O&B ACADEMY
Engineering for the Real World

DEVOPS-101

DOCKER AND KUBERNETES FUNDAMENTALS

Agenda (I/3)

- Container basics
- Dockerfile and building images
- Container networking
- Container storage
- Container registry

Agenda (2/3)

- YAML (brief review)
- Docker Compose
- Kubernetes basics
- Managing applications

Agenda (3/3)

- Kubernetes networking
- Stateful applications
- Application workloads and security

Docker Basics

Docker Basics

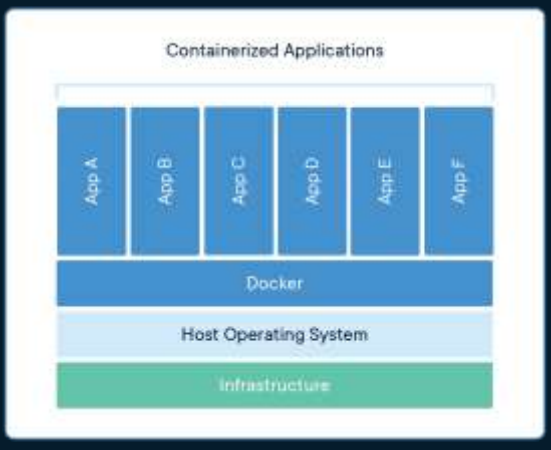
- Containerization vs Virtualization
- Installing Docker
- Docker Architecture (daemon, client, registries, images, containers)
- Getting started
  - Running containers
  - Connecting to running containers

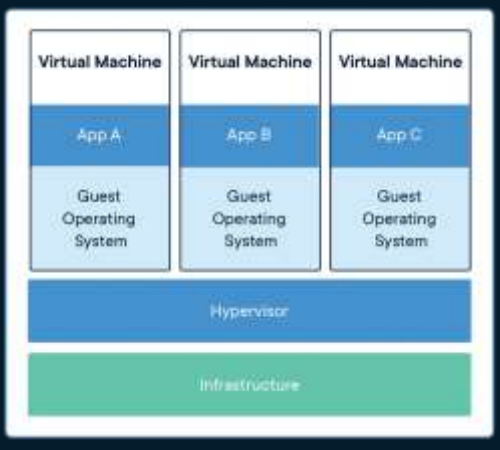
What is a Container?

A (Docker) Container wraps a piece of software in a separate filesystem that contains everything needed to run: code, things such as runtime environment, system tools, and libraries, etc, anything that is needed for the app can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

Containerization vs Virtualization

DOCKER BASICS





Virtual Machines

DOCKER BASICS

- **High Resource Consumption** - Virtual machines emulate the whole computer architecture including the BIOS. This takes up a lot of resources and must be done separately for each application.
- **Decreased Performance** - Since emulation takes up a lot of resources, each operation costs more to do.
- **Large image size** - A virtual machine is shipped with a full-blown operating system, implying that the final artifact contains a large amount of data. A small OS image can easily eat up 700mb of data.

Containers

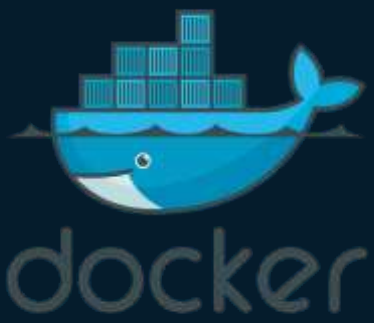
DOCKER BASICS

- Containers are more portable and efficient
- At the core of the technology, containers are just an operating system process that is run by the OS but with restrictions on what files and other resources they can consume and have access to such as CPU and networking
- In other words: a container *thinks* it's a virtual machine but in reality it's nothing more than a filesystem that shares the OS kernel.

Installing Docker

DOCKER BASICS

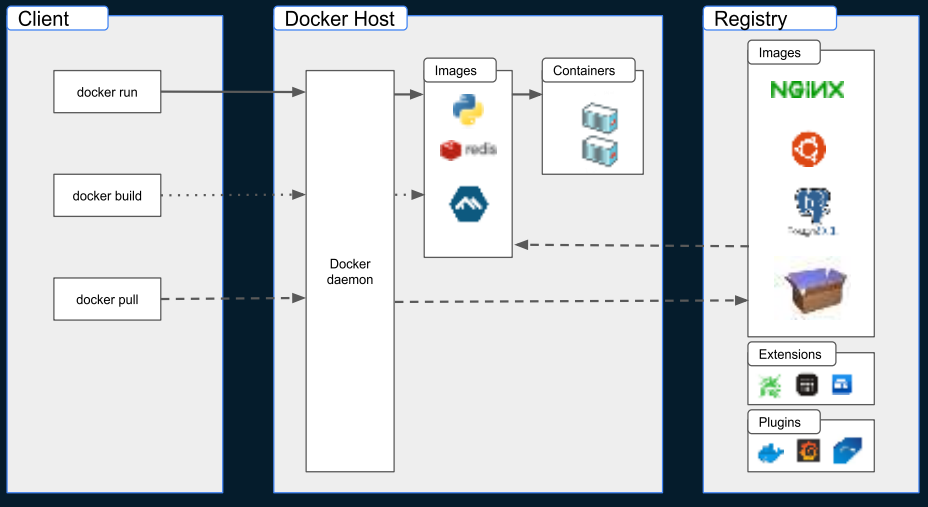
- For Windows and macOS, use Docker Desktop
- For Linux, use Docker Engine or Docker Desktop



11

Docker Architecture

DOCKER BASICS



12

Let's do some lab exercises

DOCKER BASICS

You'll get to:

- Run containers from an image
- Connect to running containers

Dockerfile and Building Images

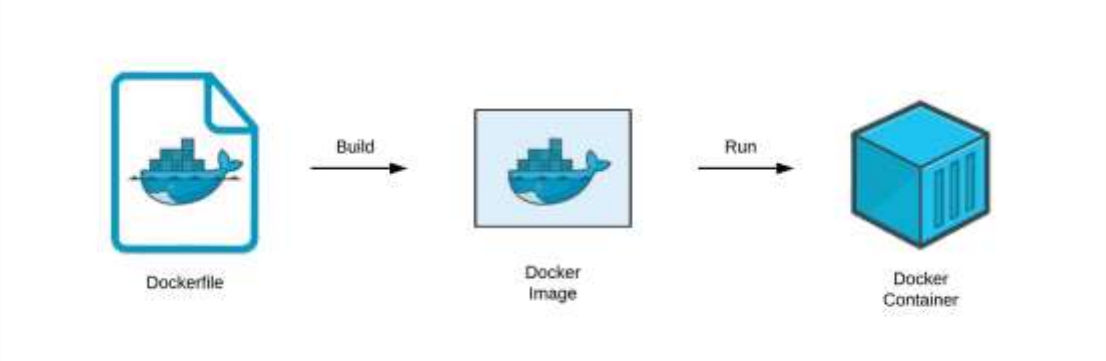
Dockerfile and Building Images

- Containers, images, and layers
- Dockerfile instructions
- Environment variables

