



# Certified Kubernetes Application Developer (CKAD) Crash Course

*Kubernetes 1.18+ Edition*



# About the trainer



**bmuschko**



**bmuschko**



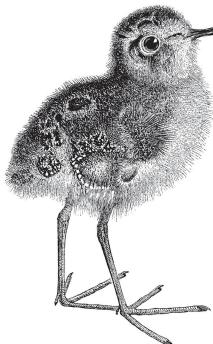
**bmuschko.com**



O'REILLY®

# Certified Kubernetes Application Developer (CKAD) Study Guide

In-Depth Guidance and Practice



Benjamin Muschko

*Companion study guide with practice questions*

**Expected release: early 2021**

Online access on O'Reilly learning platform:

<https://learning.oreilly.com/library/view/certified-kubernetes-application/9781492083726/>

# Exam Details and Resources

Objectives, Environment, Time Management

# Exam Objectives

*“Design, build, configure, and expose  
cloud native applications for Kubernetes”*



**The certification program allows users to demonstrate their competence in a hands-on, command-line environment.**

<https://www.cncf.io/certification/ckad/>



# The Curriculum

## 13% - Core Concepts

- Understand Kubernetes API primitives
- Create and configure basic Pods

## 10% Multi-Container Pods

- Understand Multi-Container Pod design patterns (e.g. ambassador, adapter, sidecar)

## 13% - Services & Networking

- Understand Services
- Demonstrate basic understanding of NetworkPolicies

## 20% - Pod Design

- Understand how to use Labels, Selectors, and Annotations
- Understand Deployments and how to perform rolling updates
- Understand Deployments and how to perform rollbacks
- Understand Jobs and CronJobs

## 18% - Configuration

- Understand ConfigMaps
- Understand SecurityContexts
- Define an application's resource requirements
- Create & consume Secrets
- Understand ServiceAccounts

## 18% - Observability

- Understand LivenessProbes and ReadinessProbes
- Understand container logging
- Understand how to monitor applications in Kubernetes
- Understand debugging in Kubernetes

## 8% - State Persistence

- Understand PersistentVolumeClaims for storage



# Candidate Skills



**kubernetes**

Architecture & Concepts



**kubectl**

Running Commands



Underlying Concepts



# Exam Environment

*Online and proctored exam*

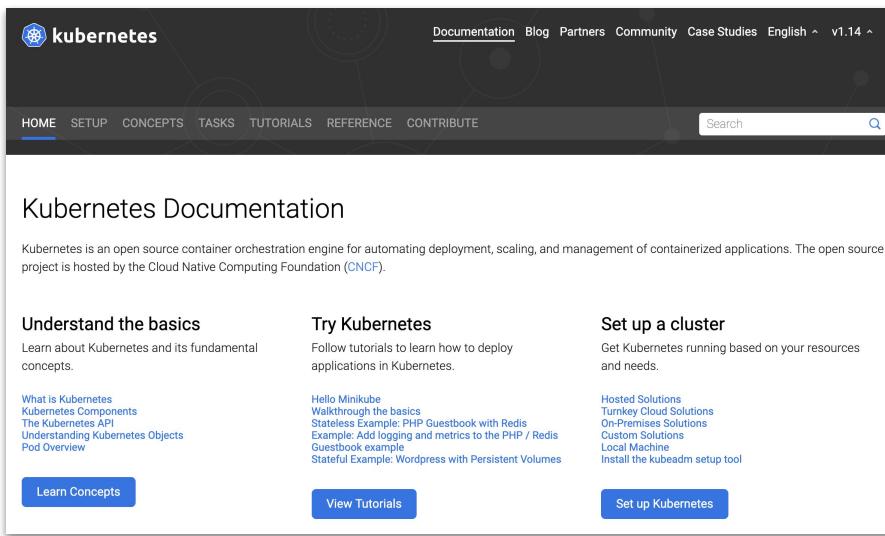


*The trinity of tooling you need to be familiar with*



# Using Documentation

*Know where and how to find relevant documentation*



The screenshot shows the Kubernetes Documentation website. At the top, there's a navigation bar with links for Documentation, Blog, Partners, Community, Case Studies, English, and v1.14. Below that is a secondary navigation bar with links for HOME, SETUP, CONCEPTS, TASKS, TUTORIALS, REFERENCE, and CONTRIBUTE. A search bar is also present. The main content area features a title "Kubernetes Documentation" and a brief introduction: "Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF)." There are three main sections: "Understand the basics", "Try Kubernetes", and "Set up a cluster". Each section has a brief description and a list of related topics or tutorials. At the bottom, there are three blue call-to-action buttons: "Learn Concepts", "View Tutorials", and "Set up Kubernetes".

<https://kubernetes.io/docs>



# Getting Help on a Command

*Render subcommands and options with --help*

```
$ kubectl create --help
Create a resource from a file or from stdin.

JSON and YAML formats are accepted.

...
Available Commands:

...
  configmap      Create a configmap from a local file, directory or literal
  value
  deployment     Create a deployment with the specified name.
  ...

Options:
  ...
```



# Zeroing in on Command Details

*Drill into object details with the `explain` command*

```
$ kubectl explain pods.spec  
KIND:     Pod  
VERSION:  v1
```

```
RESOURCE: spec <Object>
```

```
DESCRIPTION:
```

```
...
```

```
FIELDS:
```

```
...
```

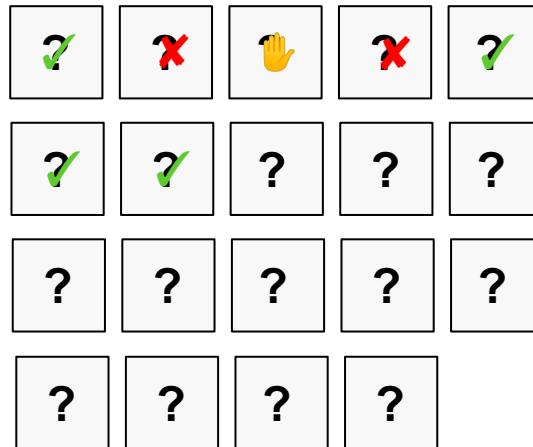
Most relevant information



---

# Time Management

*19 problems in 2 hours, use your time wisely!*



# Using an Alias for kubectl

*Your first action at the beginning of the exam*

```
$ alias k=kubectl  
$ k version  
...
```



# Setting a Context & Namespace

*Questions will ask you to run a command on a specific cluster - Make sure to execute it!*

```
$ kubectl config set-context <context-of-question>--  
--namespace=<namespace-of-question>
```



# Internalize Resource Short Names

*Some API resources provide a shortcut*

```
$ kubectl get ns
```

Usage of `ns` instead  
of namespaces

```
$ kubectl describe pvc claim
```

Usage of `pvc` instead of  
`persistentvolumeclaim`



# Deleting Kubernetes Objects

*Don't wait for a graceful deletion of objects...*

```
$ kubectl delete pod nginx --grace-period=0 --force
```



# Understand and Practice bash

*Practice relevant syntax and language constructs*

```
$ if [ ! -d ~/tmp ]; then mkdir -p ~/tmp; fi; while true; do echo $(date) >> ~/tmp/date.txt; sleep 5; done;
```



# Finding Object Information

*Filter configuration with context from a set of objects*

```
$ kubectl describe pods | grep -C 10 "author=John Doe"  
$ kubectl get pods -o yaml | grep -C 5 labels:
```

grep is your friend!



---

# How to Prepare

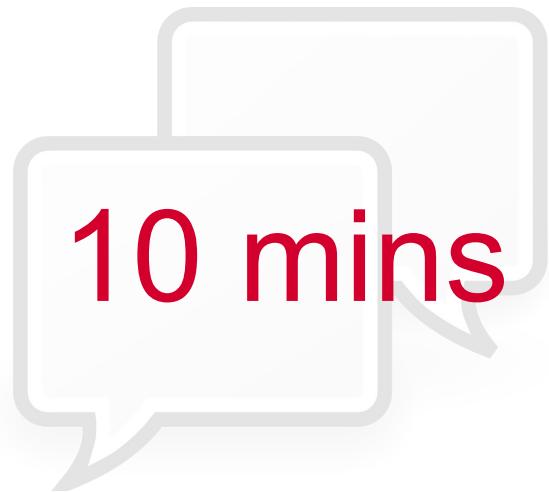
*Practice, practice, practice!*

**The key to cracking the exam**



---

# Q & A



---

# BREAK



# Core Concepts

Kubernetes API Primitives and Pod Management

# Kubernetes Object Structure

## Kubernetes Object

**API Version** *v1, apps/v1, ...*

**Kind** *Pod, Deployment, Quota, ...*

### Metadata

*Name, Namespace, Labels, ...*

### Spec

*Desired state*

### Status

*Actual state*

## Object representation in YAML

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
    name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  status: {}
```



---

# Object Management

*Different approaches for different use cases*



vs.

YAML



# Imperative Object Management

*Fast but requires detailed knowledge, no track record*

```
$ kubectl create namespace ckad
$ kubectl run nginx --image=nginx -n ckad
$ kubectl edit pod/nginx -n ckad
```



# Declarative Object Management

*Suitable for more elaborate changes, tracks changes*

