Chapter 2

Kubernetes basics

2019-08-19

This chapter introduces kubernetes, what it is, basic terminologies and the key concepts. You will learn most of the frequently referred components in kubernetes architecture. This chapter also provides some examples in kubernetes cluster environment to demonstrate the basic ideas about basic kubernetes objects.

# What is kubernetes

starting with offical description (<https://kubernetes.io/>) :

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

this tells a few important facts about kubernetes:

* an open-source project initiated by google
* a mature and stable product
* an orchestration tool
* a platform dealing with containers in a higher level

kubernetes was created by a group of engineers in google in 2014, with a design and development model influenced by Google’s internal system named "Borg" . Kubernetes defines a set of "building objects" which collectively provides mechanisms that orchestrates containerized applications across a distributed cluster of nodes, based on system resources (CPU, memory or other custom metrics). Kubernetes hides the complexity of managing a group of containers by providing REST APIs for the required functionalities.

In simple words, container technologies like docker provides you the capability of packaging and distributing containerized applications, while an orchestration system like kubernetes allows you to deploy and manage the dockers in a relatively higher level and a much easier way.

**Note**

* "Borg" is still being used in google internally today
* in many document kubernetes is frequently abbreviated as "k8s" (K - eight characters - S),
* the "current" (as of the writing of this book) major release is v1.14.

In chapter one you’ve learned docker has been a prevailing and pretty mature container technology. So naturely you may wonder why you need kubernetes.

Technically speaking, kubernetes works in a relatively higher level than dockers. what does that mean exactly? when you compare kubernetes with docker, One analogy is to compare python with C language. C is powerful enough to build almost everything including the whole bunch of fundamental OS components and APIs. but you in practice you probably would prefer to write scripts to automate tasks in your work using python much more than using C. with python most often you only need to think of which existing module already provides the magic you need, import it in your application and then quickly focus on how to use the feature to get your things done. you rarely need to worry about the low-level system API calls and hardware details.

another analogy is TCP/IP Internet protocols. when you develop a file transfer tool like ftp, naturely you prefer to start your work based on TCP socket instead of raw socket. with TCP socket you are seating on top of the TCP protocol, which provides a much more solid fundation that has all of the built-in reliability features like error detection, flow and congestion control, retransmission and so on. What you need to consider is how to deliver the data from one end and receive it from the other end. with raw socket you are working on IP protocol and even lower layer, so you have to consider and implement all of the reliability features before you can even start to work on the file transfer features of your tool.

back to our topic of kubernetes, Assuming you want to run multiple containers across multiple machines, you will have a lot of work to do if you interact with docker directly. at least the following tasks should be in your "worry list":

* login different machines and Spawning containers across the network
* Scaling up or down by adding or removing containers when demand changes
* Keeping storage consistent with multiple instances of an application
* Distributing load between the containers running in different node
* Launching new containers on different machines if something fails

you will quickly find that doing all of this manually with docker will be overwhelming. with the high-level abstractions and the objects representing them in kubernetes API, all of these tasks become much easiler.

**Note**

kubernetes is not the only tool in its kind, docker has its owen orchestration tool named "swarm". this book will focus on kubernetes.

# Kubernetes Architecture and components

in a Kubernetes cluster there are two type of nodes, each running a very well-defined set of processes:

* head node: called "master", or "master node", the head and brian that does all thinking and decisions, all of intelligence are located here.
* worker node: called "node", or "minion", the arms and feet that conduct the workforce.

The "nodes" are controlled by the "master" and in most of the time you will only need to talk to master .

One of the most common interface between you and the cluster is a command-line tool kubectl. It is installed as a client application either in the same "master" node or in a seperate machine like in your PC. Regardless of where it is, it can talk to the master via the REST-API exposed by the master.

later you will read example of using kubectl to create kubernetes objects. For now just remember: Whenever you are working with "kubectl" command, you’re communicating with the cluster’s "master".

**Note**

the term "node" may sound semantically ambiguous - it could mean two things in the context of this book. Usually a "node" refers to a logical unit in a cluster - something we call a "server", which can be either physical server or virtual machine. in context of kubernetes clusters, a "node" often specifically refers to a "worker node".

**Note**

you rarely need to "bypass" the master and work with nodes directly. but you can login to node and run all docker command to check running status of containers. there is an example showing this later in this chapter.