15

A container is an instance of an image

CONTAINERS, IMAGES, AND LAYERS



16

An image is read-only

CONTAINERS, IMAGES, AND LAYERS

Dockerfile

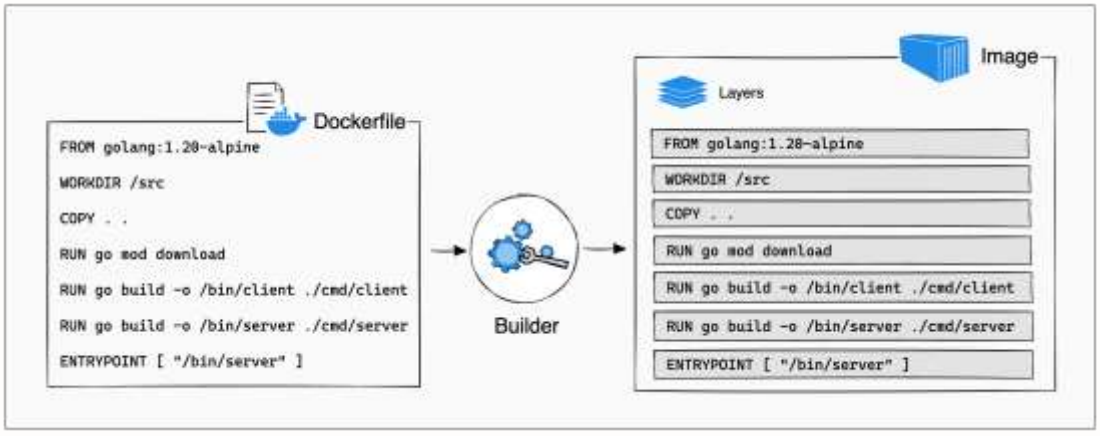
- Format

```
# Comment
INSTRUCTION arguments
```

- Example

```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT [ "/bin/server" ]
```

Dockerfile to Build an Image



Dockerfile Instructions

DOCKERFILE AND BUILDING IMAGES

Command	Purpose
`FROM`	To specify the parent image.
`WORKDIR`	To set the working directory for any commands that follow in the Dockerfile.
`RUN`	Run commands to install applications and packages required for your container.
`COPY`	To copy files or directories from a specific location.
`ADD`	Like `COPY`, but also able to handle remote URLs and unpack compressed files.

Dockerfile Instructions

DOCKERFILE AND BUILDING IMAGES

Command	Purpose
`ENTRYPOINT`	Command that will always be executed when the container starts. If not specified, the default is `/bin/sh -c`.
`CMD`	Provides arguments to `ENTRYPOINT`. If `ENTRYPOINT` is not set, the `CMD` will be the commands the container executes.
`LABEL`	To add metadata to an image

Dockerfile Sample

DOCKERFILE AND BUILDING IMAGES

```
# base image
FROM node:latest
# setting the work direcotry
WORKDIR /usr/src/app
# copy <src>... <dest>
COPY ./package.json .
# install dependencies
RUN npm install
# copy index.js
COPY ./index.js .

ENTRYPOINT ["node"]
CMD ["index.js"]
```

Dockerfile Instructions

DOCKERFILE AND BUILDING IMAGES

Command	Purpose
`EXPOSE`	To define which port through which to access your container application.
`ENV`	To set an environment variable

- Set environment variables using:
  - `-e VARNAME1=value1`
  - Or `--env`
- Publish container ports to host ports:
  - `-p <host-port>:<container-port>`
  - Or `--publish`

Let's do some lab exercises

DOCKER BASICS

You'll get to:

- Use an existing Dockerfile to build an image
- Write your own Dockerfile(s) and build images from it

Container Networking

Container Networking

- Networking allows Docker containers to talk with one another.
- This functionality also allows Docker to connect to non-docker workloads without even needing awareness on where they are deployed.

Types of Networks

CONTAINER NETWORKING

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- bridge
- host
- none
- overlay
- macvlan

Some Network Commands

CONTAINER NETWORKING

- To create a network

```
docker network create <network-name>
```

- To show a list of networks

```
docker network ls
```

- To delete a network

```
docker network rm <network-name>
```

- To delete all unused networks on a host

```
docker network prune
```

Service Discovery

CONTAINER NETWORKING

- Docker has an embedded DNS server. All user-defined (non-default) bridge networks supports DNS resolutions by container names.
- Containers can talk to each other without even knowing the IP address. Docker's embedded DNS server will do all the work.

Container Networking

- To document the port(s) used, use the `EXPOSE` instruction in the Dockerfile
- When running containers, publish container ports to host ports using:
  - `-p <host-port>:<container-port>`
  - Or `--publish`

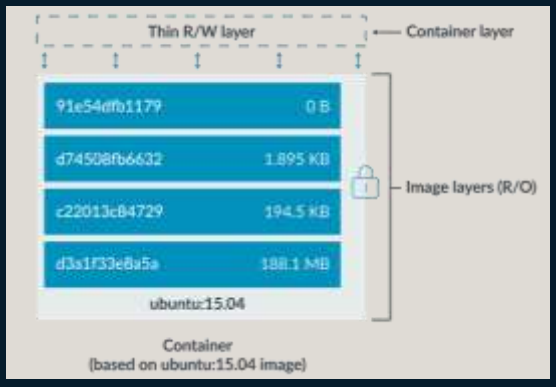
Let's do some lab exercises

CONTAINER NETWORKING

- You'll get to:
  - Use the default bridge network
  - Use a user-defined (non-default) bridge network
  - Run a database client container that connects to a database server container

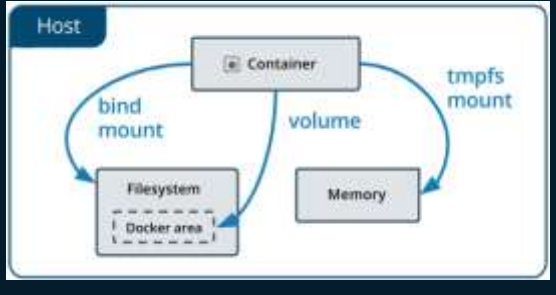
Container Storage

Container Storage



By default all files created inside a container are stored on a writable container layer.

Container Storage

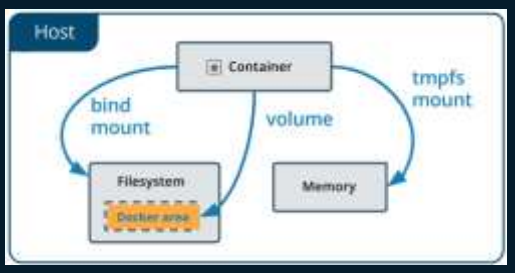


Docker has two options for containers to store files on the host machine, so that the files are persisted even after the container stops:

- volumes,
- and bind mounts.

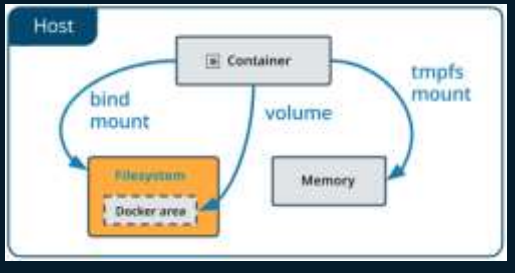
Volumes

Volumes are stored in a part of the host filesystem which is *managed by Docker*. Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.



Bind Mounts

Bind mounts may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.



Good Use Cases for Volumes

CONTAINER STORAGE

- Sharing data among multiple running containers.
- When the Docker host is not guaranteed to have a given directory or file structure.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally.
- When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice.

Good Use Cases for Bind Mounts

CONTAINER STORAGE

- Sharing configuration files from the host machine to containers.
- Sharing source code or build artifacts between a development environment on the Docker host and a container.
  - If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.
- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

Some Volume Commands

CONTAINER STORAGE

- To create a volume

```
docker volume create <volume-name>
```

- To show a list of volumes

```
docker volume ls
```

- To delete a volume

```
docker volume rm <volume-name>
```

- To delete all unused volumes on a host

```
docker volume prune
```

Let's do some lab exercises

CONTAINER NETWORKING

You'll get to:

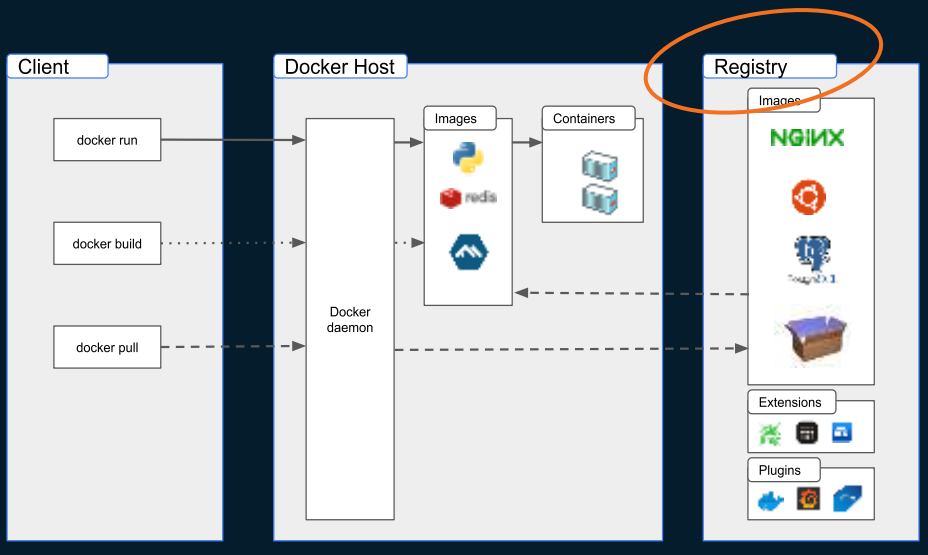
- Try out bind mounts
- Try out volumes (or volume mounts)
- Run containers that serves HTML pages from the host machine

Container Registry

41

Docker Architecture

CONTAINER REGISTRY



42

Some Commands

CONTAINER REGISTRY

- Upload an image to a registry

```
docker push [OPTIONS] NAME[:TAG]
```

- Download an image from a registry

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

- Create a tag `TARGET_IMAGE` that refers to `SOURCE_IMAGE`

```
docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

- Log in to / log out from a registry

```
docker login [OPTIONS] [SERVER]    docker logout [SERVER]
```

Image Name

CONTAINER REGISTRY

A full image name has the following format and components:

```
[HOST[:PORT]/]PATH
```

- `HOST` (optional) registry hostname specifies where the image is located. Defaults to `registry-1.docker.io`.
  - `PORT` If hostname is present, it may optionally be followed by a port number
- `PATH` consists of slash-separated components. Each component may contain lowercase letters, digits and separators. A separator is defined as a period, one or two underscores, or one or more hyphens.

Image Name

CONTAINER REGISTRY

A full image name has the following format and components:

```
[HOST[:PORT]/]PATH
```

- `HOST` (optional)
- `PATH`

  For Docker's public registry, the `PATH` format is as follows:

  ```
  [NAMESPACE/]REPOSITORY
  ```

  The first, optional component is typically a user's or an organization's namespace. The second, mandatory component is the repository name. When the namespace is not present, Docker uses `library` as the default namespace.

Image Name Samples

CONTAINER REGISTRY

- Amazon ECR
  - *aws_account_id*.dkr.ecr.*region*.amazonaws.com/image-name:tag
- Google container registry
  - gcr.io/*project-id*/image-name:tag
- Microsoft Azure container registry
  - *registry-name*.azurecr.io/image-name:tag

Let's do some lab exercises

CONTAINER REGISTRY

You'll get to:

- Pull an image from a different container registry
- Push an image to your Docker Hub account
- Push an image to a locally running registry

YAML

BRIEF REVIEW

YAML

- Is a data serialization language designed to be directly writable and readable by humans
- Does not allow the use of tabs
- Must have space between the element parts
- Is CASE sensitive
- End your file with the `.yaml` or `.yml` extension
- Is a superset of JSON
- Is used by Docker Compose and Kubernetes

Scalar Types and Comments

YAML

```yaml
# Comment example
key: value

n1: 1          # integer
n2: 1.234      # float

s1: 'abc'      # string
s2: "abc"      # string
s3: abc        # string

b: false       # boolean type

d: 2015-04-05 # date type
```

```json
{
  "key": "value",

  "n1": 1,
  "n2": 1.234,

  "s1": "abc",
  "s2": "abc",
  "s3": "abc",

  "b": false,

  "d": "2015-04-05"
}
```

Collections

YAML

- Block collections *use indentation for scope* and begin each entry on its own line.
- Mappings use a colon and space (": ") to mark each key/value pair.
- Block sequences indicate each entry with a dash and space ("- ").

Mappings

YAML COLLECTIONS

```yaml
my hash:
  subkey:
    subsubkey1: 5
    subsubkey2: 6
  another:
    somethingelse: 'Hi!'
```

```json
{
  "my hash": {
    "subkey": {
      "subsubkey1": 5,
      "subsubkey2": 6
    },
    "another": {
      "somethingelse": "Hi!"
    }
  }
}
```

Sequences

YAML COLLECTIONS

```yaml
- John
- Paul
- George
- Ringo
-
  - [1, 2, 3]
  - [4, 5, 6]
  -
    - 7
    - 8
    - 9
    - 0
```

```json
[
  "John",
  "Paul",
  "George",
  "Ringo",
  [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9, 0]
  ]
]
```

Some More YAML

```yaml
description: |
  hello
  world
---
description: >
  hello
  world
---
web:
  ports:
    - "8000:5000"
redis:
  image: "redis:alpine"
```

```json
{ "description": "hello\nworld\n" }


{ "description": "hello world\n" }


{
  "web": {
    "ports": [ "8000:5000" ]
  },
  "redis": {
    "image": "redis:alpine"
  }
}
```

YAML

For more information on YAML, go to
The Official YAML Web Site
https://yaml.org/

Docker Compose

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications.
- With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.
- `docker compose` (V2) vs `docker-compose` (V1)
  - Unlike Compose V1, Compose V2 integrates into the Docker CLI platform and the recommended command-line syntax is `docker compose`.

Compose File

DOCKER COMPOSE

A YAML file defining the following top-level keys:

- `version` (deprecated)
- `services` (required)
- `networks`
- `volumes`
- `configs`
- `secrets`

Illustrative Example

DOCKER COMPOSE FILE

```yaml
services:
  backend:
    # ...
  db:
    # ...
  frontend:
    # ...
volumes:
  db-data: {}
secrets:
  db-password:
    file: db/password.txt
networks:
  react-spring: {}
  spring-mysql: {}
```

```yaml
services:
  backend:
    secrets:
      - db-password
    networks:
      - react-spring
      - spring-mysql
    depends_on:
      db:
        condition: service_healthy
  db:
    image: mariadb:10.6.4-focal
    secrets:
      - db-password
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - spring-mysql
```

Services

TOP-LEVEL KEY IN DOCKER COMPOSE FILE

```yaml
services:
  frontend:
    image: awesome/webapp
    build: ./webapp
  backend:
    image: awesome/backend
    build:
      context: ./backend
      dockerfile: ../backend.Dockerfile
  db:
    # Use mariadb image for both amd64 & arm64 architecture
    image: mariadb:10.6.4-focal
    # If you really want to use MySQL
    #image: mysql:8.0.33
```

Services

TOP-LEVEL KEY IN DOCKER COMPOSE FILE

```yaml
services:
  postgres:
    image: postgres:latest
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PW}
    ports:
      - "5432:5432"
  pgadmin:
    image: dpage/pgadmin4:latest
    environment:
      - PGADMIN_DEFAULT_EMAIL=${PGADMIN_MAIL}
      - PGADMIN_DEFAULT_PASSWORD=${PGADMIN_PW}
    ports:
      - "5050:80"
```

Demo and Follow-Along

DOCKER COMPOSE

Others

TOP-LEVEL KEY IN DOCKER COMPOSE FILE

- `networks`
  - See https://docs.docker.com/compose/compose-file/06-networks/
- `volumes`
  - See https://docs.docker.com/compose/compose-file/07-volumes/
- `configs`
- `secrets`

Some Commands

DOCKER COMPOSE

- To create and start containers (expects `compose.yml`)

```
docker compose up
```

- Use `-f` option to use a different configuration file

```
docker compose -f some-other.yml up
```

- Use -d option to run containers in the background

```
docker compose up -d
```

- To get help

```
docker compose --help
```

```
docker compose [COMMAND] --help
```

Some Commands

DOCKER COMPOSE

- To stop and remove containers