```
$ vim ngix-pod.yaml  
$ kubectl create -f ngix-pod.yaml  
$ kubectl delete pod/nginx
```



# Hybrid Approach

*Generate YAML file with kubectl but make further edits*

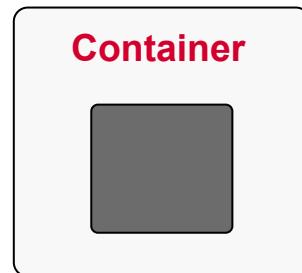
```
$ kubectl run nginx --image=nginx --dry-run=client<  
-o yaml > ngix-pod.yaml  
$ vim ngix-pod.yaml  
$ kubectl create -f ngix-pod.yaml
```



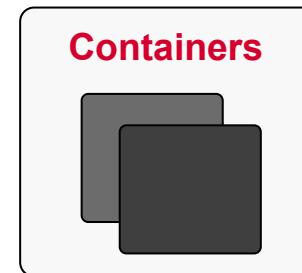
# Understanding Pods

*Wrapper around one or many containers*

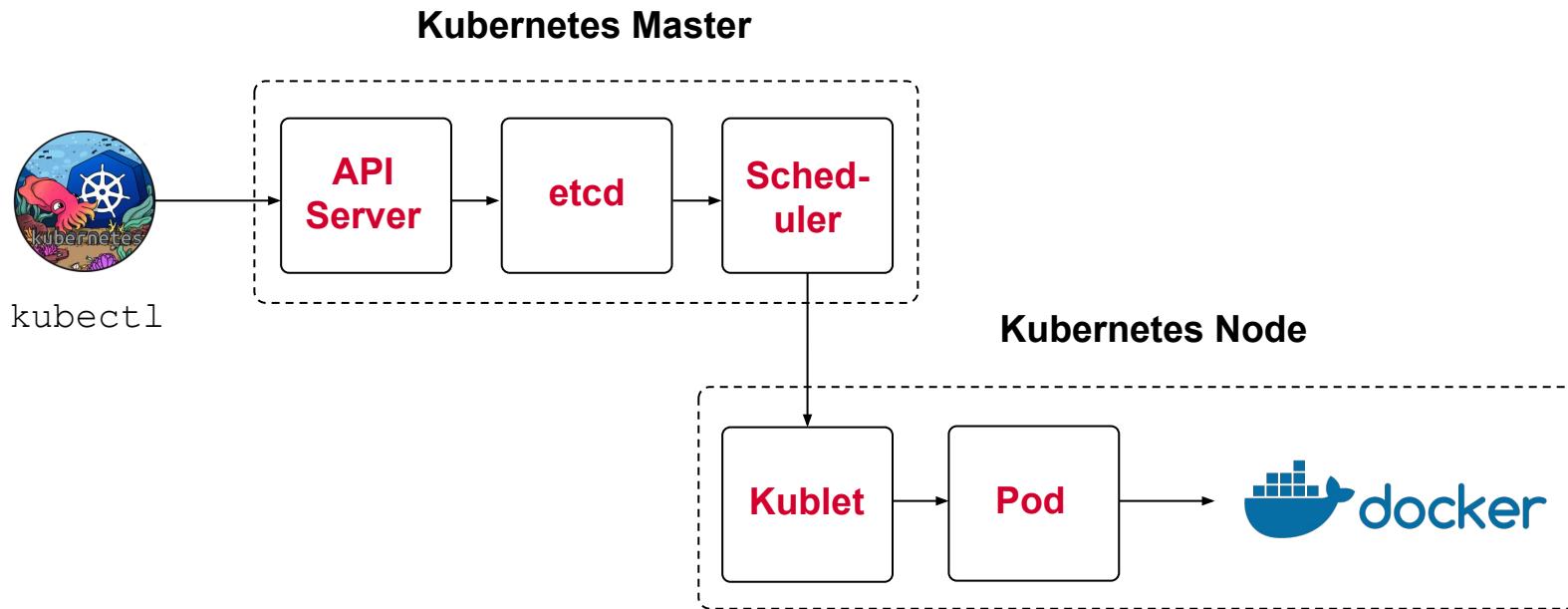
Single-container Pod



Multi-container Pod



# Pod Creation Flow



# Pod Lifecycle Phases

Phases and their meaning	
<b>Pending</b>	The Pod has been accepted by the Kubernetes system, but one or more of the container images has not been created.
<b>Running</b>	At least one container is still running, or is in the process of starting or restarting.
<b>Succeeded</b>	All containers in Pod terminated successfully.
<b>Failed</b>	Containers in Pod terminated, at least one failed with an error.
<b>Unknown</b>	State of the Pod could not be obtained.



# Inspecting a Pod's Status

```
$ kubectl describe pods nginx | grep Status:  
Status:          Running
```

Get current status  
and event logs

```
$ kubectl get pods nginx -o yaml  
...  
status:  
  conditions:  
  ...  
  containerStatuses:  
    ...  
    state:  
      running:  
        startedAt: 2019-04-24T16:56:55Z  
    ...  
  phase: Running
```

Get current  
lifecycle phase



# Configuring Env. Variables

*Injecting runtime behavior*

```
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-app
spec:
  containers:
  - image: bmuschko/spring-boot-app:1.5.3
    name: spring-boot-app
    env:
    - name: SPRING_PROFILES_ACTIVE
      value: production
```



---

# Commands and Arguments

*Running a command inside of a container*

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:1.15.12
    name: nginx
    args:
    - /bin/sh
    - -c
    - echo hello world
```



# Other Useful kubectl Commands

```
$ kubectl logs busybox  
hello world
```

Dump the Pod's logs

```
$ kubectl exec nginx -it -- /bin/sh  
# pwd
```

Connecting to a running Pod



---

# EXERCISE

Creating a Pod  
and Inspecting it



---

# Q & A



---

# BREAK

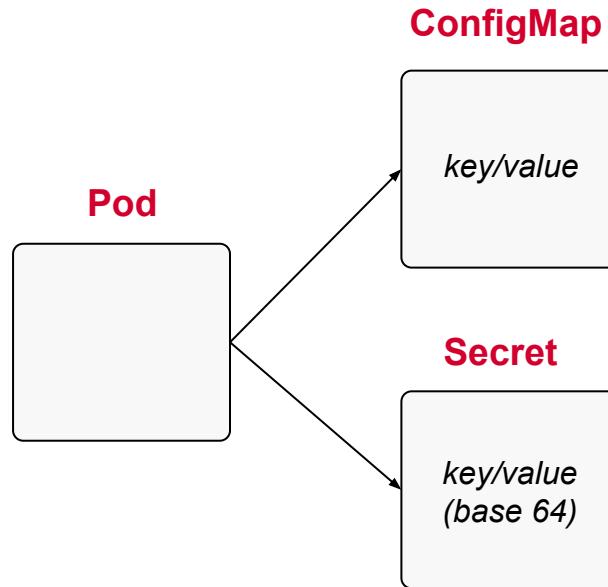


# Configuration

ConfigMaps, Secrets, Security Contexts, Resource Requirements and Service Accounts

# Centralized Configuration Data

*Injects runtime configuration through object references*



# Creating ConfigMaps (imperative)

*Fast, easy and flexible, can point to different sources*

```
# Literal values
$ kubectl create configmap db-config --from-literal=db=staging

# Single file with environment variables
$ kubectl create configmap db-config --from-env-file=config.env

# File or directory
$ kubectl create configmap db-config --from-file=config.txt
```



# Creating ConfigMaps (declarative)

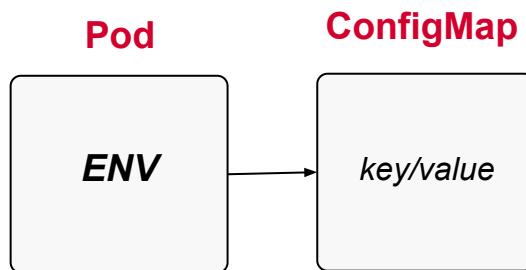
*Definition of a ConfigMap is fairly short and on point*

```
apiVersion: v1
data:
  db: staging
  username: jdoe
kind: ConfigMap
metadata:
  name: db-config
```

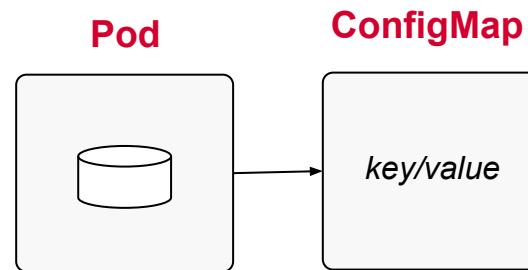


# Mounting a ConfigMap

*Two options for consuming data*



Injected as environment variables



Mounted as volume



# ConfigMap Env. Variables in Pod

*Convenient if ConfigMap reflects the desired syntax*

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
      envFrom:
        - configMapRef:
            name: db-config
```

```
$ kubectl exec -it nginx -- env
DB=staging
USERNAME=jdoe
...
```



# ConfigMap in Pod as Volume

*Each key becomes file in mounted directory*

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - name: backend
      image: nginx
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: db-config
```

```
$ kubectl exec -it backend -- /bin/sh
# ls /etc/config
db
username
# cat /etc/config/db
staging
```



---

# EXERCISE

Configuring a Pod  
to Use a ConfigMap



# Creating Secrets (imperative)

*Similar usage to creation of ConfigMap*

```
# Literal values
$ kubectl create secret generic db-creds ←
  --from-literal=pwd=s3cre!

# File containing environment variables
$ kubectl create secret generic db-creds ←
  --from-env-file=secret.env

# SSH key file
$ kubectl create secret generic db-creds ←
  --from-file=ssh-privatekey=~/ssh/id_rsa
```



# Creating Secrets (declarative)

*Value has to be base64-encoded manually*

```
$ echo -n 's3cre!' | base64  
czNjcmUh
```

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
  type: Opaque  
data:  
pwd: czNjcmUh
```



# Secret in Pod as Volume

*Value has to be base64-encoded manually*

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - name: backend
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret
  volumes:
    - name: secret-volume
      secret:
        secretName: mysecret
```

```
$ kubectl exec -it backend -- /bin/sh
# ls /etc/secret
pwd
# cat /etc/secret/pwd
s3cre!
```



---

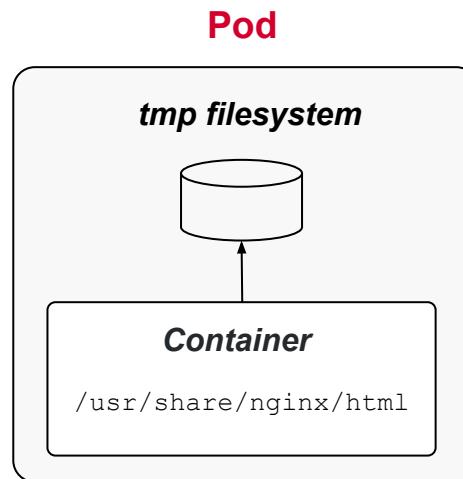
# EXERCISE

Configuring a Pod  
to Use a Secret



# Understanding Security Contexts

*Privilege and access control settings for a Pod or container*



“Create files with a specific group ID”

“Run this container with a specific user ID”



# Defining a Security Context

*Pod- vs. container-level definition*

```
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
spec:
  securityContext: ← Defined on the Pod-level
    runAsUser: 1000
  containers:
  - securityContext: ← Defined on the container-level
    runAsGroup: 3000
```



---

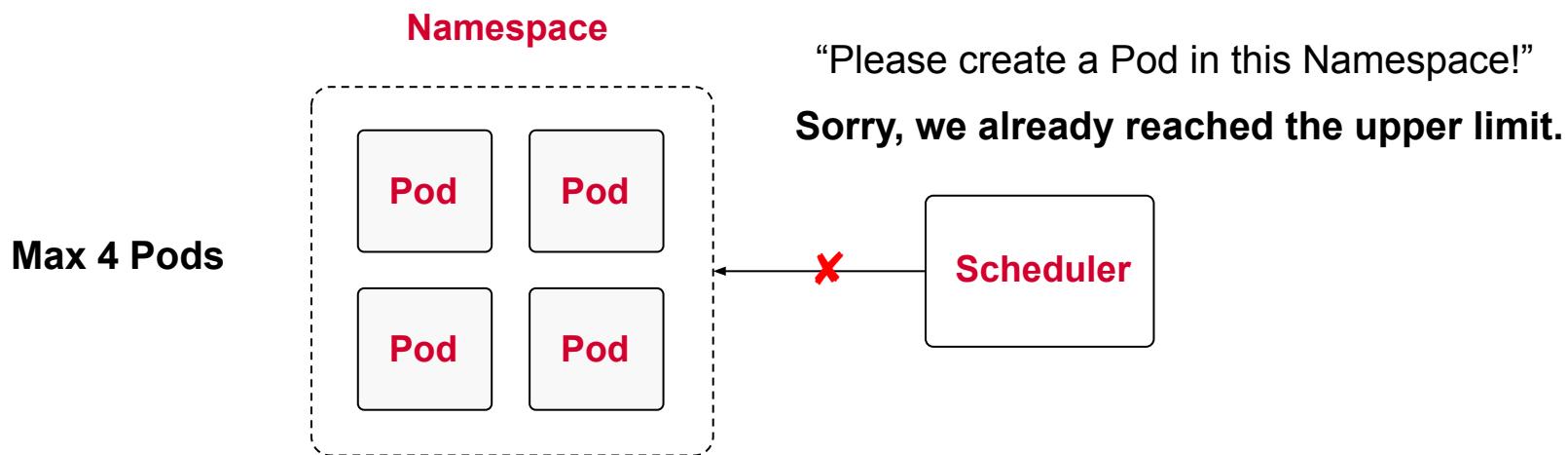
# EXERCISE

Creating a Security  
Context for a Pod



# Defining Resource Boundaries

*Defines # of Pods, CPU and memory usage per Namespace*



# Creating a Resource Quota

*Definition on the Namespace-level*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: app
spec:
  hard:
    pods: "2"
    requests.cpu: "2"
    requests.memory: 500m
```

