Getting Started with Kubernetes

CONTENTS

UPDATED BY ALAN HOHN, LOCKHEED MARTIN FELLOW

ORIGINAL BY ARUN GUPTA, PRINCIPAL OPEN SOURCE TECHNOLOGIST AT AMAZON WEB SERVICES

WHAT IS KUBERNETES?

Kubernetes is an open source container orchestration system. It manages containerized applications across multiple hosts for deploying, monitoring, and scaling containers. Originally created by Google, in March of 2016 it was donated to the Cloud Native Computing Foundation (CNCF).

Kubernetes, or "k8s" or "kube" for short, allows the user to declare the desired state of an application using concepts such as "deployments" and "services." For example, the user may specify that they want a deployment with three instances of a Tomcat web application running. Kubernetes starts and then continuously monitors containers, providing auto-restart, re-scheduling, and replication to ensure the application stays in the desired state.

Kubernetes is available as a standalone installation or in a variety of distributions, such as Red Hat OpenShift, Pivotal Container Service, CoreOS Tectonic, and Canonical Kubernetes.

KEY KUBERNETES CONCEPTS

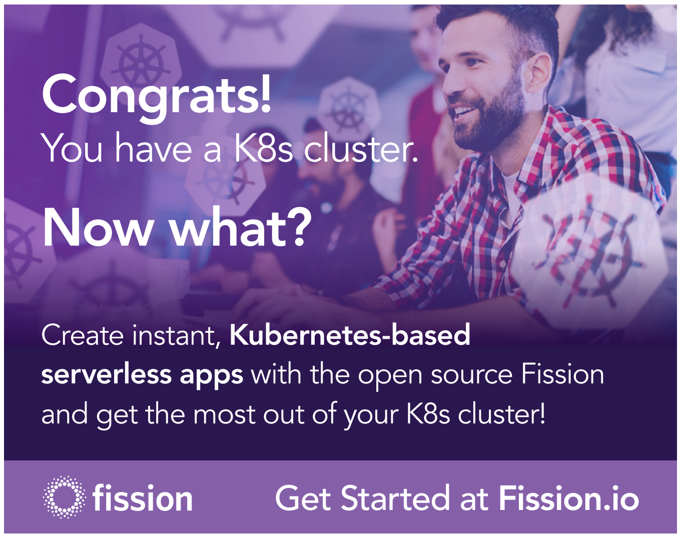
Kubernetes resources can be created directly on the command line but are usually specified using Yet Another Markup Language (YAML). The available resources and the fields for each resource may change with new Kubernetes versions, so it's important to double-check the API reference for your version to know what's available. It's also important to use the correct "apiVersion" that matches your version of Kubernetes. This Refcard uses the API from Kubernetes 1.12, released 27 September 2018.

POD

A Pod is a group of one or more containers. Kubernetes will schedule all containers for a pod into the same host, with the same network namespace, so they all have the same IP address and can access each other using `localhost`. Here is an example pod definition:

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: /srv/www
      name: www-data
      readOnly: true
  - name: git-monitor
    image: kubernetes/git-monitor
    env:
    - name: GIT_REPO
      value: http://github.com/some/repo.git
    volumeMounts:
    - mountPath: /data
       name: www-data
  volumes:
  - name: www-data
    emptyDir: {}
```

Not only can containers in a pod share the same network interfaces, but they can also share volumes, as shown in the


Congrats!
You have a K8s cluster.
Now what?
Create instant, Kubernetes-based serverless apps with the open source Fission and get the most out of your K8s cluster!
fission
Get Started at Fission.io

fission

fission.io

Create Instant, Kubernetes-based Serverless Apps with Fission

Fission is an open-source, Kubernetes-native, serverless framework.

Code serverless functions in any language and have them run wherever you have a Kubernetes cluster: in the public cloud, in your own datacenter, or even on your laptop.

Fission automatically manages the infrastructure for you. There's no need for in-depth knowledge of Kubernetes at scale, no containers to build or registries to manage.

Fission lets you get started quickly and get the most out of your k8s cluster, benefiting from the speed and cost savings of serverless – on any environment.

Get Started Today at Fission.io

example. This example uses the "git-monitor" container to keep a Git repository updated in `/data` so the "nginx" container can run a web server to serve the repository files.

When a pod is created, Kubernetes will monitor it and automatically restart it if a process terminates. In addition, Kubernetes can be configured to attempt to connect to a container over the network to determine if the pod is ready (`readinessProbe`) and still alive (`livenessProbe`).

DEPLOYMENT

A deployment provides pod scaling and rolling updates. Kubernetes will make sure that the specified number of pods is running, and on a rolling update will replace pod instances one at a time, allowing for application updates with zero downtime.

Deployments graduated from beta in Kubernetes 1.11, and replace the older concept of Replica Sets. A deployment creates a replica set, but it is not necessary to interact with the replica set directly.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

Deployments include a "template" that specifies what the created pods should look like, so there is no need to separately define a pod. Kubernetes will automatically create the required number of pods from the deployment template.

Note that a deployment will identify matching pods using the `matchLabels` field. This field must always have the same data as the `metadata.labels` field inside the template. The deployment will take ownership of any other running pods that match the `matchLabels` selector, even if they were created separately, so keep these names unique.

SERVICE

A service provides load balancing to a deployment. In a deployment, each pod is assigned a unique IP address, and when a pod is replaced, the new pod typically receives a new IP address.

By declaring a service, we can provide a single point of entry for all the pods in a deployment. This single point of entry (hostname and IP address) remains valid as pods come and go.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
```

