Hands On Exercises (and Break)

Swarm Mode Introduction

http://training.play-with-docker.com

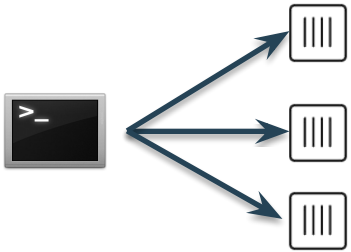


Section 4:
Docker Compose
Stacks
Secrets

Docker Compose: Multi Container Applications

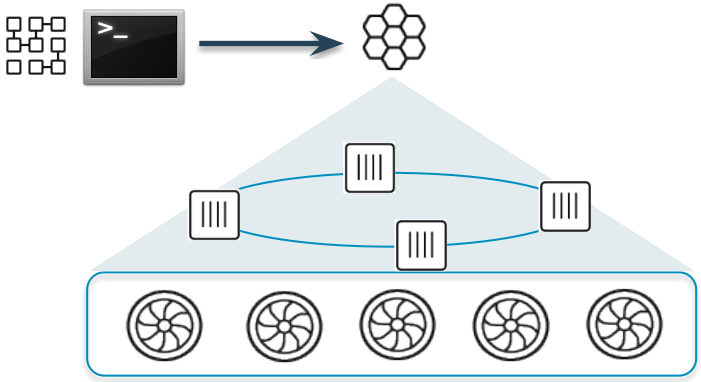
Without Compose

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order

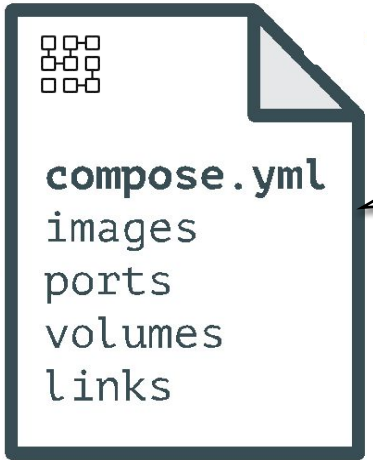


With Compose

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane



Docker Compose: Multi Container Applications

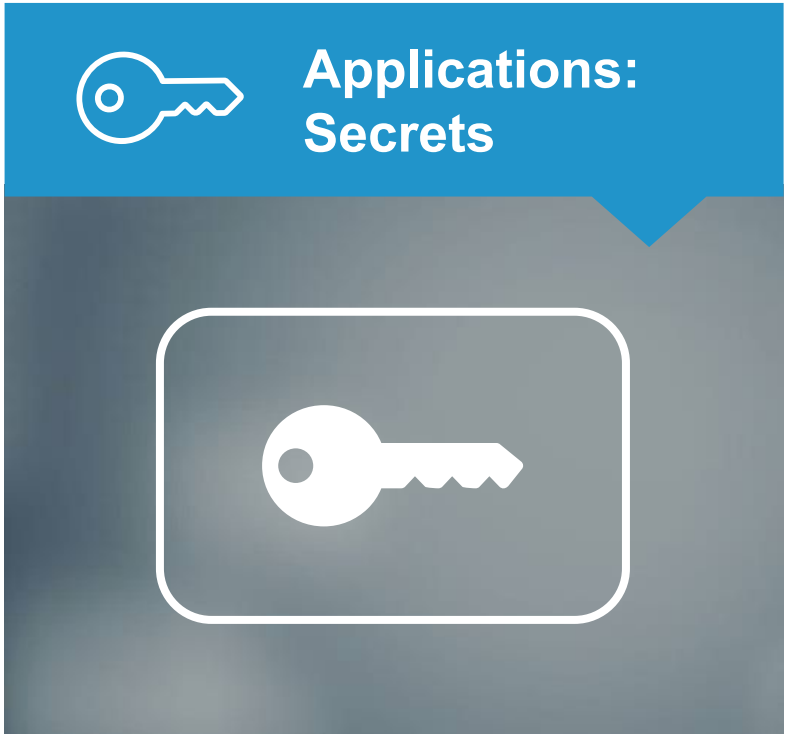



```
containers:
  web:
     build: .
     command: python app.py
     ports:
     - "5000:5000"
     volumes:
     - .:/code
     environment:
     - PYTHONUNBUFFERED=1
  redis:
     image: redis:latest
     command: redis-server --appendonly yes
```

Stacks: Multi-Container Applications

- A stack is a collection of related services
  - Requires Swarm
- Stacks are a Docker primitive
  - `docker stack deploy`
  - `docker stack ps`
  - `docker stack rm`
- Can be implemented via a compose file or application bundle

What is a Secret?