## Kubernetes "master"

A kubernetes "master node", or "master", is like one’s head and brian. in the cluster master provides the "control plane" that makes all of the global decisions about the cluster.

for example, when you need the cluster to spawn a container, the master will decide which node to dispatch the task and spawn a container. this procedure is called "scheduling".

master is also responsible for maintaining the desired state for the cluster. when you give an order "for this web server make sure there are always 2 containers backing up each other!", the master will keep monitor the running status and spawning new container anytime when the number of the web server containers in "running" status becomes less than 2 due to any failures.

The master is also responsible for other many jobs.

Typically you only need a single master node in the cluster, however, the master can also be replicated for higher availability (HA) and redundancy.

the master’s functions is implemented by a collection of processes running in node. The processes in a master node providing the primary features are:

* **kube-apiserver**: front-end of the control plane, providing REST APIs
* **kube-scheduler**: do the "scheduling": decide where to place the containers depending on system requirement (CPU, memory, harddisk, etc) and other custom parameters or constraints (e.g. affinity specification)
* **kube-controller-manager**: the single process implementing most of the different type of "controllers", which makes sure that the state of the system is what it should be. some controller examples:
  + Replication Controller
  + ReplicaSet
  + Deployment
  + Service Controller
* **etcd**: database to store the state of the system.

**Note**

for the sake of simplicity a few other components are not listed (e.g. **cloud-controller-manager**, **DNS server**, **kubelet**). they are not trival negligible components, but skipping them for now does not stop you from understanding the kubernetes basics.

## Kubernetes "node"

nodes in a cluster are the machines that run the user end applications. in production there can be dozens or hundreds of nodes in one cluster depending on the designed scales. nodes are the real workforce under the hood provided by a cluter. usually all of the containers and workloads are running in nodes. A "node" runs following processes:

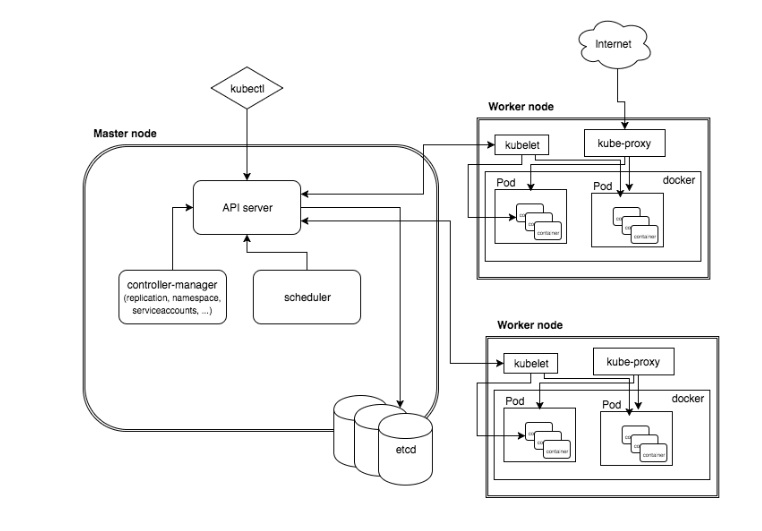
* **Kubelet**: the Kubernetes agent process that runs on master and all the nodes. it interacts with master (through kube-apiserver process) and manage the containers in local host.
* **kube-proxy**: process that implements "kubernetes service" (will introduce in chapter three) using linux iptable in the node
* **container-runtime**: local container - mostly 'docker' in today’s market, holding all of the running "dockerized" applications.

**Note**

the name "proxy" may sound confusing for kubernetes beginners. it’s not really a "proxy" in current kubernetes architecture. kube-proxy is a system that manipulates linux IP tables in that node so that the the traffic between the pods and the nodes will flows correctly.

## kubernetes work flow

after you get some basic idea about the master and node and the main processes running in each, it is time to look at how things works together in figure 2.1



kubernetes architecture

At the top behind kubectl is where you are. via kubectl commands you talk to kubernetes "master", which manages the 2 "node" boxes on the right. it interacts with the master process "kube-apiserver" via its REST-API exposed to the user and other processes in the system.

Now let’s say you send some kubectl commands - something like kubectl create  
x, to spawn a new container. You can give details about how exactly you want your container to be spawned along with the running behaviors. the container specifications can be provided either as kubectl command line parameters, or options and values defined in a config file. You will read an example on this shortly.

