NobleProg

The World's Local Training Provider

# About Your Trainer

Your trainer: Mr. Lakshmi Narayanan

Education: B.Tech Information Technology, M.Tech Software Engineering

Training Experience: Certified Corporate Trainer > 9 Years, DevOps Architect.

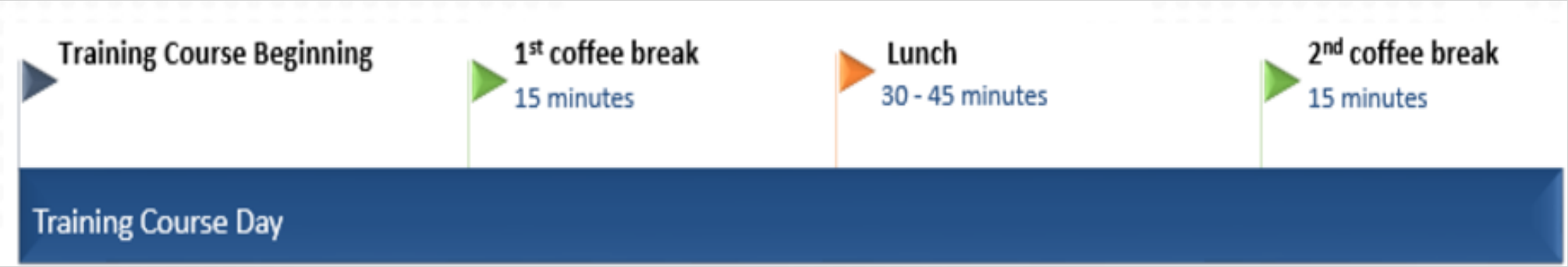
Certifications:

- Microsoft Certified Professional (MCP)
- Microsoft Specialist (MS)
- Microsoft® Certified Professional Developer (MCPD)
- Microsoft® Certified Technology Specialist (MCTS)
- Certified Lean Six Sigma Yellow Belt (ICYB)
- Associate, Customer Service (ACS)
- Associate, Annuity Products and Administration (AAPA)
- Docker Technical Sales Professional (DTSP)

# Organization

**Days** : Sep 2, 3 & 4 – 2019

**Hours** : 09:30 AM to 05:30 PM with breaks.






NobleProg

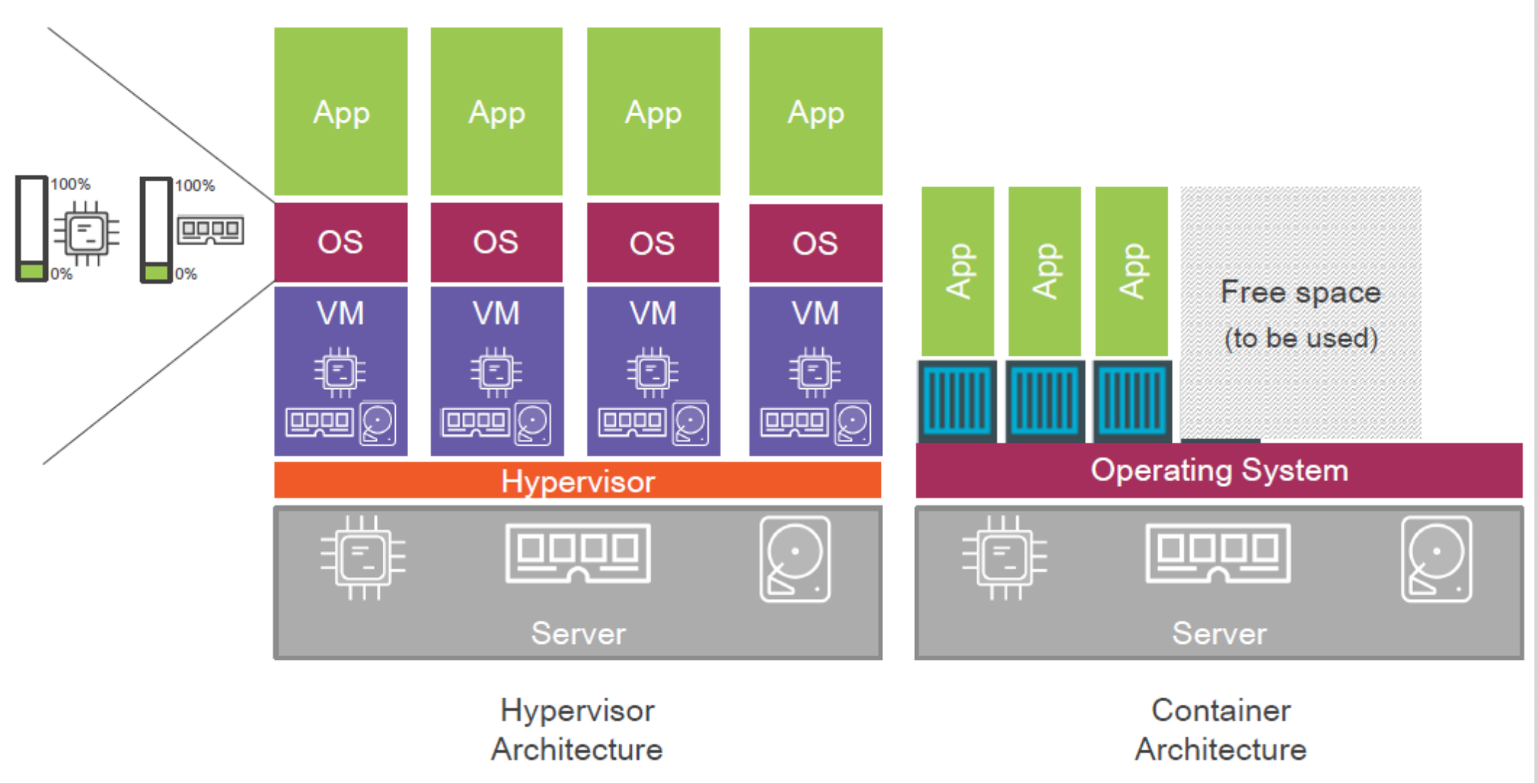
# After the course

- After the training course, you will receive:




NobleProg

# Container Architecture

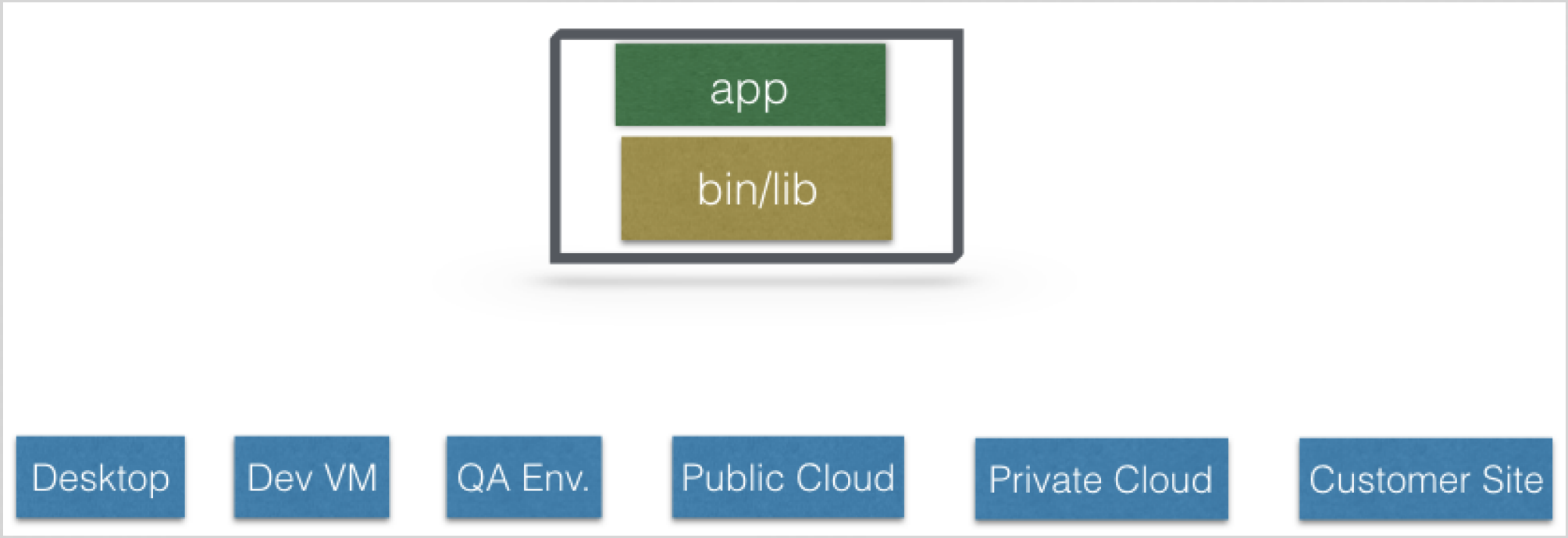





NobleProg

# Containers

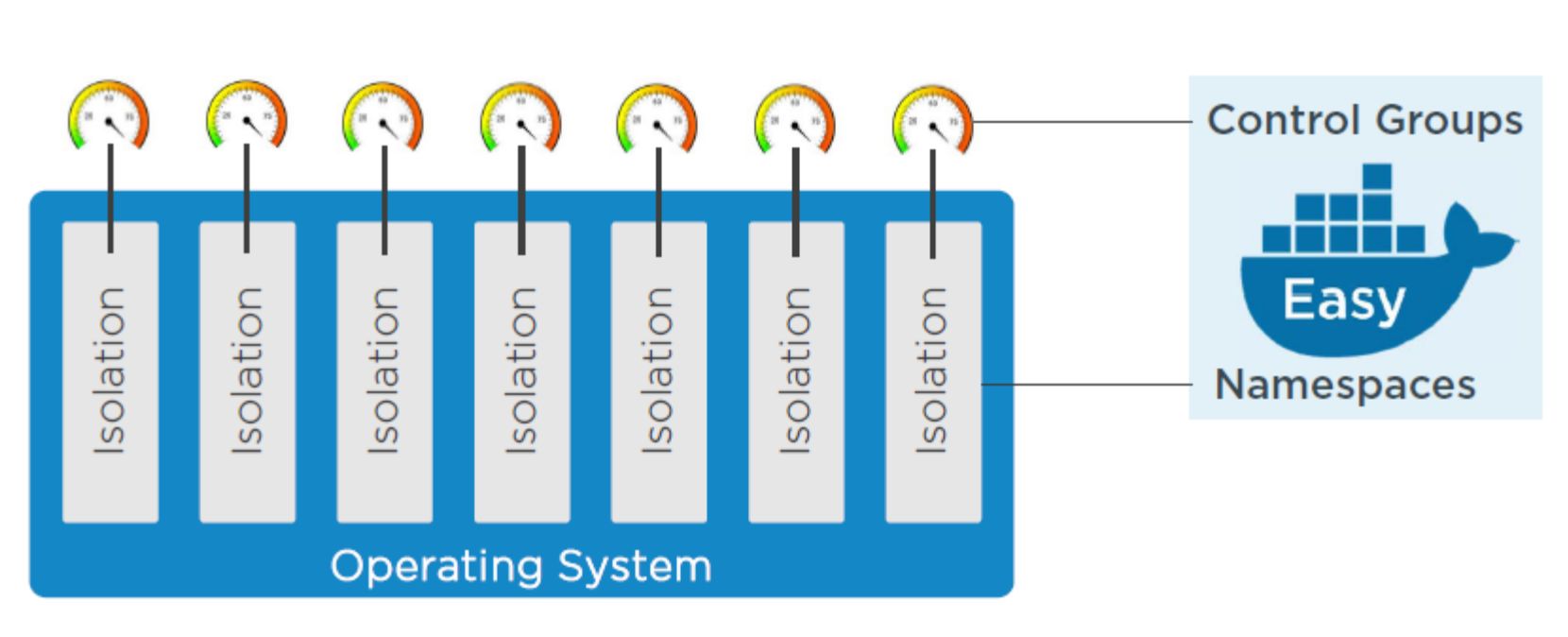
- An application-centric way to deliver high-performing, scalable applications on the infrastructure of your choice.
- With a **container image**, we bundle the application along with its runtime and dependencies. We use that image to create an isolated executable environment, also known as container.






NobleProg

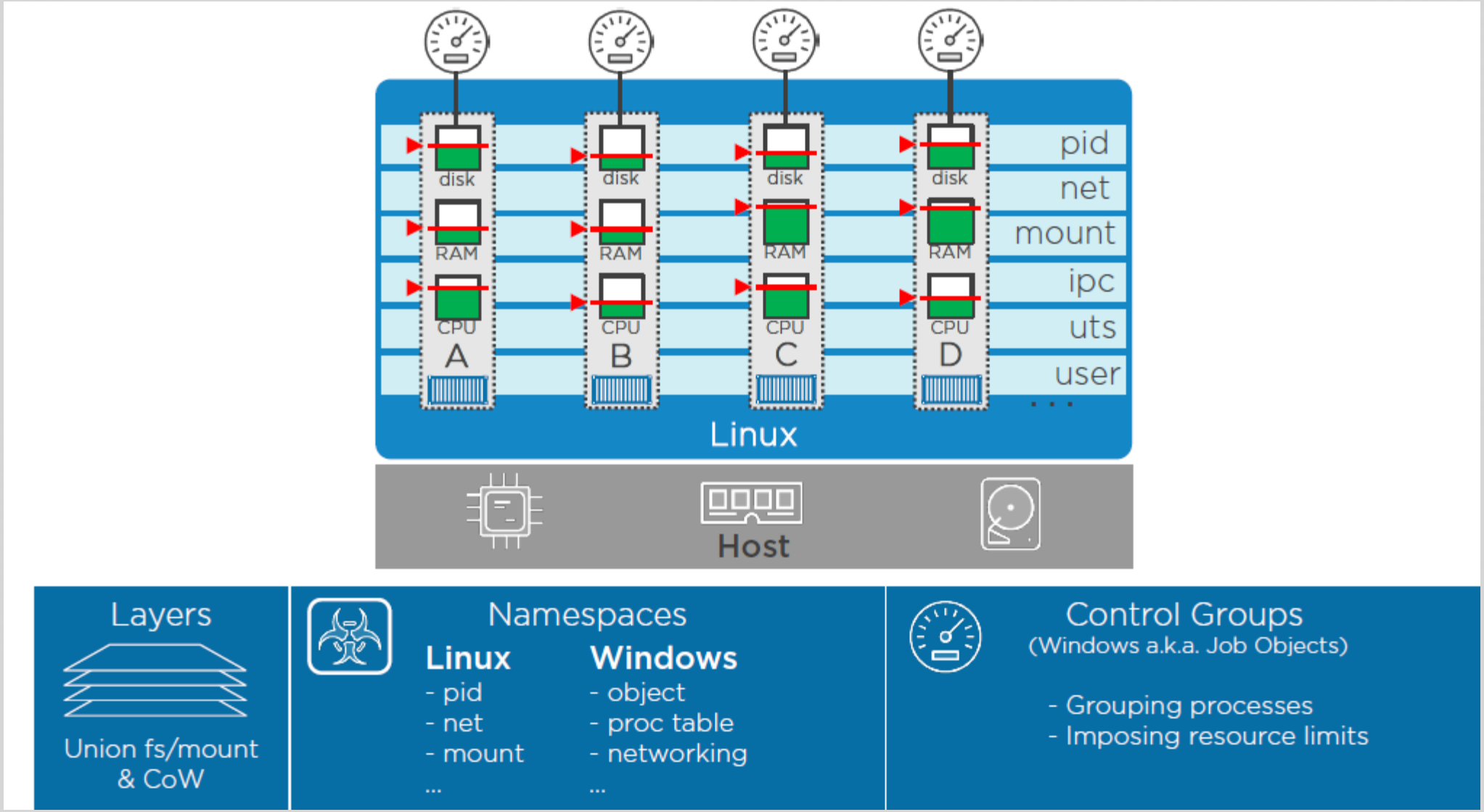
# Containers






NobleProg

# Container primitives in the Kernel

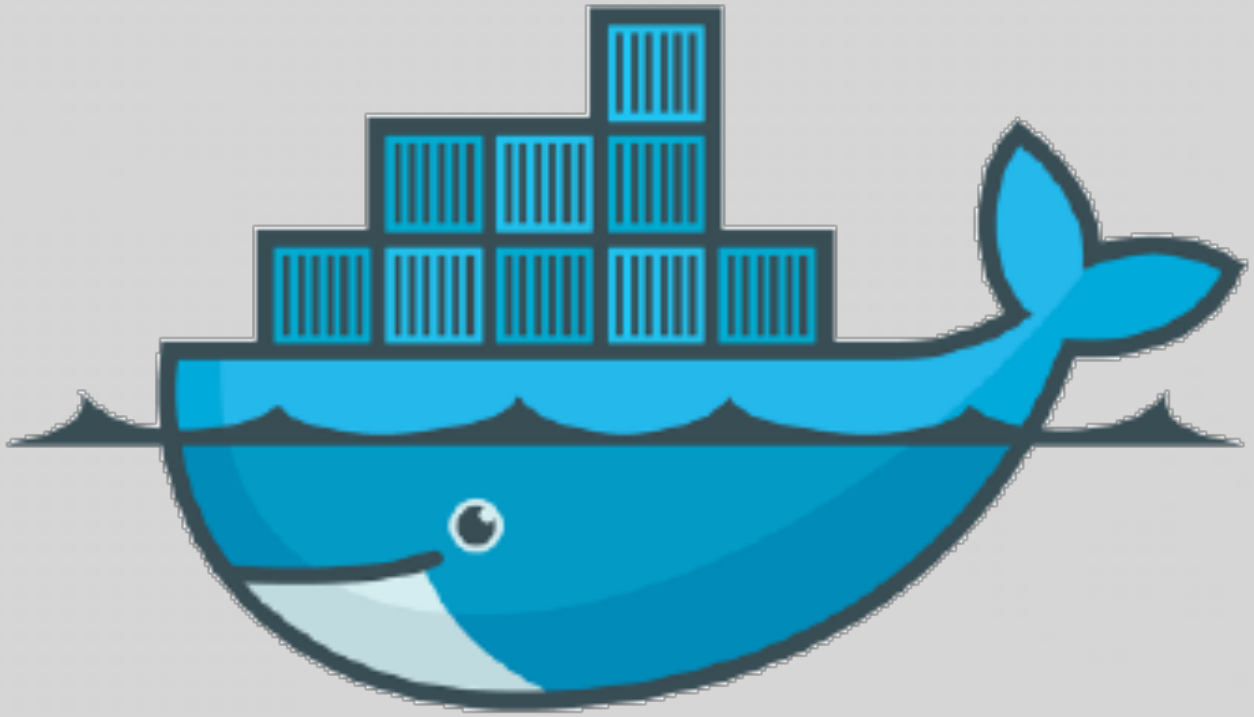




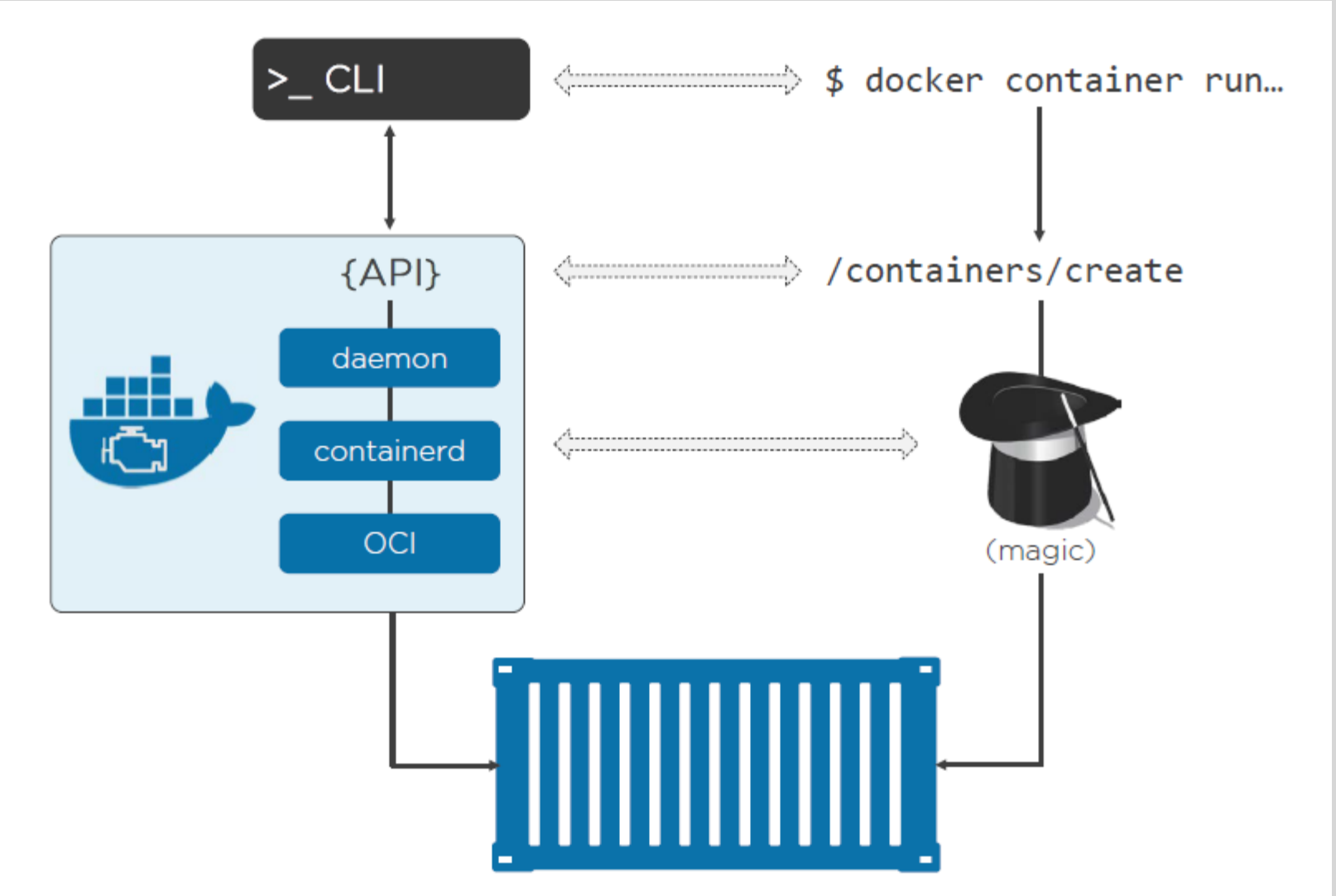

NobleProg

# Docker

- Lightweight, open, secure platform
- Simplify building, shipping, running apps
- Shipping container system for code
- Runs natively on Linux or Windows Server
- Runs on Windows or Mac Development machines (with a virtual machine)
- Relies on "images" and "containers"



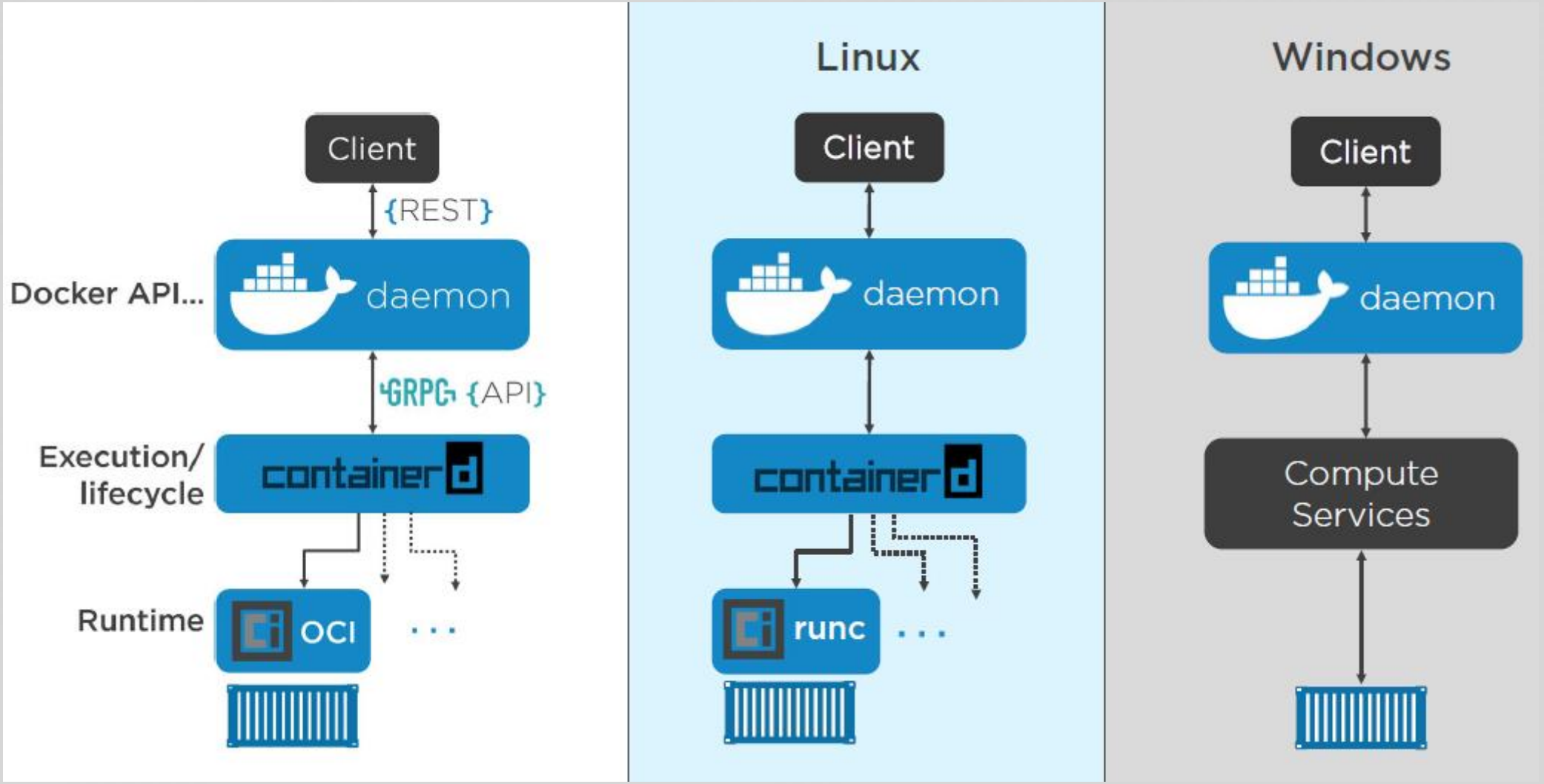
# Containers






NobleProg

# The Docker Engine






NobleProg

# What's in this section

- Check version of our docker cli and engine
- Create a Nginx (webserver) container
- Learn common container management commands
- Learn Docker networking basics
- Requirements: Have latest Docker installed from last section
- command: docker version
  - verified cli can talk to engine
- command: docker info
  - most config values of engine
- docker command line structure
  - old way (still works): docker <command> (options)
  - new: docker <command> <sub-command> (options)

