

Using Kubernetes Documentation

You are allowed to open documentation pages during the exam (see [FAQ](#))

- Docs: <https://kubernetes.io/docs>
- GitHub: <https://github.com/kubernetes>
- Blog: <https://kubernetes.io/blog>

Using the Alias for kubectl

Preconfigured in exam environment

```
$ alias k=kubectl
```

```
$ k version
```

```
...
```

Create an alias for the `kubectl` command line tool in your local environment

Use the shortcut to refer to `kubectl`

Using Auto-Completion for kubectl

Preconfigured in exam environment

```
$ kubectl cre<tab>
```

```
$ kubectl create
```



<https://kubernetes.io/docs/reference/kubectl/cheatsheet/#kubectl-autocomplete>

Working in a Context and Namespace

It's required to solve problems in the correct cluster and namespace

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

```
$ kubectl config use-context <context-of-question>
```

Switches to context
in kubeconfig file

Sets a context entry
for the namespace in
the kubeconfig file

Application Design and Build





Image and Container Management

Defining, building, and modifying container images

Container Terminology

The container and its runtime environment

Container

- Packages an application into a single unit of software including its runtime environment, application binary, dependencies, and configuration.

Container Runtime Engine

- A software component that can run containers on a host operating system.
- Example: [Docker Engine](#) or [containerd](#).

Container Orchestrator

- Uses a container runtime engine to instantiate a container.
- Automates and manages workload with features like scalability, networking.
- Example: [Kubernetes](#) or [Nomad](#)

Container Terminology

Building, running, and publishing a container image

Containerfile

- Spells out what needs to happen when the software is built in the form of a list of instructions.
- Synonymous with Dockerfile.

Container Image

- Packages an application into a single unit of software including its runtime environment and configuration.
- Built from the Containerfile.

Container Registry

- Shares container images in a central location and makes them available for consumption.
- Example: [Docker Hub](#) or [GCR](#)

Containerization Process

Building and publishing an image, running image in container

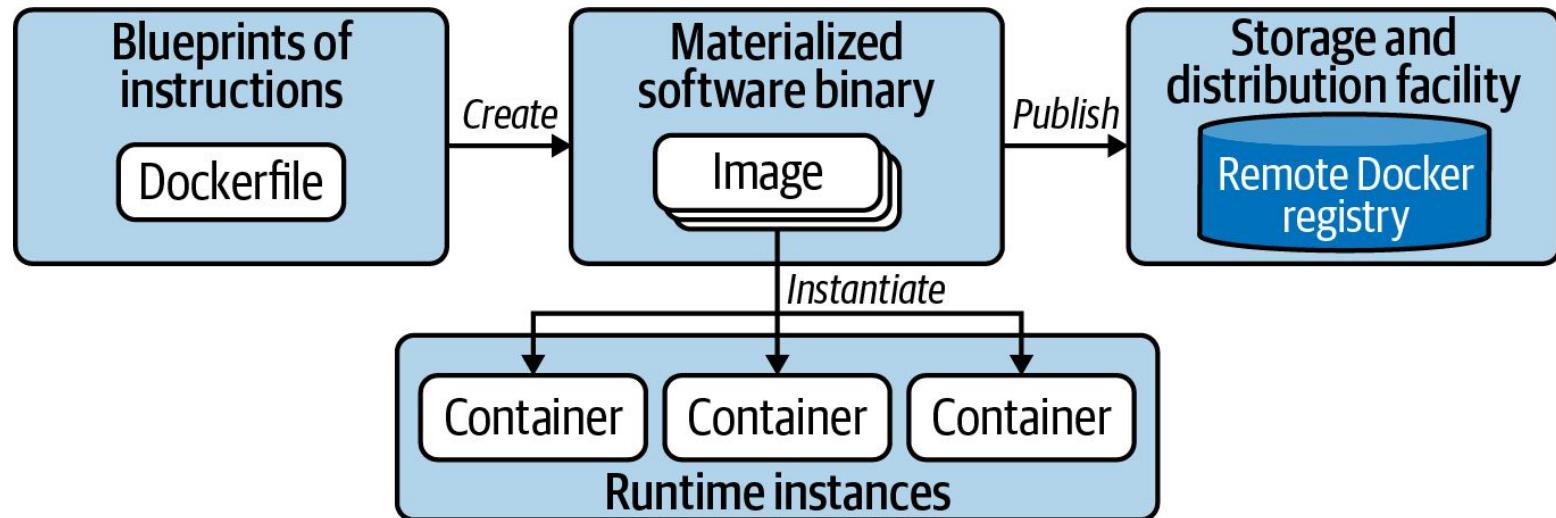




Image Building Instructions

A Dockerfile is a plain-text file with instructions

```
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/java-hello-world-0.0.1.jar java-hello-world.jar
ENTRYPOINT ["java", "-jar", "/app/java-hello-world.jar"]
EXPOSE 8080
```

Building the Container Image

Provide name and tag of image + context directory

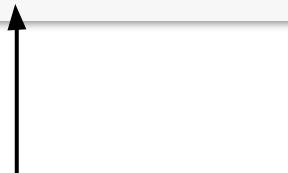
```
$ docker build -t java-hello-world:1.0.0 . ← Context directory
Sending build context to Docker daemon 8.32MB
Step 1/5 : FROM openjdk:11-jre-slim
--> 973c18dbf567
Step 2/5 : WORKDIR /app
--> Using cache
--> 31f9c5f2a019
...
Successfully built 3e9c22451a17 ← Container Image ID
Successfully tagged java-hello-world:1.0.0
```

Listing Container Images

Render all local images including their name + hash

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
java-hello-world	1.0.0	3e9c22451a17	About a minute ago	213MB
openjdk	11-jre-slim	973c18dbf567	20 hours ago	204MB



Base image is
downloaded automatically



Running the Container

You can interact with container once its running

```
$ docker run -d -p 8080:8080 java-hello-world:1.0.0  
b0ee04accf078ea7c...  
  
$ docker logs b0ee04accf078ea7c...  
Hello World!  
  
$ docker exec -it b0ee04accf078ea7c... bash  
root@b0ee04accf078ea7c:/# pwd  
/  
root@b0ee04accf078ea7c:/# exit
```

Tagging the Container Image

Name + tag need to conform to registry conventions

```
$ docker tag java-hello-world:1.0.0 bmuschko/java-hello-world:1.0.0
```

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	...
java-hello-world	1.0.0	3e9c22451a17	...
openjdk	11-jre-slim	973c18dbf567	...
bmuschko/java-hello-world	1.0.0	3e9c22451a17	...



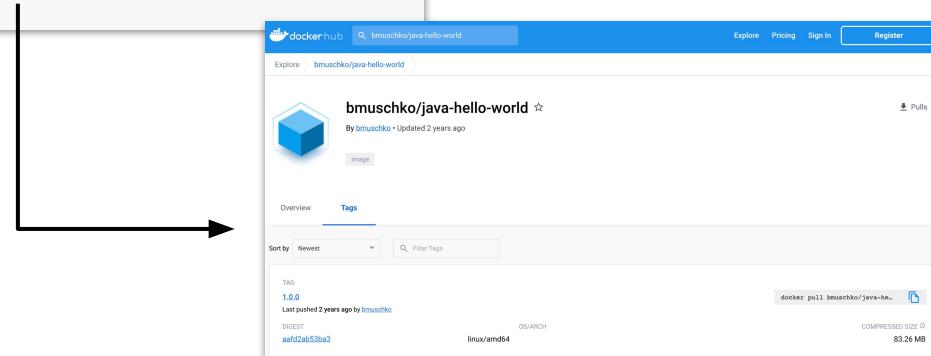
Tagged image



Publishing the Container Image

Authenticate against registry and push a tag

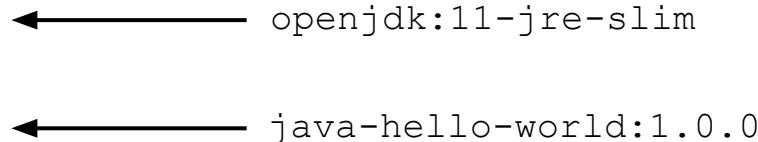
```
$ docker login --username=bmuschko  
$ docker push bmuschko/java-hello-world:1.0.0
```



Saving a Container Image

Produces a backup TAR archive file with all parent layers

```
$ docker save -o java-hello-world.tar java-hello-world:1.0.0  
  
$ ls  
java-hello-world.tar
```



Loading a Container Image

Restores image from TAR archive file

```
$ docker load --input java-hello-world.tar
```

```
Loaded image: java-hello-world:1.0.0
```

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	...
java-hello-world	1.0.0	3e9c22451a17	...
openjdk	11-jre-slim	973c18dbf567	...

Exercise

Defining, building, and
running a container image



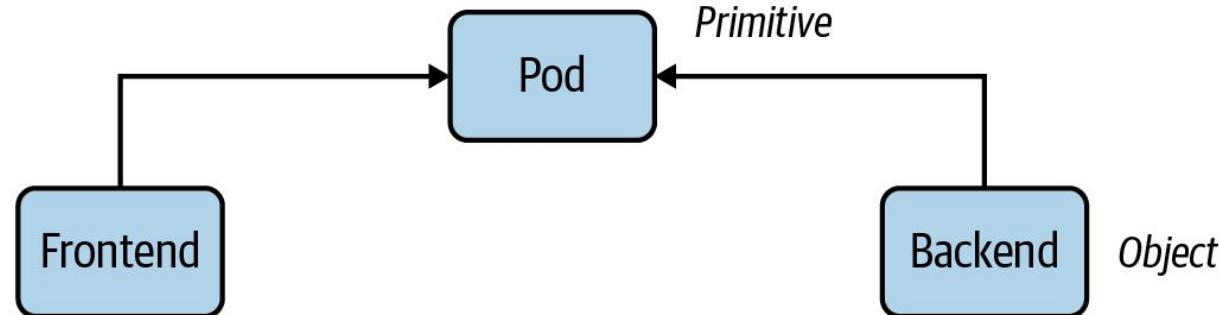


Object Management

Kubernetes primitives, interaction with kubectl, imperative vs. declarative approach

Kubernetes Primitives

Basic building blocks for creating and operating an application



UID: 4ef7b090-37ed-4b33-8fb7-c5693a48eef5

UID: b674f2e0-f0bc-40be-af8e-7985442c21a2

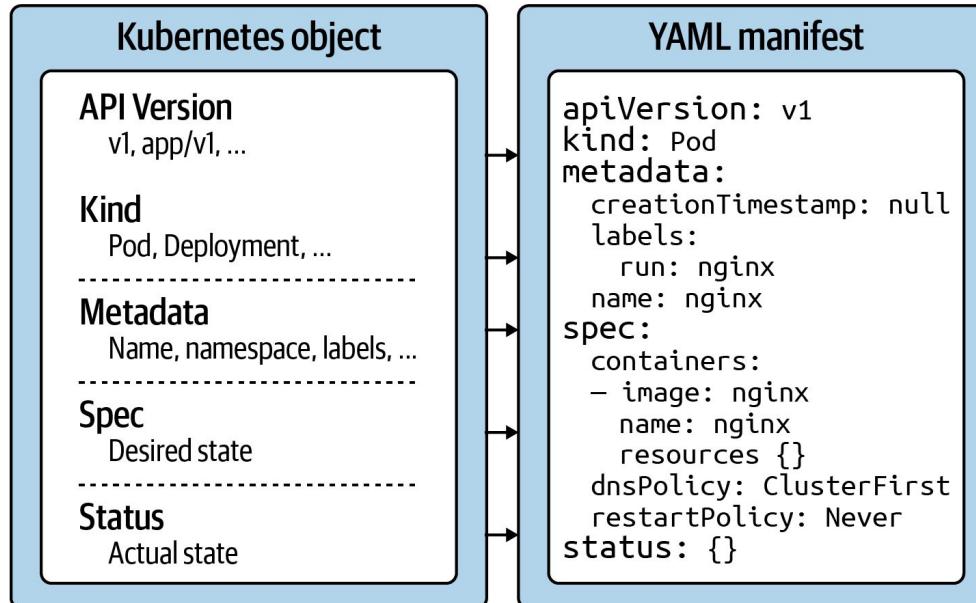
General Primitive Sections

Each section has a specific function

- *API version*: Defines the structure of a primitive and uses it to validate the correctness of the data.
- *Kind*: Defines the kind defines the type of primitive, e.g. a Pod or a Service.
- *Metadata*: Describes higher-level information about the object.
- *Spec*: Desired state of object.
- *Status*: Actual state of object. Empty if not created yet.

Kubernetes Object Structure

General structure followed by most primitives



Using kubectl

Primary tool to interact with the Kubernetes clusters from the CLI

```
kubectl [command] [TYPE] [NAME] [flags]  
        get      pod     app   -o yaml
```

Imperative Object Management

Fast but requires detailed knowledge, no track record

```
$ kubectl create namespace ckad  
namespace/ckad created
```

Creates an object
e.g. a namespace

```
$ kubectl run nginx --image=nginx -n ckad  
pod/nginx created
```

Creates a Pod object
in a namespace

```
$ kubectl edit pod nginx -n ckad  
pod/nginx edited
```

Modifies the live
object e.g. a Pod

Declarative Object Management

Suitable for more elaborate changes, tracks changes

```
$ kubectl create -f nginx-deployment.yaml  
deployment.apps/nginx-deployment created
```

Creates object if it
doesn't exist yet

```
$ kubectl apply -f nginx-deployment.yaml  
deployment.apps/nginx-deployment configured
```

Creates object or
update if it already
exists

```
$ kubectl delete -f nginx-deployment.yaml  
deployment.apps "nginx-deployment" deleted
```

Deletes object

Hybrid Approach

Generate YAML file with `kubectl` but make further edits

```
$ kubectl run nginx --image=nginx  
--dry-run=client -o yaml > nginx-pod.yaml
```

Capture YAML for Pod object in file

```
$ vim nginx-pod.yaml
```

Create the object from YAML file

```
$ kubectl create -f nginx-pod.yaml  
pod/nginx created
```



Pods and Namespaces

Running and inspecting workload, organizing objects

What is a Pod?

Runs a process or application in container(s)

- The smallest, most basic deployable objects in Kubernetes.
- It contains one or more containers, executed by the configured container runtime.
- Kubernetes enhances the basic functionality of a container with features such as security, storage, and more.

Creating a Pod with Imperative Command

"Execute Hazelcast and configure it using parameters"

```
$ kubectl run hazelcast  
  --image=hazelcast/hazelcast  
  --restart=Never  
  --port=5701  
  --env="DNS_DOMAIN=cluster"  
  --labels="app=hazelcast,env=prod"  
pod/hazelcast created
```

Pod YAML Manifest

"Execute Hazelcast and configure it using parameters"

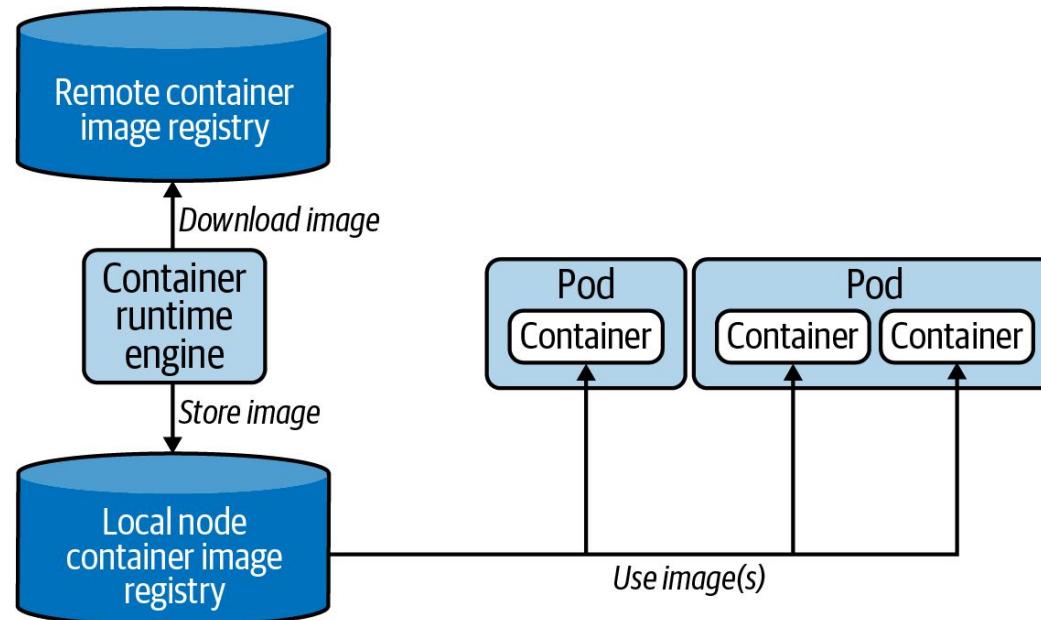
```
apiVersion: v1
kind: Pod
metadata:
  name: hazelcast
spec:
  containers:
    - image: hazelcast/hazelcast
      name: hazelcast
      ports:
        - containerPort: 5701
  restartPolicy: Never
```

← Container image

← Exposed container port(s)

Container Image Retrieval

Downloads container image from registry upon Pod creation



Listing Pods

Get high-level information on all Pods or a specific one

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hazelcast	1/1	Running	0	17s

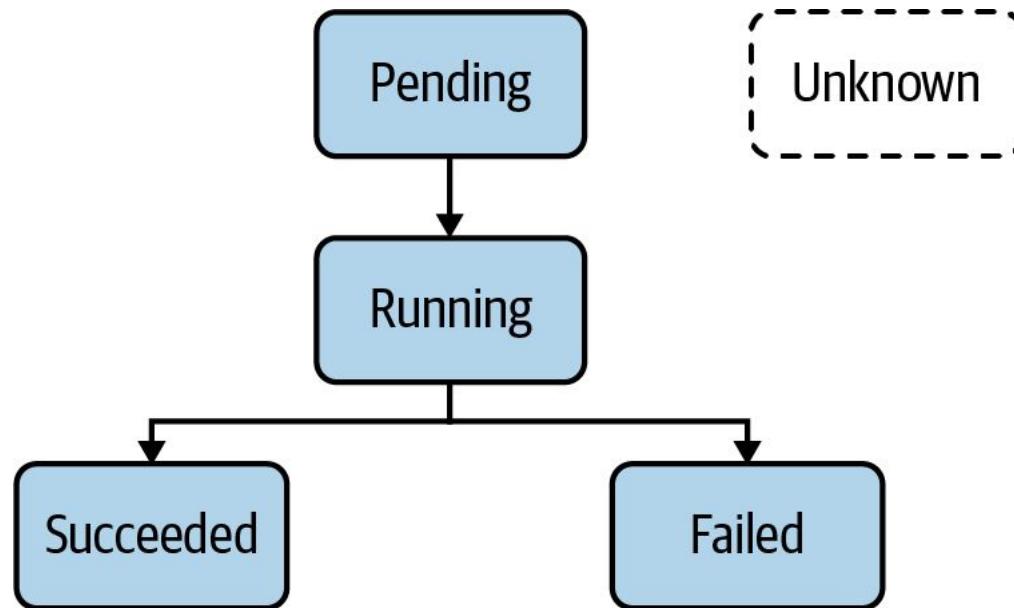
```
$ kubectl get pods hazelcast
```

NAME	READY	STATUS	RESTARTS	AGE
hazelcast	1/1	Running	0	17s

Specific Pod name

Pod Life Cycle Phases

Indicates operational status of Pod



Rendering Pod Details

Shows Pod configuration, events, and status

```
$ kubectl describe pods hazelcast
Name:           hazelcast
Namespace:      default
...
Status:         Running
IP:             10.1.0.41
Containers:
  ...
Events:
```

Virtual IP address assigned to
Pod to communicate with it

Accessing Container Logs

Renders logs produced by application or process running in a container

```
$ kubectl logs hazelcast
...
May 25, 2020 3:36:26 PM com.hazelcast.core.LifecycleService
INFO: [10.1.0.46]:5701 [dev] [4.0.1] [10.1.0.46]:5701 is STARTED
```

- You can stream the logs with the command line option `-f`. This option is helpful if you want to see logs in real time.
- You can still get back to the logs of the previous container by adding the `-p` command line option in case the container crashes or is restarted.

Executing a Command in a Container

Interactive exploration of a container, or seeing the direct effect of a command

```
$ kubectl exec -it hazelcast -- /bin/sh  
# ...  
  
$ kubectl exec hazelcast -- env  
...  
DNS_DOMAIN=cluster
```

- The command line option `-it` opens an interactive shell. This is useful if you want to inspect the configuration of your application or debug the current state of your application.
- For direct execution of a command in the container simply append to `--`.

Creating a Temporary Pod

Meant for experimentation, deleted after use

```
$ kubectl run busybox  
  --image=busybox  
  --rm  
  -it  
  --restart=Never  
  -- wget 10.1.0.41  
Connecting to 10.1.0.41:80 (10.1.0.41:80)  
...  
pod "busybox" deleted
```

Removes Pod after executing
the provided command

Declaring Environment Variables

Define zero to many key-value pairs

```
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: prod
        - name: VERSION
          value: '1.5.3'
```

Typical naming conventions
for keys are not enforced

Declaring Command and Args

Assign or overwrite container entrypoint and arguments

```
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-app
spec:
  containers:
    - image: bmuschko/spring-boot-app:1.5.3
      name: spring-boot-app
      command: ["/bin/sh"]
      args: ["-c", "while true; do date; sleep 10; done"]
```

Deleting a Pod

Graceful deletion of workload can take up to 30 seconds

```
$ kubectl delete pod hazelcast  
pod "hazelcast" deleted
```

← Imperative approach

```
$ kubectl delete -f pod.yaml  
pod "hazelcast" deleted
```

← Declarative approach

```
$ kubectl delete -f pod.yaml --now  
pod "hazelcast" deleted
```

← Force-kills Pod without
graceful deletion of workload

What is a Namespace?