```
docker compose down
```

- To stop services

```
docker compose stop
```

- To restart service containers

```
docker compose restart
```

- To remove stopped service containers

```
docker compose rm
```


Let's do some lab exercises

CONTAINER REGISTRY

You'll get to:

- Incrementally build a Compose (YAML) file
- Run an application composed from a set of containers

Kubernetes Basics

67

What is Kubernetes?

- An open-source system for automating deployment, scaling, and management of containerized applications.
  - Also known as K8s
- Originally designed at Google, initial release in 2014
- Maintained by Cloud Native Computing Foundation (CNCF) in 2016

68

Why Kubernetes?

WHY CONTAINERS NEED ORCHESTRATION?

- Kubernetes provides you with a framework to run containers (distributed systems) resiliently.
- Takes care of scaling and failover for your application
- Provides deployment patterns, and more.

Let's do some lab exercises

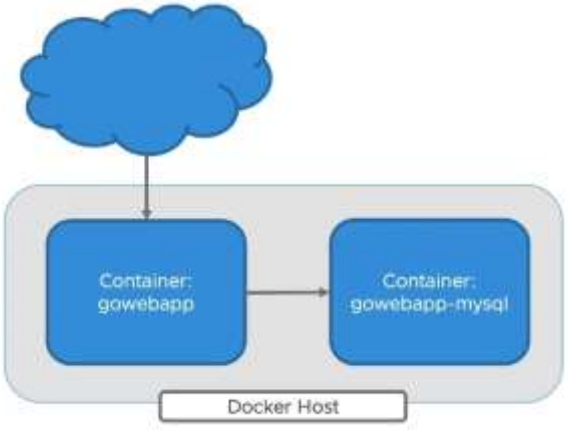
CONTAINERIZE APPLICATION

You'll get to:

- Use Dockerfile to build images for the application
- Test the containerized application
- Images will be used in subsequent lab exercises

Let’s do some lab exercises

CONTAINERIZE APPLICATION



Basic Kubernetes Objects

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

- Pod
- Service
- Deployment

Pods

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

- *Pods* are the smallest deployable units of computing that you can create and manage in Kubernetes.
- A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
- A Pod's contents are always co-located and co-scheduled, and run in a shared context.

Pods

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  containers:
  - name: gowebapp
    image: gowebapp:v1
    env:
    - name: DB_PASSWORD
      value: mypassword
    ports:
    - containerPort: 8080
```

Services

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

- A *Service* is a method for exposing a network application that is running as one or more *Pods* in your cluster.
  - Load balancing for *Pods*
- Uses selectors to determine the target *Pods*

Services

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

```yaml
apiVersion: v1
kind: Service
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  ports:
  - port: 8080
  selector:
    app: gowebapp
    tier: frontend
```

Deployments

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

- What if I need 10 `nginx` *Pods*?
  - Do I create 10 YAML files with unique *Pod* names? 😢
- Single object that will create other resources (specifically, *ReplicaSets*, which in turn, create *Pods*)

Deployments

BASIC KUBERNETES OBJECTS FOR RUNNING WORKLOADS

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gowebapp
  labels:
    app: gowebapp
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: gowebapp
      tier: frontend
  template: # continued
```

```yaml
  template:
    metadata:
      labels:
        app: gowebapp
        tier: frontend
    spec:
      containers:
      - name: gowebapp
        image: gowebapp:v1
        env:
        - name: DB_PASSWORD
          value: mypassword
        ports:
        - containerPort: 8080
```

Creating Objects

KUBECTL

- Declarative approach (preferred)

```
kubectl apply -f [<file>|<directory>|<url>]
```

- Imperative approach

```
kubectl create

kubectl edit

kubectl delete

kubectl patch
```


Let's do some lab exercises

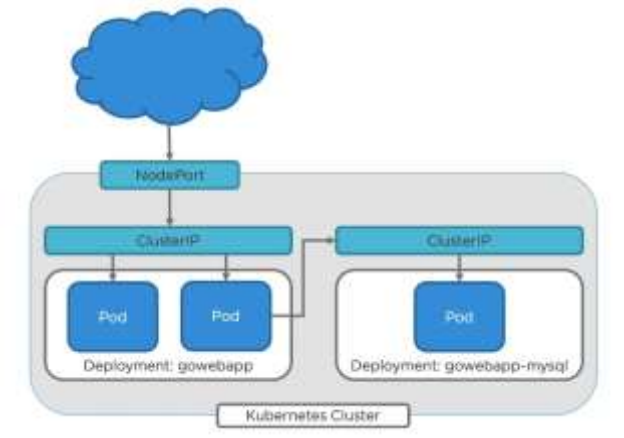
DEPLOY APPLICATION USING KUBERNETES

You'll get to:

- Create YAML files that define the objects that make up the application
- Deploy the application using Kubernetes

Let's do some lab exercises

DEPLOY APPLICATION USING KUBERNETES



Kubernetes Architecture

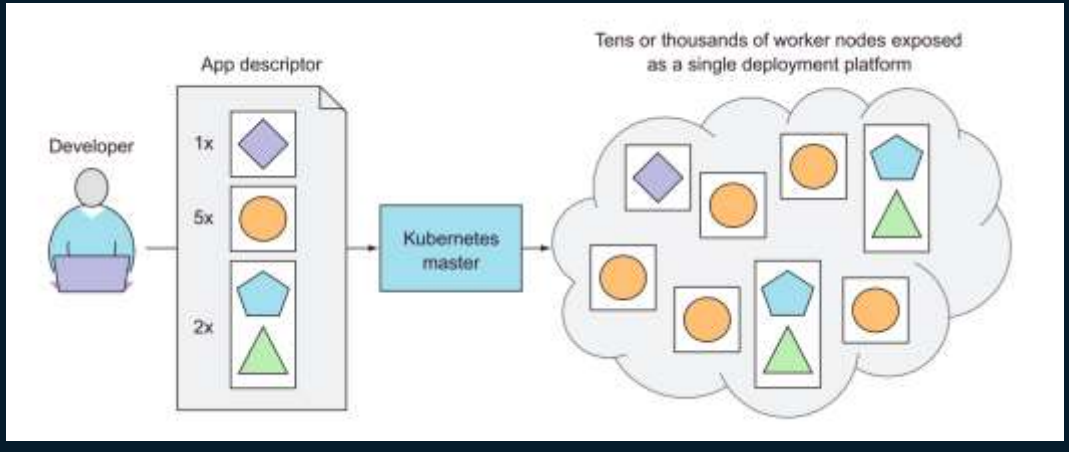
- Cluster
  - A set of nodes that can run containerized workloads. The number of nodes represent the number of resources of a cluster.
- Node
  - A machine that runs containerized workloads as part of a cluster.

Kubernetes Architecture

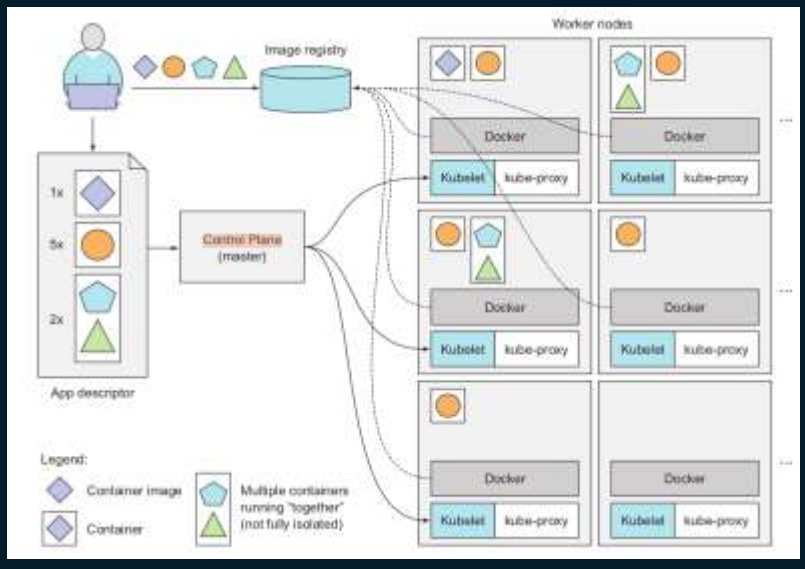
A cluster is composed of two types of nodes:

- Worker node
  - kubelet
  - kube-proxy
  - Container runtime (e.g. Docker, rkt, containerd, CRI-O)
- Control plane
  - etcd (cluster store)
  - API server
  - Scheduler
  - Controller manager

High-Level View of Kubernetes



Lower-Level View of Kubernetes



More kubectl Commands

USEFUL FOR TROUBLESHOOTING

- View event log

