
Kubernetes - Basics

Jul 31, 2020

Contents:

1	Introduction	3
2	Creating your Kubernetes cluster	5
3	Kubernetes Terminology	11
4	Basic Kubernetes commands	13
5	Running our first pod	17
6	First deployment	19
7	Exposing services	21
8	Conclusions	25

Walkthrough of basic Kubernetes concepts

This tutorial will guide you through basic Kubernetes features and concepts.

1.1 Module Learning Objectives

This module will be fully interactive. Participants are **strongly encouraged** to follow along on the command line. After completing this module, participants should be able to:

- Create a 2-node Kubernetes cluster
- Learn basic commands for interacting with Kubernetes
- Deploy and inspect a running pod
- Create a templated deployment
- Scale a deployment
- Increase response throughput with a LoadBalancer

1.2 Why is this important?

Similar to Docker Swarm, Kubernetes is a container management system. It runs and manages containerized applications on a single system, or up to 5000 nodes in a distributed cluster.

Kubernetes enables many sophisticated features through simple templating:

- Basic autoscaling
- Long-running and batch services
- Overcommit our cluster while also removing low-priority jobs
- Run services with stateful data (databases etc.)
- Fine-grained permissions on resources

- Automating complex tasks (operators)

Utilizing Kubernetes will make your deployments both more reproducible, resilient, and performant.

1.3 Requirements

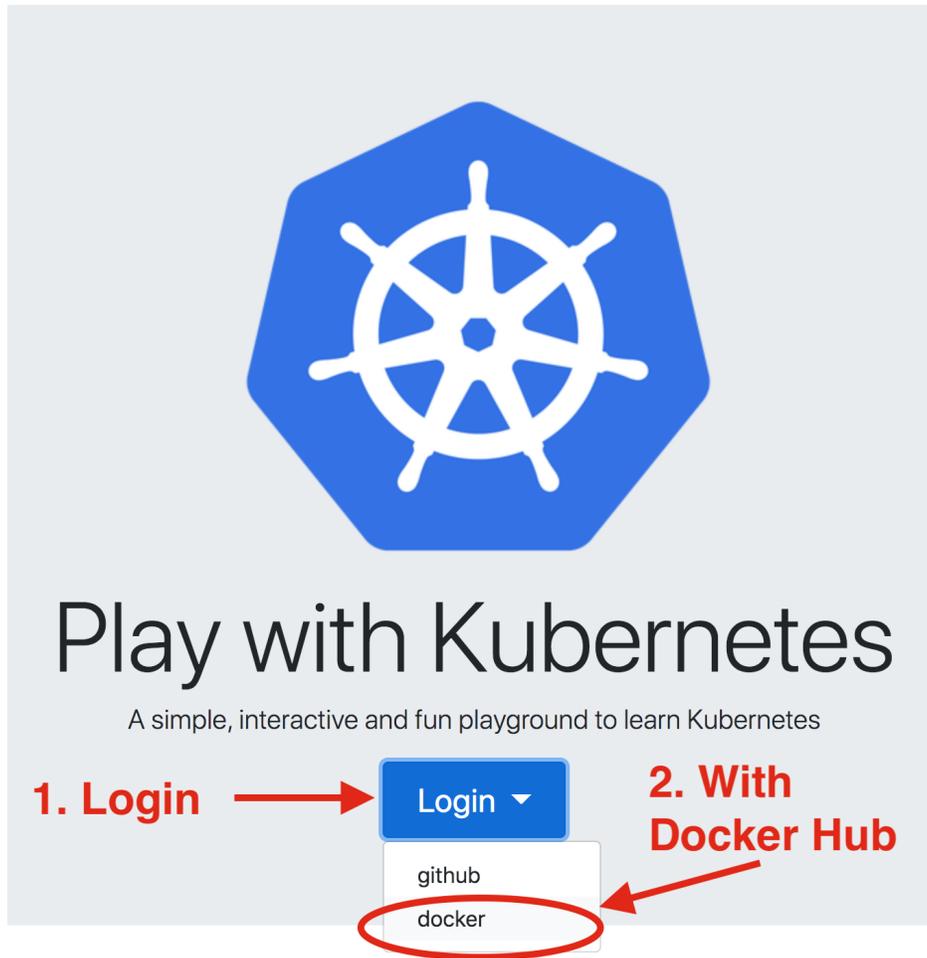
- Accounts
 - Docker Hub

Creating your Kubernetes cluster

In this section, we will launch and configure a 2-node Kubernetes cluster on labs.play-with-k8s.com.