Organizes and isolates Kubernetes object that belong together

- An API construct to avoid naming collisions and represent a scope for object names. A good use case for namespaces is to isolate the objects by team or responsibility.
- The `default` namespace hosts object that haven't been assigned to an explicit namespace.
- Namespaces starting with the prefix `kube-` are not considered end user-namespaces. They have been created by the Kubernetes system. You will not have to interact with them as an application developer.
- Reference a namespace with the command line option `--namespace` or `-n`.



Creating a Namespace with Imperative Command

"Organize objects in namespace called code-red"

```
$ kubectl create namespace code-red
namespace/code-red created

$ kubectl run nginx --image=nginx --restart=Never -n code-red
pod/nginx created

$ kubectl get pods -n code-red
NAME      READY     STATUS    RESTARTS   AGE
nginx    1/1      Running   0          13s
```

Listing Namespaces

Includes default, Kubernetes-internal, and custom namespaces

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	157d
kube-node-lease	Active	157d
kube-public	Active	157d
kube-system	Active	157d

Namespace YAML Manifest

"Organize objects in namespace called code-red"

```
apiVersion: v1
kind: Namespace
metadata:
  name: code-red
```



Deleting a Namespace

Cascade-deletes containing objects

```
$ kubectl delete namespace code-red
namespace "code-red" deleted

$ kubectl get pods -n code-red
No resources found in code-red namespace.
```



Exercise

Creating, inspecting, and interacting with a Pod in a Namespace





Jobs and CronJobs

One-time operations and scheduled operations

Job and CronJob Terminology

Kubernetes primitives for specific use cases

Pod

- *Continuous* operation of an application kept running without interrupts if possible.
- Example: An application exposing a RESTful API for user management.

Job

- Runs functionality until a specified number of completions has been reached, making it a good fit for *one-time* operations.
- Example: Import/export data processes or I/O-intensive processes with a finite end.

CronJob

- Essentially a Job, but it's run *periodically* based a schedule.
- Example: Running a database backup

What is a Job?

Kubernetes primitive for executing a one-time operation

- It runs functionality until a specified number of completions has been reached.
- The actual work managed by a Job is still running inside of a Pod. Therefore, you can think of a Job as a higher-level coordination instance for Pods executing the workload.
- Upon completion of a Job and its Pods, Kubernetes does not automatically delete the objects - they will stay until they're explicitly deleted. Keeping those objects helps with debugging the command run inside of the Pod and gives you a chance to inspect the logs.

Creating a Job with Imperative Command

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

```
$ kubectl create job<br>  counter<br>  --image=nginx:1.24.0<br>  -- /bin/sh -c 'counter=0; while [ $counter -lt 3 ]; do<br>    counter=$((counter+1)); echo "$counter"; sleep 3; done;'<br>job.batch/counter created
```

Job YAML Manifest

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

```
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:1.24.0
          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

Inspecting a Job

Job is completed when configured # of executions has been reached

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
counter	0/1	13s	13s

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
counter	1/1	15s	19s

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
counter-z6kdj	0/1	Completed	0	51s



Inspecting Pod Logs

Workload can be inspected during and after execution

```
$ kubectl logs counter-z6kdj  
1  
2  
3
```

Job Operation Types

Execution runtime behavior of workload can be fine-tuned

- The default behavior of a Job is to run the workload in a single Pod and expect one successful completion (non-parallel Job).
- The attribute `spec.completions` controls the number of required successful completion.
- The attribute `spec.parallelism` allows for executing the workload by multiple pods in parallel.

Job Operation Types

Pick combination of completions and parallelism based on workload use case

Type	Completions	Parallelism	Description
Non-parallel with one completion count	1	1	Completes as soon as its Pod terminates successfully.
Parallel with a fixed completion count	≥ 1	≥ 1	Completes when specified number of tasks finish successfully.
Parallel with worker queue	unset	≥ 1	Completes when at least one Pod has terminated successfully and all Pods are terminated.

Restart Behavior

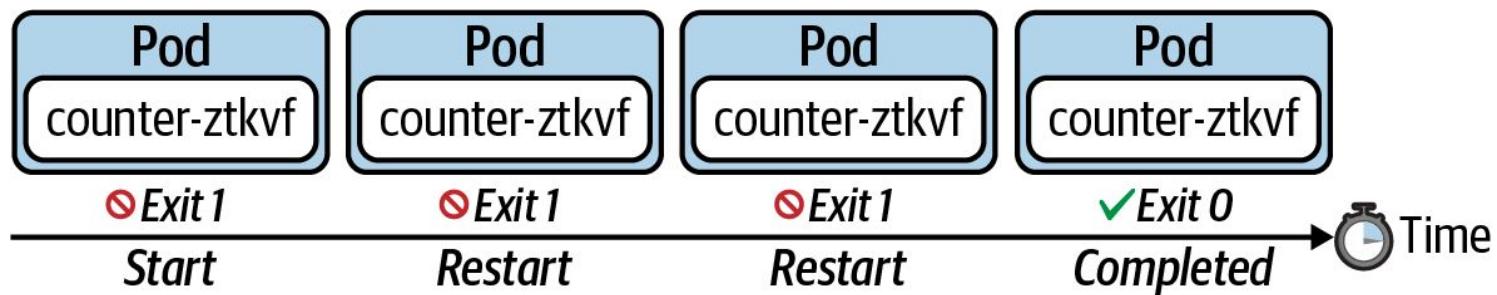
Re-running workload if execution is unsuccessful

- The attribute `spec.backoffLimit` determines the number of retries a Job attempts to successfully complete the workload until the executed command finishes with an exit code 0. The default is 6, which means it will execute the workload 6 times before the Job is considered unsuccessful.
- The attribute `spec.template.spec.restartPolicy` needs to be declared explicitly. The default restart policy of a Pod is `Always`, which tells the Kubernetes scheduler to always restart the Pod even if the container exits with a zero exit code. The restart policy of a Job can only be `OnFailure` or `Never`.

Restarting a Container on Failure

Restart container exits with a non-zero exit code

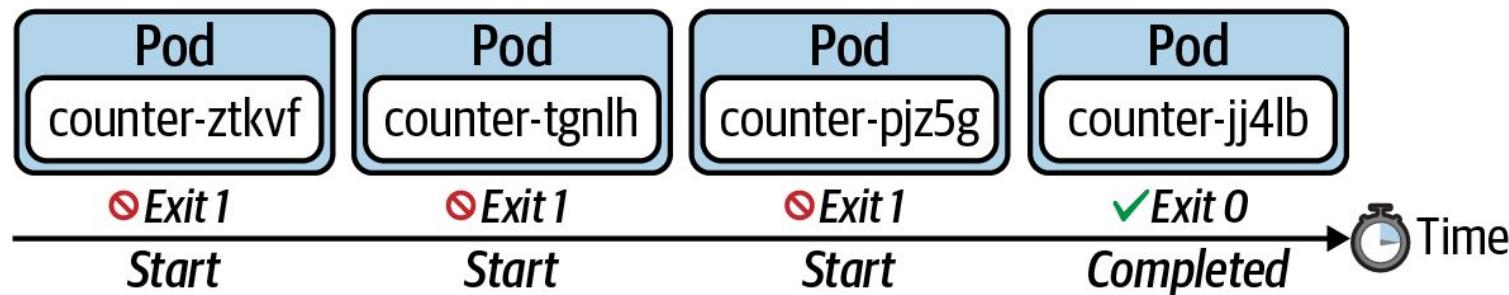
```
spec.template.spec.restartPolicy: OnFailure
```



Starting a New Pod on Failure

The policy does not restart the container upon a failure, it starts a new Pod.

```
spec.template.spec.restartPolicy: Never
```



Exercise

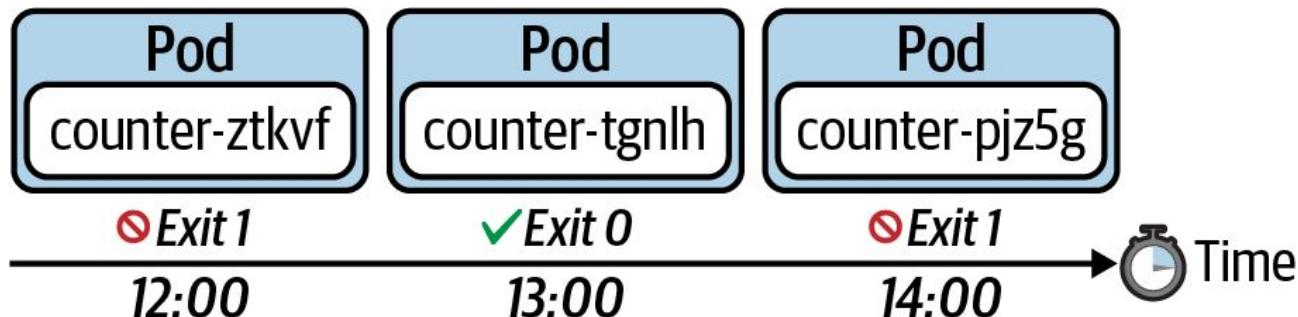
Creating and inspecting a one-time operation using a Job



What is a CronJob?

Kubernetes primitive for executing workload periodically based a schedule

- A CronJob is essentially a Job with similar characteristics.
- The schedule can be defined with a cron-expression you may already know from Unix cron jobs.





Creating a CronJob with Imperative Command

"Render the current date on standard output every hour"

```
$ kubectl create cronjob<br>  current-date<br>    --schedule="* * * * *"  
    --image=nginx:1.24.0  
    -- /bin/sh -c 'echo "Current date: $(date)"'  
cronjob.batch/current-date created
```

CronJob YAML Manifest

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

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: counter
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - args:
                - /bin/sh
                - -c
                - ...
              image: nginx:1.24.0
              name: counter
```

The crontab expression used
to run CronJob periodically

Run in a new Pod

Inspecting a CronJob

Keeps a history of failed and successful Pods

```
$ kubectl get cronjobs
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
counter	*/1 * * * *	False	0	26s	1h

```
$ 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

Configuring Retained History

Being able to know what happened during workload execution retroactively

- Even after completing a task in a Pod controlled by a CronJob, it will not be deleted automatically. Keeping a historical record of Pods can be tremendously helpful for troubleshooting failed workloads or inspecting the logs.
- The attribute `spec.successfulJobsHistoryLimit` configures the number of successful executions. Defaults to 3 successful Jobs.
- The attribute `spec.failedJobsHistoryLimit` configures the number of failed executions. Defaults to 1 failed Job.

Exercise

Creating and inspecting a periodic operation using a CronJob





Volumes

Ephemeral and persistent storage for Pods

What is a Volume?

A directory, possibly with some data in it accessible to the containers in a Pod

- Each container manages files in a temporary file system. Data written to the file system is lost when the container is restarted.
- *Ephemeral Volumes* exist for the lifespan of a Pod. They are useful if you want to share data between multiple containers running in the Pod.
- *Persistent Volumes* preserve data beyond the lifespan of a Pod. They are a good option for applications that require data to exist longer, e.g. in the form of storage for a database-driven application.
- Define the Volume type with `spec.volumes[]` and then reference it in a container with `spec.containers[] .volume.volumeMounts`

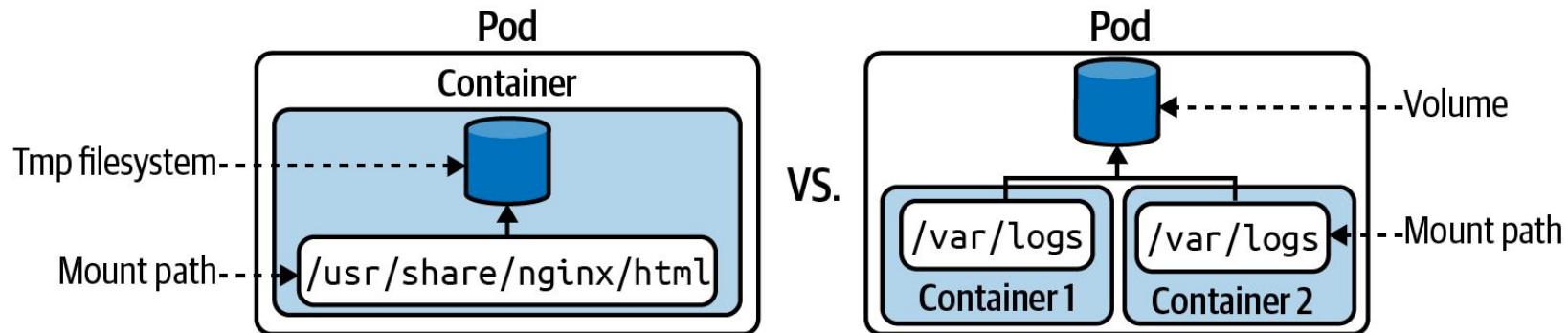
Volume Types

Determines the medium that backs the Volume and its runtime behavior

Type	Description
emptyDir	Empty directory in Pod with read/write access. Only persisted for the lifespan of a Pod.
hostPath	File or directory from the host node's filesystem.
configMap, secret	Provides a way to inject configuration data.
nfs	An existing NFS (Network File System) share. Preserves data after Pod restart.
persistentVolumeClaim	Claims a Persistent Volume.

Ephemeral Volume

Useful for sharing data between containers, or for caching data



Defining a Volume

Provide type and assign mount path per container

```
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 name and type

Specify mount path in container

Using a Volume

You can shell into container and navigate to mount path

```
$ kubectl create -f pod-with-vol.yaml
pod/my-container created

$ kubectl exec -it my-container -- /bin/sh
# cd /var/logs ←
# pwd
/var/logs
# touch app-logs.txt
# ls
app-logs.txt
```

Mount path became
accessible in container

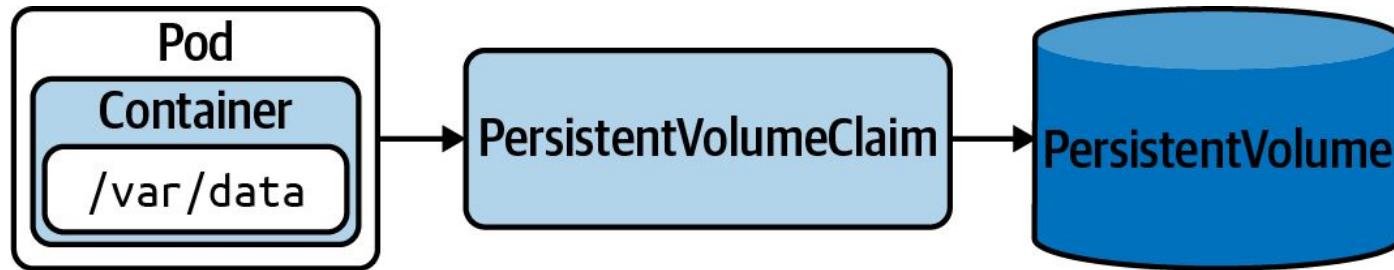
Exercise

Creating and using an
ephemeral Volume



Persistent Volume

Persist data that outlives a Pod, node, or cluster restart



Static vs. Dynamic Provisioning

PersistentVolume object is created manually or automatically

- *Static Provisioning*: Storage device needs to be created first. The PersistentVolume object references the storage device and needs to be created manually.
- *Dynamic Provisioning*: The PersistentVolumeClaim object references a storage class. The PersistentVolume object is created automatically.
- *Storage Class*: Object that knows how to provision a storage device with specific performance requirements.

Defining a PersistentVolume

Define storage capacity, access mode, and host path

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/db
```

Access Mode

Defines read/write capabilities of Volume

Type	Description
ReadWriteOnce	Read-write access by a single node.
ReadOnlyMany	Read-only access by many nodes.
ReadWriteMany	Read-write access by many nodes.
ReadWriteOncePod	Read/write access mounted by a single Pod.

Reclaim Policy

What should happen to PersistentVolume when Claim is deleted?

Type	Description
Retain	Default. When PVC is deleted, PV is “released” and can be reclaimed.
Delete	Deletion removes PV and associated storage.
Recycle	<i>Deprecated.</i> Use dynamic binding instead.

Creating a PersistentVolume

Summarized most important configuration and status

```
$ kubectl create -f db-pv.yaml  
persistentvolume/db-pv created
```

```
$ kubectl get pv db-pv  
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      ...  
db-pv     1Gi        RWO          Retain        Available    ...
```



The object hasn't been consumed by a claim yet

Defining a PersistentVolumeClaim

Will bind to a PersistentVolume based on resource request

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 256Mi
  storageClassName: ""
```

Assign an empty storage class name to
use a statically-created PersistentVolume

Creating a PersistentVolumeClaim

Binds to PersistentVolume if requirements can be fulfilled

```
$ kubectl create -f db-pvc.yaml  
persistentvolumeclaim/db-pvc created
```

```
$ kubectl get pvc db-pvc  
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE  
db-pvc   Bound     pvc       512m      RWO          
```



Binding to the PersistentVolume
object was successful

Mounting a PersistentVolumeClaim in a Pod

Use Volume type persistentVolumeClaim

```
apiVersion: v1
kind: Pod
metadata:
  name: app-consuming-pvc
spec:
  volumes:
    - name: app-storage
      persistentVolumeClaim:
        claimName: db-pvc
  containers:
    - image: alpine
      ...
      volumeMounts:
        - mountPath: "/mnt/data"
          name: app-storage
```



Reference the Volume by claim name

Exercise

Creating and using a
PersistentVolume using
static provisioning



Assigning a StorageClass for Dynamic Provisioning

Creates PersistentVolume object automatically via storage class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: aws-ebs
provisioner: kubernetes.io/aws-ebs
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 256Mi
  storageClassName: aws-ebs
```

Dynamic Creation of PersistentVolume Object

Adds hash to the name of the object

```
$ kubectl get pv,pvc
```

```
NAME ...
```

```
persistentvolume/pvc-b820b919-f7f7-4c74-9212-ef259d421734
```

STORAGECLASS

aws-ebs

```
NAME ...
```

```
persistentvolumeclaim/db-pvc
```

STORAGECLASS

aws-ebs



Created by storage
class provisioner

Exercise

Creating and using a
PersistentVolume using
dynamic provisioning





Multi-Container Pods

Defining and interacting with multiple containers, design patterns

Defining Multiple Containers in a Pod

There are warranted use cases for wanting to run more than a single container

- The general guideline is to operate a microservice in a single container per Pod which promotes a decentralized, decoupled, and distributed architecture.
- Specific use cases call for running multiple containers per Pod. For example, you may want to provide helper functionality that runs alongside the application. In other cases, you may want to initialize your Pod by executing setup scripts, commands, or any other kind of preconfiguration procedure before the application container should start.

Design Patterns

Emerged from Pod requirements

Init Container

- Provide initialization logic concerns to be run before the main application even starts.
- Example: Downloading a configuration files required by application.

Adapter

- Transforms the output produced by the application to make it consumable in the format needed by another part of the system.
- Example: Massaging log data.

Sidecar

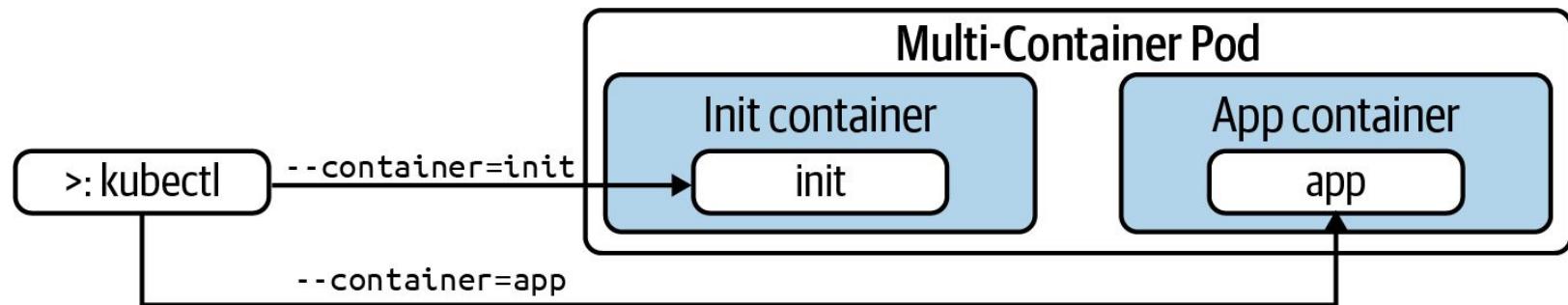
- The sidecars are not part of the main traffic or API of the primary application and operate asynchronously.
- Example: Watcher capabilities.

Ambassador

- Provides a proxy for communicating with external services to hide and/or abstract the complexity.
- Example: Rate-limiting functionality for HTTP(S) calls to an external service.