# What happens in 'docker container run'

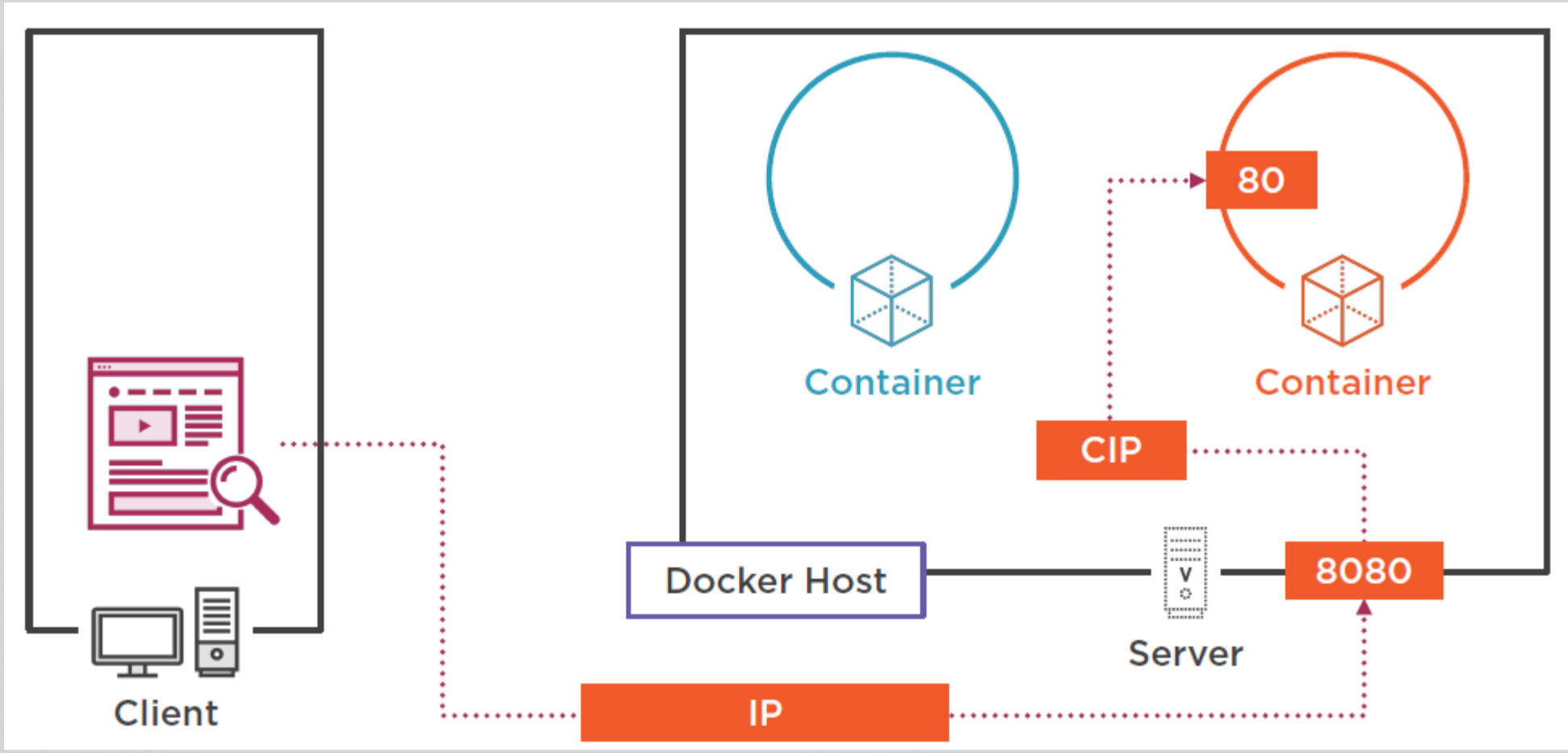
```
docker container run --publish 8080:80 --name webhost -d nginx:1.11
```

1. Looks for that image locally in image cache, doesn't find anything
2. Then looks in remote image repository(defaults to Docker Hub)
3. Downloads the latest version (nginx: latest by default)
4. Create new container based on that image and prepares to start
5. Gives it a virtual IP on a private network inside docker engine
6. Open up port 80 on host and forwards to port 80 in container
7. Starts container by using the CMD in the image Dockerfile

- Containers aren't Mini-VM's
- They are just processes
- Limited to what resources they can access
- Exit when process stops



NobleProg






NobleProg

# Assignment: Manage Multiple Containers

- docs.docker.com and --help are your friend
- Run a nginx, a mysql, and a httpd(apache) server
- Run all of them --detach (or -d), name them with --name
- nginx should listen on 80:80, httpd on 8080:80. mysql on 3306:3306
- When running mysql, use the --env option(or -e) to pass in MYSQL_RANDOM_ROOT_PASSWORD=yes
- Use docker container logs on mysql to find the randowm password it created on startup
- Clean it all up with docker container stop and docker container rm(both can accept multiple names or ID's)
- Use docker container ls to ensure everything is correct before and after cleanup

# What's going on in Containers

docker container top - process list in one container

docker container inspect - details of one container config

docker container stats - performance stats for all containers

# Getting a Shell Inside Containers

docker container run -it - start new container interactively

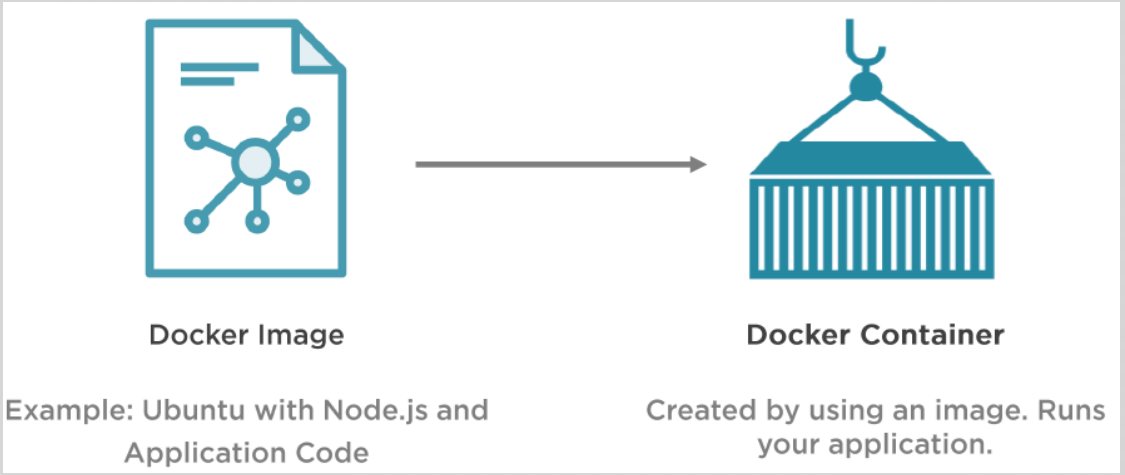
docker container exec -it - run additional command in existing container

# What's in this section

- All about images, the building blocks of containers
- What's in an image(and what isn't)
- Using Docker Hub registry
- Managing our local image cache
- Building our own images

# What's in an Image(and What isn't)

- App binaries and dependencies
- Metadata about the image data and how to run the image
- Official definition:
  - "An image is an ordered collection of root file system changes and the corresponding execution parameters for use within a container runtime."
- Not a complete OS. No kernel, kernel modules(e.g. drivers)
- Small as one file(your app binary) like a golang static binary
- Big as a Ubuntu distro with apt, and Apache, PHP, and more installed




## Image

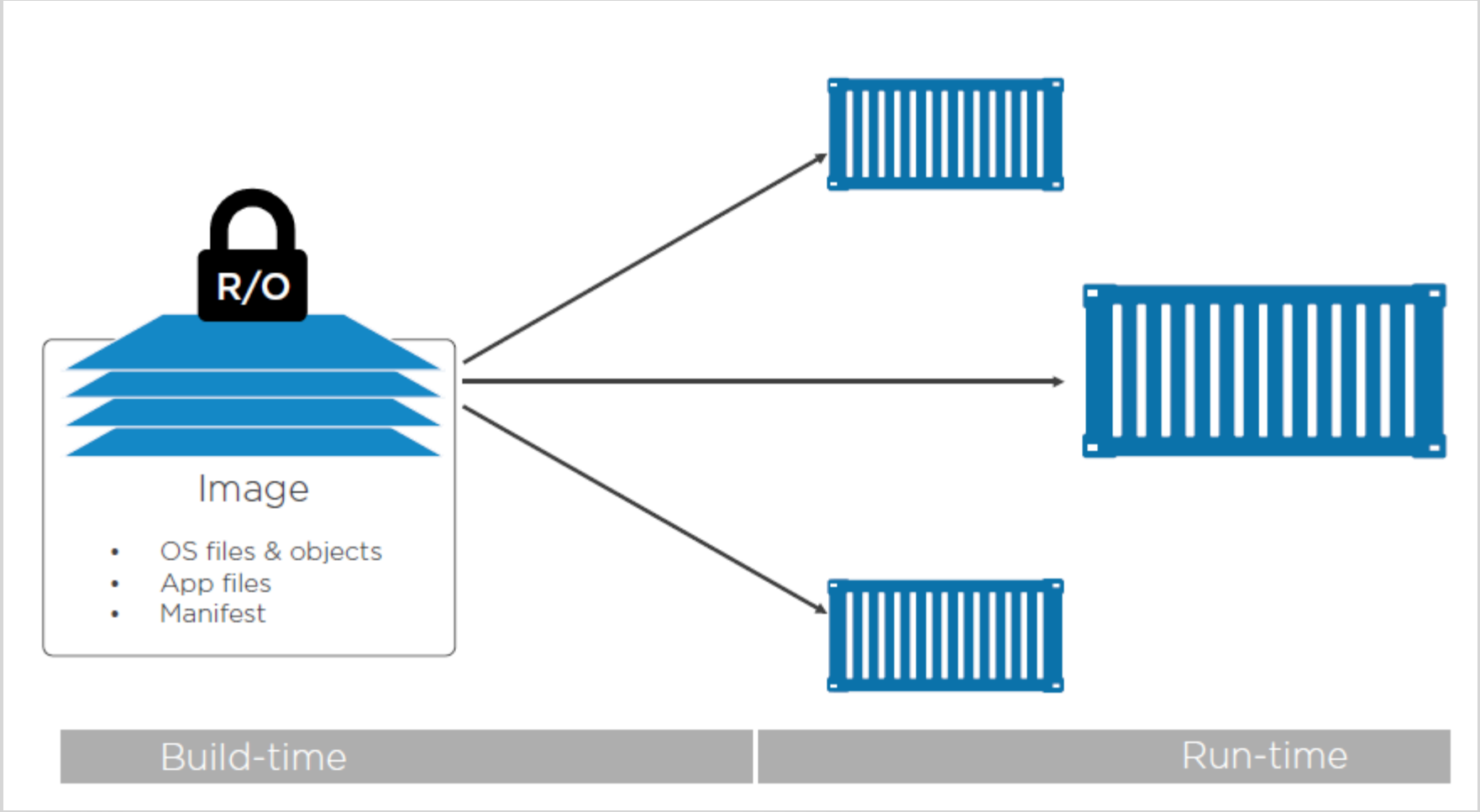
**A read-only template** composed of layered filesystems used to share common files and create Docker container instances.

## Container

**An isolated and secured shipping container** created from an image that can be run, started, stopped, moved and deleted.



NobleProg

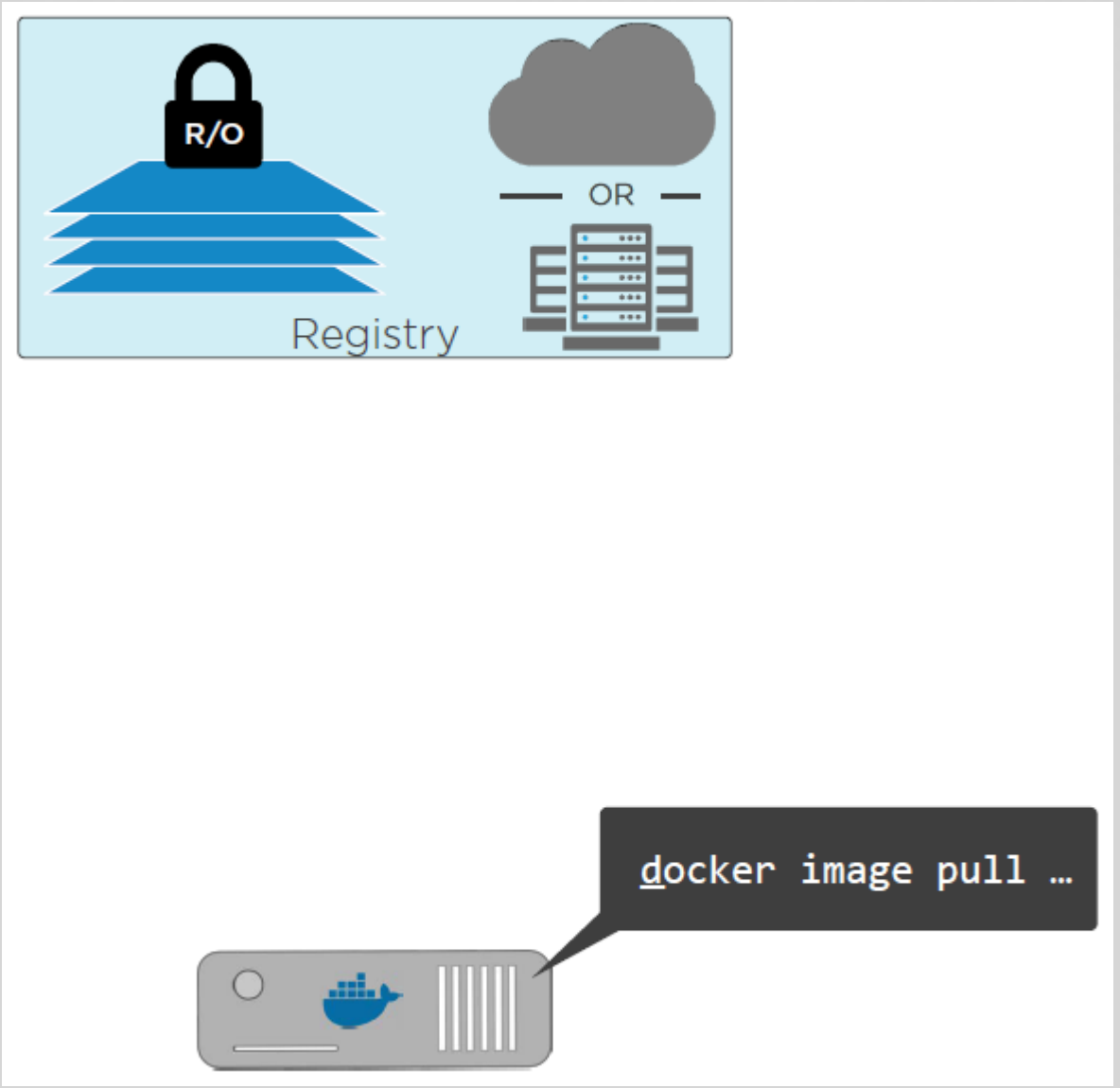





NobleProg

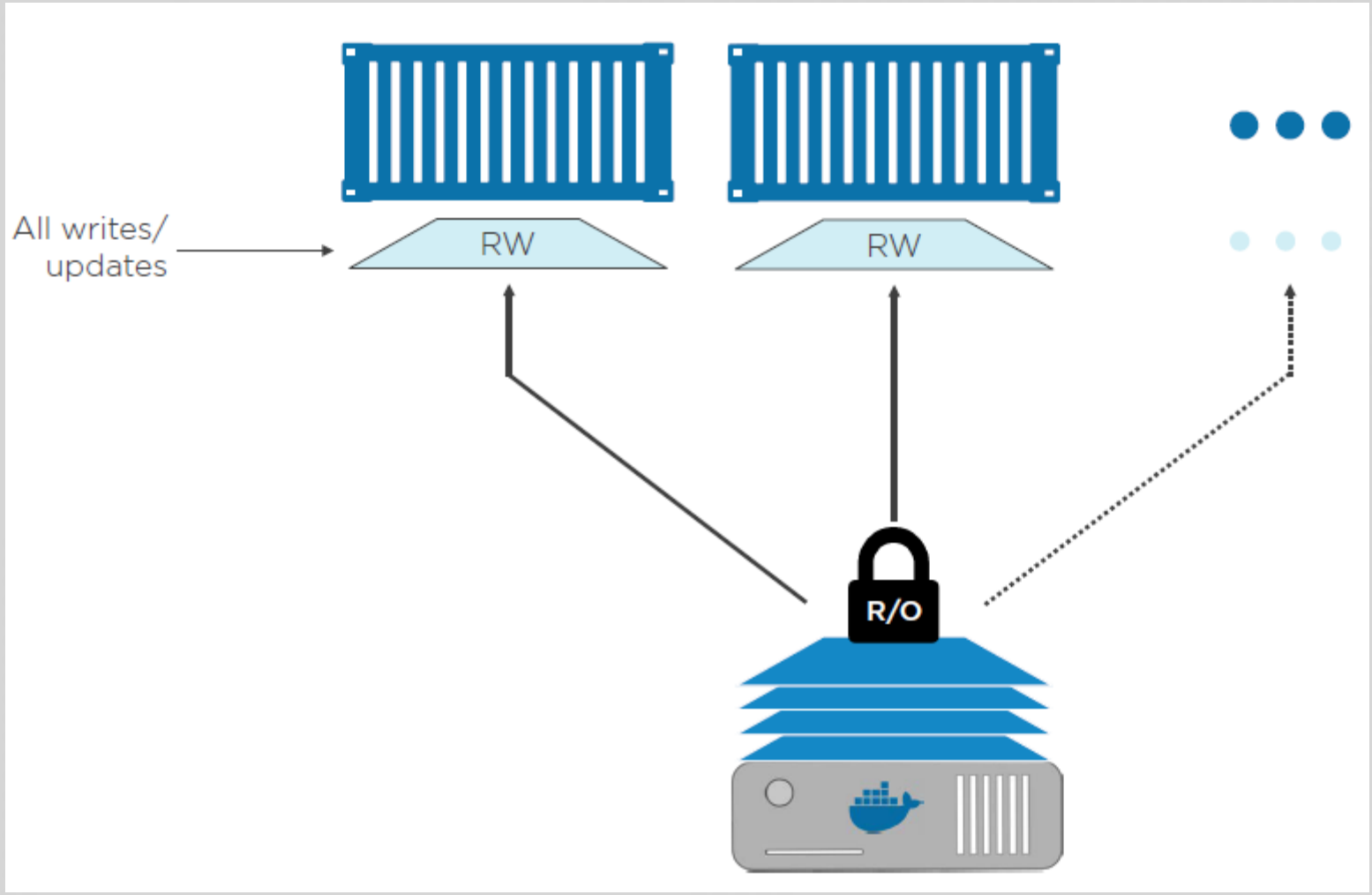
# What's in this section

- All about image tags
- How to upload to Docker Hub
- Image ID vs. Tag
- Properly tagging images
- Tagging images for upload to Docker Hub
- How tagging is related to image ID
- The Latest Tag
- Logging into Docker Hub from docker cli
- How to create private Docker Hub images



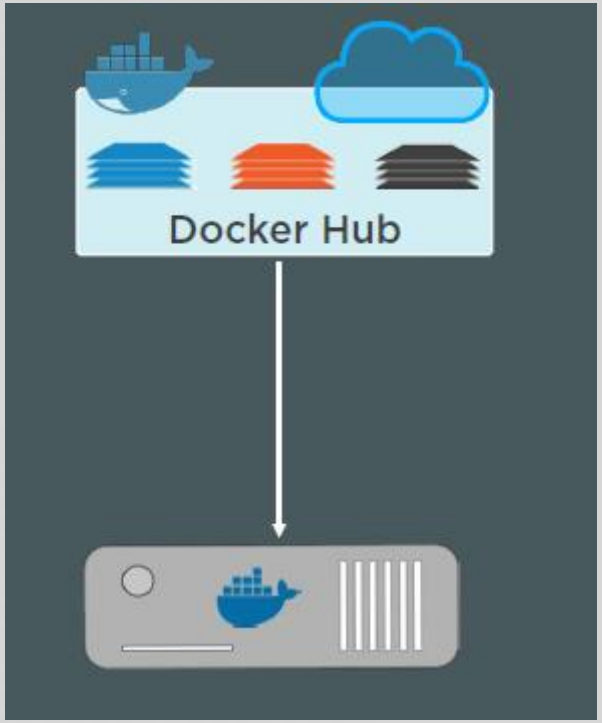



NobleProg






NobleProg




REGISTRY / REPO / IMAGE (TAG)

docker.io/redis/latest

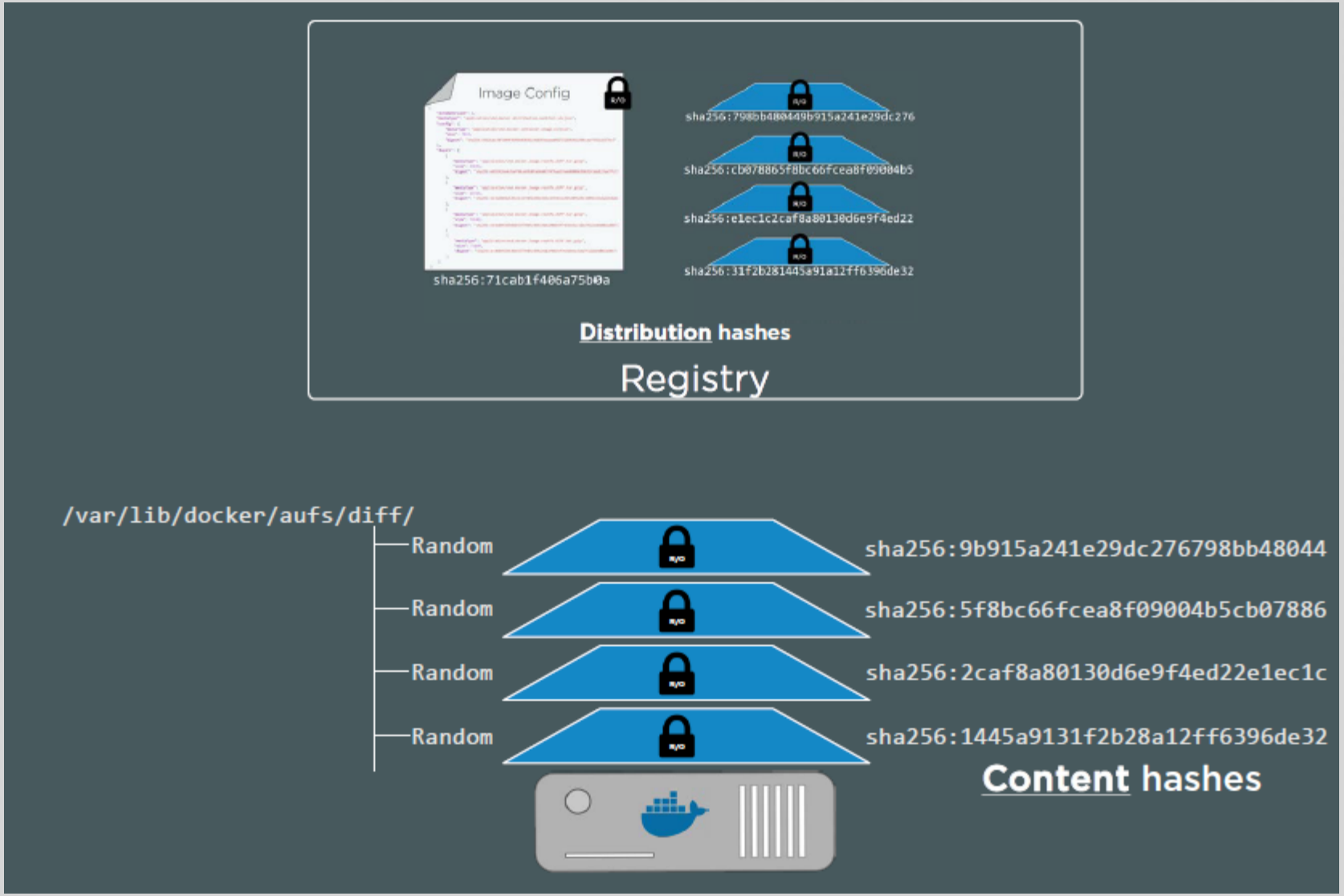
docker.io/nginx/1.13.5

**docker.io**/<repo>/**latest**

(default) (default)

