Deploying Applications with Kubernetes

Lindis Webb

Version 1.0, 2017-05-20

Table of Contents

Legal Mombo Jombo

Copyright

Distribution

Student Feedback

Thanks for singing up to take the Deploying Applications with Kubernetes with Solinea. To help us improve this course, please send a message to training@solinea.com.

Lab Conventions

There will be a number of text styles and icons throughout this lab guide. Here are examples and explanations of their intended meanings.

$ `reboot`	Any test a student needs to enter is printed like this.
<your.ip>	Any time a student needs to insert their own value, the text has brackets.
File	User Interface (UI) buttons and objects are bold.
Special Font	Unusual or important words or phrases are marked with italics.
A **RED** *arrow*	A Red Focus arrow for calling attention as shown in Figure 1 (below)

Figure 1. arrow

A block of code:

```
{
    "apiVersion": "extensions/v1beta1",
    "kind": "Deployment",
    "metadata": {
        "annotations": {
            "deployment.kubernetes.io/revision": "1"
        },
        "creationTimestamp": "2017-05-23T17:56:15Z",
        "generation": 1,
        "labels": {
            "run": "nginx"
```

A block of code with title:

nginx.conf

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}
```

Warnings or important notes appear in a box like this.

Notes appear in a box like this.

Cautions appear in a box like this.

Tips and tricks in a box like this.

Lab Environment

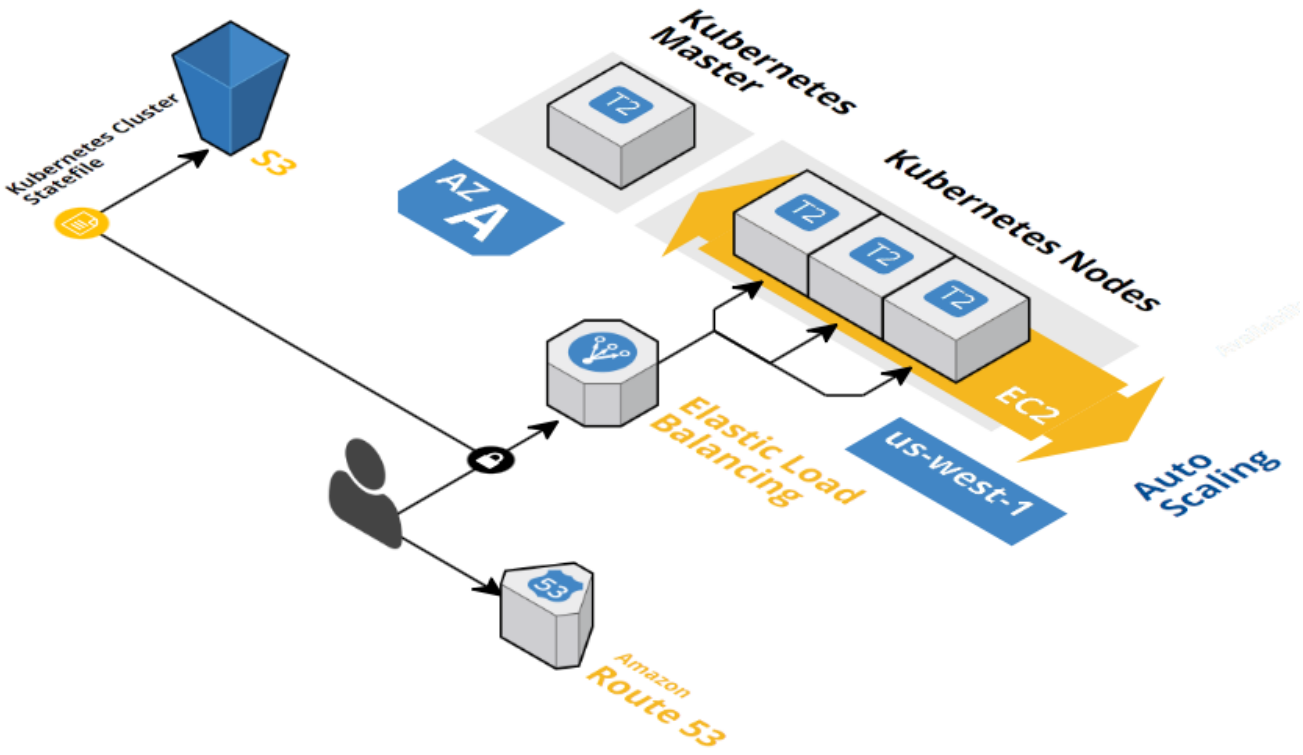
Lab Objectives

In this section, we'll connect to the provided Kubernetes cluster using the `kubectl` tool.

The Lab Topology

Before we connect to the Kubernetes Cluster, let's take a moment to review the configuration of the provided cluster. The cluster is hosted in an AWS region (the region closest to the location of the class). The cluster has access to all AWS services required to build a production ready environment.

The below image is a quick high level topology of your Kubernetes Cluster.



Each student is assigned a dedicated Kubernetes cluster with the following specs:

- 1 x Master
- 3 x Nodes
- 1 x VPC (Virtual Private Cloud)
- 1 x Dedicated DNS Hosted Zone

This course environment **will be deleted** at the end of the course.

Consider setting up a github repo to hold your project code.

Lab Overview

In this lab

Step 1. Install the kubectl client

To configure your designated system, install the `kubectl` client and place the provided `config` file into their .kube folder.

Installing kubectl on MacOS

On MacOS, kubectl can be installed using one of the following methods:
via the curl command

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/darwin/amd64/kubectl
```

Once downloaded, the file must be made executable (`chmod +x <filename>`) and moved to the `/usr/local/bin/` directory.

via homebrew
`brew install kubectl`

Installing kubectl on Windows

On Windows kubectl can be installed using on of the following methods:
via the curl command

```

```

via downloading from the Internet

Installing kubectl on Linux

On Linux, kubectl can be installed using one of the following methods:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/linux/amd64/kubectl
```

Once downloaded, make the file executable (`chmod +x <filename>`) and move it to the `/usr/local/bin/` directory.

Step 2. Testing kubectl

To give `kubectl` a quick test, run the following command:
`kubectl version`
This should show something very similar to the following:

```
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.1",
GitCommit:"b0b7a323cc5a4a2019b2e9520c21c7830b7f708e", GitTreeState:"clean",
BuildDate:"2017-04-03T23:37:53Z", GoVersion:"go1.8", Compiler:"gc",
Platform:"darwin/amd64"}

Server Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.4+coreos.0",
GitCommit:"97c11b097b1a2b194f1eddca8ce5468fcc83331c", GitTreeState:"clean",
BuildDate:"2017-03-08T23:54:21Z", GoVersion:"go1.7.4", Compiler:"gc",
Platform:"linux/amd64"}
```

Step 3. Configuring kubectl to work with the provided cluster

In the `lab_resources` folder, look for the folder with your username. In this folder, there will be a file called `config`. This file contains the configuration required to interact with the course assigned Kubernetes Cluster.

Placing the `config` file in your `/.kube` folder will **overwrite** any configs here; backup or rename any `config` file that might be in this directory.

Place the appropriate `config` file in your `./kube/` folder.

Once the `config` file is located in the correct folder, run the following command to get a list of configured nodes:
`kubectl get nodes`
returns something that looks like the following

```
NAME                                         STATUS         AGE       VERSION
ip-172-20-33-181.us-west-1.compute.internal   Ready,node     48m       v1.6.2
ip-172-20-51-116.us-west-1.compute.internal   Ready,master   49m       v1.6.2
ip-172-20-55-103.us-west-1.compute.internal   Ready,node     48m       v1.6.2
ip-172-20-54-113.us-west-1.compute.internal   Ready,node     48m       v1.6.2
```

If you do not see all nodes in `Ready`, please let the instructor know.

Kubernetes Dashboard (UI)

Lab Objectives

Deploy the Kubernetes Dashboard to a vanilla Kuberetes Cluster.

Lab Structure Overview

Lab Overview

Depending on the Kubernetes implementation, some distros come with a pre-configured dashboard. For the lab environment, a pre-configured dashboard is not using. This allows us to deploy a dashboard.

Step 1.

In the working_files directory, cd into the **dashboard_lab** directory. Validate that a file named kubernetes-dashboard.yaml exists.

Run the following command to deploy the Kubernetes dashboard.

```
$ kubectl create -f kubernetes-dashboard.yaml
```

```
deployment "kubernetes-dashboard" created
service "kubernetes-dashboard" created
```

This might take a few minutes to complete as Kubernetes will need to pull the images and start the containers.

Step 2.

Run this command to get the status of the deployment.