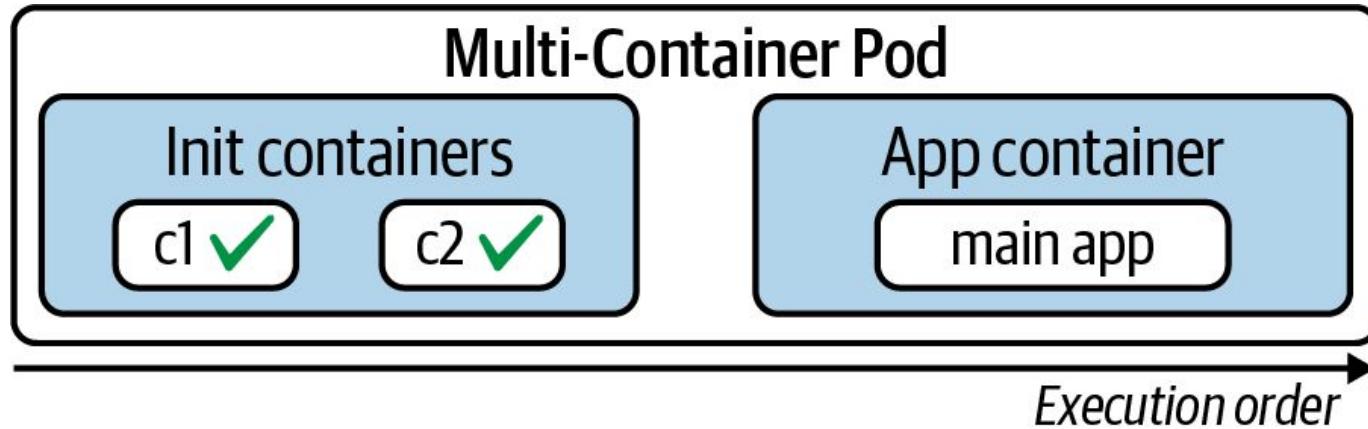
Interacting with a Container

Run kubectl with `--container` or `-c` CLI option



Init Container Life Cycle

Initialization logic before main application containers



Defining an Init Container

Declared adjacent to main application containers with `spec.initContainers`

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container
spec:
  initContainers:
    - image: init:3.2.1
      name: app-initializer
  containers:
    - image: nginx
      name: web-server
```

Uses the same attributes as
main application containers



Creating a Pod with an Init Container

Init container execution renders in status column

```
$ kubectl create -f init.yaml
```

```
pod/business-app created
```

```
$ kubectl get pod business-app
```

NAME	READY	STATUS	RESTARTS	AGE
business-app	0/1	Init:0/1	0	2s

```
$ kubectl get pod business-app
```

NAME	READY	STATUS	RESTARTS	AGE
business-app	1/1	Running	0	8s



Shelling into a Container

Defaults to main application container if not explicit

```
$ kubectl exec -it -c web-server business-app -- /bin/sh
# exit

$ kubectl exec -it business-app -- /bin/sh
Defaulted container "web-server" out of: web-server, ↴
app-initializer (init)
# exit
```



Exercise

Adding an Init Container



Defining More Than One Main Application Container

Add another container definition under `spec.containers`

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container
spec:
  containers:
    - image: nginx:1.13.0
      name: web-server
    - image: transformer:1.0.0
      name: helper
```

Defines a second container
running alongside the
application container



Creating a Pod with Multiple Application Containers

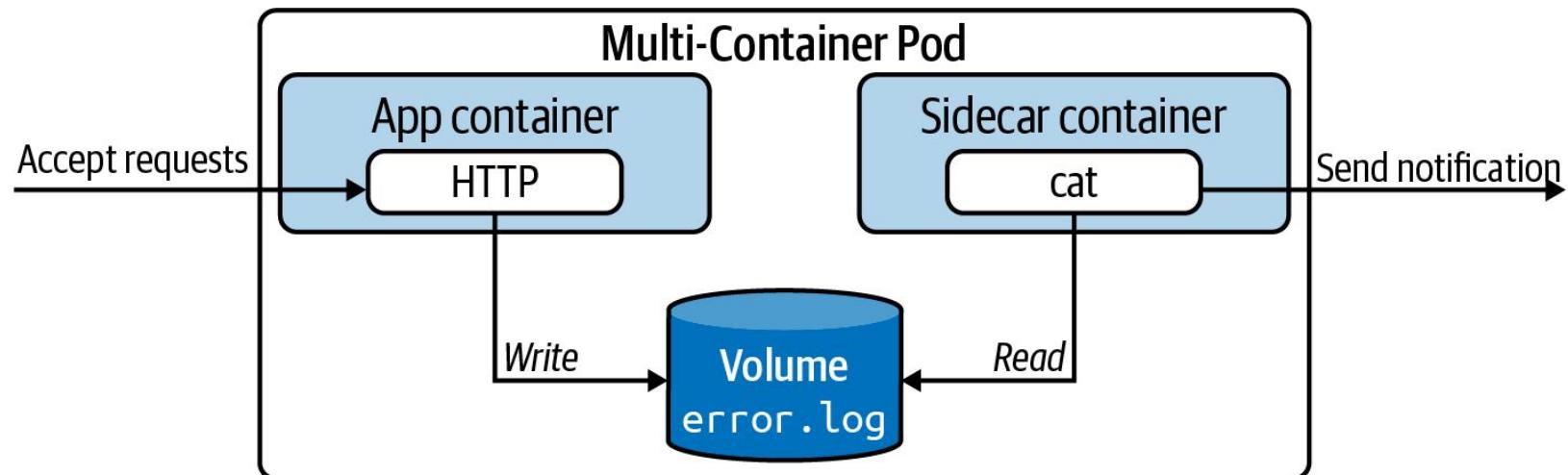
The ready column indicates the number of started containers

```
$ kubectl create -f multi-container.yaml  
pod/multi-container created
```

```
$ kubectl get pod business-app  
NAME          READY   STATUS    RESTARTS  
AGE  
multi-container  2/2     Running   0           2s
```

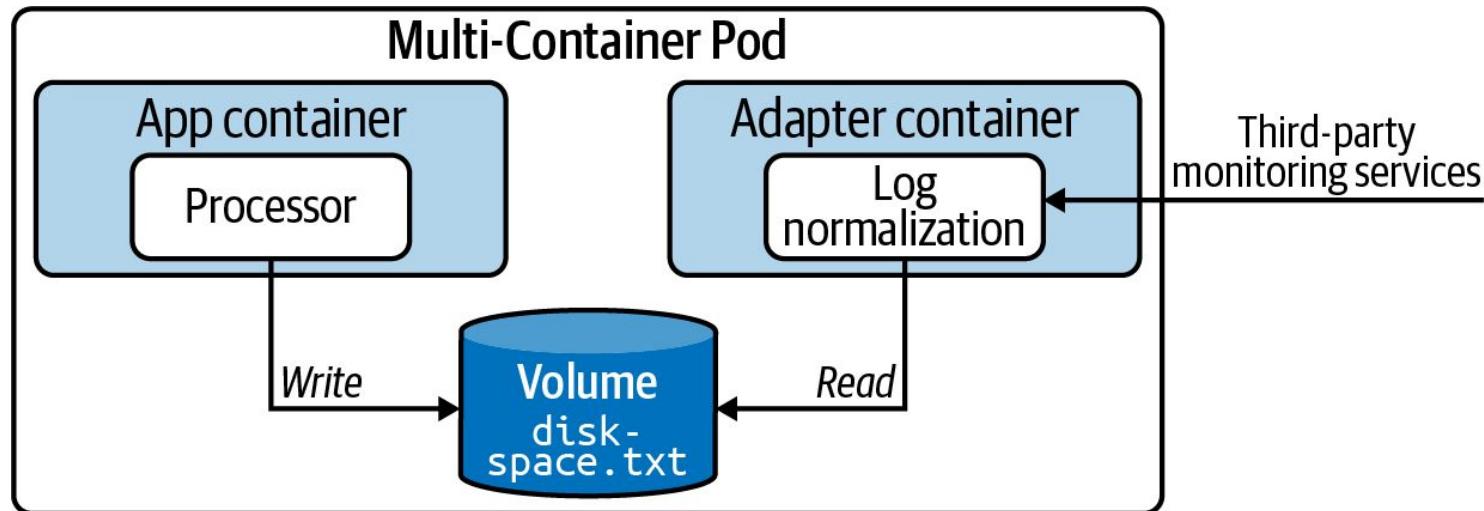
Sidecar Pattern

"Send a notification of error log contains entries"



Adapter Pattern

"Transform log data to JSON to make it consumable by external system"





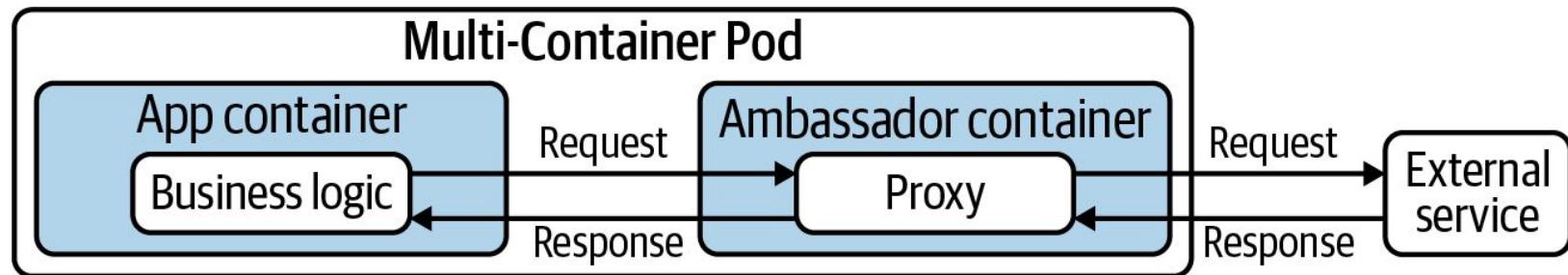
Exercise

Implementing the Adapter pattern



Ambassador Pattern

"OAuth authentication handled by network communication proxy"



Exercise

Implementing the Ambassador pattern



Application Deployment





Labels and Annotations

Label assignment and selection, providing meta information to objects

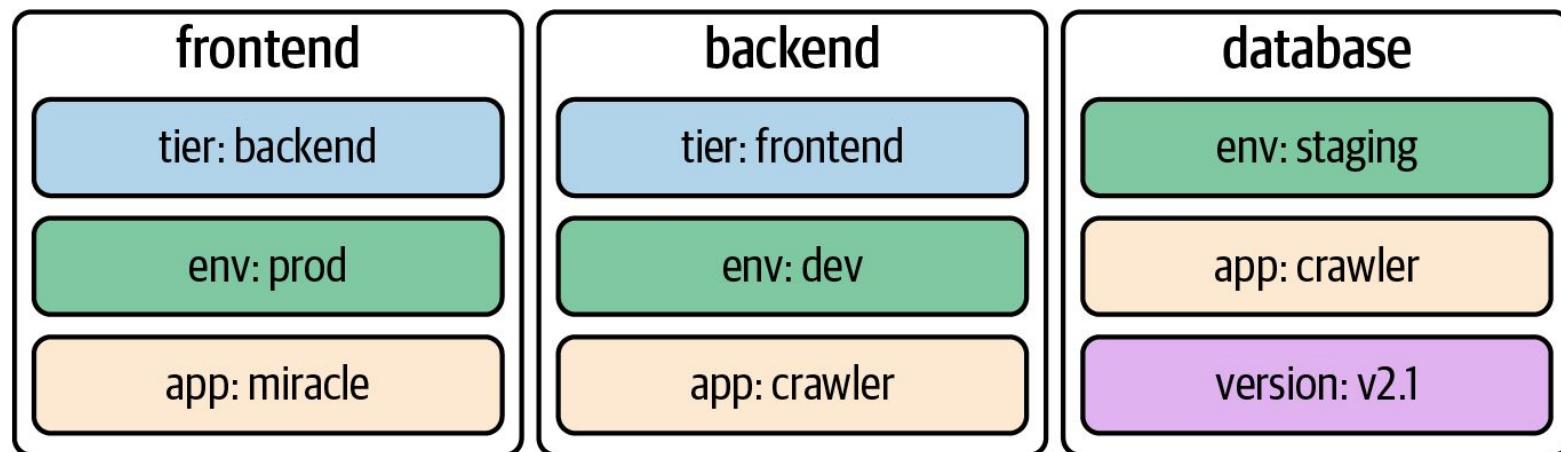
What is a Label?

Essential to querying, filtering and sorting Kubernetes objects

- Key-value pairs assigned to Kubernetes objects with the imperative `label` command or using the `metadata.labels[]` attribute.
- Not meant for elaborate, multi-word descriptions of its functionality.
- Kubernetes limits the length of a label to a maximum of 63 characters and a range of allowed alphanumeric and separator characters.

Example Labels

Three different Pods with each three key-value pairs



Assigning Labels with Imperative Approach

The `label` command lets you add or remove one or many labels

```
$ kubectl label pod nginx tier=backend env=prod app=miracle  
pod/nginx labeled
```

← Declares three label key-value pairs

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

```
$ kubectl label pod nginx tier-  
pod/nginx labeled
```

← Removes the label with key tier

```
$ kubectl get pods --show-labels  
NAME ... LABELS  
pod1 ... env=prod,app=miracle
```

Assigning Labels in a YAML Manifest

Can be provided for a new object or modified for a live object

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



Declares three label
key-value pairs

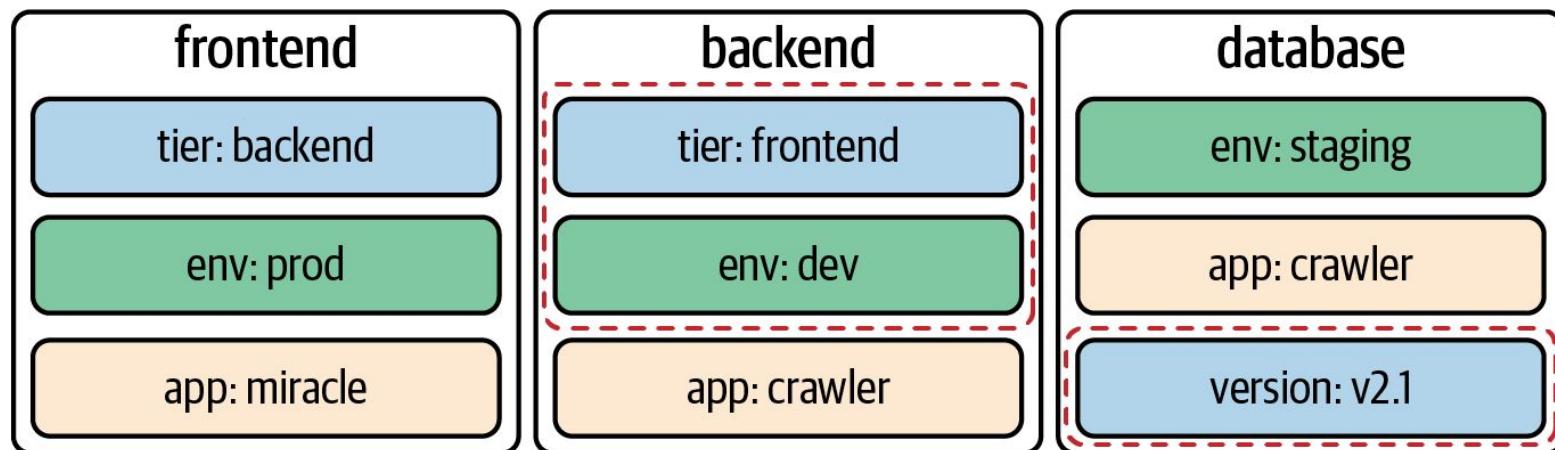
Recommended Labels

Naming convention starts with `app.kubernetes.io`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

Selecting Labels

Label selector uses a set of criteria to query for objects



Label Selection from the CLI

You can specify equality-based and set-based requirements

```
$ kubectl get pods -l tier=frontend,env=dev --show-labels
NAME    ... LABELS
pod2   ... app=crawler,env=dev,tier=frontend
```



```
$ kubectl get pods -l version --show-labels
NAME    ... LABELS
pod3   ... app=crawler,env=staging,version=v2.1
```



```
$ kubectl get pods -l 'tier in (frontend,backend),env=dev' --show-labels
NAME    ... LABELS
pod2   ... app=crawler,env=dev,tier=frontend
```

Boolean AND expression

Key-only expression

Set-based expression
follows Boolean OR

Label Selection in YAML

Certain objects provide label selection via their API

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-network-policy
spec:
  podSelector:
    matchLabels:
      tier: frontend
...
...
```



Select Pods by labels that this network policy should apply to

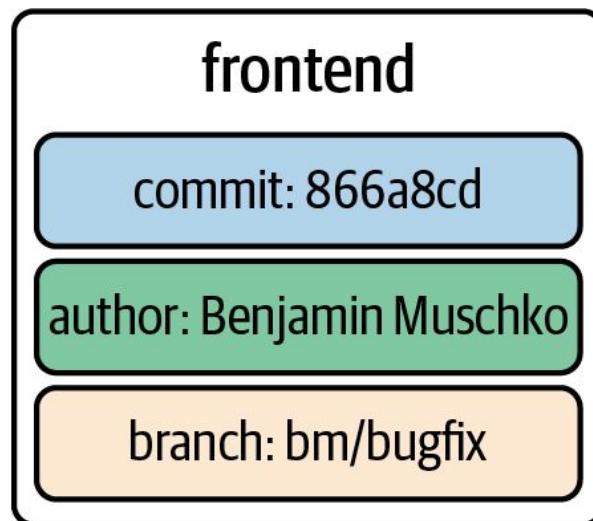
What is an Annotation?

Descriptive metadata without the ability to be queryable

- Key-value pairs for providing descriptive, human-readable metadata assigned to Kubernetes objects with the imperative `annotation` command or using the `metadata.annotations[]` attribute.
- Cannot be used for querying or selecting objects.
- Make sure to put the value of an annotation into single- or double-quotes if it contains special characters or spaces.

Example Annotations

A Pod with three key-value pairs



Assigning Annotations with Imperative Approach

The `annotation` command lets you add or remove one or many annotations

```
$ kubectl annotate pod nginx commit='866a8dc' branch='bm/bugfix'  
pod/nginx annotated
```

← Declares three annotation key-value pairs

```
$ kubectl describe pods my-pod  
Annotations: branch: bm/bugfix  
              commit: 866a8dc
```

```
$ kubectl annotate pod nginx commit-  
pod/nginx annotated
```

← Removes the annotation with key commit

```
$ kubectl describe pods my-pod  
Annotations: branch: bm/bugfix
```

Assigning Annotations in a YAML Manifest

Can be provided for a new object or modified for a live object

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  annotations:
    commit: 866a8dc
    author: 'Benjamin Muschko'
    branch: 'bm/bugfix'
spec:
  ...

```

← Declares three annotation
key-value pairs

Well-Known Annotations

Reserved annotations that may influence runtime treatment

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    kubernetes.io/description: "Manages objects for business app."
    pod-security.kubernetes.io/warn: "baseline"
```



Enforces Pod
Security Standards
at the namespace
level

Exercise

Using Labels and Annotations





Deployments

Scaling workload, deployment strategies



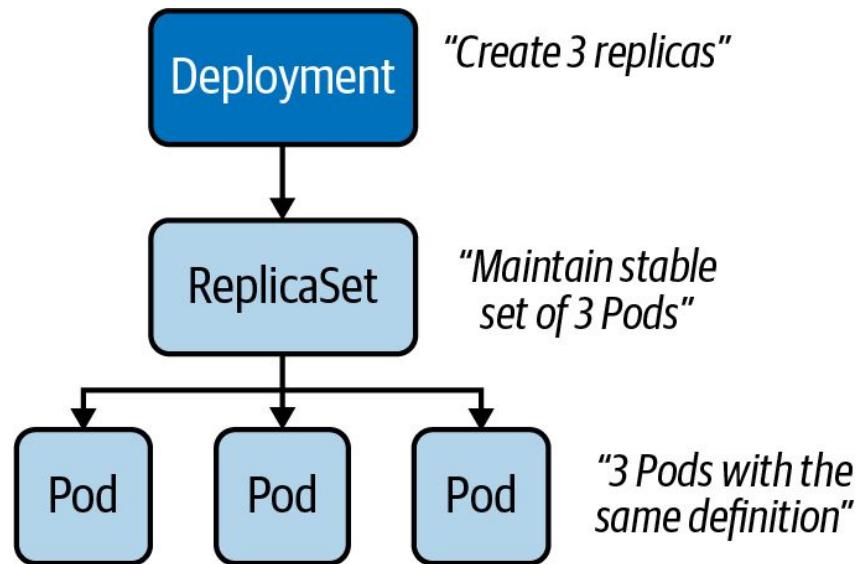
What is a Deployment?

Scaling and replication features for a set of Pods

- Controls a predefined number of Pods with the same configuration, so-called *replicas*.
- The number of replicas can be scaled up or down to fulfill load requirements.
- Updates to the replica configuration can be updated easily and is rolled out automatically.

Example Deployment

A Deployment that controls three replicas



Creating a Deployment with Imperative Approach

Creates objects for the Deployment, ReplicaSet and Pod(s)

```
$ kubectl create deployment  
my-deploy  
--image=nginx:1.14.2  
--replicas=3  
deployment.apps/my-deploy created
```

The default number of replicas is 1 if the parameter wasn't provided

Deployment Template Attributes

Replica configuration is defined under `spec.template`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-deploy
  name: my-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: backend
  template:
    metadata:
      labels:
        tier: backend
    spec:
      containers:
        - image: nginx:1.14.2
          name: nginx
```

Deployment

Template spec attributes
are the same as for a
Pod spec



Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx:1.14.2
      name: nginx
```

Deployment YAML Manifest

Creates an error if label selector and template labels do not match

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-deploy
    name: my-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: backend
  template:
    metadata:
      labels:
        tier: backend
    spec:
      containers:
        - image: nginx:1.14.2
          name: nginx
```