```
$ kubectl create namespace rq-demo
$ kubectl create -f rq.yaml
--namespace=rq-demo
resourcequota/app created
$ kubectl describe quota --namespace=rq-demo
Name:          app
Namespace:     rq-demo
Resource       Used   Hard
-----
pods           0      2
requests.cpu   0      2
requests.memory 0     500m
```



# Defining Container Constraints

*Required if Namespace defines Resource Quota*

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: nginx
    name: mypod
    resources:
      requests:
        cpu: "0.5"
        memory: "200m"
```

Requires at least 0.5 CPU resources  
and 200m of memory



---

# EXERCISE

Defining a Pod's  
Resource  
Requirements



# Declaring Service Accounts

*Provides identity for processes running in a Pod*

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  serviceAccountName: myserviceaccount
```



---

# EXERCISE

Using a Service  
Account



---

# Q & A



---

# BREAK

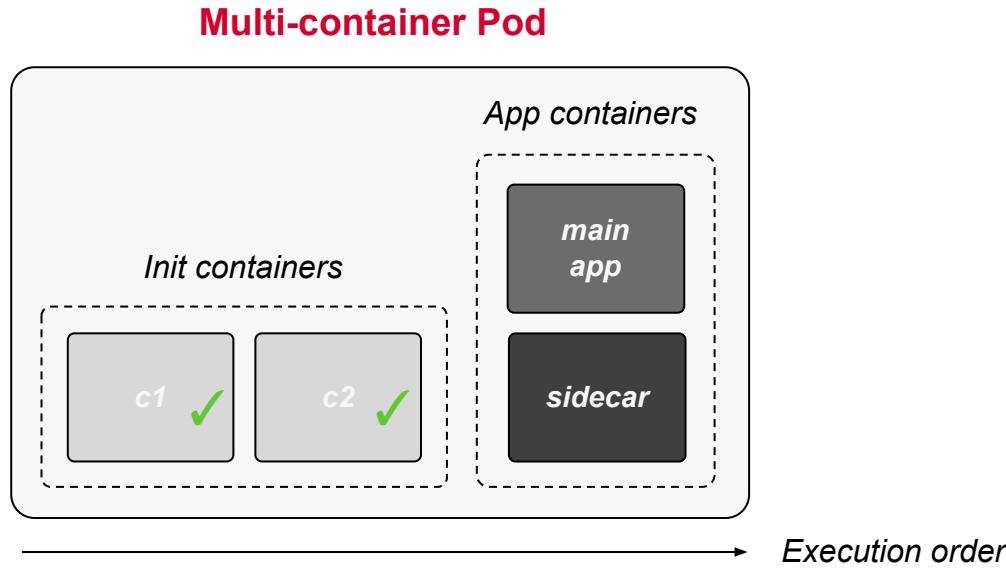


# Multi-Container Pods

Common Design Patterns

# Init Container

*Initialization logic before main application containers*



---

# EXERCISE

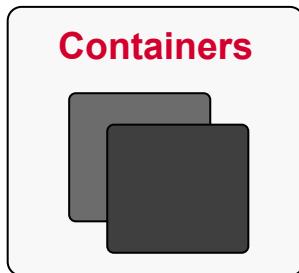
Creating an Init  
Container



# Defining Multiple Containers

*Shared container lifecycle and resources*

## Multi-container Pod



```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container
spec:
  containers:
  - image: nginx
    name: container1
  - image: nginx
    name: container2
```



# Targeting Different Containers

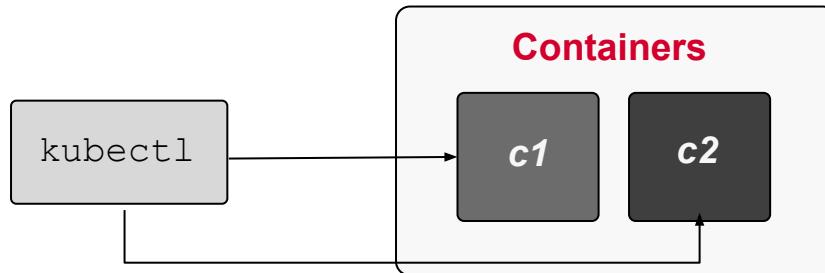
```
$ kubectl logs busybox --container=c1
```

Dump logs of  
container 1

```
$ kubectl exec busybox -it --container=c2 -- /bin/sh
```

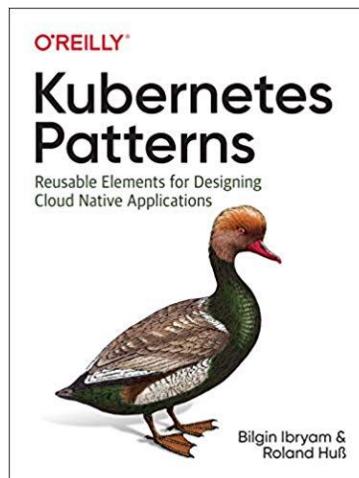
Log into  
container 2

Multi-container Pod



# Multi-Container Patterns

*Understand patterns on a high-level*

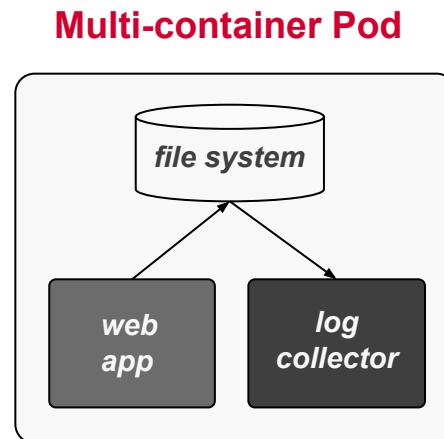


- Init container
- Sidecar
- Adapter
- Ambassador



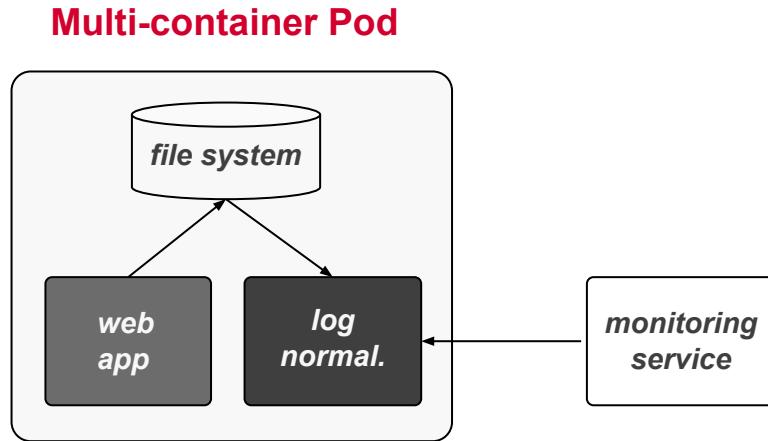
# Sidecar Pattern

*Enhance logic of main application container*



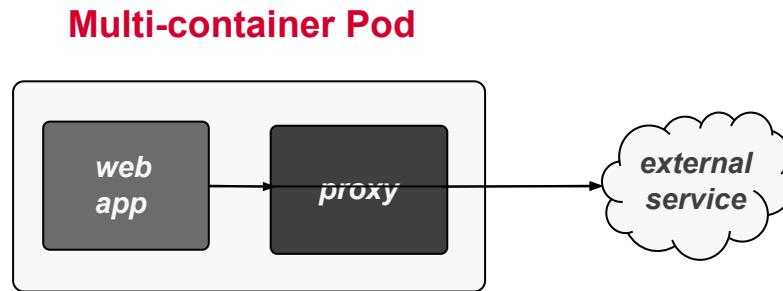
# Adapter Pattern

*Standardizes and normalizes application output read by external monitoring service*



# Ambassador Pattern

*Proxy for main application container*



---

# EXERCISE

Implementing the  
Adapter Pattern



---

# Q & A



# Observability

Probes, Logging, Monitoring and Debugging

---

# Container Health

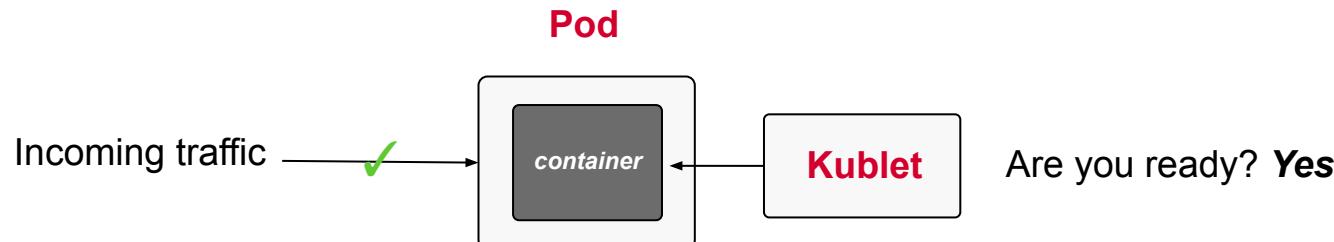
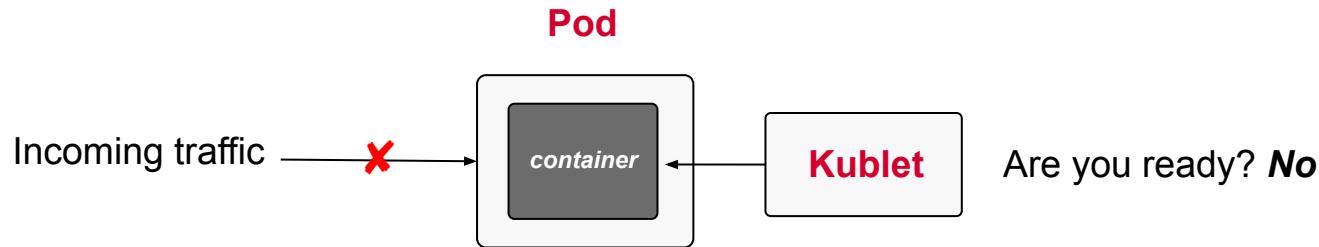
*“How does Kubernetes know if a container is up and running?”*

Probes can detect  
and correct failures



# Understanding Readiness Probes

*“Is application ready to serve requests?”*



# Defining a Readiness Probe

*HTTP probes are very helpful for web applications*

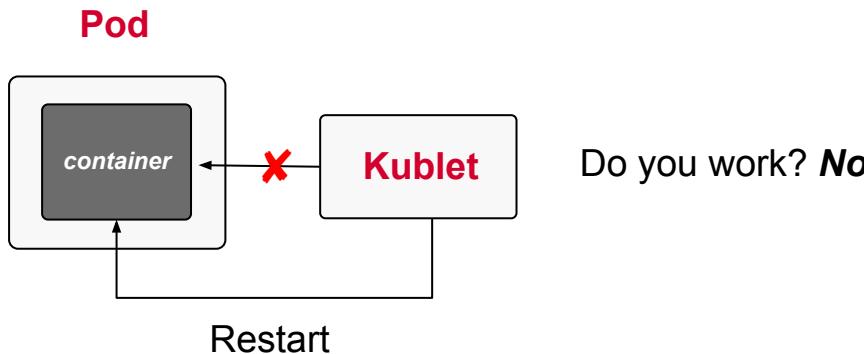
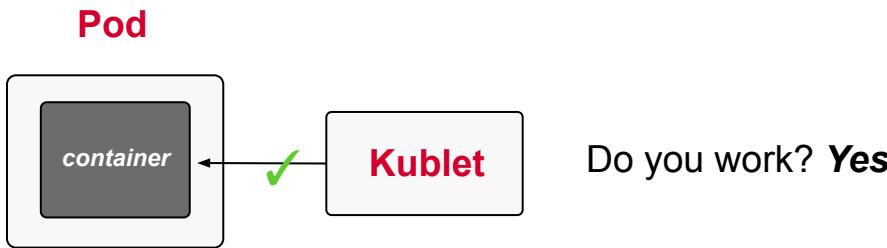
```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
    - name: web-app
      image: eshop:4.6.3
      readinessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 2
```

Successful if HTTP status code is  
between 200 and 399



# Understanding Liveness Probes

*“Does the application still function without errors?”*



# Defining a Liveness Probe

*An event log can be queried with a custom command*

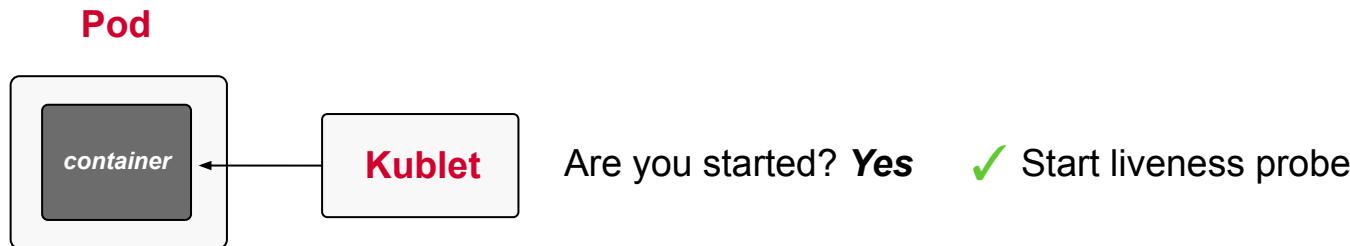
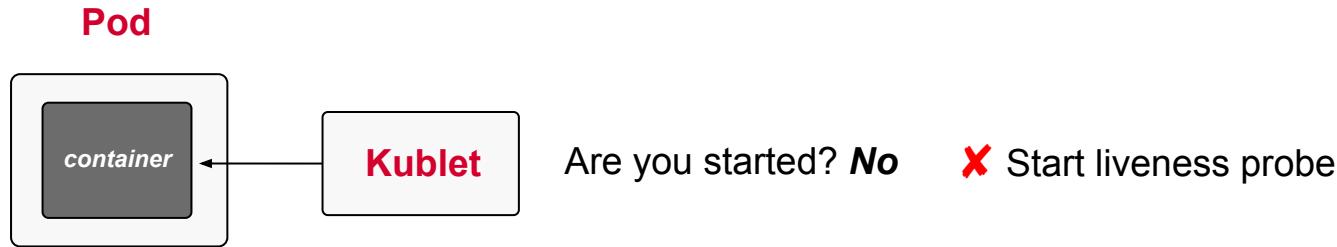
```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
    - name: web-app
      image: eshop:4.6.3
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 10
        periodSeconds: 5
```

It makes sense to delay the initial check as the application to fully start up first



# Understanding Startup Probes

*“Legacy application may need longer to start. Hold off on probing.”*



# Defining a Startup Probe

*TCP socket connection if exposed by application*