```
$ kubectl get deployment kubernetes-dashboard --namespace kube-system
```

```
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kubernetes-dashboard   1         1         1            1           1m
```

We'll cover the details of this command in a follow up module; for now, it's fine to continue without knowing what this all means.

Step 3.

Let's find the Kubernetes-Dashboard URL by running the following command:

```
$ kubectl cluster-info
```

```
Kubernetes master is running at https://api.kubernetes.kubectl.guru
KubeDNS is running at
https://api.kubernetes.kubectl.guru/api/v1/proxy/namespaces/kube-system/services/kube-
dns
kubernetes-dashboard is running at
https://api.kubernetes.kubectl.guru/api/v1/proxy/namespaces/kube-
system/services/kubernetes-dashboard
```

This will output the addresses of the master and services with a label `kubernetes.io/cluster-service=true`

output of `kubectl get deploy/kubernetes-dashboard -o json --namespace kube-system`

```
"labels": {
    "k8s-addon": "kubernetes-dashboard.addons.k8s.io",
    "k8s-app": "kubernetes-dashboard",
    "kubernetes.io/cluster-service": "true",
    "version": "v1.5.0"
```

We'll cover the details of this command in a follow up module; for now, it's fine to continue without knowing what this all means.

Step 4.

In a browser of your choosing, put the link that is associated with **kubernetes-dashboard** in the output of `kubectl cluster-info`.

You'll be promted for a username and password. For this deployment, these can be found by running the following command:

```
$ kubectl config view
```

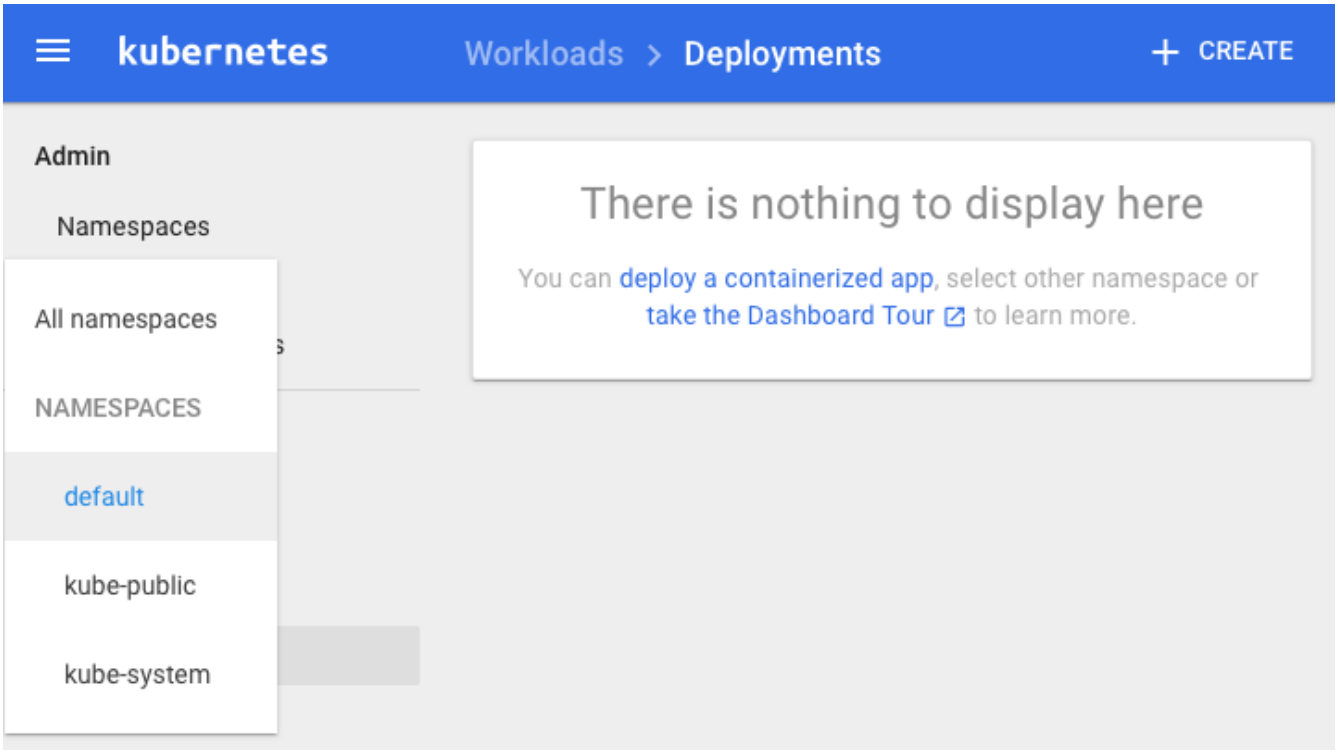
```
- name: kubernetes.kubectl.guru-basic-auth
  user:
    password: rigmEr57YGcHwS7Ujt9ssg6OsGnhiIfL
    username: admin
```

Use this username and password to sign into the dashboard. This password was generate randomly when the cluster was created. Updating this password is outside the scope of this course.

There are other methods used to connect to the Dashbaord; to control scope, only the above method is mentioned. Please refer to your operations team or the documentation when connecting to your enterprise's Kubernetes Dashboard.

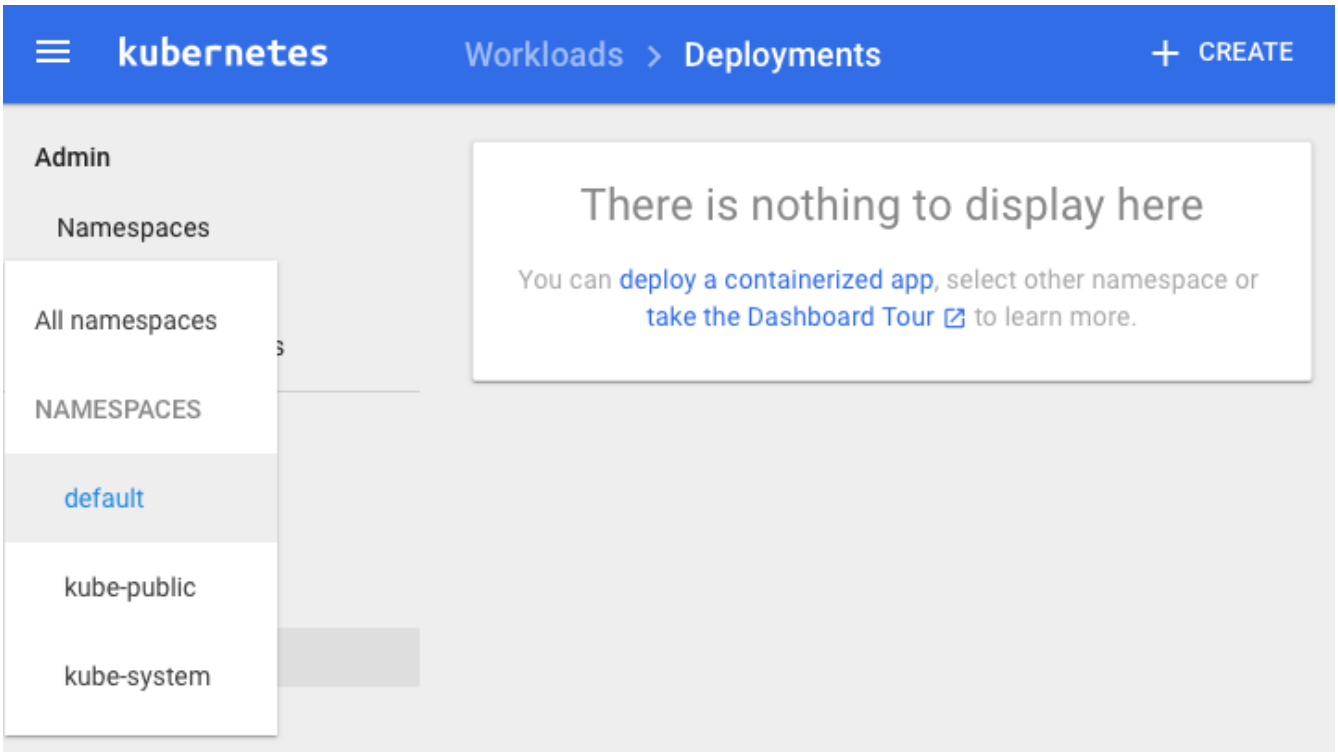
Step 5.

This is the basic screen that will output. The different sections are listed on the left side. When any of those are clicked, the body of the site updates with the output and options of the section. Navigate through the different sections to get familiar with its usage.



Step 6. Viewing Namespaces in the Dashboard

Although covered in more depth in a follow up module, take note of the **NAMESPACE** section. This will cascade through the other categories so please note the different default namespaces.



Conclusion

Pods and Labels

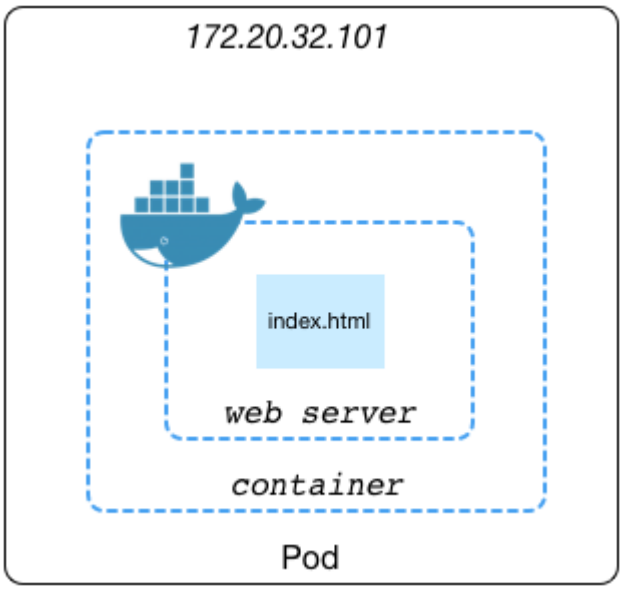
Lab Objectives

Lab Structure Overview

Lab Overview

Pods are the atomic unit of Kubernetes.

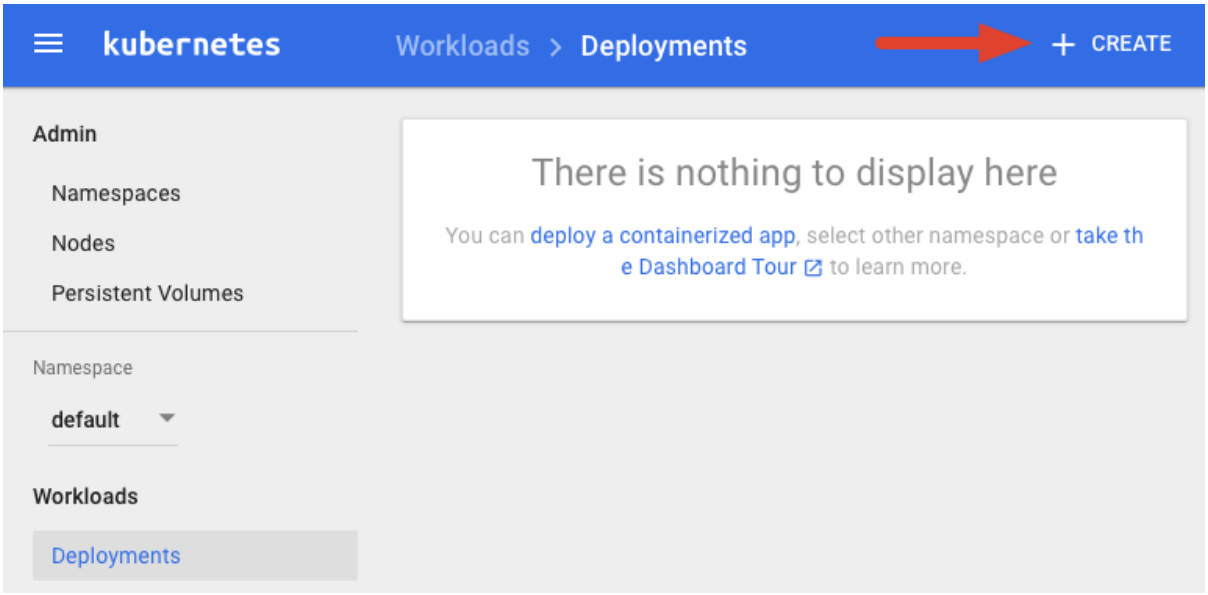
Using the dashboard; let's create the following pod using the `nginx:1.11-alpine` container.



