

 Use the command line option --grace-period=0 and --force to send a SIGKILL signal. The signal will delete a Kubernetes object immediately

**Deleting Kubernetes Objects**

**kubectl delete pod nginx --grace-period=0 --force**

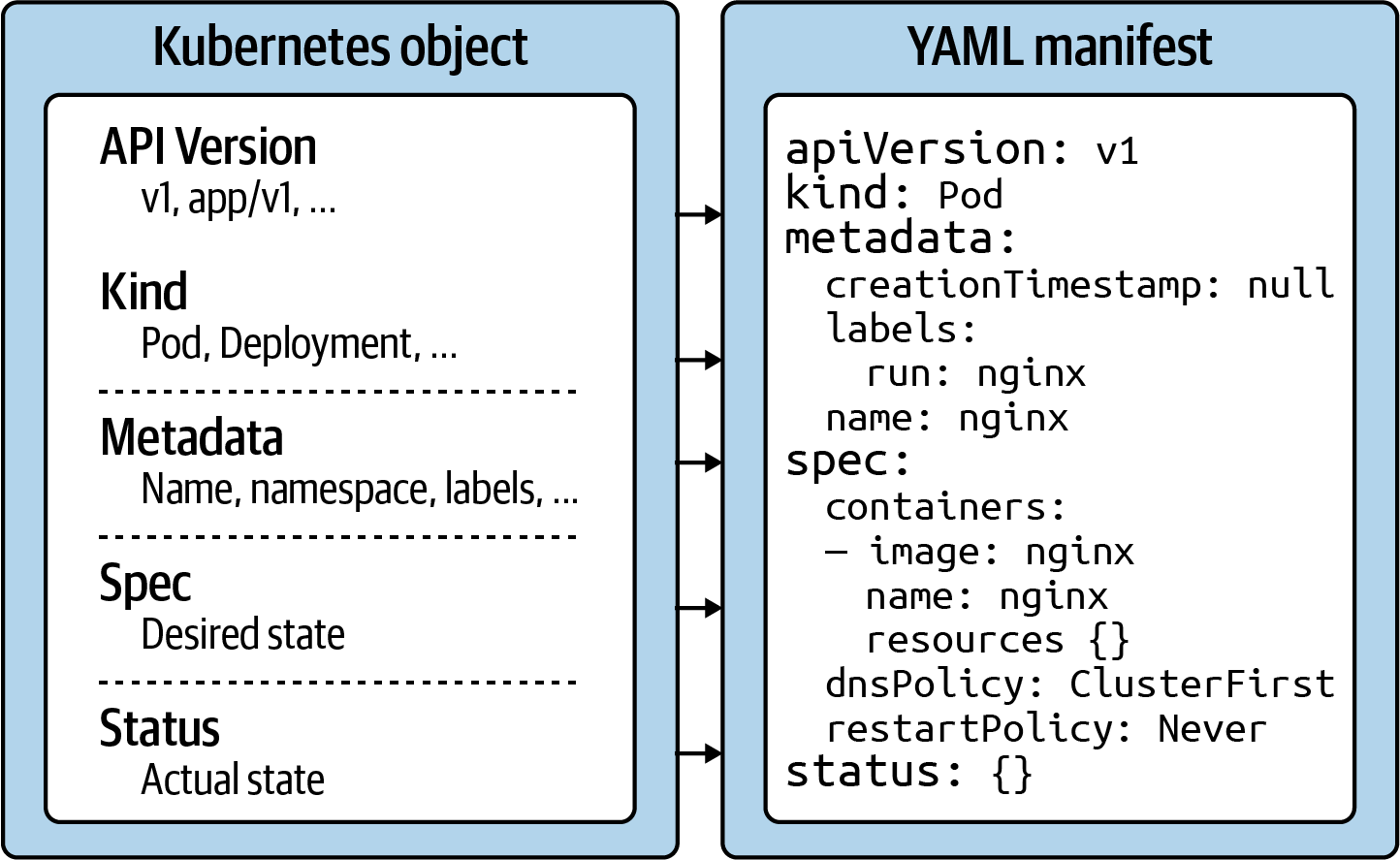
## Finding Object Information

**$ kubectl describe pods | grep -C 10 "author=John Doe"**

**$ kubectl get pods -o yaml | grep -C 5 labels:**

**Help**

**kubectl explain pods.spec**



*API version*

The Kubernetes API version defines the structure of a primitive and uses it to validate the correctness of the data. The API version serves a similar purpose as XML schemas to a XML document or JSON schemas to a JSON document. The version usually undergoes a maturity process—e.g., from alpha to beta to final. Sometimes you see different prefixes separated by a slash (e.g., apps). You can list the API versions compatible with your cluster version by running the command kubectl api-versions.

*Kind*

The kind defines the type of primitive—e.g., a Pod or a Service. It ultimately answers the question, “What type of object are we dealing with here?”

*Metadata*

Metadata describes higher-level information about the object—e.g., its name, what namespace it lives in, or whether it defines labels and annotations. This section also defines the UID.

*Spec*

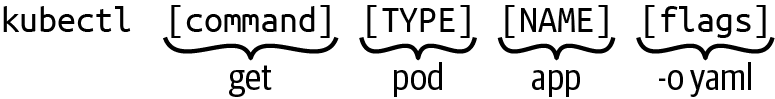