Label selector and
template label assignment
have to match

Listing Deployments

ReplicaSet and Pods can be identified by name prefix

```
$ kubectl get deployments,pods,replicasets
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/my-deploy	1/1	1	1	7m56s

NAME	READY	STATUS	RESTARTS	AGE
pod/my-deploy-8448c488b5-mzx5g	1/1	Running	0	7m56s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/my-deploy-8448c488b5	1	1	1	7m56s



Rendering Deployment Details

Object details show relationship between parent and child

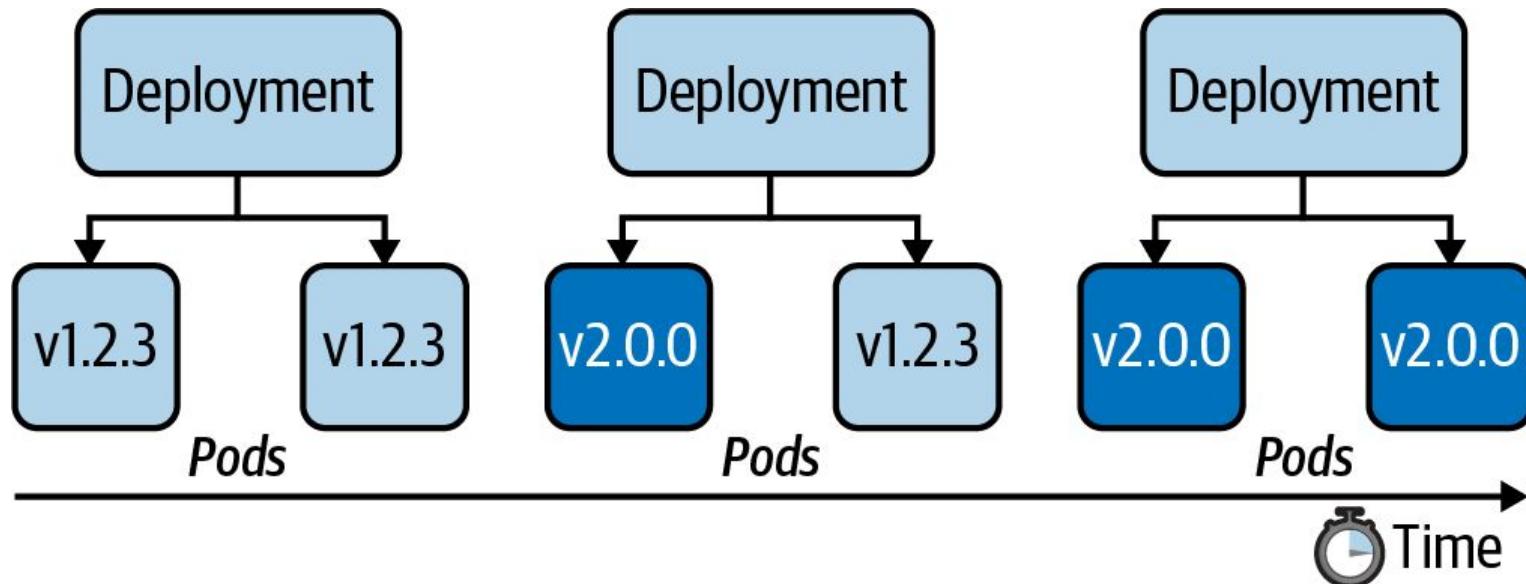
```
$ kubectl describe deployment.apps/my-deploy
Replicas:           1 desired | 1 updated | 1 total | 1 available |
                  0 unavailable
NewReplicaSet:    my-deploy-8448c488b5 (1/1 replicas created)
```

```
$ kubectl describe replicaset.apps/my-deploy-8448c488b5
Controlled By:  Deployment/my-deploy
```

```
$ kubectl describe pod/my-deploy-8448c488b5-mzx5g
Controlled By:  ReplicaSet/my-deploy-8448c488b5
```

Roll Out and Roll Back Feature

"Look ma, shiny new features. Let's deploy them to production!"



Rolling Out a New Revision

The rollout history keeps track of changes

```
$ kubectl rollout history deployment my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
1          <none>
```

```
$ kubectl set image deployment my-deploy nginx=nginx:1.19.2
deployment.apps/my-deploy image updated
```

```
$ kubectl rollout history deployment my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```

Changes the assigned image for Pod template

Rolling Back to Previous Revision

You can revert to any revision available in the history

```
$ kubectl rollout undo deployment my-deploy --to-revision=1
deployment.apps/my-deploy rolled back
```

```
$ kubectl rollout history deployment my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
2          <none>
3          <none>
```



Kubernetes deduplicates a revision
if it points to the same changes

Setting a Change Cause for a Revision

Tracked by annotation with key `kubernetes.io/change-cause`

```
$ kubectl rollout history deployment my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
2          <none>
3          <none>
```

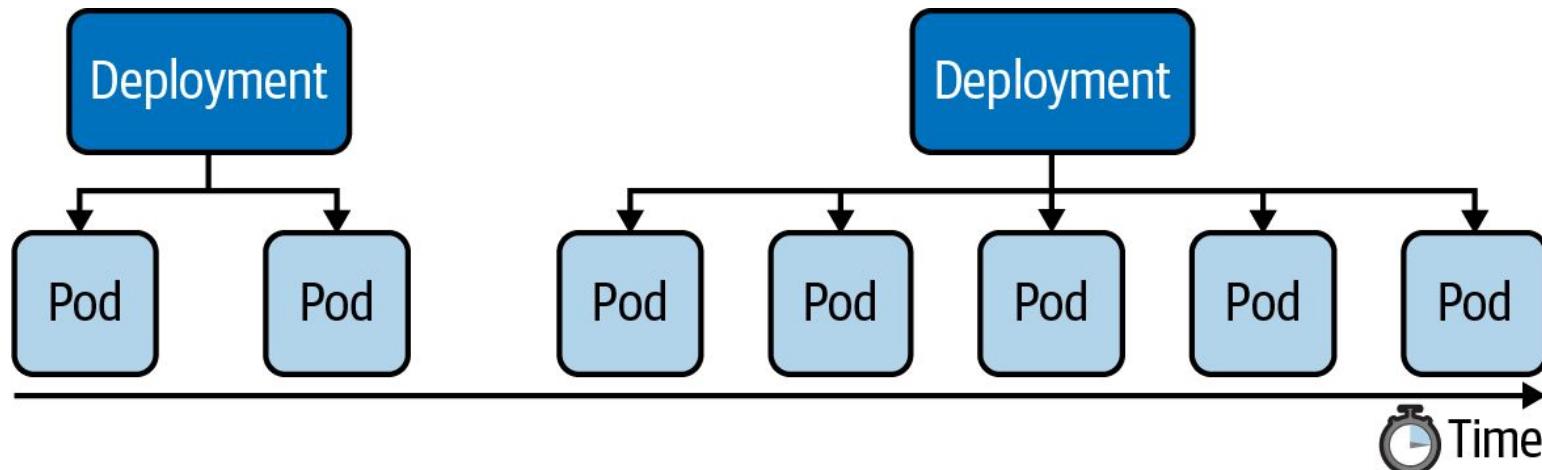
```
$ kubectl annotate deployment my-deploy
kubernetes.io/change-cause="image updated to 1.16.1"
deployment.apps/my-deploy annotated
```

```
$ kubectl rollout history deployment my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
2          <none>
3          image updated to 1.16.1
```

What changed
with this revision?

Manually Scaling a Deployment

"We measured performance under load. Scale up to exactly x number of Pods."



Changing the Number of Replicas

Use scale command or change spec.replicas attribute in live object

```
$ kubectl scale deployment my-deploy --replicas=5
```

```
deployment.apps/my-deploy scaled
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
my-deploy-8448c488b5-5f5tg	1/1	Running	0	44s
my-deploy-8448c488b5-9xplx	1/1	Running	0	44s
my-deploy-8448c488b5-d8q4t	1/1	Running	0	44s
my-deploy-8448c488b5-f5kkm	1/1	Running	0	44s
my-deploy-8448c488b5-mzx5g	1/1	Running	0	3d19h

Exercise

Performing rolling updates
and manually scaling a
Deployment



Common Deployment Strategies

Choosing the right deployment procedure depends on the needs

- *Recreate*: Terminate the old version and release the new one.
- *Ramped*: Release a new version on a rolling update fashion, one after the other.
- *Blue/green*: Release a new version alongside the old version then switch traffic.
- *Canary*: Release a new version to a subset of users, then proceed to a full rollout.

Recreate Deployment Strategy

Terminates all the running instances then recreate them

```
spec:  
  replicas: 3  
  strategy:  
    type: Recreate
```

← Set the strategy explicitly

- *Pros:* Good option for development environments as everything is renewed at once
- *Cons:* Can cause downtime while applications are starting up

Ramped Deployment Strategy

Secondary ReplicaSet is created with new version of application

```
spec:  
  replicas: 3  
  strategy:  
    type: RollingUpdate  
    rollingUpdate:  
      maxSurge: 2  
      maxUnavailable: 0
```



Determines how update should be performed

- *Pros:* New version is slowly distributed over time, no downtime
- *Cons:* Breaking API changes can make consumers incompatible

Blue/Green Deployment Strategy

Deployment object with new version is created alongside

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-deploy
    version: 1.0.0
  name: blue
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-deploy
    version: 1.1.0
  name: green
```

- *Pros:* No downtime, traffic can be routed when ready
- *Cons:* Resource duplication, configuration of network routing

Canary Deployment Strategy

Two Deployment objects with different traffic distribution

```
spec:  
  replicas: 3
```

```
spec:  
  replicas: 1
```

- *Pros:* New version released to subset of users, A/B testing of features and performance
- *Cons:* May require a load balancer for fine-grained distribution

Exercise

Implementing the
blue-green deployment
strategy



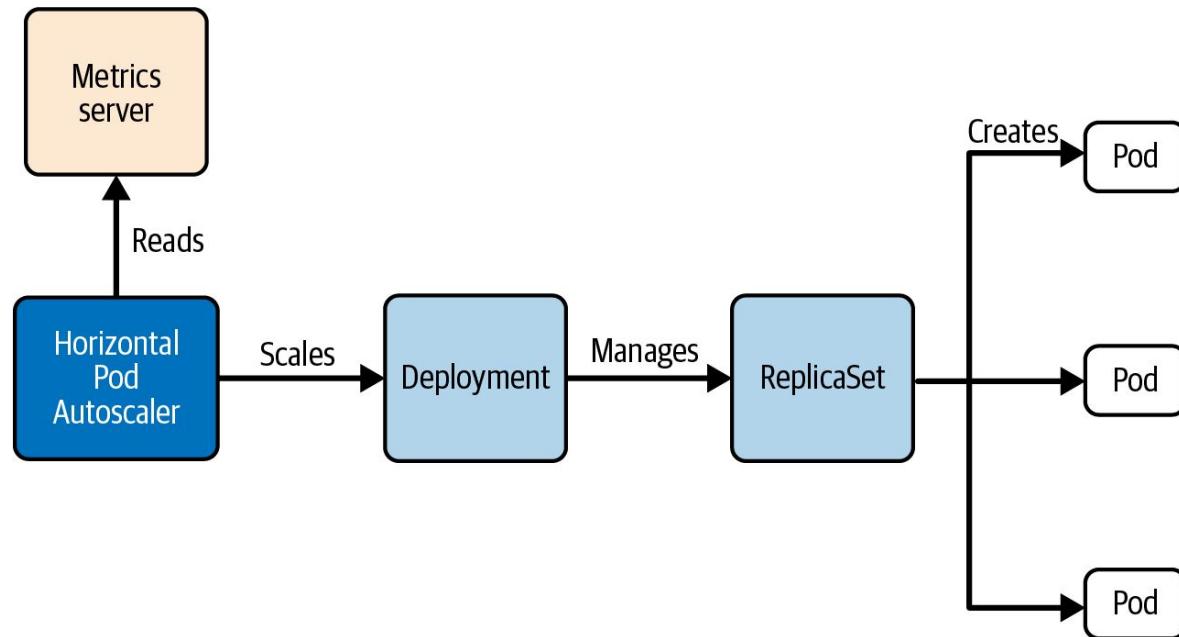
Types of Autoscalers

Scaling decisions are made based on metrics

- *Horizontal Pod Autoscaler (HPA)*: A standard feature of Kubernetes that scales the number of Pod replicas based on CPU and memory thresholds.
- *Vertical Pod Autoscaler (VPA)*: Scales the CPU and memory allocation for existing Pods based on historic metrics. Supported by a cloud provider as an add-on or needs to be installed manually.
- Both autoscalers use the [metrics server](#). You need to install the component. A Pod should also set the resource requests and limits.

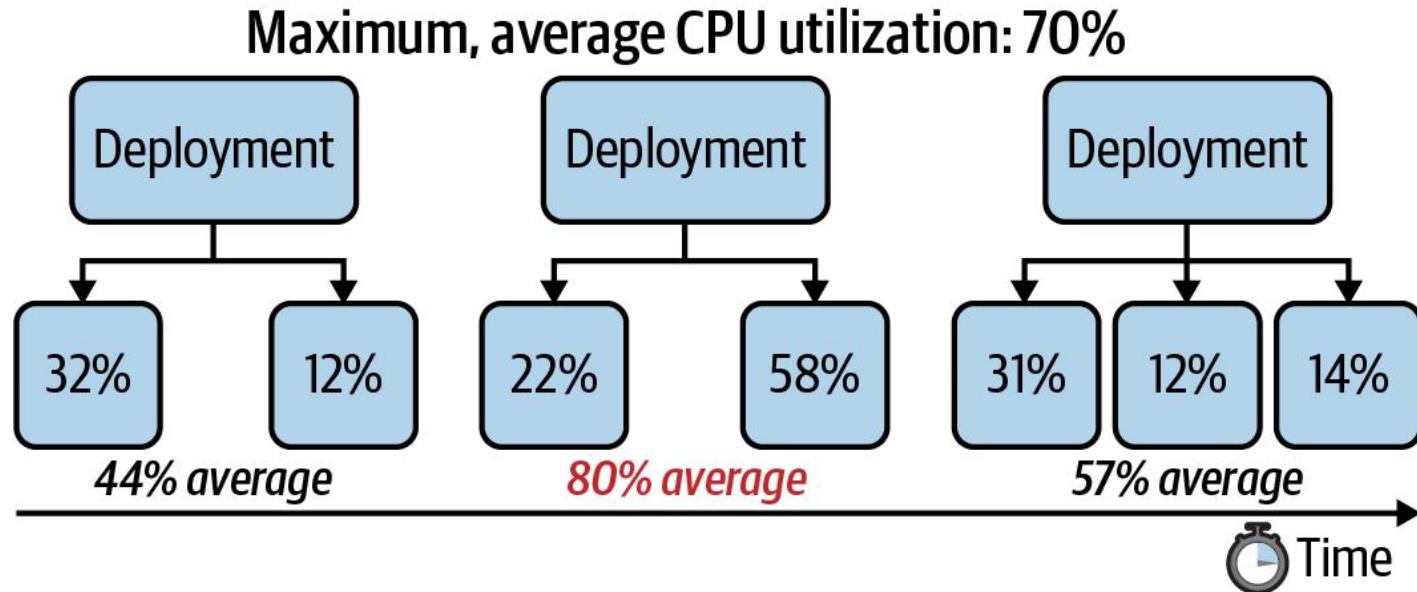
Horizontal Pod Autoscaler (HPA)

Kubernetes primitive that reaches into metrics server



Autoscaling a Deployment by CPU

"Auto-scale based on average CPU utilization across all replicas."



Horizontal Pod Autoscaler YAML Manifest

Creates an error if label selector and template labels do not match

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-cache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app-cache
  minReplicas: 3
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

The scaling target,
a Deployment

One or many resource types that
the thresholds should apply to

Inspecting the Horizontal Pod Autoscaler

Use scale command or change spec.replicas attribute in live object

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
app-cache	Deployment/app-cache 2m14s	15%/80%	3	5	3	



Renders current resource utilization
and compares them with thresholds



Exercise

Creating a Horizontal Pod
Autoscaler for a
Deployment





Helm

Deploying an application defined by a set of YAML manifests

What is Helm?

Templating engine and package manager for a set of manifests

- The artifact produced by the Helm executable is a so-called *chart file* bundling the manifests that comprise the API resources of an application.
- At runtime, it replaces placeholders in YAML template files with actual, end-user defined values.
- Chart files can be distributed to a chart repository e.g. [central chart repository](#) and consumed from there (e.g., for running Grafana or PostgreSQL).



Finding a Chart

Searches the Artifact Hub for a chart with a matching name

```
$ helm search hub jenkins
```

URL

<https://artifacthub.io/packages/helm/bitnami/jenkins>

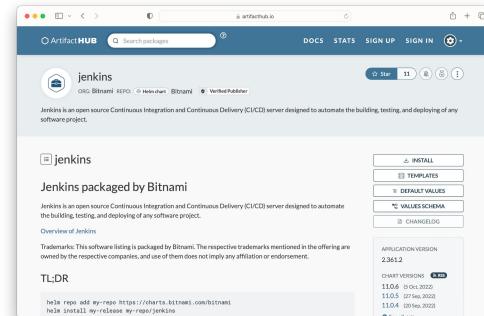
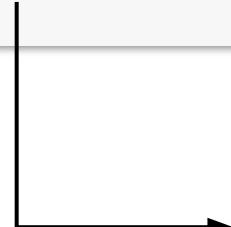
...

CHART

11.0.6

VERSION APP

2.361.2



Installing a Chart

You may have to add an external repository that hosts chart file

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories

$ helm install jenkins bitnami/jenkins
...
CHART NAME: jenkins
CHART VERSION: 11.0.6
APP VERSION: 2.361.2

** Please be patient while the chart is being deployed **
```

Listing Installed Charts

Information about chart details

```
$ helm list
```

NAME	NAMESPACE	REVISION	...	STATUS	CHART	APP VERSION
jenkins	default	1		deployed	jenkins-11.0.6	2.361.2

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
pod/bitnami-jenkins-764b44cc99-twf6f	1/1	Running	0	7m34s
...				



Uninstalling a Chart

Deletes Kubernetes objects controlled by chart

```
$ helm uninstall jenkins
release "jenkins" uninstalled

$ kubectl get pods
No resources found in default namespace.
```

Exercise

Installing an existing Helm chart from the central chart repository



Standard Chart Structure

Predefined directory structure with mandatory `Chart.yaml` and `values.yaml`

```
$ tree
.
├── Chart.yaml
└── templates
    ├── web-app-pod-template.yaml
    └── web-app-service-template.yaml
└── values.yaml
```

Chart File

Describes the meta information of the chart (e.g., name and version)

Chart.yaml

```
apiVersion: 1.0.0
name: web-app
version: 2.5.4
```

Values File

Key-value pairs used at runtime to replace placeholders in the YAML manifests

values.yaml

```
db_host: mysql-service
db_user: root
db_password: password
service_port: 3000
```

Additional [built-in objects](#) are available for use.

Template File

YAML manifest that can define placeholders using double curly braces

web-app-pod-template.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: web-app
      env:
        - name: DB_HOST
          value: {{ .Values.db_host }}
        - name: DB_USER
          value: {{ .Values.db_user }}
        - name: DB_PASSWORD
          value: {{ .Values.db_password }}
...
...
```



Previewing Final Templates

Replaces placeholders with actual values from `values.yaml`

```
$ helm template .  
---  
# Source: Web Application/templates/web-app-pod-template.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
  name: web-app  
spec:  
  containers:  
    - image: bmuschko/web-app:1.0.1  
      name: web-app  
      env:  
        - name: DB_HOST  
          value: mysql-service  
        - name: DB_USER  
          value: root  
        - name: DB_PASSWORD  
          value: password  
  ...
```

Installing a Chart from Local Files

Helpful for trying out the deployment before publishing the chart file

```
$ helm install web-app .
```

```
NAME: web-app
LAST DEPLOYED: Wed Apr 19 11:58:34 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

```
$ kubectl get pods,services
```

NAME	READY	STATUS	RESTARTS	AGE
pod/web-app	1/1	Running	0	3m24s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT (S)	AGE
service/web-app-service	NodePort	10.109.44.239	<none>	3000:30456/TCP	3m24s

Overriding Values

End user can tweak runtime behavior to personal requirements

```
$ helm install --namespace custom --create-namespace --set service_port=9090 web-app .  
NAME: web-app  
LAST DEPLOYED: Wed Apr 19 11:58:34 2023  
NAMESPACE: custom  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None
```



Providing a custom namespace (defaults to default namespace)

Override values in values.yaml

Bundling the Template Files into a Chart Archive File

Archive file is usually published to a chart repository for consumption

```
$ helm package .
```

Successfully packaged chart and saved it to:

```
/Users/bmuschko/helm/web-app-2.5.4.tgz
```

```
$ ls
```

```
web-app-2.5.4.tgz
```

Exercise

Implementing, packaging,
and installing a custom
Helm chart



Application Observability & Maintenance





API Deprecations

Kubernetes release cycle, API deprecations and removals

Deprecation Policy

Some APIs will be deprecated and eventually removed

- The Kubernetes project releases 3 versions per year (see [release cadence](#)).
- With each release, an API can become deprecated which means that is scheduled for removal or replacement. You can find more information about the details in the [Kubernetes Deprecation Policy](#).
- The [Deprecated API Migration Guide](#) shows a list of deprecated APIs and the scheduled releases that will remove support for the API.
- The use of a deprecated API renders a warning message when creating the object that uses it.

Listing Available API version

kubectl can discover API version useable in manifests

```
$ kubectl api-versions
admissionregistration.k8s.io/v1
apiextensions.k8s.io/v1
apiregistration.k8s.io/v1
apps/v1
authentication.k8s.io/v1
authorization.k8s.io/v1
autoscaling/v1
autoscaling/v2
...
```



Deprecated HPA API

API was deprecated with Kubernetes 1.23+, to be removed with 1.26

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
```

The **autoscaling/v2beta2** API version of HorizontalPodAutoscaler is no longer served as of v1.26.

- Migrate manifests and API clients to use the **autoscaling/v2** API version, available since v1.23.
- All existing persisted objects are accessible via the new API



Deprecated HPA API Warning Message

kubectl renders a warning message, but creates object

```
$ kubectl create -f hpa.yaml
Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated ←
in v1.23+, unavailable in v1.26+; use autoscaling/v2 ←
HorizontalPodAutoscaler
```



Removed ClusterRole API

Uses removed API with Kubernetes version 1.22

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

The **rbac.authorization.k8s.io/v1beta1** API version of ClusterRole, ClusterRoleBinding, Role, and RoleBinding is no longer served as of v1.22.