Services and deployments can be created in any order. The service actively monitors Kubernetes for pods matching the `selector` field. For example, in this case the service will match pods with `metadata.labels` content of `app: nginx`, like the one shown in the deployment example above.

Services rely on Kubernetes to provide a unique IP address and route traffic to them, so the way services are configured can be different depending on how your Kubernetes installation is configured.

PERSISTENT VOLUME CLAIM

Kubernetes has multiple types of storage resources. The Pod example above shows the simplest, an empty directory mounted into multiple containers in the same pod. For truly persistent storage, the most flexible approach is to use a Persistent Volume Claim.

A Persistent Volume Claim requests Kubernetes to dynamically allocate storage from a Storage Class. The Storage Class is typically created by the administrator of the Kubernetes cluster and must already exist. Once the Persistent Volume Claim is created, it can be attached to a Pod. Kubernetes will keep the storage while the Persistent Volume Claim exists, even if the attached Pod is deleted.

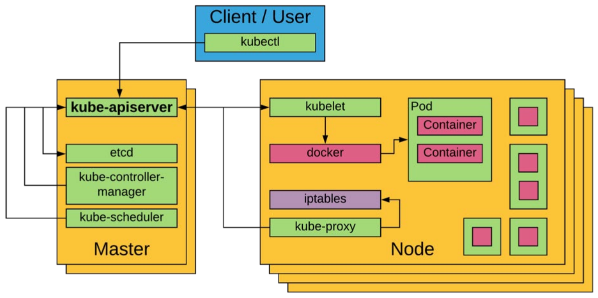
```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: web-static-files
spec:
  resources:
    requests:
      storage: 8Gi
  storageClassName: file-store
---
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: web-files
  volumes:
  - name: web-files
    persistentVolumeClaim:
      claimName: web-static-files
```

The above example declares both the Persistent Volume Claim and a Pod that uses it. This example takes advantage of the ability to place multiple YAML documents in the same file using `---` as a separator. Of course, in a real example, it would be better to separate the Persistent Volume Claim so it can easily be retained even if the Pod is deleted.

For more information on the available providers for Kubernetes Storage Classes, and for multiple examples on configuring persistent storage, see the DZone Refcard *Persistent Container Storage*.

KUBERNETES ARCHITECTURE

Kubernetes uses a client-server architecture, as seen here:



A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources that are used to run applications. The machines that manage the cluster are called Masters, and the machines that run the containers are called Nodes.

MASTER

The Master runs services that manage the cluster. The most important is kube-apiserver, which is the primary service that clients and nodes use to query and modify the resources running in the cluster. The API server is assisted by: etcd, a distributed key-value store used to record cluster state; kube-controller-manager, a monitoring program that decides what changes to make when resources are added, changed, or removed; and kube-scheduler, a program that decides where to run pods based on the available nodes and their configuration.

In a highly-available Kubernetes installation, there will be multiple masters, with one acting as the primary and the others as replicas.

NODE

A Node is a physical or virtual machine with the necessary services to run containers. A Kubernetes cluster should have as many nodes as necessary for all the required pods. Each node has two Kubernetes services: kubelet, which receives commands to run containers and uses the container engine (e.g. Docker) to run them; and kube-proxy, which manages networking rules so connections to service IP addresses are correctly routed to pods.

As shown in the picture, each node can run multiple pods, and each pod can include one or more containers. The pod is purely a Kubernetes concept; the kubelet configures the container engine to place multiple containers in the same network namespace so those containers share an IP address.