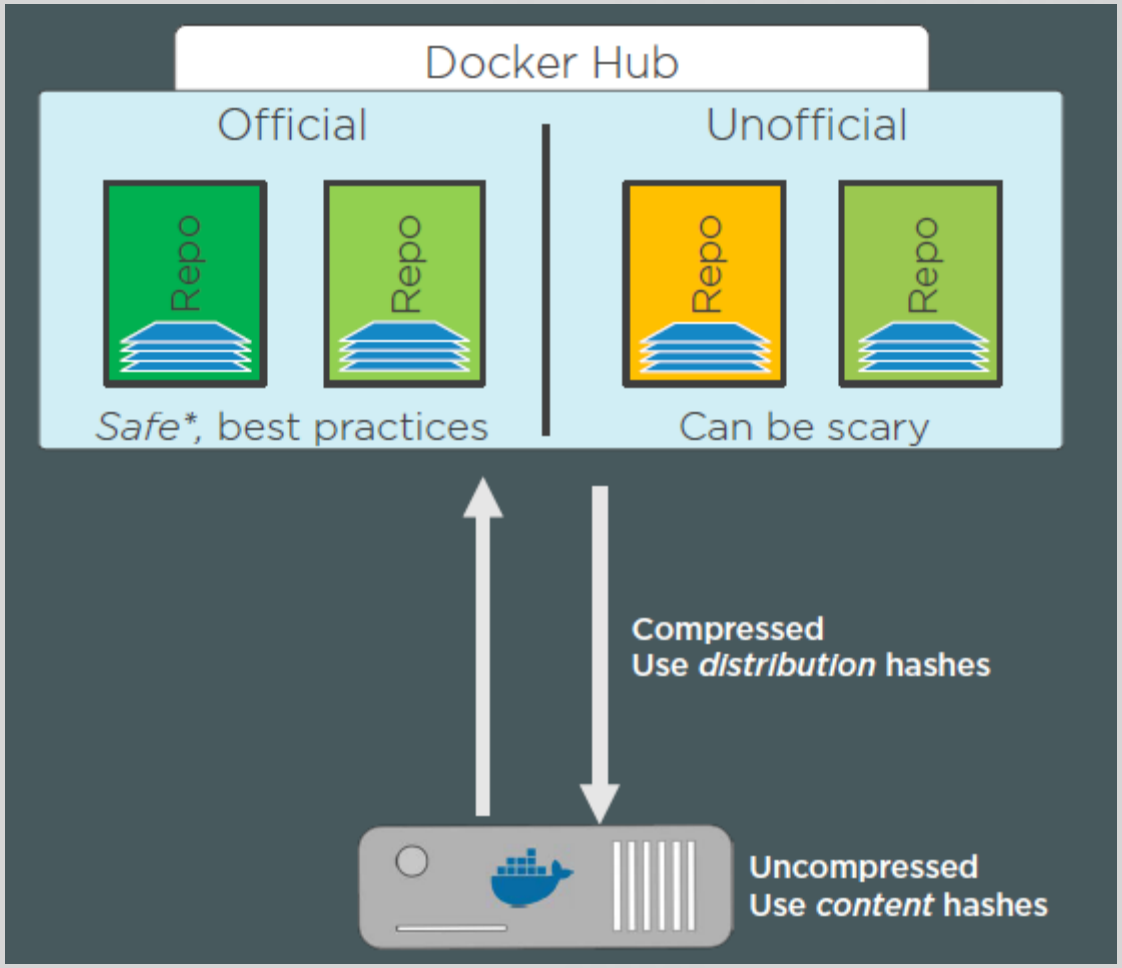


NobleProg






NobleProg

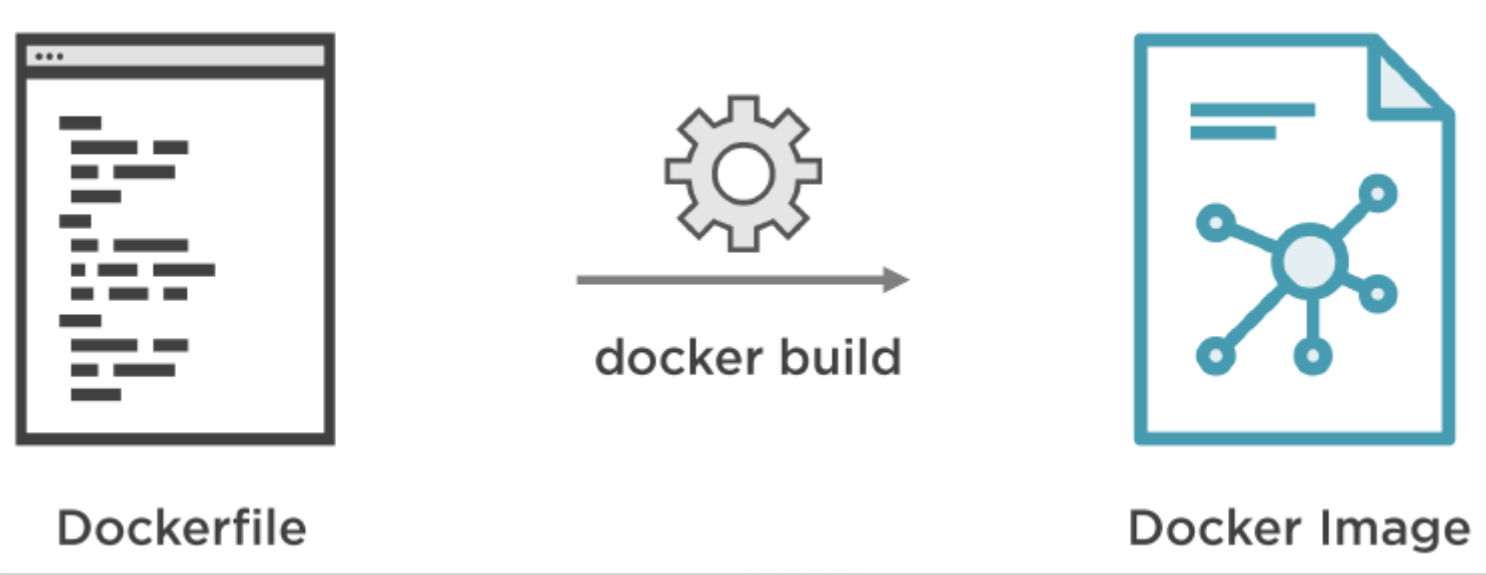





NobleProg

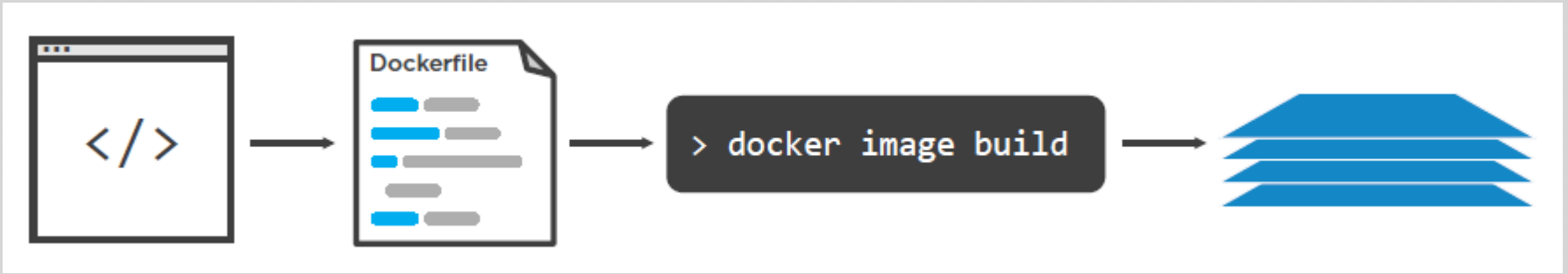
# Dockerfile and Images

- Text file used to build Docker images
- Contains build instructions
- Instructions create intermediate image that can be cached to speed up future builds
- Used with "docker build" command






NobleProg




## Dockerfile notes

- Instructions for building images
- CAPITALIZE instructions
- <INSTRUCTION> <value>
- FROM always first instruction
- FROM = base image
- Good practice to list maintainer
- RUN = execute command and create layer
- COPY = copy code into image as new layer
- Some instructions add metadata instead of layers
- ENTRYPOINT = default app for image/container



NobleProg

# Key Dockerfile Instructions

| | | |
|---|---|---|
| FROM | MAINTAINER | RUN |
| COPY | ENTRYPOINT | WORKDIR |
| EXPOSE | ENV | VOLUME |



NobleProg

# Assignment: Build your own image

- Dockerfiles are part process workflow and part art
- Take existing Node.js app and Dockerize it
- Make Dockerfile. Build it. Test it. Push it.(rm it). Run it.
- Expect this to be iterative. Rarely do I get it right the first time.
- Details in dockerfile-assignment-1/Dockerfile
- Use the Alpine version of the official 'node' 6.x image
- Expected result is web site at http://localhost
- Tag and push to your Docker Hub account (Free)
- Remove your image from local cache, run again from Hub

# Docker Networks

- Each container connected to a private virtual network "bridge"
- Each virtual network routes through NAT firewall on host IP
- All containers on a virtual network can talk to each other without -p
- Best practice is to create a new virtual network for each app:
  - network "my_web_app" for mysql and php/apache containers
  - network "my_api" for mongo and nodejs containers
- Batteries Included, But Removable
  - Default works well in many cases, but easy to swap out parts to customize it
- Make new virtual networks
- Attach containers to more than one virtual network(or none)
- Skip virtual networks and use host IP(--net=host)
- Use different Docker network drivers to gain new abilities and much more..

# Docker Networks: CLI Management

- Show networks docker network ls
- Inspect a network docker network inspect
- Create a network docker network create --driver
- Attach a network to container docker network connect
- Detach a network from container docker network disconnect

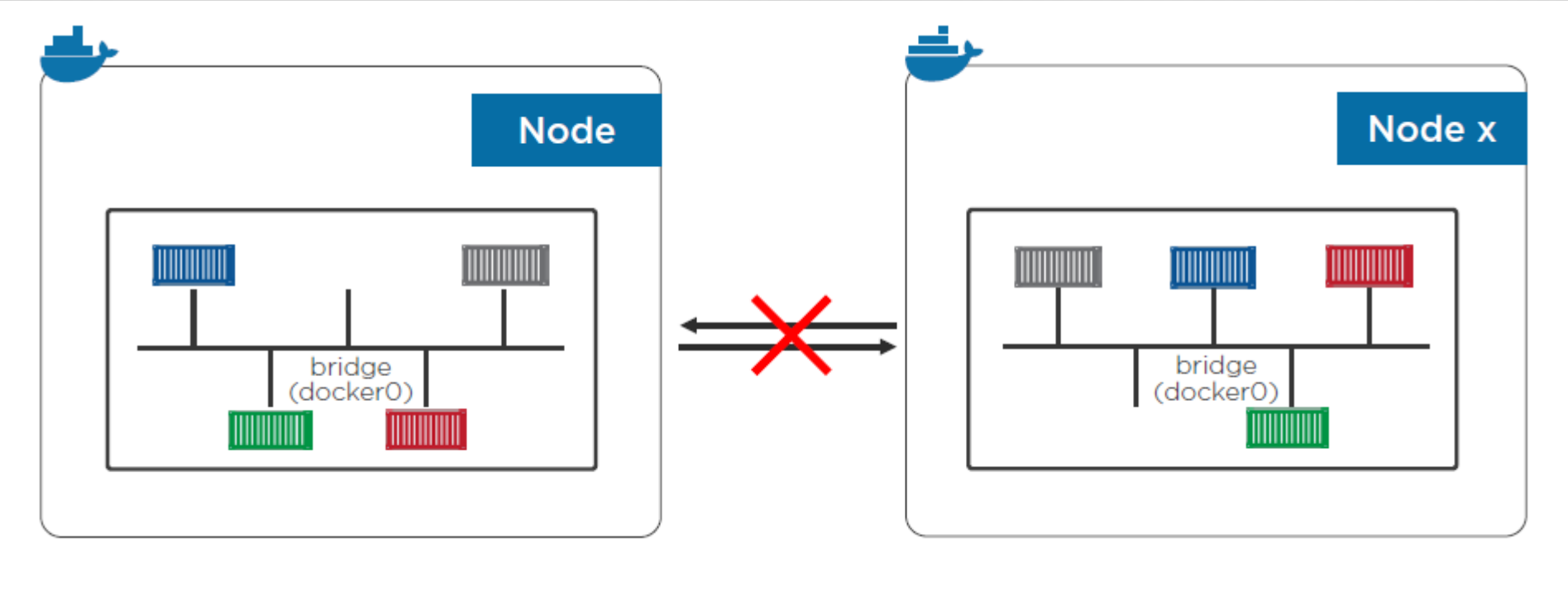
# Docker Networks: DNS

- Understand how DNS is the key to easy inter-container comms
- See how it works by default with custom networks
- Learn how to use --link to enable DNS on default bridge network
- Container shouldn't rely on IP's for inter-communication
- DNS for friendly names is built-in if you use custom networks
- Your using custom networks right?
- This gets way easier with Docker Compose in future section



NobleProg

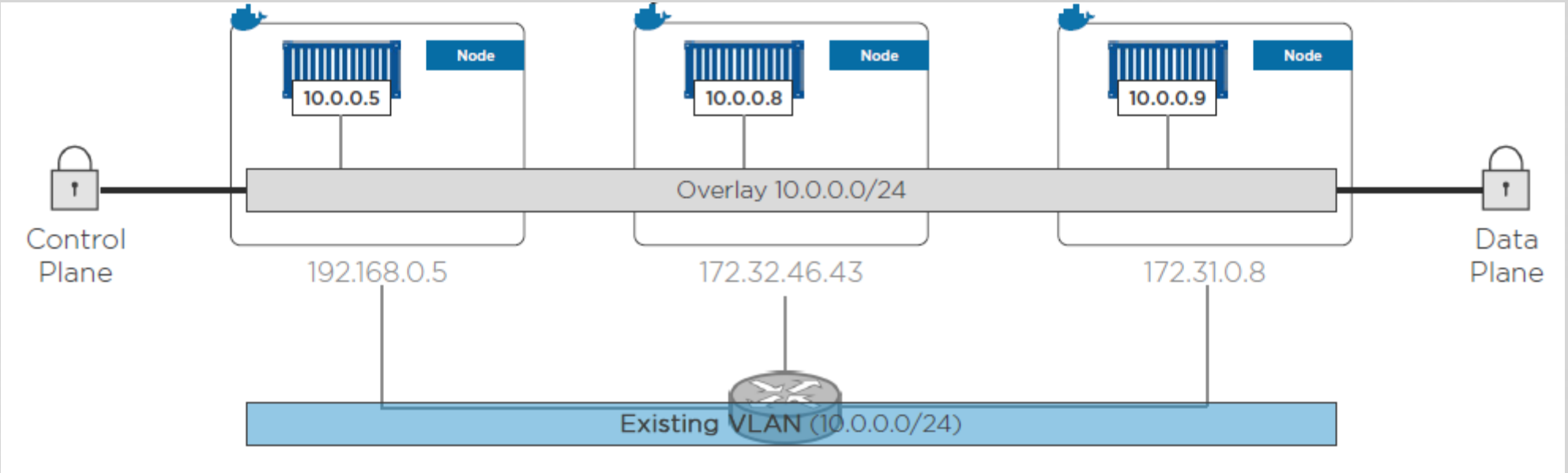
# Bridge Networking






NobleProg

# Overlay Networking






NobleProg

# Assignment : CLI App Testing

- Know how to use -it to get shell in container
- Understand basics of what a Linux distribution is like Ubuntu and CentOS
- Know how to run a container
- Use different Linux distro containers to check curl cli tool version
- Use two different terminal windows to start bash in both centos:7 and ubuntu:14.04, using -it
- Learn the docker container --rm option so you can save cleanup
- Ensure curl is installed and on latest version for that distro
  - ubuntu: apt-get update && apt-get install curl
  - centos: yum update curl
- Check curl --version



NobleProg

# Assignment: DNS Round Robin Test

- Ever since Docker Engine 1.11, we can have multiple containers on a created network respond to the same DNS address
- Create a new virtual network (default bridge driver)
- Create two container from elasticsearch:2 image
- Research and use --net-alias search when creating them to give them an additional DNS name to respond to
- Run alpine nslookup search with --net to see the two containers list for the same DNS name
- Run centos curl -s search:9200 with --net multiple time until you see both "name" fields show



NobleProg

# What's in this section

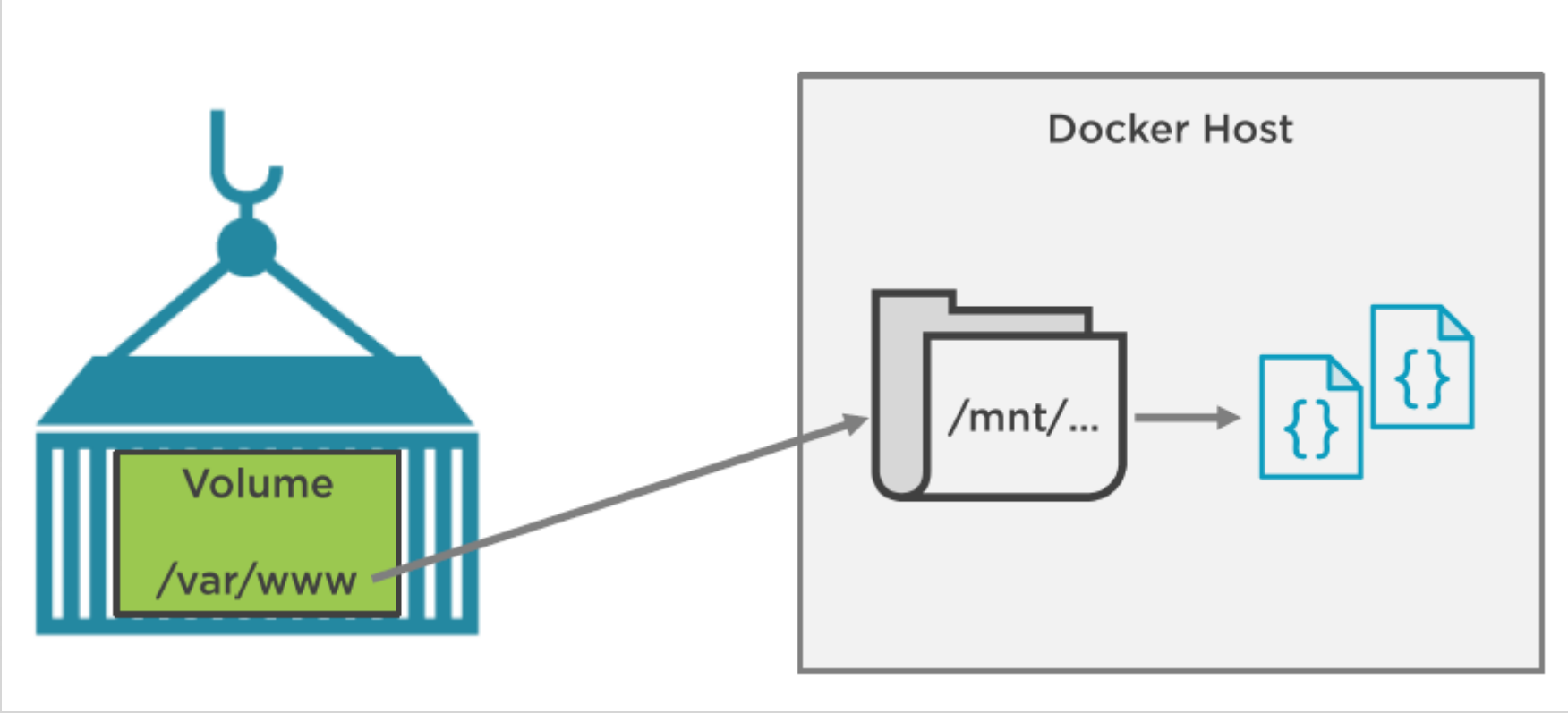
- Defining the problem of persistent data
- Key concepts with containers: immutable, ephemeral
- Learning and using Data Volumes
- Learning and using Bind Mounts
- Assignments




NobleProg

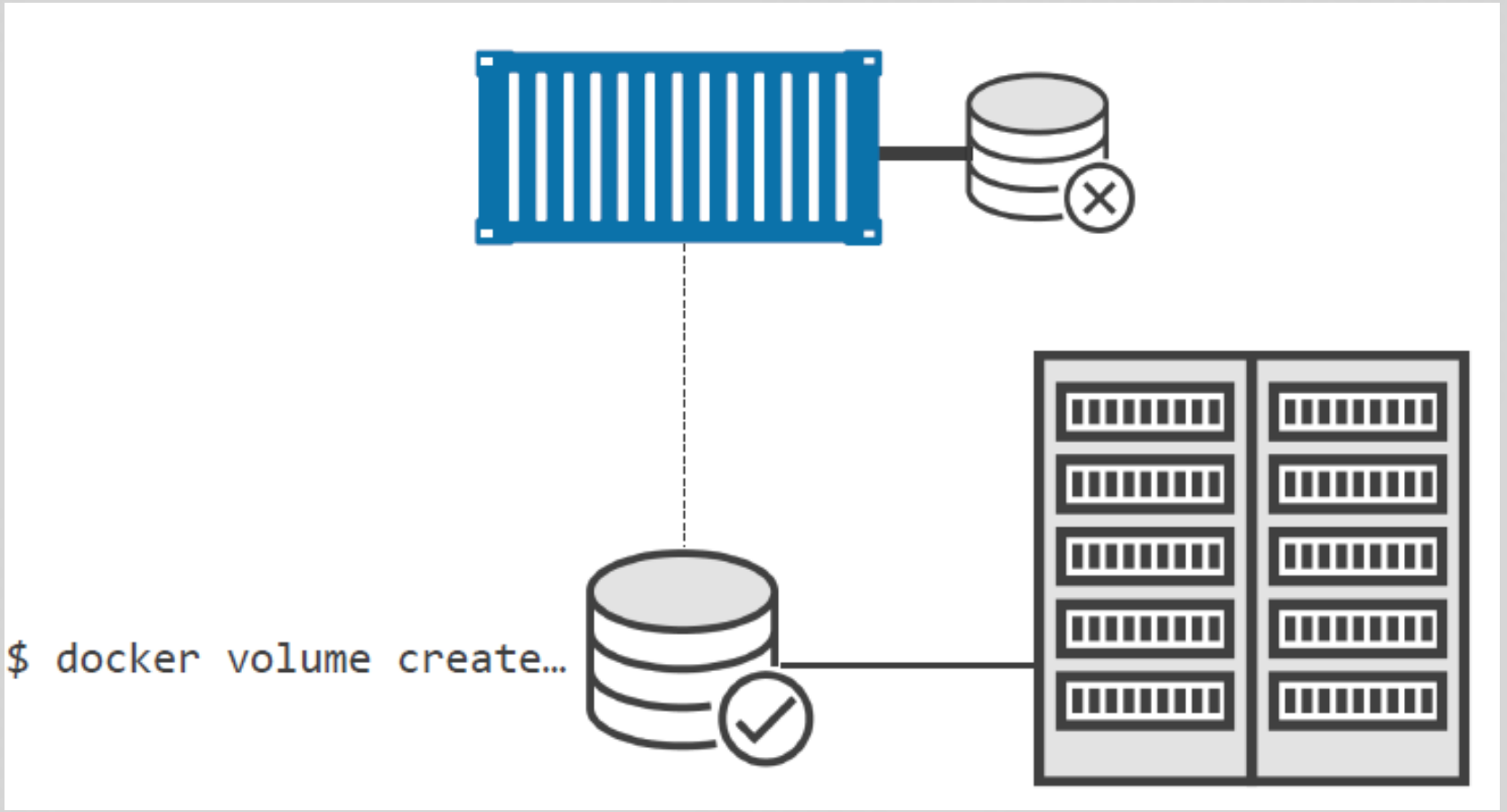
# What is a Volume?