Removed ClusterRole API Error Message

kubectl renders an error message, as API does not exist anymore

```
$ kubectl create -f clusterrole.yaml
error: resource mapping not found for name: "pod-reader" namespace: <-
"" from "clusterrole.yaml": no matches for kind "ClusterRole" in    <
version "rbac.authorization.k8s.io/v1beta1"
```

Using the ClusterRole API Replacement

Usage of replacement API

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- Migrate manifests and API clients to use the **rbac.authorization.k8s.io/v1** API version, available since v1.8.
- All existing persisted objects are accessible via the new APIs
- No notable changes

Exercise

Identifying and replacing a
deprecated API





Probes

Readiness, liveness, and startup probes



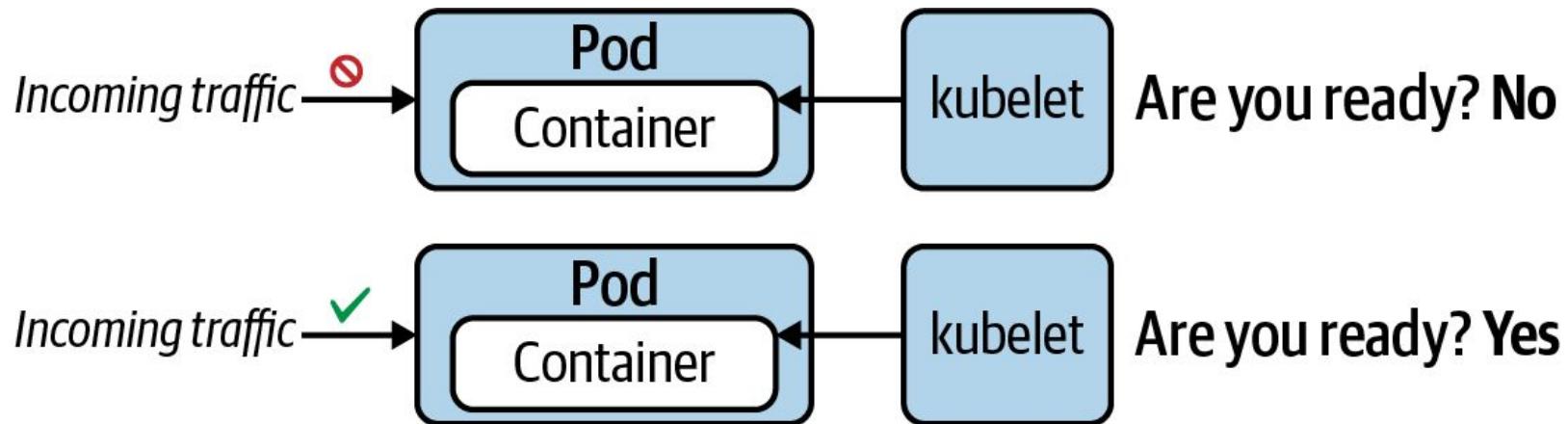
What is a Probe?

Detection and potential correction of application runtime issues

- You can configure a container with *health probes* to execute a periodic mini-process that checks for certain conditions.
- Verification methods include running a custom command, sending a HTTP GET request, or opening a TCP socket connection. Pick the method most conducive to your application.
- The verification method defined by a probe is executed by the kubelet.

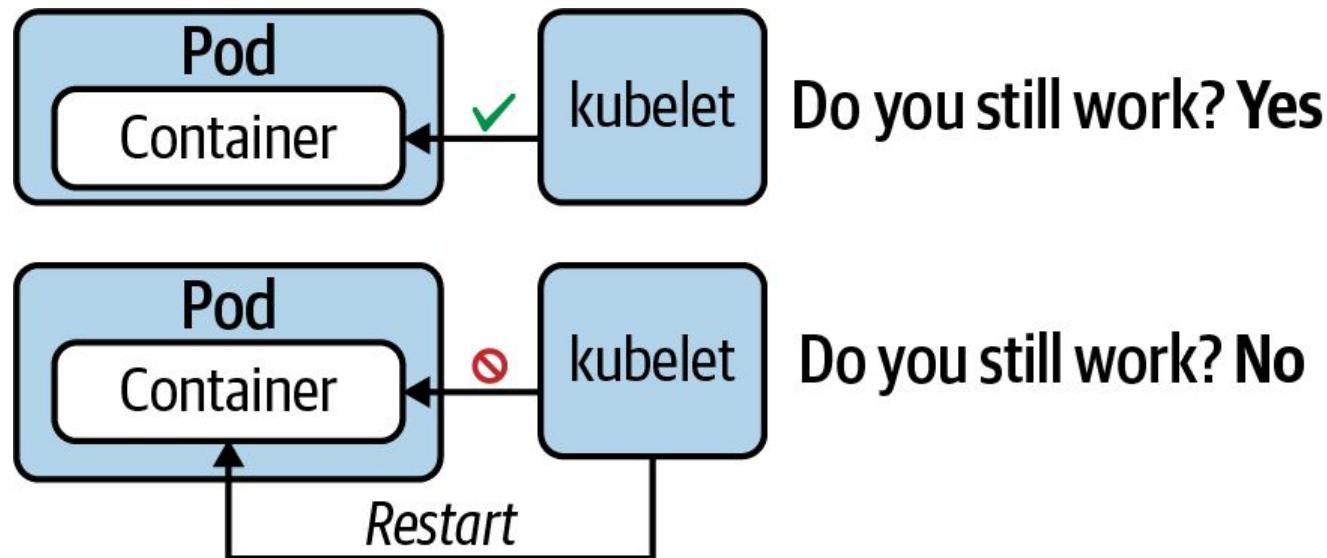
Readiness Probe

"Is application ready to serve requests?"



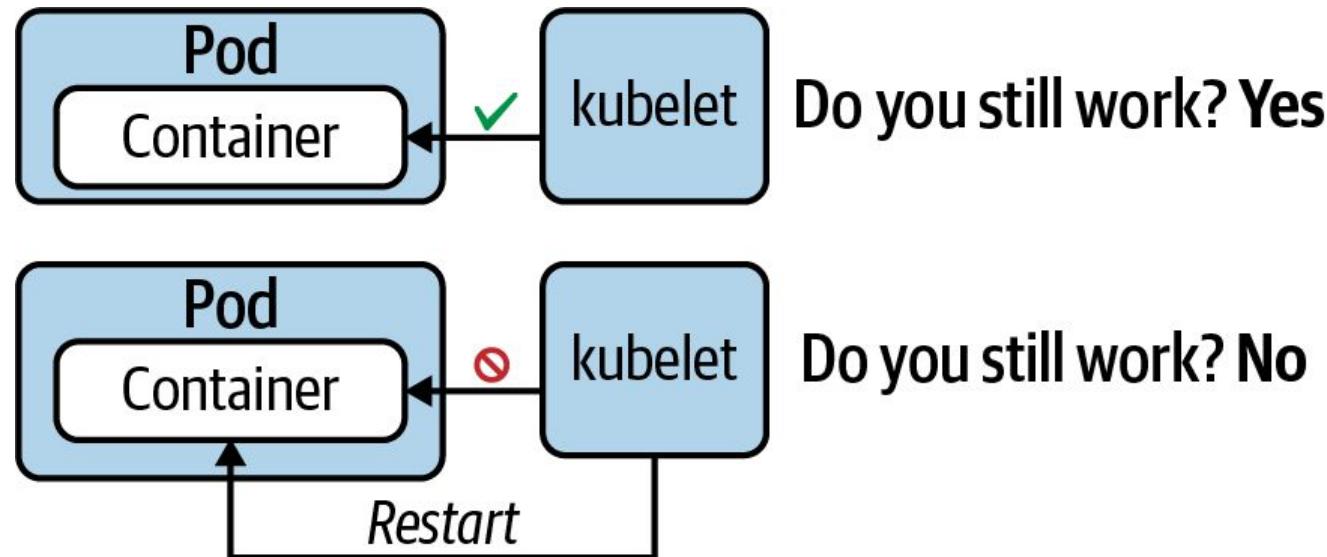
Liveness Probe

"Does the application still function without errors?"



Startup Probe

"Legacy application may need longer to start. Hold off on liveness probing."



Health Verification Methods

Any verification method can be used by any of the probe types

Method	Option	Description
Custom Command	exec.command	Executes a command inside of the container e.g. a cat command and checks its exit code. Kubernetes considers an zero exit code to be successful. A non-zero exit code indicates an error.
HTTP GET Request	httpGet	Sends an HTTP GET request to an endpoint exposed by the application. A HTTP response code in the range of 200 and 399 indicates success. Any other response code is regarded as an error.
TCP socket connection	tcpSocket	Tries to open a TCP socket connection to a port. If the connection could be established, the probing attempt was successful. The inability to connect is accounted for as an error.
gRPC	grpc	The application implements the GRPC Health Checking Protocol . Verifies whether the server is able to handle rpcs.

Health Check Attributes

Any verification method can be used by any of the probe types

Attribute	Default Value	Description
initialDelaySeconds	0	Delay in seconds until first check is executed.
periodSeconds	10	Interval for executing a check (e.g., every 20 seconds).
timeoutSeconds	1	Maximum number of seconds until the check operation times out.
successThreshold	1	Number of successful check attempts until probe is considered successful after a failure.
failureThreshold	3	Number of failures for check attempts before probe is marked failed and takes action.
terminationGracePeriodSeconds	30	Grace period before forcing a container stop upon failure.

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

Defining a Liveness Probe

Querying an event log 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:
        - test `find /tmp/heartbeat.txt -mmin -1` 
    initialDelaySeconds: 10
    periodSeconds: 5
```

Does the file get updated periodically by the application process?



Defining a Startup Probe

Opening a TCP socket connection exposed by the 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
      livenessProbe:
        ...
...
```

Tries to open a TCP socket
connection to a port

Using a Named Port

Referencing a port by assigned name in multiple probes

```
ports:  
- name: http-port  
  containerPort: 8080  
  
readinessProbe:  
  httpGet:  
    path: /healthz  
    port: http-port  
  
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: http-port
```

Exercise

Defining probes for a container





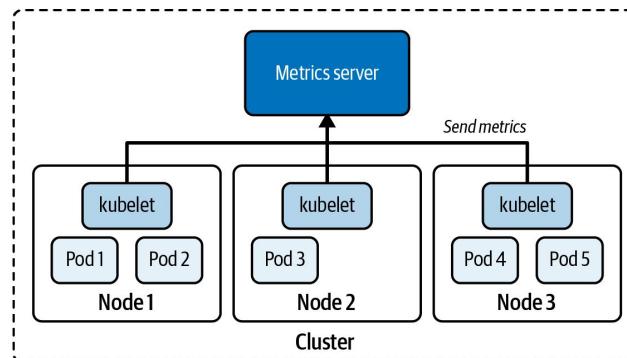
Metrics Server

Retrieving, inspecting, and using resource metrics

What is the Metrics Server?

Cluster-wide resource metrics aggregator

- Gathers resource consumption metrics on nodes, e.g. CPU and memory. The kubelet send those metrics to the Kubernetes API server and exposes them through the Metrics API.
- The metrics server stores data in memory and does not persist data over time.



Installing the Metrics Server

A Kubernetes cluster does not come with the metrics server by default



```
$ minikube addons enable metrics-server  
The 'metrics-server' addon is enabled
```



```
$ kubectl apply -f  
https://github.com/kubernetes-sigs/metrics-server/r  
eleases/latest/download/components.yaml
```

Querying for Metrics

Metrics for nodes and Pods can be rendered with the `top` command

```
$ kubectl top nodes
```

NAME	CPU (cores)	CPU%	MEMORY (bytes)	MEMORY%
controlplane	193m	9%	1239Mi	32%
node01	119m	5%	889Mi	23%

```
$ kubectl top pods
```

NAME	CPU (cores)	MEMORY (bytes)
frontend	38m	85Mi
backend	29m	59Mi



Kubernetes Features that Tap into Metrics

Where else does the metrics server come into play? ([docs](#))

- A Horizontal Pod Autoscaler (HPA) can only determine if Deployment replicas need to be scaled up or down based on available resource consumption determined by metrics server. If the current resource consumption cannot be determined, it is indicated by <unknown> in the TARGETS column of an HPA.
- Pod scheduling on cluster nodes is based on the definition of resource requests. The scheduler looks at the resources capacity available on a cluster node and decides if the workload can be scheduled for the Pod's resource request.



Exercise

Determining metrics for
Pods





Troubleshooting Pods and Containers

Built-in mechanisms for identifying the root cause of a failure

Retrieving High-Level Pod Information

Check the status of a Pod first

```
$ kubectl get pods -n t67
```

NAME	READY	STATUS	RESTARTS	AGE
misbehaving-pod	0/1	ErrImagePull	0	2s

```
$ kubectl get pods -n k31
```

NAME	READY	STATUS	RESTARTS	AGE
successful-pod	1/1	Running	0	5s

Failed to pull
container image

Looks successful on
the surface-level



Common Pod Error Statuses

Poorly-documented in Kubernetes docs, encountered often in practice

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 authentication.
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 Engine).
CreateContainerConfigError	ConfigMap or Secret referenced by container cannot be found.	Check correct name of the configuration object, verify the existence of the configuration object in the namespace.



Inspecting Events of a Specific Pod

Detailed event information can be found in `Events` section

```
$ kubectl describe pod myapp
...
Events:
  Type      Reason     Age           From            Message
  ----      ----     --           ----           -----
  Normal    Scheduled  <unknown>    default-scheduler  Successfully<
  assigned default/secret-pod to minikube
  Warning   FailedMount 3m15s        kubelet, minikube  Unable to<
  attach or mount volumes: unmounted volumes=[mysecret], unattached<
  volumes=[default-token-bf8rh mysecret]: timed out waiting for the condition
  Warning   FailedMount  68s (x10 over 5m18s)  kubelet, minikube  ↴
  MountVolume.SetUp failed for volume "mysecret" : secret "mysecret" not found
  ...
...
```

Inspecting Events Across all Pods

Lists events for all Pods in the given namespace

```
$ kubectl get events
```

LAST SEEN	TYPE	REASON	OBJECT	MESSAGE
3m14s	Warning	BackOff	pod/custom-cmd	Back-off restarting ↴ failed container
2s	Warning	FailedNeedsStart	cronjob/google-ping	Cannot determine if ↴ job needs to be ↴ started: too many ↴ missed start time ↴ (> 100). Set or ↴ decrease .spec. ↴ startingDeadline ↴ Seconds or check ↴ clock skew



Inspecting Container Logs

Application failure may not necessarily surface on the Kubernetes-level

```
$ kubectl create -f crash-loop-backoff.yaml  
pod/incorrect-cmd-pod created
```

```
$ kubectl get pods incorrect-cmd-pod  
NAME                  READY   STATUS            RESTARTS   AGE  
incorrect-cmd-pod    0/1     CrashLoopBackOff   5          3m20s
```

```
$ kubectl logs incorrect-cmd-pod  
/bin/sh: unknown: not found
```



Opening an Interactive Shell to a Container

Allows for exploring container file system and processes

```
$ kubectl logs failing-pod
/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory

$ kubectl exec failing-pod -it -- /bin/sh
# mkdir -p ~/tmp
# cd ~/tmp
# ls -l
total 4
-rw-r--r-- 1 root root 112 May  9 23:52 curr-date.txt
# exit
```

Distroless Containers

Some container images do not provide a shell

```
$ kubectl run minimal-pod --image=k8s.gcr.io/pause:3.1
pod/minimal-pod created

$ kubectl exec minimal-pod -it -- /bin/sh
OCI runtime exec failed: exec failed: container_linux.go:349:
↳
Starting container process caused "exec: \"/bin/sh\": stat   ↳
/bin/sh: no such file or directory": unknown
command terminated with exit code 126
```



Debugging Using an Ephemeral Container

Using a disposable container for troubleshooting distroless containers

```
$ kubectl debug -it minimal-pod --image=busybox
Defaulting debug container name to debugger-jf98g.
If you don't see a command prompt, try pressing enter.
/ # pwd
/
/ # exit
Session ended, resume using 'kubectl attach minimal-pod -c debugger-jf98g -i -t' command when the pod is running
```

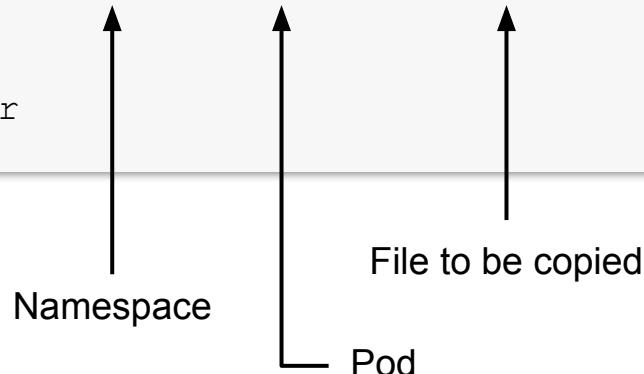
You can create a copy of a container in a new Pod with `--copy-to` for more complex debugging scenarios.

Copying a File from the Container

The `tar` binary needs to exist in container

```
$ kubectl exec -it nginx -n code-red -- /bin/sh  
# tar cvf error-log.tar /var/log/nginx/error.log  
# exit  
  
$ kubectl cp code-red/nginx:/error-log.tar error-log.tar
```

```
$ ls  
error-log.tar
```





Exercise

Troubleshooting a Pod



Application Environment, Configuration & Security





Custom Resource Definitions (CRDs)

Extending the Kubernetes API

What is a Custom Resource Definition (CRD)?

Extension point for introducing custom API primitives

- Some use cases with custom requirements cannot be fulfilled with the built-in Kubernetes primitives and functionality.
- A CRD allows you to define, create, and persist one or many custom objects and expose them with the Kubernetes API.
- CRDs are combined with *controllers* to make them actually useful. Controllers interact with the API server and implement the custom logic. This is called the *Operator pattern*.

Example CRD

Smoke testing against a Service after deployment

- *Assumption:* An web-based application stack is deployed via a Deployment with one or more than one replica. The Service object routes network traffic to the Pods.
- *Desired functionality:* After the deployment happened, we want to run a quick smoke test against the Service's DNS name to ensure that application is operational. The result of the smoke test will be sent to an external service for the purpose of rendering it on the UI dashboard.
- *Goal:* Implement the Operator pattern (CRD and controller) for the use case.

CRD YAML Manifest

Defines the schema for the custom object

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: smoketests.stable.bmuschko.com
spec:
  group: stable.bmuschko.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                service:
                  type: string
              ...
  scope: Namespaced
  names:
    plural: smoketests
    singular: smoketest
    kind: SmokeTest
    shortNames:
      - st
```

Combination of <plural>.<group>

Versions supported by CRD

Attributes for custom type

Identifiers for custom type

Custom Resource Object YAML Manifest

Instantiation of CRD kind

```
apiVersion: stable.bmuschko.com/v1
kind: SmokeTest
metadata:
  name: backend-smoke-test
spec:
  service: backend
  path: /health
  timeout: 600
  retries: 3
```

← Group and Version of custom kind

Kind defined by CRD

Attributes and values that make
custom kind configurable



Creating the CRD and Custom Object

CRD object needs to be created before custom objects

```
$ kubectl create -f loadtest-resource.yaml
customresourcedefinition.apiextensions.k8s.io/smoketests.stable.bmuschko.com
created

$ kubectl create -f loadtest.yaml
smoketest.stable.bmuschko.com/backend-smoke-test created
```

Discovering CRDs

Once installed, you can list all available CRDs

```
$ kubectl api-resources --api-group=stable.bmuschko.com
```

NAME	SHORTNAMES	APIVERSION	NAMESPACED	KIND
smoketests	st	stable.bmuschko.com/v1	true	SmokeTest

```
$ kubectl get crd
```

NAME	CREATED AT
smoketests.stable.bmuschko.com	2023-05-04T14:49:40Z

Controller Implementation

The actual logic for smoke tests lives in the controller

- You can interact with the custom resource object using CRUD (create/read/update/delete) operations; however, no smoke test logic will actually be initiated.
- A controller acts as a reconciliation process by inspecting the state through the API server. At runtime, it polls for new `SmokeTest` objects and sends the result to the external service.
- Controllers can use one of the Kubernetes [client libraries](#), written in Go or Python, to access custom resources.

Exercise

Defining and interacting
with a CRD



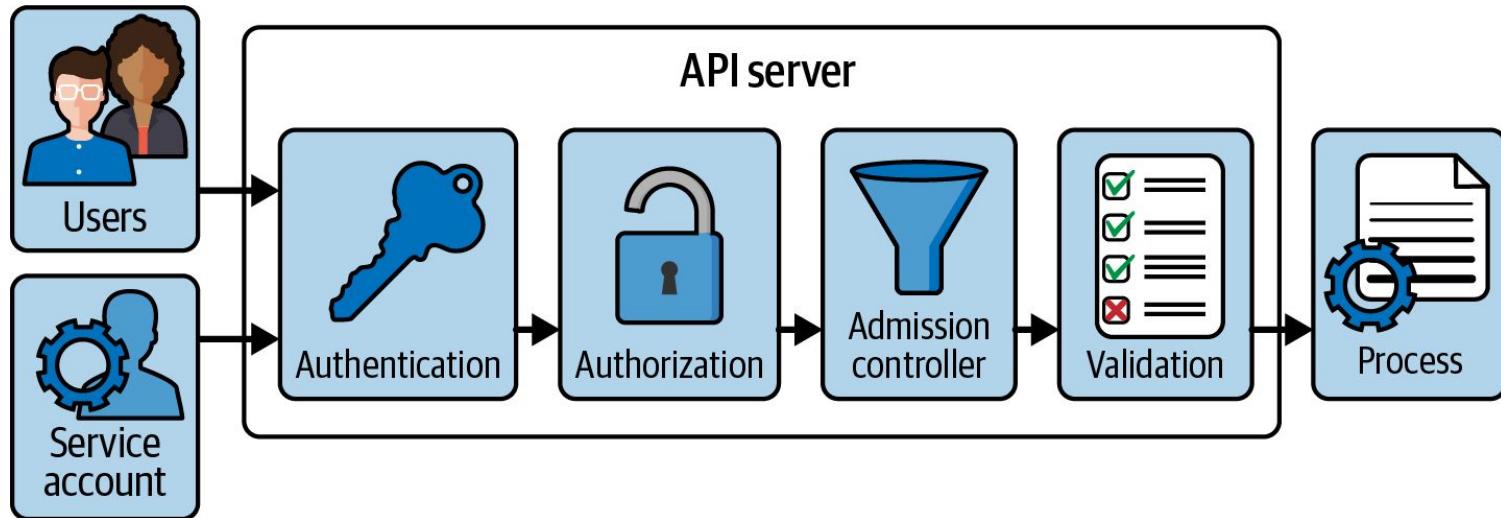


Authentication and Authorization

Access control for the Kubernetes API

Processing a Request to the API Server

Any client request to the API server needs to pass four phases



API Request Processing Phases

Every request has to pass those phases in the following order

- *Authentication*: Validates the identity of the caller using a supported authentication strategy, e.g. client certificates or bearer tokens.
- *Authorization*: Determines if the identity provided in the first stage can access the verb and HTTP path request. This is usually implemented with RBAC.
- *Admission Control*: Verifies if the request is well-formed and potentially needs to be modified before the request is processed.
- *Validation*: Ensures that the resource included in the request is valid (could also be implemented as part of admission control).

Authentication via Credentials in Kubeconfig

The kubeconfig file at `$HOME/.kube/config` is evaluated by `kubectl`