1. The kubectl client will first translate your CLI command to one more REST-API call(s) and send to "kube-apiserver".
2. After validating these REST-API calls, "kube-apiserver" understands the task and calls "kube-scheduler" process to select one "node" from all 3 available ones to execute the job. this is the scheduling procedure.
3. Once "kube-scheduler" returns the "target node", "kube-apiserver" will dispatch the task with all of the details describing the task.
4. "kubelet" process in the target node receives the task and talks to the container engine, for example the "docker engine" in figure 2.1, to spawn a container with all provided parameters.
5. This job and its specification will be recorded in a centralized database etcd. its job is to preserve and provide access to all data of the cluster.

**Note**

actually a master can be also a fully-featured node and carry pods workforce just like a node does. therefore, kubelet and kubec proxy components existing in node also exists in master. in the figure above we didn’t include these components in master, just to give a simplified conceptual seperation of master and node. in your setup you can use command kubectl get  
pod --all-namespaces -o wide to list all pods with their location. pods spawned in master is usually running as part of the kubernetes system itself - typically within kube-system namespace. kubernetes namespace will be discussed in chapter 3.

Of course This is just a very simplified work flow, but you get the basic idea. In fact with the power of kubernetes you rarely need to work with containers directly. you will work with some higher level objects which, hide most of the low level operation and details and present the task in a higher level and much simpler form.

for example, in figure 2.1 when you give the task to spawn containers, instead of saying:

"create two containers and make sure to spawn new ones if either one would fail"

in practice you just say:

"create a RC object ('replica controller') with replica two".

what will happen now is that once the 2 docker containers are up and running, kube-apiserver will interact with kube-controller-manager to keep monitoring the job status, and take all necessary actions to make sure the running status is what it was defined. for example you will observe that if any one of two docker containers goes down, a third container will be spawned and the broken one will be removed automatically.

the 'RC' in this example, is one of the objects that is provided by kubernetes kube-controller-manager process. The kubernetes objects provide an extra layer of abstraction that gets the same (and usually more) work done under the hood, in a simpler and clean way. Furthermore, because you are working in a higher level and staying away from the low level details, kubernetes sharply reduces your overall deployment time, brain effort, and troubleshooting pains.

The small "cost" of working in a level higher than docker engine is to understand a few extra "kubernetes objects".

you will read more about kubernetes objects in the next section.

## kubernetes objects

Now you understand the role of 'master' and 'node' in a kubernetes cluster, and in figure 2.1 you see how a basic workflow looks. now let’s start to look at more kubernetes "objects" in the kubernetes architecture.

Kubernetes’s objects represent:

* deployed containerized applications and workloads
* their associated network and disk resources
* other information about what the cluster is doing.

the most oftenly used objects are:

* basic Kubernetes objects
  + Pod
  + Service
  + Volume
  + Namespace
* higher-level objects (Controllers):
  + ReplicationController
  + ReplicaSet
  + Deployment
  + StatefulSet
  + DaemonSet
  + Job

**Note**

"high-level" objects are build upon the basic objects. They provide additional functionality and convenience features.

in the frontend, kubernetes get all things done via a group of "object". with kubernetes you only needs to think of how to describe your task in the config file of the objects, no need to worry about how it will be implemented in container level. "under the hood", kubernetes interact with the container engine to coordinate the scheduling and execution of containers on Kubelets. The container engine itself is responsible for running the actual container image (e.g. by 'docker build').

you will read more about each object and their magic power with examples in chapter 3. later in this chapter we’ll look at the the most fundamental object: POD.

# Building Kubernetes POD

"POD" is the first kubernetes object you will learn. the kubernetes website describe a "pod" as:

A pod (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers

this brings 2 facts:

* basically pod is nothing but a group of containers
* all containers in a pod shares storage and network resources.

what is the benefit of using "pod" comparing with the old way of dealing with each individual containers? considering a simple usage case that you are deploying a web service with docker. you will need not only the frontend service, e.g. an apache server, but also some "supporting services" like a database server, a logging server, a monitoring server, etc. each of these supporting services needs to be running in its own docker. so essentially you will find yourself always working with a group of docks whenever "a web service" docker is needed. In production the same scenario applys to most of the other docker service as well. eventually you will ask: is there a way to group a bunch of docker containers in a higher-level "unit", so you only need to worry about the low-level inter-docker interaction details once?

"pod" gives the exact higher-level abstraction you are asking for. it wraps one or more containers into one object. If your web service becomes too popular and a single pod instance can’t carry the load, with the help of other objects (RC, deployment) you can replicate and scale up and down the same group of containers (now in the form of one pod object) very easily - normally in a few seconds. this sharply increased the deployment and maintenance efficiency.

besides that, containers in the same pod will share the same network space. Containers can easily communicate with other containers in the same pod as though they were on the same machine while maintaining a degree of isolation from others. you’ll see more about these advantages later.

now, let’s get your feet wet. we’ll look at how to use a config file to launch a "pod" in kubernetes cluster.