- Special type of directory in a container typically referred to as a "data volume"
- Can be shared and reused among containers
- Updates to an image won't affect a data volume
- Data volumes are persisted even after the container is deleted



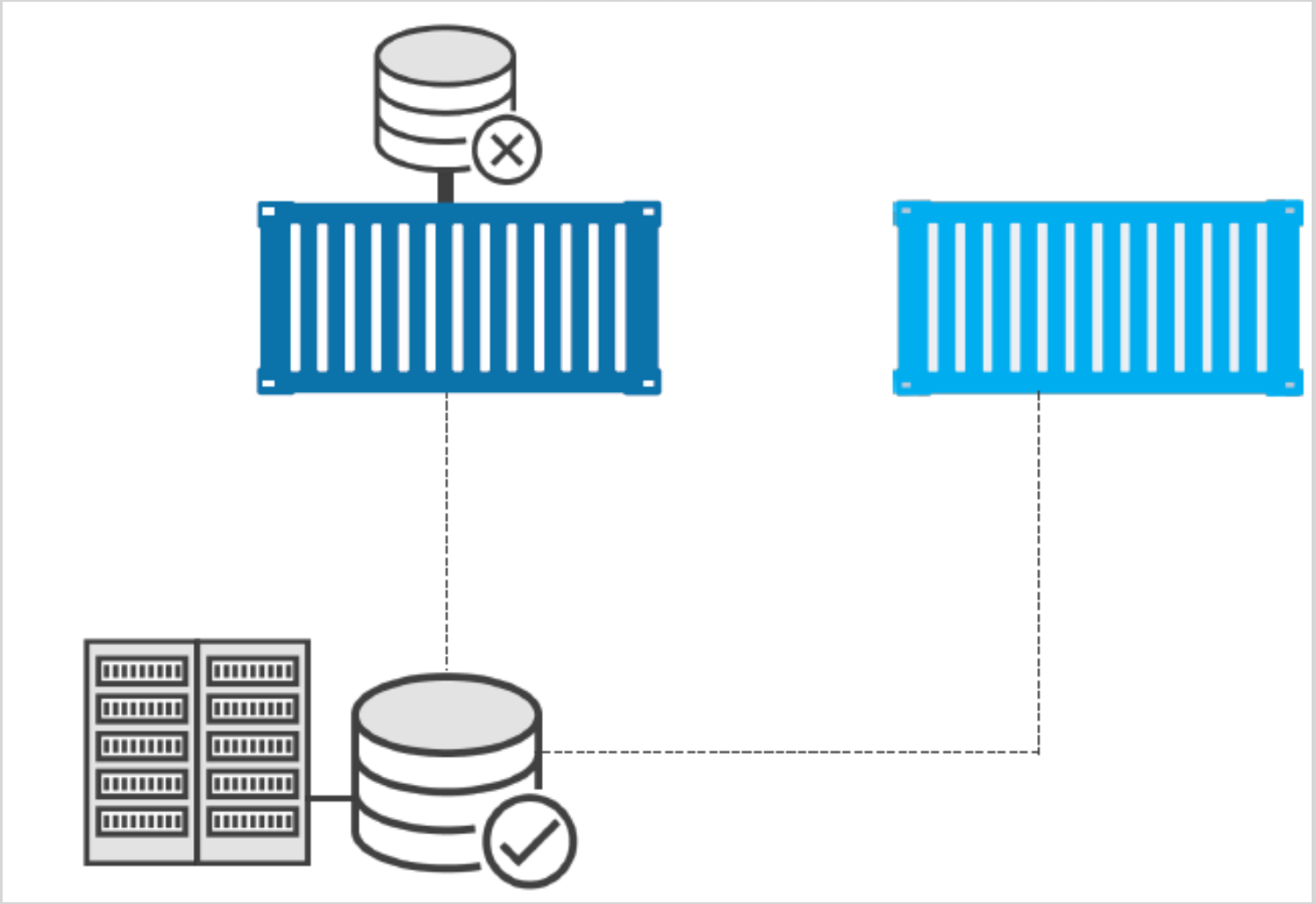



NobleProg






NobleProg



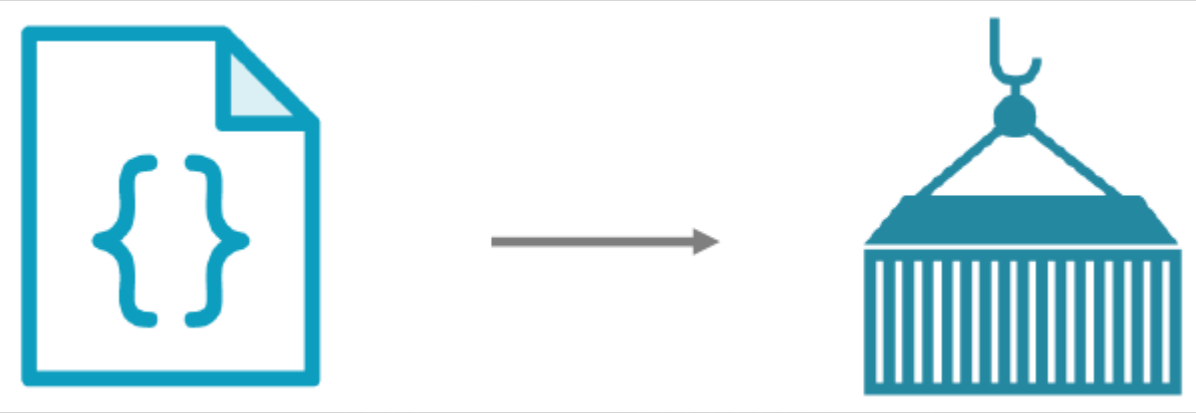


NobleProg

# Container Lifetime & Persistent Data

- Containers are usually immutable and ephemeral
- "immutable infrastructure": only re-deploy containers, never change. This is the ideal scenario, but what about databases, or unique data?
- Docker gives us features to ensure these "separation of concerns“. This is know as "persistent data"
- Two ways: Volumes and Bind Mounts
  - Volumes: make special location outside of container UFS
  - Bind Mounts: link container path to host path
- VOLUME command in Dockerfile

# How do you get source code into a container?



1. Create a container volume that points to the source code
2. Add your source code into a custom image that is used to create a container



NobleProg

# Persistent Data: Bind Mounting

- Maps a host file or directory to a container file or directory
- Basically just two locations pointing to the same file(s)
- Again, skip UFS, and host files overwrite any in container
- Can't use in Dockerfile, must be at container run
  - ... run -v /Users/laks/stuff:/path/container (mac/linux)
  - ... run -v //c/Users/laks/stuff:/path/conatiner (windows)

# Assignment: Named Volumes

- Database upgrade with containers
- Create a postgres container with named volume psql-data using version 9.6.1
- Use Docker Hub to learn VOLUME path and versions needed to run it
- Check logs, stop container
- Create a new postgres container with same named volume using 9.6.2
- Check logs to validate
- (this only works with patch versions, most SQL DB's require manual commands to upgrade DB's to major/minor versions. i.e, it's a DB limitation not a container one)

# Assignment: Bind Mounts

- Use a Jekyll "Static Site Generator" to start a local web server
- Don't have to be web developer: this is example of bridging the gap between local file access and apps running in containers
- Source code is in the course repo under bindmount-sample-1
- We edit files with editor on our host using native tools
- Container detects changes with host files and updates web server
- start container with docker run -p 80:4000 -v $(pwd):/site laks/jekyll-serve
- Refresh our browser to see changes
- Change the file in _posts\ and refresh browser to see changes



NobleProg

# Docker Compose

- Why: configure relationships between containers
- Why: save our docker container run settings in easy-to-read file
- Why: create one-liner developer environment startups
- Comprised of 2 separate but related things
    1. YAML-formatted file that describes our solution options for:
        containers
        networks
        volumes
    2. A CLI tool docker-compose used for local dev/test automation with those YAML files

# docker-compose.yml

- Compose YAML format has it's own versions: 1, 2, 2.1, 3, 3.1
- YAML file can be used with docker-compose command for local docker automation or..
- with docker directly in production with Swarm(as of v1.13)
- docker-compose --help
- docker-compose.yml is default filename, but any can be used with docker-compose -f

# docker-compose CLI

- CLI tool comes with Docker for Windows/Mac, but separate download for Linux
- Not a production-grade tool but ideal for local development and test
- Two most common commands are
  - docker-compose up #setup volumes/networks and start all containers
  - docker-compose down #stop all containers and remove cont/vol/net
- If all your projects had a Dockerfile and docker-compose.yml then "new developer onboarding" would be:
  - git clone github.com/some/software
  - docker-compose up



NobleProg

# Assignment: Writing a Compose File

- Build a basic compose file for a Drupal content management system website. Docker Hub is your friend.
- Use the drupal image along with the postgres image
- Use ports to expose Drupal on 8080 so you can localhost:8080
- Be sure to set POSTGRES_PASSWORD for postgres
- Walk through Drupal setup via browser
- Tip: Drupal assumes DB is localhost, but it's service name
- Extra Credit: Use volumes to store Drupal unique data

# Using Compose to Build

- Compose can also build your custom images
- Will build them with docker-compose up if not found in cache
- Also rebuild with docker-compose build
- Great for complex builds that have lots of vars or build args

# Assignment: Build and Run Compose

- Building custom drupal image for local testing
- Compose isn't just for developers. Testing apps is easy/fun!
- Maybe your learning Drupal admin, or are a software tester
- Start with Compose file from previous assigment
- Make your Dockerfile and docker-compose-yml in dir compose-assignment-2
- Use the drupal image along with the postgres image as before
- Use README.md in that dir for details



NobleProg

# Containers Everywhere = New Problems

- How do we automate container lifecycle?
- How can we easily scale out/in/up/down?
- How can we ensure our containers are re-created if they fail?
- How can we replace containers without downtime (blue/green deploy)?
- How can we control/track where containers get started?
- How can we create cross-node virtual networks?
- How can we ensure only trusted servers run our containers?
- How can we store secrets, keys, passwords and get them to the right container(and only that container)?

# Container Orchestration

**Container orchestrators** are the tools which group hosts together to form a cluster, and help us fulfill the requirements below,

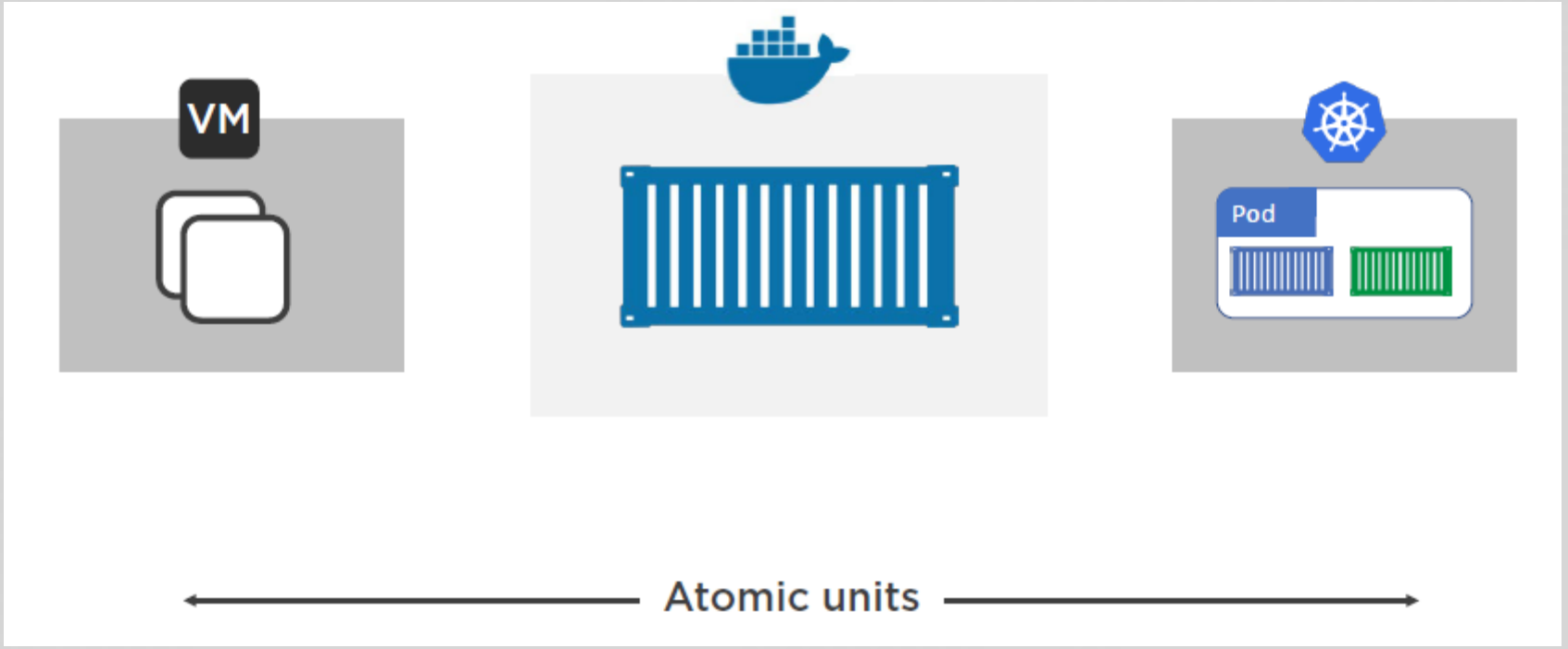
- Are fault-tolerant
- Can scale, and do this on-demand
- Use resources optimally
- Can discover other applications automatically, and communicate with each other
- Are accessible from the external world
- Can update/rollback without any downtime.

- Docker Swarm is a container orchestrator provided by Docker, Inc. It is part of Docker Engine.
- Kubernetes was started by Google, but now, it is a part of the Cloud Native Computing Foundation project.
- Marathon is one of the frameworks to run containers at scale on Apache Mesos.
- Amazon EC2 Container Service (ECS) is a hosted service provided by AWS to run Docker containers at scale on its infrastructure.
- Nomad is the container orchestrator provided by HashiCorp.

# Container Orchestration

Container orchestrators can:

- Bring multiple hosts together and make them part of a cluster
- Schedule containers to run on different hosts
- Help containers running on one host reach out to containers running on other hosts in the cluster
- Bind containers and storage
- Bind containers of similar type to a higher-level construct, like services, so we don't have to deal with individual containers
- Keep resource usage in-check, and optimize it when necessary
- Allow secure access to applications running inside containers.






NobleProg

# Swarm Mode: Built-In Orchestration

- Swarm Mode is a clustering solution built inside Docker
- Not related to Swarm "classic" for pre-1.12 versions
- Added in 1.12(Summer 2016) via SwarmKit toolkit
- Enhanced in 1.13(January 2017) via Stacks and Secrets
- Not enabled by default, new commands once enabled

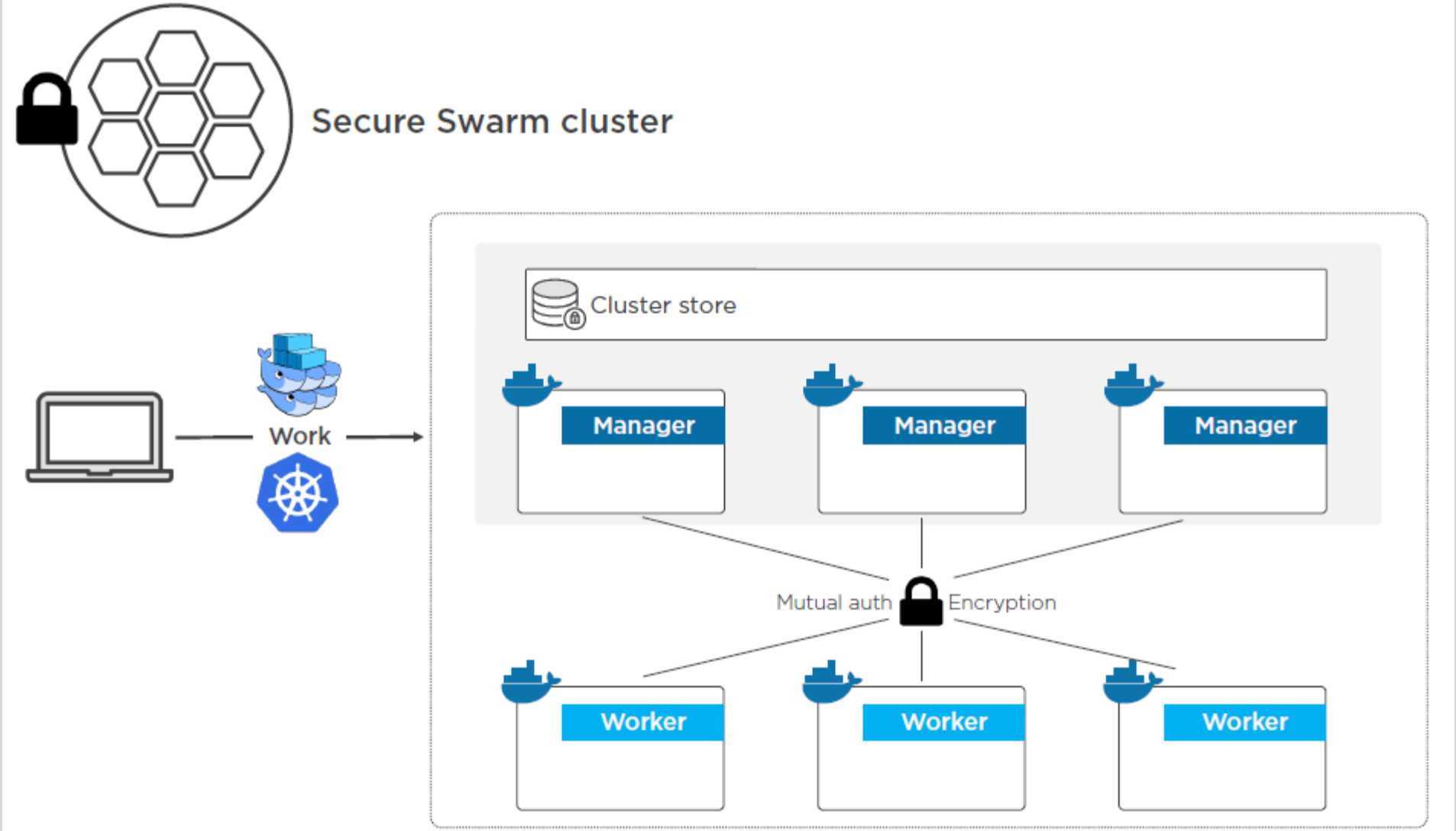
  docker swarm

  docker node

  docker service

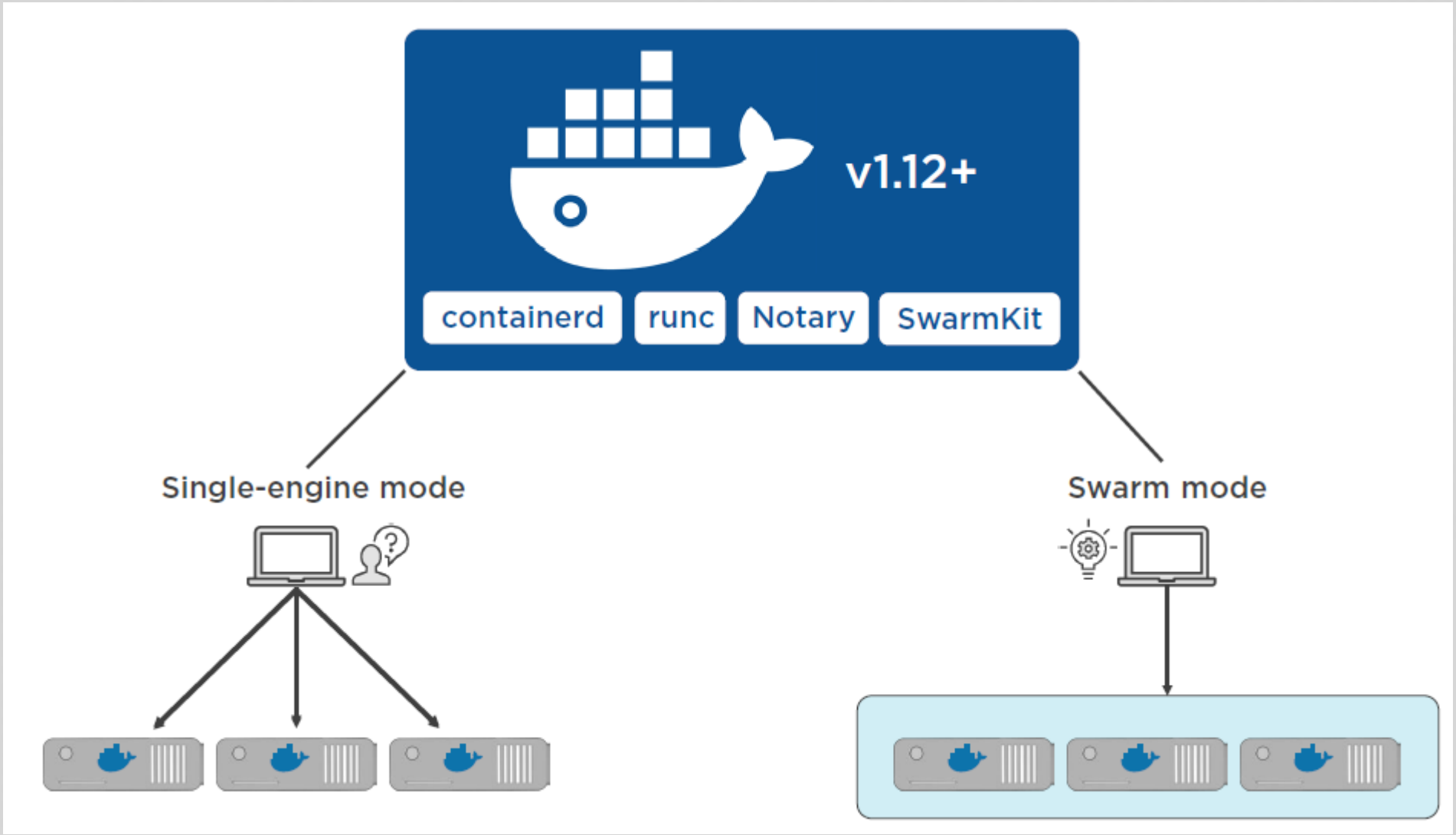
  docker stack

  docker secret



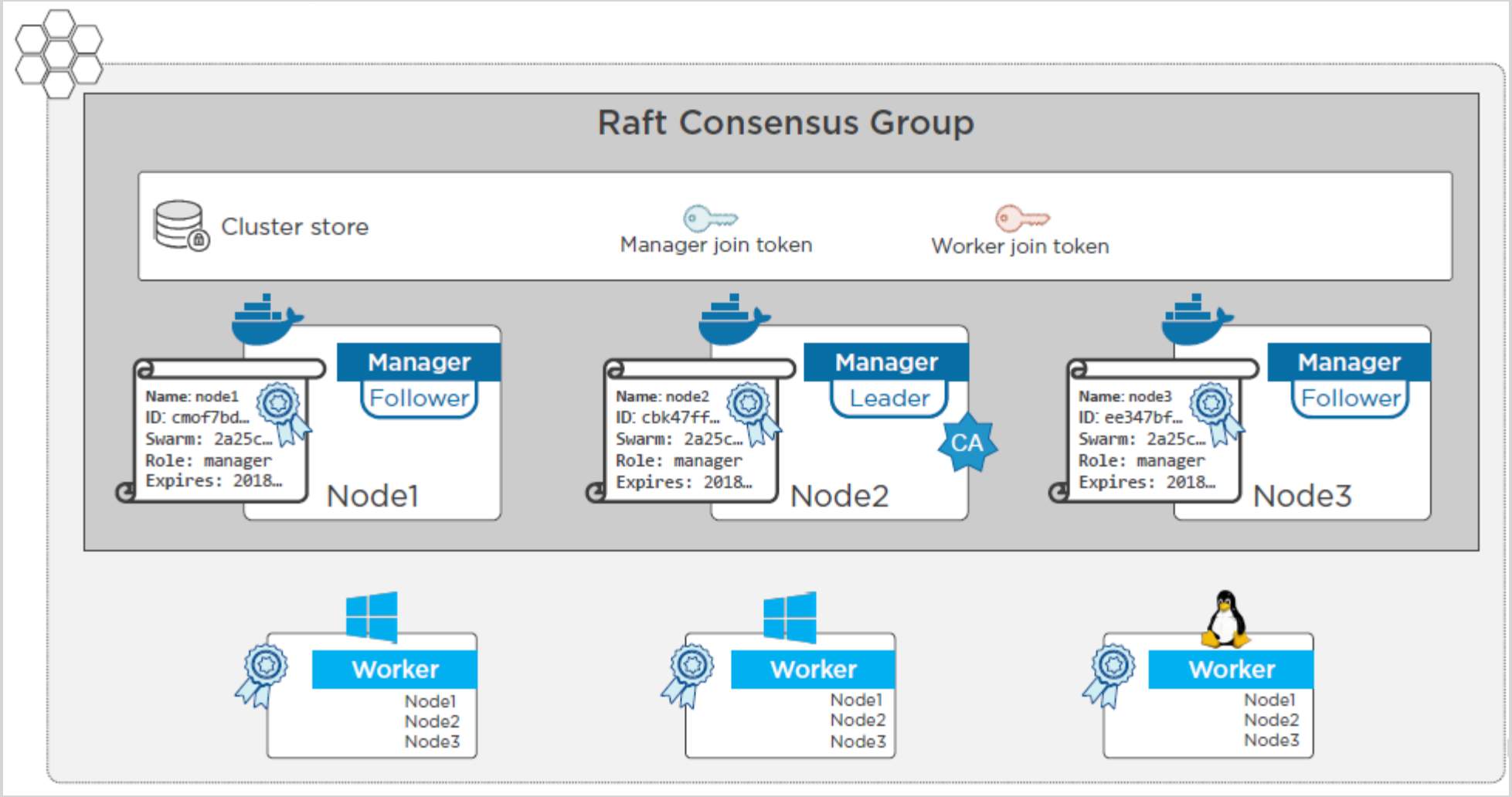



NobleProg






NobleProg






NobleProg

# docker swarm init

- Lots of PKI and security automation
  - Root signing certificate created for our Swarm
  - Certificate is issued for first Manager node
  - Join tokens are created
- Raft database created to store root CA, configs and secrets
  - Encrypted by default on disk(1.13+)
  - No need for another key/value system to hold orchestration/secrets
  - Replicates logs amongst Managers via mutual TLS in "control plane"

# Creating 3-Node Swarm: Host Options

A. play-with-docker.com

Only needs a browser, but resets after 4 hours

B. docker-machine + VirtualBox

Free and runs locally, but requires a machine with 8GB memory

C. Digital Ocean+Docker install

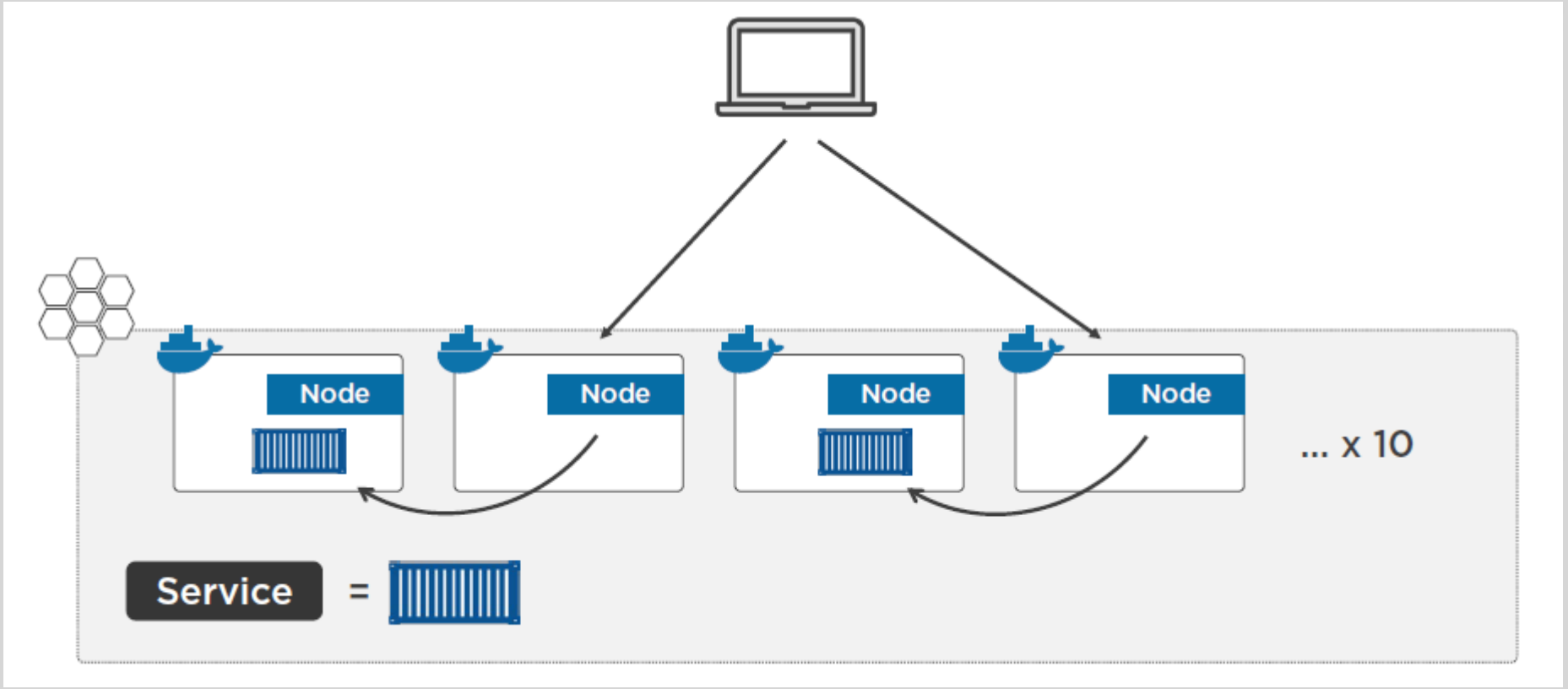
Most like a production setup, but costs $5-10/node/month while learning

D. Roll your own

docker-machine can provision machines for Amazon, Azure, DO, Google, etc.