Step 1. Deploy a pod using the GUI

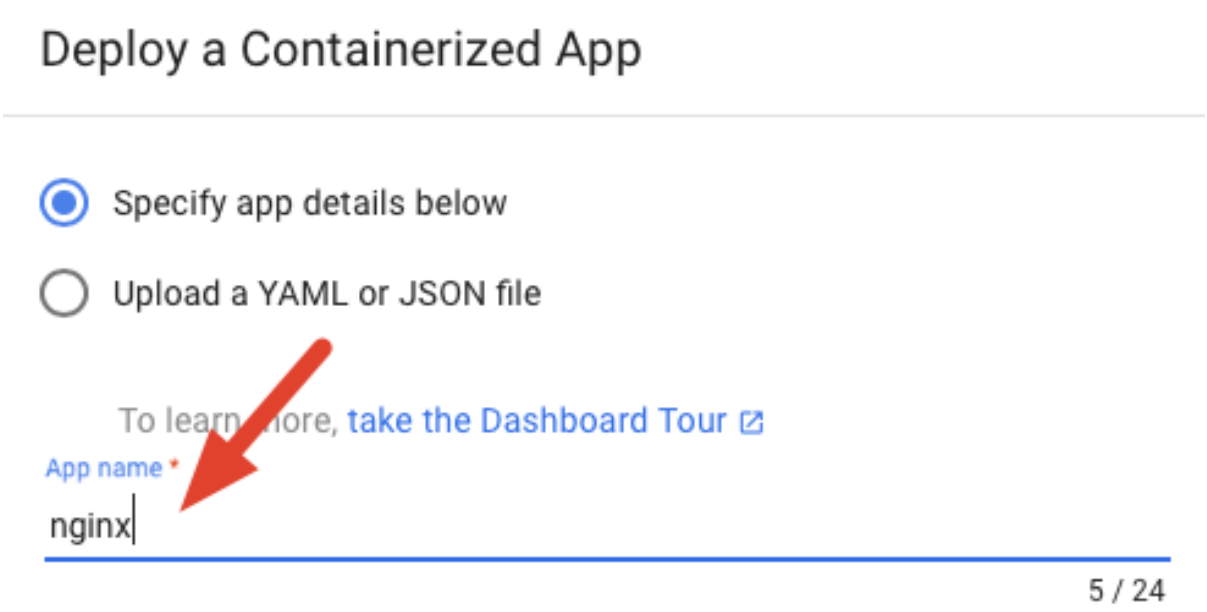
Create

Click on the + **CREATE** button on the top right of the screen.



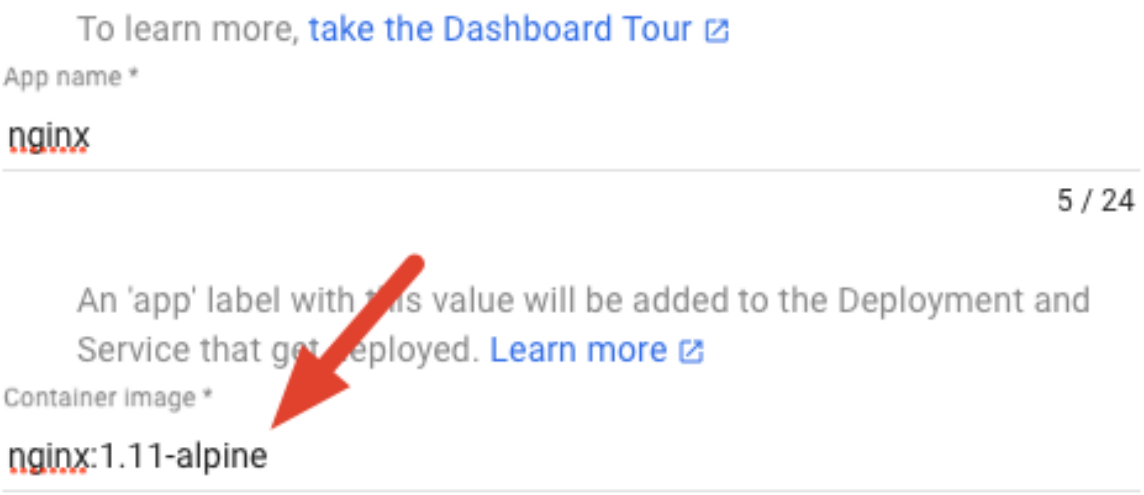
Define the name

Ensure the radio button for **Specify app details below** is clicked and populate the **App name** with `nginx`



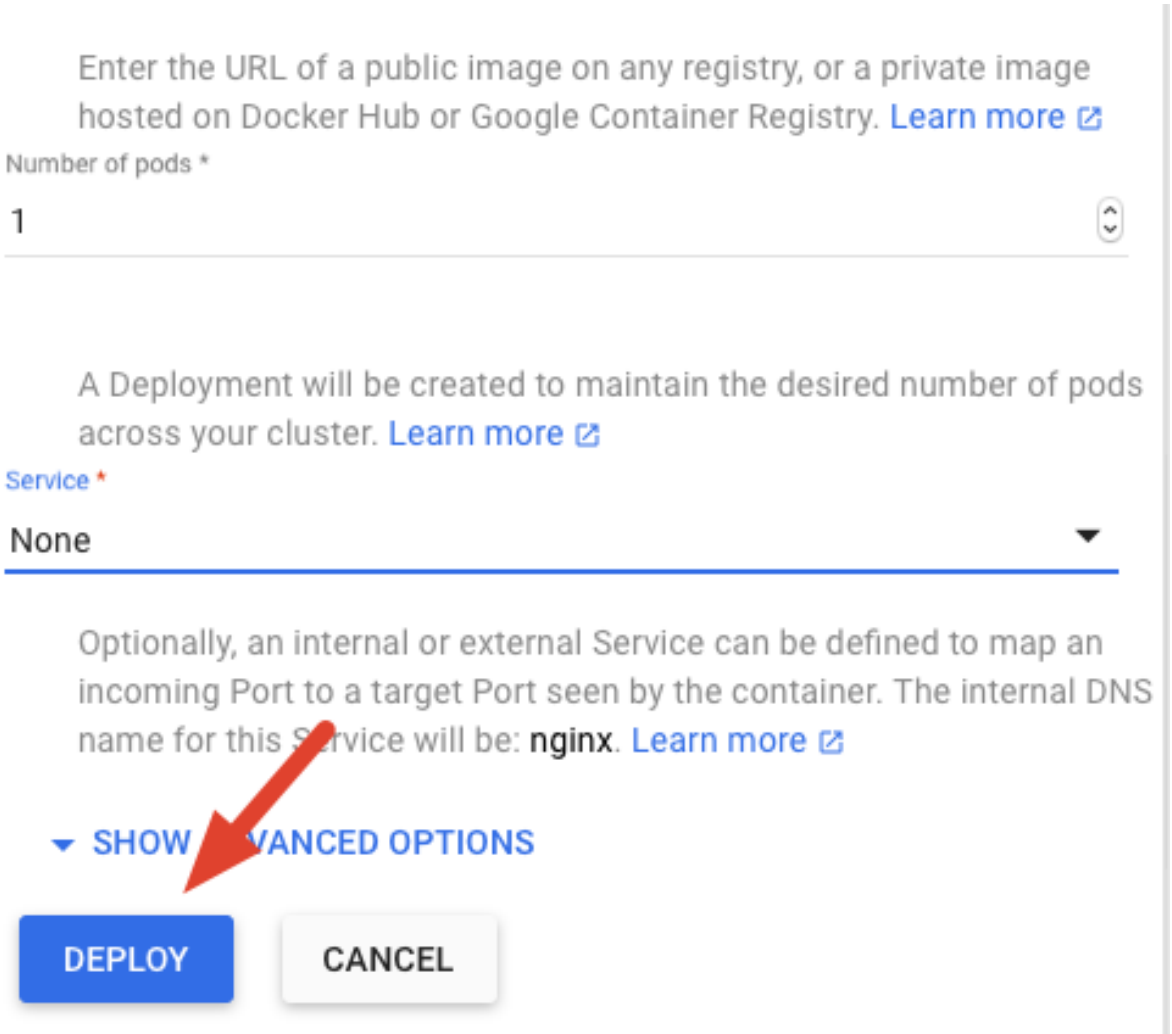
Define the image

Use `nginx:1.11-alpine` for the container image name.



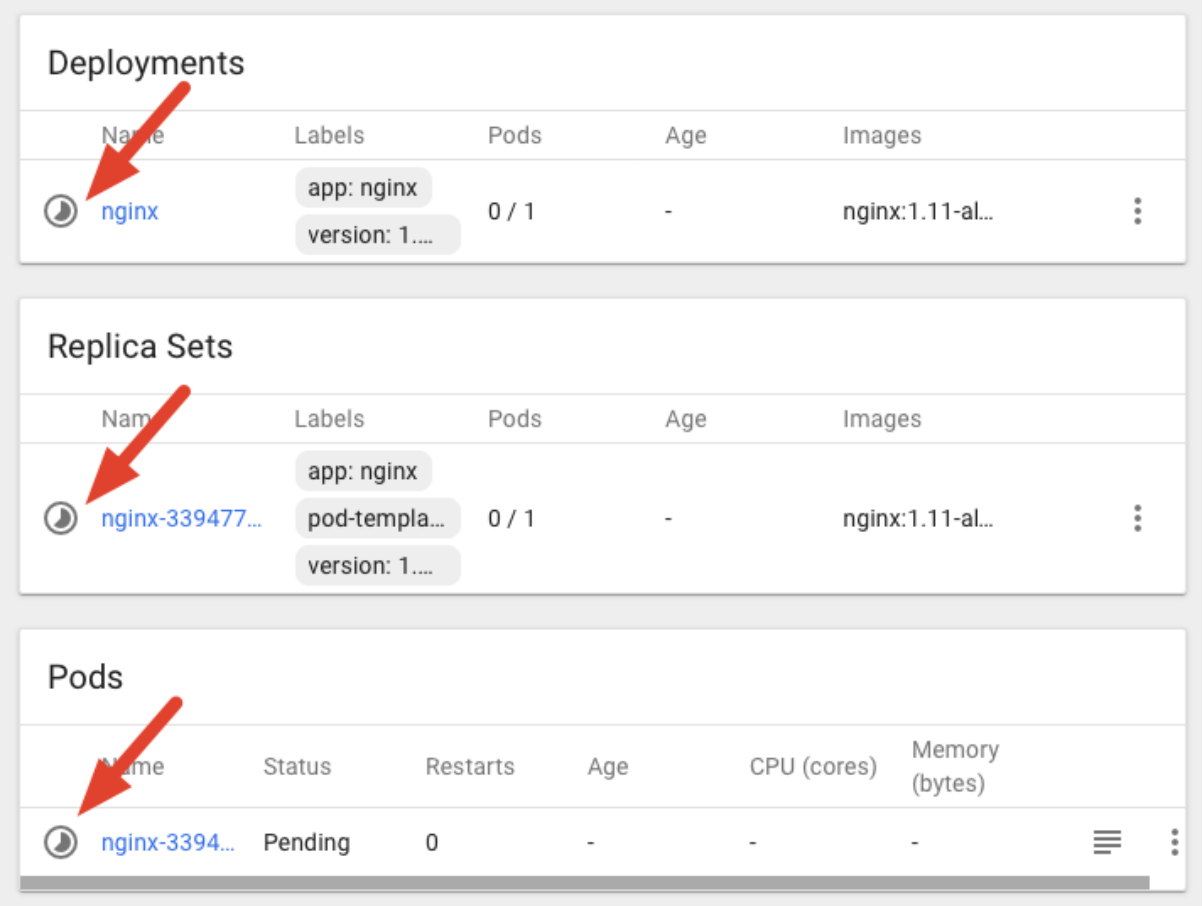
Deploy the container

Leave **Number of Pods** at 1, **Services** at none, and click the **DEPLOY** button.



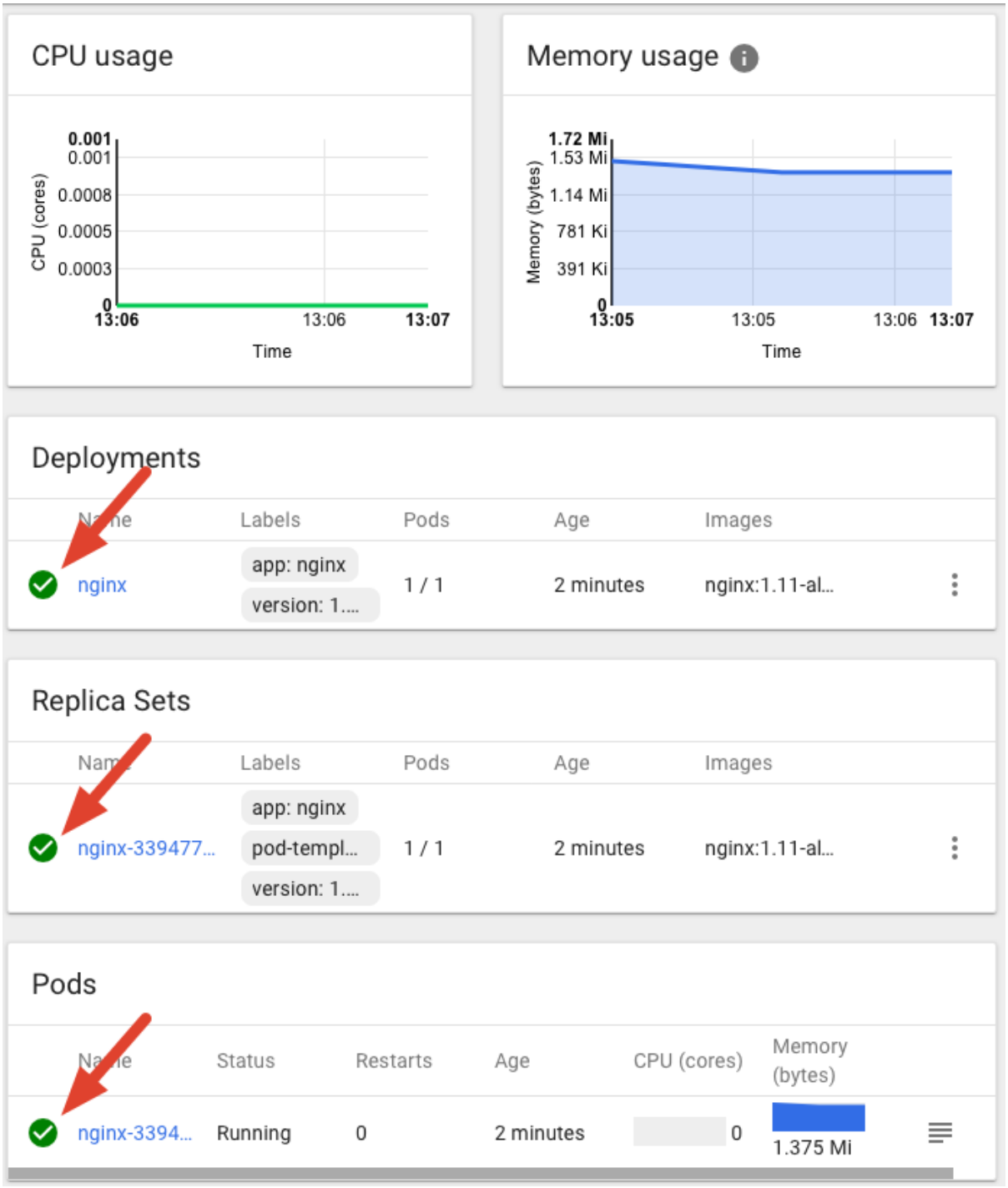
Validate deployed pod

Once **DEPLOY** is clicked, you will see the status in a small pie chart icon.



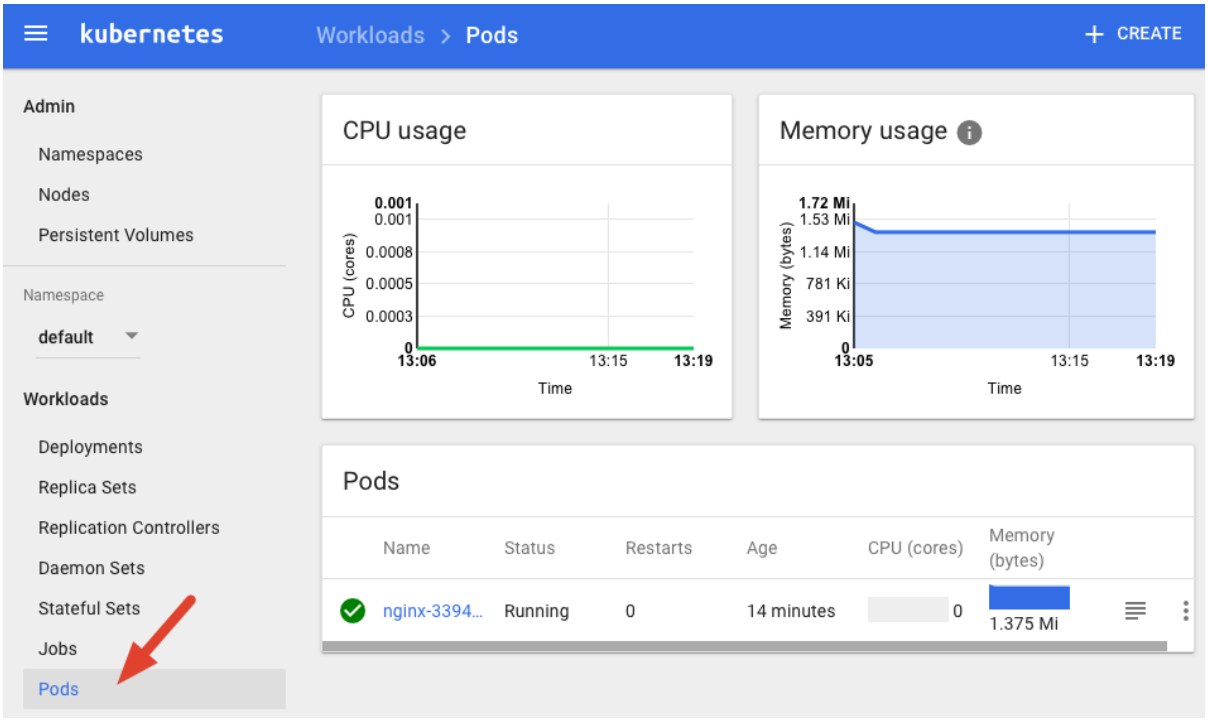
You'll need to refresh the page to see the status change, when all are green circles with checks, the rollout will be complete.

Creating this pod will create three Kubernetes objects (Deployments, Replica Sets, and Pods). Remember, pods are the atomic unit of Kubernetes.



Step 2. Review Pods

Click on the **Pods** section and it'll output the pod.



Step 3. Review Pods from the command-line

From the command line, run the `kubectl get pods` command and take a look at the data outputs.

READY
: The number of containers that are ready in the pod.

STATUS
: The status of the pod

RESTARTS
: Number of restarts during the life of the pod.

AGE
: Age of the pod