```
kubectl get events
```

- Get list of objects

```
kubectl get pods  # all or comma-separated
```

- Get details of an object

```
kubectl describe pod <pod-id>
```

- Show log output of a pod

```
kubectl logs <pod-id>
```

Let's do some lab exercises

KUBERNETES TROUBLESHOOTING

You'll get to:

- Try some more `kubectl` commands

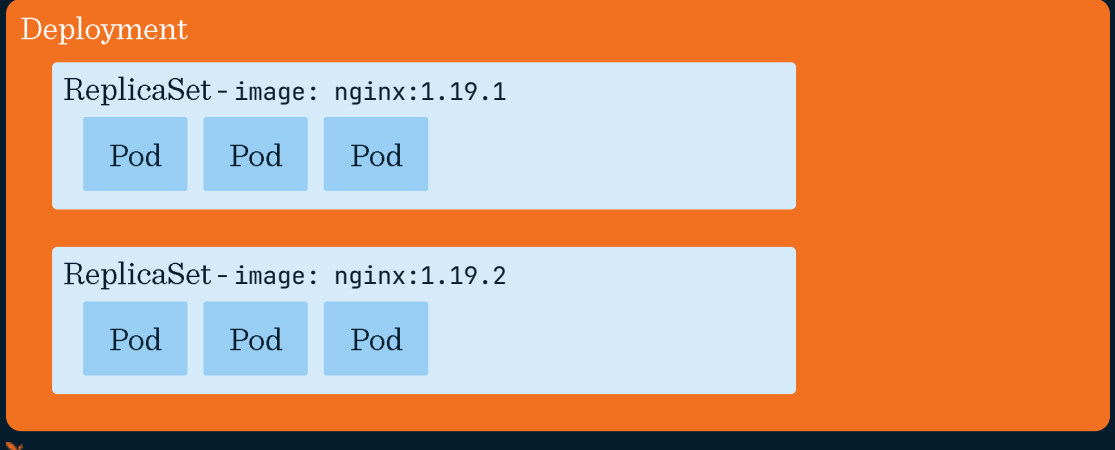
Managing Applications

Deployment Strategies – Built-In

- RollingUpdate
  - Default, if not specified
  - New ReplicaSet is created, then scaled up as the existing ReplicaSet is scaled down
- Recreate
  - Removes all old Pods in the existing ReplicaSet first
  - Then creates new Pods in the new ReplicaSet

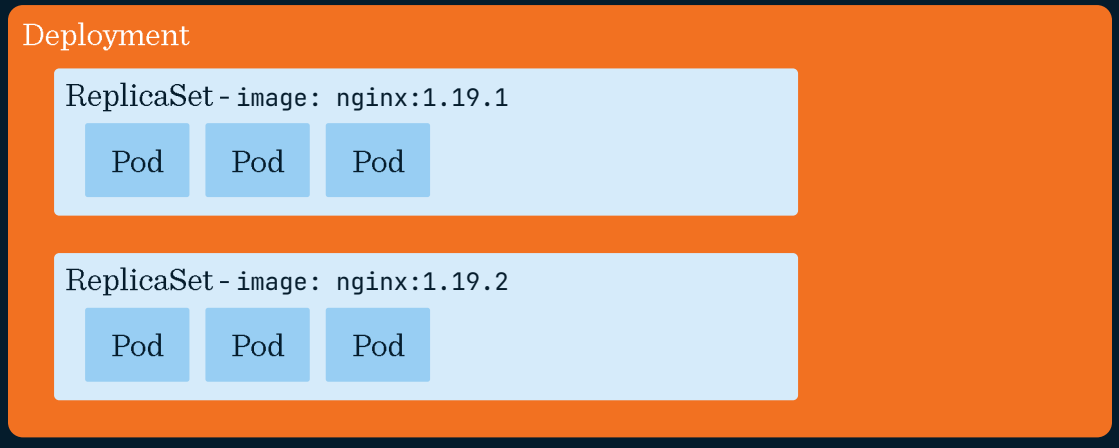
RollingUpdate

DEPLOYMENT STRATEGIES – BUILT-IN



Recreate

DEPLOYMENT STRATEGIES – BUILT-IN

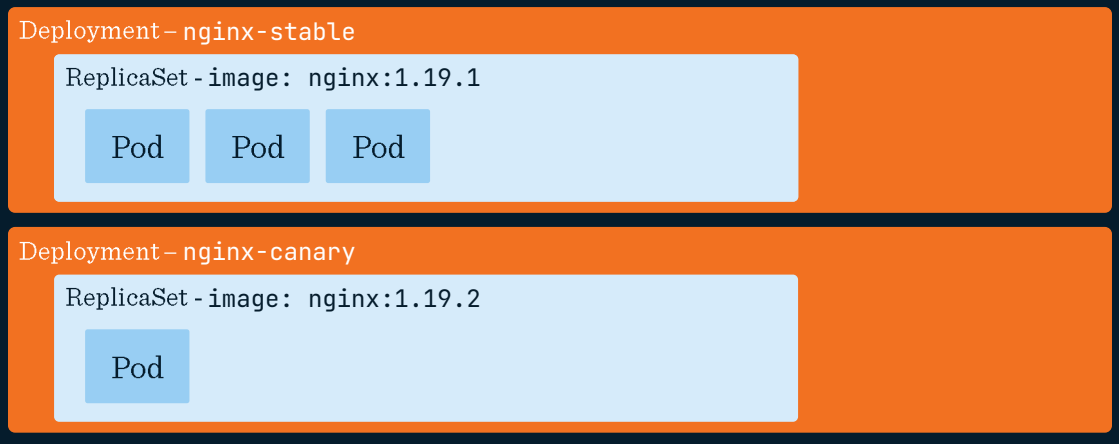


Deployment Strategies – Others

- Canary
  - Deploy new container to subset of traffic
  - Involves two Deployments and label management between them
- Blue/Green
  - Deploy all new containers, test, then "flip/switch" traffic to new containers
  - Involves two Deployments and label management between them

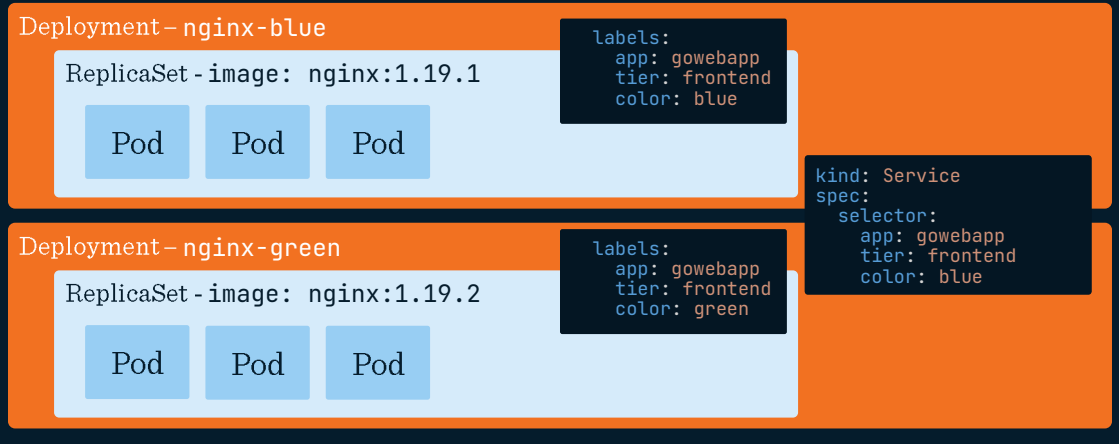
Canary

DEPLOYMENT STRATEGIES – OTHERS



Blue/Green

DEPLOYMENT STRATEGIES – OTHERS



Let's do some lab exercises

DEPLOYMENT MANAGEMENT

You'll get to:

- Create a new version of the application
- Try a rolling update deployment
- Try a canary deployment

Probes

- Liveness
  - Determine whether container is running (or not)
  - Upon failure – container will be restarted according to policy
- Readiness
  - Determine whether container is ready to service requests
  - Upon failure – pod is removed from service (so no requests are sent to it)
- Startup
  - Determine whether container **has properly started**
  - If specified, disables other (liveness, readiness) probes until successful

Probe Handlers

- Exec
  - Run a command inside the container
  - Success: return code from command is `0`
- HTTP
  - Invoke HTTP GET against a URL
  - Success: Any 2xx or 3xx HTTP response
- TCP
  - TCP check
  - Success: port is open and accepting connection
- gRPC
  - Since 1.24+
  - Success: health check returns SERVING status

Example

PROBES

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep int
# continued
```

```yaml
# .spec.containers[0]
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Configuration Options

PROBES

- `initialDelaySeconds`
  - How long to delay (first attempt at) probing
- `periodSeconds`
  - How often to perform the probe; default = 10
- `timeoutSeconds`
  - Default = 1
- `successThreshold`
  - Minimum consecutive successes for the probe to be considered successful; default = 1
- `failureThreshold`
  - Minimum consecutive failures for the probe to be considered failed; default = 3

Resource Management

- Resource *requests* and *limits*
- Resource types: CPU, memory

Resource Types

RESOURCE MANAGEMENT

- 1 CPU unit
  - 1 AWS vCPU == 1 GCP Core == 1 Azure vCore == 1 Hyperthread on a bare-metal processor with Hyperthreading
- CPU values
  - 0.1 == 100m == one hundred millicpu == 10% of a CPU unit
- Memory unit
  - E, P, T, G, M, K - 10-based
  - Ei, Pi, Ti, Gi, Mi, Ki - power-of-2-based
- Memory values
  - 1Gi == 1.073741824 GB
  - 1G == 1.0 GB

Resource Requests

RESOURCE MANAGEMENT

- Can be thought of as "minimum resource required" specification
- Helps Kubernetes more efficiently schedule Pods
- Pods will be scheduled if the sum of the resource requests of the scheduled containers is less than the capacity of the node