Install docker anywhere with get.docker.com

# Load Balancing





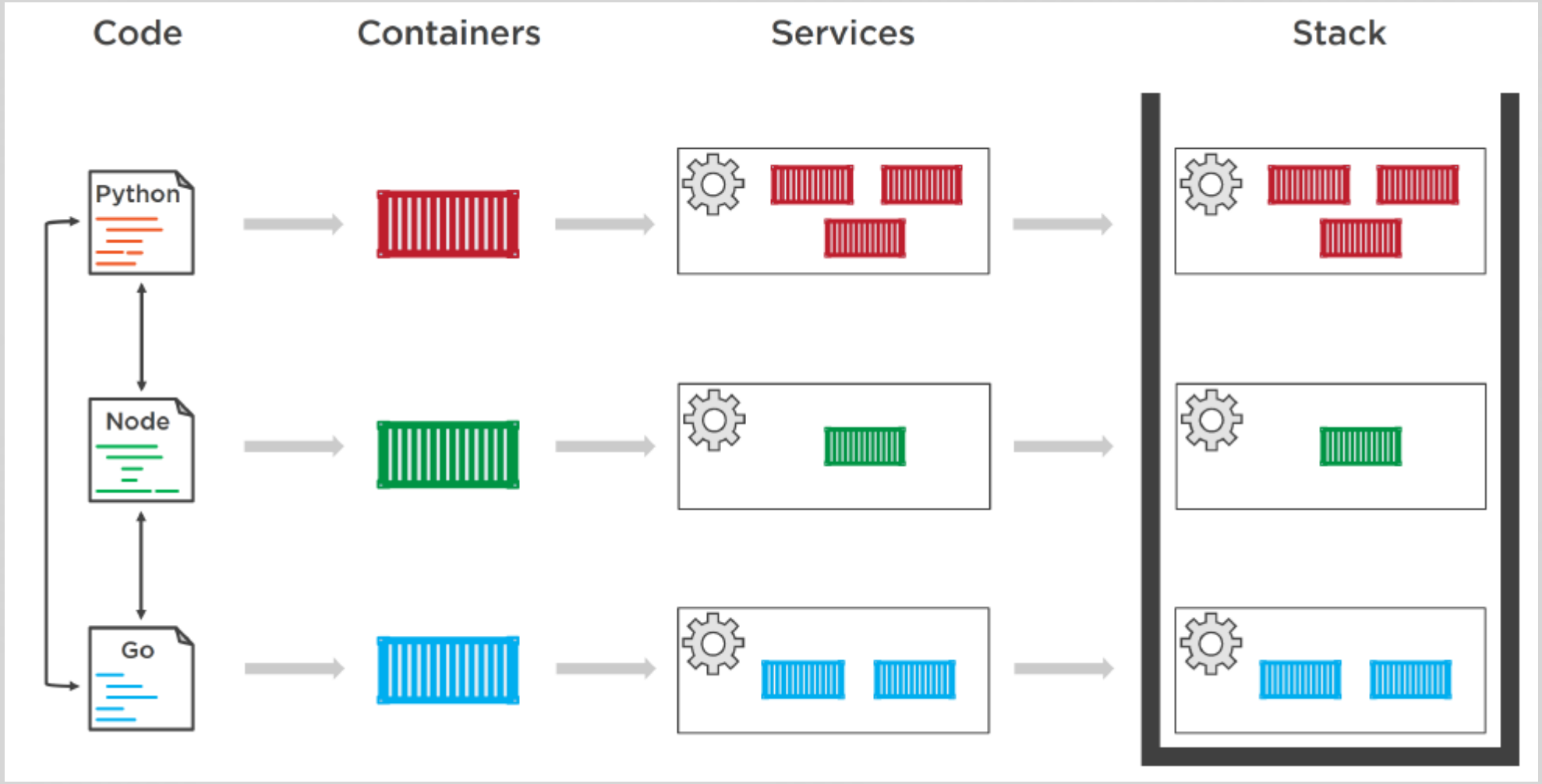

NobleProg

# Assignment: Create Multi-Service App

- Using Docker's Distributed Voting App
- use swarm-app-1 directory in our course repo for requirements
- 1 volume, 2 networks, and 5 services needed
- Create the commands needed, spin up services, and test app
- Everything is using Docker Hub images, so no data needed on swarm
- Like many computer things, this is 1/2 art form and 1/2 science.

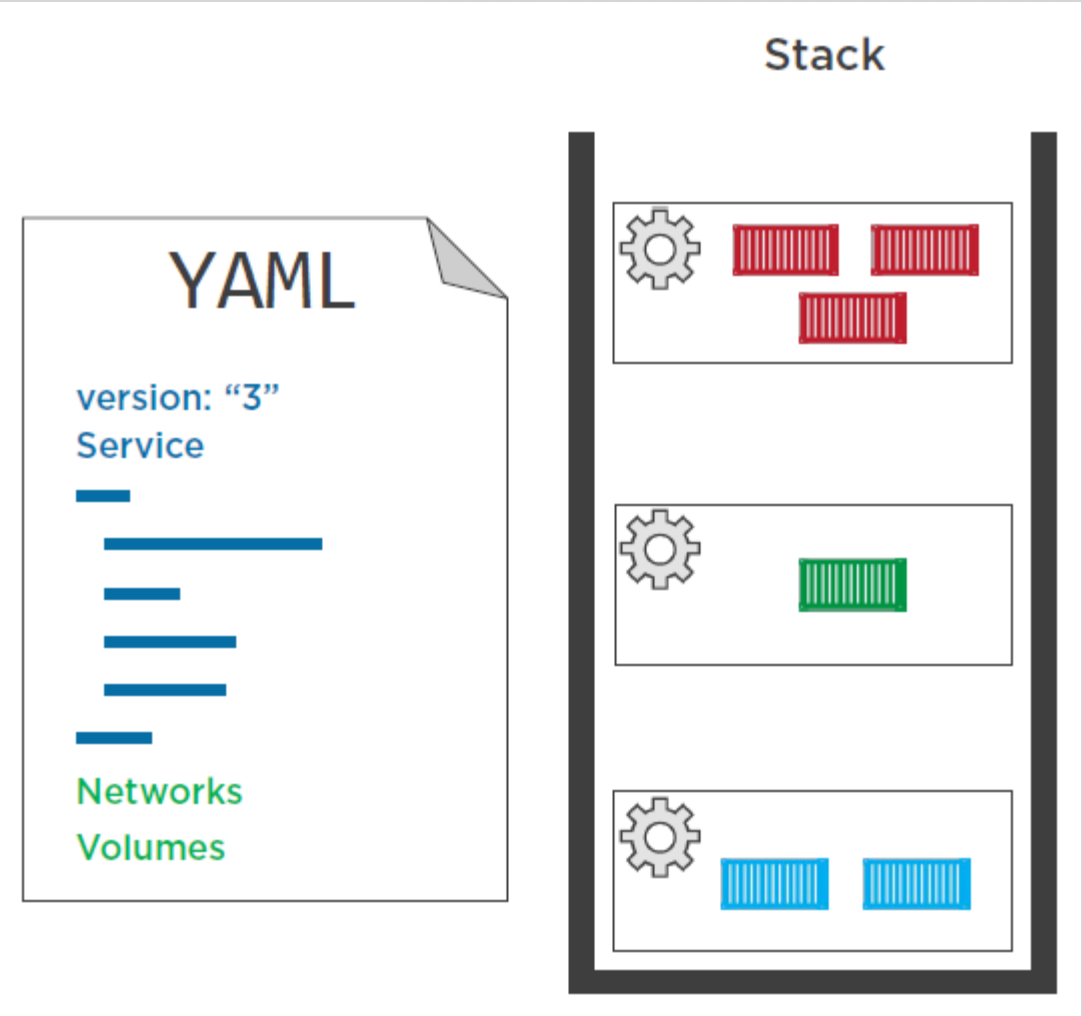
# Stacks: Production Grade Compose

- In 1.13 Docker adds a new layer of abstraction to Swarm called Stacks
- Stacks accept Compose files as their declarative definition for services, networks, and volumes
- We use docker stack deploy rather than docker service create
- Stacks manages all those objects or us, including overlay network per stack. Adds stack name to start of their name.
- New deploy: key in Compose file. Can't do build:
- Compose now ignores deploy:, Swarm ignores build:
- docker-compose cli not needed on Swarm server



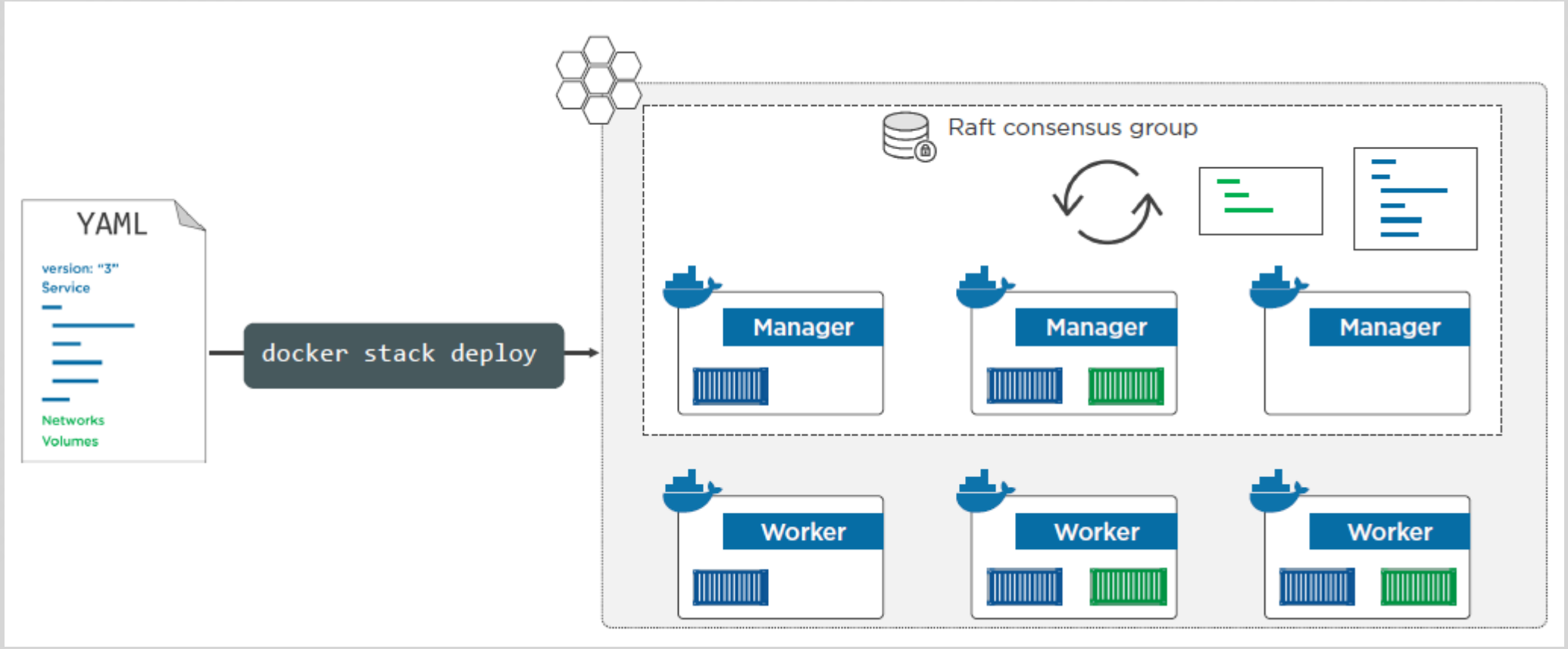



NobleProg






NobleProg






NobleProg

# Secrets Storage

- Easiest "secure" solution for storing secrets in Swarm
- What is Secret?

  Usernames and passwords

  TLS certificates and keys

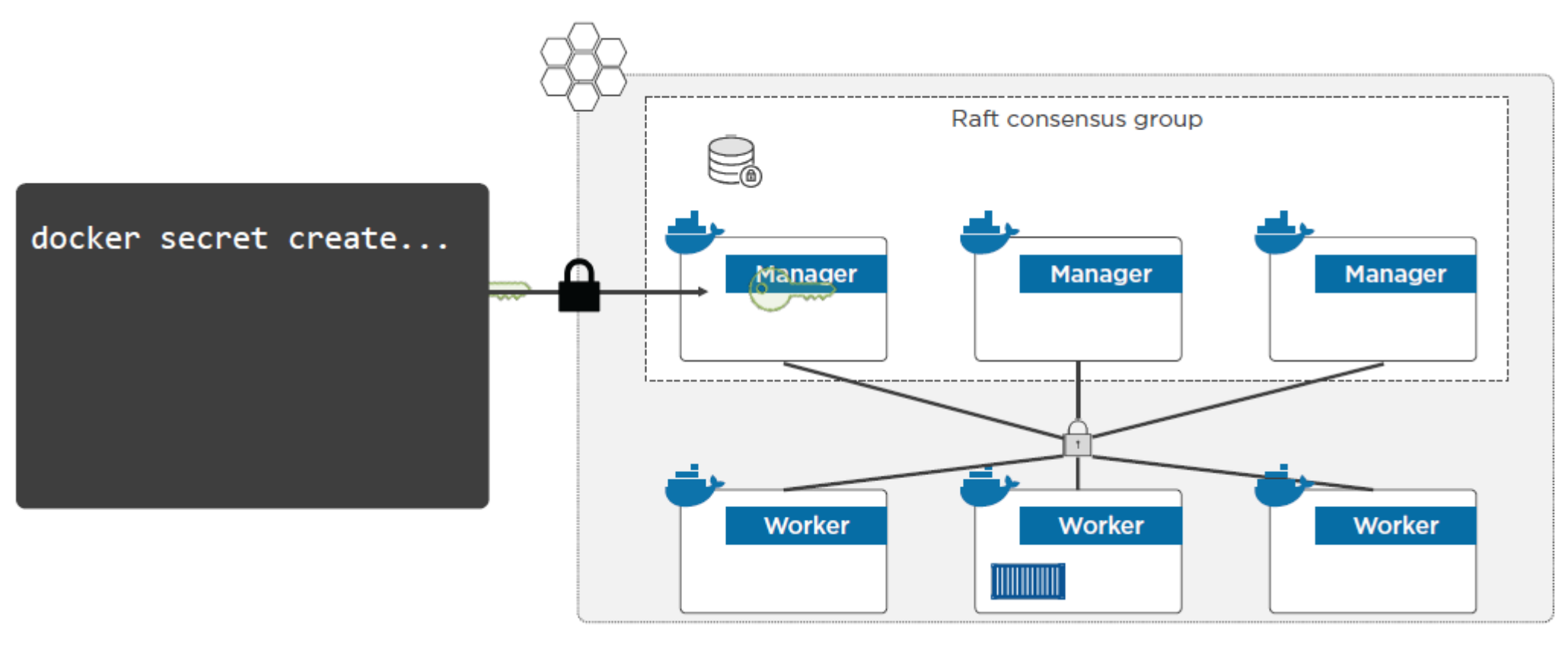
  SSH keys

  Any data you would prefer not be "on front page of news"
- Supports generic strings or binary content up to 500kb in size
- Doesn't require apps to be rewritten
- As of Docker 1.13.0 Swarm Raft DB is encrypted on disk
- Only stored on disk on Manger nodes
- Default is Managers and Workers "control plane" is TLS+Mutual Auth
- Secrets are first stored in Swarm, then assigned to a Service(s)
- Only Containers in assigned Service(s) can see them
- They look like files in container but are actually in-memory fs

  /run/secrets/<secret_name> or

  /run/secrets/<secret_alias>
- Local docker-compose can use file-based secrets, but not secure

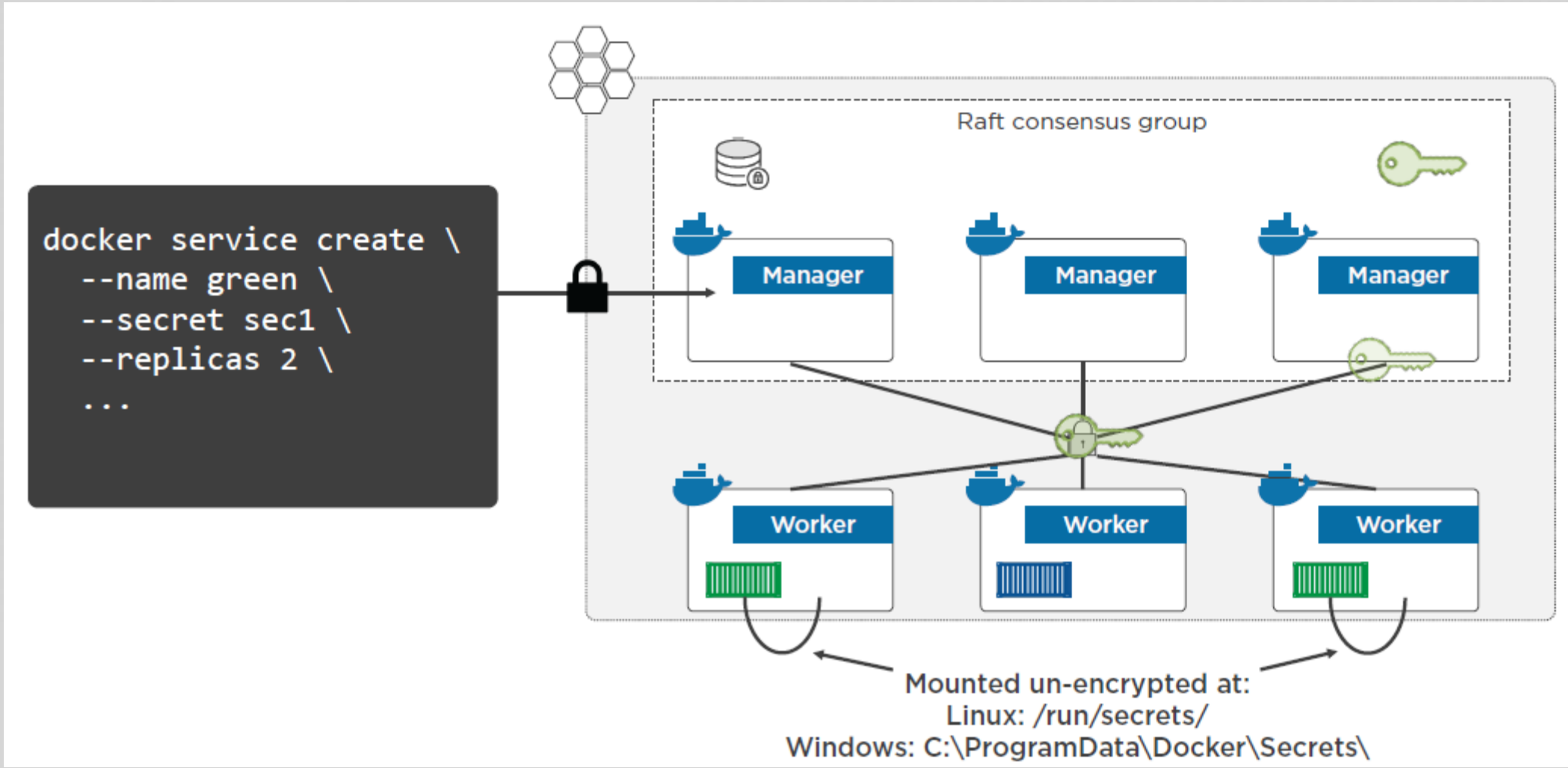
# Docker Secrets






NobleProg

```
docker service create \
  --name green \
  --secret sec1 \
  --replicas 2 \
  ...
```






NobleProg

# Assignment: Create Stack w/ Secrets

- Let's use our Drupal compose file from last assignment
  - compose-asssignment-2
- Rename image back to official drupal:8.2
- Remove build:
- Add secret via external:
- use environment variable POSTGRES_PASSWORD_FILE
- Add secret via cli echo "<pw>" | docker secret create psql-pw -
- Copy compose into a new yml file on you Swarm node1

# Service Updates

- Provides rolling replacement of tasks/containers in a service
- Limits downtime(be careful with "prevents" downtime)
- Will replace containers for most changes
- Has many, many cli options to control the update
- Create options will usually change, addding -add or -rm to them
- Includes rollback and healthcheck options
- Also has scale & rollback subcommand for quicker access

  docker service scale web=4 and docker service rollback web
- A stack deploy, when pre-existing, will issue service updates

# Swarm Update Examples

- Just update the image used to a newer version

  docker service update --image myapp:1.2.1 <servicename>

- Adding an environmental variable and remove a part

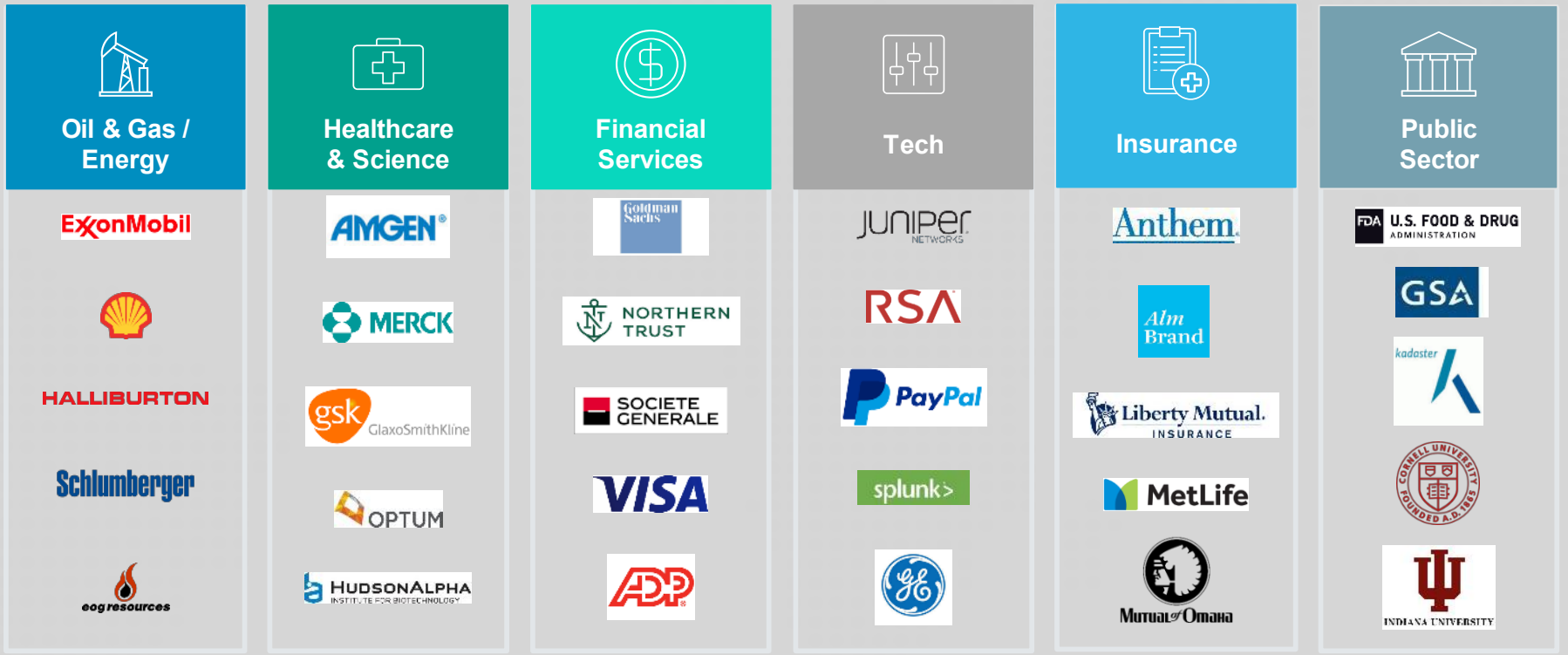
  docker service update --env-add NODE_ENV=production --publish-rm 8080

- Change number of replicas of two services

  docker service scale web=8 api=6

- Swarm Updates in Stack files
  - Same command. Just edit the YAML file, then
  - docker stack deploy -c file.yml <stackname>

# 400+ Industry Leaders Build Their Containerization Strategy on Docker






NobleProg

VISA

# Supporting Global Growth with Microservices

## KEY CHALLENGES

Infrastructure & tech refresh demands growing while IT resources remain flat.