## YAML file for Kubernetes

First thing to look at is YAML. Along with many other many ways of configuring kubernetes, YAML is the "standard" format being used in kubernetes config file. YAML is widely used in a lot of software fields so mostly likely you are already familiar with it. In case you are not, its not a big deal because YAML is a pretty easy language to learn. We’ll explain each line of the YAML config of a pod and you will understand the YAML format as a "by-product" of your POD learning process.

**POD configuration file in YAML format.**

$ cat pod-2containers.yaml  
apiVersion: v1   
kind: Pod   
metadata:   
 name: pod-1   
 labels:   
 name: pod-1   
spec:   
 containers:   
 - name: apach   
 image: pingdocker/apache-frontend   
 ports:   
 - containerPort: 80   
 - name: db   
 image: pingdocker/redis-db   
 ports:   
 - containerPort: 6379

YAML uses 3 basic data types:

* scalars (strings/numbers): atom data item. strings like pod-1, port number 80.
* mappings (hashes/dictionaries): key-value pairs, can be nested. apiVersion:  
  v1 is a mapping. key apiVersion has a value of v1.
* sequences (arrays/lists): collection of ordered values, without a "key". list items are indicated by a - sign. value of key contrains is a list including 2 containers.

in this example you are also seeing "nested" YAML data structure:

* "mapping of a mapping": spec is the key of a map, where you define a pod’s specification. in this example we only define behavior of the containers to be launched in the pod. the value is another map with the key being containers.
* "mapping of a list". values of the key "containers" is a list of two items: frontend and redis container, each of which again, are a mapping describing the individual container with a few attributes like name, image and ports to be exposed.

**Note**

* case sensitive
* elements in same level share same left indentation, the amount of indentation does not matter
* tab characters are not allowed to be used as indentation
* blank lines does not matter
* comment a line with "#"
* use quote ' to escape special meaning of any character

before we dive into more details of the yaml file, let’s finish the pod creation:

$ kubectl create -f pod-2containers.yaml  
pod/pod-1 created  
  
$ kubectl get pod -o wide  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
pod-1 0/2 ContainerCreating 0 18s 10.47.255.233 cent222 <none>  
  
$ kubectl get pod -o wide  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
pod-1 2/2 Running 0 18s 10.47.255.233 cent222 <none>

we created our first kubernetes "object" - a pod named pod-1. but where are the containers? the above output tells the clues. it reads:

a pod pod-1 (NAME), containing 2 containers(READY /2), has been launched in kubernetes worker node cent222 with an IP address 10.47.255.233 assigned. both containers in the pod is up (READY 2/) and has been in running STATUS for 27s without any RESTARTS.

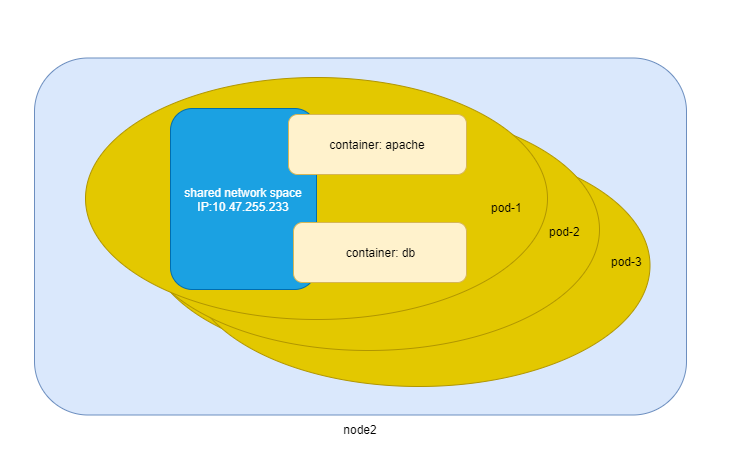
here is a brief line-by-line comments about what the yaml config says:

* line 1,2,3,7: the 4 yaml mappings are the main components of a pod definition.
  + apiVersion: there are different versions, for example, v2. here specifically it is version 1.
  + kind: remember there are different type of kubernetes object, here we want kubernetes to create a 'pod' object. later you will see kind being ReplicationController or Service in example of other objects.
  + metadata: to identify the created objects. besides the name of the object to be created, another important meta data is "labels". you will read more about it in chapter3.
  + spec: gives the specification about the pod behavior.
* line 9-16: the pod specification here is just about the 2 containers. the system downloads the images, launches each container with a name and expose the specified ports respectively.