The specification (“spec” for short) declares the desired state—e.g., how should this object look after it has been created? Which image should run in the container, or which environment variables should be set for?

*Status*

The status describes the actual state of an object. The Kubernetes controllers and their reconcilliation loops constantly try to transition a Kubernetes object from the desired state into the actual state. The object has not yet been materialized if the YAML status shows the value {}.

With this basic structure in mind, let’s have a look at how to create a Kubernetes object with the help of kubectl.

**kubectl** [command] [TYPE] [NAME] [flags]



# Object Management

## Imperative Approach

**$ kubectl run frontend --image=nginx --restart=Never --port=80**

## Declarative Approach

**$ vim pod.yaml**

**$ kubectl create -f pod.yaml**

## Hybrid Approach

**kubectl run frontend --image=nginx --restart=Never --port=80 -o yaml --dry-run=client > pod.yaml**

### DELETING AN OBJECT

**$ kubectl delete pod frontend**

Or

**$ kubectl delete -f pod.yaml**

### EDITING A LIVE OBJECT

**kubectl edit pod frontend**

### REPLACING A LIVE OBJECT

**kubectl d.yaml**

### UPDATING A LIVE OBJECT

**kubectl apply -f pod.yaml**

# Creating Pods

**kubectl run hazelcast --image=hazelcast/hazelcast --restart=Never \**

**--port=5701 --env="DNS\_DOMAIN=cluster" --labels="app=hazelcast,env=prod"**

**Or**

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: hazelcast

**labels**:

**app**: hazelcast

**env**: prod

**spec**:

**containers**:

- **env**:

- **name**: DNS\_DOMAIN

**value**: cluster

**image**: hazelcast/hazelcast

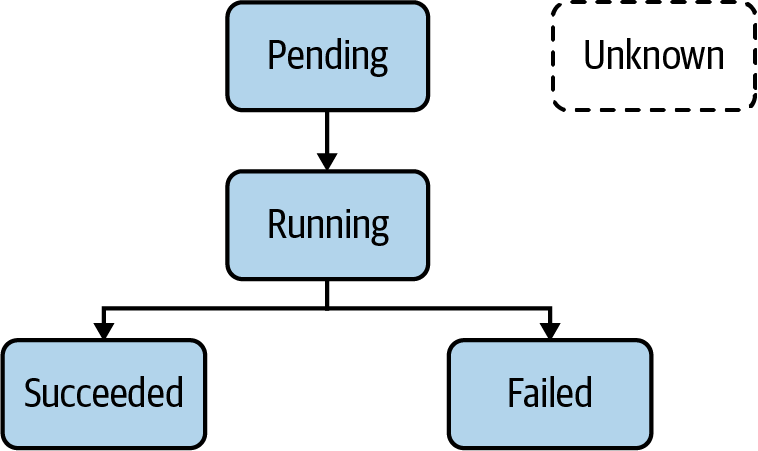
**name**: hazelcast

**ports**:

- **containerPort**: 5701

**restartPolicy**: Never

# Pod Life Cycle Phases



# Rendering Pod Details

**kubectl describe pods hazelcast**

# Accessing Logs of a Pod

**kubectl logs hazelcast**

# Executing a Command in Container

**kubectl exec -it hazelcast -- /bin/sh**

##### ***Example YAML manifest for a Pod defining environment variables***

**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'

**kubectl run mypod --image=busybox -o yaml --dry-run=client --restart=Never \**

**> pod.yaml -- /bin/sh -c "while true; do date; sleep 10; done"**

# Understanding Namespaces

**kubectl get namespaces**

**kubectl create namespace code-red**

**apiVersion**: v1

**kind**: Namespace

**metadata**:

**name**: code-red

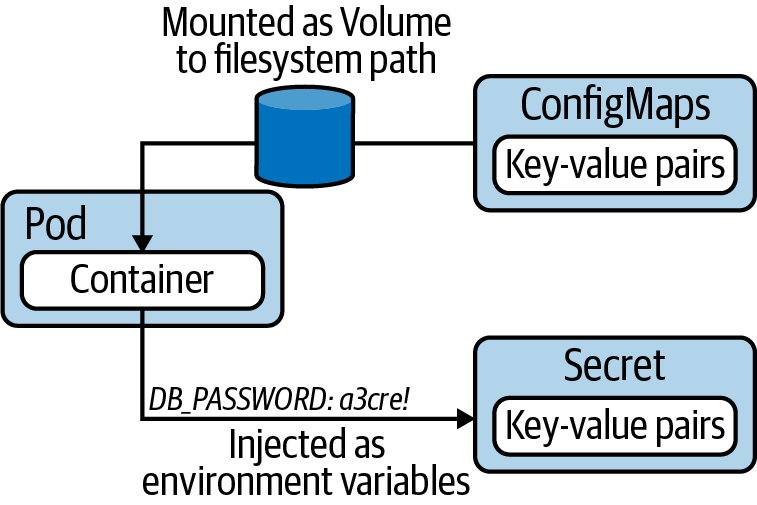
**kubectl run pod --image=nginx --restart=Never -n code-red**

**kubectl delete namespace code-red**

**kubectl get pods -n code-red**

# Configuration

* ConfigMap
* Secret
* Volume
* Security Context
* Resource Boundaries
* ResourceQuota
* Service Account



## Creating a ConfigMap

* Literal values, which are key-value pairs as plain text.

**kubectl create configmap db-config --from-literal=db=staging**

* A file that contains key-value pairs and expects them to be environment variables.
* **kubectl create configmap db-config --from-env-file=config.env**
* A file with arbitrary contents.

**kubectl create configmap db-config --from-file=config.txt**

* A directory with one or many files.
* **kubectl create configmap db-config --from-file=app-config**

**apiVersion**: v1

**kind**: ConfigMap

**metadata**:

**name**: backend-config

**data**:

**database\_url**: jdbc:postgresql://localhost/test

**user**: fred

------------------------------------

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: configured-pod

**spec**:

**containers**:

- **image**: nginx:1.19.0

**name**: app

**envFrom**:

- **configMapRef**:

**name**: backend-config

**kubectl exec configured-pod – env**

**-----------------------------------------**

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: configured-pod

**spec**:

**containers**:

- **image**: nginx:1.19.0

**name**: app

**volumeMounts**:

- **name**: config-volume

**mountPath**: /etc/config

**volumes**:

- **name**: config-volume

**configMap**:

**name**: backend-config

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: configured-pod

**spec**:

**containers**:

- **image**: nginx:1.19.0

**name**: app

**env**:

- **name**: DATABASE\_URL

**valueFrom**:

**configMapKeyRef**:

**name**: backend-config

**key**: database\_url

- **name**: USERNAME

**valueFrom**:

**configMapKeyRef**:

**name**: backend-config

**key**: user

**$ kubectl exec -it configured-pod -- /bin/sh**

# ls -1 /etc/config

database\_url

user

# cat /etc/config/database\_url

jdbc:postgresql://localhost/test

# cat /etc/config/user

fred

## Creating a Secret

|  |  |
| --- | --- |
| generic | Creates a secret from a file, directory, or literal value. |
| docker-registry | Creates a secret for use with a Docker registry. |
| tls | Creates a TLS secret. |

**kubectl create secret generic db-creds --from-literal=pwd=s3cre!**

**kubectl create secret generic db-creds --from-env-file=secret.env**

**kubectl create secret generic ssh-key --from-file=id\_rsa=~/.ssh/id\_rsa**

**echo -n 's3cre!' | base64**

**apiVersion**: v1

**kind**: Secret

**metadata**:

**name**: db-creds

**type**: Opaque

**data**:

**pwd**: czNjcmUh

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: configured-pod

**spec**:

**containers**:

- **image**: nginx:1.19.0

**name**: app

**envFrom**:

- **secretRef**:

**name**: db-creds

## Creating a ResourceQuota

**kubectl create namespace team-awesome**

**kubectl get namespace**

* Limit the number of Pods to 2.
* Define the minimum resources requested by a Pod to 1 CPU and 1024m of RAM.
* Define the maximum resources used by a Pod to 4 CPUs and 4096m of RAM
* **apiVersion**: v1
* **kind**: ResourceQuota
* **metadata**:
* **name**: awesome-quota
* **spec**:
* **hard**:
* **pods**: 2
* **requests.cpu**: "1"
* **requests.memory**: 1024m
* **limits.cpu**: "4"
* **limits.memory**: 4096m

**kubectl create -f awesome-quota.yaml --namespace=team-awesome**

**kubectl describe resourcequota awesome-quota --namespace=team-awesome**

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: nginx

**spec**:

**containers**:

- **image**: nginx:1.18.0

**name**: nginx

**resources**:

**requests**:

**cpu**: "0.5"

**memory**: "512m"

**limits**:

**cpu**: "1"

**memory**: "1024m"

# Understanding Service Accounts



**kubectl get serviceaccounts**

**kubectl get serviceaccount default -o yaml | grep -A 1 secrets:**

**kubectl get secret default-token-bf8rh -o yaml**

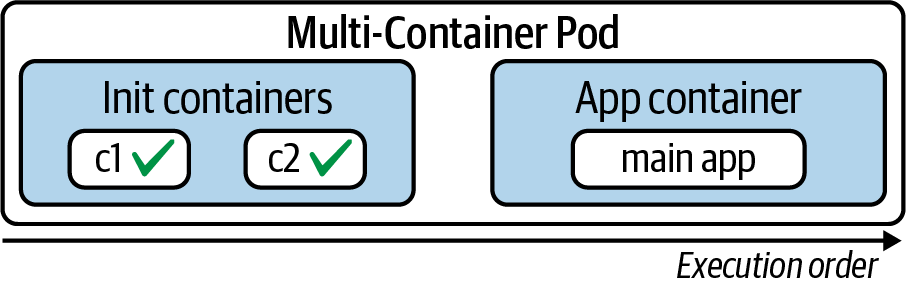
**Service Accounts**

# Multi-Container Pods

* Pod
* Container
* Volume
* Design patterns

# Init Containers

In Kubernetes, this functionality can be achieved with the help of init containers. Init containers are always started before the main application containers, which means they have their own lifecycle