```
# .spec.containers[0]
resources:
  requests:
    memory: "128Mi"
    cpu: "500m"
  limits:
    memory: "512Mi"
    cpu: "1"
```

Resource Limits

RESOURCE MANAGEMENT

- Protect against a runaway app
- If a container exceeds its memory limit, it might be terminated

```
# .spec.containers[0]
resources:
  requests:
    memory: "128Mi"
    cpu: "500m"
  limits:
    memory: "512Mi"
    cpu: "1"
```


Let's do some lab exercises

POD AND CONTAINER CONFIGURATION

You'll get to:

- Work with resource requests and limits
- Work with liveness and readiness probes

Kubernetes Networking

Networking

- Within a Pod
- Pod to Pod
- Services to Pods
- External to Cluster

Within a Pod

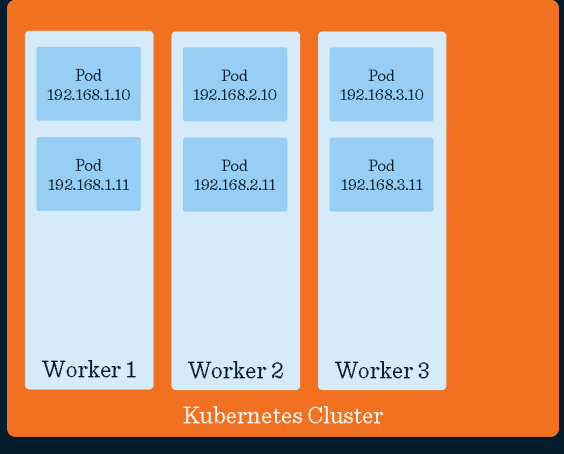
NETWORKING

- Containers within a Pod
  - Can connect to each other using `localhost`
  - Share an IP address accessible throughout the cluster
  - Share a common port space, beware of conflicts
- These capabilities closely mimic those of processes running on the same virtual machine

Pod to Pod

NETWORKING

- Every pod is assigned an IP address
- This IP address is routable anywhere within the cluster



Services to Pods

NETWORKING

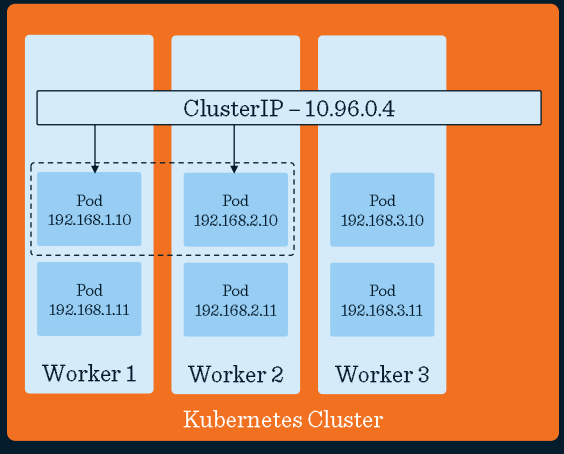
- A Service is a Kubernetes resource that:
  - Provides layer-4 load balancing for a group of pods
  - Service discovery using the cluster's internal DNS
- Several types of services are available:
  - ClusterIP *(default)*
  - NodePort
  - LoadBalancer
  - ExternalName