to get more details of what is running inside of the pod:

$ kubectl describe pod pod-1 | grep -iC1 container  
IP: 10.47.255.233  
Containers:  
 apache:  
 Container ID: docker://489e67aec14890092378a1e47b27d40b26c1e051b93958db037091212f7db76e  
 Image: pingdocker/apache-frontend  
--  
 db:  
 Container ID: docker://b5492678744548f7c394e89e26af185f576273d23ab7ee91161340fad417ca60  
 Image: pingdocker/redis-db  
--  
 Ready True  
 ContainersReady True  
 PodScheduled True  
--  
 Normal Pulled <invalid> kubelet, cent222 Successfully pulled image "pingdocker/apache-frontend"  
 Normal Created <invalid> kubelet, cent222 Created container  
 Normal Started <invalid> kubelet, cent222 Started container  
 Normal Pulling <invalid> kubelet, cent222 pulling image "pingdocker/redis-db"  
 Normal Pulled <invalid> kubelet, cent222 Successfully pulled image "pingdocker/redis-db"  
 Normal Created <invalid> kubelet, cent222 Created container  
 Normal Started <invalid> kubelet, cent222 Started container

not surprisingly, our pod pod-1 is composed of 2 containers declared in the YAML file, apache and db respectively, with an IP address assigned by kubernetes cluster and shared between all containers as shown in following figure:



node, pod and containers

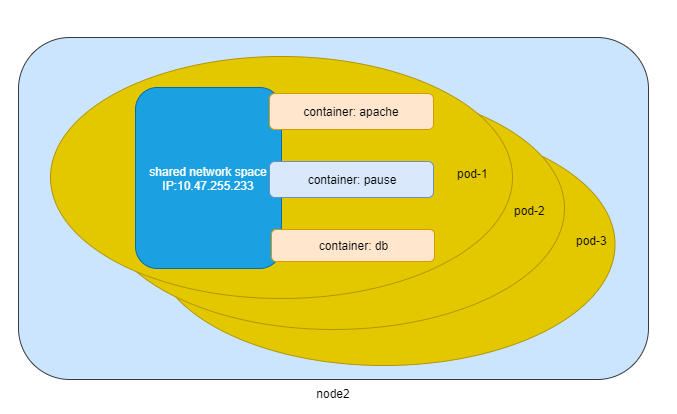
## pause container

if you login to node cent222, you will see the docker containers running inside of the pod:

$ docker ps | grep -E "ID|pod-1"  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
e2d76084a1d5 pingdocker/redis-db "redis-server /etc/r…" About a minute ago Up About a minute k8s\_db\_pod-1\_default\_c8675d37-c22d-11e9-add6-0050569e6cfc\_0  
694f5eb781e7 pingdocker/apache-frontend "apache2-foreground" About a minute ago Up About a minute k8s\_apache\_pod-1\_default\_c8675d37-c22d-11e9-add6-0050569e6cfc\_0  
7abae4273cfd k8s.gcr.io/pause:3.1 "/pause" About a minute ago Up About a minute k8s\_POD\_pod-1\_default\_c8675d37-c22d-11e9-add6-0050569e6cfc\_0

the third container with image name k8s.gcr.io/pause is a special container that was created by the kubernetes system for each pod. The pause container is created to manage the network resources for the pod which would be shared by all the containers of that pod.

below figure shows a pod including a few user containers and a pause container.



pod, user containers and the special pause container

## intra-pod communication

in kubernetes master, to login to a container:

#login to pod-1's container apache  
root@test1:~# kubectl exec -it pod-1 -c apache bash  
root@pod-1:/var/www/html#

#login to pod-1's container db  
root@test1:~# kubectl exec -it pod-1 -c db bash  
[ root@pod-1:/data ]$

**Note**

if you ever played with docker you will immediately realized that this is pretty neat. remember the containers were launched at one of the "node", with docker you will have to first login to the correct remote node, and then use a similiar docker exec command to login to each container. kubernetes hides these details and allow you to do everything from one node - the master.

now check processes running in the container:

**apache container.**

root@pod-1:/var/www/html# ps aux  
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
root 1 0.5 0.2 166260 19176 ? Ss 17:08 0:00 apache2 -DFOREGROUND  
www-data 13 0.0 0.0 166284 7136 ? S 17:08 0:00 apache2 -DFOREGROUND  
www-data 14 0.0 0.0 166284 7136 ? S 17:08 0:00 apache2 -DFOREGROUND  
www-data 15 0.0 0.0 166284 7136 ? S 17:08 0:00 apache2 -DFOREGROUND  
www-data 16 0.0 0.0 166284 7136 ? S 17:08 0:00 apache2 -DFOREGROUND  
www-data 17 0.0 0.0 166284 7136 ? S 17:08 0:00 apache2 -DFOREGROUND  
root 18 0.0 0.0 20244 3072 pts/0 Ss 17:08 0:00 bash  
root 25 0.0 0.0 17492 1964 pts/0 R+ 17:08 0:00 ps aux  
  
root@pod-1:/var/www/html# ss -at  
State Recv-Q Send-Q Local Address:Port Peer Address:Port  
LISTEN 0 128 \*:6379 \*:\*  
LISTEN 0 128 \*:http \*:\*  
LISTEN 0 128 :::6379 :::\*  
  
root@pod-1:/var/www/html# ip a  
1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000  
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
 inet 127.0.0.1/8 scope host lo  
 valid\_lft forever preferred\_lft forever  
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default  
 link/ether 02:c8:7a:0b:22:c2 brd ff:ff:ff:ff:ff:ff  
 inet 10.47.255.233/12 scope global eth0  
 valid\_lft forever preferred\_lft forever

**db container.**

[ root@pod-1:/data ]$ ps aux  
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
root 1 0.0 0.0 35200 3776 ? Ssl 17:08 0:00 redis-server \*:6379  
root 13 0.0 0.0 19352 4484 pts/0 Ss 17:09 0:00 bash  
root 75 0.0 0.0 15576 2168 pts/0 R+ 17:10 0:00 ps aux  
  
[ root@pod-1:/data ]$ ss -at  
State Recv-Q Send-Q Local Address:Port Peer Address:Port  
LISTEN 0 128 \*:6379 \*:\*  
LISTEN 0 128 \*:http \*:\*  
LISTEN 0 128 :::6379 :::\*  
  
[ root@pod-1:/data ]$ ip a  
1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000  
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
 inet 127.0.0.1/8 scope host lo  
 valid\_lft forever preferred\_lft forever  
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default  
 link/ether 02:c8:7a:0b:22:c2 brd ff:ff:ff:ff:ff:ff  
 inet 10.47.255.233/12 scope global eth0  
 valid\_lft forever preferred\_lft forever

the ps command output shows that each container is running its own process. however, th ss and ip command output indicate that both container share the same exact network environment so both see the port exposed by each other. Therefore, communication between containers in a pod can happen simply by using localhost. we can test this out by starting a tcp connection using curl command:

from container apache, start a TCP connection toward redis port that container db is listening, using localhost IP address:

root@pod-1:/var/www/html# curl localhost:6379  
^Z  
[1]+ Stopped curl localhost:6379  
  
root@pod-1:/var/www/html# bg  
[1]+ curl localhost:6379 &

now monitor the TCP connection state: the connection is established successfully.

root@pod-1:/var/www/html# ss -at  
State Recv-Q Send-Q Local Address:Port Peer Address:Port  
LISTEN 0 128 \*:6379 \*:\*  
LISTEN 0 128 \*:http \*:\*  
ESTAB 0 0 127.0.0.1:46378 127.0.0.1:6379 #<---  
ESTAB 0 0 127.0.0.1:6379 127.0.0.1:46378 #<---  
LISTEN 0 128 :::6379 :::\*

## Kubectl tool

so far you’ve seen we created the object by kubectl command. this command, just like the docker command in docker world, is the interface in kubernetes world to talk to the cluster, or more precisely, the kubernetes master, via kubernetes API. it is a very versatile tool that provides many options to fulfill all kinds of tasks you would need to deal with kubernetes.

as a quick example, assuming you have enabled the auto-completion feature for kubectl, you can list all your current environment supported options by logging into the master and typing kubectl, followed by two tab keystrokes.

root@test1:~# kubectl<TAB><TAB>  
alpha attach completion create exec  
logs proxy set wait annotate auth  
config delete explain options replace  
taint api-resources autoscale convert describe  
patch rollout top api-versions certificate  
drain get plugin run uncordon apply  
cluster-info cp edit label port-forward  
scale version expose cordon

**Note**

to setup auto-completion for kubectl command, follow the instruction from help of completion option: kubectl completion -h

you will see and learn the some of these options in the rest part of this book.