Docker Secrets Management

Secrets management architected for containerized applications

- **Usable Security:** Integrated and designed with dev and ops workflows in mind
- **Trusted Delivery:** Encrypted storage and secure transit with TLS
- **Infrastructure Independent:** A portable security model across any infrastructure across the lifecycle

All apps are safer - Only the assigned app can access the secret, even with multiple apps on the same cluster

Safer Apps with Docker Secrets Management

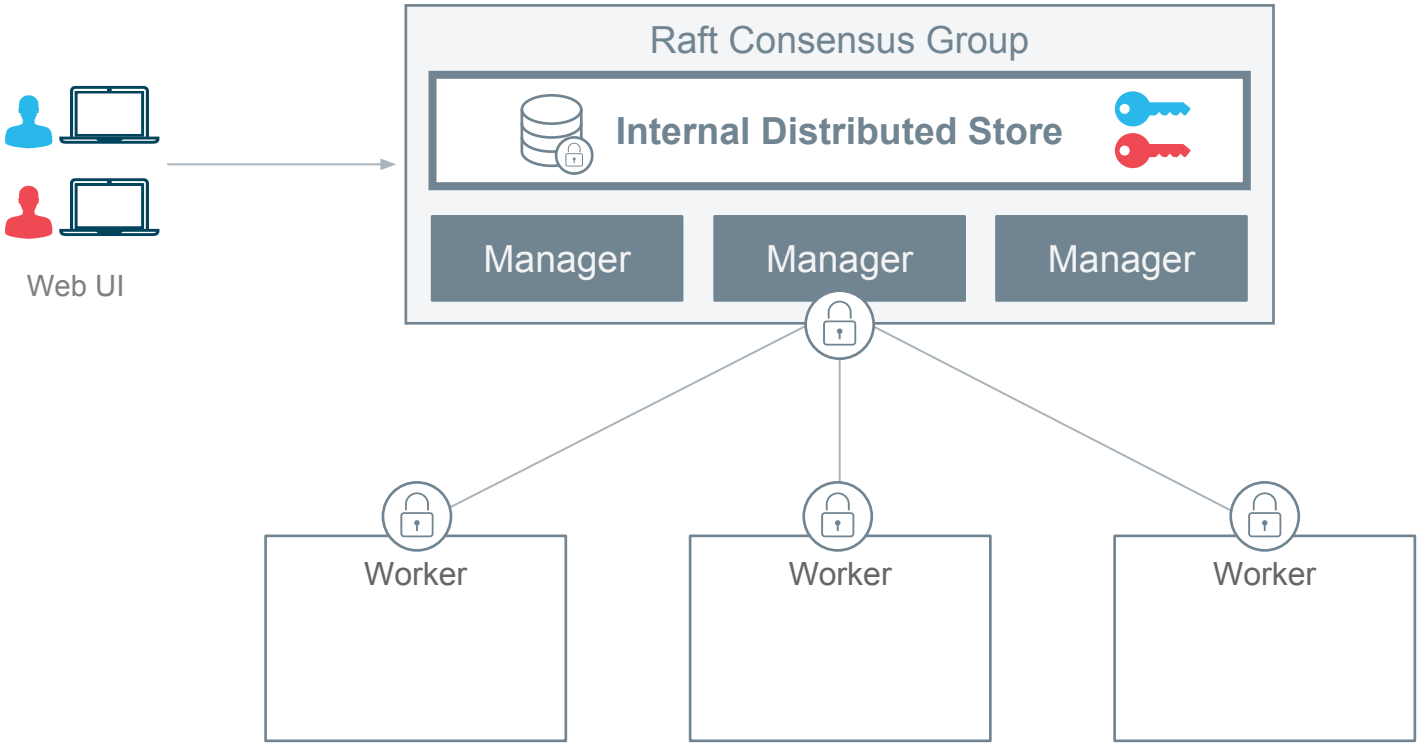
- Apps are safer when there is a standardized interface for accessing secrets
  - Legacy/microservices
  - Dev & Ops
  - Linux & Windows