```
apiVersion: v1
kind: Pod
metadata:
  name: startup-pod
spec:
  containers:
    - image: httpd:2.4.46
      name: http-server
      startupProbe:
        tcpSocket:
          port: 80
        initialDelaySeconds: 3
        periodSeconds: 15
```

Tries to open a TCP socket  
connection to a port



---

# EXERCISE

Defining a Pod's  
Readiness and  
Liveness Probe



# Debugging YAML Manifests

*Not available during exam but helpful for practice*

Kube YAML  
Validating Kubernetes objects since 2018

⚠ Please only enter one YAML document at a time.

Example YAML

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: backend
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
    envFrom:
      - configMapRef:
          name: db-config
  resources: {}
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

unknown key: images

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: backend
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
    envFrom:
      - configMapRef:
          name: db-config
  resources: {}
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

1.38 !!    1.17 !!    1.16 !!    1.15 !!

Kube YAML  
Validating Kubernetes objects since 2018

⚠ Please only enter one YAML document at a time.

Example YAML

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: backend
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
    envFrom:
      - configMapRef:
          name: db-config
  resources: {}
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

No errors were found!

1.18 ✅    1.17 ✅    1.16 ✅    1.15 ✅



# Common Pod Error Statuses

Status	Root cause	Potential fix
ImagePullBackOff or ErrImagePull	Image could not be pulled from registry	Check correct image name, check that image name exists in registry, verify network access from node to registry, ensure proper
CrashLoopBackOff	Application or command run in container crashes	Check command executed in container, ensure that image can properly execute (e.g. by creating a container with Docker)
CreateContainerConfigError	ConfigMap or Secret references by container cannot be found	Check correct name of the configuration object, verify the existence of the configuration object in the namespace



# Debugging Existing Pods

*It's crucial to know how to debug and fix errors*

```
$ kubectl get all
```

“What’s running on a high-level?” *Pod xyz shows failure.*

```
$ kubectl describe pod xyz
```

“What exactly is the issue?” *Event shows CrashLoopBackOff.*

```
$ kubectl logs xyz
```

“Does an output indicate root cause?” *Misconfiguration in image.*

```
$ kubectl exec xyz -it --  
/bin/sh
```

“Does config look correct?” *Directory is missing.*



---

# EXERCISE

Fixing a  
Misconfigured Pod



---

# Q & A



---

# BREAK



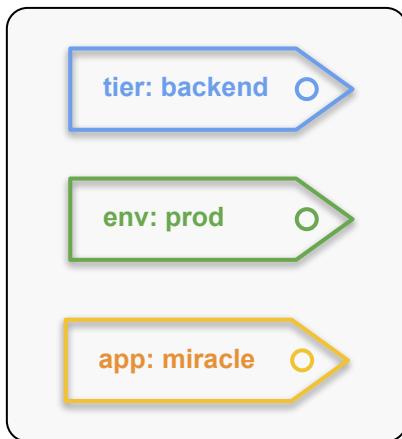
# Pod Design

Labels & Annotations, Deployments, Jobs and  
CronJobs

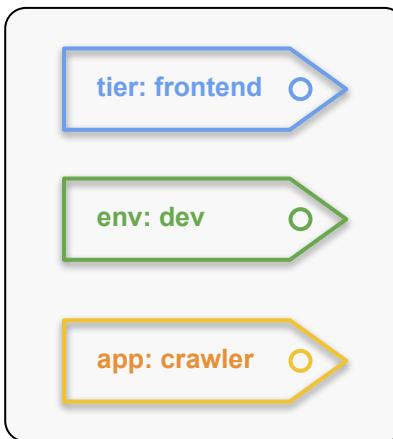
# Purpose of Labels

*Essential to querying, filtering and sorting Kubernetes objects*

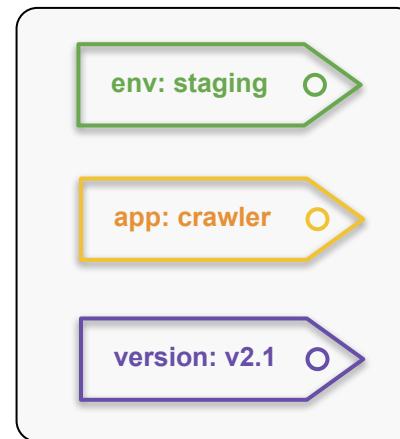
**Pod 1**



**Pod 2**



**Pod 3**



# Assigning Labels

*Defined in the metadata section of a Kubernetes object definition*

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    tier: backend
    env: prod
    app: miracle
spec:
  ...
```

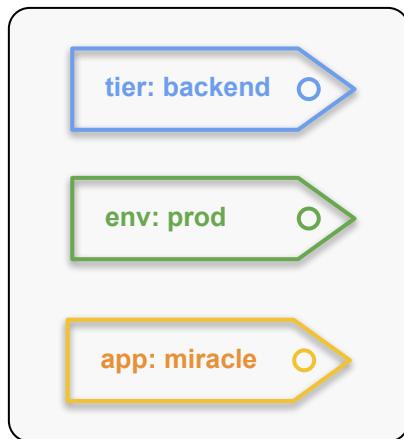
```
$ kubectl label pod nginx tier=backend
env=prod app=miracle
$ kubectl get pods --show-labels
NAME    ... LABELS
pod1   ... tier=backend,env=prod,app=miracle
```