```
$ kubectl get pods
```

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3394779010-z1q9h   1/1       Running   0          57m
```

Step 4. Deploy a pod from the command-line

Let's deploy a similar pod from the command-line.

```
$ kubectl run nginx-cli --image nginx:1.11-alpine
```

To see the avaialble options and switched, run `kubectl run --help`

Let's get the pods and review the out.

```
$ kubectl get pods
```

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3394779010-z1q9h   1/1       Running   0          57m
nginx-cli-2713055296d3   1/1       Running   0          4m
```

Step 5. Labels

Let's create some nginx pod with labels, first from the command-line and next from a `yaml` file. Let's create one for prod and one for test. Let's also define the app, in this case, let's use `nginx`.

```
$ kubectl run nginx-prod \
    --image=nginx:1.11-alpine \
    --replicas=3 \
    --labels="app=nginx,env=prod"
```

```
$ kubectl run nginx-test \
    --image=nginx:1.9.10 \
    --replicas=2 \
    --labels="app=nginx,env=test"
```

Labels are the only way to group Kubernetes objects.

```
$ kubectl get pods
```

```
$ kubectl get pods --selector="app=nginx"
```

```
$ kubectl get pods --selector="env=prod"
```

```
$ kubectl get pods --selector="env=test"
```

Let's create another pod that runs apache.

```
$ kubectl run httpd-test \
    --image=httpd:alpine \
    --replicas=2 \
    --labels="app=httpd,env=test"
```

Let's run the following to see the different ways a label can be queried.

First, let's select all pods where `app=nginx`

```
$ kubectl get pods --selector="app=nginx"
```

Now, let's select all pods where `app=httpd`

```
$ kubectl get pods --selector="app=httpd"
```

```
$ kubectl get pods --selector="env=test"
```

One or more labels can be used with the `--selector` switch.

```
$ kubectl get pods --selector="env=test,app=nginx"
```

Before going to the next step; let's cleanup the environment. This will delete the pods we've made up to now.

```
$ kubectl delete all --all
```

Do not run this is production as it will delete most everything associated with the current namespace.

Step 6. Ports

Now that we are able to deploy a pod and associated some metadata with the pod; let's work on getting the pod ports defined. Let's deploy two pods; one without `--ports` defined and one with.

```
$ kubectl run nginx-no-ports --image=nginx:1.11-alpine
```

```
$ kubectl run nginx-ports --image=nginx:1.11-alpine --port=80
```

Let's output the pods using the `kubectl get pods` command.

Run `kubectl describe` on each of the pods. Notice that `no-ports` has no defined ports.

describe output of `nginx-no-ports`:

```
Containers:
  nginx-ports:
    Port:
```

describe output of `nginx-ports`:

```
Containers:
  nginx-ports:
    Port:       80/TCP
```

Step 7. Creating a pod manifest

```
$ kubectl run nginx-pod-cli --image nginx:1.11-alpine --port=80 --labels="app=nginx,env=dev"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-yaml
  labels:
    app: nginx
    env: dev
spec:
  containers:
  - name: nginx-pod-yaml
    image: nginx:1.11-alpine
    ports:
    - containerPort: 80
```

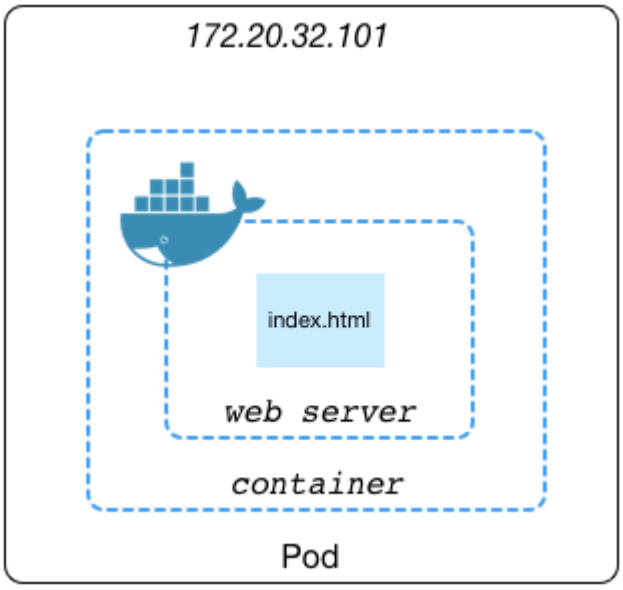
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-two-container
  labels:
    app: nginx
    env: dev
spec:
  containers:
  - name: nginx-container-1
    image: nginx:1.11-alpine
    ports:
    - containerPort: 80
  - name: nginx-container-2
    image: nginx:1.11-alpine
    ports:
    - containerPort: 80
```

Pods Appendix

A previous stated, pods are the atomic unit of Kubernetes; however, it is important to know that multiple containers and volumes can reside in a single pod. In a follow up lab, we'll create a deployment that uses two images.

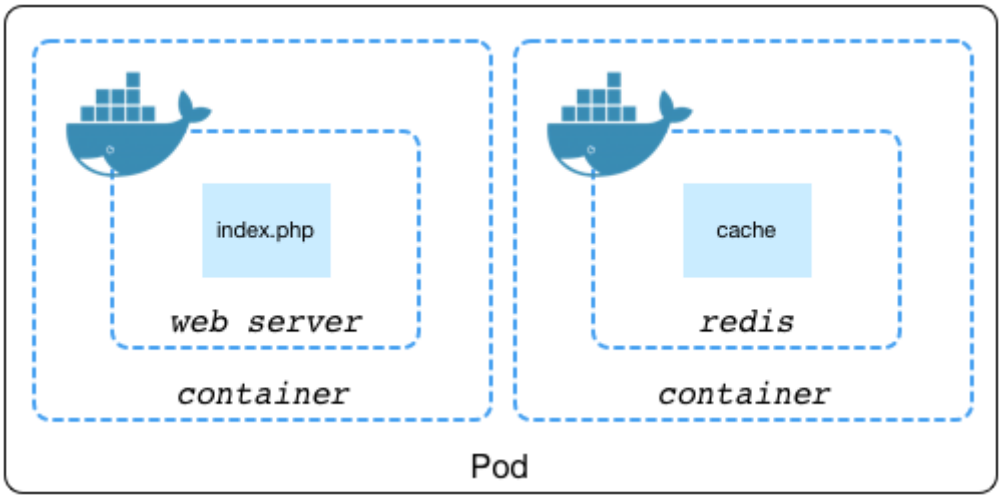
Appendix A. Standalone Container

In this scenario, a single pod runs a single container.



Appendix B. Container Grouping

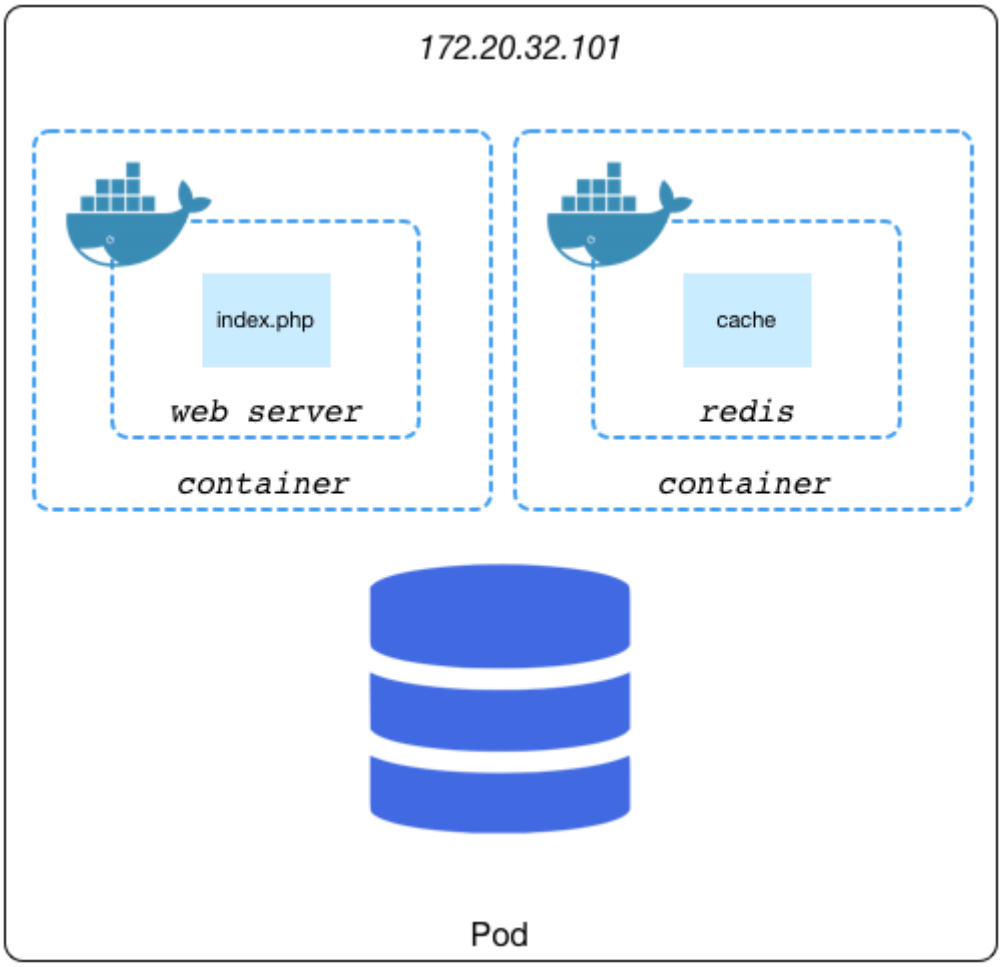
In this scenario, two containers may have low latency requirements between each other. Two container deployed in the same pod will reside on the same Kubernetes Node.