2.1 Launching your k8 instance

Head over to <https://labs.play-with-k8s.com> and log in with your Docker Hub account.



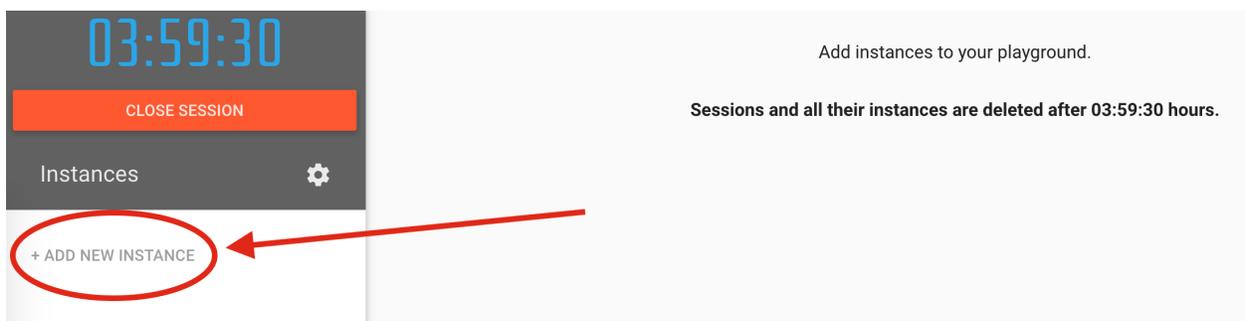
You should now have the option to “start” your remote Kubernetes (k8) instance.



You'll notice a timer in the upper-left, counting down the time left on your instance. This is a great free resource, but all changes will be lost between shutdowns. The pane on the left will list any VMs you launch in your instance, and the right pane will expose a terminal of the selected VM.

2.2 Creating your orchestrator

Now, create your first VM, which will serve as the orchestrator of your Kubernetes cluster, by clicking **ADD NEW INSTANCE**.



Once this is running, you'll be presented with a terminal. Run the first two suggested commands to provision your orchestrator and allow other VMs to join your cluster:

2.2.1 1. Initialize the orchestrator (this) node

```
kubeadm init --apiserver-advertise-address $(hostname -i) --pod-network-cidr 10.5.0.0/16
```

You may see some warnings after this command, no errors. This command also generates a command for registering other nodes to this Kubernetes cluster, which you should save.

The screenshot shows a cloud management interface. On the left, there's a sidebar with a clock showing 03:44:18, a 'CLOSE SESSION' button, and an 'Instances' section with a gear icon and '+ ADD NEW INSTANCE' button. Below that, a card shows an instance named 'node1' with IP '192.168.0.38'. On the right, a detailed view of the instance 'bshj9ruj_bshjcnuj2b7000bk2mpg' is shown, including its IP (192.168.0.38), memory usage (99.73%), CPU usage (41.82%), and a URL. A 'DELETE' button is visible. Below the instance details is a terminal window showing the output of the 'kubeadm init' command. The terminal output includes: '[addons] Applied essential addon: kube-proxy', 'Your Kubernetes control-plane has initialized successfully!', and instructions for starting the cluster. A red box highlights the 'kubeadm join' command: 'kubeadm join 192.168.0.38:6443 --token w3g1ow.162kqxu7vfzcg9m \ --discovery-token-ca-cert-hash sha256:db13c0b1056fc0314b91bda9efa6cf51428791f3655540d302b54dae0059a167'.

Each join command contains a unique key, so you should save yours and not rely on mine.

2.2.2 2. Initialize networking

```
kubectl apply -f https://raw.githubusercontent.com/cloudnativelabs/kube-router/master/daemonset/kubeadm-kuberouter.yaml
```

2.3 Creating your worker

Create a second VM to serve as the worker in your Kubernetes cluster by clicking **ADD NEW INSTANCE** again. Once that VM done initializing and you are presented with a terminal, run your unique `kubadm join` command.

