Cockroach Labs

DZONE REFCARDZ

Getting Started with Kubernetes

DZone
A DEVADA MEDIA PROPERTY

BROUGHT TO YOU IN PARTNERSHIP WITH Cockroach Labs

Getting Started With Kubernetes

CONTENTS

ALAN HOHN
LOCKHEED MARTIN FELLOW

INTRODUCTION

Containers are a great way to package, deploy, and manage applications. However, to build a reliable, scalable containerized application, you need a place to host your containers, update them, and provide them with networking and storage. Kubernetes has become the most popular container orchestration system. It simplifies deploying, monitoring, and scaling application components, making it easy to develop flexible, reliable applications.

WHAT IS KUBERNETES?

Kubernetes (also known by its abbreviation "k8s") is an open source container orchestration system. It manages a "cluster" of multiple hosts that are used to deploy, monitor, and scale containers. Originally created by Google in March of 2016, it was donated to the Cloud Native Computing Foundation (CNCF).

Kubernetes is declarative. This means that you can create and update "resources" that describe which containers to run, how to configure them, and how to route network traffic to them.

Kubernetes continuously updates and monitors the cluster to ensure it matches the desired state, including auto-restart, re-scheduling, and replication to ensure applications start and remain running.

Kubernetes is available as a standalone installation via cloud providers including Google, Amazon, and Microsoft, or in a variety of distributions including Red Hat OpenShift, Rancher Kubernetes, and VMWare Tanzu.

KEY KUBERNETES CONCEPTS

Because Kubernetes is declarative, getting started in Kubernetes mostly means understanding what resources we can create and how they are used to deploy and configure containers in the cluster.

To define resources, we use YAML format. 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 1.18, released 23 April 2020. Kubernetes 1.19 was released on 25 September but there are still many clusters using 1.18.

CockroachDB

K8s + CockroachDB = Effortless App Deployment

Run your application on the cloud-native database uniquely suited to Kubernetes.

Scale elastically with distributed SQL

Say goodbye to sharding and time-consuming manual scaling.

Survive anything with bullet-proof resilience

Rest easy knowing your application data is always on and always available.

Build fast with PostgreSQL compatibility

CockroachDB works with your current applications and fits how you work today.

Get started for free today

cockroachlabs.com/k8s

Trusted by innovators

COMCAST SPACEX BOSE LUSH rubrik wework

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. The containers in a pod can also share storage volumes.

We don't typically create pod resources directly. Instead, we have Kubernetes manage them through a deployment, so we get fault tolerance, scalability, and rolling updates.

DEPLOYMENT

A deployment manages one or more identical pod instances. Kubernetes will make sure that the specified number of pods is running, and on a rolling update, it will replace pod instances one at a time, allowing for application updates with zero downtime. Here is an example deployment:

```
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:1.19.3-alpine
        volumeMounts:
        - mountPath: /usr/share/nginx
          name: www-data
          readOnly: true
        ports:
        - containerPort: 80
      initContainers:
      - name: git-clone
        image: alpine/git
        args: ["clone", "https://github.com/AlanHohn/
hello-world-static.git", "/www/html"]
        volumeMounts:
        - mountPath: /www
          name: www-data
      volumes:
      - name: www-data
        emptyDir: {}
```

The "template" section of the deployment specifies exactly what the created pods should look like. Kubernetes will automatically create the required number of pods from the template. When a pod is created, Kubernetes will monitor it and automatically restart it if the container 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`).

The example above defines one container in the pod and also defines an "init container". The init container runs before the main pod container starts. In this case, it uses Git to populate the files that the NGINX web server should serve. We use an "init container" because Git runs once and then exits, so it cannot be a regular container or Kubernetes will think the pod has failed.

Deployments identify the pods they should manage using the `matchLabels` `selector` field. This field must always have the same data as the `metadata.labels` field inside the template. The deployment will take ownership of any 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 group of pods. Every time Kubernetes creates a pod, it is assigned a unique IP address. When a pod is replaced, the new pod typically receives a new IP. 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, and the Kubernetes cluster even provides a DNS server so that we can use service names as regular hostnames.

```
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. 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, so the way services are configured can be different depending on how your Kubernetes installation is configured. The most common type of service is ClusterIP. This means the service has an IP address accessible only from within the Kubernetes cluster; exposing the service outside the cluster requires another resource like an ingress.

It's important to know that when network traffic is sent to a service address and port, Kubernetes uses port-forwarding to route traffic to a specific pod. Only the declared protocols and ports are forwarded, so other kinds of traffic (like `ICMP ping`) will not work to a service address, even within the cluster.

INGRESS

An ingress is one approach for routing traffic from outside the cluster. (An alternate and more advanced approach is a service mesh such as Istio.) To use an ingress, a cluster administrator first deploys an "ingress controller". This is a regular Kubernetes deployment, but it registers with the Kubernetes cluster to be notified when an Ingress resource is created, updated, or deleted. It then configures itself to route traffic based on the ingress resources.

The advantage of this approach is that only the ingress controller needs an IP address that is reachable from outside the cluster, simplifying configuration and potentially saving money. Here is an example ingress:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: nginx-service
          servicePort: 80
```

This example routes all traffic in the cluster to our NGINX service, so it is only useful for getting started. In a production cluster, you can use DNS to route many hostnames to the same ingress IP address, and then use host rules to route traffic to the correct application.