- Apps are safer when secrets are not stored in the app itself

Trusted Delivery with Docker Secrets

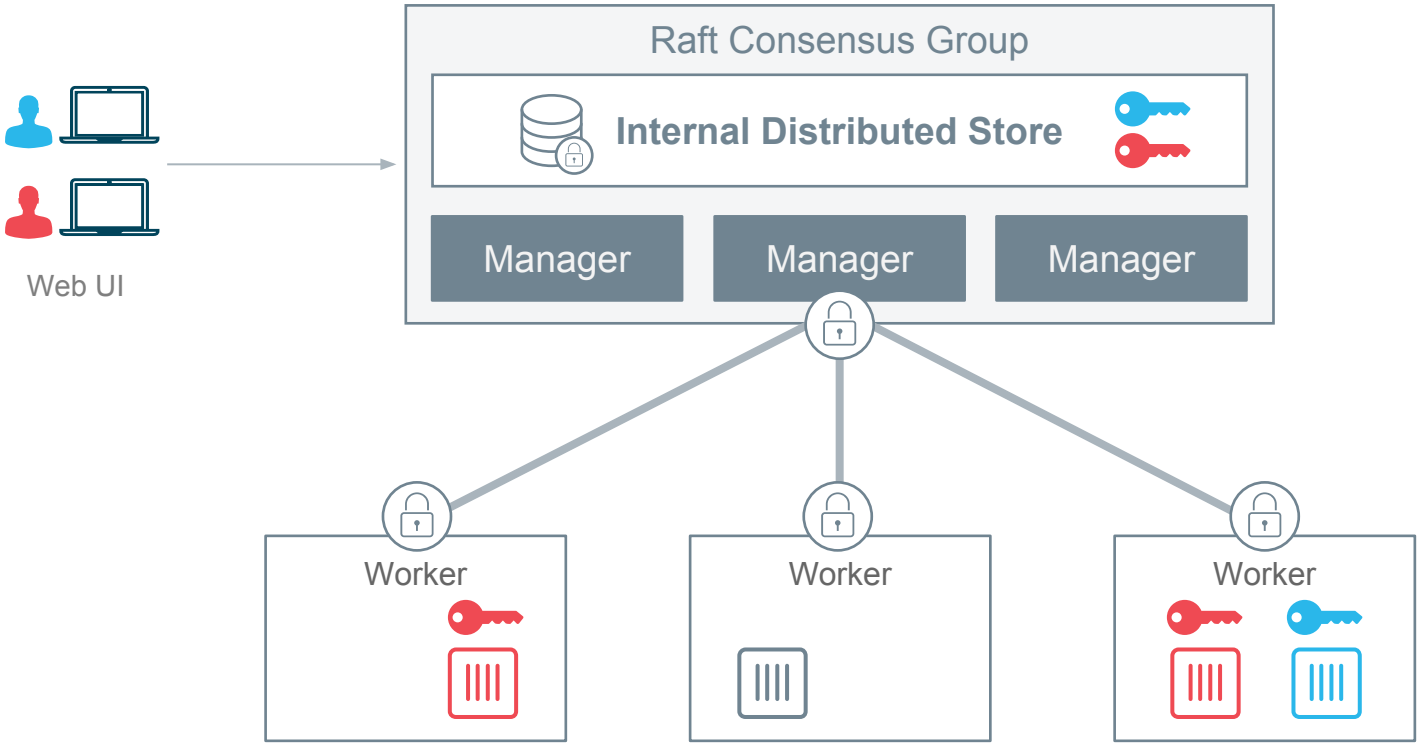
- Encrypted at rest in the cluster store
- Encrypted while in motion on the network
- Delivered only to the exact authorized app
- Available to containers only in memory, never saved to disk