## SOLUTION

Use Docker EE to refactor critical transaction processing applications

## RESULTS

- Deployed applications globally: grew transactions from 100K to millions daily
- Provision in seconds rather than days or weeks
- Container efficiencies = 10x increase in scalability

> *The more we move workloads to microservices, the better the efficiency.*

Global Head of Infrastructure and Operations




NobleProg

MetLife

# Forecasting 66% Cost Savings with Faster App Delivery

## KEY CHALLENGES

Enable innovation by reducing TCO of existing apps and infrastructure

## SOLUTION

Docker Enterprise Edition for Modernizing Traditional Applications

## RESULTS

- 66% TCO reduction for single technology stack
- 70%+ VM consolidation
- 10X increase in CPU utilization
- 3X improvement in application delivery, from 15 to 5 months

> *Docker Enterprise Edition creates a self-funding model to fuel change and innovation at scale.*

Jeff Murr
Director of Platform Engineering, MetLife



NobleProg

PayPal

# Managing 150,000 Containers to Speed Transactions

## KEY CHALLENGES

Process 200 user payments per second across PayPal, Braintree and Venmo, with different architectures and clouds

## SOLUTION

Unified application delivery, operations and centralized CaaS management for consistent packaging of existing applications

## RESULTS

- Migrated 700+ apps to Docker, running in 150,000 containers
- 50% productivity increase in building, testing and deploying applications
- 50% increase in QA CPU utilization; 25% increase in production utilization

> *With Docker CaaS, we are deprecating 15 years worth of toolsets and building a consistent operating model across multiple clouds.*

Meghdoot Bhattacharya, Cloud Engineer



NobleProg

GlaxoSmithKline

# Using Containers to Accelerate Drug Discovery

## KEY CHALLENGES

Faster delivery of technology stacks to individual research teams enabling faster mining of clinical trials and research data

## SOLUTION

Docker EE's "Edge Node On Demand" platform secures and isolates research environments

## RESULTS

- 10 cluster, 300+ node deployment of Cloudera & MongoDB with Active Directory
- Centralized different app stacks in one interface giving each researcher an isolated, secure sandbox

> *Docker Enterprise Edition allows GSK to support a multitude of tools and technologies and interfaces so that scientists can run data analysis at scale.*

Ranjith Raghunath
Director of Big Data Solutions



NobleProg

SOCIETE GENERALE

# Migrating Thousands of Applications to the Cloud

## KEY CHALLENGES

150+ year old banks needed to accelerate innovation and cut costs by moving 80% of apps to the cloud by 2020 across multiple data centers

## SOLUTION

Use Docker Enterprise to build a CaaS solution more flexible than PaaS to support thousands of existing applications

## RESULTS

- First 10 applications in production, with 50 more in development
- 400 developers now using Docker EE
- Integrated Docker EE with CI/CD tools: Jenkins, GitHub, and Nexus

> *"With Docker Enterprise Edition, everyone wants to work with containers; everyone wants to work with Docker and it's a change of mindset in the company."*
>
> *Thomas Boussardon*
> *Middleware Specialist*



NobleProg

# Typical ROI Results from Customers

## Infrastructure Savings

| | |
|---|---|
| • Hardware and software licensing savings | 20% – 40% |
| • Operations support savings | 20% – 40% |
| • Reduction in number of virtual machines (VMs) | 50% – 90% |

## Productivity Gains

30% – 60%

| | |
|---|---|
| • Agility | Deploy and scale in minutes |
| • Security | Secure app isolation, threat scanning, and monitoring |
| • Portability | Streamlined workflow from development to production |



NobleProg

# Kubernetes

*"Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."*

- **Kubernetes** comes from the Greek word **κυβερνήτης:**, which means *helmsman* or *ship pilot*.
- Kubernetes is also referred to as **k8s**, as there are 8 characters between *k* and *s*.
- Kubernetes is highly inspired by the Google Borg system
- It is an open source project written in the Go language, and licensed under the Apache License Version 2.0.
- Kubernetes was started by Google and, with its v1.0 release in July 2015, Google donated it to the Cloud Native Computing Foundation (CNCF).
- Generally, Kubernetes has new releases every three months. The current stable version is 1.13 (as of February 2019).

# Kubernetes Features

| | |
|---|---|
| **Automatic binpacking** | Kubernetes automatically schedules the containers based on resource usage and constraints, without sacrificing the availability. |
| **Self-healing** | Kubernetes automatically replaces and reschedules the containers from failed nodes. It also kills and restarts the containers which do not respond to health checks, based on existing rules/policy. |
| **Horizontal scaling** | Kubernetes can automatically scale applications based on resource usage like CPU and memory. In some cases, it also supports dynamic scaling based on customer metrics. |
| **Service discovery and Load balancing** | Kubernetes groups sets of containers and refers to them via a Domain Name System (DNS). This DNS is also called a Kubernetes **service**. Kubernetes can discover these services automatically, and load-balance requests between containers of a given service. |
| **Automated rollouts and rollbacks** | Kubernetes can roll out and roll back new versions/configurations of an application, without introducing any downtime. |
| **Secrets and configuration management** | Kubernetes can manage secrets and configuration details for an application without re-building the respective images. With secrets, we can share confidential information to our application without exposing it to the stack configuration, like on GitHub. |
| **Storage orchestration** | With Kubernetes and its plugins, we can automatically mount local, external, and storage solutions to the containers in a seamless manner, based on software-defined storage (SDS). |
| **Batch execution** | Besides long running jobs, Kubernetes also supports batch execution. |

# Kubernetes Users

We can find numerous user case studies on the Kubernetes website:

- Pearson
- Box
- eBay
- Wikimedia
- Huawei
- Haufe Group
- BlackRock
- BlaBlaCar
- And many more.

# Cloud Native Computing Foundation (CNCF)

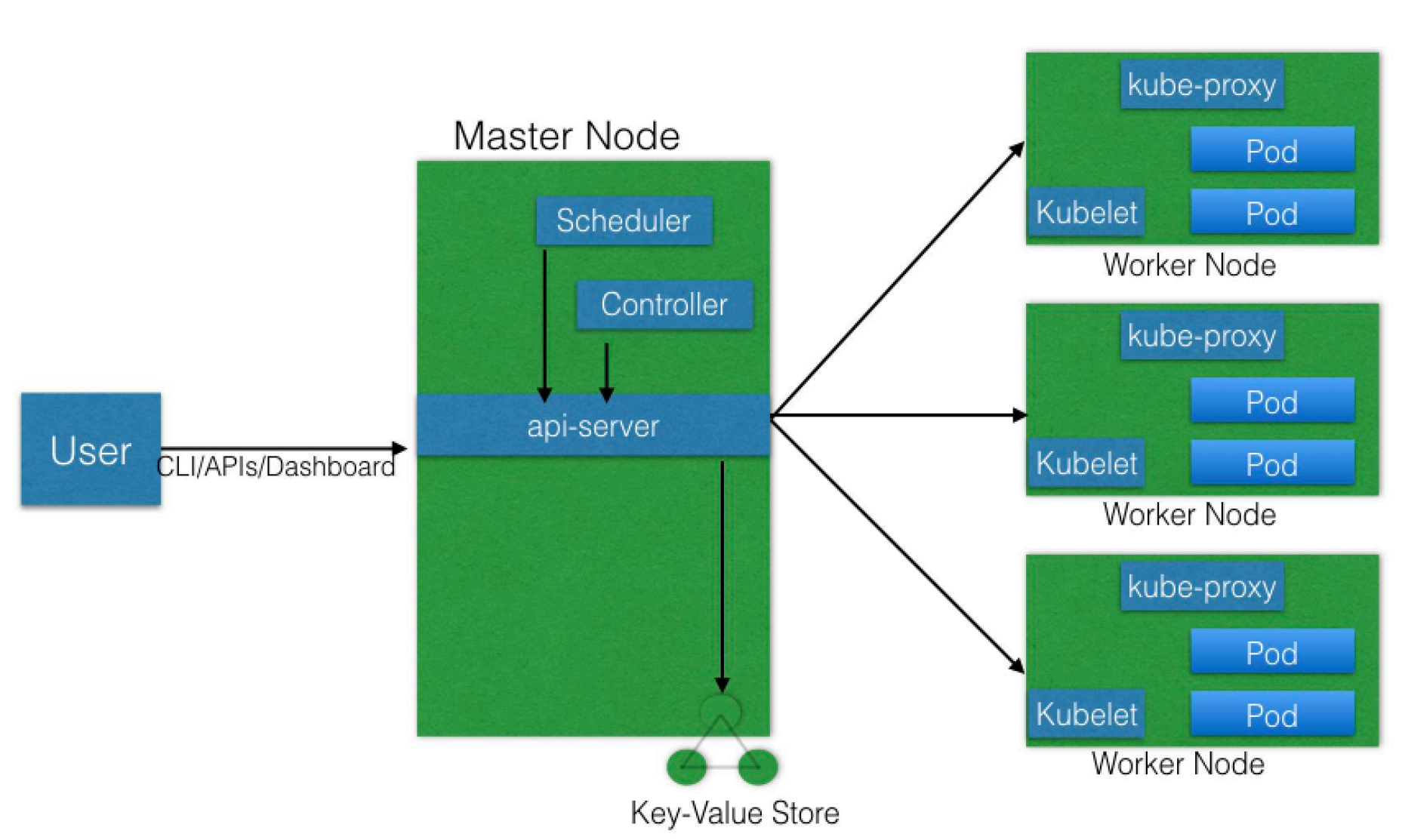
- The Cloud Native Computing Foundation (CNCF) is one of the projects hosted by The Linux Foundation. CNCF aims to accelerate the adoption of containers, microservices, and cloud-native applications.

- The following projects were part of CNCF:
  - containerd for container runtime
  - rkt for container runtime
  - Kubernetes for container orchestration
  - Linkerd for service mesh
  - Envoy for service mesh
  - gRPC for remote procedure call (RPC)
  - Container Network Interface (CNI) for networking API
  - CoreDNS for service discovery
  - Rook for cloud-native storage
  - Notary for security
  - The Update Framework (TUF) for software updates
  - Prometheus for monitoring
  - OpenTracing for tracing
  - Jaeger for distributed tracing
  - Fluentd for logging
  - Vitess for storage



NobleProg

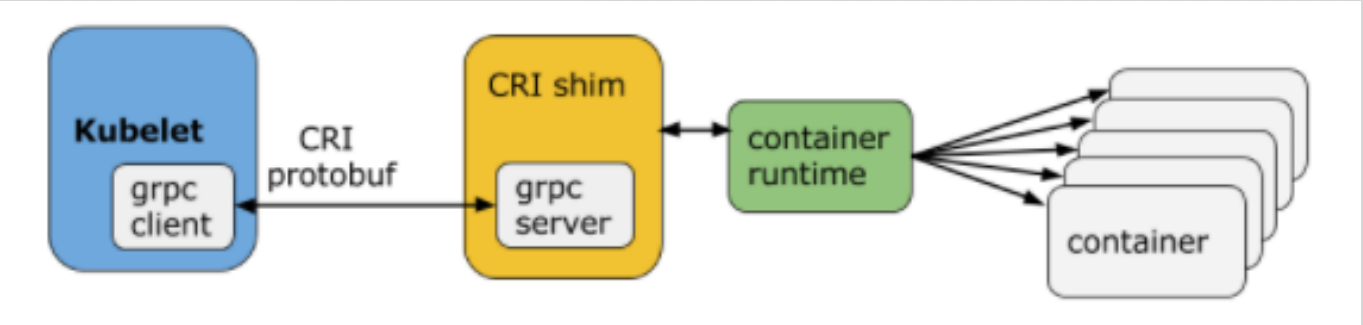
# Kubernetes Architecture

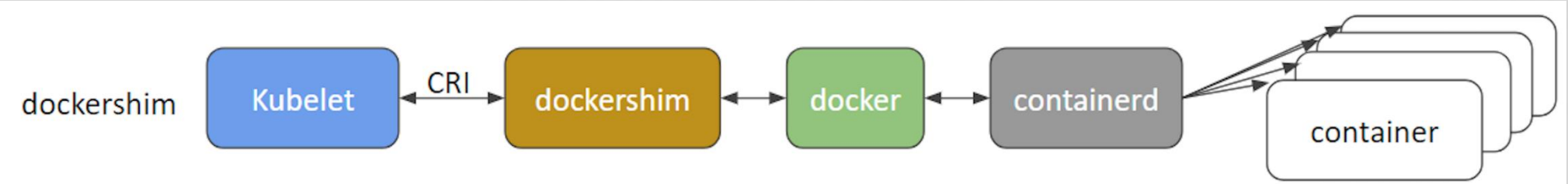


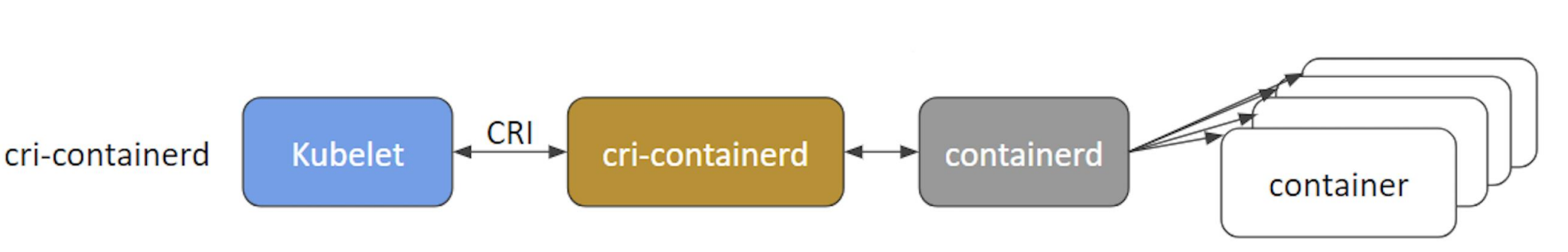



NobleProg

# kubelet






NobleProg

# Kubernetes Configuration

- All-in-One Single-Node Installation
- Single-Node etcd, Single-Master, and Multi-Worker Installation
- Single-Node etcd, Multi-Master, and Multi-Worker Installation
- Multi-Node etcd, Multi-Master, and Multi-Worker Installation

# Infrastructure for Kubernetes Installation

Once we decide on the installation type, we also need to make some infrastructure-related decisions, such as:

- Should we set up Kubernetes on bare metal, public cloud, or private cloud?
- Which underlying system should we use? Should we choose RHEL, CoreOS, CentOS, or something else?
- Which networking solution should we use?
- And so on.

The Kubernetes documentation has details in regards to choosing the right solution.

# Localhost Installation

There are a few localhost installation options available to deploy single- or multi-node Kubernetes clusters on our workstation/laptop:

- Minikube
- Ubuntu on LXD.

Minikube is the preferred and recommended way to create an all-in-one Kubernetes setup.

# On-Premise Installation

Kubernetes can be installed on-premise on VMs and bare metal.

- **On-Premise VMs -** Kubernetes can be installed on VMs created via Vagrant, VMware vSphere, KVM, etc. There are different tools available to automate the installation, like Ansible or kubeadm.
- **On-Premise Bare Metal -** Kubernetes can be installed on on-premise bare metal, on top of different operating systems, like RHEL, CoreOS, CentOS, Fedora, Ubuntu, etc. Most of the tools used to install VMs can be used with bare metal as well.

# Cloud Installation

Hosted Solutions

- Google Kubernetes Engine (GKE)
- Azure Container Service (AKS)
- Amazon Elastic Container Service for Kubernetes (EKS)
- OpenShift Dedicated
- Platform9
- IBM Cloud Container Service.

Turnkey Cloud Solutions

- Google Compute Engine
- Amazon AWS
- Microsoft Azure
- Tectonic by CoreOS.

Bare Metal

- Kubernetes can be installed on bare metal provided by different cloud providers.



NobleProg

# Kubernetes Installation Tools/Resources

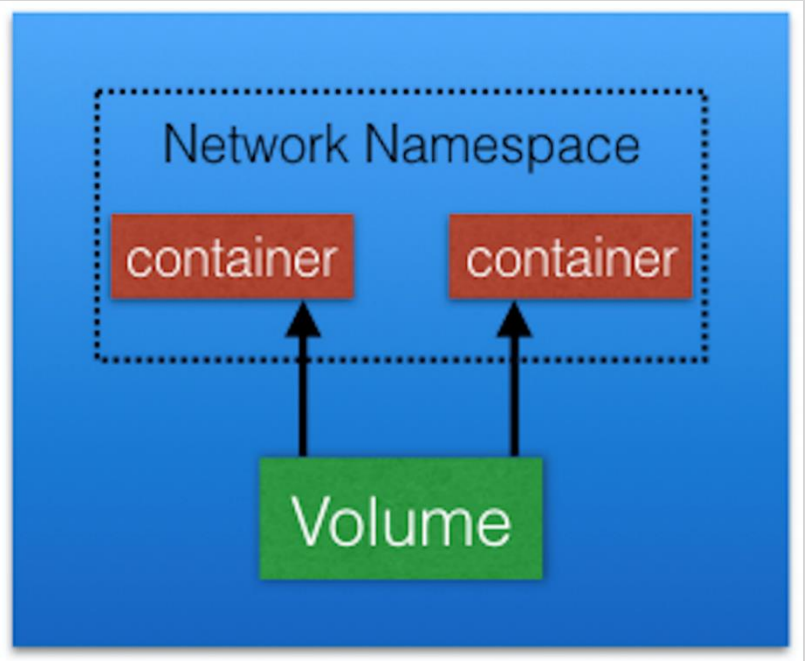
- kubeadm is a first-class citizen on the Kubernetes ecosystem. It is a secure and recommended way to bootstrap the Kubernetes cluster. It has a set of building blocks to setup the cluster, but it is easily extendable to add more functionality. Please note that kubeadm does not support the provisioning of machines.
- With KubeSpray (formerly known as Kargo), we can install Highly Available Kubernetes clusters on AWS, GCE, Azure, OpenStack, or bare metal. KubeSpray is based on Ansible, and is available on most Linux distributions. It is a Kubernetes Incubator project.
- With Kops, we can create, destroy, upgrade, and maintain production-grade, highly-available Kubernetes clusters from the command line. It can provision the machines as well. Currently, AWS is officially supported. Support for GCE and VMware vSphere are in alpha stage, and other platforms are planned for the future.

If the existing solutions and tools do not fit your requirements, then you can always install Kubernetes from scratch.

# Pods

A Pod is the smallest and simplest Kubernetes object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, which:

- Are scheduled together on the same host
- Share the same network namespace
- Mount the same external storage (volumes).

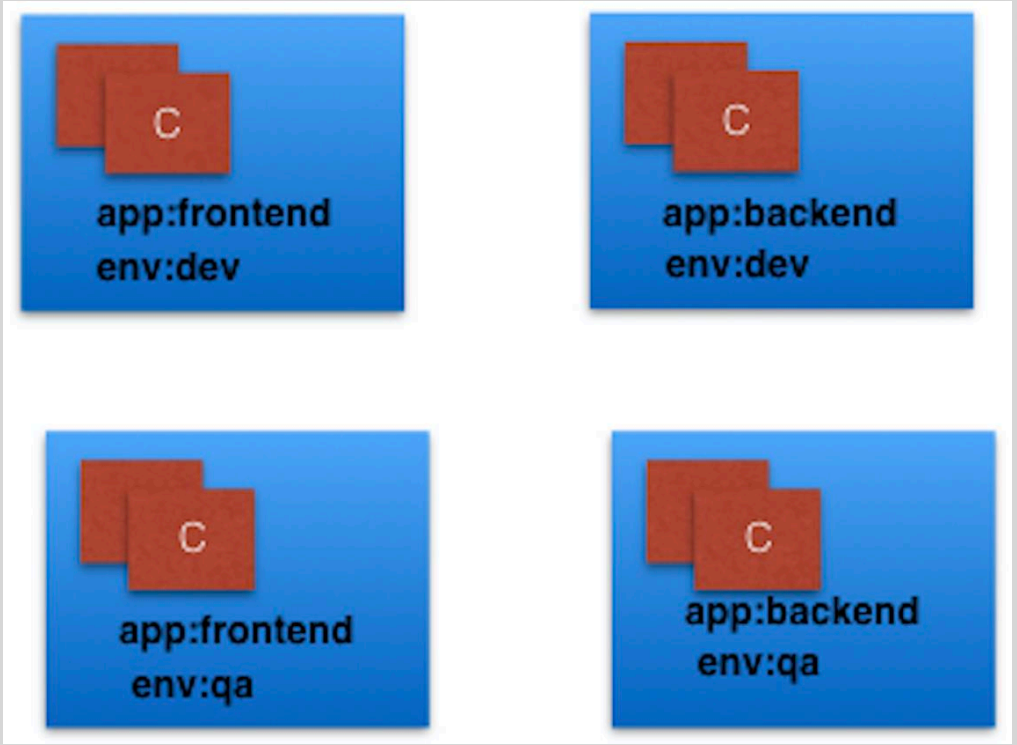





NobleProg

# Labels

- Labels are key-value pairs that can be attached to any Kubernetes objects (e.g. Pods).
- Labels are used to organize and select a subset of objects, based on the requirements in place.
- Many objects can have the same Label(s).
- Labels do not provide uniqueness to objects.



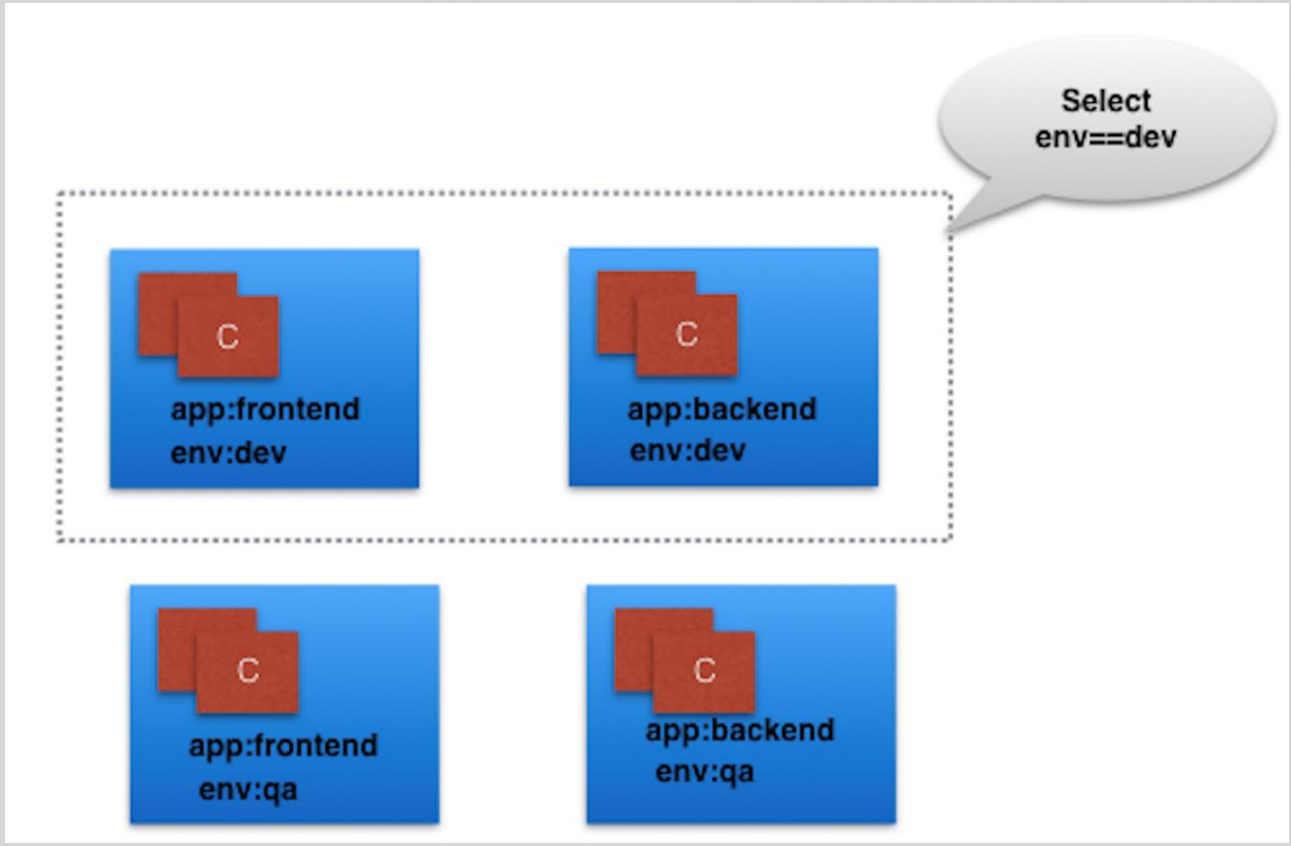



NobleProg

# Label Selectors

With Label Selectors, we can select a subset of objects. Kubernetes supports two types of Selectors:

- **Equality-Based Selectors** With this type of selectors, we can use the **=**, **==**, or **!=**operators.
- **Set-Based Selectors** With this type of Selectors, we can use the **in**, **notin**, and **exist** operators.





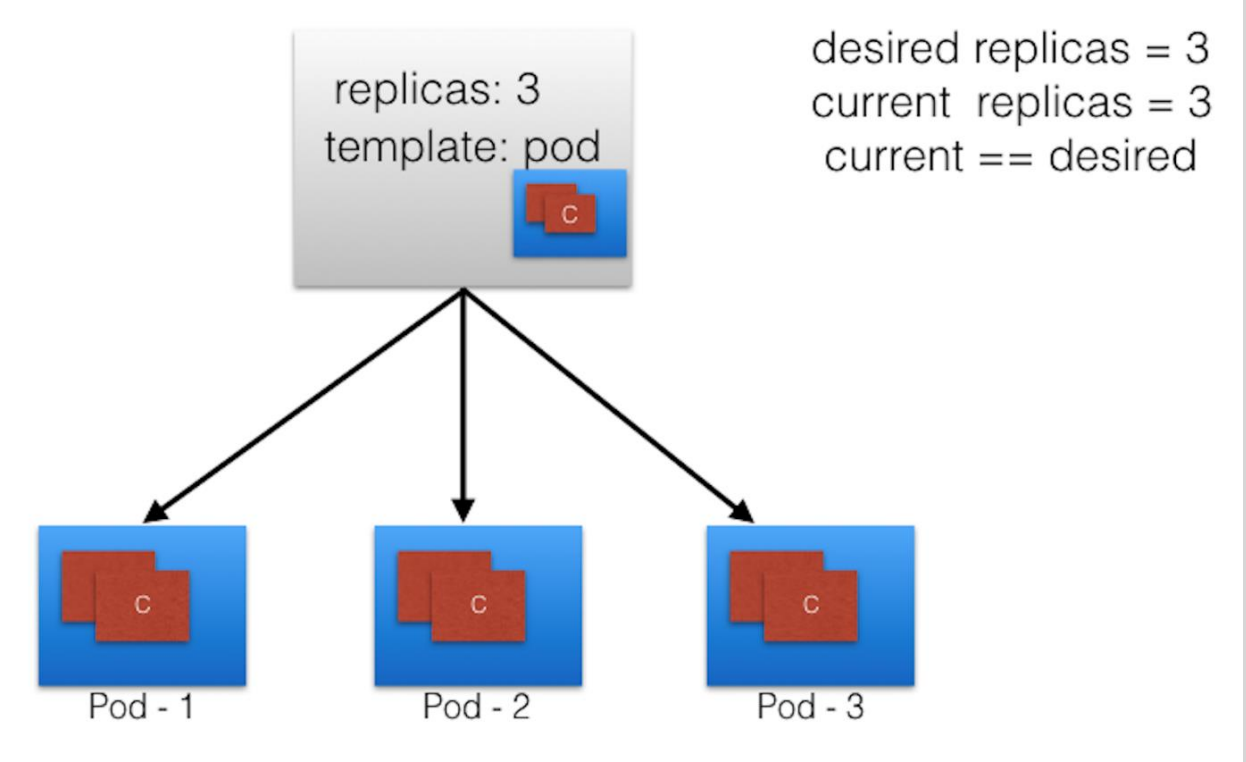

NobleProg

# ReplicationControllers

- A ReplicationController (rc) is a controller that is part of the master node's controller manager.
- It makes sure the specified number of replicas for a Pod is running at any given point in time.
- If there are more Pods than the desired count, the ReplicationController would kill the extra Pods, and, if there are less Pods, then the ReplicationController would create more Pods to match the desired count.
- Generally, we don't deploy a Pod independently, as it would not be able to re-start itself, if something goes wrong. We always use controllers like ReplicationController to create and manage Pods.