This should complete fairly quickly, and you can confirm it worked by switching back to your orchestrator (`node1`) and running

```
kubectl get nodes
```

if everything was set up correctly, you should see output similar to the following.

NAME	STATUS	ROLES	AGE	VERSION
node1	Ready	master	19m	v1.18.4
node2	Ready	<none>	21s	v1.18.4

At this point, your 2-node Kubernetes cluster with one orchestrator and one worker is set up and ready to accept tasks. If you wanted to add another worker to your cluster, just repeat the steps in this sub-section. Kubernetes can scale up to 5,000 workers, and each worker can be a VM or a physical machine that “joins” the cluster.

Kubernetes Terminology

Now that you've seen a basic cluster, let's cover some terminology before jumping into commands.

Pod Smallest deployable unit. Consists of 1 or more containers. Kind of like "localhost".

Deployment Multiple pods.

Service Expose a pod or deployment to network.

Volume Attach storage.

Namespace Permissions-based grouping of objects.

Job Run a container to completion.

And more not covered today:

ConfigMap Store strings or files for pods to use.

Secret Encrypted configmap.

Ingress Expose HTTP+S routes to the network. Like a HTTP-specific Service.

Basic Kubernetes commands

On your orchestrator node (node1), lets run through the following commands to learn about what they do.

4.1 Kubernetes commands

4.1.1 Listing

Listing nodes

```
kubectl get nodes
```

Listing everything

```
kubectl get all --all-namespaces
```

The main program for interacting with Kubernetes is `kubectl`, which is a CLI tool that talks to the Kubernetes API.

Note: You can also use `--kubeconfig` to pass a whole config to `kubectl`

4.2 Getting information

Information can be queried with the `kubectl get` command. Which can query resources like nodes

```
kubectl get nodes
```

and auto-complete to the best matching target

```
kubectl get no  
kubectl get node
```

You can also increase the amount of information with the `-o wide` argument

```
kubectl get nodes -o wide
```

or output in standardized formats like:

- JSON `-o json`
- YAML `-o yaml`

Note: Outputting information to JSON is useful since it allows you to query information with `jq`

```
kubectl get nodes -o json | jq ".items[] | {name:.metadata.name} + .status.capacity"
```

4.3 Viewing details

The `kubectl get` command is great for listing resources, but details about each specific item can also be returned with `kubectl describe` which is used with the format

```
kubectl describe [type]/[name]
kubectl describe [type] [name]
```

We can get information about **node1** with

```
kubectl describe node node1
```

You can also get an explanation about different *types* of resources with

```
kubectl explain [type]

# What is a node?
kubectl explain node
# What is a service?
kubectl explain service
```

4.4 Exploring deployments

4.4.1 Services

A service is a pod or deployment exposed to a network. We can see that our cluster already has the API service running with

```
kubectl get services
```

A ClusterIP service is internal, meaning that it's only available from inside the cluster.

Note: The API requires authentication, so it returns a 403 error if you try to connect.

```
# Assuming this is the IP of your Kubernetes service
curl -k https://10.96.0.1
```

4.4.2 Running containers

Containers are manipulated through pods, and a pod is a group of containers:

- running together (on the same node)
- sharing resources (RAM, CPU; but also network, volumes)

Running pods can be listed with

```
kubectl get pods
```

You'll quickly find that there are no pods running in the default namespace.

4.4.3 Namespaces

Namespaces are a way to separate resources by named tag. Namespaces can be listed with

```
kubectl get namespaces
```

When we looked for pods, we queried the “default” namespace. We list pods running in specific name spaces with the `-n` argument.

```
kubectl -n kube-system get pods
```

Note: Information about these services can be found [here](#). The `READY` column indicates the number of containers in each pod, and pods with a name ending with `-node` are the main components (they have been specifically “pinned” to the orchestrator node)

Running our first pod

In this example we are going to:

1. Create a pod that *happens* to run a single alpine linux container
2. The container will run the `ping`
3. We'll ping Google's public DNS (8.8.8.8)

Note: Just to be clear, Kubernetes runs “pods”, not containers.

5.1 Starting the pod

Start the pod with `kubectl run` which will look similar to `docker run`.

```
kubectl run pingpong --image alpine ping 8.8.8.8
```

Kubernetes should only report that `pod/pingpong` was created. You can confirm this by listing all running pods.

```
kubectl get pods -o wide
```

It should have been started on our only worker node, `node2`.