In this scenario, two containers may have low latency requirements between each other. Two container deployed in the same pod will reside on the same Kubernetes Node.

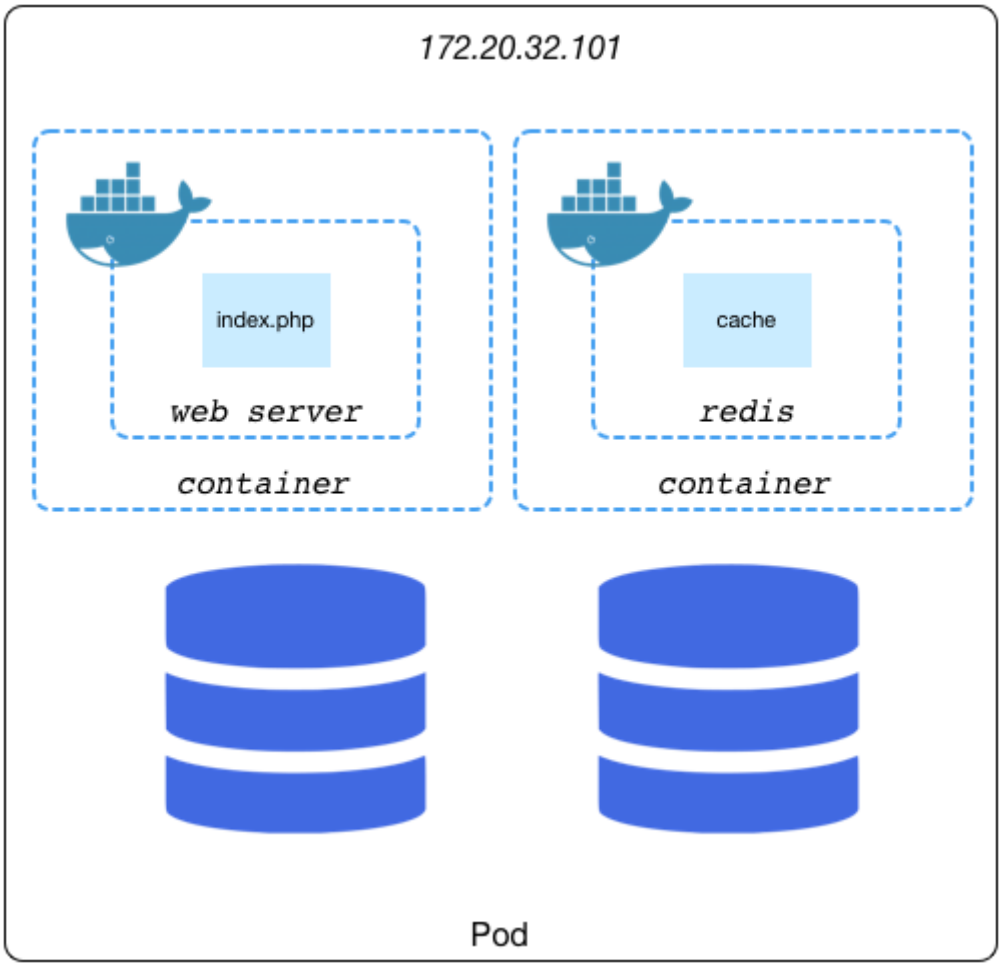
Appendix C.

In this scenario, two containers may have low latency requirements between each other and may require a persistent volume claim. Two container deployed in the same pod will reside on the same Kubernetes Node.



Appendix D.

In this scenario, two containers may have low latency requirements between each other and each may require persistent volumes claim. Two container deployed in the same pod will reside on the same Kubernetes Node.



Conclusion

Kubernetes Namespaces

Lab Objectives

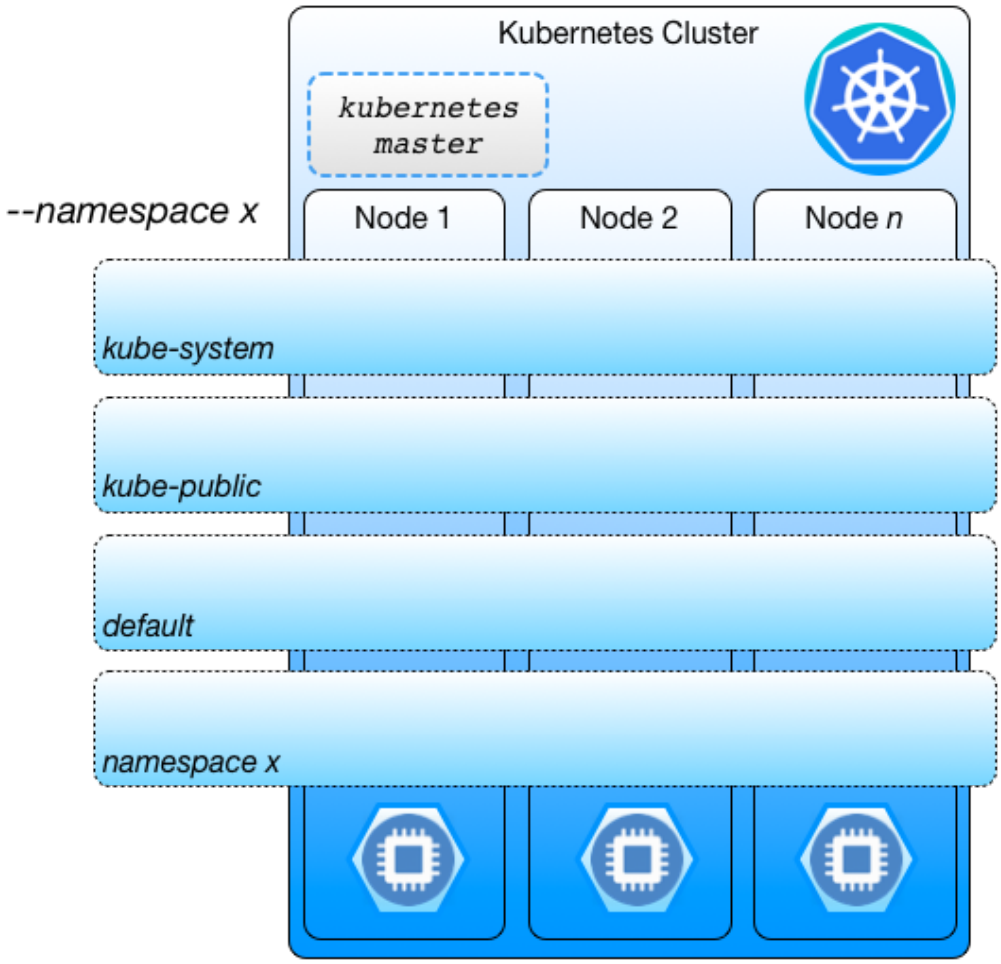
In this lab, we'll interact with Kubernetes namespaces and create a virtual cluster.

Lab Structure Overview

The lab follows the general structure, an overview, some steps, and a closing optional challenge.

Lab Overview

In this lab, we'll create a Kubernetes namespace. A namespace is a virtual clusters backed by the same physical cluster. Namespaces are used in enviroments where many users, spread across many teams, share the same physical cluster, as depicted in the diagram below.



Kubernetes clusters with tens of users should consider not using namespaces. An example might be a dedicated cluster for a development team.

Step 1. Get into a terminal

Open a terminal console (iTerm, Terminal, PowerShell, Ubuntu Bash, Git Bask, etc)

Step 2. Review the default namespaces for the environment

Run the following command to get the basic namespaces configured by default.
`$ kubectl get namespace`

```
NAME          STATUS    AGE
default       Active    3h
kube-public   Active    3h
kube-system   Active    3h
```

Step 3. Using the `-o yaml` switch

Run the following command to get detailed information on the namespace.
`$ kubectl get namespace default -o yaml`

```
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: 2017-05-16T04:10:34Z
  name: default
  resourceVersion: "15"
  selfLink: /api/v1/namespacesdefault
  uid: 9bc3e46c-39ed-11e7-9968-08002774bad8
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

Step 4. Using the `-o json` switch

If preferred; changing the `-o yaml` switch to `-o json` will output in json.
`$ kubectl get namespace default -o json`

```
{
    "apiVersion": "v1",
    "kind": "Namespace",
    "metadata": {
        "creationTimestamp": "2017-05-16T04:10:34Z",
        "name": "default",
        "resourceVersion": "15",
        "selfLink": "/api/v1/namespacesdefault",
        "uid": "9bc3e46c-39ed-11e7-9968-08002774bad8"
    },
    "spec": {
        "finalizers": [
            "kubernetes"
        ]
    },
    "status": {
        "phase": "Active"
    }
}
```

Step 5. Review existing pods in the `default` namespace

Let's take a look at what we have running. We should see the pod that was created during the dashboard section.

`$ kubectl get pods`

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3146706294-17gjb   1/1       Running   0          6s
```

By default, Kubernetes assumes you are referring to the --namespace default when running commands. The below command will yield the same result as the above command.

```
$ kubectl get pods --namespace default
```

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3146706294-17gjb   1/1       Running   0          18m
```

Step 6. Review namespaces

Let's take a look at the pods running in the `kube-system` namespace. To do this, we'll use the `--namespace` switch with the `kubectl get pods` command.

```
$ kubectl get pods --namespace kube-system
```

The below output is trauncated.