# ReplicaSets

- A ReplicaSet (rs) is the next-generation ReplicationController.
- ReplicaSets support both equality- and set-based selectors, whereas ReplicationControllers only support equality-based Selectors. Currently, this is the only difference.

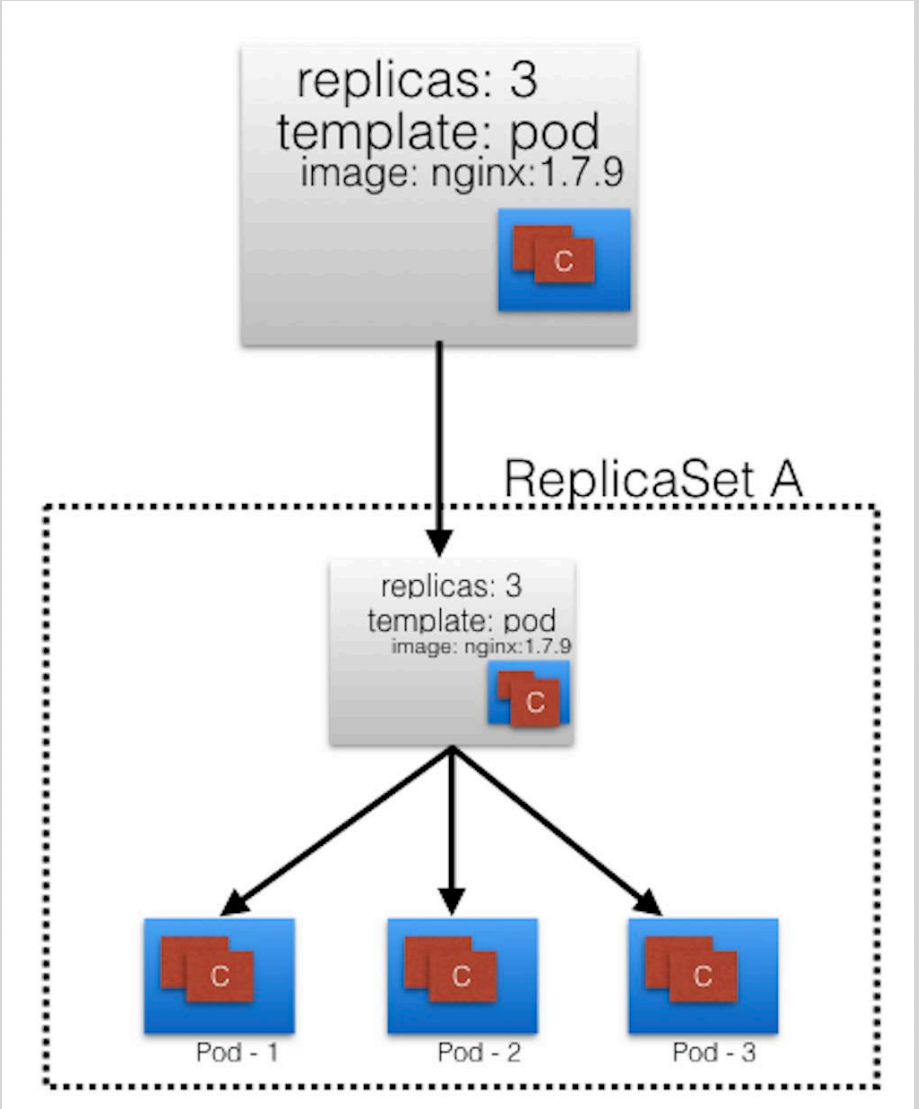





NobleProg

# Deployments

- Deployment objects provide declarative updates to Pods and ReplicaSets.
- The DeploymentController is part of the master node's controller manager, and it makes sure that the current state always matches the desired state.






NobleProg

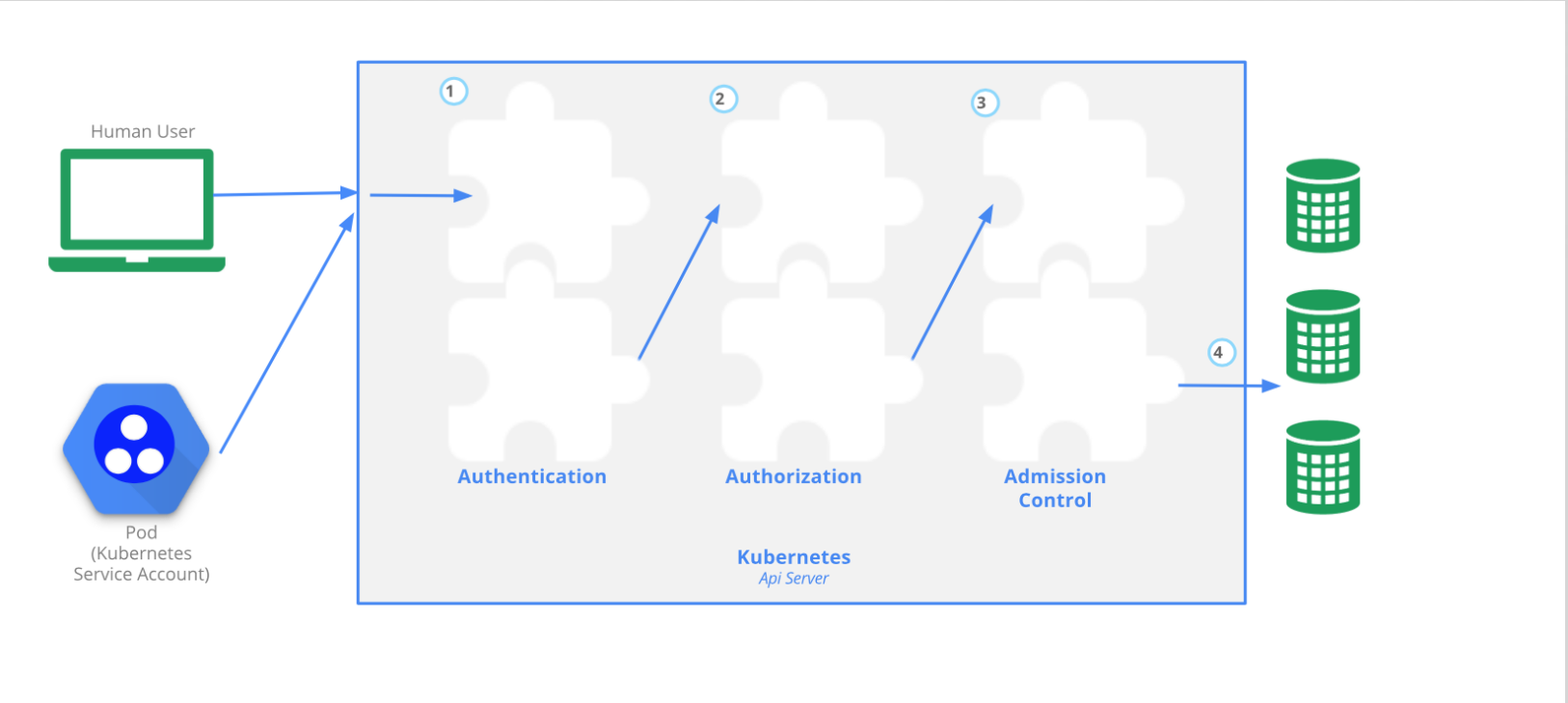
# Namespaces

- If we have numerous users whom we would like to organize into teams/projects, we can partition the Kubernetes cluster into sub-clusters using Namespaces.
- The names of the resources/objects created inside a Namespace are unique, but not across Namespaces.
- Generally, Kubernetes creates two default Namespaces: **kube-system** and **default.**
- The **kube-system** Namespace contains the objects created by the Kubernetes system.
- The **default** Namespace contains the objects which belong to any other Namespace.
- By default, we connect to the **default** Namespace.
- **kube-public** is a special Namespace, which is readable by all users and used for special purposes, like bootstrapping a cluster.
- Using Resource Quotas, we can divide the cluster resources within Namespaces.

# Authentication, Authorization, and Admission Control

To access and manage any resources/objects in the Kubernetes cluster, we need to access a specific API endpoint on the API server. Each access request goes through the following three stages:

- **Authentication** Logs in a user.
- **Authorization** Authorizes the API requests added by the logged-in user.
- **Admission Control** Software modules that can modify or reject the requests based on some additional checks, like **Quota**.






NobleProg

# Authentication

- Kubernetes does not have an object called *user*, nor does it store *usernames* or other related details in its object store.
- However, even without that, Kubernetes can use usernames for access control and request logging

Kubernetes has two kinds of users:

- **Normal Users**

  They are managed outside of the Kubernetes cluster via independent services like User/Client Certificates, a file listing usernames/passwords, Google accounts, etc.
- **Service Accounts**

  With Service Account users, in-cluster processes communicate with the API server to perform different operations. Most of the Service Account users are created automatically via the API server, but they can also be created manually. The Service Account users are tied to a given Namespace and mount the respective credentials to communicate with the API server as Secrets.

If properly configured, Kubernetes can also support **anonymous requests**, along with requests from Normal Users and Service Accounts.

# Authentication

| | |
|---|---|
| **Client Certificates** | To enable client certificate authentication, we need to reference a file containing one or more certificate authorities by passing the **--client-ca-file=SOMEFILE** option to the API server. The certificate authorities mentioned in the file would validate the client certificates presented to the API server. |
| **Static Token File** | We can pass a file containing pre-defined bearer tokens with the **--token-auth-file=SOMEFILE** option to the API server. Currently, these tokens would last indefinitely, and they cannot be changed without restarting the API server. |
| **Bootstrap Tokens** | This feature is currently in an alpha status, and is mostly used for bootstrapping a new Kubernetes cluster. |
| **Static Password File** | It is similar to *Static Token File.* We can pass a file containing basic authentication details with the **--basic-auth-file=SOMEFILE** option. These credentials would last indefinitely, and passwords cannot be changed without restarting the API server. |
| **Service Account Tokens** | This is an automatically enabled authenticator that uses signed bearer tokens to verify the requests. These tokens get attached to Pods using the ServiceAccount Admission Controller, which allows in-cluster processes to talk to the API server. |
| **OpenID Connect Tokens** | OpenID Connect helps us connect with OAuth 2 providers, such as Azure Active Directory, Salesforce, Google, etc., to offload the authentication to external services. |
| **Webhook Token Authentication** | With Webhook-based authentication, verification of bearer tokens can be offloaded to a remote service. |
| **Keystone Password** | Keystone authentication can be enabled by passing the **--experimental-keystone-url=<AuthURL>** option to the API server, where**AuthURL** is the Keystone server endpoint. |
| **Authenticating Proxy** | If we want to program additional authentication logic, we can use an authenticating proxy. |

# Authorization

- After a successful authentication, users can send the API requests to perform different operations.
- Then, those API requests get authorized by Kubernetes using various authorization modules.
- Some of the API request attributes that are reviewed by Kubernetes include user, group, extra, Resource or Namespace, to name a few.
- Next, these attributes are evaluated against policies. If the evaluation is successful, then the request will be allowed, otherwise it will get denied.
- Similar to the Authentication step, Authorization has multiple modules/authorizers. More than one module can be configured for one Kubernetes cluster, and each module is checked in sequence. If any authorizer approves or denies a request, then that decision is returned immediately.

# Authorization

- Node authorization is a special-purpose authorization mode which specifically authorizes API requests made by kubelets.
- **With Attribute-Based Access Control (ABAC) Authorizer**,
  - Kubernetes grants access to API requests, which combine policies with attributes.
  - To enable the ABAC authorizer, we would need to start the API server with the --authorization-mode=ABAC option. We would also need to specify the authorization policy, like --authorization-policy-file=PolicyFile.json.
- With the **Webhook** authorizer,
  - Kubernetes can offer authorization decisions to some third-party services, which would return true for successful authorization, and false for failure.
  - In order to enable the Webhook authorizer, we need to start the API server with the --authorization-webhook-config-file=SOME_FILENAME option, where SOME_FILENAME is the configuration of the remote authorization service.

# Authorization

- **Role-Based Access Control (RBAC) Authorizer** can regulate the access to resources based on the roles of individual users.
  - In Kubernetes, we can have different roles that can be attached to subjects like users, service accounts, etc. While creating the roles, we restrict resource access by specific operations, such as **create**, **get**, **update**, **patch**, etc. These operations are referred to as verbs.
  - **Role** With Role, we can grant access to resources within a specific Namespace.
  - **ClusterRole** The ClusterRole can be used to grant the same permissions as Role does, but its scope is cluster-wide.
  - Once the role is created, we can bind users with RoleBinding.
- **RoleBinding** It allows us to bind users to the same namespace as a Role. We could also refer a ClusterRole in RoleBinding, which would grant permissions to Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.
- **ClusterRoleBinding** It allows us to grant access to resources at a cluster-level and to all Namespaces.
- To enable the RBAC authorizer, we would need to start the API server with the --authorization-mode=RBAC option. With the RBAC authorizer, we dynamically configure policies.



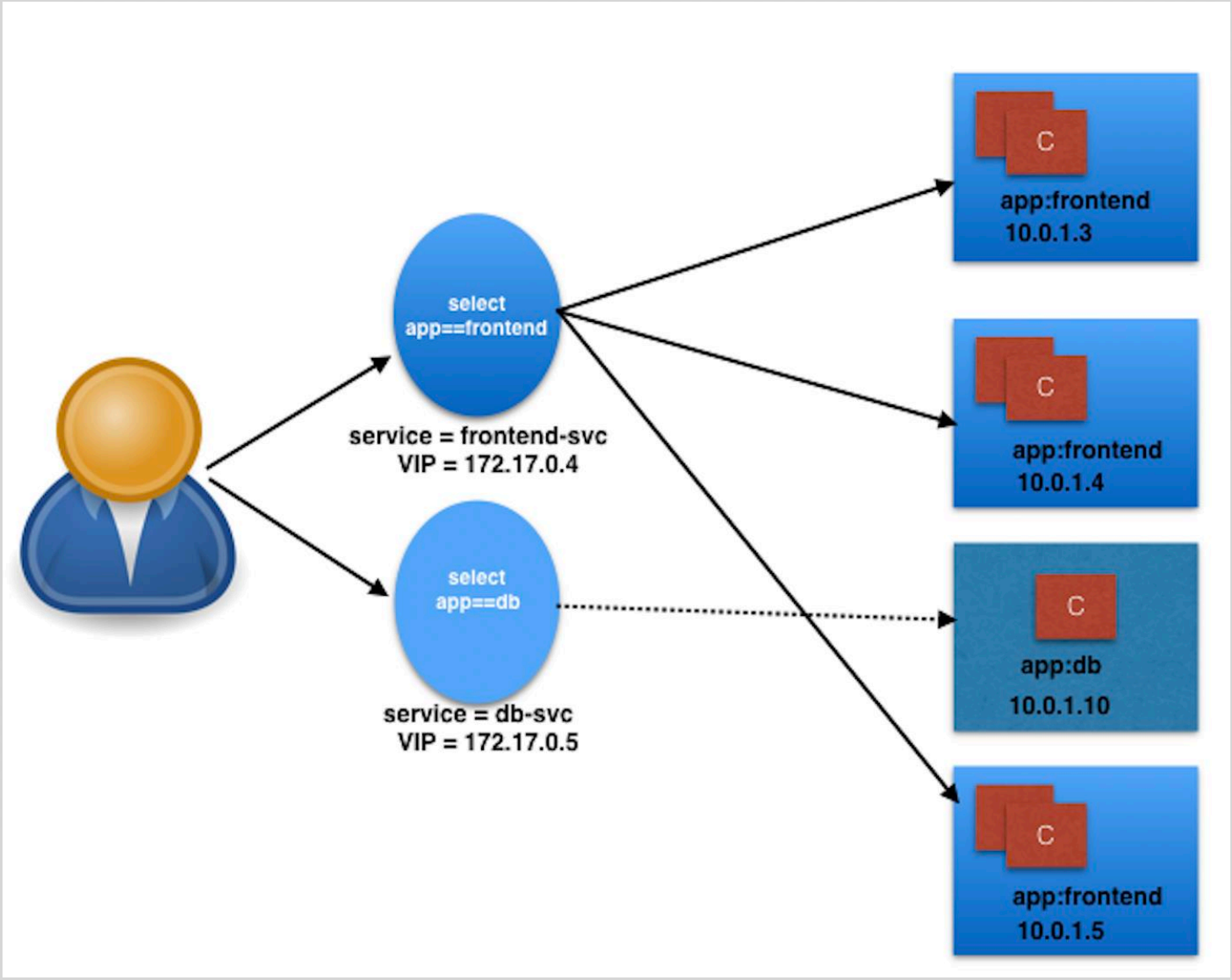
NobleProg

# Admission Control

- Admission control is used to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc.
- We force these policies using different admission controllers, like ResourceQuota, AlwaysAdmit, DefaultStorageClass, etc.
- They come into effect only after API requests are authenticated and authorized.
- To use admission controls, we must start the Kubernetes API server with the **admission-control**, which takes a comma-delimited, ordered list of controller names, like in the following example:

  **--admission-control=NamespaceLifecycle,ResourceQuota,PodSecurityPolicy,DefaultStorageClass**.
- By default, Kubernetes comes with some built-in admission controllers.

# Services

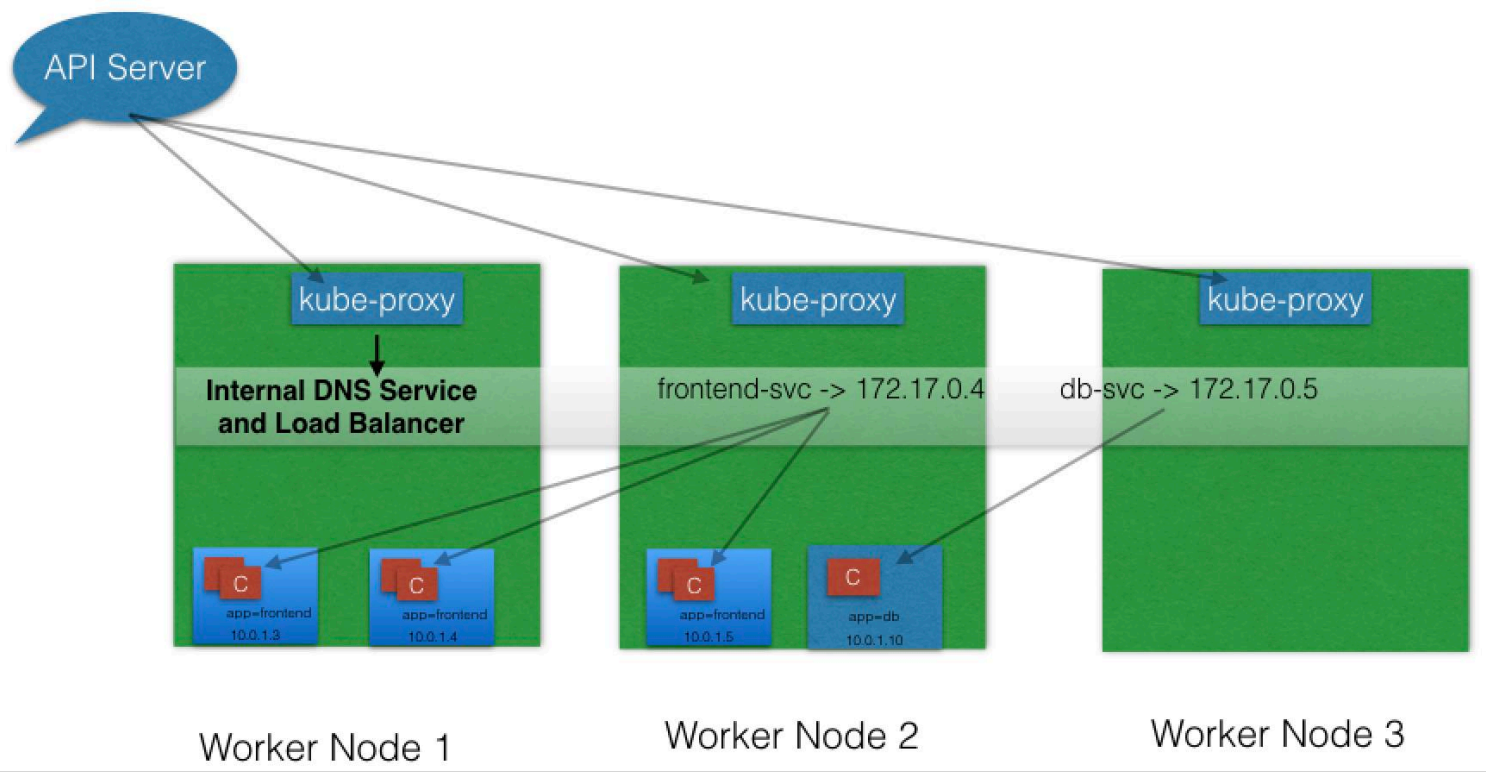





NobleProg

# kube-proxy

- All of the worker nodes run a daemon called kube-proxy, which watches the API server on the master node for the addition and removal of Services and endpoints.
- For each new Service, on each node, kube-proxy configures the iptables rules to capture the traffic for its ClusterIP and forwards it to one of the endpoints.
- When the service is removed, kube-proxy removes the iptables rules on all nodes as well.



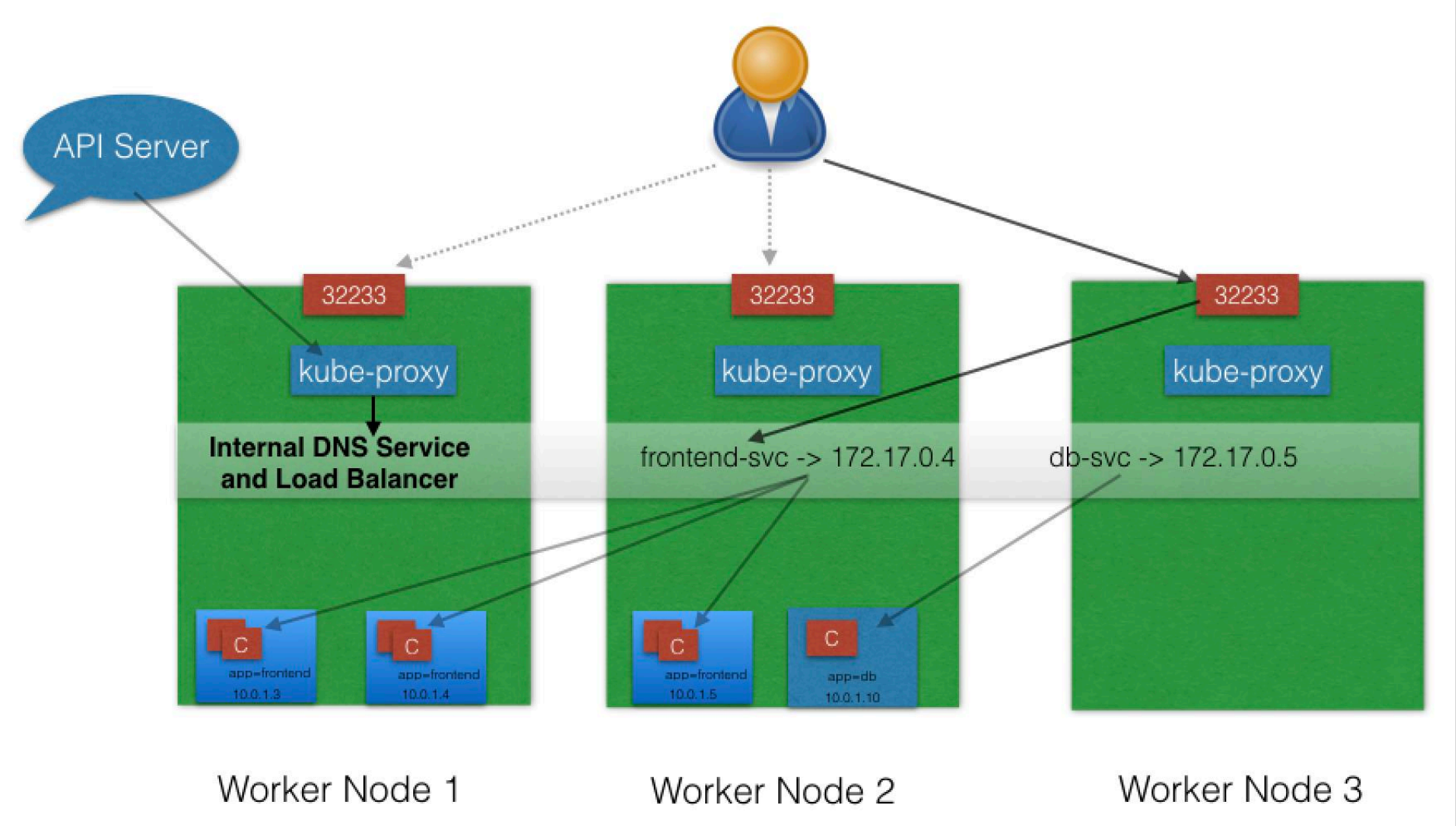



NobleProg

# ServiceType

- While defining a Service, we can also choose its access scope. We can decide whether the Service:
  - Is only accessible within the cluster
  - Is accessible from within the cluster and the external world
  - Maps to an external entity which resides outside the cluster.
- Access scope is decided by S*erviceType*, which can be mentioned when creating the Service.
- ClusterIP is the default ServiceType. A Service gets its Virtual IP address using the ClusterIP. That IP address is used for communicating with the Service and is accessible only within the cluster.
- With the NodePort ServiceType, in addition to creating a ClusterIP, a port from the range 30000-32767 is mapped to the respective Service, from all the worker nodes.