GETTING STARTED WITH KUBERNETES

SETTING UP KUBERNETES

There are a variety of ways to set up, configure, and run Kubernetes. It can be run in the cloud using providers such as Amazon Elastic Container Service for Kubernetes, Google Kubernetes Engine, Azure Kubernetes Service, Packet, Pivotal Container Service, and others. It can be also run on-premise by building a cluster from scratch on physical hardware or via virtual machines. The various options are described in the Kubernetes setup documentation, where you can find out which solution is best for you and get step-by-step instructions. The most popular option for building a multi-host Kubernetes cluster from scratch is kubeadm, while the recommended way to get started and run a single-node cluster for development and testing is to use Minikube.

However you set up your cluster, you will interact with it using the standard Kubernetes command-line client program kubectl.

MINIKUBE

Minikube uses virtualization software like VirtualBox, VMware, or KVM to run the cluster. Once Minikube is installed, you can use the `minikube` command-line to start a cluster by running the following command:

```
minikube start
```

To stop the cluster, you can run:

```
minikube stop
```

To determine the IP address of the cluster is using:

```
minikube ip
```

If you are having problems, you can view the logs or ssh into the host to help debug the issue by using:

```
minikube logs
minikube ssh
```

You can also open a dashboard view in the browser to see and change what is going on in the cluster.

```
minikube dashboard
```

KUBECTL

kubectl is a command-line utility that controls the Kubernetes cluster. Commands use this format:

`kubectl [command] [type] [name] [flags]`

- `[command]` specifies the operation that needs to be performed on the resource. For example, create, get, describe, delete, or scale.
- `[type]` specifies the Kubernetes resource type. For example, pod (po), service (svc), deployment (deploy), or persistentvolumeclaim (pvc). Resource types are case-insensitive, and you can specify the singular, plural, or abbreviated forms.
- `[name]` Specifies the name of the resource, if applicable. Names are case-sensitive. If the name is omitted, details for all resources will be displayed (for example, kubectl get pods).
- `[flags]` Options for the command.

Some examples of kubectl commands and their purpose:

COMMAND	**PURPOSE**
`kubectl create -f nginx.yaml`	Create the resources specified in the YAML file. If any specified resources exist, an error is returned.
`kubectl delete -f nginx.yml`	Delete the resources specified in the YAML file. If any resources do not exist, they are ignored.
`kubectl get pods`	List all pods in the "default" namespace. See below for more information on namespaces.
`kubectl describe pod nginx`	Show metadata for the "nginx" pod. The name must match exactly.
`kubectl --help`	Show the complete list of available commands.

RUN YOUR FIRST CONTAINER

Most of the time when using kubectl, we create YAML resource files, so we can configure how we want our application to run. However, we can create a simple Deployment using kubectl without using a YAML file:

```
kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
```

This command will start a Deployment, which contains a Repilca Set, which contains a Pod, which contains a Docker container running an NGINX web server. We can use kubectl to get the status of the deployment:

```
kubectl get deploy
NAME    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
nginx   1         1         1            1
1m
```

The status of the Replica Set can be seen by using:

```
kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
nginx-65899c769f   1         1         1       1m
```

The status of the Pod can be seen by using:

```
kubectl get po
NAME                     READY   STATUS    RESTARTS
AGE
nginx-65899c769f-kp5c7   1/1     Running   0
1m
```

Of course, most of the time we will use a YAML configuration file:

`kubectl create -f nginx-deployment.yaml`

The file nginx-deployment.yaml contains the Deployment definition shown above.

SCALE APPLICATIONS

Deployments can be scaled up and down:

```
kubectl scale --replicas=3 deploy/nginx
deployment.extensions/nginx scaled
```

The Kubernetes controller will then work with the scheduler to create or delete pods as needed to achieve the requested number. This is reflected in the deployment:

```
kubectl get deploy
NAME    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
nginx   3         3         3            3
3m
```

You can verify there are three pods by running:

```
kubectl get po
NAME                     READY   STATUS    RESTARTS
AGE
nginx-65899c769f-c46xx   1/1     Running   0
38s
nginx-65899c769f-j484j   1/1     Running   0
38s
nginx-65899c769f-kp5c7   1/1     Running   0
3m
```

Note that the original Pod continued to run while two more were added. Of course, we would also want to create a Service to assist in load balancing across these instances; see below for a more complete example.

DELETE APPLICATIONS

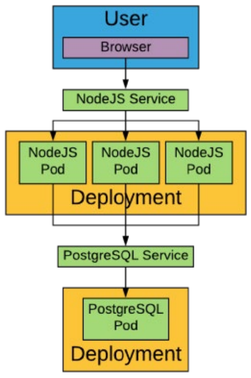
Once you are done using the application, you can destroy it with the `delete` command.

```
kubectl delete deployment nginx
deployment.extensions "nginx" deleted
```

Because Kubernetes monitors pods to achieve the desired number of replicas, we must delete the Deployment to remove the application. Simply stopping the container or deleting the pod will just cause Kubernetes to create another pod.