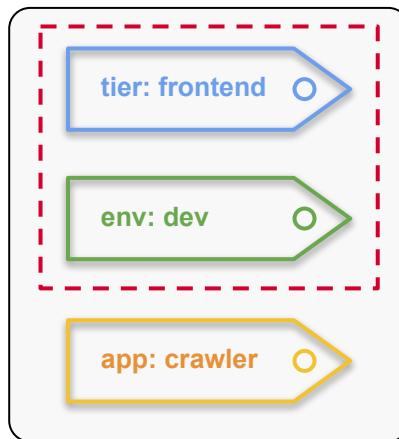
# Selecting Labels

*Querying objects from the CLI or via spec.selector*

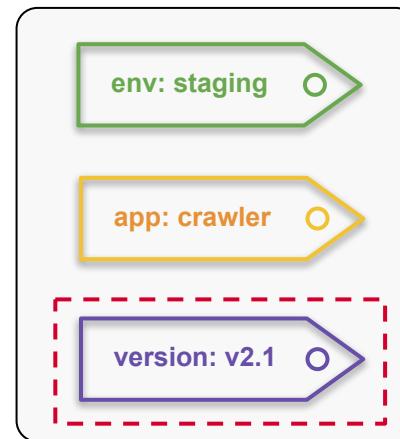
**Pod 1**



**Pod 2**



**Pod 3**



# Querying by CLI

*You can specify equality-based and set-based requirements*

```
# Tier is "frontend" AND is "development" environment
$ kubectl get pods -l tier=frontend,env=dev --show-labels
NAME    ... LABELS
pod2    ... app=crawler,env=dev,tier=frontend

# Has the label with key "version"
$ kubectl get pods -l version --show-labels
NAME    ... LABELS
pod3    ... app=crawler,env=staging,version=v2.1

# Tier is in set "frontend" or "backend" AND is "development" environment
$ kubectl get pods -l 'tier in (frontend,backend),env=dev' --show-labels
NAME    ... LABELS
pod2    ... app=crawler,env=dev,tier=frontend
```



# Selecting Resources in YAML

*Grouping resources by label selectors*

## Equality-based

```
apiVersion: v1
kind: Service
metadata:
  name: app-service
  ...
spec:
  ...
  selector:
    tier: frontend
    env: dev
```

## Equality- and set-based

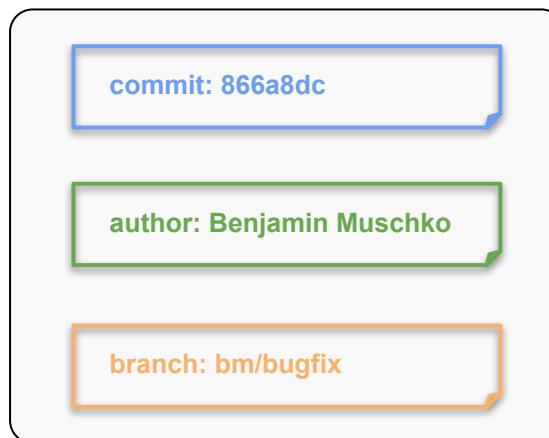
```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  ...
  selector:
    matchLabels:
      version: v2.1
    matchExpressions:
      - {key: tier, operator: In, values: [frontend,backend]}
```



# Purpose of Annotations

*Descriptive metadata without the ability to be queryable*

**Pod**



# Assigning Annotations

*Defined in the `metadata` section of a Kubernetes object definition*

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  annotations:
    commit: 866a8dc
    author: 'Benjamin Muschko'
    branch: 'bm/bugfix'
spec:
  ...
  ...
```

```
$ kubectl annotate pod nginx
commit='866a8dc' author='Benjamin Muschko'
branch='bm/bugfix'
$ kubectl describe pods my-pod
Name:           my-pod
Namespace:      default
...
Annotations:  author: Benjamin Muschko
                  branch: bm/bugfix
                  commit: 866a8dc
...
...
```



---

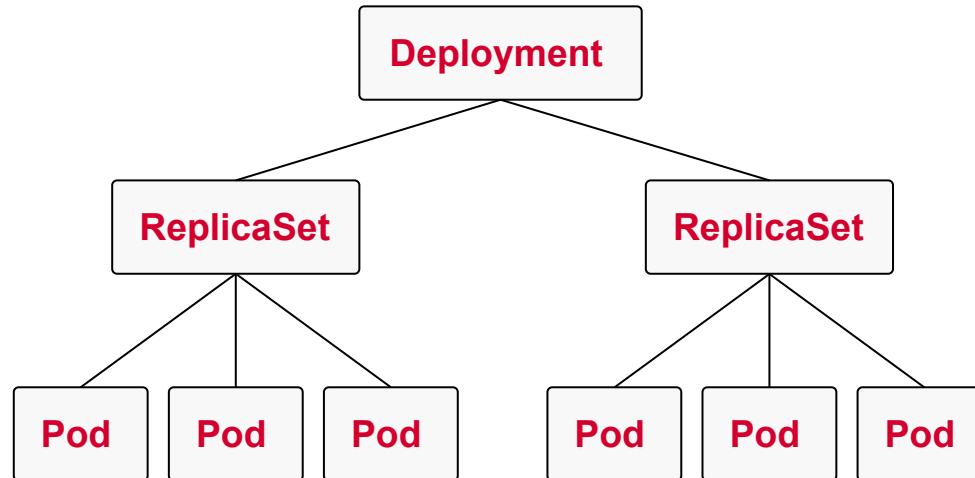
# EXERCISE

Defining and  
Querying Labels  
and Annotations



# Understanding Deployments

*Scaling and replication features for a set of Pods*



# Creating a Deployment

*The `create` command doesn't support replicas option*

```
$ kubectl create deployment my-deploy --image=nginx←  
--dry-run=client -o yaml > deploy.yaml  
$ vim deploy.yaml  
$ kubectl create -f deploy.yaml  
deployment.apps/my-deploy created
```



# Creating a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-deploy
    name: my-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-deploy
  template:
    metadata:
      labels:
        app: my-deploy
    spec:
      containers:
        - image: nginx
          name: nginx
```

The number of Pods running a specific set of containers

Selects the Pods for this deployment

The labels of the Pods



# Inspecting Deployment State

*Indicator between desired state and actual state*

```
$ kubectl get deployments  
NAME                      READY   UP-TO-DATE   AVAILABLE   AGE  
deployment.apps/my-deploy  3        3           3           25m
```



# Underlying Replication Feature

*Automatically created by Deployment, not meant to be modified*

```
$ kubectl get replicsets
NAME             DESIRED   CURRENT   READY   AGE
my-deploy-7786f96d67   3         3         3       6h

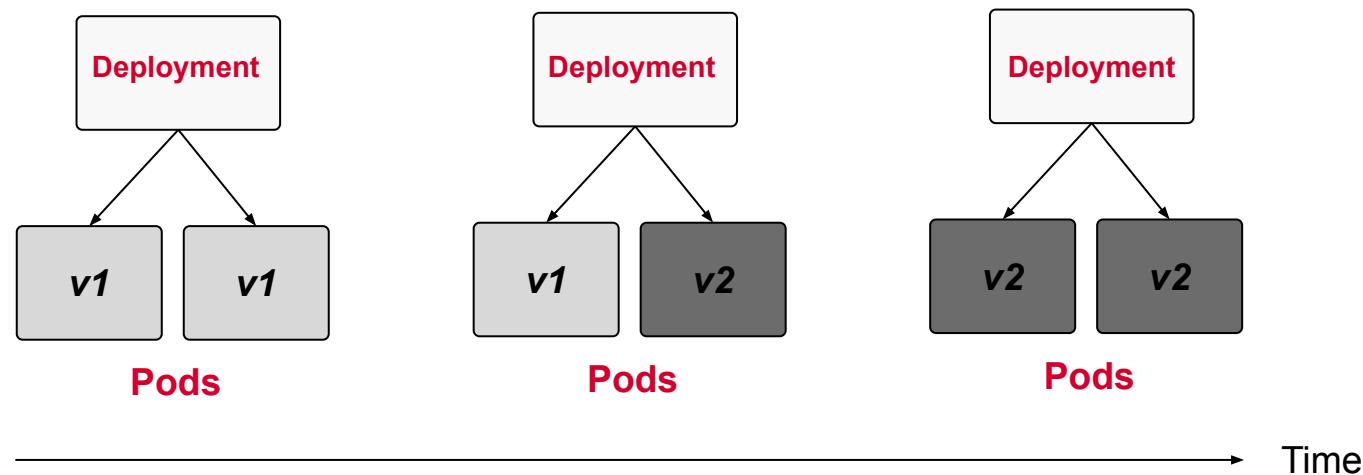
$ kubectl describe deploy my-deploy
...
OldReplicaSets:  <none>
NewReplicaSet:   my-deploy-7786f96d67 (3/3 replicas created)
...

$ kubectl describe replicsets my-deploy-7786f96d67
...
Controlled By:  Deployment/my-deploy
...
```



# Rolling Updates

*“Look ma, shiny new features. Let’s deploy them to production!”*



# Rollout Revision Log

```
# Check initial deployment revisions
$ kubectl rollout history deployments my-deploy
deployment.extensions/my-deploy
REVISION  CHANGE-CAUSE
1          <none>

# Make a change to the deployment
$ kubectl edit deployments my-deploy

# Revision history indicates changed version
$ kubectl rollout history deployments my-deploy
deployment.extensions/my-deploy
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```



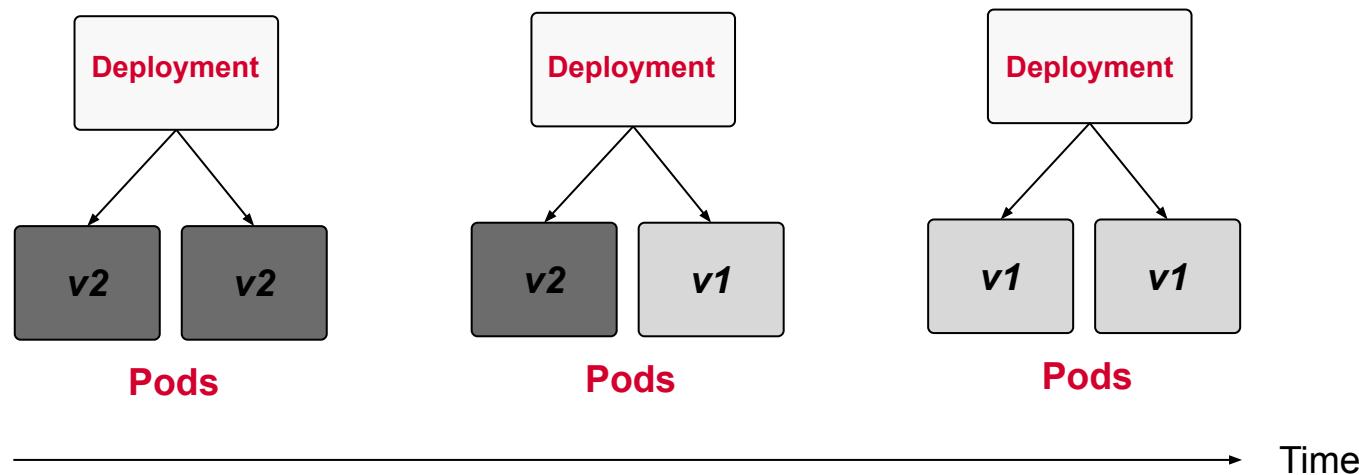
# Rendering Revision Details