5.2 Viewing output

If you remember, the `ping` command, by default, pings an address and prints the response time until it is terminated. That means our pod is still printing to standard output somewhere on the cluster. That output can be viewed with the `kubectl logs [type]/[name]` command.

```
kubectl logs pod/pingpong
```

You can also select specific parts of the output with:

- `--tail N` - View the last N lines of output

```
kubectl logs pod/pingpong --tail 3
```

- `--since N[unit]` - View logs since the last N hours (h), minutes (m), seconds (s)

```
kubectl logs pod/pingpong --since 10s
```

- `--follow` - Update the output in real time (similar to watch)

```
kubectl logs pod/pingpong --tail 1 --follow
```

5.3 Deleting the pod

Pods can be deleted with the `kubectl delete` command

```
kubectl delete pod/pingpong
```

First deployment

While running pods with `kubectl run` is both convenient and familiar, you have to use templated deployments to take advantage of advanced Kubernetes features.

6.1 Pingpong deployment

We originally deployed the pingpong pod using

```
kubectl run pingpong --image alpine ping 8.8.8.8
```

We can create a deployment with the same behavior by creating the `pingpong.yaml` config

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: pingpong
5 spec:
6   selector:
7     matchLabels:
8       app: ping-app
9   replicas: 1
10  template:
11    metadata:
12      labels:
13        app: ping-app
14    spec:
15      containers:
16      - name: pingpong
17        image: alpine
18        command: ["ping"]
19        args: ["8.8.8.8"]
```

and submitting it for deployment.

```
kubectl apply -f pingpong.yaml
```

You'll notice that it created a deployment, replicaset, and a pod.

```
kubectl get all
```

6.2 Scaling the deployment

Now that our pingpong application is a proper deployment, we can scale it up with `kubectl scale`

```
kubectl scale --replicas=2 deployment.apps/pingpong
```

If you look at everything running, you'll see that there are now 2 replicas and 2 pods.

```
kubectl get all
```

However, the longer running pod will have a longer output. Each pod has a randomized name, so you'll need to fill in your own.

```
kubectl logs [pod1] | wc -l  
kubectl logs [pod2] | wc -l
```

You can also view the last few lines of each pod's log by selecting by app

```
kubectl logs -l app=ping-app --tail 1
```

6.3 Removing your deployment

Once done, deployments are removed in the same way as pods, but all pods and replicaset will be removed as well.

```
kubectl delete deployment.apps/pingpong
```

Exposing services

The `kubectl expose` command creates a *service* for existing pods. A *service* is a stable address for a pod or deployment.

A *service* is always required to make an external connection, and once created, `kube-dns` will allow us to resolve it by name (i.e. after creating service *hello*, the name *hello* will resolve to something).

7.1 Basic service types

ClusterIP (default type) A virtual IP address is allocated for the service (in an internal, private range), which is reachable only from within the cluster (nodes and pods). Code can connect to the service using the original port number.

NodePort A port is allocated for the service (by default, in the 30000-32768 range) that port is made available on all our nodes and anybody can connect to it. Code must be configured to connect to that new port number

LoadBalancer An external load balancer is allocated for the service, and sends traffic to a `NodePort`.

ExternalName The DNS entry managed by `kube-dns` will just be a *CNAME* to a provided record. No port, no IP address, no nothing else is allocated.

7.2 Exposing a port

The docker container `gzynda/sleepy-server` serves a webpage on a specified port after sleeping a specified number of seconds. The configuration file `sleepy.yaml` creates a deployment and exposes it outside the cluster on port 80. It then creates the `sleepy-svc` *service* to expose port 80 of the deployment to the outside world.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: sleepy
5 spec:
```

(continues on next page)

(continued from previous page)

```

6  selector:
7    matchLabels:
8      app: sleepy-server
9  replicas: 1
10 template:
11   metadata:
12     labels:
13       app: sleepy-server
14   spec:
15     containers:
16     - name: sleepy-server
17       image: gzynda/sleepy-server:latest
18       command: ["sleepy-server.py"]
19       args: ["-p", "80"]
20       ports:
21       - containerPort: 80
22 ---
23 apiVersion: v1
24 kind: Service
25 metadata:
26   name: sleepy-svc
27   labels:
28     app: sleepy
29 spec:
30   type: LoadBalancer
31   ports:
32   - protocol: TCP
33     port: 80
34   selector:
35     app: sleepy-server

```

Get the external port of your deployment by listing your available services and looking for the (second) port after 80.