```
apiVersion: v1
kind: Config
clusters:
- cluster:
  certificate-authority: /Users/bmuschko/.minikube/ca.crt
  extensions:
  - extension:
    last-update: Thu, 04 May 2023 08:48:09 MDT
    provider: minikube.sigs.k8s.io
    version: v1.30.1
  name: cluster_info
  server: https://127.0.0.1:58936
name: minikube
contexts:
- context:
  cluster: minikube
  user: bmuschko
  name: bmuschko
current-context: bmuschko
users:
- name: bmuschko
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

API server endpoint for connecting with cluster

Groups access parameters under a convenient name

Client certificate for user

Managing Kubeconfig and Current Context

Interaction with `kubeconfig` configuration

```
$ kubectl config view ←  
apiVersion: v1  
kind: Config  
current-context: bmuschko  
...
```

Renders the contents of
the kubeconfig file

```
$ kubectl config current-context ←  
bmuschko
```

Shows the
currently-selected context

```
$ kubectl config use-context johndoe ←  
Switched to context "johndoe".
```

Switches to the context
defined in the kubeconfig

```
$ kubectl config current-context  
johndoe
```

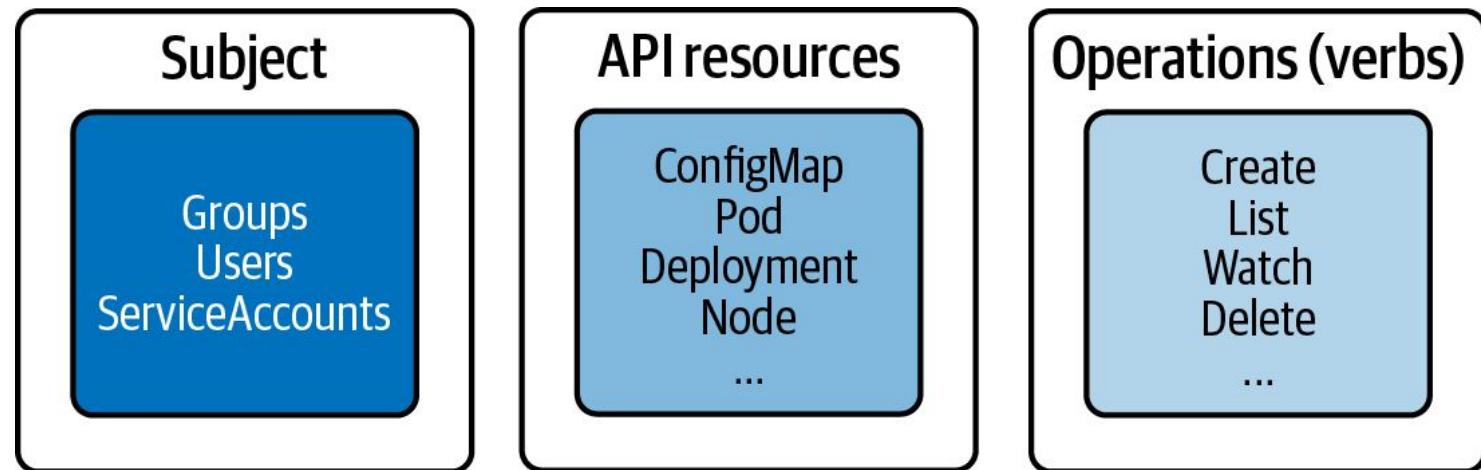
What is RBAC?

Restricting access control for clients interacting with API server

- Defines policies for users, groups, and processes by allowing or disallowing access to manage API resources.
- Enabling and configuring RBAC is mandatory for any organization with a strong emphasis on security.
- *Example:* “The human user John is only allowed to list and create Pods and Deployments, but nothing else.”

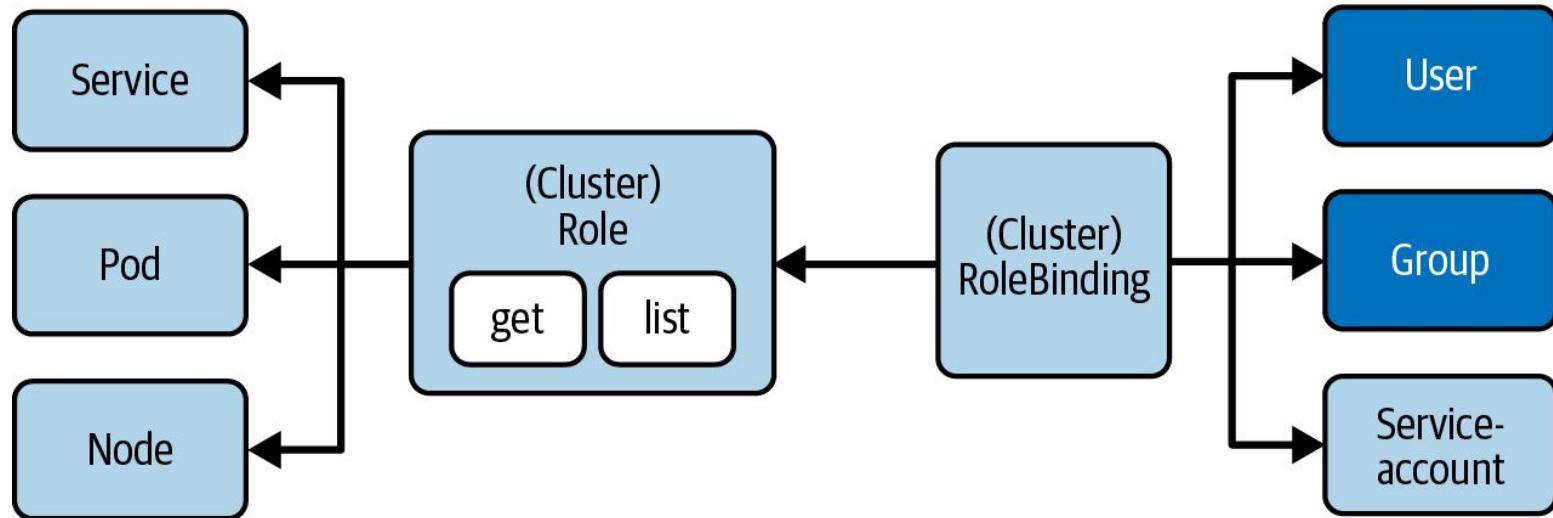
RBAC High-Level Overview

Three key elements for understanding concept



Involved RBAC Primitives

Restrict access to certain operations of API resources for subjects





Namespace and Cluster-Wide Permissions

Defining permission scopes

- The *Role* maps API resources to verbs for a single namespace.
- The *RoleBinding* maps a reference of a Role to one or many subjects for a single namespace.
- To apply permissions across all namespaces of a cluster or cluster-wide resources, use the corresponding API primitives *Cluster* and *ClusterRoleBinding*. The configuration options of the manifests are the same.

Creating a Role with Imperative Approach

Defines verbs and resources in comma-separated list

```
$ kubectl create role read-only --verb=list,get,watch ↴  
  --resource=pods,deployments,services  
role.rbac.authorization.k8s.io/read-only created
```

Resources: Primitives the operations should apply to

Operations: Only allow listing, getting, watching

Role YAML Manifest

Can define a list of rules in an array

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: read-only
rules:
- apiGroups: []
  resources: ["pods", "services"]
  verbs: ["list", "get", "watch"]
- apiGroups: ["apps"] ←
  resources: ["deployments"]
  verbs: ["list", "get", "watch"]
```

Resources with groups need
to be spelled out explicitly (in
this case apps/Deployment)

Getting Role Details

Maps objects of a Kubernetes primitive to verbs

```
$ kubectl describe role read-only
Name:           read-only
Labels:         <none>
Annotations:   <none>
PolicyRule:
  Resources      Non-Resource URLs  Resource Names  Verbs
  -----          -----            -----          -----
  pods           [ ]                [ ]            [list get watch]
  services        [ ]                [ ]            [list get watch]
  deployments.apps [ ]              [ ]            [list get watch]
```

Creating a RoleBinding with Imperative Approach

Maps subject to Role

```
$ kubectl create rolebinding read-only-binding --role=read-only --user=johndoe  
rolebinding.rbac.authorization.k8s.io/read-only-binding created
```

Subject: Binds a user to the Role

Role: Reference the name of the object

RoleBinding YAML Manifest

Roles can be mapped to multiple subjects if needed

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-only-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: read-only
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: johndoe
```

← Reference to Role

← Reference to User

Getting RoleBinding Details

Only shows mapping between Role and subjects

```
$ kubectl describe rolebinding read-only-binding
Name:           read-only-binding
Labels:         <none>
Annotations:   <none>
Role:
  Kind:  Role
  Name:  read-only
Subjects:
  Kind  Name      Namespace
  ----  ----      -----
  User  johndoe
```

What is a Service Account?

Non-human request to Kubernetes API from a process

- Processes running outside of Kubernetes or processes running inside of a container may need to interact with the Kubernetes API. A Service Account allows for authenticating with the API server through an authentication token.
- Authorization is controlled through RBAC and assigned to the Service Account.
- If not assigned explicitly, a Pod uses the `default` Service Account. The `default` Service Account has the same permissions as an unauthenticated user.
- *Example:* “I have a CI/CD process that retrieves Pod information by making calls to the Kubernetes API.”

Service Account as Subject

Pod assigns the Service Account by name



Using the kubernetes Service

Convenient way to get API server endpoint from within the cluster

```
$ kubectl get service kubernetes
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	13h

```
$ kubectl run api-call --image=alpine:3.16.2 -it --rm <  
--restart=Never -- https://kubernetes.default.svc.cluster.local/<  
api/v1/namespaces/k97/pods
```

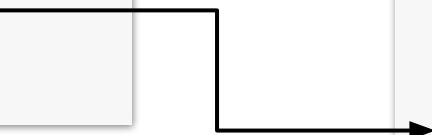
API endpoint for
listing all Pods in
namespace k97

Service Account YAML Manifest

In a Pod manifest, refer to the Service Account by name

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-api
  namespace: k97
```

```
apiVersion: v1
kind: Pod
metadata:
  name: list-pods
  namespace: k97
spec:
  serviceAccountName: sa-api
  ...
```





Accessing Service Account Authentication Token

Automounted at specific Volume mount path

```
$ kubectl describe pod list-pods -n k97
...
Containers:
  pods:
    ...
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from  ↪
        kube-api-access-xnkwd (ro)
      ...
$ kubectl exec -it list-pods -n k97 -- /bin/sh
# cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1N...
# exit
```

Using Service Account in Pod

Enables authentication token auto-mounting by default

```
apiVersion: v1
kind: Pod
metadata:
  name: list-pods
  namespace: k97
spec:
  serviceAccountName: sa-api
  containers:
    - name: pods
      image: alpine/curl:3.14
      command: ['sh', '-c', 'while true; do curl -s -k -m 5 -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://kubernetes.default.svc.cluster.local/api/v1/namespaces/k97/pods; sleep 10; done']
```

Assigned name of
Service Account

List all Pods in the
namespace k97
via an API call

RoleBinding YAML Manifest

Default RBAC policies don't span beyond kube-system

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: serviceaccount-pod-rolebinding
  namespace: k97
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: list-pods-clusterrole
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: ServiceAccount
  name: sa-api
```

Maps the Service
Account as a
subject



Exercise

Regulating Access to API Resources with RBAC





Resource Management

Resource requests and limits, resource quotas, limit ranges

Managing Resources for Objects and Namespaces

It's best practice to make a statement about resource consumption

- Containers can define a minimum amount of resources needed to run the application, as well as the maximum amount of resources the application is allowed to consume.
- Application developers should determine the right-sizing with load tests or at runtime by monitoring the resource consumption.
- Enforcement of resources can be constrained on a namespace-level with a ResourceQuota.
- A LimitRange is a policy that constrains the resource allocations for a single object (e.g., for a Pod or PersistentVolumeClaim).



Resource Units in Kubernetes

CPU units and memory as fixed-point number or power-of-two equivalents

- Kubernetes measures CPU resources in millicores and memory resources in bytes. That's why you might see resources defined as 600m or 100Mib.
- For a deep dive on those resource units, it's worth cross-referencing the section "[Resource units in Kubernetes](#)" in the official documentation.



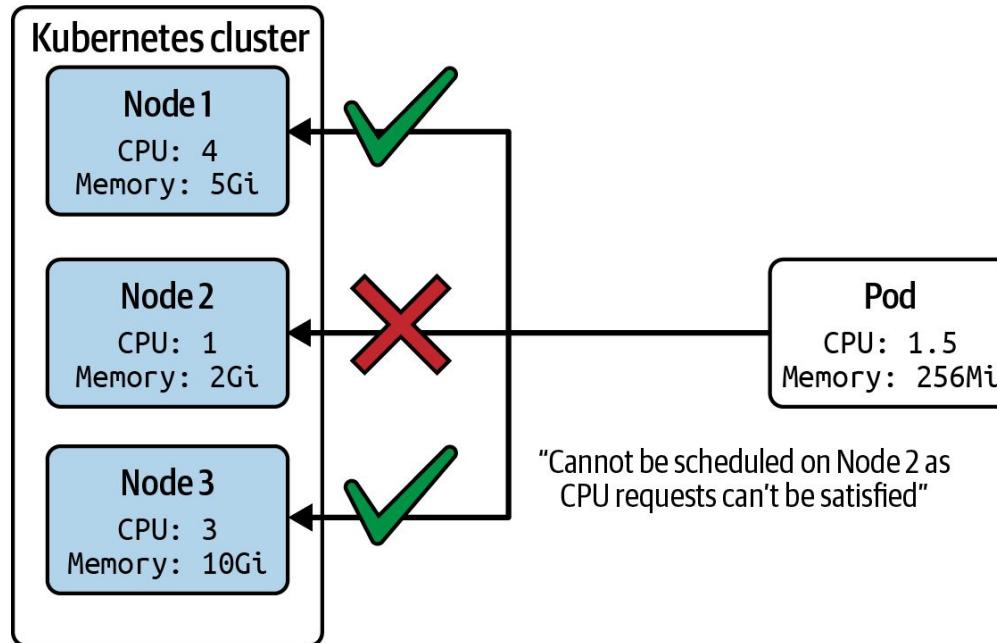
Container Resource Requests

Definition and runtime effects

- Containers can define a minimum amount of resources needed to run the application via `spec.containers[] .resources.requests`
- Resource types include CPU, memory, huge page, ephemeral storage.
- If Pod cannot be scheduled on any node due to insufficient resource availability, you may see a `PodExceedsFreeCPU` or `PodExceedsFreeMemory` status.

Example Scheduling Scenario

Node's resource capacity needs to be able to fulfill it



Container Resource Requests Options

Configuration options and example values

YAML Attribute	Description	Example Value
<code>spec.containers[].resources.requests.cpu</code>	CPU resource type	500m (five hundred millicpu)
<code>spec.containers[].resources.requests.memory</code>	Memory resource type	64Mi (2^26 bytes)
<code>spec.containers[].resources.requests.hugepages-<size></code>	Huge page resource type	hugepages-2Mi: 60Mi
<code>spec.containers[].resources.requests.ephemeral-storage</code>	Ephemeral storage resource type	4Gi

Container Resource Requests YAML Manifest

Minimum amount of resources defined for a container

```
apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
  containers:
    - name: business-app
      image: bmuschko/nodejs-business-app:1.0.0
      ports:
        - containerPort: 8080
      resources:
        requests:
          memory: "256Mi"
          cpu: "1"
```

Container Resource Limits

Definition and runtime effects

- Containers can define a maximum amount of resources allowed to be consumed by the application via `spec.containers[] .resources.limits`
- Resource types include CPU, memory, huge page, ephemeral storage.
- Container runtime decides how to handle situation where application exceeds allocated capacity, e.g. termination of application process.



Container Resource Limits Options

Configuration options and example values

YAML Attribute	Description	Example Value
<code>spec.containers[].resources.limits.cpu</code>	CPU resource type	500m (five hundred millicpu)
<code>spec.containers[].resources.limits.memory</code>	Memory resource type	64Mi (2^26 bytes)
<code>spec.containers[].resources.limits.hugepages-<size></code>	Huge page resource type	hugepages-2Mi: 60Mi
<code>spec.containers[].resources.limits.ephemeral-storage</code>	Ephemeral storage resource type	4Gi

Container Resource Limits YAML Manifest

Do not allow more than the allotted resource amounts

```
apiVersion: v1
kind: Pod
metadata:
  name: rate-limiter
spec:
  containers:
    - name: business-app
      image: bmuschko/nodejs-business-app:1.0.0
      ports:
        - containerPort: 8080
  resources:
    limits:
      memory: "512Mi"
      cpu: "2"
```



Pod Scheduling Details

After scheduling a Pod, you can check on runtime information

```
$ kubectl get pod rate-limiter -o yaml | grep nodeName:  
nodeName: worker-3  
  
$ kubectl describe node worker-3  
...  
Non-terminated Pods:           (3 in total)  
  Namespace          Name            CPU Requests  CPU Limits  ...  
  -----            ----  
  default           rate-limiter   1250m (62%)  0 (0%)    ...
```

Container Resource Requests/Limits YAML Manifest

It is considered best practice to define requests and limits for every container

```
spec:  
  containers:  
    - name: business-app  
      resources:  
        requests:  
          memory: "256Mi"  
          cpu: "1"  
        limits:  
          memory: "512Mi"  
          cpu: "2"
```

Exercise

Defining container
resource requests and
limits



What is a ResourceQuota?

Defines resource constraints per namespace

- Constraints that limit aggregate resource consumption or limit the quantity of objects (e.g., number of Pods or Secrets) that can be created.
- `requests.cpu/requests.memory`: Across all pods in a non-terminal state, the sum of CPU/memory requests cannot exceed this value.
- `limits.cpu/limits.memory`: Across all pods in a non-terminal state, the sum of CPU/memory limits cannot exceed this value.

ResourceQuota YAML Manifest

Limits # of objects and aggregate container resource consumption

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: awesome-quota
  namespace: team-awesome
spec:
  hard:
    pods: 2
    requests.cpu: "1"
    requests.memory: 1024Mi
    limits.cpu: "4"
    limits.memory: 4096Mi
```



No Requests/Limits Definition

Scheduler rejects the creation of the object

```
$ kubectl create -f nginx-pod.yaml -n team-awesome
Error from server (Forbidden): error when creating "nginx-pod.yaml": ←
pods "nginx" is forbidden: failed quota: awesome-quota: must specify ←
limits.cpu,limits.memory,requests.cpu,requests.memory
```



Exceeds Requests/Limits Definition

Scheduler rejects the creation of the object

```
$ kubectl create -f nginx-pod.yaml -n team-awesome
Error from server (Forbidden): error when creating "nginx-pod.yaml": ←
pods "nginx" is forbidden: exceeded quota: awesome-quota, requested: ←
pods=1,requests.cpu=500m,requests.memory=512Mi, used: pods=2, ←
requests.cpu=1,requests.memory=1024Mi, limited: pods=2, ←
requests.cpu=1,requests.memory=1024Mi
```

Inspecting a ResourceQuota

Renders consumed resources and defined hard limits

```
$ kubectl describe resourcequota awesome-quota -n team-awesome
```

Name:	awesome-quota	
Namespace:	team-awesome	
Resource	Used	Hard
-----	----	----
limits.cpu	2	4
limits.memory	2048m	4096m
pods	2	2
requests.cpu	1	1
requests.memory	1024m	1024m

Exercise

Defining a resource quota
for a namespace



What is a LimitRange?

Constrains or defaults the resource allocations for an object type

- Enforces minimum and maximum compute resources usage per Pod or container in a namespace.
- Enforces minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforces a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically injects them to containers at runtime.