**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: business-app

**spec**:

**initContainers**:

- **name**: configurer

**image**: busybox:1.32.0

**command**: ['sh', '-c', 'echo Configuring application... && \

mkdir -p /usr/shared/app && echo -e "{\"dbConfig\": \

{\"host\":\"localhost\",\"port\":5432,\"dbName\":\"customers\"}}" \

> /usr/shared/app/config.json']

**volumeMounts**:

- **name**: configdir

**mountPath**: "/usr/shared/app"

**containers**:

- **image**: bmuschko/nodejs-read-config:1.0.0

**name**: web

**ports**:

- **containerPort**: 8080

**volumeMounts**:

- **name**: configdir

**mountPath**: "/usr/shared/app"

**volumes**:

- **name**: configdir

**emptyDir**: {}

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

pod/business-app created

**$ kubectl get pod business-app**

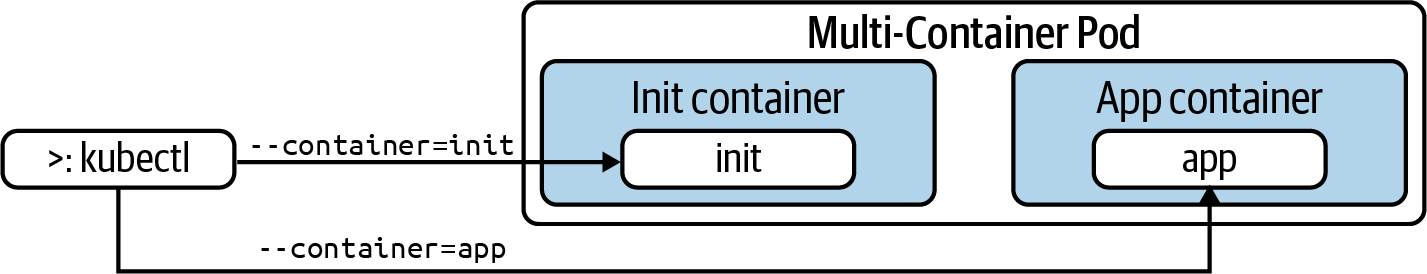
NAME READY STATUS RESTARTS AGE

business-app 0/1 Init:0/1 0 2s

**$ kubectl get pod business-app**

NAME READY STATUS RESTARTS AGE

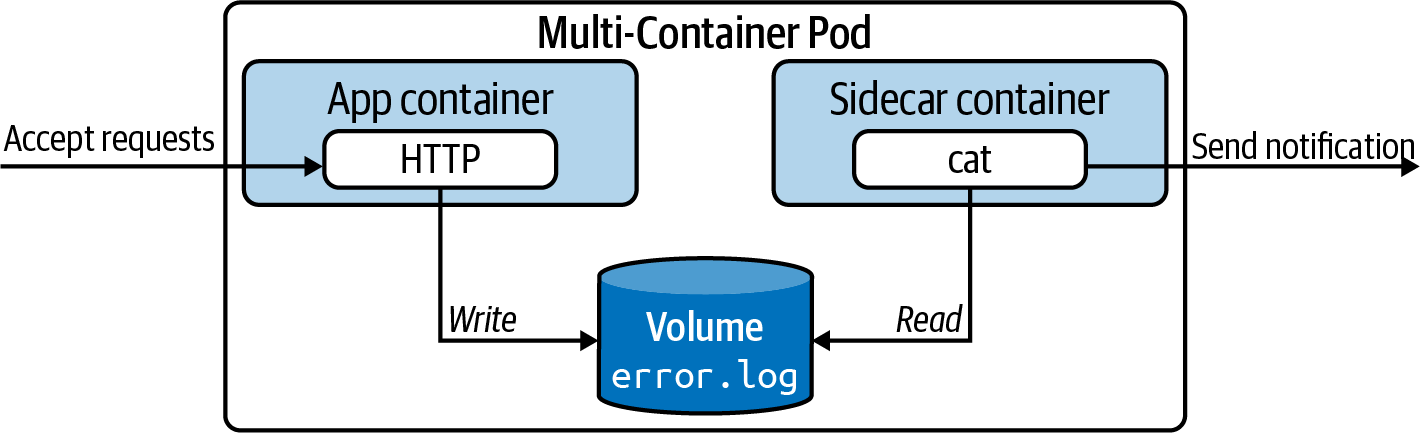
business-app 1/1 Running 0 8s



# Sidecar Pattern

Typically, there are two different categories of containers: the container that runs the application and another container that provides helper functionality to the primary application. In the Kubernetes space, the container providing helper functionality is called a sidecar