IP
192.168.0.13

Memory
80.77% (3.155GiB / 3.906GiB)

CPU
13.43%

URI
ip172-18-0-10-bshpjbed7q4g0087s840.direct.labs.play-wi

DELETE

External IP URL

External PORT

```

[node1 ~]$ kubectl get services
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes    ClusterIP     10.96.0.1     <none>         443/TCP         47m
sleepy-svc    LoadBalancer 10.99.228.193 <pending>     80 31395/TCP      4m29s
[node1 ~]$

```

Paste both the URL and PORT into your web browser in the URL:PORT format to view the running webpage from this deployment.

You'll notice that the webpage displays the name of the host it is being served from, sleeps for half a second, and then finishes printing "Hello World!". If the page takes around half a second to complete loading every time it is requested,

how many pages can be served per second?

We can confirm this by stress-testing the web server with `siege` using 1 worker (`-c1`), 0 delay (`-d0`), and 10 tries (`-r10`). I recommend running the docker container directly for this use case.

```
$ docker run --rm -t yokogawa/siege -d0 -r10 -c1 [node2 IP]:[external PORT]

** SIEGE 3.0.5
** Preparing 1 concurrent users for battle.
The server is now under siege..      done.

Transactions:                10 hits
Availability:                 100.00 %
Elapsed time:                 5.02 secs
Data transferred:            0.00 MB
Response time:               0.50 secs
Transaction rate:            1.99 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                 1.00
Successful transactions:      10
Failed transactions:          0
Longest transaction:         0.51
Shortest transaction:         0.50
```

Which should confirm that the deployment can serve ~2 responses per second, since it sleeps for half a second when rendering the page.

7.3 Scaling the web server

Assuming we want more people to enjoy our sleepy server, you can serve more concurrent requests by scaling up the deployment.

```
kubect1 scale --replicas=2 deployment.apps/sleepy
```

The LoadBalancer will automatically balance the traffic between pods. If you **refresh** the webpage, you should notice that the host name periodically changes.

You should also see an increased transaction rate if you re-run `siege`. Just be sure to increase the number of concurrent users to at least 3.

```
$ docker run --rm -t yokogawa/siege -d0 -r10 -c3 [node2 IP]:[external PORT]

** SIEGE 3.0.5
** Preparing 3 concurrent users for battle.
The server is now under siege..      done.

Transactions:                30 hits
Availability:                 100.00 %
Elapsed time:                 9.04 secs
Data transferred:            0.00 MB
Response time:               0.87 secs
Transaction rate:            3.32 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                 2.89
Successful transactions:      30
Failed transactions:          0
```

(continues on next page)

(continued from previous page)

Longest transaction:	1.51
Shortest transaction:	0.50

Try scaling up the server to more replicas and benchmarking your transaction rate again.

At this point you have

- Created a 2-node Kubernetes cluster
 - 1 Orchestrator
 - 1 Worker
- Learned basic commands for interacting with Kubernetes
 - `kubectl get`
 - `kubectl log`
 - `kubectl run`
 - `kubectl apply`
 - `kubectl delete`
- Deployed and inspected the log of a running pod
 - `kubectl run [pod name] --image=[image] [args]`
 - `kubectl logs pod/[pod name]`
- Created a deployment from a template
 - `kubectl apply -f [template]`
- Scaled up the number of replica pods in your deployment to increase response throughput
 - `kubectl scale --replicas=2 [deployment]`

8.1 Continuing your education

8.1.1 Kubernetes

Documentation <https://kubernetes.io/docs/home/>

Tutorials <https://kubernetes.io/docs/tutorials/>

Cheat Sheet <https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

TACC Institute <https://ancantu.github.io/SCICLD2019/docs/kubernetes/kubernetes.html>

8.1.2 TACC Tutorials

- [Reproducible Science](#)
- [Containers](#)
- [TAPIS API](#)
- [Workflow Automation](#)