# ServiceType



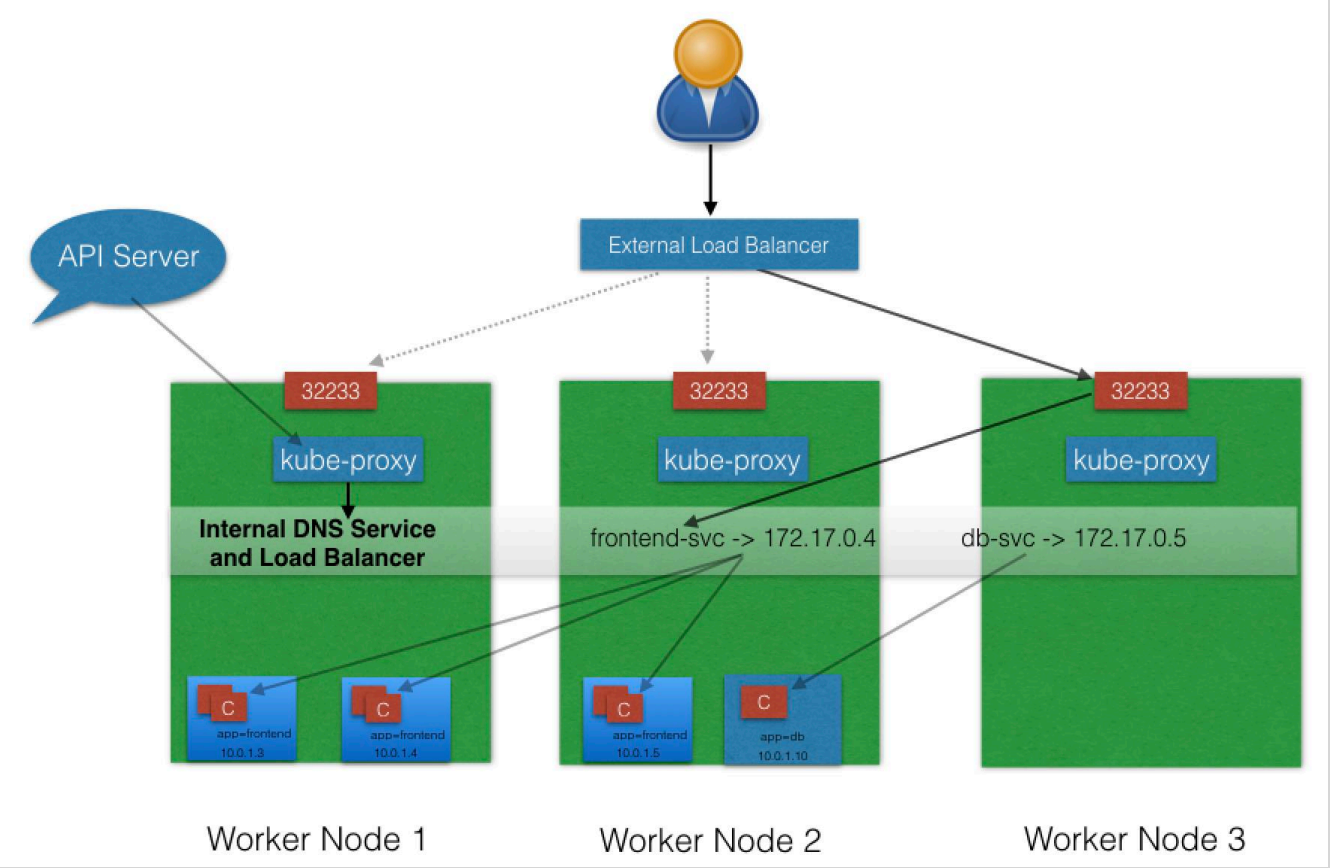



NobleProg

# ServiceType

With the **LoadBalancer** *ServiceType*:

- NodePort and ClusterIP Services are automatically created, and the external load balancer will route to them
- The Services are exposed at a static port on each worker node
- The Service is exposed externally using the underlying cloud provider's load balancer feature.

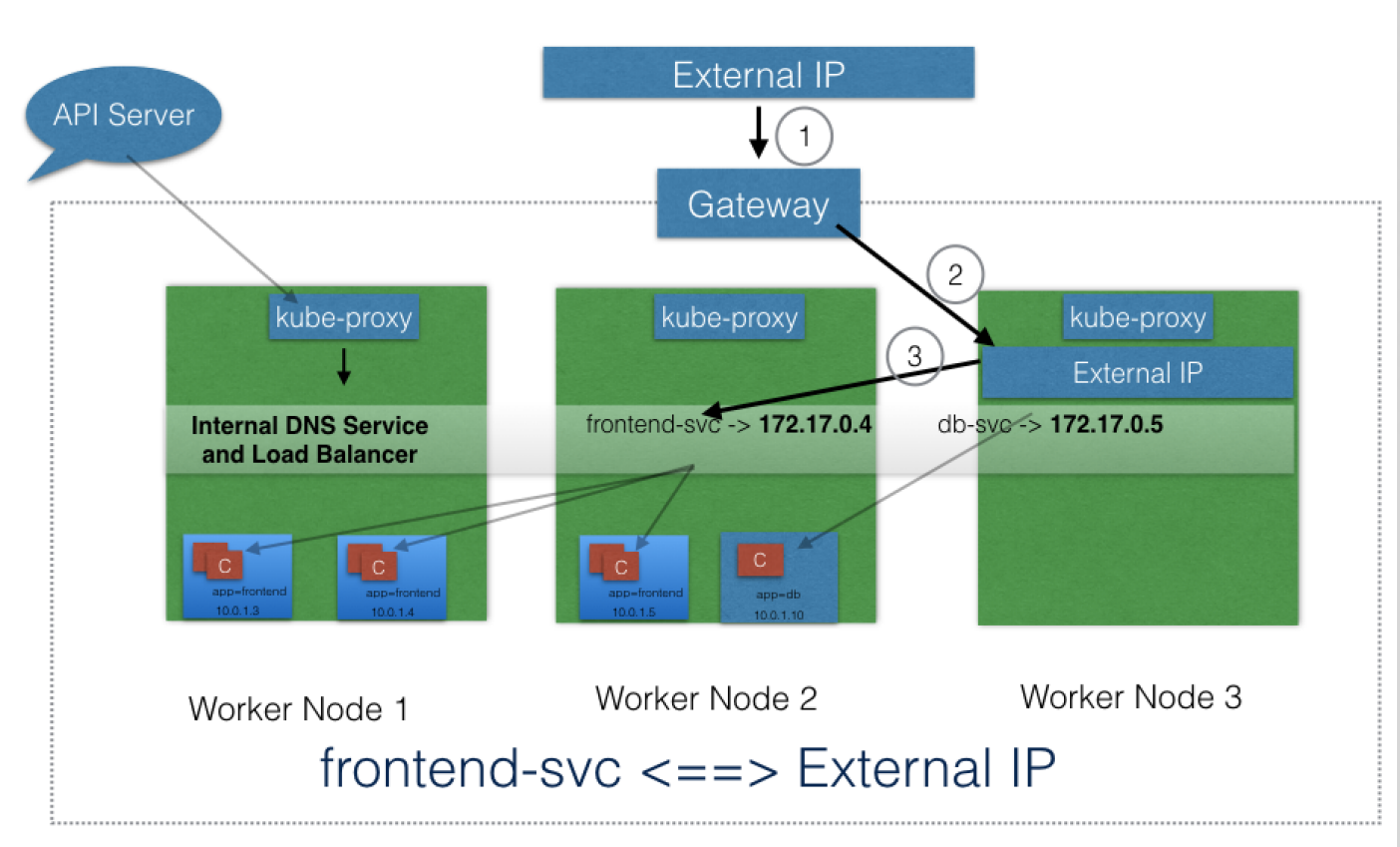





NobleProg

# ServiceType: ExternalIP

A Service can be mapped to an **ExternalIP** address if it can route to one or more of the worker nodes. Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the the Service endpoints.




Please note that ExternalIPs are not managed by Kubernetes. The cluster administrators has configured the routing to map the ExternalIP address to one of the nodes.



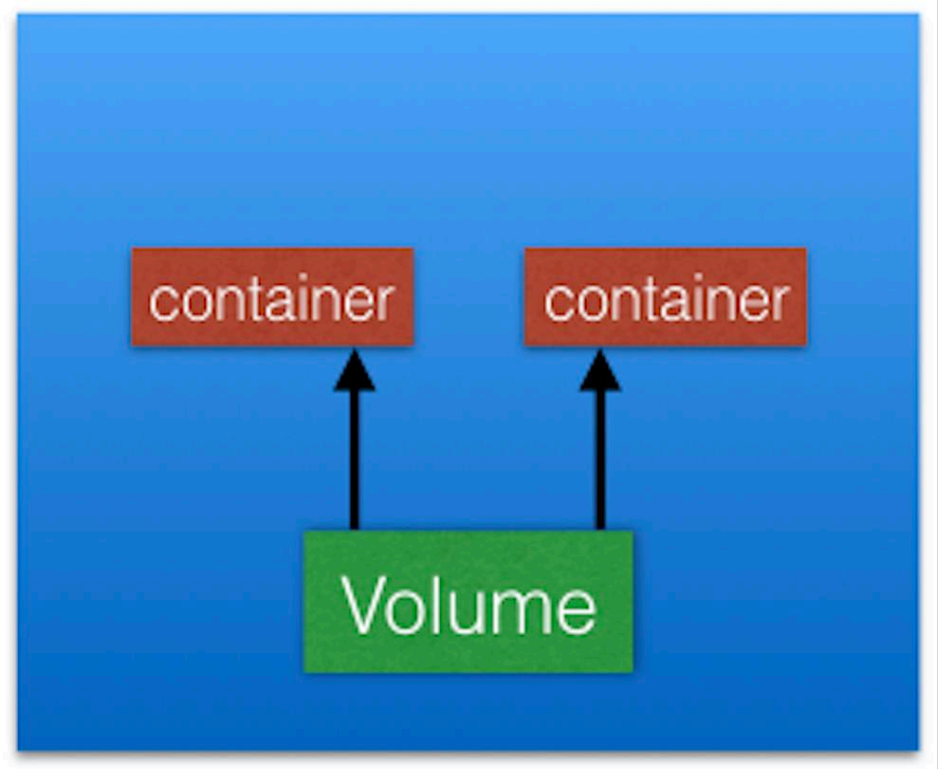
NobleProg

# ServiceType: ExternalName

- **ExternalName** is a special *ServiceType*, that has no Selectors and does not define any endpoints. When accessed within the cluster, it returns a **CNAME** record of an externally configured Service.
- The primary use case of this *ServiceType* is to make externally configured Services like **my-database.example.com** available inside the cluster, using just the name, like **my-database**, to other Services inside the same Namespace.

# Volumes

- A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.



# Volume Types

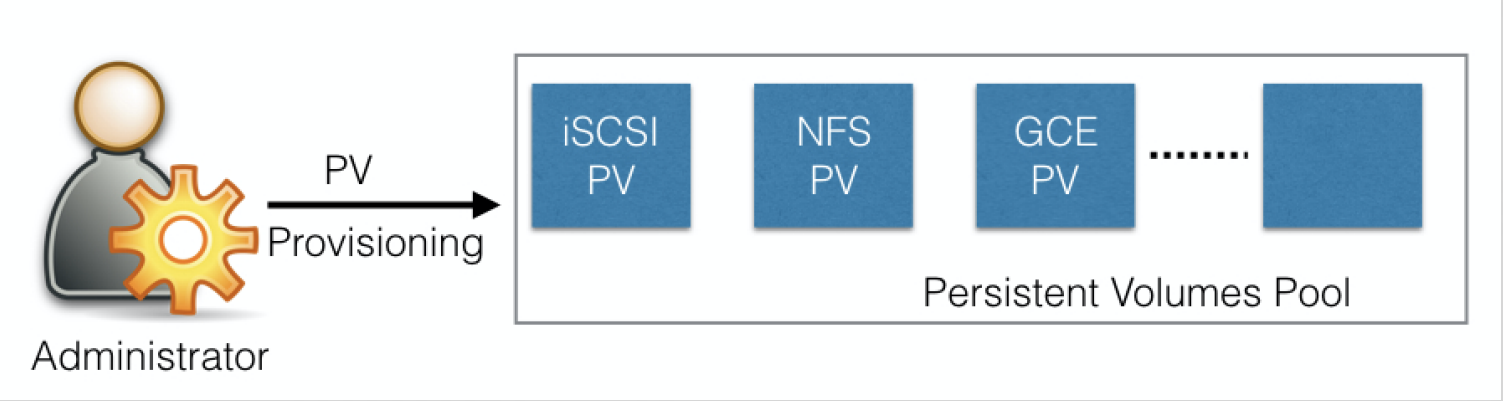
| | |
|---|---|
| **emptyDir** | An **empty** Volume is created for the Pod as soon as it is scheduled on the worker node. The Volume's life is tightly coupled with the Pod. If the Pod dies, the content of **emptyDir** is deleted forever. |
| **hostPath** | With the **hostPath** Volume Type, we can share a directory from the host to the Pod. If the Pod dies, the content of the Volume is still available on the host. |
| **gcePersistentDisk** | With the **gcePersistentDisk** Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod. |
| **awsElasticBlockStore** | With the **awsElasticBlockStore** Volume Type, we can mount an AWS EBS Volume into a Pod. |
| **nfs** | With nfs, we can mount an NFS share into a Pod. |
| **iscsi** | With iscsi, we can mount an iSCSI share into a Pod. |
| **secret** | With the **secret** Volume Type, we can pass sensitive information, such as passwords, to Pods. |
| **persistentVolumeClaim** | We can attach a PersistentVolume to a Pod using a **persistentVolumeClaim**. |



NobleProg

# PersistentVolumes

- A Persistent Volume is a network-attached storage in the cluster, which is provisioned by the administrator.
- PersistentVolumes can be dynamically provisioned based on the StorageClass resource.
- A StorageClass contains pre-defined provisioners and parameters to create a PersistentVolume.
- Using PersistentVolumeClaims, a user sends the request for dynamic PV creation, which gets wired to the StorageClass resource.
- Some of the Volume Types that support managing storage using PersistentVolumes are:
    - GCEPersistentDisk
    - AWSElasticBlockStore
    - AzureFile
    - NFS
    - iSCSI.



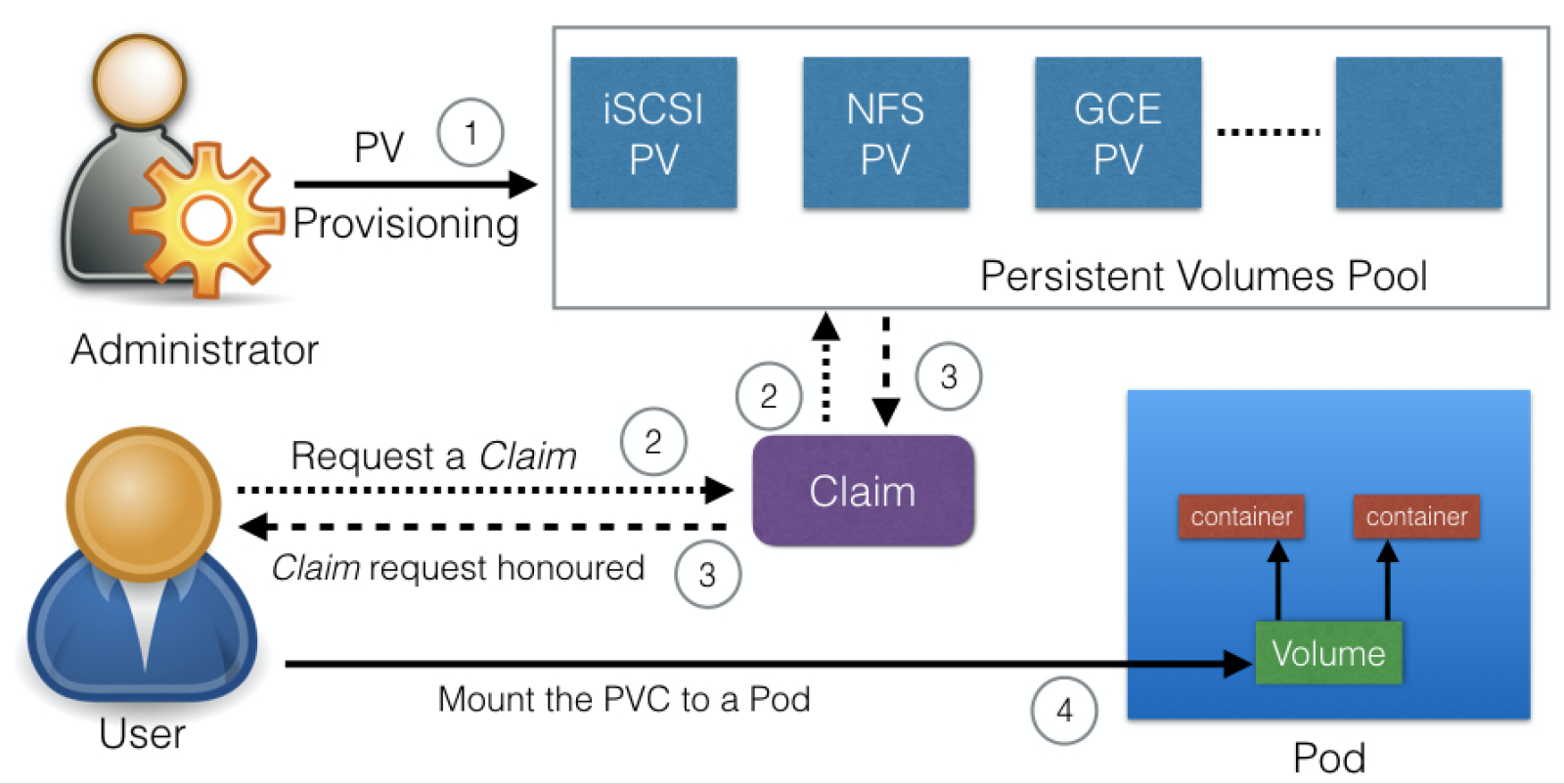

For a complete list, as well as more details, you can check out the Kubernetes documentation.



NobleProg

# PersistentVolumeClaims

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. Users request for PersistentVolume resources based on size, access modes, etc. Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.




To learn more, you can check out the [Kubernetes documentation].



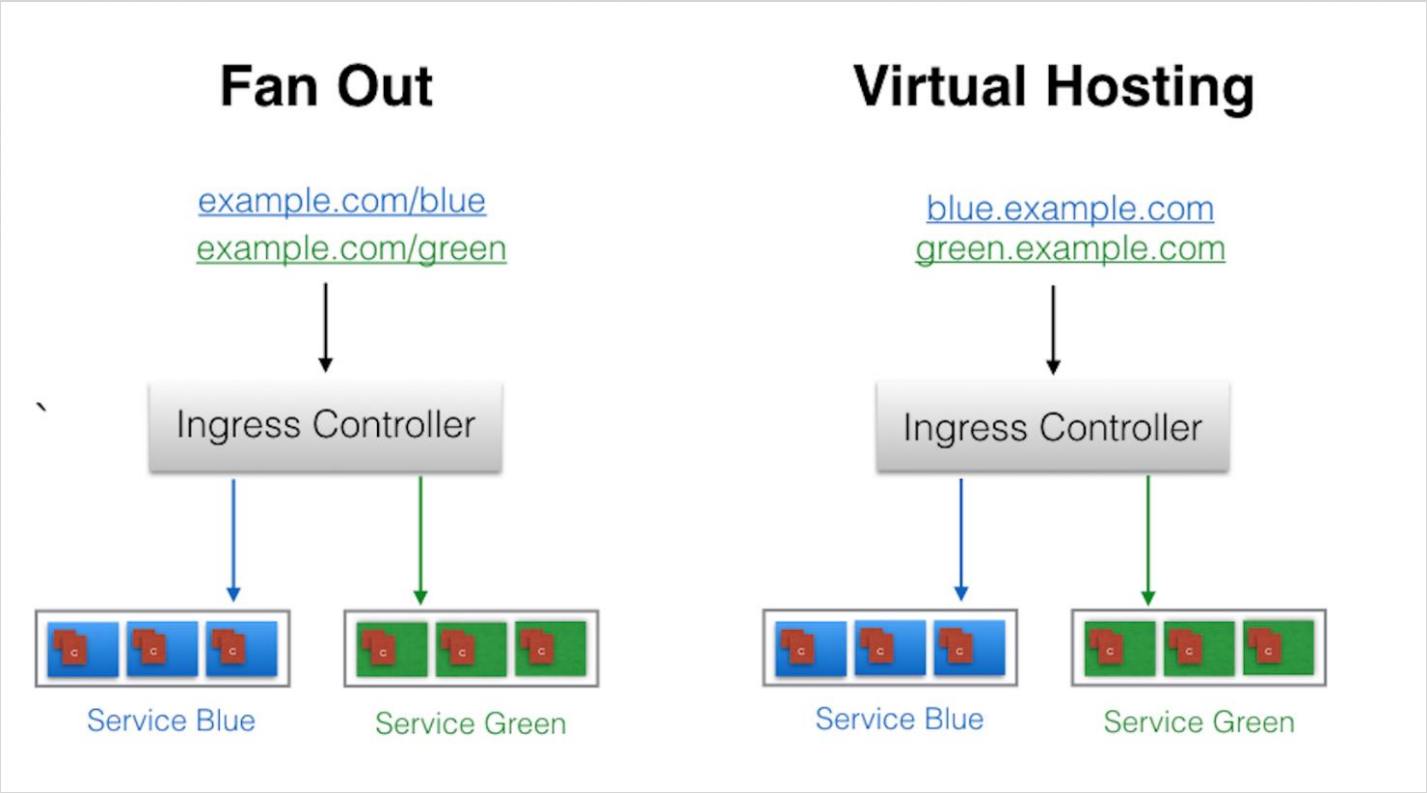
NobleProg

# Ingress

*"An Ingress is a collection of rules that allow inbound connections to reach the cluster Services."*

To allow the inbound connection to reach the cluster Services, Ingress configures a Layer 7 HTTP load balancer for Services and provides the following:

- TLS (Transport Layer Security)
- Name-based virtual hosting
- Path-based routing
- Custom rules.






NobleProg

# Annotations

- With Annotations, we can attach arbitrary non-identifying metadata to any objects, in a key-value format
- In contrast to Labels, annotations are not used to identify and select objects.
- Annotations can be used to:
  - Store build/release IDs, PR numbers, git branch, etc.
  - Phone/pager numbers of people responsible, or directory entries specifying where such information can be found
  - Pointers to logging, monitoring, analytics, audit repositories, debugging tools, etc.
  - Etc.

# Jobs

- A Job creates one or more Pods to perform a given task.
- The Job object takes the responsibility of Pod failures. It makes sure that the given task is completed successfully. Once the task is over, all the Pods are terminated automatically.
- Starting with the Kubernetes 1.4 release, we can also perform Jobs at specified times/dates, such as cron jobs.

# Quota Management

ResourceQuota object, which provides constraints that limit aggregate resource consumption per Namespace.

We can have the following types of quotas per Namespace:

- **Compute Resource Quota**

  We can limit the total sum of compute resources (CPU, memory, etc.) that can be requested in a given Namespace.
- **Storage Resource Quota**

  We can limit the total sum of storage resources (PersistentVolumeClaims, requests.storage, etc.) that can be requested.
- **Object Count Quota**

  We can restrict the number of objects of a given type (pods, ConfigMaps, PersistentVolumeClaims, ReplicationControllers, Services, Secrets, etc.).

# DaemonSets

- In some cases, like collecting monitoring data from all nodes, or running a storage daemon on all nodes, etc., we need a specific type of Pod running on all nodes at all times. A DaemonSet is the object that allows us to do just that.
- Whenever a node is added to the cluster, a Pod from a given DaemonSet is created on it.
- When the node dies, the respective Pods are garbage collected.
- If a DaemonSet is deleted, all Pods it created are deleted as well.

# StatefulSets

- The StatefulSet controller is used for applications which require a unique identity, such as name, network identifications, strict ordering, etc. For example, **MySQL cluster**, **etcd cluster**.
- The StatefulSet controller provides identity and guaranteed ordering of deployment and scaling to Pods.

# Kubernetes Federation

- With the Kubernetes Cluster Federation we can manage multiple Kubernetes clusters from a single control plane.
- We can sync resources across the clusters and have cross-cluster discovery. This allows us to do Deployments across regions and access them using a global DNS record.
- The Federation is very useful when we want to build a hybrid solution, in which we can have one cluster running inside our private datacenter and another one on the public cloud.
- We can also assign weights for each cluster in the Federation, to distribute the load as per our choice.

# Helm

- To deploy an application, we use different Kubernetes manifests, such as Deployments, Services, Volume Claims, Ingress, etc. Sometimes, it can be tiresome to deploy them one by one.
- We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata. Such a bundle is referred to as *Chart*. These Charts can then be served via repositories, such as those that we have for **rpm** and **deb** packages.
- Helm is a package manager (analogous to **yum** and **apt**) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster.
- Helm has two components:
  - A client called *helm*, which runs on your user's workstation
  - A server called *tiller*, which runs inside your Kubernetes cluster.
- The client *helm* connects to the server *tiller* to manage Charts. Charts submitted for Kubernetes are available here.

# Monitoring and Logging

- In Kubernetes, we have to collect resource usage data by Pods, Services, nodes, etc., to understand the overall resource consumption and to make decisions for scaling a given application.
- Two popular Kubernetes monitoring solutions are Heapster and Prometheus.
  - Heapster is a cluster-wide aggregator of monitoring and event data, which is natively supported on Kubernetes.
  - Prometheus, now part of CNCF (Cloud Native Computing Foundation), can also be used to scrape the resource usage from different Kubernetes components and objects. Using its client libraries, we can also instrument the code of our application.
- Another important aspect for troubleshooting and debugging is Logging, in which we collect the logs from different components of a given system. In Kubernetes, we can collect logs from different cluster components, objects, nodes, etc.
- The most common way to collect the logs is using Elasticsearch, which uses fluentd with custom configuration as an agent on the nodes. **fluentd** is an open source data collector, which is also part of CNCF.

# Happy Learning. Thank You!



NobleProg