Among the most commonly used capabilities of a sidecar container are file synchronization, logging, and watcher capabilities



**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: webserver

**spec**:

**containers**:

- **name**: nginx

**image**: nginx

**volumeMounts**:

- **name**: logs-vol

**mountPath**: /var/log/nginx

- **name**: sidecar

**image**: busybox

**command**: ["sh","-c","while true; do if [ \"$(cat /var/log/nginx/error.log \

| grep 'error')\" != \"\" ]; then echo 'Error discovered!'; fi; \

sleep 10; done"]

**volumeMounts**:

- **name**: logs-vol

**mountPath**: /var/log/nginx

**volumes**:

- **name**: logs-vol

**emptyDir**: {}

# Adapter Pattern

# Ambassador Pattern

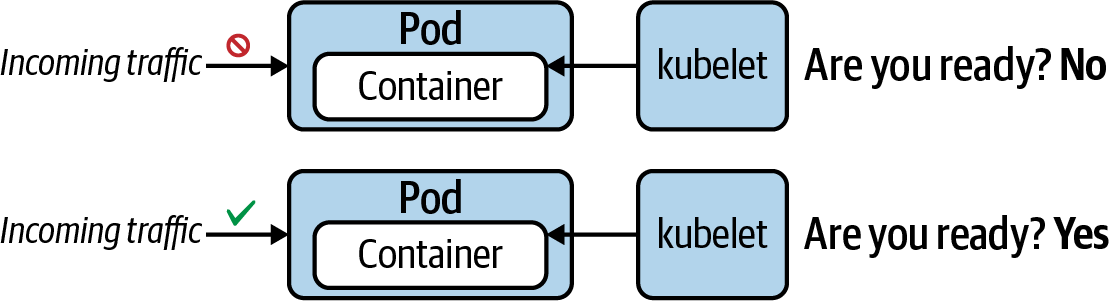
# Observability

* Readiness probe
* Liveness probe
* Startup probe
* Troubleshooting Kubernetes objects
* Monitoring

Kubernetes provides a concept called health probing to automate the detection and correction of such issues. You can configure a container to execute a periodic mini-process that checks for certain conditions. These processes are defined as follows:

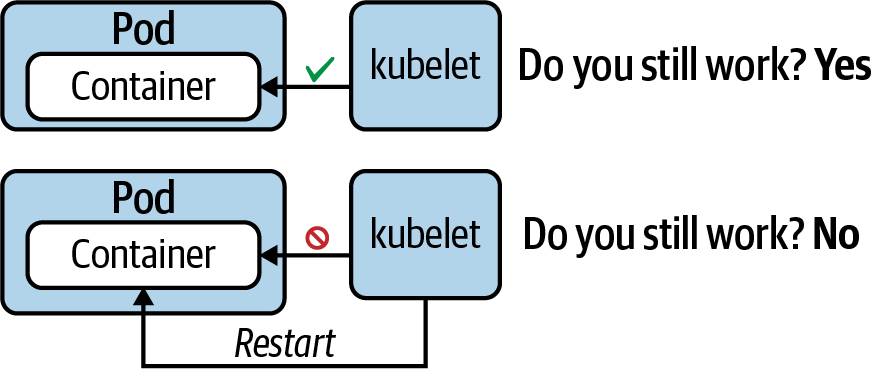
*Readiness probe*

Even after an application has been started up, it may still need to execute configuration procedures—for example, connecting to a database and preparing data. This probe checks if the application is ready to serve incoming requests.