```
$ kubectl rollout history deployments my-deploy --revision=2
deployment.extensions/my-deploy with revision #2
Pod Template:
  Labels:    app=my-deploy
             pod-template-hash=1365642048
  Containers:
    nginx:
      Image:  nginx:latest
      Port:    <none>
      Host Port: <none>
      Environment: <none>
      Mounts:   <none>
      Volumes:  <none>
```



# Rolling Back

*“Bug in the application. Let’s revert to the previous version!”*



# Rolling Back to a Revision

```
# Roll back to previous revision
$ kubectl rollout undo deployments my-deploy
deployment.extensions/my-deploy

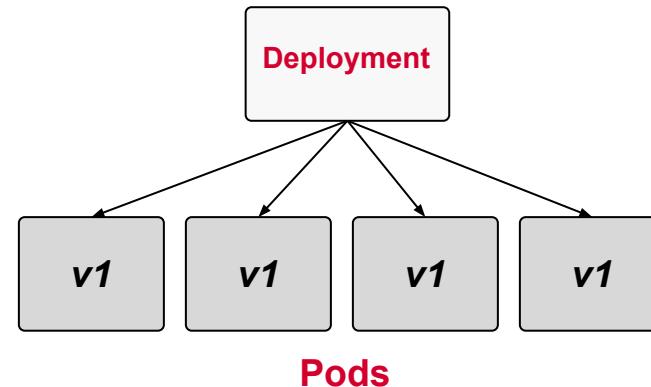
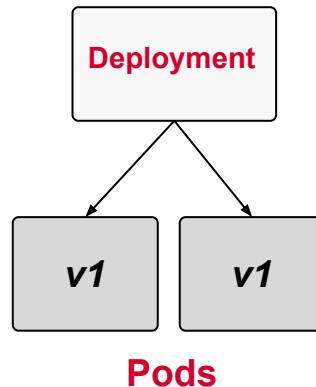
# Check rollout status
$ kubectl rollout status deployments my-deploy
deployment "my-deploy" successfully rolled out

# Revision history indicates changed version
$ kubectl rollout history deployments my-deploy
deployment.extensions/my-deploy
REVISION  CHANGE-CAUSE
2          <none>
3          <none>
```



# Manually Scaling a Deployment

*“Load is increasing. We need to scale up the application.”*



# Providing a Specific # of Replicas

```
# Check current deployment replicas
$ kubectl get deployments my-deploy
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
my-deploy   2        2           2           9h

# Scaling from 2 to 4 replicas
$ kubectl scale deployments my-deploy --replicas=4
deployment.extensions/my-deploy scaled

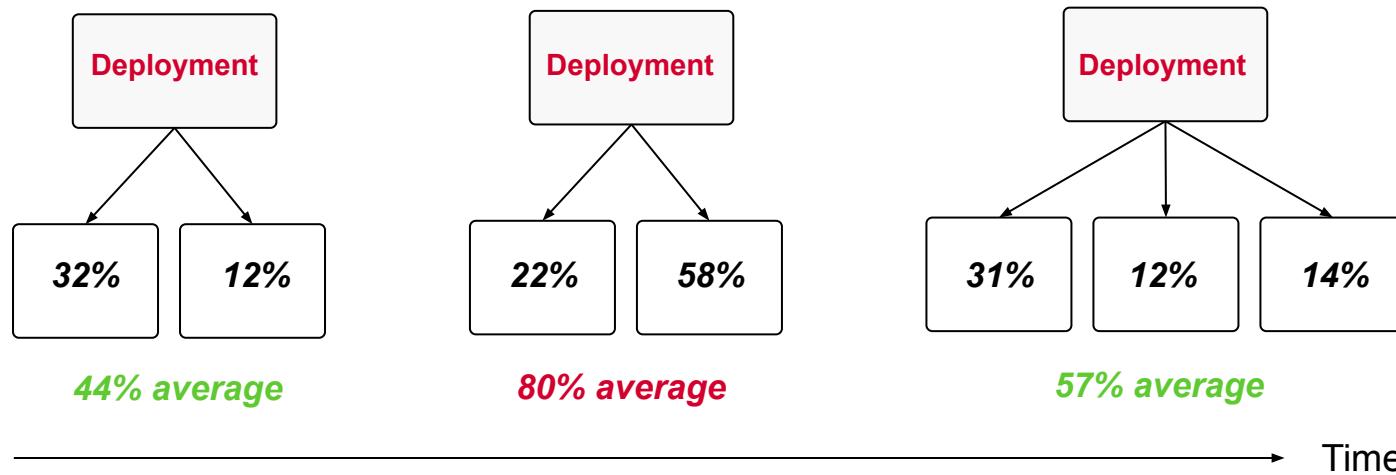
# Check the changed deployment replicas
$ kubectl get deployments my-deploy
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
my-deploy   4        4           4           9h
```



# Autoscaling a Deployment

*“Don’t make me think. Autoscale based on CPU utilization.”*

maximum, average CPU utilization: 70%



# Create Horizontal Pod Autoscaler

```
# Maintain average CPU utilization across all Pods of 70%
$ kubectl autoscale deployments my-deploy --cpu-percent=70 ←
  --min=1 --max=10
horizontalpodautoscaler.autoscaling/my-deploy autoscaled

# Check the current status of autoscaler
$ kubectl get hpa my-deploy
NAME          REFERENCE           TARGETS           MINPODS   ←
MAXPODS      REPLICAS      AGE
my-deploy    Deployment/my-deploy  0%/70%          1         ←
10           4                  23s
```



---

# EXERCISE

Performing Rolling  
Updates and Scaling  
a Deployment



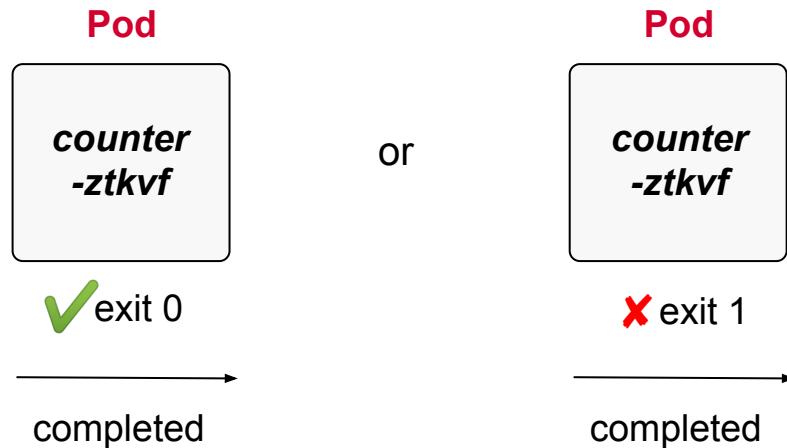
# Pods vs. Jobs vs. CronJobs

*Infinite vs. one-time vs. periodic processes*



# Understanding Jobs

*Job is complete when specific number of completions is reached*



# Creating a Job (imperative)

*"Increment a counter and render its value on the terminal"*

```
$ kubectl create job counter --image=nginx -- /bin/sh -c  
'counter=0; while [ $counter -lt 3 ]; do  
    counter=$((counter+1)); echo "$counter"; sleep 3; done;'  
job.batch/counter created
```



# Creating a Job (declarative)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: counter
spec:
  completions: 1
  parallelism: 1
  backoffLimit: 6
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - args:
            - /bin/sh
            - -c
            - ...
          image: nginx
          name: counter
```

Define # of successful completions and whether task should be run in parallel

How many times do we try before Job is marked failed?

Restart Pod upon failure or start a new Pod



# Creating a Job (mixed approach)

*The `create` command does not provide parameters yet*

```
$ kubectl create job counter --image=nginx --dry-run=client
  -o yaml -- /bin/sh -c 'counter=0; while [ $counter -lt 3 ]; do
    counter=$((counter+1)); echo "$counter"; sleep 3; done;' > job.yaml
$ vim job.yaml
$ kubectl create -f job.yaml
job.batch/counter created
```



# Different Types of Jobs

```
spec.completions: x  
spec.parallelism: y
```

Type	Completion criteria
Non-parallel	Complete as soon as its Pod terminates successfully
Parallel with fixed completion count	Complete when specified number of tasks finish successfully
Parallel with a work queue	Once at least one Pod has terminated with success and all Pods are terminated



# Inspecting Jobs

```
# List all jobs
$ kubectl get jobs
NAME      DESIRED   SUCCESSFUL   AGE
counter   1          1            3m

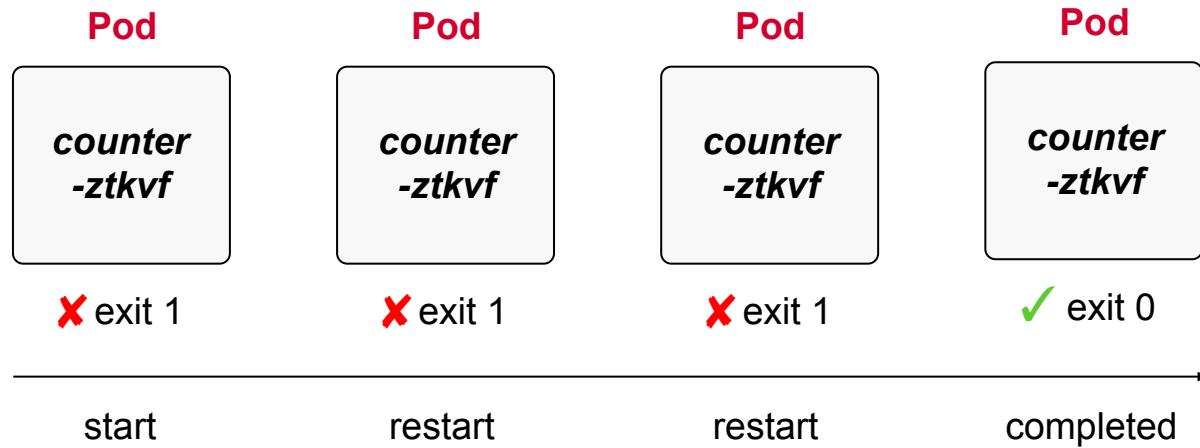
# Identify correlating Pods
$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
counter-9241c                 0/1     Completed   0          22m

# Get the logs of the Pod
$ kubectl logs counter-9241c
1
2
3
```