Secrets Architecture



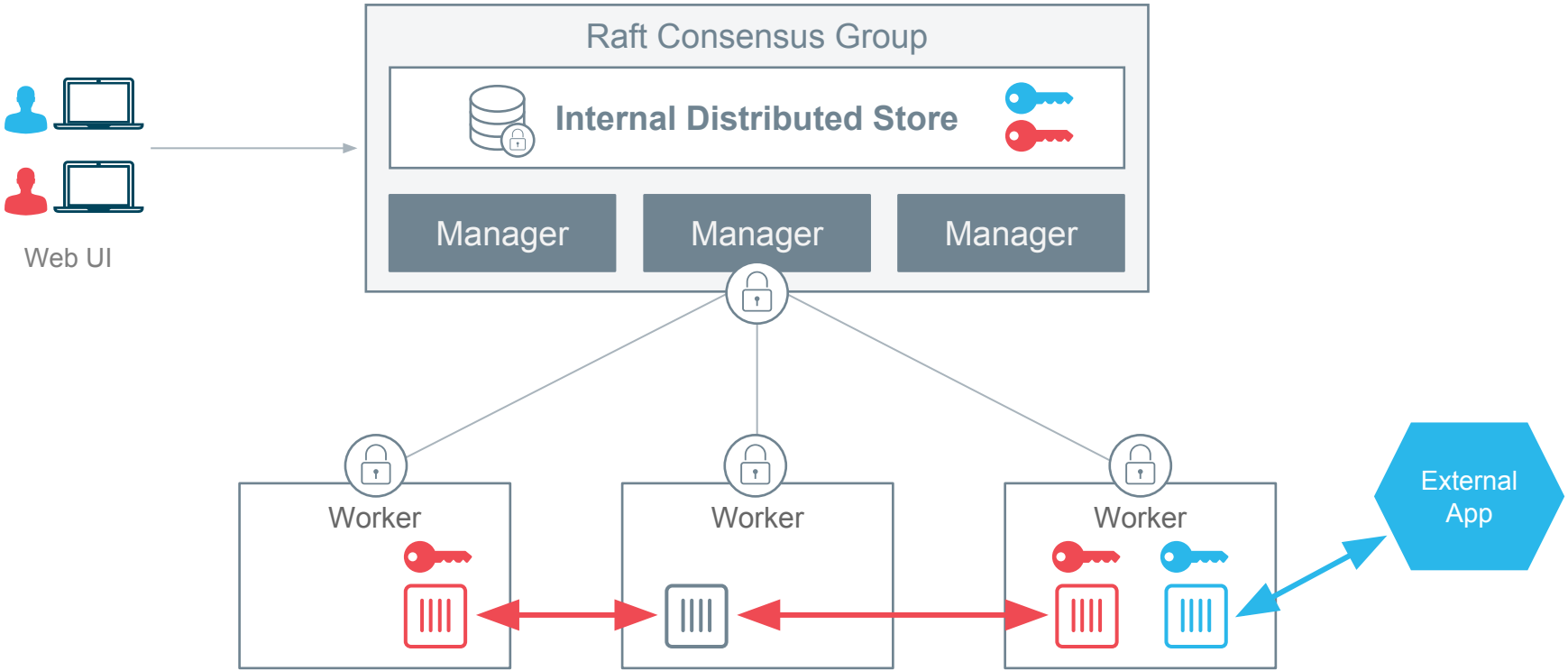
docker

Secrets Architecture



docker

Secrets Architecture



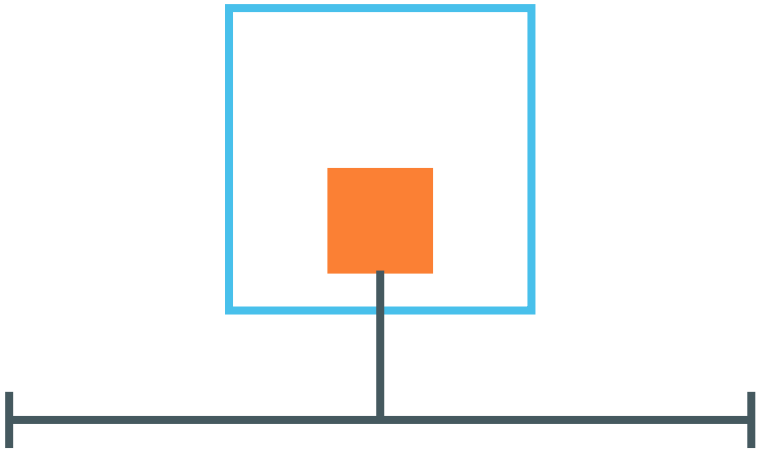
Hands On Exercises

Swarm Stack Introduction
Docker Compose with Secrets

http://training.play-with-docker.com

docker

Container Network Model (CNM)



- Network
- Sandbox
- Endpoint

Containers and the CNM