```yaml
apiVersion: v1
kind: Service
spec:
  type: NodePort #
  selector:
    app: gowebapp
    tier: frontend
```


ClusterIP

SERVICES TO PODS

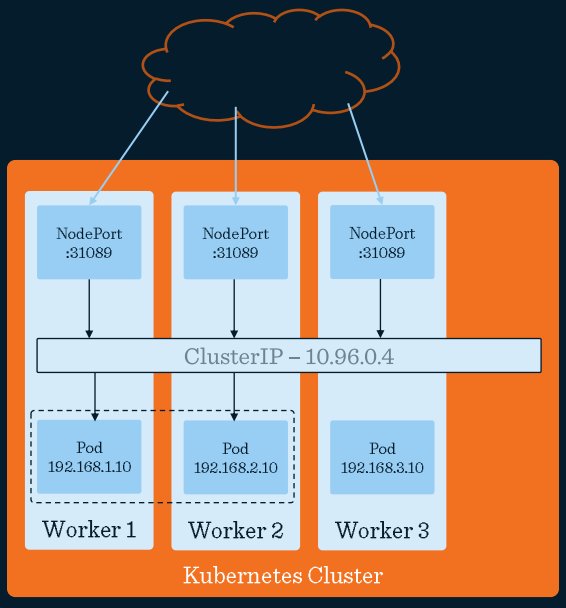
- Used for *internal*-facing services
- Implementation
  - A virtual IP address that load balances requests to a set of (backend) pods
  - Accessible anywhere within the cluster
  - Not externally accessible



NodePort

SERVICES TO PODS

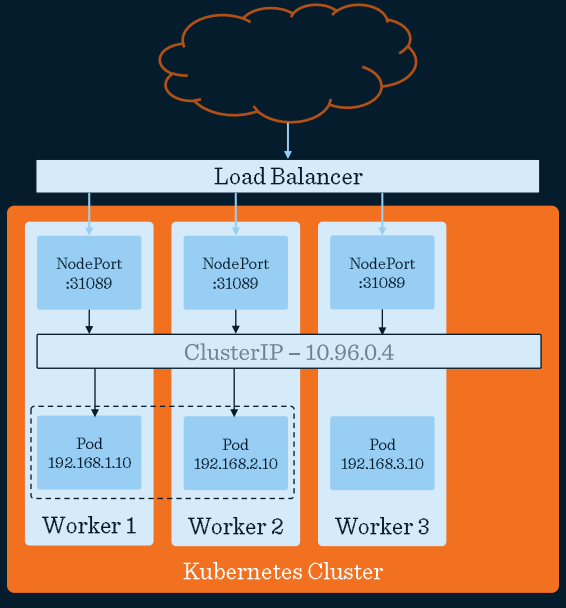
- Used for *external*-facing services
- Implementation
  - Exposes a port on each worker node
  - Externally accessible
  - Leverages ClusterIP for load balancing to pods



LoadBalancer

SERVICES TO PODS

- Creates and manages an external load balancer
- Implementation
  - Leverages NodePort for traffic ingress
  - Leverages ClusterIP for load balancing to pods
  - Plugins available for different load balancer implementations



Ingress Controller to Pods

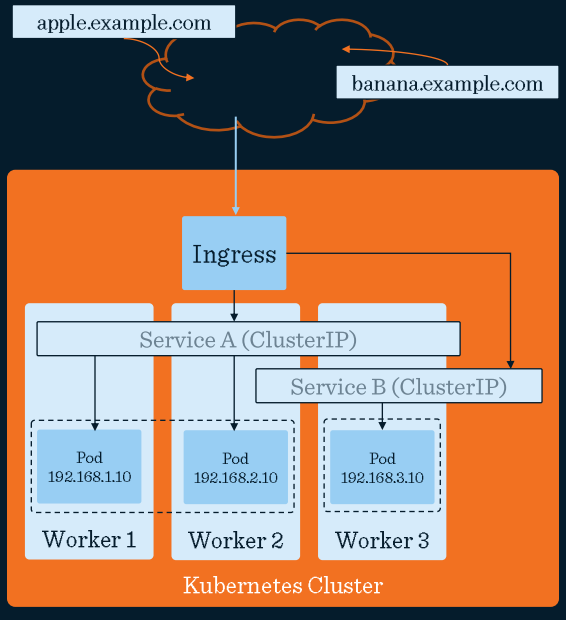
EXTERNAL TO CLUSTER

- An Ingress Controller is a feature which provides:
  - Layer-7 load balancing for one or more services
  - Additional capabilities, depending on implementation
- Many Ingress Controller implementations are available
  - NGINX
  - Contour
  - Traefik
  - Amazon ALB

Ingress Controller

EXTERNAL TO CLUSTER

- Used for external-facing layer 7 services
- Implementation
  - Uses host header and path evaluation to direct traffic
  - Externally accessible
  - Configured with Ingress object



Example

INGRESS

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
spec:
  ingressClassName: nginx
  rules:
  - host: apple.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: apple
# continued
```

```yaml
# .spec.rules[1]
  - host: banana.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: banana
```


Let's do some lab exercises

KUBERNETES NETWORKING

You'll get to:

- Enhance the existing application configuration with an ingress (via ingress controller)

Resource Organization

119

Clusters

RESOURCE ORGANIZATION

- Highest level of isolation, but comes with extra management
- Examples
  - Environments: Production, QA, Dev
  - Security: Compliance requirements
  - Geography: Different datacenters (e.g. to avoid WAN latency)

120

Namespaces

RESOURCE ORGANIZATION

- Names of resources must be unique *within* a namespace
- Scopes DNS
  - `<service-name>.<namespace-name>.svc.cluster.local`
- Can apply resource and security/access restrictions
- Examples:
  - Teams: r&d, contractors, etc.
  - Systems: email, CRM, corporate-website, etc.

Labels

RESOURCE ORGANIZATION

- Can exist on basically any resource in Kubernetes
- Keys/Values are not enforced
- Use in selectors (e.g. services use selectors to target pods)
- Tips:
  - Keep registration list and report of labels across
  - Avoid compound label values:
    - `app: twitter-api` vs `app: twitter / tier: api`
  - Be consistent across namespaces and clusters (imagine what labels you would need if everything was in one huge namespace)

kubeconfig (kubectl Configuration)

- Lives at `~/.kube/config` (or `%USERPROFILE%\.kube\config`)
- Sections:
  - clusters
  - users (more like credentials)
  - contexts
- The name is *not* really the name of the cluster or user. It is just the name used to reference it in the kubeconfig.

Example kubeconfig

```yaml
apiVersion: v1
kind: Config
current-context: docker-desktop
preferences: {}
clusters:
- name: docker-desktop
  cluster:
    certificate-authority: #REDACTED
    server: #REDACTED
contexts:
- name: docker-desktop
  context:
    cluster: docker-desktop
    user: docker-desktop
users:
- name: docker-desktop
  user:
    token: #REDACTED
```

Global Options

KUBECTL

- Applies to all commands
- Namespace
  - Run command for the specified namespace
    - `--namespace=''` or `-n`
  - Change *default* namespace for a context
    - `kubectl config set-context <context> --namespace=<namespace>`

Let's do some lab exercises

RESOURCE ORGANIZATION

You'll get to:

- Work with namespaces
- Use the `kubectl` configuration file

Storage and Stateful Applications

Storage and Stateful Applications

- Volumes
- Persistent volumes
- Persistent volume claims
- StatefulSets

Volumes

- Abstraction of the storage provider (e.g. AWS EBS)
- Exposed at the pod-level and backed different ways
- `emptyDir` - ephemeral scratch directory that lives for the life of a pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: test
    image: busybox
    volumeMounts: # 👈
    - name: cache-vol
      mountPath: /cache
  volumes: # 👈
  - name: cache-vol
    emptyDir:
      sizeLimit: 500Mi
```

Persistent Volumes

- Have a lifecycle independent of any individual pod that uses it.
- Can be provisioned:
  - Statically
    - A pre-provisioned pool of volumes such as iSCSI (SCSI over IP) or Fiber Channel disk from a SAN (Storage Area Network)
  - Dynamically
    - Volumes are created *on-demand* by calling a storage provider's API (such as Amazon EBS)

Persistent Volume Claims

- Created to request a persistent volume
- The request includes various properties such as capacity and access mode (e.g. can be mounted once read/write, can be mounted many times read-only)

Example

PVC AND POD

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  resources:
    requests:
      storage: 4G
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
```

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: my-mysql
    image: mysql:8.0
    volumeMounts:
    - mountPath: "/var/lib/mysql"
      name: mysql-pv
  volumes:
  - name: mysql-pv
    persistentVolumeClaim:
      claimName: mysql-pvc
```

StatefulSets

- Stable, unique pod identifiers and DNS names
  - `${statefulset-name}-${ordinal}.${service-name}.${namespace}`
- Stable, persistent storage
  - One PV per VolumeClaimTemplate
- Controlled deployment order
  - Pods created in ascending order, deleted in descending order
  - Before scaling, all pods must be Running or Ready

Example

STATEFULSET

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-sts
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: web # 👈
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
# continued
```

```yaml
# .spec.template
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: www
          mountPath: #...
  volumeClaimTemplates: # 👈
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Let's do some lab exercises

STORAGE AND STATEFUL APPLICATIONS

You'll get to:

- Create a StatefulSet
- See how a StatefulSet ends up using a PersistentVolume
- Demonstrate that the application state can be restored after deleting and recreating the pod.

Dynamic Application Configuration

Image Configurability

- All runtime configuration should be overridable
  - Defaults inside a container image are fine
  - Enables configuration to be supplied by Kubernetes and the environment
- Choose from two approaches
  - Provide defaults and startup if no configuration is provided
  - Require configuration and fail to startup if missing

ConfigMaps

- Collection of arbitrary key/value pairs
- Namespace specific

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
  namespace: qa
data:
  # property-like keys
  TLS_ENABLED: "false"
  # file-like keys
  server.properties: |
    MAX_THREADS = 5
    STRATEGY = FIFO
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
  namespace: prod
data:
  # property-like keys
  TLS_ENABLED: "true"
  # file-like keys
  server.properties: |
    MAX_THREADS = 25
    STRATEGY = FIFO
```