PERSISTENT VOLUME CLAIM

Kubernetes has multiple types of storage resources. The deployment 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:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

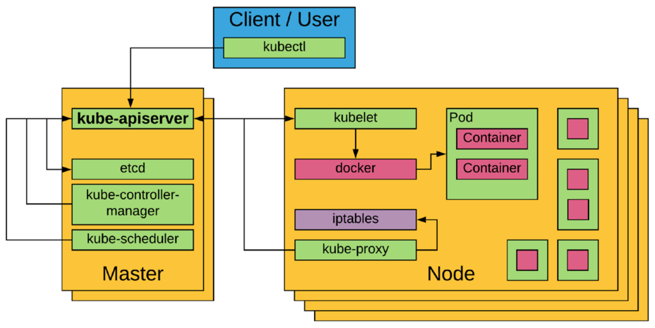
The above YAML declares a persistent volume claim. We would then use this persistent storage in a deployment similar to the example above. Only the volumes section would change:

```
      volumes:
       - name: www-date
         persistentVolumeClaim:
           claimName: web-static-files
```

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

CREATING A DEVELOPMENT CLUSTER

Running a production Kubernetes cluster is a complex job. Unless you're deeply familiar with Kubernetes configuration, it's best to use one of the many cloud options or distributions above. Kubernetes is capable of running anything that can be packaged in a container, so insecure public clusters are quickly exploited for Bitcoin mining or other nefarious purposes.

For a development environment, there are many great options, including Microk8s or Rancher k3s. This guide shows k3s as the setup, which is the same on any system with a working Docker installation.

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

K3S

K3s is a lightweight Kubernetes distribution in a single binary. While it can be run directly, we will use k3d to run it inside Docker. First, install Docker, `kubectl`, and `k3d`. Then run:

```
k3d cluster create -p "8081:80@loadbalancer"
```

This will create the cluster and configure `kubectl` to communicate with it. The extra "`-p`" option tells `k3d` to forward port 80 to the internal load balancer from port 8081 on our host machine. This allows us to access the Traefik ingress controller that is part of k3s.

To verify that the cluster is running as expected, run `kubectl version` as shown here:

```
kubectl version
Client Version: version.Info{Major:"1",
Minor:"19", GitVersion:"v1.19.2",
GitCommit:"f5743093fd1c663cb0cbc89748f730662345d44d",
GitTreeState:"clean", BuildDate:"2020-09-16T13:41:02Z",
GoVersion:"go1.15", Compiler:"gc", Platform:"linux/
amd64"}
Server Version: version.Info{Major:"1",
Minor:"18", GitVersion:"v1.18.9+k3s1",
GitCommit:"630bebf94b9dce6b8cd3d402644ed023b3af8f90",
GitTreeState:"clean", BuildDate:"2020-09-17T19:05:07Z",
GoVersion:"go1.13.15", Compiler:"gc", Platform:"linux/
amd64"}
```

This will report the exact version in use, which is valuable in making sure you are using the correct API documentation.

Finally, to stop the cluster, you can run:

```
k3d cluster delete
```

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, including `pod(po)`, `service(svc)`, `deployment(deploy)`, ingress, or `PersistentVolumeClaim` (`pvc`). Resource types are case-insensitive and found in 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 resources are displayed (for example, `kubectl get pods`).
- `[flags]` Options for the command.

Some examples of `kubectl` commands and their purpose:

COMMAND	PURPOSE
`kubectl apply -f nginx.yaml`	Create or update the resources specified in the YAML file. Kubernetes records the state of the resource when it was last applied so that it can figure out what changes were made.
`kubectl delete -f nginx.yml`	Delete the resources specified in the YAML file. If any resources do not exist, they are ignored.

Continued on next page

`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.