LimitRange YAML Manifest

Sets defaults, minimum and maximum CPU allocation for containers

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
    - default:
        cpu: 500m
      defaultRequest:
        cpu: 500m
    max:
      cpu: "1"
    min:
      cpu: 100m
  type: Container
```

Defines default limits

Defines default requests

Minimum and maximum resource range

Automatically Uses Container Defaults for Pod

Container doesn't provide any resource requests/limits

```
$ kubectl create -f pod.yaml
pod/rate-limiter created

$ kubectl describe pod rate-limiter
...
Containers:
  business-app:
    ...
    Limits:
      cpu: 500m
    Requests:
      cpu: 500m
    ...
  
```



Rejected Pod Creation for Exceeded Limits

Requests a higher maximum for CPU resources than defined by LimitRange

```
$ kubectl create -f pod.yaml
Error from server (Forbidden): error when creating "pod.yaml": pods "rate-limiter" is forbidden: maximum cpu usage per Container is 1,
but limit is 2
```

Exercise

Creating a Pod conforming
with LimitRange in
namespace





ConfigMaps and Secrets

Defining and consuming configuration data

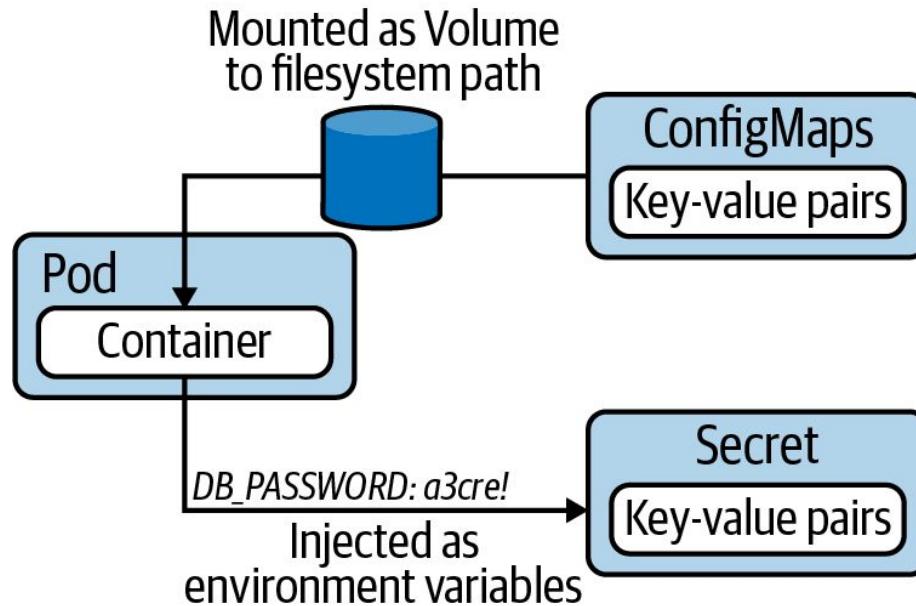
Defining Configuration Data

Control runtime behavior with centralized information

- Stored as key-value pairs in dedicated Kubernetes primitives with a lifecycle decoupled from consuming Pod.
- *ConfigMap*: Plain-text values suited for configuring applications e.g. flags, or URLs.
- *Secret*: Base64-encoded values suited for storing sensitive data like passwords, API keys, or SSL certificates. **Values are not encrypted!**
- ConfigMap and Secret objects are stored in etcd in unencrypted form by default. Encryption of data in etcd is configurable.

Consuming Configuration Data

Applicable to ConfigMap and Secret



Creating a ConfigMap with Imperative Approach

Key-value pairs can be parsed from different sources

```
$ kubectl create configmap db-config --from-literal=db=staging  
configmap/db-config created  
  
$ kubectl create configmap db-config --from-env-file=config.env  
configmap/db-config created  
  
$ kubectl create configmap db-config --from-file=config.txt  
configmap/db-config created  
  
$ kubectl create configmap db-config --from-file=app-config  
configmap/db-config created
```

Source Options for Data Parsed by a ConfigMap

Configuration options and example values

Option	Description	Example Value
--from-literal	Literal values, which are key-value pairs as plain text.	--from-literal=locale=en_US
--from-env-file	A file that contains key-value pairs and expects them to be environment variables.	--from-env-file=config.env
--from-file	Huge page resource type.	--from-file=app-config.json
--from-file	A directory with one or many files.	--from-file=config-dir

Creating a ConfigMap from Literal Values

Imperative creation by pointing to plain-text key-value pairs

```
$ kubectl create configmap db-config  
--from-literal=DB_HOST=mysql-service  
--from-literal=DB_USER=backend  
configmap/db-config created
```

Parameter can
be repeated
multiple times



ConfigMap YAML Manifest

Key-value pair definition under data attribute

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: mysql-service
  DB_USER: backend
```

Follows typical
naming conventions
for environment
variables



Consuming a ConfigMap as Environment Variables

Controlling runtime behavior with the help of simple values

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      envFrom:
        - configMapRef:
            name: db-config
```

```
$ kubectl exec -it backend -- env
DB_HOST=mysql-service
DB_USER=backend
```

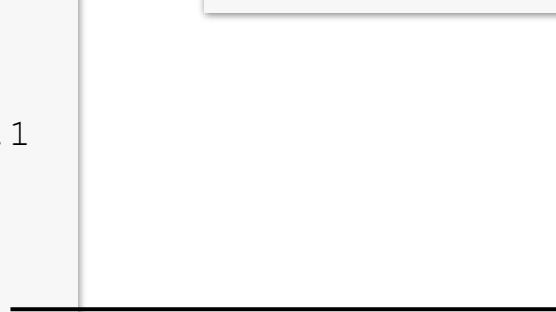


Reassigning Environment Variable Keys

Keys can be renamed or reformatted to make them suitable for consumption

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      env:
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: backend-config
              key: database_url
```

```
$ kubectl exec -it backend -- env
DATABASE_URL=jdbc:postgresql://localhost/test
```



Creating a ConfigMap from a JSON file

Imperative creation by pointing to file

```
$ kubectl create configmap db-config --from-file=db.json  
configmap/db-config created
```

db.json

```
{  
  "db": {  
    "host": "mysql-service",  
    "user": "backend"  
  }  
}
```

ConfigMap YAML Manifest

Key-value pair definition under data attribute

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db.json: |
    {
      "db": {
        "host": "mysql-service",
        "user": "backend"
      }
    }
```

The file name
becomes the key, the
contents of the file
become the value

Consuming a ConfigMap as a Volume

A good choice for processing a machine-readable configuration file

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - name: backend
      image:
        bmuschko/web-app:1.0.1
      volumeMounts:
        - name: db-config-volume
          mountPath: /etc/config
  volumes:
    - name: db-config-volume
      configMap:
        name: db-config
```



```
$ kubectl exec -it backend -- /bin/sh
# ls -l /etc/config
db.json
# cat /etc/config/db.json
{
  "db": {
    "host": "mysql-service",
    "user": "backend"
  }
}
```



Consuming Changed ConfigMap Data

Containers will not refresh consumed data upon a change

- The key-value pairs of a ConfigMap can be changed for a live object.
- Changed ConfigMap key-value pairs consumed by containers will not be reloaded automatically for an already-running Pod.
- The application code needs to implement logic to either reload the configuration data periodically or on-demand.

Exercise

Creating and consuming a
ConfigMap





Creating a Secret with Imperative Approach

Parsed values are base64-encoded automatically

```
$ kubectl create secret generic db-creds --from-literal=pwd=s3cre!
secret/db-creds created

$ kubectl create secret docker-registry my-secret <
  --from-file=.dockerconfigjson=~/docker/config.json
secret/my-secret created

$ kubectl create secret tls accounting-secret --cert=accounting.crt <
  --key=accounting.key
secret/accounting-secret created
```

Imperative CLI Options for Creating a Secret

You will need to provide one of the options

Option	Description	Type
generic	Creates a secret from a file, directory, or literal value.	Opaque
docker-registry	Creates a secret for use with a Docker registry, e.g. to pull images from a private registry when requested by a Pod.	kubernetes.io/dockercfg
tls	Creates a TLS secret.	kubernetes.io/tls

Source Options for Data Parsed by a Generic Secret

Configuration options and example values

Option	Description	Example Value
--from-literal	Literal values, which are key-value pairs as plain text.	--from-literal=password=secret
--from-env-file	A file that contains key-value pairs and expects them to be environment variables.	--from-env-file=config.env
--from-file	Huge page resource type.	--from-file=id_rsa=~/ssh/id_rsa
--from-file	A directory with one or many files.	--from-file=config-dir

Specialized Secret Types

Can be set with `--type` CLI option, or the `type` attribute in manifest ([docs](#))

Type	Description
kubernetes.io/basic-auth	Credentials for basic authentication
kubernetes.io/ssh-auth	Credentials for SSH authentication
kubernetes.io/service-account-token	ServiceAccount token
bootstrap.kubernetes.io/token	Node bootstrap token data

Creating a Secret from Literal Values

Imperative command will base64-encode values automatically

```
$ kubectl create secret generic db-creds --from-literal=pwd=s3cre!  
secret/db-creds created
```

← Plain-text value

Secret YAML Manifest

Assigned values need to be provided in base64-encoded form

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

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

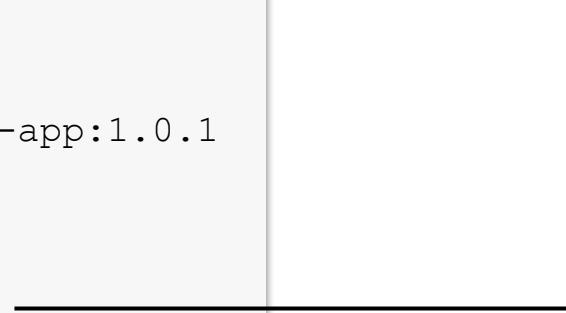


Consuming a Secret as Environment Variables

Accessed values are base64-decoded

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: bmuschko/web-app:1.0.1
      name: backend
      envFrom:
        - secretRef:
            name: mysecret
```

```
$ kubectl exec -it backend -- env
pwd=s3cre!
```



Creating a Secret from a SSH Private Key File

Imperative creation by pointing to file

```
$ kubectl create secret generic secret-ssh-auth  
  --type=kubernetes.io/ssh-auth --from-file=ssh-privatekey  
secret/secret-ssh-auth created
```

ssh-privatekey

```
-----BEGIN RSA PRIVATE KEY-----  
Proc-Type: 4,ENCRYPTED  
DEK-Info:  
AES-128-CBC,8734C9153079F2E8497C8075289EBBF1  
...  
-----END RSA PRIVATE KEY-----
```

Consuming a Secret as a Volume

A good choice for processing a machine-readable configuration file

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - name: backend
      image:
        bmuschnko/web-app:1.0.1
      volumeMounts:
        - name: ssh-volume
          mountPath: /var/app
          readOnly: true
  volumes:
    - name: ssh-volume
      secret:
        secretName:
secret-ssh-auth
```



```
$ kubectl exec -it backend -- /bin/sh
# ls -l /var/app
ssh-privatekey
# cat /var/app/ssh-privatekey
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info:
AES-128-CBC,8734C9153079F2E8497C8075289E
BBF1
...
-----END RSA PRIVATE KEY-----
```

Plain-Text Secret Values

The `stringData` attribute allows for plain-text values

`secret.yaml`

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
```

```
kubectl create -f
secret.yaml
```

Live object

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: YWRtaW4=
  password: dDBwLVNlY3JldA==
```



Secrets Are Not Really Secret

Techniques that help with making Secrets more secure

- Secret objects are stored in etcd in unencrypted form by default. Encryption of data in etcd is [configurable](#).
- Define RBAC permissions to only allow developers to create, view, and modify Secret objects in dedicated namespaces. An even stricter policy could only allow administrators to manage Secrets.
- Use Kubernetes-external, third-party Secrets services for managing sensitive data, e.g. AWS Secrets Manager or HashiCorp Vault. You can consume the data with the help of the [External Secrets Operator](#).



Exercise

Creating and consuming a
Secret





Security Context

Privilege and access control for a Pod

What is a Security Context?

Defines security settings for containers of a Pod

- Modeled as a set of attributes within the Pod specification, e.g. for configuring Linux capabilities, file access control, privileges to parent process or host system.
- Can be applied to all containers in Pod with `spec.securityContext` or for individual containers with `spec.containers[].securityContext`
- Some of the attributes on the Pod- and container-level are the same. Container-level definition of an attribute takes precedence.
- *Example:* “This container needs to run with a non-root user.”



Security Context API

All attributes can be discovered in documentation for the Kubernetes version

API Type	Description
<u>PodSecurityContext</u>	Defines Pod-level security attributes.
<u>SecurityContext</u>	Defines container-level security attributes.

Defining a Security Context

Pod- and/or container-level definition

```
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
spec:
  securityContext:
    fsGroup: 3000
  volumes:
  - name: data-vol
    emptyDir: {}
  containers:
  - image: nginx:1.18.0
    name: secured-container
    volumeMounts:
    - name: data-vol
      mountPath: /data/app
```

Defined on the Pod-level (applies to all containers)

Inspecting the Runtime Behavior

Rendering the file system Linux group ID

```
$ kubectl exec secured-pod -it -- /bin/sh
# cd /data/app
# touch hello.txt
# ls -l
total 0
-rw-r--r-- 1 root 3000 0 Apr 30 18:27 hello.txt
# exit
```



File system group ID is
assigned automatically

Overriding a Value on the Container-Level

Container-level definition wins

```
apiVersion: v1
kind: Pod
metadata:
  name: non-root-success
spec:
  securityContext:
    runAsNonRoot: false
  containers:
    - image: bitnami/nginx:1.23.1
      name: nginx
      securityContext:
        runAsNonRoot: true
```



Takes precedence

Inspecting the Runtime Behavior

Rendering the current user ID and group ID

```
$ kubectl exec non-root-success -it -- /bin/sh
$ id
uid=1001 gid=0(root) groups=0(root)
$ exit
```



Non-root user ID



Exercise

Defining a Security Context for a Pod



Service & Networking





Services

Providing a stable network endpoint to Pods

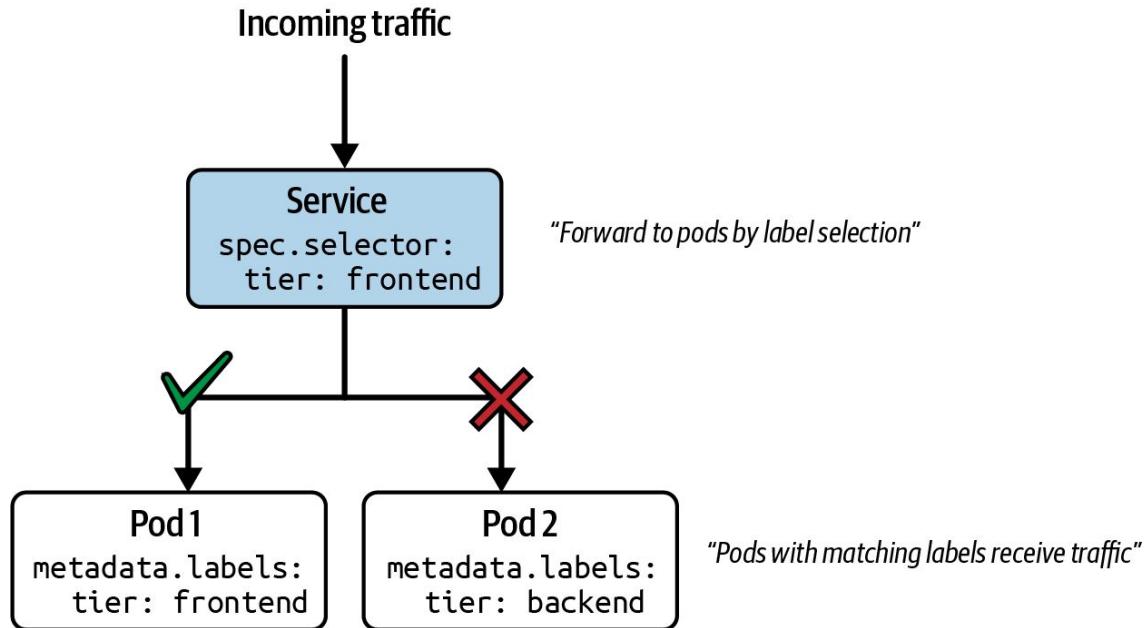
What is a Service?

Stable endpoint for routing traffic to a set of Pods

- A Pod's virtual IP address is not considered stable over time. A Pod restart leases a new IP address.
- Building a microservices architecture on Kubernetes requires network endpoints that are stable over time.
- A Service provides discoverable names and load balancing to a set of Pods.

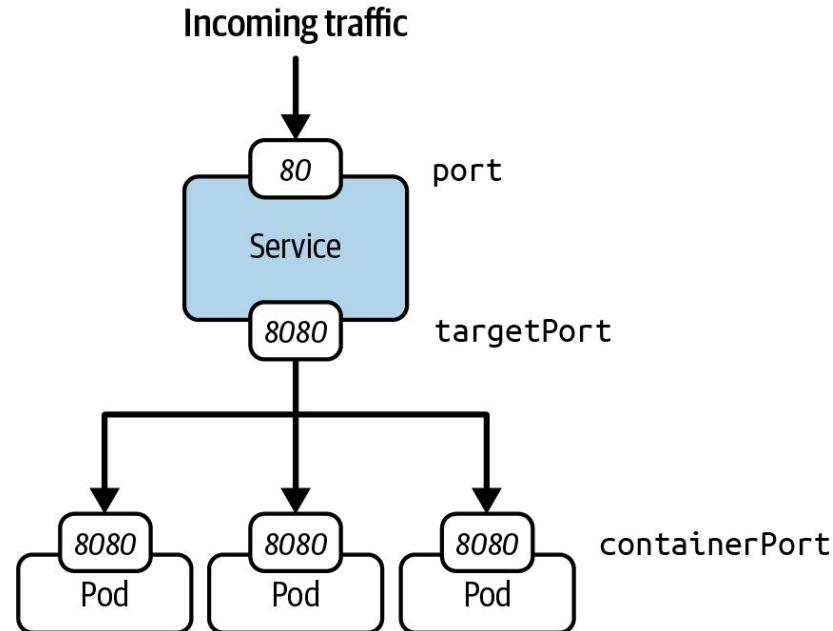
Request Routing

"How does a Service decide which Pod to forward the request to?"



Port Mapping

"How to map the Service port to the container port in a Pod?"



Creating a Service with Imperative Approach

Needs to provide the service type and ports

```
$ kubectl run echoserver  
  --image=k8s.gcr.io/echoserver:1.10  
  --port=8080 ← Exposed container port  
pod/echoserver created
```



```
$ kubectl create  
  service clusterip  
  echoserver  
  --tcp=80:8080 ← Incoming and outgoing port  
service/echoserver created
```



```
$ kubectl run echoserver  
  --image=k8s.gcr.io/echoserver:1.10  
  --port=8080  
  --expose ← Shortcut for creating a Pod + Service  
service/echoserver created  
pod/echoserver created
```

Service YAML Manifest

Needs to provide the service type and ports

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  selector:
    app: echoserver ← Label selection for Pods
  ports:
  - port: 80
    targetPort: 8080 ← Mapping of incoming to outgoing port
    protocol: TCP
```

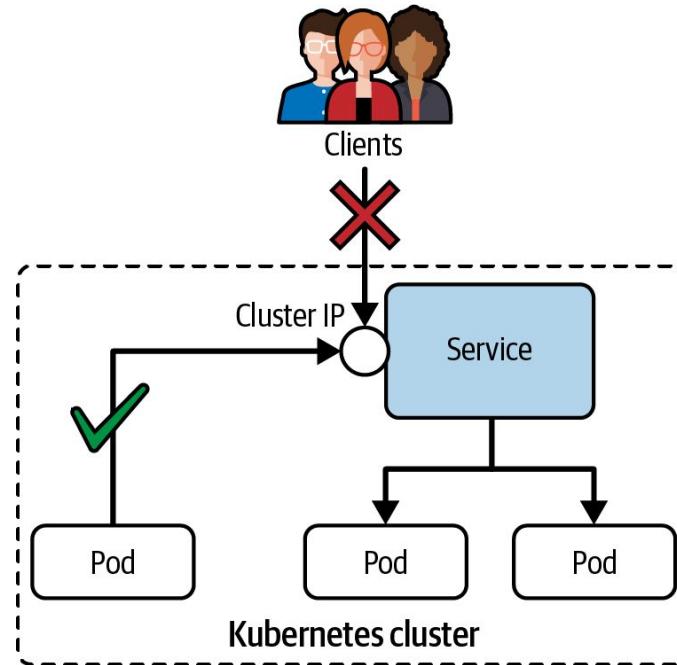
Service Types

The following table shows a limited list of Service types

Type	Description
ClusterIP	Exposes the service on a cluster-internal IP. Only reachable from Pods within the cluster. Default if value for <code>spec.type</code> has not been assigned in manifest.
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.

ClusterIP Service Type

Only reachable from within the cluster, e.g. another Pod



ClusterIP Service YAML Manifest at Runtime

ClusterIP service type has been assigned if not already provided

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: ClusterIP
  clusterIP: 10.96.254.0
  selector:
    app: echoserver
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
```

Accessing a ClusterIP Service

Use the combination of cluster IP and incoming service port

```
$ kubectl get pod,service
NAME                  READY   STATUS    RESTARTS   AGE
pod/echoserver        1/1     Running   0          23s

NAME            TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/echoserver  ClusterIP  10.96.254.0  <none>       8080/TCP  8s

$ kubectl run tmp --image=busybox --restart=Never -it --rm <
  -- wget 10.96.254.0:8080
Connecting to 10.96.254.0:8080 (10.96.254.0:8080)
...