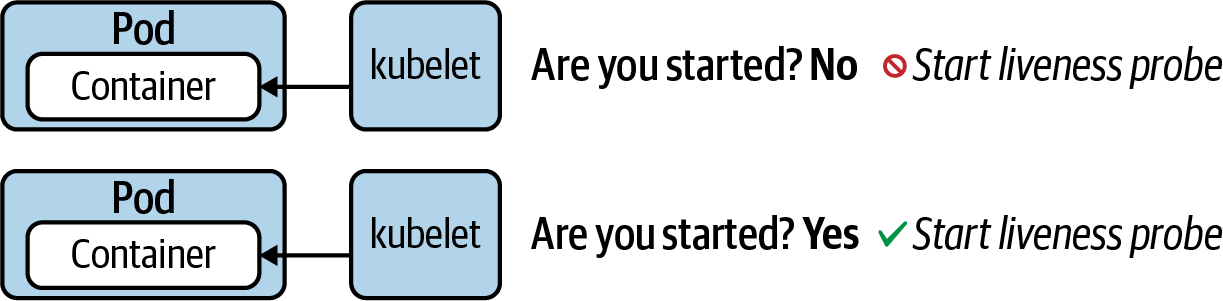
*Liveness probe*

Once the application is running, we’ll want to make sure that it still works as expected without issues. This probe periodically checks for the application’s responsiveness. Kubernetes restarts the Pod automatically if the probe considers the application be in an unhealthy state.



*Startup probe*

Legacy applications in particular can take a long time to start up—we’re talking minutes sometimes. This probe can be instantiated to wait for a predefined amount of time before a liveness probe is allowed to start probing. By setting up a startup probe, you can prevent overwhelming the application process with probing requests. Startup probes kill the container if the application couldn’t start within the set time frame



**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: readiness-pod

**spec**:

**containers**:

- **image**: bmuschko/nodejs-hello-world:1.0.0

**name**: hello-world

**ports**:

- **name**: nodejs-port

**containerPort**: 3000

**readinessProbe**:

**httpGet**:

**path**: /

**port**: nodejs-port

**initialDelaySeconds**: 2

**periodSeconds**: 8

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: liveness-pod

**spec**:

**containers**:

- **image**: busybox

**name**: app

**args**:

- /bin/sh

- -c

- 'while true; do touch /tmp/heartbeat.txt; sleep 5; done;'