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 to manage pods containing a Docker container. The Docker container will run 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
```

As described above, the deployment automatically creates a pod. To see the pod, run:

```
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; for example:

```
kubectl apply -f deploy.yaml
```

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
```

You can also scale the deployment by editing the replicas field in the YAML and re-running "`kubectl apply -f <file>`".

DELETE APPLICATIONS

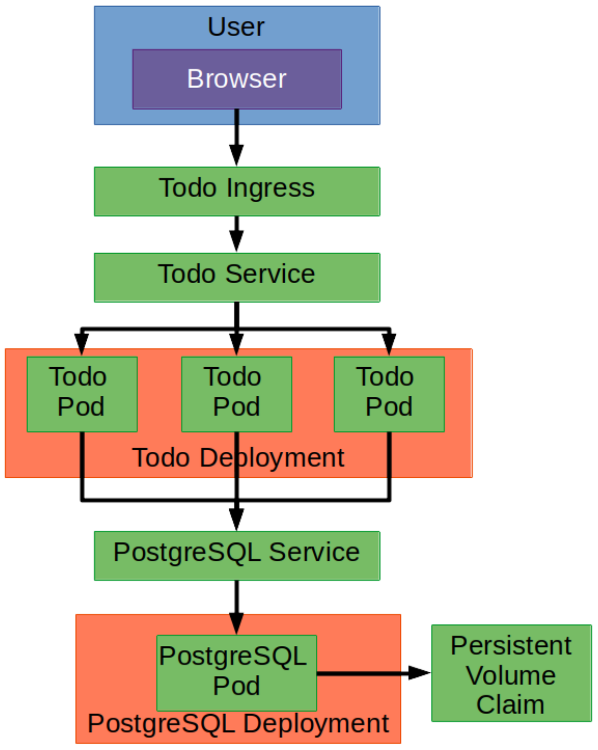
Once you are done using the application, you can destroy it with the `delete` command.

```
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 deleting the pod will just cause Kubernetes to create another pod.

EXAMPLE APPLICATION

Let's put multiple Kubernetes features together to deploy an example Todo application, written in Node.js, together with a PostgreSQL database server. Here is the planned architecture:



We'll work up from the bottom of the diagram. First, before we deploy our PostgreSQL database, we'll create persistent storage for it:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: postgresql-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

Next, 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:13.0
        env:
        - name: POSTGRES_HOST_AUTH_METHOD
          value: "trust"
        - name: PGDATA
          value: "/data/pgdata"
        - name: POSTGRES_DB
          value: "todo"
        volumeMounts:
        - mountPath: /data
          name: postgresql-data
      volumes:
        - name: postgresql-data
          persistentVolumeClaim:
           claimName: postgresql-data
```

For a production system, you should choose a secure database password and configure PostgreSQL to use it.

A Kubernetes Secret is a good way to insert protected environment variables into your containers.

Even though there will only be one database instance, we will create a `Service` so that 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:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: todo
  labels:
    app: todo-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
      - name: todo
        image: alanhohn/todo
        env:
        - name: NODE_ENV
          value: "production"
        - name: DATABASE_URL
          value: "postgres://postgres:postgres@
postgresql/todo"
        ports:
        - containerPort: 5000
```

Note that we configure the application to use the hostname "postgresql" for the database. This matches the name of the service we created above. We can use the plain host name because this deployment is in the same namespace as the database service (see the section on namespaces below).

Next, we create the service that provides the user entry point for our application:

```
kind: Service
apiVersion: v1
metadata:
  name: todo
spec:
  selector:
    app: todo
  ports:
  - protocol: TCP
    port: 5000
```

Finally, we create an ingress to expose this service. Like the example above, this ingress routes all traffic to a single service, so it is only useful for a development cluster. Be sure to delete any other ingress in the cluster before creating this one so that traffic will be routed correctly.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: todo
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: todo
          servicePort: 5000
```

Now that you've deployed all of the components, you can visit http://localhost:8081 in your browser and you should see the todo application. If you're having issues, use `kubectl get` to inspect all the resources you created and the pods to see if one is failing. You can use `kubectl logs` to see the output from any pod.

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
…
coredns-7944c66d8d-rmxnr    1/1     Running   0
64m
…
```

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. Also, Kubernetes DNS works with namespaces. Simple host names look for services in the current namespace, but we can use the full name for services in other namespaces. For example, we could find our PostgreSQL database from outside the default namespace by using `postgresql.default.svc`.

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 cluster role, 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 role binding to apply this cluster role 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
```

Continued on next page

```
  name: developers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-read-only
  apiGroup: rbac.authorization.k8s.io
```

Alternatively, we can use a cluster role binding 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.

CONCLUSION

Kubernetes is the most popular container orchestration framework. It is a powerful and reliable way to run containerized applications in production, providing reliability and scalability.

This Refcard has shown a few of the most important resource types in Kubernetes to help you get started deploying applications, and with what you've learned here, you can start exploring all that Kubernetes has to offer.

WRITTEN BY ALAN HOHN,
LOCKHEED MARTIN FELLOW

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

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300