```
NAME                                 READY     STATUS    RESTARTS   AGE
dns-controller-484843949-2k1kp       1/1       Running   0          54m
etcd-server-events-ip-172-20-54-1    1/1       Running   0          54m
etcd-server-ip-172-20-54-106.us-wes  1/1       Running   0          54m
kube-apiserver-ip-172-20-54-106.us   1/1       Running   0          55m
```

Step 7. Create a new namespace

In this step, we'll create a `development` namespace for hosting development workloads.

```
$ kubectl create namespace development
```

```
namespace "development" created
```

Step 8. Review the new namespace

Review the new namespace

```
$ kubectl get namespaces
```

```
NAME          STATUS    AGE
default       Active    1h
development   Active    12m
kube-public   Active    1h
kube-system   Active    1h
```

Let's also show the pods currently in the development namespace.

```
$ kubectl get pods --namespace development
```

```
No resources found.
```

This should return "**no resources founds.**" If it returns anything else, check that the `--namespace` switch was used.

You can also use the `-o yaml` or `-o json` switch to get more detailed information when running the `get` command with kubectl

Step 9. Deploy and validate a workload

Let's deploy an nginx workload into the namespace.

```
$ kubectl run nginx2 --image nginx:1.11-alpine --namespace development
```

```
deployment "nginx2" created
```

```
$ kubectl get pods --namespace development
```

```
NAME                      READY     STATUS              RESTARTS   AGE
nginx2-2483678633-b65kx   1/1       Running             0          21s
```

Step 10. Set the default namespace to `development`

Let's see what our default namespace is set to.

```
$ kubectl config view | grep namespace
```

```
    namespace: default
```

Let's change the default namespace to use the `development` namepsace.

```
$ kubectl config set-context $(kubectl config current-context) --namespace=development
```

```
Context "kubernetes.kubectl.guru" set.
```

`$(kubectl config current-context)` can be replaced with the name Kubernetes cluster.

We can review this by reviewing the config.

```
$ kubectl config view | grep namespace
```

```
    namespace: development
```

Step 11. Change the default namespace back to `default`

Let's roll back to using the default namespace.

```
$ kubectl config set-context $(kubectl config current-context) --namespace=default
```

Remember, a namespace is literally a virtual cluster inside a Kubernetes cluster.

When using a shared Kubernetes cluster with a dedicated namespace, consider change the default namespace.

Challenge

Create a namespace called `QA` and use the Kubernetes Dashboard to deploy a workload into the `QA` namespace.

Conclusion

Deployments

Lab Objectives

Deployment declarations used by Kubernetes allows you to create app deployments and update app deployments.

Lab Structure Overview

Lab Overview

Step 1. Create a basic deployment

In this step, we'll create a basic nginx **Deployment Object** using Kubernetes and check the status of the rollout.

```
$ kubectl run nginx --image nginx
```

```
deployment "nginx" created
```

Now check the status of the Deployments rollout using

```
$ kubectl rollout status deployment nginx
```

This should return the following:

```
deployment "nginx" successfully rolled out
```

For larger deployments, the rollout status will show the status of the rollout.

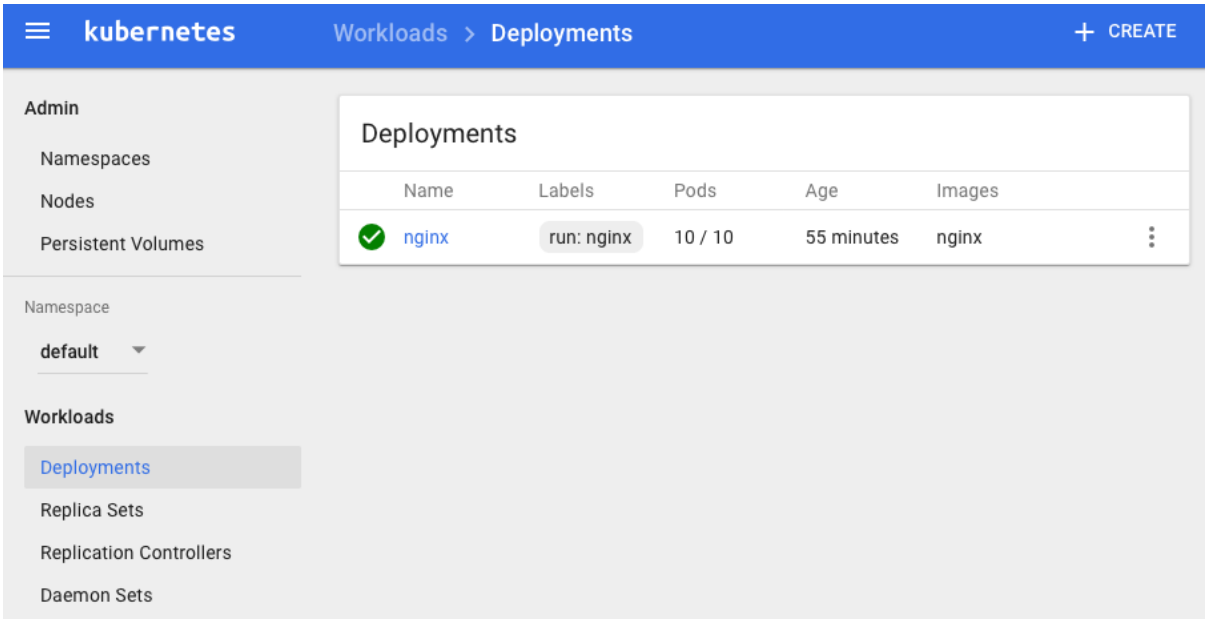
Let's take a quick look at the Kubernetes objects created.

```
$ kubectl get all
```

```
NAME                        READY     STATUS    RESTARTS   AGE
po/nginx-3447197284-whqkl   1/1       Running   0          4m

NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   100.64.0.1   <none>        443/TCP   9m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/nginx   1         1         1            1           4m

NAME                  DESIRED   CURRENT   READY     AGE
rs/nginx-3447197284   1         1         1         4m
```

Three Kubernetes objects were created

You can also use the dashboard to see the deployment



Step 2. Looking at existing deployments

```
$ kubectl get deployments
```

```
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx           1         1         1            0         10 m
```

The above commands will only output the deployments of the selected namespace, which by default is "default."

When wanting to output the deployments of other namespaces uses the `--namespace` switch

Run the following to see the obecti

To look at their values; run `kubectl get deployments -o json`

DESIRED

This shows the number of desired replicas, this is defined by the (`.spec.replicas`)

```
    "spec": {
        "replicas": 1,
        "selector": {
            "matchLabels": {
                "run": "nginx"
            }
```

CURRENT

This shows the number of current replicas, this is defined by the (`.status.replicas`)

```
    "status": {
        "replicas": 1,
```

UP-TO-DATE

This shows the number of up-to-date replicas in the deployment (`.status.updatedReplicas`)

```
    "status": {
        "updatedReplicas": 1
```

AVAILABLE

This shows the number of pods available for being updated (`.status.availableReplicas`)

```
    "status": {
        "unavailableReplicas": 1,
```

AGE

This shows the age of the deployment.

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-05-23T16:17:48Z
```

Step 3. Looking deeper at a deployment

Looking at more detail of a Kubernetes Deployment Object.

```
$ kubectl describe deployment <deployment_name>
```

```
$ kubectl describe deploy/<deployment_name>
```

Both above methods achieve the same result.

Truncated output of the above commands

```
Name:           nginx
Namespace:      default
CreationTimestamp:  Tue, 23 May 2017 10:56:15 -0700
Labels:         run=nginx
Annotations:        deployment.kubernetes.io/revision=1
Selector:       run=nginx
Replicas:       1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:       RollingUpdate ...
```

The describe command doesn't work with the -o (output) switch. This is namely due to how the describe commands collects the data points required to present the data.

Step 4. Scaling the deployment

By default, all deployments are deployed with a ReplicationController and a `--replica` value of 1. Notice in the previous step, **Replicas** has a value of 1. Thus, let's add more `--replicas` to the nginx deployment.

```
$ kubectl scale deployment nginx --replicas 10
```

```
deployment "nginx" scaled
```

Let's take a closer look at the deployment.

```
$ kubectl get deployment nginx
```

```
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     10        10        10           10          28m
```

Let's delete the deployment `nginx`

```
$ kubectl delete deployment nginx
```

```
service "kubernetes" deleted
```

Step 5. Create a Deployment Manifest

Create a file called `nginx.yaml` and populate it with the following content.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11-alpine
        ports:
        - containerPort: 80
```

Let's create a deployment from it using the *kubectl create* command.

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

```
deployment "nginx" created
```

Writing manifest is the better means to deploying Kubernetes objects.

Step 6. Check the status of the manifest

Run a check to see the status of the `rollout`

```
$ kubectl rollout status deployment nginx
```

```
deployment "nginx" successfully rolled out
```

```
$ kubectl get all
```

```
NAME                         READY     STATUS    RESTARTS   AGE
po/nginx-2371676037-4cpr0    1/1       Running   0          15m

NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   100.64.0.1   <none>        443/TCP   25m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/nginx   1         1         1            1           15m

NAME                   DESIRED   CURRENT   READY     AGE
rs/nginx-2371676037    1         1         1         15m
```

The same objects were created by the manifest.

Step 7.

Step 8. Resource Management (CPU/RAM)

There are many cases where resources may need to be limited; in these cases, we can add the resources spec to the `spec: container` section of the `yaml` file.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11-alpine
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "500m"
            memory: "128Mi"
```

Defining resource is especially usefully in defining the minimum required resources for a pod. In the above example, the container is only deployable to a system with .5 CPU (1/2 Core). Once deployed to a system with .5 core available, the container will consume all CPU until another container is deployed into the pod.

This helps the scheduler avoid resource shortages.