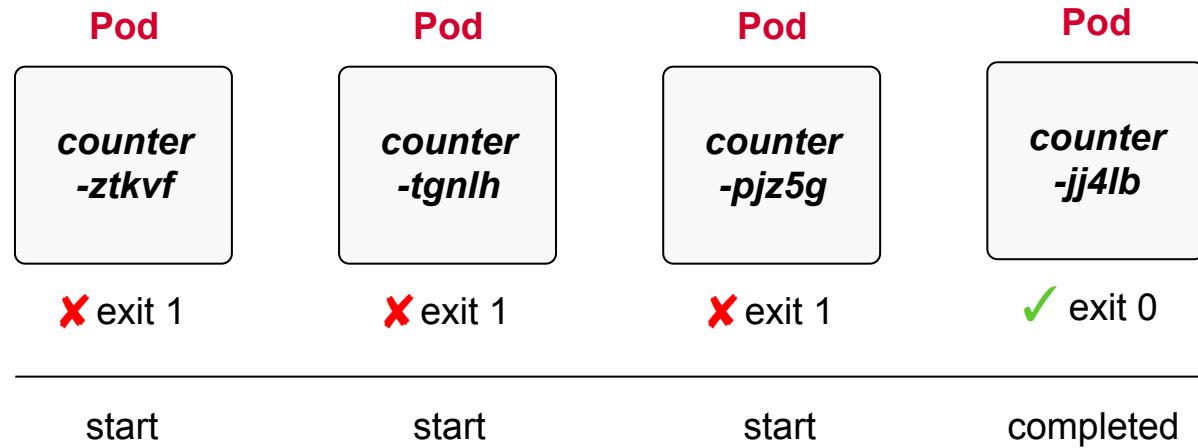
# Restarting a Container on Failure

`spec.template.spec.restartPolicy: OnFailure`



# New Pod on Failure

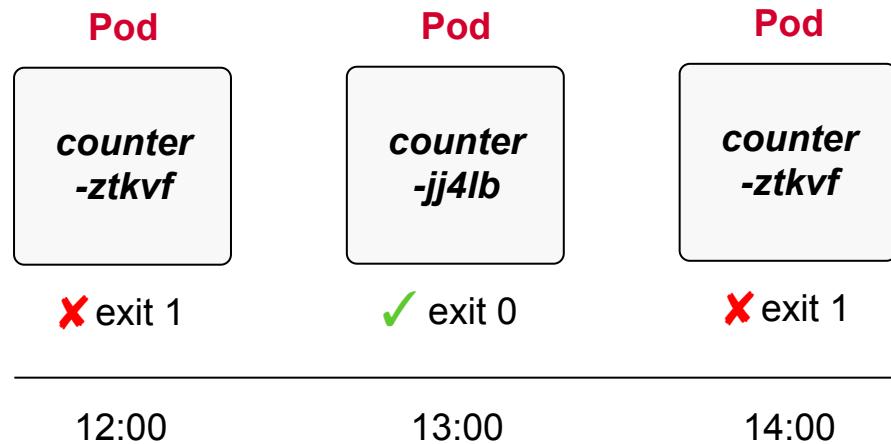
`spec.template.spec.restartPolicy: Never`



# Understanding CronJobs

*Similar to Job but task is run on a predefined schedule*

```
spec.schedule: "0 * * * *"
```



# Creating a CronJob (imperative)

*"Every hour increment a counter and render its value on the terminal"*

```
$ kubectl create cronjob counter --image=nginx<br/>--schedule="*/1 * * * *" -- /bin/sh -c 'counter=0;<br/>while [ $counter -lt 3 ]; do counter=$((counter+1));<br/>echo "$counter"; sleep 3; done;'<br/>job.batch/counter created
```



# Creating a CronJob (declarative)

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: counter
spec:
  schedule: "*/1 * * * *" ←
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: Never ←
          containers:
            - args:
                - /bin/sh
                - -c
                - ...
              image: nginx
              name: counter
```

The crontab expression used  
to run CronJob periodically

Run in a new Pod



# Inspecting CronJobs

```
# List all cron jobs
$ kubectl get cronjobs
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST SCHEDULE      AGE
counter   */1 * * * *    False        0           26s          1h

# Watch Pods executing the scheduled command
$ kubectl get jobs --watch
NAME              COMPLETIONS      DURATION      AGE
counter-1557334380 1/1            3s           2m24s
counter-1557334440 1/1            3s           84s
counter-1557334500 1/1            3s           24s
counter-1557334560 0/1            0s
counter-1557334560 0/1            0s
counter-1557334560 1/1            4s           4s
counter-1557334380 1/1            3s           3m10s
```



---

# EXERCISE

Creating a  
Scheduled  
Container Operation



---

# Q & A



---

# BREAK

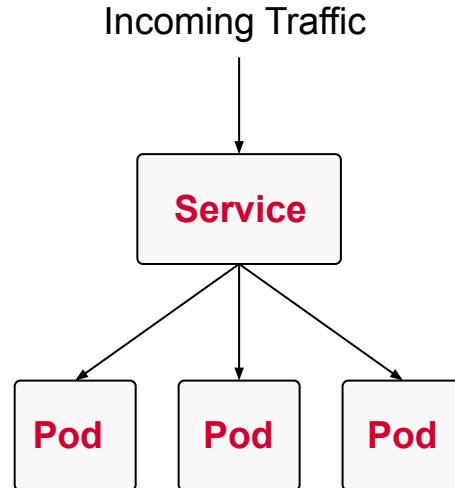


# **Services & Networking**

Services and Network Policies

# Understanding Services

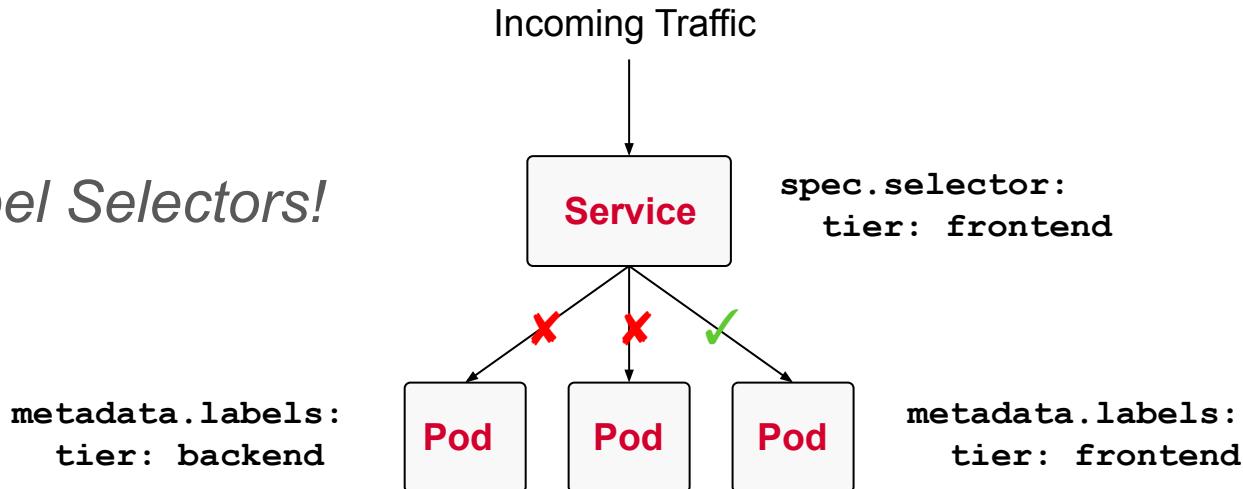
*Enables network access for a set of Pods*



# Request Routing

*“How does a service decide which Pod to forward the request to?”*

*Label Selectors!*



# Creating a Service (imperative)

*“Create a Service with explicit type”*

```
$ kubectl create service clusterip nginx --tcp=80:80  
service/nginx created
```



# Creating a Service (imperative)

*“Create a Pod and expose it with a Service”*

```
$ kubectl run nginx --image=nginx --port=80 --expose
service/nginx created
pod/nginx created
```



# Creating a Service (declarative)

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector: ←
    tier: frontend
  ports:
  - port: 3000 ←
    protocol: TCP
    targetPort: 80
  type: ClusterIP ←
```

Determines the Pod(s) for routing traffic

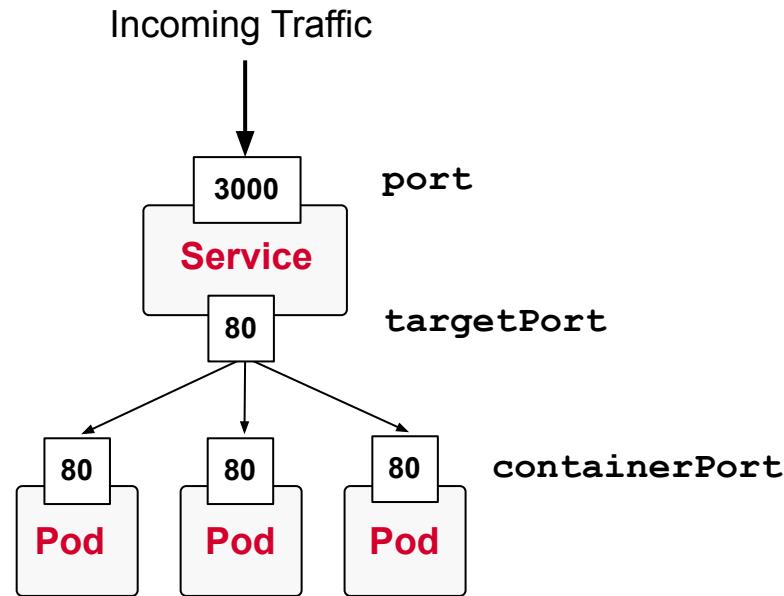
Maps incoming port to port of the Pod

Specifies how to expose the Service (inside/outside of cluster or LoadBalancer)



# Port Mapping

*“How to map the service port to the container port in Pod?”*



# Different Types of Services

Type	Behavior
ClusterIP	Exposes the service on a cluster-internal IP. Only reachable from within the cluster.
NodePort	Exposes the service on each node's IP at a static port. Accessible from outside of the cluster.
LoadBalancer	Exposes the service externally using a cloud provider's load balancer.
ExternalName	Map a Service to a DNS name.

`spec.type: xyz`



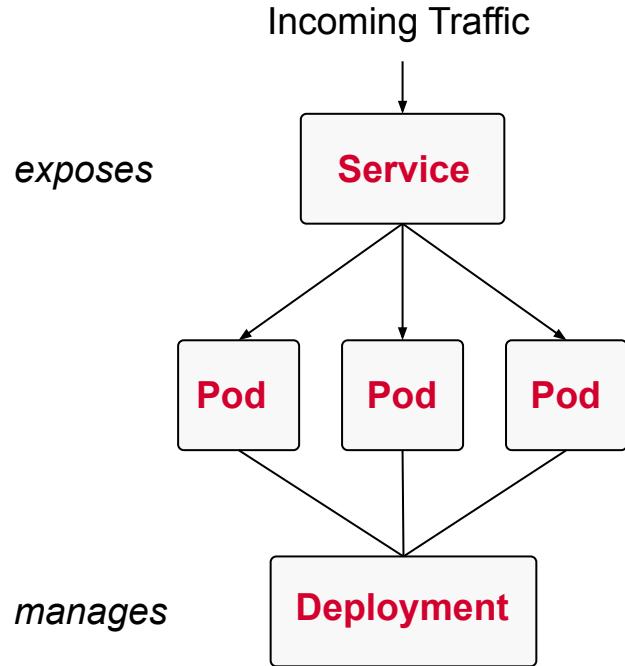
# Inspecting a Service

```
# Only reachable from within the cluster
$ kubectl get service nginx
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx    ClusterIP  10.105.201.83  <none>          80/TCP       3h

# Accessible from outside of the cluster
$ kubectl get service nginx
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx    NodePort   10.105.201.83  <none>          80:30184/TCP  3h
```