EXAMPLE APPLICATION

Let's put multiple Kubernetes features together to deploy a Node.js application together with a PostgreSQL database server. Here is the planned architecture (see right).



We'll work from the bottom of the diagram. First, we'll define the PostgreSQL deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql
  labels:
    app: postgresql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgresql
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      containers:
      - name: postgresql
        image: postgres:10.4
        env:
        - name: PGDATA
          value: "/data/pgdata"
        volumeMounts:
        - mountPath: /data
          name: postgresql-data
      volumes:
        - name: postgresql-data
          persistentVolumeClaim:
           claimName: postgresql-data
```

Note that we use a `PersistentVolumeClaim` so we get persistent data; we'll assume this has been created already since we want it to stay around for a long time even as we update the application.

Even though there will only be one database instance, we will create a Service so the IP address will stay the same even if the PostgreSQL pod is replaced.

```
kind: Service
apiVersion: v1
metadata:
  name: postgres-service
spec:
  selector:
    app: postgresql
  ports:
  - protocol: TCP
    port: 5432
```

Next, we create the deployment for the Node.js application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs
  labels:
    app: nodejs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodejs
  template:
    metadata:
      labels:
        app: nodejs
    spec:
      containers:
      - name: nodejs
        image: nodejs:10-alpine
        command: ["npm"]
        args: ["start"]
        env:
        - name: NODE_ENV
          value: production
        workingDir: /app
        volumeMounts:
        - mountPath: /app
          name: node-app
          readOnly: true
      - name: git-monitor
        image: kubernetes/git-monitor
        env:
        - name: GIT_REPO
          value: http://github.com/some/repo.git
        volumeMounts:
        - mountPath: /data
          name: node-app
      volumes:
      - name: www-data
        emptyDir: {}
```

This example uses a “sidecar” container, kubernetes/git-monitor, to keep our application up to date based on a Git repository. The sidecar populates a volume which is shared with the Node.js container.

Finally, we create the service that provides the user entrypoint for our application:

```
kind: Service
apiVersion: v1
metadata:
  name: nodejs-service
spec:
  selector:
    app: nodejs
  ports:
  - protocol: TCP
    port: 3000
  type: LoadBalancer
```

Providing services with external IPs so they can be visible from outside the cluster is a complex topic because it depends on your cluster’s environment (e.g. cloud, virtual machine, or bare metal). This example uses a LoadBalancer service, which requires some external load balancer such as an Amazon Elastic Load Balancer to be available and configured in the cluster.

NAMESPACE, RESOURCE QUOTAS, AND LIMITS

Kubernetes uses namespaces to avoid name collisions, to control access, and to set quotas. When we created resources above, these went into the namespace `default`. Other resources that are part of the cluster infrastructure are in the namespace `kube-system`.

To see pods in `kube-system`, we can run:

```
$ kubectl get po -n kube-system
NAME                        READY    STATUS
RESTARTS    AGE
…
kube-apiserver-minikube     1/1      Running    0
17m
…
```

RESOURCE ISOLATION

A new namespace can be created from a YAML resource definition:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

Once we’ve created the namespace, we can create resources in it using the `--namespace` (`-n`) flag, or by specifying the namespace in the resource’s metadata:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  namespace: development
…
```

By using separate namespaces, we can have many pods called `webserver` and not have to worry about name collisions.

ACCESS CONTROL

Kubernetes supports Role Based Access Control (RBAC).

Here’s an example that limits developers to read-only access for pods in production. First, we create a `ClusterRole`, a common set of permissions we can apply to any namespace:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pod-read-only
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Next, we use a `RoleBinding` to apply this `ClusterRole` to a specific group in a specific namespace:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-only
  namespace: production
subjects:
- kind: Group
  name: developers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-read-only
  apiGroup: rbac.authorization.k8s.io
```

Alternatively, we can use a `ClusterRoleBinding` to apply a role to a user or group in all namespaces.

RESOURCE QUOTAS

By default, pods have unlimited resources. We can apply a quota to a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
  namespace: sandbox
spec:
  hard:
    cpu: "5"
    memory: 10Gi
```

Kubernetes will now reject unlimited pods in this namespace. Instead, we need to apply a limit:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  namespace: sandbox
spec:
  containers:
  - image: nginx
    name: nginx
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Note that we can request fractions of a CPU and use varying units for memory.

Written by **Alan Hohn**, Lockheed Martin Fellow

Alan Hohn is a Lockheed Martin Fellow who has worked as a software architect, lead, and manager. After many years writing and teaching Java, he now mostly creates services in Go and Python. He is an advocate, trainer, and coach for Agile and DevOps, and is the author of recent video courses on Ansible.