As Environment Variables

USING CONFIGMAPS

```yaml
# Load all keys

# .spec.containers[]
image: myapp:v1.0
envFrom:
- configMapRef:
    name: my-config
```

```yaml
# Load individual key

# .spec.containers[]
image: myapp:v1.0
env:
- name: USE_HTTPS
  valueFrom:
    configMapRef:
      name: my-config
      key: TLS_ENABLED
```

As Mounted Volumes

USING CONFIGMAPS

```yaml
# Load all keys

# .spec
containers:
- name: #...
  image: myapp:v1.0
  volumeMounts:
  - name: my-config-vol
    mountPath: /app/config
volumes:
- name: my-config-vol
  configMap: # 👈
    name: my-config
```

```yaml
# Load individual key

# .spec
containers:
- name: #...
  image: myapp:v1.0
  volumeMounts:
  - name: my-config-vol
    mountPath: /app/config
volumes:
- name: my-config-vol
  configMap:
    name: my-config
    items:
    - key: server.propties
      path: server.properties
```

Secrets

- Resource that stores sensitive data such as a password, a token, or a key
- Must be Base64 encoded
- Like ConfigMaps, Secrets can be exposed to a Pod via environment variables and mounted volumes

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: <Base64-value>
```


Let's do some lab exercises

DYNAMIC APPLICATION CONFIGURATION

You'll get to:

- Update a container image to accept configuration at runtime
- Create a ConfigMap and a Secret and use them

Additional Workloads and Security

Additional Workloads

- Jobs
- CronJobs
- DaemonSets

Jobs

- Run to completion
- Container/Pod based
- Ensure to run to completion; retry forever by default
- Post-Job completion
  - Pod and Job resources will remain (to allow viewing of logs, output, etc)
  - Up to the administrator to implement/determine cleanup

Example

JOBS

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Job Types

- Non-parallel Jobs
  - Normally, only one Pod is started, unless the Pod fails.
  - Job is complete as soon as its Pod terminates successfully.
- Parallel Jobs with a *fixed completion count*
  - Set a non-zero positive value for `.spec.completions`.
- Parallel Jobs with a work queue
  - Leave `.spec.completions` unset. Set a non-zero positive value for `.spec.parallelism`.
  - Pods must coordinate amongst themselves or an external service to determine what each should work on.
  - When any Pod from the Job terminates with success, no new Pods are created.

CronJobs

- Creates Jobs on a repeating schedule
- Types:
  - Scheduled once at specified time
  - Repeated at specified time
- New Job resource objects created for reach run
- By default, 3 successful jobs, and 1 failed job are retained
- Jobs should be idempotent

Example

CRONJOBS

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

DaemonSets

- Ensures that all Nodes run a copy of a Pod
  - As nodes are added to the cluster, Pods are added to them
  - As nodes are removed from the cluster, those Pods are deleted
- Often used for cluster administration functions
- Common use cases
  - Cluster-wide logging alerts
  - Cluster-wide monitoring agents

Let's do some lab exercises

ADDITIONAL WORKLOADS

You'll get to:

- Create a Job
- Create a parallel Job
- Create a CronJob

NetworkPolicy

- Specification of how groups of pods are allowed to communicate with:
  - Each other
  - Other network endpoints
- Use labels to select pods and define rules which specify what traffic is allowed to the selected pods

NetworkPolicy

- Pods are non-isolated (default)
  - Accept traffic from any source
- Pods become isolated by:
  - Defining a NetworkPolicy that selects them in a namespace
  - Pod will reject any connections that are not explicitly allowed by a NetworkPolicy
  - Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic

Example - Inbound

NETWORKPOLICY

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress # or Egress, or both

# continued
```

```yaml
# .spec.ingress
  ingress: # or egress.to, or both
  - from:
    # any pod in namespaces
    # w/ label project=myproject
    - namespaceSelector:
        matchLabels:
          project: myproject
    # OR any pod in local namespace
    # w/ label tier=frontend
    - podSelector:
        matchLabels:
          tier: frontend
    ports:
    - protocol: TCP
      port: 3306
```

Example - Inbound

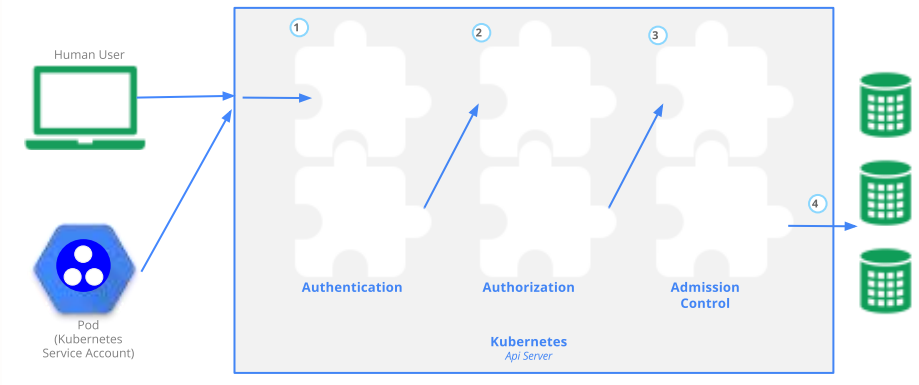
NETWORKPOLICY

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress

# continued
```

```yaml
# .spec.ingress
  ingress:
  - from:
    # any pod in namespaces
    # w/ label project=myproject
    - namespaceSelector:
        matchLabels:
          project: myproject
    # AND any pod
    # w/ label tier=frontend
      podSelector:
        matchLabels:
          tier: frontend
    ports:
    - protocol: TCP
      port: 3306
```


Controlling Access to Kubernetes API



Authentication Methods

- Client certificates
- Tokens
- External authentication

157

Role-Based Access Control (RBAC)

- Role / ClusterRole
- RoleBinding / ClusterRoleBinding

158

Role

ROLE-BASED ACCESS CONTROL (RBAC)

- Collection of permissions
- Standalone and must be bound to a subject
- Namespace specific

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
    # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```


RoleBinding

ROLE-BASED ACCESS CONTROL (RBAC)

- Connects one or more subjects to a Role

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef: # specifies the binding to a
  # Role / ClusterRole
  kind: Role # this must be Role / ClusterRole
  name: pod-reader # this must match the name of
    # the Role / ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

ClusterRole

ROLE-BASED ACCESS CONTROL (RBAC)

- Same as Role, except is global to the cluster
- Reusable across entire cluster

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since
  # ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  # at the HTTP level, the name of the
  # resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

ClusterRoleBinding

ROLE-BASED ACCESS CONTROL (RBAC)

- Same as RoleBinding, except is global to the cluster

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: john
```

Built-In Roles (1/2)

ClusterRole	Description
cluster-admin	Allows super-user access to perform any action on any resource. When used in a **ClusterRoleBinding**, it gives full control over every resource in the cluster and in all namespaces. When used in a **RoleBinding**, it gives full control over every resource in the role binding's namespace, including the namespace itself.
admin	Allows admin access, intended to be granted within a namespace using a **RoleBinding**. If used in a **RoleBinding**, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself.
edit	
view	

163

Built-In Roles (2/2)

ClusterRole	Description
cluster-admin	
admin	
edit	Allows read/write access to most objects in a namespace. This role does not allow viewing or modifying roles or role bindings.
view	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. This role does not allow viewing Secrets, since those are escalating.

164

Let’s do some lab exercises

SECURITY

You’ll get to:

- Scan the containerized image for vulnerabilities
- Create a new user and connect it to a role