**livenessProbe**:

**exec**:

**command**:

- test `find /tmp/heartbeat.txt -mmin -1`

**initialDelaySeconds**: 5

**periodSeconds**: 30

**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

TroubleShooting

<https://kubeyaml.com/>

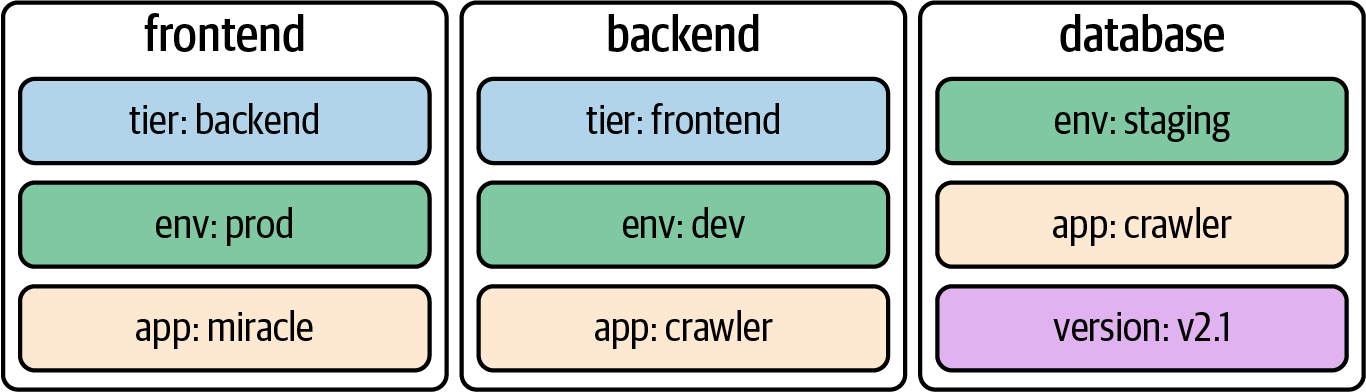
# Kube YAML

**kubectl top nodes**

# Pod Design

* Label
* Annotation
* Deployment
* ReplicaSet
* Horizontal Pod Autoscaler
* Job
* CronJob

# Understanding Labels



**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: labeled-pod

**labels**:

**env**: dev

**tier**: backend

**spec**:

**containers**:

- **image**: nginx

**name**: nginx

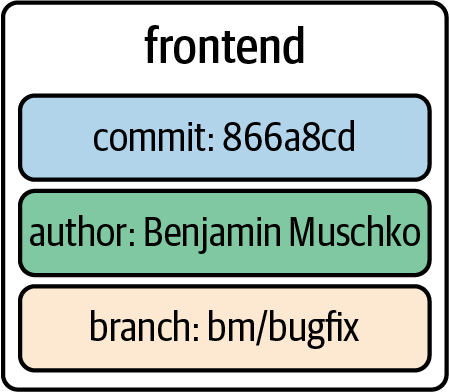
**kubectl describe pod labeled-pod | grep -C 2 Labels:**

**kubectl get pods -l env=prod --show-labels**

**kubectl get pods -l 'team in (shiny, legacy)' --show-labels**

# Annotations

Annotations are declared similarly to labels, but they serve a different purpose. They represent key-value pairs for providing descriptive metadata. The most important differentiator is that annotations cannot be used for querying or selecting objects. Typical examples of annotations may include SCM commit hash IDs, release information, or contact details for teams operating the object. Make sure to put the value of an annotation into single- or double-quotes if it contains special characters or spaces



**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: annotated-pod

**annotations**:

**commit**: 866a8dc

**author**: 'Benjamin Muschko'

**branch**: 'bm/bugfix'

**spec**:

**containers**:

- **image**: nginx

**name**: nginx

**$ kubectl describe pod annotated-pod | grep -C 2 Annotations:**

...

Annotations: author: Benjamin Muschko

branch: bm/bugfix

commit: 866a8dc

**$ kubectl annotate pod annotated-pod oncall='800-555-1212'**

pod/annotated-pod annotated

**$ kubectl annotate pod annotated-pod oncall='800-555-2000' --overwrite**

pod/annotated-pod annotated

**$ kubectl annotate pod annotated-pod oncall-**

pod/annotated-pod annotated

ReplicaSet’

**apiVersion**: apps/v1

**kind**: Deployment

**metadata**:

**name**: my-deploy

**labels**:

**app**: my-deploy

**spec**:

**replicas**: 1

**selector**:

**matchLabels**:

**app**: my-deploy

**template**:

**metadata**:

**labels**:

**app**: my-deploy

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**kubectl get deployments,pods,replicasets**

**kubectl describe deployment.apps/my-deplo**

**kubectl rollout history deployment my-deploy**

**kubectl set image deployment my-deploy nginx=nginx:1.19.2**

**kubectl rollout history deployment my-deploy**

**kubectl rollout history deployments my-deploy --revision=2**

**kubectl rollout undo deployment my-deploy --to-revision=1**

**kubectl scale deployment my-deploy --replicas=5**

# Autoscaling a Deployment

*Horizontal Pod Autoscaler (HPA)*

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 metric

Jobs

# Services & Networking

* Service
* Deployment
* Network Policy