```

Discovering the Service by Environment Variables

Application code running in container can get access to Service endpoint

```
$ kubectl exec -it echoserver -- env  
ECHOSERVER_SERVICE_HOST=10.96.254.0  
ECHOSERVER_SERVICE_PORT=8080  
...
```

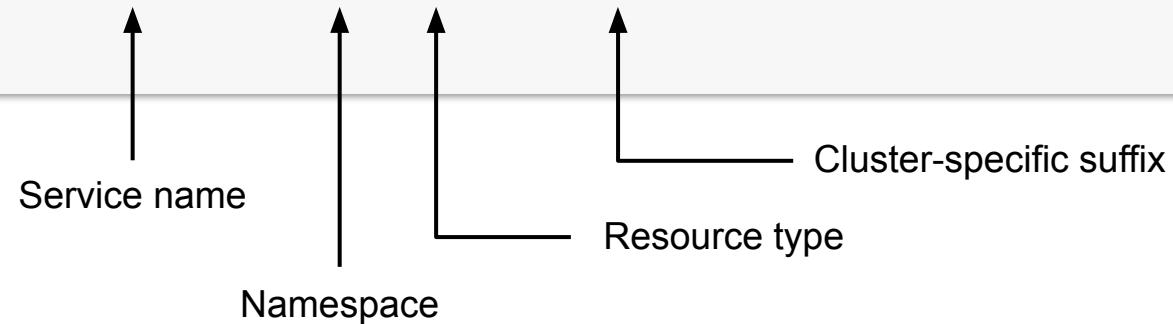
Discovering the Service by DNS Lookup

Kubernetes' DNS service maps cluster IP address to Service name

```
$ kubectl run tmp --image=busybox --restart=Never -it --rm <  
-- wget echoserver.default.svc.cluster.local:8080
```

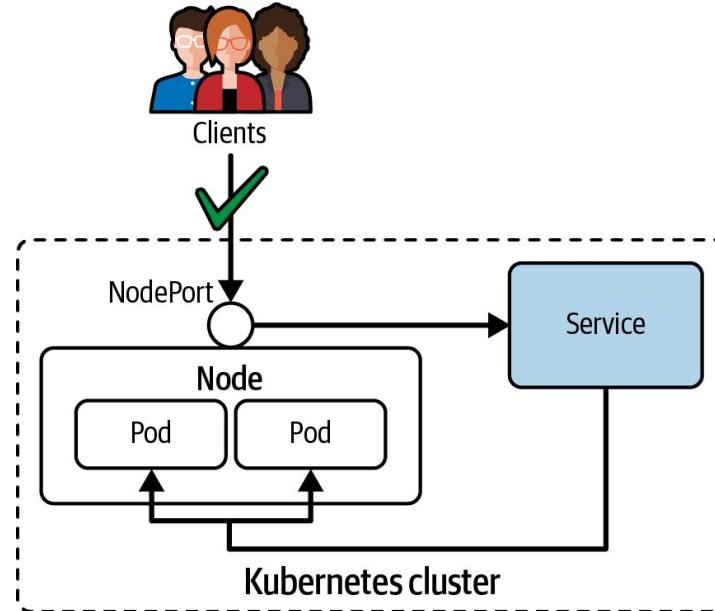
Connecting to echoserver.default.svc.cluster.local:8080 (10.96.254.0:8080)

...



NodePort Service Type

Can be resolved from outside of the Kubernetes cluster



NodePort Service YAML Manifest at Runtime

Service type is NodePort, node port has been assigned

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: NodePort
  clusterIP: 10.96.254.0
  selector:
    app: echoserver
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30158
    protocol: TCP
```

Accessing a NodePort Service

Use the combination of node IP and statically-assigned port

```
$ kubectl get pod,service
```

NAME	READY	STATUS	RESTARTS	AGE
pod/echoserver	1/1	Running	0	23s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	
AGE					
service/echoserver	NodePort	10.96.254.0	<none>	8080:30158/TCP	8s

```
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="InternalIP")].address }'
```

192.168.64.15

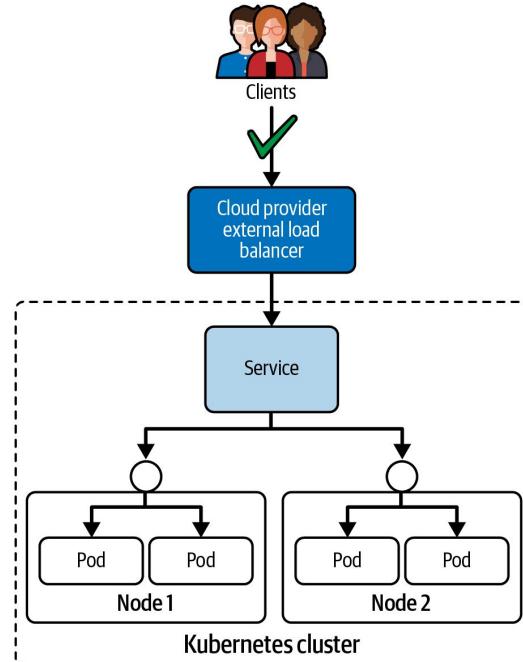
```
$ wget 192.168.64.15:30158
```

Connecting to 192.168.64.15:30158... connected.

...

LoadBalancer Service Type

Routes traffic from existing, external load balancer to Service



LoadBalancer Service YAML Manifest at Runtime

Service type is LoadBalancer, external IP address has been assigned

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: LoadBalancer
  clusterIP: 10.96.254.0
  loadBalancerIP: 10.109.76.157
  selector:
    app: echoserver
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30158
    protocol: TCP
```

Accessing a LoadBalancer Service

Use the combination of external IP and statically-assigned port

```
$ kubectl get pod,service
```

NAME	READY	STATUS	RESTARTS	AGE
pod/echoserver	1/1	Running	0	23s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/echoserver	LoadBalancer	10.109.76.157	10.109.76.157	8080:30158/TCP	5s

```
$ wget 10.109.76.157:8080
```

Connecting to 10.109.76.157:8080... connected.

...

Exercise

Routing traffic to Pods
from inside and outside of
a cluster





Troubleshooting Services

Built-in mechanisms for identifying the root cause of a failure

Checking Service Connectivity

Identify Service Type and try to connect to it

```
$ kubectl get service echoserver
NAME            TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
echoserver      ClusterIP  10.96.254.0    <none>          8080/TCP    8s

$ kubectl run tmp --image=busybox --restart=Never -it --rm \
  -- wget echoserver:8080
Connecting to echoserver (10.96.254.0:8080)
wget: can't connect to remote host (10.96.254.0:8080): Connection refused
pod "tmp" deleted
pod tmp terminated (Error)
```



Connection attempt failed

Verifying Endpoints

Value needs to expose targeted IP address and ports of Pods

```
$ kubectl get endpoints echoserver
```

NAME	ENDPOINTS	AGE
echoserver	<none>	63s

Indicates issues with connection to Pods

```
$ kubectl get endpoints echoserver
```

NAME	ENDPOINTS	AGE
echoserver	10.244.0.3:8080,10.244.0.4:8080	63s



Proper configuration of Service provides virtual IP address and ports to Pods

Checking Label Selection and Port Assignment

Value needs to expose targeted IP address and ports of Pods

```
$ kubectl describe service echoserver
Name:           echoserver
Namespace:      default
Labels:          svc=myservice
Annotations:    <none>
Selector:        app=echoserver
Type:           ClusterIP
IP Family Policy: SingleStack
IP Families:    IPv4
IP:             10.111.148.223
IPs:            10.111.148.223
Port:           8080-8080  8080/TCP
TargetPort:     8080/TCP
Endpoints:      10.244.0.3:8080,10.244.0.4:8080
Session Affinity: None
Events:
```

```
$ kubectl describe pod echoserver
...
Labels:          app=echoserver
Containers:      echoserver:
                  Port:  8080/TCP
```

Checking Network Policies

Does any network policy block ingress traffic to Pod?

```
$ kubectl get networkpolicies
```

No resources found in default namespace.

```
$ kubectl get networkpolicies
```

NAME	POD-SELECTOR	AGE
default-deny-ingress	<none>	9s



Potentially blocks
ingress traffic to all
Pods in namespace

Ensuring Pod Functionality

The Pod or container may have a runtime issue

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
echoserver	0/1	ErrImagePull	0	2s

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
echoserver	1/1	Running	0	5s

Failed to pull
container image

Looks successful on
the surface-level



Exercise

Troubleshooting a Service





Ingress

Providing an external network entrypoint to the cluster

What is an Ingress?

Route HTTP(S) traffic from the outside of the cluster to Service(s)

- It's not a specific Service type, nor should it be confused with the Service type LoadBalancer.
- Defines rules for mapping a URL context path to one or many Service objects.
- TLS termination (support for HTTPS) needs to be configured explicitly. By default, only HTTP is supported.

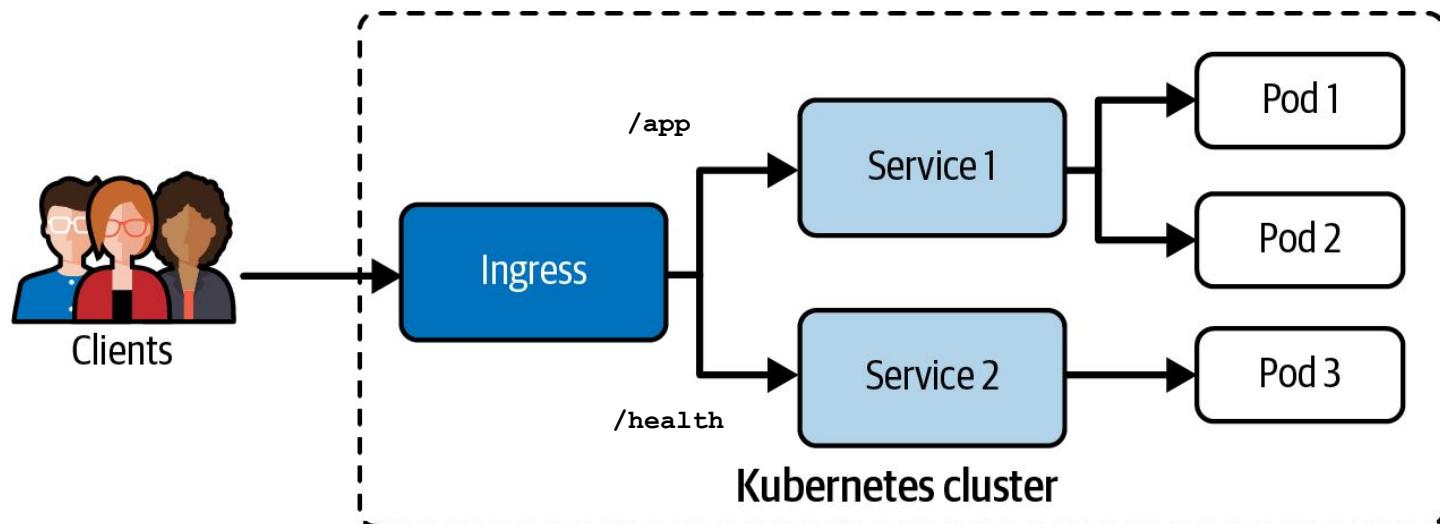
What is an Ingress Controller?

Evaluating Ingress rules

- An Ingress cannot work without an [Ingress controller](#). The Ingress controller evaluates the collection of rules defined by an Ingress that determine traffic routing.
- Multiple Ingress controller can be deployed to a cluster. An Ingress can be configured to pick a specific one via `spec.ingressClassName`.

Ingress Traffic Routing

The context path is mapped to a backend (Service name and port)



Creating an Ingress with Imperative Approach

The rule definition requires intricate knowledge of syntax

```
$ kubectl create ingress  
corellian  
--rule="star-alliance.com/corellian/api=corellian:8080"  
ingress.networking.k8s.io/corellian created
```

Ingress YAML Manifest

Allows for defining multiple rules that map to backend

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: corellian
spec:
  rules:
  - host: star-alliance.com
    http:
      paths:
      - backend:
          service:
            name: corellian
            port:
              number: 8080
          path: /corellian/api
          pathType: Exact
```

The host definition is
optional

Ingress Rules

Traffic routing is controlled by rules defined on Ingress resource

Type	Example	Description
An optional host	mycompany.abc.com	If a host is provided, then the rules apply to that host. If no host is defined, then all inbound HTTP(S) traffic is handled.
A list of paths	/corellian/api	Incoming traffic must match the host and path to correctly forward the traffic to a Service.
The backend	corellian:8080	A combination of Service name and port.

Path Types

Incoming requests match based on assigned type in rule

Path Type	Rule	Incoming Request
Exact	/corellian/api	Matches /corellian/api but does not match /corellian/test or /corellian/api/.
Prefix	/corellian/api	Matches /corellian/api and /corellian/api/ but does not match /corellian/test.

Listing and Describing Ingresses

Renders hosts, IP addresses, and ports

```
$ kubectl get ingress
NAME      CLASS      HOSTS          ADDRESS      PORTS      AGE
corellian <none>    star-alliance.com  192.168.64.15  80        10m

$ kubectl describe ingress corellian
Name:            corellian
Namespace:       default
Address:         192.168.64.15
Default backend: default-http-backend:80 (<error: \
                                         endpoints "default-http-backend" not found>
Rules:
  Host           Path  Backends
  ----          ----  -----
  star-alliance.com
                           /corellian/api   corellian:8080 (172.17.0.5:8080)
Annotations:        <none>
Events:           <none>
```

Configuring DNS for an Ingress

Making an Ingress convenient to use by end users

- To resolve the Ingress, you'll need to configure DNS entries to the external address.
- You'll need to either configure an A record, or a CNAME record. The [ExternalDNS](#) cluster add-on can help with managing those DNS records for you.
- If you have no domain or are using a local Kubernetes solution, you can add the mapping between IP address and domain name to `/etc/hosts`.

Accessing an Ingress

Use host name, port, and context path

```
$ wget star-alliance.com/corellian/api --timeout=5 --tries=1
--2021-11-30 19:34:57-- http://star-alliance.com/corellian/api
Resolving star-alliance.com (star-alliance.com) ... 192.168.64.15
Connecting to star-alliance.com (star-alliance.com)|192.168.64.15|:80... ↵
connected.
HTTP request sent, awaiting response... 200 OK
...
$ wget star-alliance.com/corellian/api/ --timeout=5 --tries=1
--2021-11-30 15:36:26-- http://star-alliance.com/corellian/api/
Resolving star-alliance.com (star-alliance.com) ... 192.168.64.15
Connecting to star-alliance.com (star-alliance.com)|192.168.64.15|:80... ↵
connected.
HTTP request sent, awaiting response... 404 Not Found
2021-11-30 15:36:26 ERROR 404: Not Found.
```

Exercise

Defining and using an
Ingress





Network Policies

Restricting Pod-to-Pod communication



What is a Network Policy?

Firewall rules for Pod-to-Pod communication

- The installed [Container Network Interface \(CNI\) plugin](#) leases and assigns a new virtual IP address to every Pod when it is created.
- Communication between Pods within the same cluster is unrestricted. You can use a Pod's IP address to communicate with it.
- Network policies implement the principle of least privilege. Only allow Pods to communicate if it is needed to fulfill the requirements of architecture.
- You can disallow and allow network communication with fine-grained rules.

What is an Network Policy Controller?

Evaluating Network Policy rules

- A Network Policy cannot work without a Network Policy controller. The Network Policy controller evaluates the collection of rules defined by a Network Policy.
- One option for such a Network Policy controller is Cilium.

Network Policy Rules

Attributes that form the rules

Attribute	Description
podSelector	Selects the Pods in the namespace to apply the network policy to.
policyTypes	Defines the type of traffic (i.e., ingress and/or egress) the network policy applies to.
ingress	Lists the rules for incoming traffic. Each rule can define <code>from</code> and <code>ports</code> sections.
egress	Lists the rules for outgoing traffic. Each rule can define <code>to</code> and <code>ports</code> sections.

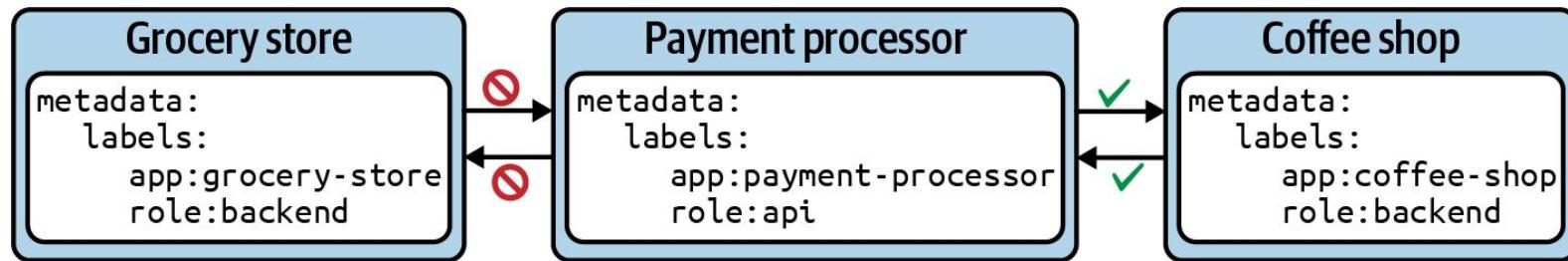
Behavior of to and from Selectors

Ingress and egress sections define the same attributes

Attribute	Description
podSelector	Selects Pods by label(s) in the same namespace as the Network Policy which should be allowed as ingress sources or egress destinations.
namespaceSelector	Selects namespaces by label(s) for which all Pods should be allowed as ingress sources or egress destinations.
namespaceSelector and podSelector	Selects Pods by label(s) within namespaces by label(s).

Example Network Policy

Restricting access to the payment processor Pod from other Pods



Network Policy YAML Manifest

Key-value pairs can be parsed from different sources

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: payment-processor
      role: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: coffeeshop
```

Selects the Pod the policy should apply to by label selection

Allows incoming traffic from the Pod with matching labels within the same namespace



Listing Network Policies

Doesn't provide much details apart from Pod selector labels

```
$ kubectl get networkpolicy
NAME          POD-SELECTOR          AGE
api-allow     app=payment-processor,role=api   83m
```



Rendering Network Policy Details

Ingress and egress rules can be inspected in details

```
$ kubectl describe networkpolicy api-allow
Name:           api-allow
Namespace:      default
Created on:    2020-09-26 18:02:57 -0600 MDT
Labels:         <none>
Annotations:   <none>
Spec:
  PodSelector:    app=payment-processor,role=api
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: app=coffeeshop
  Not affecting egress traffic
  Policy Types: Ingress
```

Verifying the Runtime Behavior

Communication from Pods not matching with labels will be blocked

```
$ kubectl run grocery-store --rm -it --image=busybox ↵
  -l app=grocery-store,role=backend -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.51
Connecting to 10.0.0.51 (10.0.0.51:80)
wget: download timed out
/ # exit
pod "grocery-store" deleted

$ kubectl run coffeeshop --rm -it --image=busybox ↵
  -l app=coffeeshop,role=backend -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.51
Connecting to 10.0.0.51 (10.0.0.51:80)
remote file exists
/ # exit
pod "coffeeshop" deleted
```

Default Deny Network Policies

Network policies are additive

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Selects all Pods in
the namespace

Applies in incoming
and outgoing traffic

Restricting Access to Ports

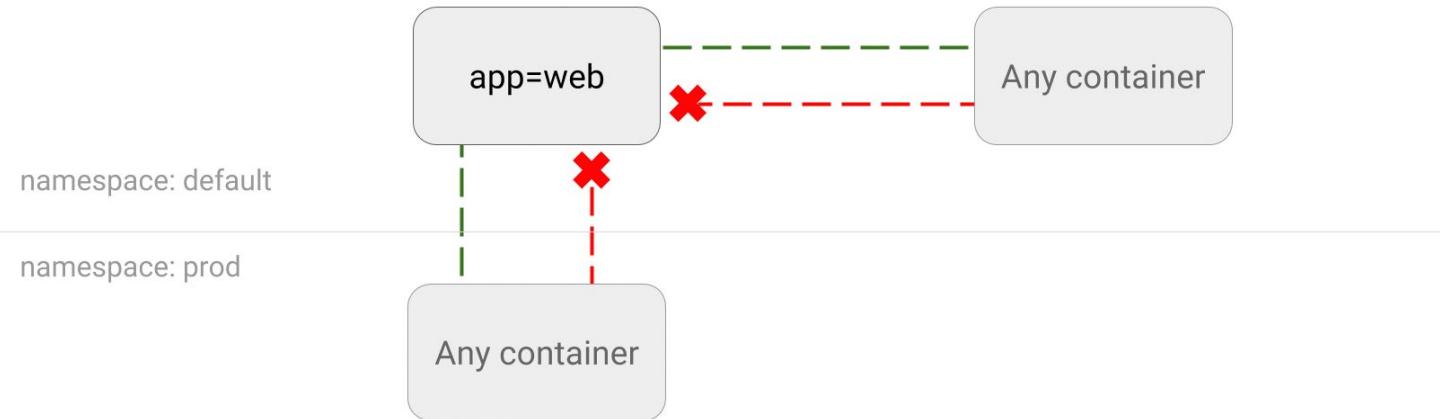
By default, all ports are accessible

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: port-allow
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
  ports:
  - protocol: TCP
    port: 8080
```

Only allow incoming
traffic to port 8080

Scenario-Based Learning Resource

Exercising use cases for Network Policies is helpful with understanding concept



<https://github.com/ahmetb/kubernetes-network-policy-recipes>

Visualizing and Analysing a Network Policy

The page networkpolicy.io provides policy editor

The screenshot shows the networkpolicy.io policy editor interface. At the top, there's a navigation bar with icons for back, forward, and search, followed by the URL 'editor.networkpolicy.io'. On the right side, there are buttons for 'Feedback/Questions?' and 'Ask on Slack'.

The main area displays a network policy visualization. It consists of several boxes representing different components:

- Outside Cluster**: Any endpoint
- In Namespace**: Any pod
app=coffeeshop
- In Cluster**: Everything in the cluster
- In Namespace**: app=payment-processor
role=api
- Outside Cluster**: Any endpoint
- In Namespace**: Any pod
- In Cluster**: Everything in the cluster
Kubernetes DNS

Red arrows point from the 'Outside Cluster' and 'In Cluster' boxes to the central 'In Namespace' box. Green arrows point from the 'In Namespace' box to the 'Outside Cluster', 'In Namespace', and 'In Cluster' boxes.

Below the visualization, there's a code editor showing the YAML representation of the policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: payment-processor
      role: api
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: coffeeshop
```

At the bottom, there are buttons for 'Download' and 'Share'. To the right, there's a sidebar titled 'Welcome to the Network Policy Editor!' with a 'Main tutorial' link and a 'Flows upload' button. Below the sidebar, there's an illustration of a cluster with namespaces and pods.

Exercise

Restricting access to and
from a Pod with Network
Policies