Step 9. Labels

Let's create some nginx pod with labels, first from the command-line and next from a `yaml` file. Let's create one for prod and one for test. Let's also define the app, in this case, let's use `nginx`.

```
$ kubectl run nginx-prod \
    --image=nginx:1.11-alpine \
    --replicas=3 \
    --labels="app=nginx,env=prod"
```

```
$ kubectl run nginx-test \
    --image=nginx:1.9.10 \
    --replicas=2 \
    --labels="app=nginx,env=test"
```

Labels are the only way to group Kubernetes objects.

```
$ kubectl get pods

$ kubectl get pods --selector="app=nginx"

$ kubectl get pods --selector="env=prod"

$ kubectl get pods --selector="env=test"
```

Let's create another pod that runs apache.

```
$ kubectl run httpd-test \
    --image=httpd:alpine \
    --replicas=2 \
    --labels="app=httpd,env=test"
```

Let's run the following to see the different ways a label can be queried.

First, let's select all pods where `app=nginx`

```
$ kubectl get pods --selector="app=nginx"
```

Now, let's select all pods where `app=httpd`

```
$ kubectl get pods --selector="app=httpd"
```

```
$ kubectl get pods --selector="env=test"
```

One or more labels can be used with the `--selector` switch.

```
$ kubectl get pods --selector="env=test,app=nginx"
```

Conclusion

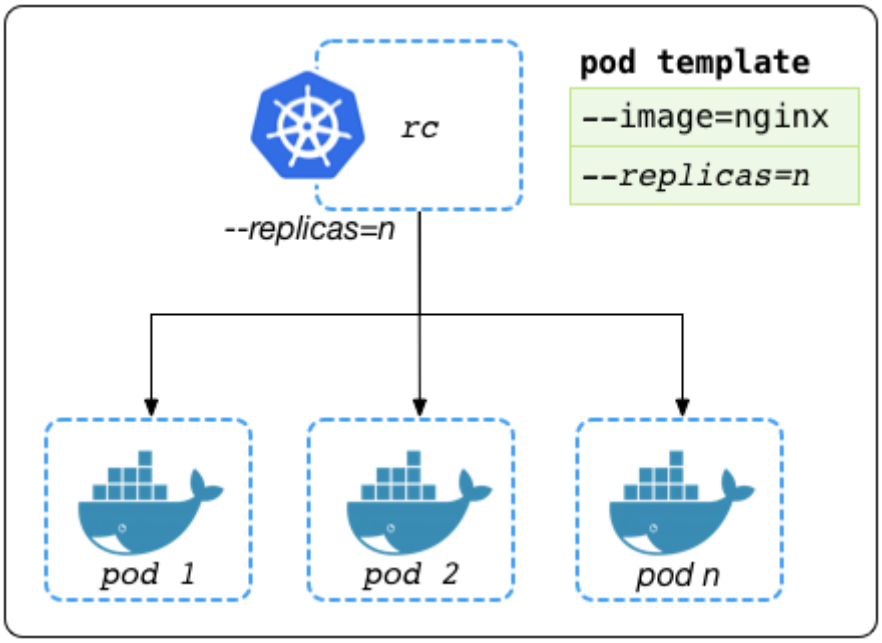
ReplicationControllers

Lab Objectives

Lab Structure Overview

Lab Overview

In this secton, we'll deploy a replication controller. A **ReplicationController** is defined in one of several ways. In the following steps, we'll configure a ReplicationController at runtime and from a file.



Step 1.

Step 2.

Step 3.

Step 4.

Step 5.

Step 6.

Step 7.

Step 8.

Step 9.

Step 10.

Step 11.

Conclusion

Services

Lab Objectives

ingress
egress

NodePorts
LoadBalancer

Lab Structure Overview

Lab Overview

In this section, we'll modify the `nginx deployment` to receive traffic from the Internet using the Kubernetes object, **service**. Services are the logical bridge between pods, other services, and users.

Step 1.

Show an output of `all` Kubernetes objects in the default namespace.

```
$ kubectl get all
```

```
NAME                        READY     STATUS    RESTARTS   AGE
po/nginx-3447197284-lm905   1/1       Running   0          1m

NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   100.64.0.1   <none>        443/TCP   13m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/nginx   1         1         1            1           1m

NAME                  DESIRED   CURRENT   READY     AGE
rs/nginx-3447197284   1         1         1         1m
```

There is already a service called `svc/kubernetes`, this service routes traffic to the apiserver (`/api/v1/namespaces/default/services/kubernetes`)

Step 2.

```
$ kubectl get services
```

```
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   100.64.0.1      <none>        443/TCP   20m
```

Like other Kubernetes objects, the describe command can help determine the config of a service.

```
$ kubectl describe svc/kubernetes
```

```
Name:           kubernetes
Namespace:      default
Labels:         component=apiserver
            provider=kubernetes
Annotations:        <none>
Selector:       <none>
Type:           ClusterIP
IP:         100.64.0.1
Port:           https   443/TCP
Endpoints:      172.20.33.17:443
Session Affinity:   ClientIP
Events:         <none>
```

Step 3.

Let's expose the nginx deployment to receive traffic.

Creating a service will create an endpoint for the pod. Endpoints can be seen by running `kubectl get endpoints`. ClusterIP is used by default.

```
$ kubectl expose deployment nginx --port=80 --target-port=80
```

```
service "nginx" exposed
```

Although the service is exposed, some additional options and switches need to be used to get the system working.

```
$ kubectl expose deployment nginx --port=80 --target-port=80 --type LoadBalancer
```

If a LoadBalancing service is not available (i.e. Minikube, Vagrant, etc), it may be required to use NodePort (`--type NodePort`)

Step 4.

```
$ kubectl get all
```

```
NAME                        READY     STATUS    RESTARTS   AGE
po/nginx-3447197284-lm905   1/1       Running   0          5m

NAME             CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   100.64.0.1      <none>        443/TCP   17m
svc/nginx        100.64.113.72   <none>        80/TCP    3s

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/nginx   1         1         1            1           5m

NAME                  DESIRED   CURRENT   READY     AGE
rs/nginx-3447197284   1         1         1         5m
```

The service is created but at the moment, the service is not routable.

Let's destroy the service

Step 5.

Step 6.

Step 7.

Step 8.

Step 9.

Step 10.

Step 11.

Conclusion

Health Checks

Lab Objectives

Lab Structure Overview

Lab Overview

Step 1.

Step 2.

Step 3.

Step 4.

Step 5.

Step 6.

Step 7.

Step 8.

Step 9.

Step 10.

Step 11.

Conclusion

Monitoring Kubernetes with Prometheus

Lab Objectives

In this lab we will deploy a fully functionally monitoring capability for Kubernetes using Prometheus, Node-Exporter, and Grafana.

Lab Structure Overview

- Deploy a configmap for Prometheus
- Deploy a deployment and service for prometheus
- Deploy a node-exporter deployment
- Deploy an updated configmap for Prometheus
- Deploy a deployment and service for grafana

Lab Overview

Step 1.

Open a terminal (iTerm, Terminal, PowerShell, Ubuntu Bash, Git Bash, etc)

Step 2.

Ensure the following files are in the `./lab/monitoring/` directory:

```
prometheus-configmap-1.yaml
prometheus-deployment.yaml
node-exporter.yaml
grafana-service.yaml
grafana-deployment.yaml
prometheus-configmap-2.yaml
```

Step 3.

Using the `kubectl` command, apply the configmap in the file `prometheus-configmap-1.yaml`

```
$ kubectl create -f prometheus-configmap-1.yaml
configmap "prometheus" created
```

Step 4.

Using the `kubectl` command, apply the deployment in the file `prometheus-configmap-1.yaml`

```
$ kubectl create -f prometheus-deployment.yaml
deployment "prometheus" created
```

Step 5.

Step 6.

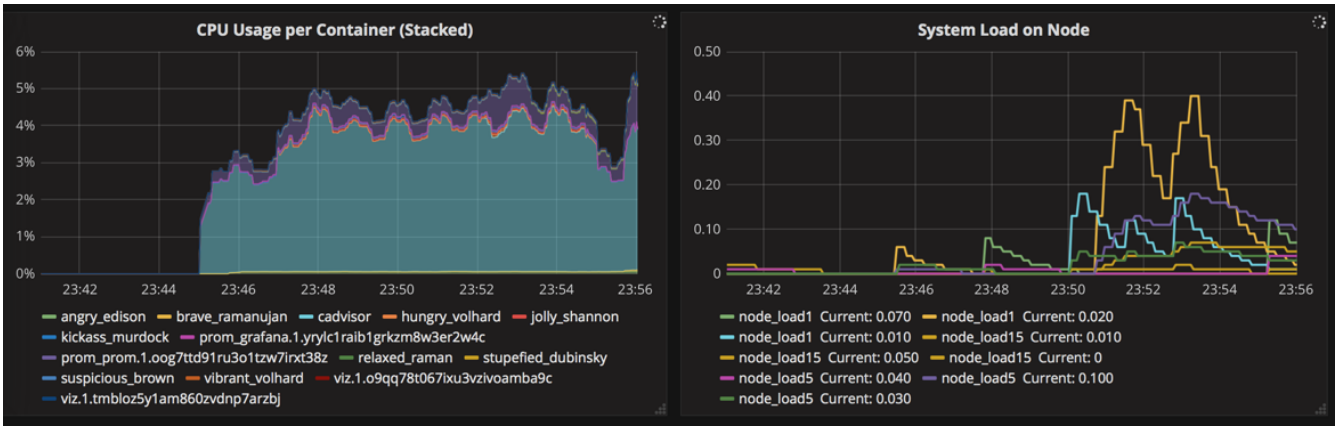
Step 7.

Step 8.

Step 9.

Step 10.

Step 11.



Conclusion

Application Deployments

Lab Objectives

Lab Structure Overview

Lab Overview

Step 1.

Step 2.

Step 3.

Step 4.

Step 5.

Step 6.

Step 7.

Step 8.

Step 9.

Step 10.

Step 11.

Conclusion