# Deployments and Services



*Two distinct concepts that complement each other*



---

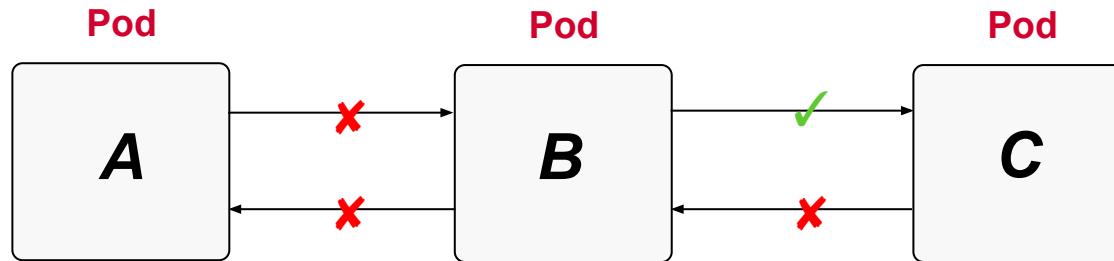
# EXERCISE

Routing Traffic to  
Pods from Inside  
and Outside of a  
Cluster



# Understanding Network Policies

*“Network Policies control traffic from and to the Pod”*

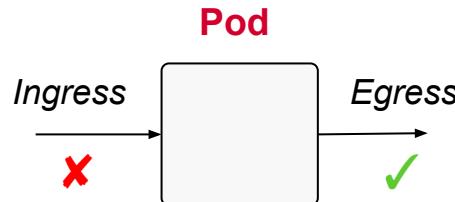


# Network Policy Rules



tier: frontend

*Which Pod does the rule apply to?* ✓



*Which direction of traffic?  
Who is allowed?*



# Creating a Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
spec:
  podSelector:
    matchLabels:
      tier: frontend
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        ...
  egress:
    - to:
        ...
```



*Label selection for Pods*

*Inbound/outbound traffic*

*Who can connect to Pod?  
Where can Pod connect to?*



# General Rule of Thumb

*Start by denying all access and allowing access as needed*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {} ← Applies to all Pods
  policyTypes:
    - Ingress ← Inbound/outbound traffic is blocked
    - Egress
```



# Behavior of from/to Selectors

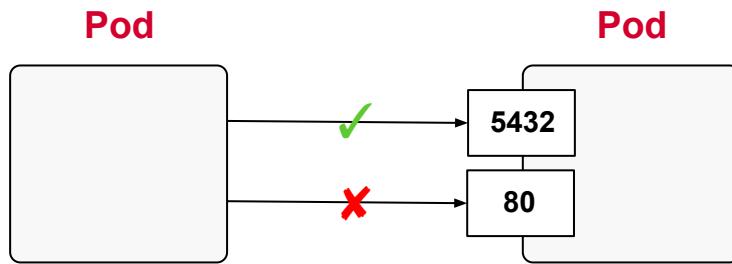
*Select by Namespace, Pod and IP address*

```
...
ingress:
- from:
  - podSelector:
    matchLabels:
      tier: backend
...
...
```

*Allow incoming traffic from  
Pod that matches the  
**label** tier=backend*



# Restricting Access to Ports



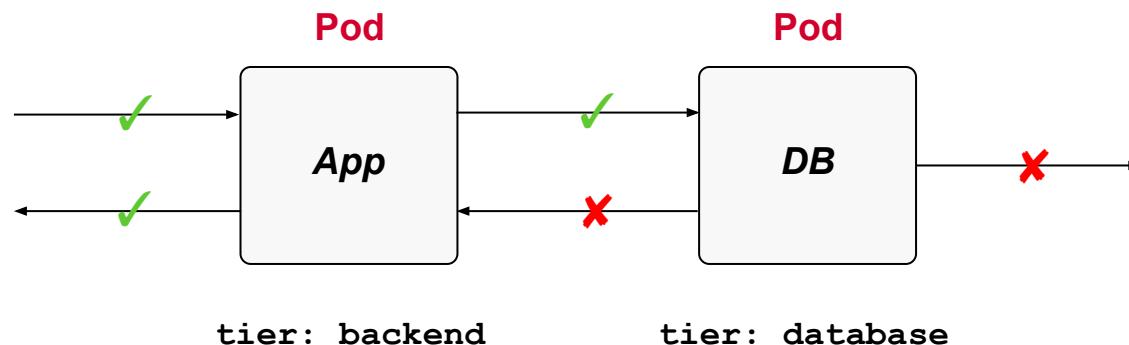
*By default all ports are open*

```
...  
ingress:  
- from:  
  - podSelector:  
    matchLabels:  
      tier: backend  
ports:  
- protocol: TCP  
  port: 5432  
...
```



# Representative Use Case

*“Application makes request to database but database cannot make any outgoing requests.”*



---

# EXERCISE

Restricting Access  
to and from a Pod



---

# Q & A



---

# BREAK

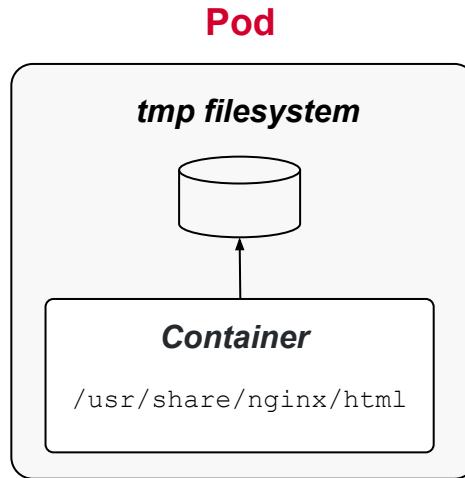


# **State Persistence**

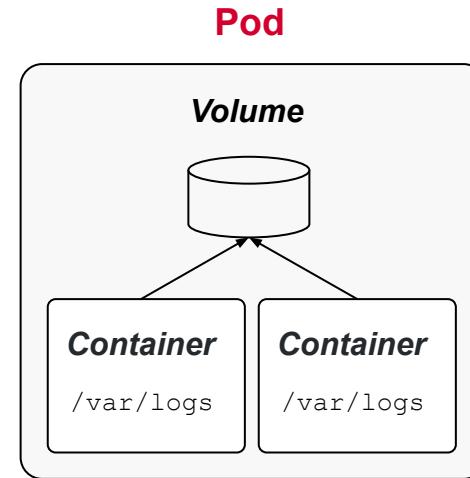
Persistent Volumes and Claims

# Understanding Volumes

*Persist data that outlives a container restart*



vs.



# Types of Volumes

Type	Description
emptyDir	Empty directory in Pod. Only persisted for the lifespan of a Pod.
hostPath	File or directory from the host node's filesystem into your Pod.
configMap, secret	Provides a way to inject configuration data and secrets into Pods.
nfs	An existing NFS (Network File System) share to be mounted into your Pod. Preserves data after Pod restart.
Cloud provider solutions	Provider-specific implementation for AWS, GCE or Azure.



# Creating a Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: my-container
spec:
  volumes: ←
    - name: logs-volume
      emptyDir: {}
  containers:
    - image: nginx
      name: my-container
      volumeMounts: ←
        - mountPath: /var/logs
          name: logs-volume
```

*Define Volume with a type*

*Mount Volume to a path*



# Using a Volume

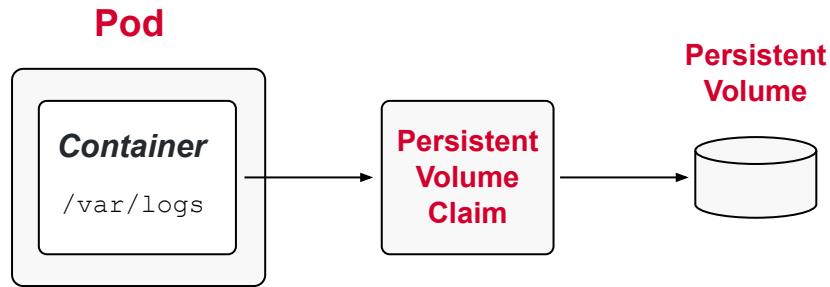
```
# Create Pod with mounted Volume
$ kubectl create -f pod-with-vol.yaml
pod/my-container created

# Shell into container and use Volume
$ kubectl exec -it my-container -- /bin/sh
# cd /var/logs
# pwd
/var/logs
# touch app-logs.txt
# ls
app-logs.txt
```



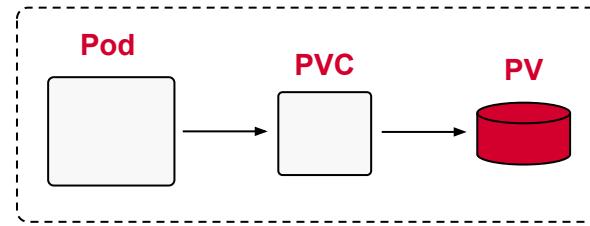
# Understanding PersistentVolumes

*Persist data that outlives a container, Pod or node restart*



# Creating a PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv
spec:
  capacity: ←
    storage: 512m
  accessModes: ←
    - ReadWriteOnce
  storageClassName: shared
  hostPath:
    path: /data/config
```



*Defines a specific storage capacity*

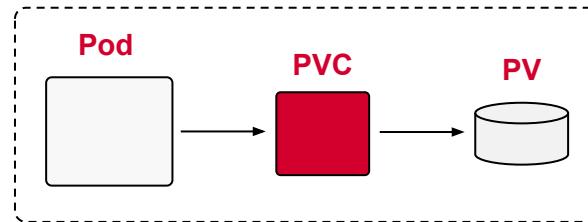
*Read and/or write access*

*How many nodes can access volume?*



# Creating a Claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 256m
  storageClassName: shared
```



*Read and/or write access*

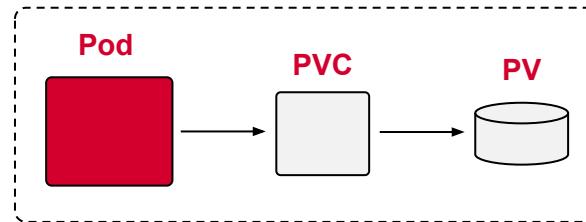
*How many nodes can access volume?*

*Defines a specific storage capacity*



# Mounting a Claim

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: app
spec:
  volumes:
    - name: configpvc
      persistentVolumeClaim:
        claimName: pvc
  containers:
    - image: nginx
      name: app
      volumeMounts:
        - mountPath: "/data/app/config"
          name: configpvc
```



*References the Volume with the  
claim name*

*Mounts Volume to path*



---

# EXERCISE

Defining and  
Mounting a  
PersistentVolume



---

# Q & A



# Summary & Wrap Up

Last words of advice...

---

# Gaining confidence

- Run through practice exams as often as you can
- Read through online documentation start to end
- Know your tools (especially vim, bash, YAML)
- Pick time you are most comfortable, get enough sleep
- Take your first attempt easy but give it your best



---

# Q & A





Thank you

